eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

DETECCIÓN DE CARRETERAS EN IMÁGENES DE RECONOCIMIENTO REMOTO MEDIANTE DEEP LEARNING

Autor

Ander Berrondo Urruzola

2020

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

DETECCIÓN DE CARRETERAS EN IMÁGENES DE RECONOCIMIENTO REMOTO MEDIANTE DEEP LEARNING

Autor

Ander Berrondo Urruzola

Director

Manuel Graña Romay

Resumen

Este proyecto explora las técnicas de detección de carreteras en imágenes *RGB* de reconocimiento remoto, empezando por métodos de *Machine Learning* y llegando hasta la aplicación de técnicas de aprendizaje profundo. Para poder entender bien tanto los métodos basados en el aprendizaje profundo como los basados en aprendizaje automático clásico, se hace una extensa revisión de sus bases teóricas. Una vez fijadas estas bases, el objetivo es comparar experimentalmente los diferentes métodos computacionales y la respuesta que ofrecen a este problema, empleando diferentes imágenes extraídas de un desafío abierto recientemente y probando múltiples valores de los metaparámetros de estos métodos. Si bien los resultados obtenidos no son competitivos con los mejores publicados para este desafío, el desarrollo del proyecto ha concluido con éxito la evaluación de las distintas aproximaciones dentro de las limitaciones computacionales.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

1.1. El desafío

Este proyecto está basado en un desafío lanzado por *CodaLab*, una plataforma de código abierto que, entre otras tantas cosas, ofrece a sus usuarios diferentes retos relacionados con el aprendizaje automático y la computación avanzada. El desafío se llama *DeepGlobe Road Extraction Challenge*, y como su nombre indica, trata de detectar carreteras en imágenes de reconocimiento remoto, utilizando técnicas de aprendizaje profundo. Asimismo, este desafío solo es una tercera parte de un reto más grande donde se proponen tareas de segmentación, detección y clasificación de imágenes satelitales (explicadas en el artículo [Demir et al., 2018b]).

Detalles del desafío

La detección de carreteras en imágenes se formula como un problema de clasificación binaria a nivel de píxel, donde la entrada es una imagen proporcionada por los creadores del desafío y el objetivo es obtener una imagen binaria del mismo tamaño, en la cual los píxeles que corresponden a una carretera en la imagen de entrada tendrán valor **1** y los que no, **0**.

El conjunto de datos sobre el que se realiza el desafío está dividido en tres partes diferentes: una para el entrenamiento, una para la validación, y, por último, una para el test. Se dispone de 6226 imágenes de entrenamiento en formato *RGB*, y cada una de estas

imágenes está acompañada por una máscara que contiene la verdad del terreno codificada en valores de gris. No hay una explicación para los valores de la máscara. Asimismo, el desafío proporciona 1243 imágenes para la validación y 1101 para el test.

Los ficheros que contienen las imágenes *RGB* están nombrados como *<id>_sat.jpg* y los de las máscaras como *<id>_mask.png*, donde el *id* es un número aleatorio (obviamente, las máscaras tienen el mismo *id* que su imagen original). Además, se ha de tener en cuenta que los píxeles de las máscaras no son exactamente **0** (negro) y **255** (blanco); por eso, los píxeles de valor menor que **128** serán clasificados como negro, y los que tengan un valor mayor como blanco. Asimismo, la métrica de evaluación empleada en el desafió será la *Intersection over Union* (***IoU***), la cual se calcula mediante la siguiente ecuación:

$$IoU = \frac{TP}{TP+FP+FN}, \tag{1.1}$$

donde *TP*, *FP* y *FN* denotan los positivos ciertos, falsos positivos y falsos negativos, respectivamente.

Finalmente, cabe destacar que los creadores del desafío ofrecen varias fuentes de información, tal como los artículos [Demir et al., 2018a] y [Máttyus et al., 2017], que sirven para tener un primer contacto con el campo de detección de carreteras.

Motivación del desafío

La extracción de carreteras a partir de imágenes de satélite ha sido un tema de investigación ampliamente desarrollado en la última década. Su utilidad se extiende a campos como la actualización de mapas de carretera, planificación de ciudades, actualización de información geográfica, navegación de automóviles... Finalmente, según cuentan los creadores del desafío, éste ha sido creado para ayudar a las **zonas de desastre** (lugares donde hayan ocurrido terremotos, explosiones...), especialmente en países subdesarrollados, a dar una respuesta a la crisis facilitando la accesibilidad a los mapas e información geográfica.

Dificultades principales del desafío

Al enfrentarse a un problema de extracción de carreteras, se ha de tener en cuenta que no es una tarea *fácil*, ya que obtener resultados aceptables puede exigir una gran cantidad de tiempo. Además, al tratarse de un proyecto en el que se ha de trabajar con técnicas de

aprendizaje profundo (redes neuronales), es necesario disponer de unos recursos computacionales que un estudiante muy difícilmente puede tener a su disposición.

Dejando a un lado las limitaciones computacionales, el ejercicio de clasificación a resolver en el desafío se basa en un conjunto de datos inherentemente **mal balanceado**, ya que la cantidad de píxeles que no son carretera son **varios ordenes de magnitud más abundantes** que los que sí lo son. Esto afecta directamente a la clase minoritaria, ya que se produce una generalización de la clase mayoritaria; es decir, si en el entrenamiento del modelo el 95 % de los píxeles son *no carretera*, el modelo siempre tendrá un fuerte sesgo hacia predecir que los nuevos casos son del tipo mayoritario.

1.2. Objetivos del proyecto

El objetivo principal de este proyecto ha sido analizar, entender, crear/probar y comparar diferentes métodos computacionales para resolver el problema de detección de carreteras en imágenes de reconocimiento. De esta manera, comparando las estructuras, comportamientos y resultados de los mismos, se puede evaluar la diferencia entre los modelos. Asimismo, para llevar a cabo este objetivo principal, también se han tenido que resolver otros objetivos menores que han sido parte del proceso de aprendizaje del alumno.

- Analizar la literatura: Debido a que los modelos utilizados son complejos, para poder entender y analizarlos ha sido necesario repasar las arquitecturas previamente creadas. Además, al ser totalmente novato en el campo de la inteligencia artificial y del aprendizaje profundo, llevar a cabo esta tarea ha sido totalmente imprescindible. Por ello, se ha invertido gran parte del tiempo en leer artículos y fuentes de información.
- Identificar el conjunto de datos: Para el problema de detección de carreteras es necesario disponer de un conjunto de datos muy amplio, además de ser válido para esta tarea. Por eso, es necesario elegir un conjunto de datos útil, para que el entrenamiento y los resultados de los modelos sean significativos; en este caso, ya que los datos son proveídos por el desafío, cumplen estos requisitos. Finalmente, cabe destacar que el conjunto de datos es público (solo hay que pedir permiso a los organizadores del desafío).
- Preprocesar el conjunto de datos: Una parte muy importante de este problema es el preprocesado; es decir, hay que modificar el conjunto de datos para que sirva

de entrada para los modelos. En vez de trabajar con las imágenes completas, se ha trabajado con ventanas; esto se debe a que hay que analizar las imágenes píxel por píxel, independientemente unas de otras. Por ello, para tratar cada píxel se crea una ventana (las dimensiones de la misma pueden variar), y los modelos se entrenan sobre ellas.

- Definir los modelos: Para llevar a cabo el proyecto se han elegido varios modelos de clasificadores. De esta manera, previamente se ha analizado la literatura, para después acabar definiendo varios sistemas que sean comparables entre ellos. Los modelos utilizados en el proyecto se han obtenido de diferente manera: algunos han sido directamente extraídos de alguna biblioteca sin sufrir ningún cambio, otros se han basado en patrones ya creados pero se han modificado para ajustarlos a las necesidades del problema, y finalmente, otros han sido creados desde cero.

- Análisis de los parámetros: A la hora de entrenar los modelos se han analizado los parámetros de los mismos; por ello, han sido ajustados con el fin de mejorar los resultados, a la par que han servido para entender que influencia tiene cada uno de ellos sobre el modelo.

2. CAPÍTULO

Bases teóricas

La detección de carreteras en imágenes de reconocimiento remoto se plantea como un problema de clasificación que se ataca mediante técnicas de aprendizaje automático. Para entrenar un modelo de clasificador es necesario un conjunto de datos que sea desconocido para el modelo y que no se tenga previo conocimiento acerca de su distribución. De esta manera, es necesario ir ajustando los parámetros hasta el punto de entender cuales son los patrones o la distribución que sigue el conjunto de datos.

Tradicionalmente se reconocen los siguientes tipos de aprendizaje:

- **Aprendizaje supervisado**: Se parte del conocimiento *a priori* de las respuestas deseadas a los patrones de entrada; teniendo como entrada pares con forma de (X, Y), el objetivo es obtener la relación entre los dos.
- **Aprendizaje no supervisado**: Esta vez, no se sabe cual es la salida para cada patrón de entrada, es decir, solo se dispone de los datos de entrada X. El objetivo es encontrar propiedades, agrupamientos o relaciones sobre los datos de entrada.
- **Aprendizaje semisupervisado**: Se encuentra a medio camino entre el aprendizaje supervisado y el no supervisado. Se dispone tanto de datos etiquetados como de datos no etiquetados; es decir, además de tener tuplas (X, Y), se utilizan datos X de los que no se sabe su respuesta Y, lo que implica un paso intermedio de generación de las etiquetas desconocidas en el conjunto de entrenamiento.
- **Aprendizaje por refuerzo**: En este caso, se trata de un proceso mínimamente supervisado donde el sistema de entrenamiento sólo recibe re-alimentaciones o re-

fuerzos (por ejemplo, gana o pierde), nunca la respuesta deseada o la etiqueta del patrón de entrada.

En este problema se trabaja con el primer caso, en el que se dispone de la verdad del terreno para cada imagen de entrada.

2.1. Máquinas de Soporte Vectorial (*Support Vector Machines)*

Información principalmente extraída de las siguientes páginas: [1], [2], [3]

El objetivo de una Máquina de Soporte Vectorial o *Support Vector Machine* (*SVM* de aquí en adelante) es encontrar un hiper-plano en un espacio *N-dimensional* (donde *N* es el número de atributos) que separe correctamente las dos clases que conforman el conjunto de datos.

Para entender bien de donde viene el concepto del *SVM*, primero es necesario saber de que tratan el *Maximal Margin Classifier* y el *Support Vector Classifier*.

2.1.1. Maximal Margin Classifier

Ante todo, es imprescindible definir que es un **hiper-plano**. Un hiper-plano es un subespacio plano y afín de dimensión *p-1*; es decir, si el espacio es bidimensional, el hiper-plano es un una recta, si el espacio es tridimensional, es un plano convencional (asimismo, a partir de esta dimensión, es difícil imaginarse que forma tiene el hiper-plano). Teniendo esto en cuenta, esta es la fórmula que sigue el hiper-plano (en un espacio de p dimensiones):

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p = 0 \tag{2.1}$$

En el caso de que todos los datos sean perfectamente separables linealmente (figura 2.1), hay muchos hiper-planos que pueden separar las dos clases correctamente. El objetivo es encontrar el hiper-plano que tenga el mayor **margen**; es decir, que tenga la mayor

distancia respecto a los puntos de datos de las dos clases. Este hiper-plano se conoce como *Maximal Margin Classifier* o Hiper-plano Óptimo de Separación, y maximizando el margen de esta manera se obtiene mayor probabilidad para que los siguientes puntos por clasificar sean correctamente clasificados.

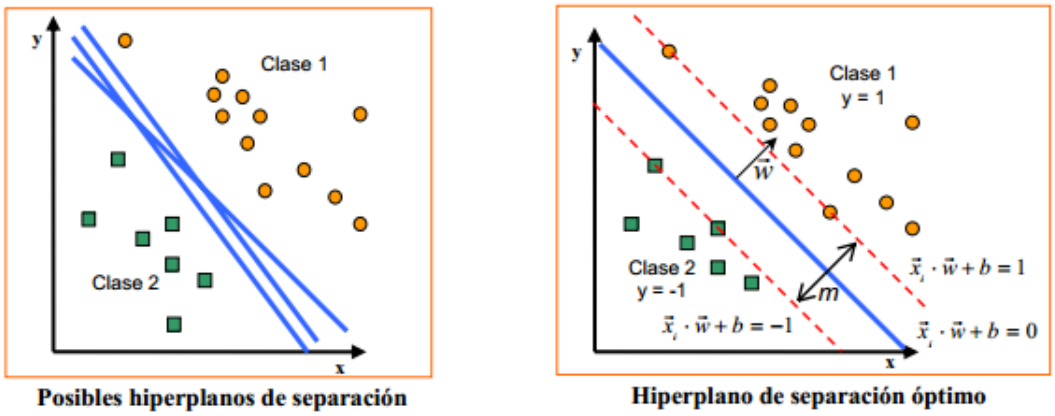


Figura 2.1: Ejemplo del *Maximal Margin Classifier*.

Fuente: http://3.bp.blogspot.com/-Zt5ab9bdQ64/UxsBSEH-3_I/AAAAAAAADRE/caQodexaP2c/s1600/36.png

2.1.2. Support Vector Classifier

El *Maximal Margin Classifier* no tiene demasiada aplicación practica, ya que rara vez se encuentra un conjunto de datos que sea perfectamente separable linealmente. Además, en el caso de conseguir un hiper-plano que cumpla los requisitos, sería muy sensible a las variaciones en los datos y es muy posible que se dé el fenómeno del *overfitting* debido a que se ajusta demasiado a los datos de entrenamiento.

Por ello, es posible utilizar un hiper-plano que, aunque no separe perfectamente las dos clases, sea más robusto y más flexible (menos *overfitting*) para los casos nuevos. Se trata del *Support Vector Classifier*, que está regulado por un parámetro C y permite que ciertas observaciones estén en el lado incorrecto del margen o incluso del hiper-plano.

El mencionado hiperparámetro C (figura 2.2) controla el número y severidad de las violaciones del margen (y del hiper-plano) que se toleran en el proceso de ajuste, y puede tener un valor cualquiera en el intervalo $[0, \infty]$. Cuando su valor es ∞, no se permite ninguna violación del margen y por lo tanto, el resultado es equivalente al *Maximal Margin Classifier*; a medida de que su valor baja, más violaciones son permitidas, hasta el punto de

que cuando $C = 0$, los errores son totalmente ignorados. Además, cabe destacar que las muestras que se encuentran justo en el margen definido por el hiper-plano son conocidas como **vectores de soporte**.

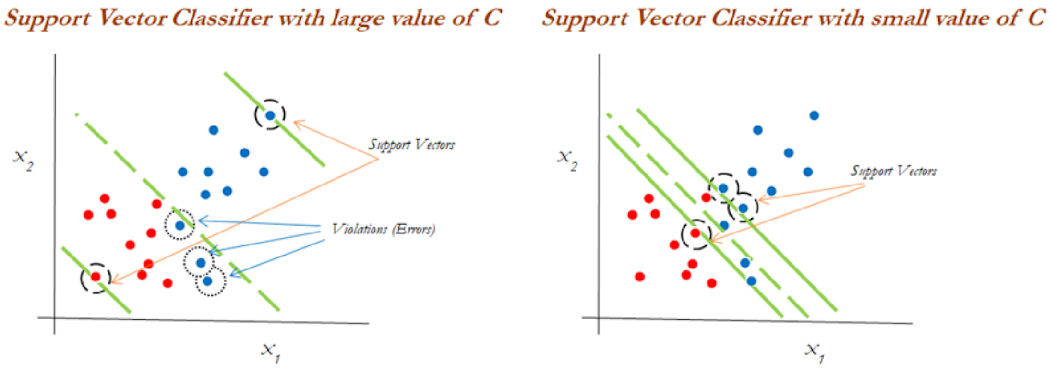


Figura 2.2: Ejemplo del *Support Vector Classifier* dependiendo del hiperparámetro C.

Fuente: https://n9.cl/8cek

2.1.3. Support Vector Machine

El *Support Vector Classifier* consigue buenos resultados en el caso de que la separación entre clases sea aproximadamente lineal, pero su efectividad cae drásticamente si no es así. Para tratar con los problemas en los que la separación de clases no es lineal, una solución muy concurrida es expandir las dimensiones del espacio original. Este tipo de modelos son conocidos como *Support Vector Machines* ([Vapnik and Lerner, 1963]), y son variaciones de los *Support Vector Classifier* en los que se aumenta la dimensión de los datos.

Kernel

Uno de los conceptos más importantes de los *SVM* es el ***Kernel***. Un *Kernel (K)* es una función que devuelve el resultado del producto escalar entre dos vectores realizado en un nuevo espacio distinto al espacio original en el que se encuentran; usualmente proyecta los datos en un espacio de más dimensiones que el espacio original buscando *desenvolver* los datos para que sean linealmente separables en ese nuevo espacio y que se pueda resolver correctamente el problema de clasificación. A pesar de que existen multitud de *Kernels* (figura 2.3), estas son las más conocidas:

- ***Kernel* Linear**:

$$K(x,x') = x \cdot x' \tag{2.2}$$

  En este caso, el clasificador *SVM* obtenido es equivalente al *Support Vector Classifier*.

- ***Kernel* Polinómico**:

$$K(x,x') = (x \cdot x' + c)^d \tag{2.3}$$

  En el caso de $d = 1$ y $c = 0$, el resultado es igual al del *kernel* lineal. A medida de que el valor de d va aumentando, la no linealidad crece también; además, es recomendable no utilizar $d > 5$ ya que el *overfitting* se convierte en un problema.

- ***Kernel Gaussiano* (radial, *RBF*)**:

$$K(x,x') = \exp^{(-\gamma||x-x'||^2)} \tag{2.4}$$

  El valor de γ controla la flexibilidad del modelo: cuando su valor es muy pequeño, el comportamiento del modelo es muy parecido a uno lineal; en cambio, a medida de que crece su valor, el modelo se vuelve más flexible.

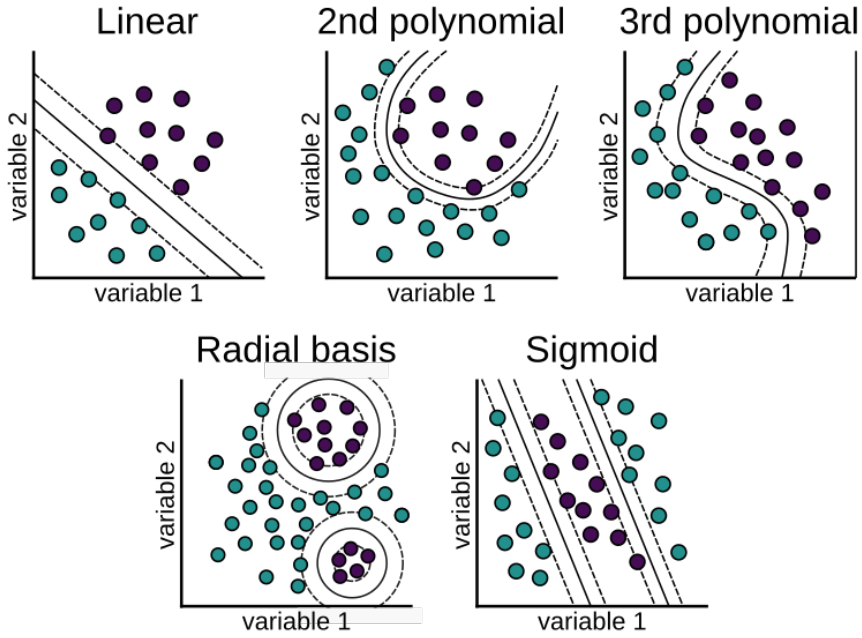


Figura 2.3: Fronteras discriminantes definidas por diferentes *kernels*.

Fuente: https://machinelearningwithmlr.files.wordpress.com/2019/10/ch06_fig_6_mlr.png?w=750

2.2. Random Forest

Información principalmente extraída de las siguientes páginas: [1], [2], [3]

El *Random Forest* (*RF* de aquí en adelante) es uno de los algoritmos más conocidos de *Machine Learning* utilizado en el ámbito del aprendizaje supervisado (es decir, cuando se conocen las respuestas deseadas para los datos usados para el aprendizaje), y fue publicado por primera vez en el artículo [Breiman, 2001]. El *RF* puede ser utilizado en problemas de clasificación (respuestas categóricas que definen la clase) y regresión (respuestas continuas).

2.2.1. Árboles de decisión

El *RF* está basado en los árboles de decisión, los cuales son una técnica para crear modelos predictivos. Son conocidos como árboles de decisión por que la predicción sigue varias ramas condicionales que tienen la siguiente forma:

si condición $\rightarrow$ **entonces** instrucción

Tal y como se puede observar en la figura 2.4, se ha de seguir el flujo del árbol respondiendo en cada nodo según las características de la muestra hasta alcanzar un nodo terminal (marcados en gris). Estos nodos son los que definirán cuál es la predicción para la entrada, precisando el valor o clase de la muestra.

En cada rama, se calcula localmente cual es la mejor división posible teniendo en cuenta las variables restantes. Las métricas más populares para esta tarea son la *impureza de Gini* y la *ganancia de información* en el caso de la clasificación y la *reducción de tareas y varianza* en el caso de la regresión.

Los árboles de decisión individuales son muy fáciles de visualizar y entender debido a que la toma de decisiones es muy parecida a la de los humanos: siguiendo una cadena de simples reglas. Aun así, no son muy robustos, ya que tienen problemas de generalización cuando se trata de muestras nuevas, y es aquí donde tiene lugar el *RF*.

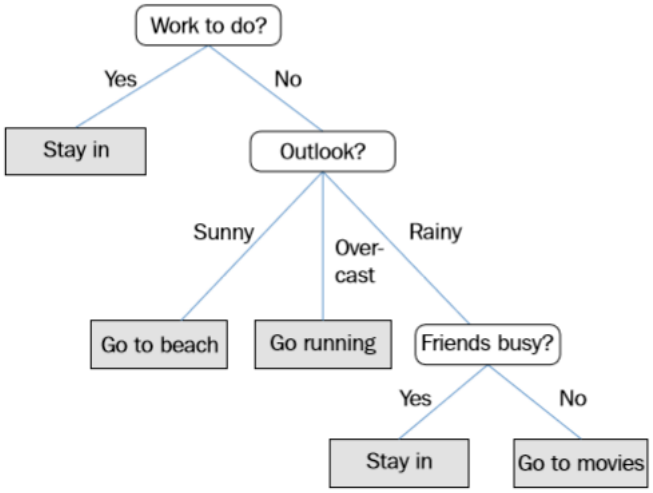


Figura 2.4: Ejemplo de un árbol de decisión.

Fuente: https://static.commonlounge.com/fp/original/3gaQtwvzuLaAtx1HpaCG5Qgnl1520487539_kc

2.2.2. Aprendizaje de modelos de comités

Los modelos de comités combinan los resultados de varios modelos simples para producir un resultado conjunto. El *RF* hace predicciones combinando los resultados de varios árboles de decisión individuales, y esa es la razón por la que se conoce como *bosque (forest)*.

Existen dos maneras principales de elegir las muestras para el entrenamiento de los múltiples árboles de decisión:

- ***Bagging***, también conocido como *agregación de Bootstrap* (utilizado en el *RF*).
- ***Boosting*** (utilizado en el *Gradient Boosting Machine*).

El *Bagging* ([Breiman, 1996]), método que se utiliza por defecto en el *RF* (imagen 2.5), funciona entrenando los árboles de decisión en subconjuntos aleatoriamente elegidos del *dataset*, utilizando el reemplazo para crear nuevos subconjuntos. De esta manera, se reduce muchísimo la varianza del modelo, ya que un gran problema de los árboles de decisión simples es que son muy sensibles al ruido de los datos; además, si los árboles utilizados en el modelo no están correlacionados entre sí, se conseguirá un modelo más robusto decreciendo el *bias* del modelo. Cabe destacar que es realmente importante que no exista

correlación entre los árboles para que el modelo sea efectivo; por ello, además de utilizar un subconjunto en vez de todos los datos, el *RF* también lleva a cabo el *Feature Bagging*, donde en cada división del árbol solo unas cuantas variables son consideradas. De esta manera, se evita que el impacto de las variables más fuertes sobre el modelo sea demasiado grande, y se consigue que las variables de menor fuerza puedan aparecer más en el modelo.

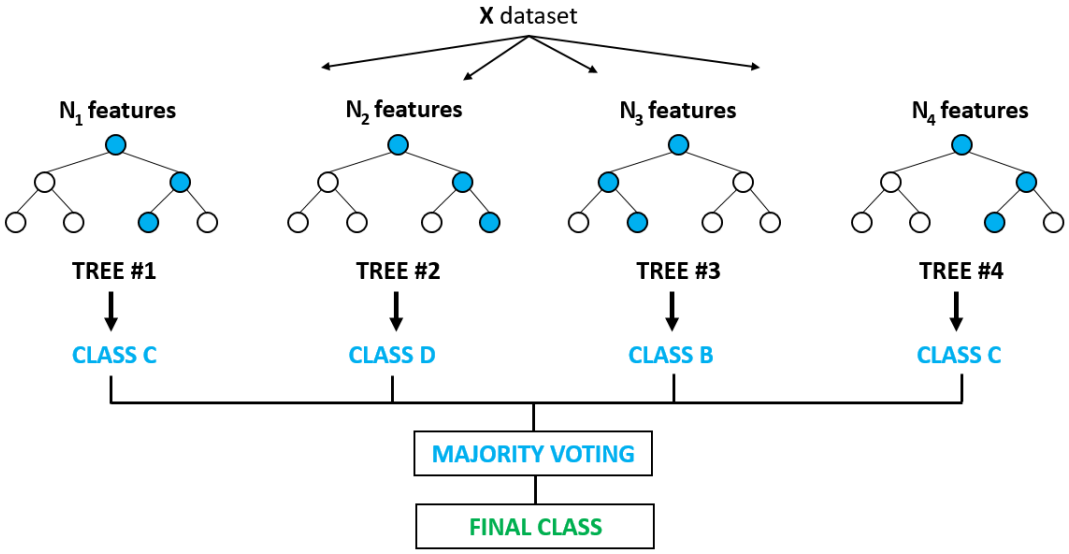


Figura 2.5: Representación gráfica del *Random Forest*.

Fuente: https://miro.medium.com/max/875/0*8nP6_SmEWu8yz5I1.png

El funcionamiento del *Boosting* ([Schapire, 1990]) es parecido al *Bagging*, pero con la diferencia de que las muestras o subconjuntos de datos tienen asignado un peso; cada vez que uno de ellos se clasifica mal, su peso crece, y los que más peso tienen son los que más presencia tendrán en los árboles. De esta manera, se hace más énfasis sobre los casos difíciles que en los fáciles, obteniendo un modelo más robusto. A diferencia del *Bagging*, donde el entrenamiento puede ejecutarse en paralelo, el *Boosting* tiene que efectuarse necesariamente de una manera secuencial.

Finalmente, la respuesta final del modelo se calcula obteniendo el promedio de todas las predicciones en problemas de regresión o utilizando la técnica del voto mayoritario en problemas de clasificación.

2.3. Redes neuronales

Información principalmente extraída de las siguientes páginas: [1], [2], [3], [4], [5], [6], [7], [8], [9]

Una red neuronal es un modelo computacional que trata de imitar el funcionamiento de las redes neuronales de los organismos vivos. Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales. De este modo, la información de entrada atraviesa la red neuronal, y después de transformar esa información sometiéndola a varias operaciones, se producen unos valores de salida (figura 2.6).

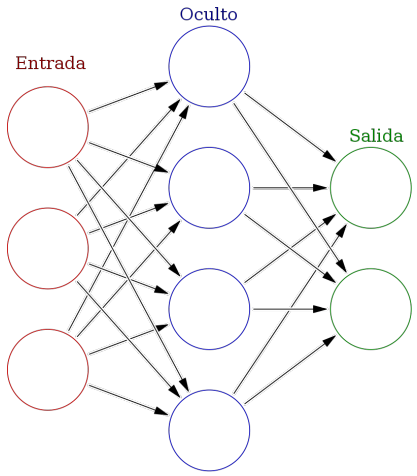


Figura 2.6: Diferentes capas de una red neuronal: capa de entrada, capa oculta y capa de salida. Son capas totalmente conectadas, ya que cada neurona esta conectada a todas y cada una de las neuronas de la capa anterior.

Fuente: https://upload.wikimedia.org/wikipedia/commons/1/11/Colored_neural_network_es.svg

Cada neurona está conectada con otras a través de unos enlaces que contienen un peso. Como se puede observar en la figura 2.7, la neurona tiene como entrada dos valores, $n1$ y $n2$, a las cuales se conecta mediante dos enlaces diferentes; además, cada enlace tiene atribuido un peso ($w1$ y $w2$, respectivamente). De esta manera, multiplicando cada valor de entrada con su respectivo peso se obtiene la entrada de la neurona en cuestión; asimismo, mediante estos pesos en los enlaces se incrementa o disminuye la importancia que

una neurona o entrada de la capa anterior puede tener sobre una neurona de la siguiente. Finalmente, además de los previamente mencionados valores de entrada, hay un valor añadido que sirve para ajustar la salida de la neurona: el valor de sesgo o *bias*.

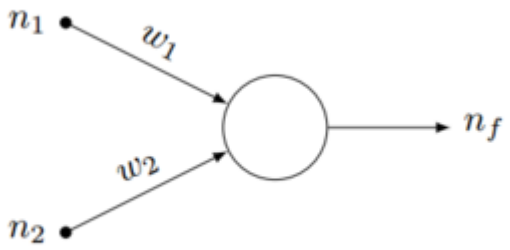


Figura 2.7: Representación gráfica de una neurona, la cual es alimentada por dos entradas diferentes ($n1$ y $n2$) y sus respectivos pesos ($w1$ y $w2$).

Fuente: https://i.blogs.es/a0eae9/perceptron/450_1000.png

Del mismo modo, a la salida de la neurona, existe una función limitadora que devuelve un valor que será generado por la neurona dada una entrada o conjunto de entradas. Esta función se llama **función de activación**, se representa con el símbolo ϕ y se intenta que su derivada sea simple, para minimizar con ello el coste computacional:

$$out = \phi(\sum_{i=1}^{n} w_i x_i) \tag{2.5}$$

La función de activación añade la propiedad no - lineal a la red, la cual es realmente necesaria para que el modelo tenga capacidad de aprendizaje. Se pueden utilizar diferentes funciones de activación, pero las más comúnmente usadas son las siguientes (figura 2.8):

- Función ***Sigmoid***: Transforma los valores introducidos a una escala $(0,1)$, donde los valores altos tienden de manera asintótica a 1 y los valores bajos tienden de manera asintótica a 0.
- Función **tangente hiperbólica**: Transforma los valores introducidos a una escala $(-1,1)$, donde los valores altos tienden de manera asintótica a 1 y los valores bajos tienden de manera asintótica a -1.

- Función ***ReLU*** (*Rectified Lineal Unit*): Presentada por primera vez en el artículo [Nair and Hinton, 2010], transforma los valores introducidos anulando los negativos y dejando los positivos tal y como entran.

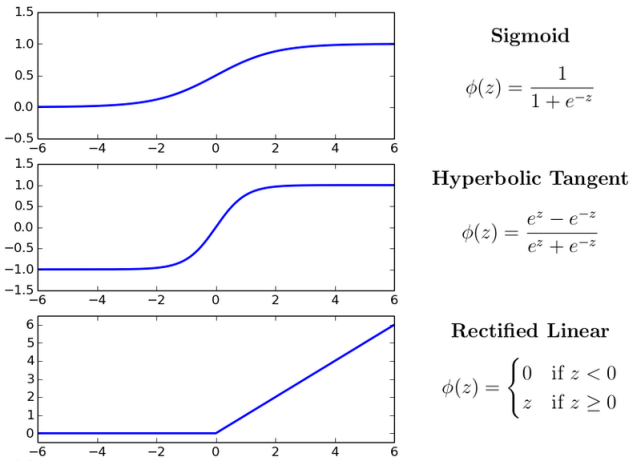


Figura 2.8: Representación gráfica de las funciones de activación más comunes, acompañadas de su respectiva ecuación matemática.

Fuente: https://www.researchgate.net/profile/Dana_Hughes3/publication/305881131/figure/fig7/AS:391681317851153@1470395511618/Common-neural-network-activation-functions.png

Básicamente, tal y como se muestra en la figura 2.9, una neurona recibe como entrada la salida de las neuronas de la capa anterior multiplicada por el peso de su respectivo enlace; después, para obtener la salida, se aplica la función de activación a la información previamente obtenida.

2.3.1. Entrenamiento de una red neuronal

Las redes neuronales, como se ha mencionado anteriormente, hacen uso de los pesos de los enlaces para calcular el valor de salida de una cierta neurona; por ello, la salida de la misma está directamente determinada por los pesos que modulan el patrón de entrada. El conjunto de todos los pesos de una red $\theta = \{w_i\}$ se define como el conjunto de parámetros, y la respuesta de la red depende del mismo. Por ello, si se quiere obtener un buen resultado para el problema, se han de ajustar los pesos hasta que la red proporcione la respuesta deseada; este proceso es conocido como *aprendizaje* o ***entrenamiento***.

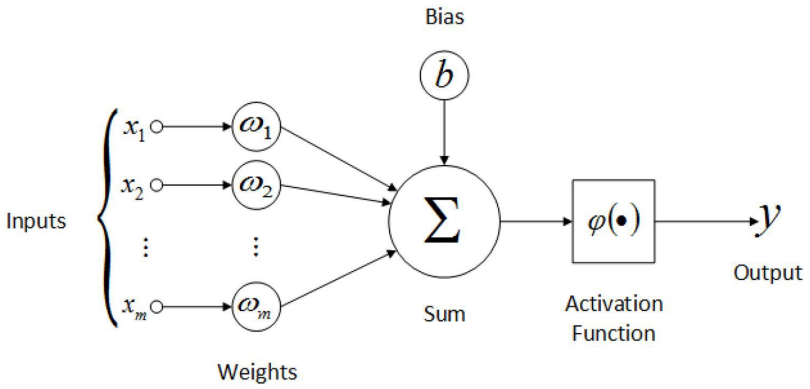


Figura 2.9: Representación gráfica de una neurona.

Fuente: https://miro.medium.com/max/3000/0*jtG_gm7tGNoFmbTy

Funciones de coste

Para guiar el proceso de optimización que produce el ajuste de los parámetros de la red y para saber si resuelven correctamente el problema planteado, es necesario tener una función que defina la calidad de los mismos: la **función de coste**. Por ello, esta función evaluará la red neuronal y calculará el error entre las predicciones y los valores reales. Así pues, es necesario escoger una función de coste que se adecue al problema en cuestión.

En este caso, la función de coste que se ha utilizado ha sido la **entropía cruzada binaria**.

La entropía cruzada binaria es una de las funciones de coste más utilizadas en cuanto a problemas de clasificación se refiere. En problemas de clasificación de dos clases, la salida de la red neuronal se puede entender como una distribución probabilística sobre cada una de ellas; de esta manera, cuanto más pequeña sea la diferencia entre la predicción y el valor real, mas pequeña será la pérdida de la entropía cruzada binaria. A continuación se da la definición formal de la entropía cruzada binaria, donde y_i es el valor real e $\hat{y}_i$ es la predicción:

$$H(y,\hat{y}) = -\sum_i y_i \cdot \log \hat{y}_i + (1-y_i) \cdot \log(1-\hat{y}_i) \tag{2.6}$$

La fase de aprendizaje consiste en la minimización de este error. Asimismo, en el entrenamiento la función de coste se obtiene calculando el promedio de todos los valores de la pérdida de la entropía cruzada binaria.

Algoritmos de optimización

En general, los procesos de optimización se utilizan para obtener los máximos y los mínimos de una función. En el ámbito de las redes neuronales, se emplean para minimizar la función de coste, de manera que el error entre las predicciones y los valorea reales sea lo más pequeño posible. Normalmente, se utiliza la técnica del **descenso del gradiente**.

Esta técnica se basa en el calculo del gradiente. El gradiente, en una función de múltiples variables, es un vector compuesto por las derivadas parciales de la función de coste respecto a estas variables, e indica cual es la dirección donde la función tiene un mayor incremento. Por lo tanto, debido al algoritmo, la función se moverá en la dirección opuesta a la indicada por el gradiente, avanzando según el paso definido por la tasa de aprendizaje. Tal y como se puede ver en la figura 2.10, el algoritmo se inicia en un punto aleatorio del dominio de la función, y poco a poco se va acercando al mínimo siendo la tasa de aprendizaje la que define la magnitud de los pasos.

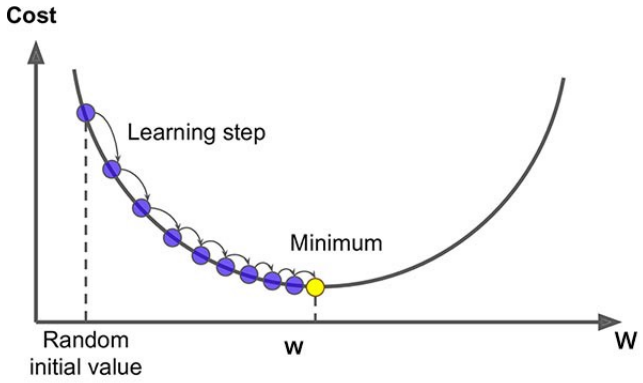


Figura 2.10: Representación gráfica del descenso del gradiente.

Fuente: https://cdn-images-1.medium.com/max/600/1*iNPHcCxIvcm7RwkRaMTx1g.jpeg

El algoritmo ***backpropagation***, presentado en el artículo [Rumelhart et al., 1986] y basado en la regla de la cadena y la programación dinámica, resuelve el problema de calcular la corrección de los pesos en las capas ocultas de las redes, retropropagando la estimación del error desde la capa de salida hacia la capa de entrada.

Muchas variaciones han sido creadas como modificaciones del descenso de gradiente; por ejemplo, la aproximación estocástica de la misma o el algoritmo *Adam*.

El **descenso del gradiente estocástico** fue presentado en el [Bottou, 2010], y trata de calcular el gradiente de cada una de las muestras del entrenamiento, para después actualizar

toda la red con ese mismo gradiente. En vez de utilizar el promedio de todos los pesos como en el descenso del gradiente clásico, se escoge un solo ejemplo, se calcula el gradiente y se utiliza ese mismo valor para actualizar los pesos de la red; después, este proceso se repite con cada uno de los ejemplos del conjunto de datos de entrenamiento.

Por último, el algoritmo ***Adam*** ([Kingma and Ba, 2015]) es el que ha sido utilizado para el proyecto, y a diferencia del descenso del gradiente estocástico, la tasa de aprendizaje no se mantiene igual durante toda la fase de entrenamiento. Este optimizador calcula diferentes tasas de aprendizaje para diversos parámetros de la red, teniendo la capacidad de ajustarlos cuando sea necesario; para llevar a cabo esta tarea, utiliza las estimaciones del primer y segundo momento del gradiente.

Generalización del conocimiento

El objetivo de entrenar el modelo es conseguir el error mas pequeño posible entre las predicciones y los valores reales. Aun así, es posible que aunque el aprendizaje sobre los datos de entrenamiento sea bueno, la función de coste incremente notablemente con un nuevo conjunto de datos que no siga la misma distribución. Para evaluar la calidad de una red neuronal se utiliza una técnica llamada ***holdout***, donde el conjunto de datos se divide en tres partes diferentes: el entrenamiento, la validación y el test.

En el **entrenamiento** se utilizan todos los datos de entrenamiento en cada época (o iteración), se comparan las predicciones con los valores reales y finalmente se actualizan los parámetros de la red mediante el calculo de la función de coste.

Después de cada época, al actualizar los pesos, se utiliza el conjunto de datos de **validación** para evaluar el modelo. Es posible que el valor de la función de coste sea pequeña para el entrenamiento, pero no para la validación; es decir, que los datos de entrenamiento se hayan aprendido correctamente pero que la red no haya sido capaz de generalizarlo sobre los datos de evaluación. Este fenómeno es conocido como ***overfitting***; asimismo, el caso opuesto es el del ***underfitting***, donde el error del entrenamiento es grande ya que el modelo no es capaz de adecuarse a la distribución del conjunto de datos del entrenamiento. La figura 2.11 ilustra estos casos.

Finalmente, después de varias épocas de entrenamiento, cuando la función de coste sobre el conjunto de validación comienza a divergir, se asume que los parámetros se han ajustado lo mejor posible y se lleva a cabo la evaluación del modelo sobre los datos del **test**.

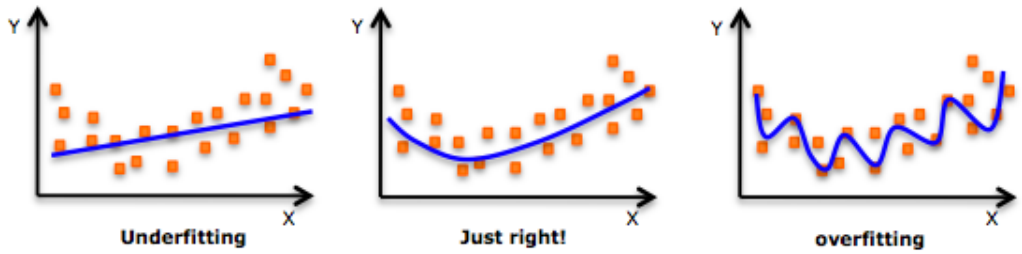


Figura 2.11: *Underfitting:* la función no es capaz de aprender la distribución correctamente; *Correcto*: La función se ha ajustado a los datos; *Overfitting:* La función se ha ajustado demasiado a los datos y no es capaz de generalizar los nuevos casos correctamente

Fuente: https://i.stack.imgur.com/0NbOY.png

2.4. Arquitecturas de redes neuronales

Dependiendo del problema que haya que resolver, se definen diferentes arquitecturas de red, para poder adecuarse lo mejor posible a las características del ejercicio. A continuación se muestran las arquitecturas más importantes, desde las más sencillas hasta las más complejas:

2.4.1. Perceptrón multicapa

Es la arquitectura más conocida y básica ([Rosenblatt, 1958]); tal y como se puede observar en la figura 2.6, las neuronas se agrupan en diferentes capas, y los enlaces entre neuronas no generan ningún ciclo. La primera capa es la capa de entrada; la última, la de salida, y todas las demás son consideradas capas ocultas. Finalmente, todas las neuronas de una capa están conectadas con todas las neuronas de la capa previa.

2.4.2. Redes recurrentes

En las redes neuronales recurrentes ([Cleeremans et al., 1989]), las neuronas están conectadas arbitrariamente, pudiendo crear ciclos de retroalimentación y consiguiendo cierta temporalidad (memoria). Este tipo de arquitectura es muy utilizada para problemas del procesamiento del lenguaje, ya que a través de las retroalimentaciones es posible conocer cual es la anterior palabra (o salida). El aspecto negativo de estas redes es que su entrenamiento puede prolongarse en exceso, necesitando grandes recursos de tiempo y memoria;

aun así, es el único modelo que puede ofrecer temporalidad (o memoria), y esta es la razón por la que se considera una estructura realmente importante. Finalmente, en la figura 2.12 se puede observar una estructura de una red neuronal recurrente.

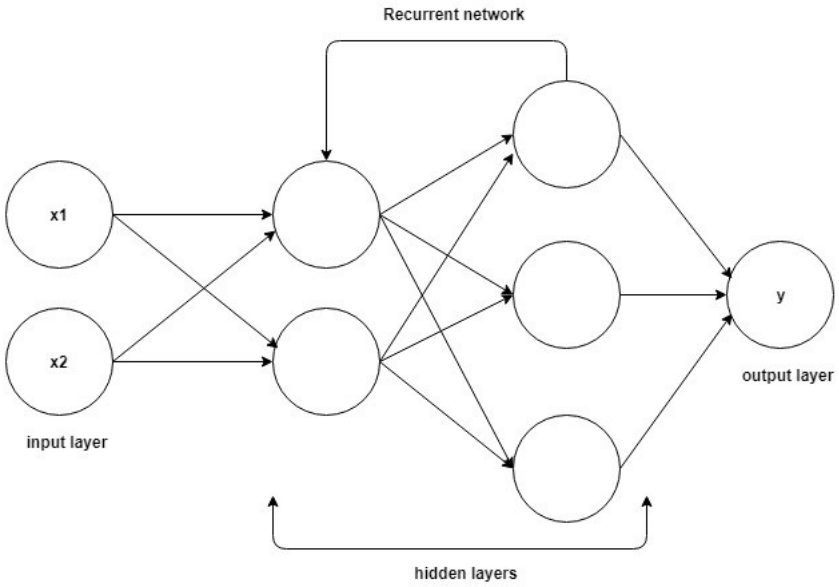


Figura 2.12: Representación gráfica de una red neuronal recurrente.

Fuente: https://miro.medium.com/proxy/1*6xj691fPWf3S-mWUCbxSJg.jpeg

2.4.3. Redes convolucionales

Este arquitectura es la más utilizada en el ámbito de la visión por computadora; por ejemplo, para llevar a cabo este proyecto de detección de carreteras, es necesario el uso de una red convolucional. La primera aparición de estas redes fue en el artículo [Fukushima, 1980], donde se presentaron las capas principales de este tipo de red (capa convolucional y *downsampling*); además, debido a la gran utilidad que tienen, han tenido un enorme crecimiento hasta el día de hoy. En cuanto a su funcionalidad, son capaces de detectar diferentes características en las imágenes, empezando desde las figuras geométricas más simples (lineas, rayas, vértices...) y llegando hasta las mas complejas (humanos, animales...).

Las redes convolucionales se basan en una combinación de capas de convolución y de *pooling*. Normalmente, al final de la red suele haber una capa totalmente conectada (*fully connected layer*), la cual se ocupa de llevar a cabo la clasificación; este último, sin em-

bargo, puede variar según el tipo de problema al que se este enfrentando. La figura 2.13 muestra una estructura típica de red convolucional.

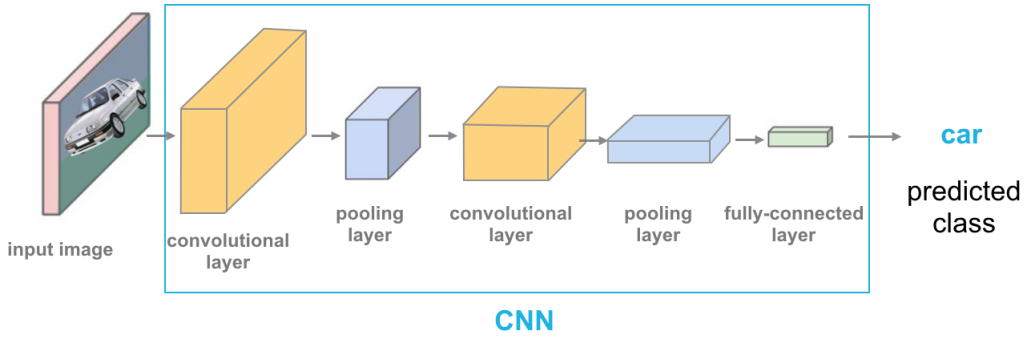


Figura 2.13: Ejemplo de una red neuronal convolucional, donde se alternan capas de convolución y de *pooling*. Al final de la red se encuentra la capa totalmente conectada que se ocupa de realizar la clasificación (coche).

Fuente: https://cezannec.github.io/assets/cnn_intro/CNN_ex.png

Capa de convolución

En una capa de convolución se aplican diferentes filtros sobre una imagen. La profundidad de la misma define cual será la cantidad de filtros que se aplicarán sobre la imagen, utilizando cada filtro para identificar un patrón diferente. Por lo tanto, después de pasar por esta capa, el tamaño de la imagen se puede mantener o disminuir, a excepción de la profundidad, que crecerá linealmente con el número de filtros aplicados.

En la figura 2.14 se puede observar que un filtro de 3x3 es aplicado sobre todos los píxeles de la imagen; de esta manera, cada combinación de 3x3 de la capa de partida se multiplicará con el filtro, y se obtendrá la capa convolucionada colocando la suma de cada combinación en su respectiva posición.

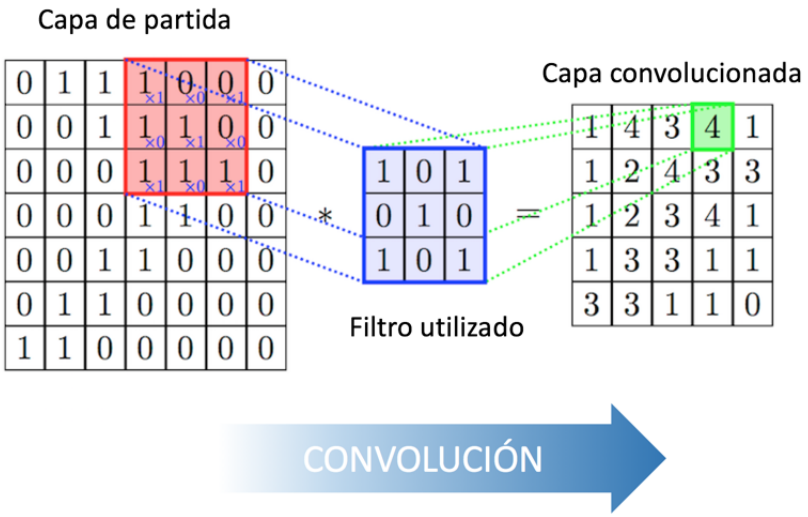


Figura 2.14: Ejemplo de una convolución.

Fuente:
https://www.diegocalvo.es/wp-content/uploads/2017/07/convoluci%C3%B3n.png

Para cada convolución se han de definir dos parámetros principales: el tamaño del filtro y su movimiento (o *stride*). Este último determina cuanto se ha de mover el filtro en cada paso, y cuanto más grande sea, más se reducirá el tamaño de la imagen.

Capa de *pooling*

El objetivo principal de la capa de *pooling* es disminuir el tamaño de la imagen con el fin de reducir la cantidad de enlaces de la red y de evitar el previamente mencionado *overfitting*.

Principalmente, se diferencian dos tipos: el *average pooling* y el *max pooling*. Los dos tienen el mismo comportamiento (una ventana de la imagen se sustituye por un único valor calculado sobre ella), pero tal y como indican sus nombres. el funcionamiento es ligeramente diferente: el *average pooling* calcula el promedio, mientras que el *max pooling* calcula el máximo.

En los ejemplos de la figura 2.15, el tamaño se reduce a la mitad, obteniendo una capa final de 2x2. Tal y como se ha mencionado, el *max pooling* elige el máximo de cada grupo; por el contrario, el *average pooling* calcula el promedio para cada uno de ellos.

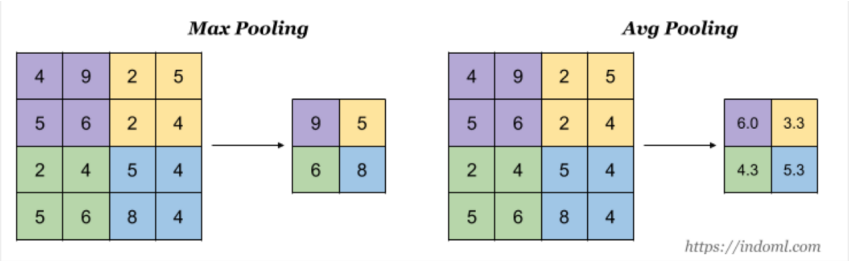


Figura 2.15: Ejemplos del *average pooling* y del *max pooling*.

Fuente: https://miro.medium.com/max/2556/1*addKTkLxYNReeWWidvzyhw.png

Capa totalmente conectada (*Fully Connected Layer*)

Normalmente, al final de una red neuronal convolucional se suele colocar una capa totalmente conectada (*fully connected layer*) que se ocupa de hacer la clasificación; en este proceso, cada píxel de la imagen se trata independientemente, y finalmente se elige a qué clase pertenece la imagen considerada en su totalidad.

2.4.4. UNet (Red de tipo U)

La red ***UNet*** es una derivación de la *red convolucional*, diseñada y publicada en el artículo [Ronneberger et al., 2015] con el objetivo de procesar imágenes biomédicas. Esta red es capaz de localizar las áreas de anomalías en los casos biomédicos; es decir, en qué lugar se ha producido la enfermedad.

La arquitectura *UNet* efectúa la clasificación píxel por píxel simultáneamente, por lo que es capaz de localizar y distinguir todos los bordes y salientes. Asimismo, esta red ha sido utilizada en otro tipo de problemas; por ejemplo, la detección de carreteras, donde cada píxel tiene que ser analizado individualmente y decidir si pertenece a la clase de *carretera* o *no carretera*. Además, ya que cada píxel es tratado de forma independiente, el tamaño de la imagen de entrada y de salida **siempre** es el mismo (una predicción por cada píxel).

Tal y como se muestra en la figura 2.16, la red tiene *forma de U*. La arquitectura es simétrica y esta dividida en dos partes: el camino de contracción, en el cual se efectúa el **proceso convolucional**; y el camino de expansión, donde se encuentran todas las capas de **convolución transpuestas**.

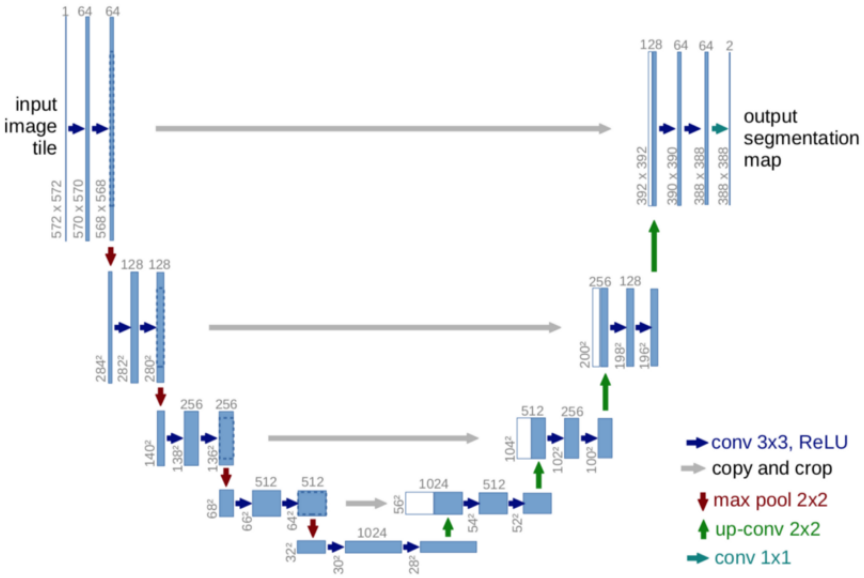


Figura 2.16: La arquitectura *UNet*.

Fuente: https://miro.medium.com/max/875/1*f7YOaE4TWubwaFF7Z1fzNw.png

Camino de contracción

En esta parte, tal y como se ha mencionado anteriormente, se llevan a cabo las **convoluciones** y reducciones de dimensión mediante ***pooling***, imitando el siguiente patrón:

Capa de convolución $\rightarrow$ Capa de convolución $\rightarrow$ *Max Pooling* $\rightarrow$ *Dropout* (opcional)

Como bien se puede observar en la figura 2.17, este proceso se repite **4** veces. Por ejemplo, en la primera etapa, la cantidad de canales (profundidad de la imagen) cambia de $1 \rightarrow 64$, ya que el proceso de convolución conlleva habitualmente el aumento de la profundidad. Además, la flecha roja con dirección hacia abajo indica el *max pooling*, el cual reduce a la mitad la dimensión de la imagen.

Para acabar, entre el camino de contracción y expansión, se utilizan otras dos capas de convolución (figura 2.18), pero esta vez sin aplicar el *max pooling*.

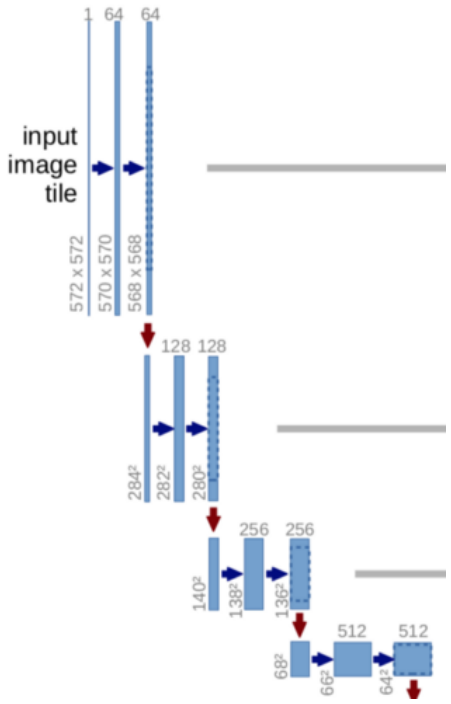


Figura 2.17: Camino de contracción de la red *UNet*

Figura 2.18: Camino central de la red *UNet*.

Camino de expansión

En esta parte, en cambio, se llevan a cabo las **convoluciones transpuestas**, imitando el siguiente patrón:

Capa de convolución transpuesta → Concatenación → Capa de convolución → Capa de convolución

La convolución transpuesta es una técnica de *upsampling* que expande la dimensión de la imagen. Básicamente, efectúa la técnica *padding* sobre la imagen original (añadir píxeles extra para que durante las convoluciones los píxeles más alejados del centro no sean ignorados) y lleva a cabo una operación de convolución.

Por ejemplo, tal y como se puede observar en la figura 2.19, las dimensiones de la imagen crecen de 28x28x1024 $\rightarrow$ 56x56x512, para después llevar a cabo la concatenación con su respectiva imagen del camino de contracción y formar entre los dos una imagen de 56x56x1024; asimismo, el objetivo de esta combinación es intentar incrementar la precisión de la imagen.

Figura 2.19: Ejemplo de una convolución transpuesta de la red *UNet*.

Para finalizar el patrón, se añaden dos capas de convoluciones más; además, al igual que en el camino de contracción, este proceso se efectúa **4** veces (imagen 2.20).

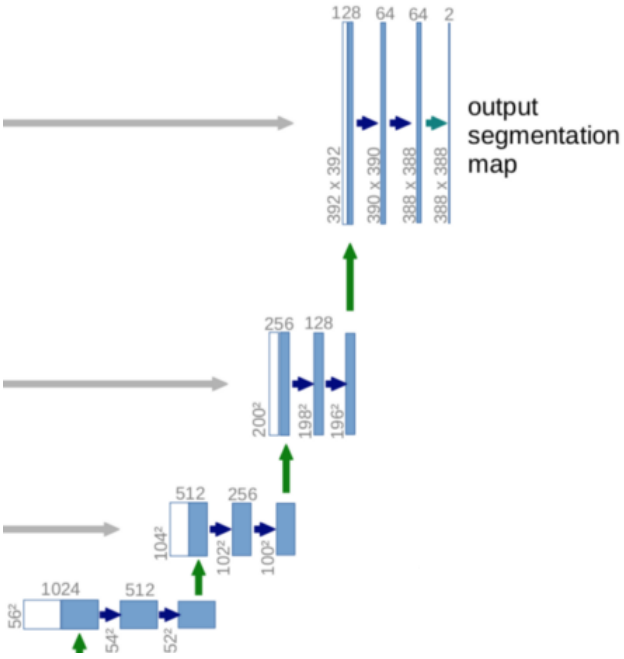


Figura 2.20: Camino de expansión de la red *UNet*

Al final de la arquitectura (imagen 2.21), se ha de ajustar la salida según el tipo de problema. En la publicación original, la última capa se trata de una capa de convolución con un filtro de tamaño 1x1, definiendo que la profundidad de la imagen en la capa de salida será de 2.

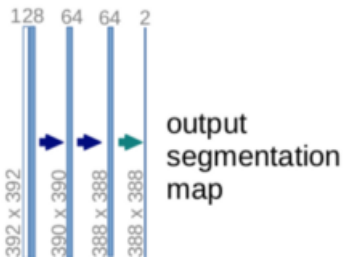


Figura 2.21: Salida de la red *UNet*.

2.5. Operaciones morfológicas

Información principalmente extraída de las siguientes páginas: [1], [2]

En cuanto al tratamiento de imágenes se refiere, las operaciones morfológicas sirven para simplificar las mismas, preservando las formas principales de los objetos. En el ámbito de visión por computadora es muy común el uso de estas técnicas para el tratamiento de regiones (explicadas por primera vez en el libro [Serra, 1983]); es decir, para determinar como pueden llegar a cambiar o transformarse ciertas regiones de la imagen después de someterlas a estas operaciones.

Las transformaciones morfológicas básicas son la erosión (extraer los *outlayers*) y la dilatación (agregar píxeles), y a partir de ellas se pueden generar diferentes operaciones como la apertura y el cierre, las cuales han sido utilizadas en el proyecto y están directamente relacionadas con la representación de las formas, la descomposición y la extracción de primitivas. Además, las operaciones morfológicas suelen aparecer en los siguientes contextos:

- Cuando es necesario intentar suavizar los bordes de una región.
- Unir ciertas regiones que han sido incorrectamente separadas en la segmentación de una imagen.
- El caso opuesto al anterior, en el que es necesario separar diversas regiones.
- En el caso de que se necesite computar las regiones de una imagen, facilitar el proceso.

2.5.1. Apertura y cierre

Estas dos operaciones (figura 2.22) se basan en la erosión y la dilatación, las cuales se obtienen de la siguiente manera:

- La erosión se trata de una convolución no lineal cuyo operador es la intersección de conjuntos.
- La dilatación, en cambio, es una convolución no lineal cuyo operador es la unión de conjuntos.

La **apertura** consiste en aplicar una erosión seguida de una dilatación:

$$A \circ B = (A \ominus B) \oplus B \tag{2.7}$$

El objetivo de la apertura es suavizar los contornos de la imagen, eliminando los pequeños salientes. Además, intenta abrir huecos pequeños, a la par que elimina franjas o zonas que sean *más estrechas* que el objeto estructural.

El **cierre**, en cambio, se trata de aplicar una dilatación seguida de una erosión:

$$A \cdot B = (A \oplus B) \ominus B \tag{2.8}$$

El objetivo del cierre es el contrario a la apertura, ya que intenta eliminar pequeños huecos y unir componentes conexos cercanos.

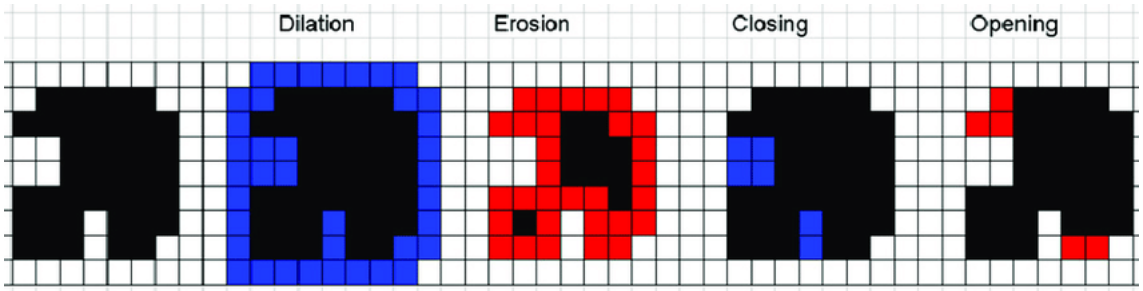


Figura 2.22: Diferentes transformaciones morfológicas.

Fuente: https://n9.cl/nxi2

3. CAPÍTULO

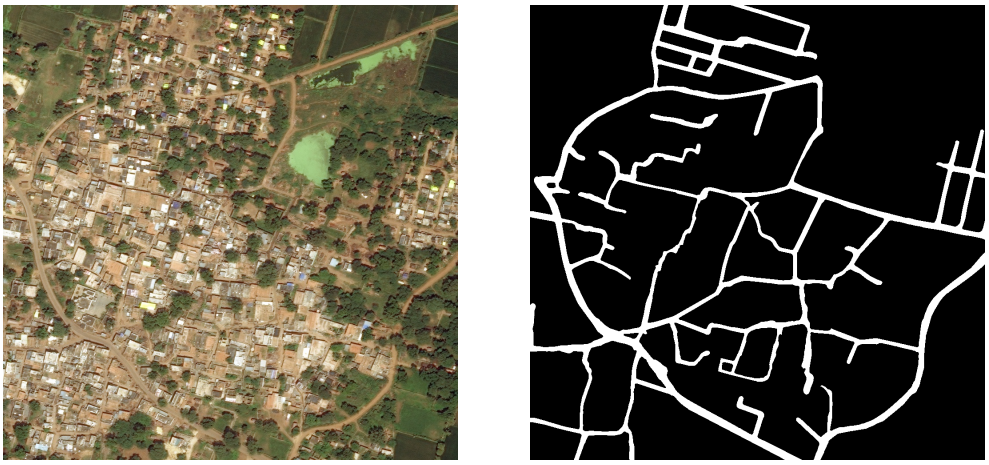
Desarrollo del proyecto

3.1. Conjunto de datos

Todos los datos utilizados en el proyecto han sido proporcionados por los creadores del reto *DeepGlobe Road Extraction Challenge*, y se puede acceder a ellos creando una cuenta y haciendo una petición de ingreso al reto. El conjunto de datos está preparado para la identificación, segmentación y clasificación de carreteras, y aunque principalmente su uso sea para la realización del reto, también puede ser utilizado para otros fines (por ejemplo, este proyecto).

El conjunto de datos está dividido en tres grupos diferentes: uno para el entrenamiento, otro para la validación y, por último, otro para el test. Tal y como se ha explicado en la introducción (1.1), solo el subconjunto para el entrenamiento dispone de máscaras binarias; debido a esto, ha sido el único subconjunto que se ha podido utilizar para realizar el proyecto, ya que en las tres fases (entrenamiento, validación y test) han sido necesarias las máscaras para poder medir la calidad del modelo.

Teniendo esto en cuenta, se dispone de 6226 imágenes de 1024x1024 para realizar todo el proyecto (figura 3.1a), donde cada imagen está acompañada por su respectiva máscara con la verdad del terreno (figura 3.1b). Las imágenes *RGB* están etiquetadas como *<id>_sat.jpg* y las máscaras como *<id>_mask.png*; además, las máscaras están formadas por píxeles blancos (*carretera*) y negros (*no carretera*), y las dimensiones serán las mismas que las de la imagen original.



(a) Imagen *RGB*. **(b)** Máscara.

Figura 3.1: Ejemplo de una imagen del *dataset* y su respectiva máscara.

3.2. Planteamiento inicial del proyecto

En este proyecto de detección de carreteras se hace frente a un problema en el que cada píxel de la imagen ha de ser tratada independientemente para decidir si ha de ser clasificada como carretera o no. Por esta razón, el planteamiento inicial ha sido el de trabajar con **ventanas** en vez de con la imagen en su totalidad; de esta manera, la clasificación se produce píxel por píxel, y el resultado final se puede obtener uniendo todas las predicciones de los píxeles en una sola imagen.

Las ventanas se extraen alrededor de un píxel; es decir, el primer paso es elegir el píxel, y después se extraen los píxeles que le rodean creando una ventana de la dimensión deseada. Se trabaja con la idea de la ventana ya que para que los modelos puedan aprender ciertos patrones es necesario que la entrada tenga una dimensión considerable, notablemente más grande que la de un único píxel. Además, el trabajar con ventanas ofrece la oportunidad de entender cómo influye el tamaño de la entrada en los resultados finales, siendo una faceta importante que comparar en las evaluaciones.

Asimismo, el hecho de trabajar con ventanas también se convierte en una ventaja para afrontar el problema del mal balanceo de los datos. Tal y como se explica en la sección 3.3, se selecciona el mismo número de ventanas para cada clase (*carretera* y *no carretera*), de manera que se obtiene balancear el entrenamiento; además, si no se hubiera trabajado con ventanas, los datos tendrían que haber sido preprocesados de una manera

diferente (modificando el conjunto de datos o creando muestras artificiales), complicando notablemente el proceso de adaptación de los datos.

En cuanto al entrenamiento, la validación y el test, cabe destacar que se han realizado de distinta manera. El entrenamiento y la validación se han llevado a cabo sobre conjuntos de ventanas aleatoriamente obtenidas de distintas imágenes, totalmente independientes unas de otras; en cuanto al test se refiere, en cambio, se ha efectuado de dos distintas maneras: primero, al igual que en el entrenamiento y validación, los modelos se han puesto a prueba con ventanas obtenidas aleatoriamente, y segundo, se han segmentado imágenes enteras por ventanas, para poder hacer predicciones independientemente y finalmente obtener el resultado final juntando todas ellas.

3.3. Preprocesado

El preprocesado de los datos ha sido, sin lugar a dudas, una de las partes más importantes del proyecto y donde un gran tiempo ha sido invertido. En él se ha realizado la extracción de ventanas para el entrenamiento y la validación, de la misma manera que se ha llevado a cabo la segmentación por ventanas de las imágenes que se han utilizado para el test.

3.3.1. Parámetros a definir

Antes de empezar con el preprocesado, se han de elegir los siguientes parámetros que influirán directamente en el resultado final:

- Número de imágenes que se utilizarán para el entrenamiento.
- Número de imágenes que se utilizarán para la validación.
- Número de imágenes que se utilizarán para el test.
- Número de ventanas por cada tipo de píxel (*carretera* y *no carretera*) que se extraerán de cada imagen; es decir, si el valor es 10, se obtendrán 10 ventanas clasificadas como *carretera* y otras 10 clasificadas como *no carretera*.
- Tamaño de las ventanas.

3.3.2. Extracción de ventanas (obtención del conjunto de datos ***X***)

Elección de imágenes

En esta parte se seleccionan las imágenes que se emplearán en las fases de aprendizaje, validación y test mediante la función *get_images_to_select* (ver figura 3.2); para que las imágenes no se repitan, se guardarán sus identidades en una variable y se comprobará en cada selección que no haya sido escogida previamente.

```
def get_images_to_select(quantity, path, ext_png, images_previous):
    i = 0
    images = []
    while i < quantity:
        random_image = random.choice(os.listdir(path))
        image_extension = random_image[-3:]
        if ((image_extension == "jpg") and (random_image not in column(images,0)) and (random_image not in column(images,0))):
            image_number = random_image.split('_')[0]
            random_image_2 = image_number + ext_png
            images.append([random_image, random_image_2])
            i = i + 1
    return(images)
```

Figura 3.2: Función para obtener imágenes aleatoriamente y sin repetición.

Las imágenes se obtienen aleatoriamente utilizando la clase *random*, después se asegura de que sea una imagen *RGB* y no la máscara (comprobando su extensión), y finalmente se garantiza que la imagen no este repetida, devolviendo la cantidad especificada de imágenes acompañadas de sus respectivas máscaras.

Extracción de las ventanas

Después de seleccionar las imágenes, se deben obtener las ventanas, de las cuales la mitad contendrán *carretera* y la otra mitad, no. Además, todas las ventanas se guardaran localmente, para poder visualizarlas o utilizarlas posteriormente; asimismo, las ventanas conservarán el *ID* de la imagen que provienen, para poder facilitar la búsqueda que se llevará a cabo en el método *get_set*, el cual servirá por igual para el entrenamiento, la validación y el test.

El primer paso será crear los directorios donde se guardarán las ventanas; por cada imagen, se creará una carpeta con su mismo *ID*, y dentro se encontrarán otras dos carpetas con los nombres *Carretera* y *No_Carretera*, donde se guardarán las ventanas.

A continuación, mediante las funciones *get_white_pixels* (figura 3.3) y *get_black_pixels*, se obtendrán aleatoriamente los píxeles en torno a los cuales las ventanas serán creadas. Para ello, se buscarán en la máscara valores superiores (blanco, *carretera*) e inferiores

(negro, *no carretera*) a **128**, obteniendo aleatoriamente y sin repeticiones la cantidad de píxeles que se ha especificado.

```
def get_white_pixels(image, size):
    pixels_selected_white = []
    i = 0
    while i < size:
        x = random.randint(0, 1023)
        y = random.randint(0, 1023)
        if (image[x][y][0] >= 128) and ([x, y] not in pixels_selected_white):
            pixels_selected_white.append([x, y])
            i = i + 1
    return(pixels_selected_white)
```

Figura 3.3: Función para obtener píxeles de *carretera* aleatoriamente y sin repetición.

Posteriormente, se creará una ventana para cada uno de los píxeles obtenidos en el paso anterior. Para ello, se utilizará la función *get_array_of_pixels* (imagen 3.4) que recibirá como argumento la imagen original, la máscara y el tamaño de la ventana.

Primero se creará un *array* de la librería *Numpy* que tendrá las siguientes dimensiones: (tamaño_ventana, tamaño_ventana, 3); la profundidad de la imagen será de 3 ya que se trata de una imagen *RGB*. Después, copiando en el *array* de manera ordenada los píxeles que rodean al píxel central, se irá creando la ventana. Asimismo, en el caso de que el píxel central esté escorado y los píxeles que le rodean estén fuera de la imagen original, a estos últimos se les asignará el valor 0. Una vez que todo los píxeles estén correctamente copiados y el *array* este listo con todos sus valores, se procederá a procesar la siguiente ventana.

```
def get_array_of_pixels(image, pixel, ventana_real):
    ventana = int(ventana_real/2)
    image_array = np.ndarray(shape = (ventana_real, ventana_real, 3), dtype = np.uint8)
    x = 0
    y = ventana_real - 1
    for i in range (pixel[0] - ventana, pixel[0] + ventana):
        for j in range (pixel[1] - ventana, pixel[1] + ventana):
            if ((i not in range(0, 1024)) or (j not in range(0, 1024))):
                image_array[x][y][0] = 0
                image_array[x][y][1] = 0
                image_array[x][y][2] = 0
            else:
                image_array[x][y][0] = image[i][j][0]
                image_array[x][y][1] = image[i][j][1]
                image_array[x][y][2] = image[i][j][2]
            y = y - 1
        y = ventana_real - 1
        x = x + 1
    return(image_array)
```

Figura 3.4: Función para crear ventanas.

La función de la figura 3.4 se ocupará de crear una ventana por cada píxel y de guardar cada una de ellas en su respectiva carpeta. Dentro de cada directorio (identificados por el *ID*) se encuentran dos carpetas, *Carretera* y *No_Carretera*, y estos son los patrones que siguen las ventanas que se guardan dentro de ellas (figura 3.5):

- **Carretera**: *ID* de la imagen + *C* + número de ventana
- **No carretera**: *ID* de la imagen + *NC* + número de ventana

Figura 3.5: Ejemplos de ventanas guardadas localmente.

El siguiente paso es juntar todas las ventanas en una misma estructura para que sirva de entrada para los modelos computacionales; aun así, debido a que la entrada para todos ellos no es la misma, se utilizarán dos estructuras diferentes:

- El *SVM* y *RF* necesitan recibir la información en un *array* unidimensional; es decir, que toda la información esté representada en una misma dimensión. Para esta tarea se utilizara el método *flatten*, ofrecido por la librería *Numpy*.
- Las redes neuronales, en cambio, admiten la información de la ventana tal y como fue creada, sin sufrir ningún cambio.

En este momento, la información se encuentra distribuida en 2 variables (en realidad 4, debido al uso de dos diferentes estructuras para las redes y los métodos clásicos de aprendizaje):

- X_no_carretera (*X_no_carretera_red* para las redes).
- X_carretera (*X_carretera_red* para las redes).

Al final de la función *get_set* se juntan las dos variables previamente mencionadas (figura 3.6), para crear el conjunto de datos (***X***) que servirá de entrada para los modelos computacionales. Primero se colocan todas las ventanas que no contengan carretera, y después

todas las que sí la contengan; de esta manera, se facilita mucho el trabajo de etiquetar (generar ***Y***) el conjunto de datos.

```
X = np.array(X_no_carretera + X_carretera)
X_red = np.array(X_no_carretera_red + X_carretera_red)
return X, X_red
```

Figura 3.6: Salida de la función de extracción de ventanas.

3.3.3. Etiquetar (***Y***) el conjunto de datos (***X***)

Finalmente, es necesario etiquetar el conjunto de datos creado en la anterior sección (imagen 3.7). Debido a que los datos están ordenados, solo se necesita crear un *array* unidimensional de la misma longitud que la cantidad de ventanas, donde la primera mitad sera etiquetada con el valor **0** (*no carretera*) y la segunda mitad con el valor **1** (*carretera*).

```
Y_carretera_train = [1] * (quantity_images_to_select_train * pixels_to_select_per_image)
Y_no_carretera_train = [0] * (quantity_images_to_select_train * pixels_to_select_per_image)
Y_train = np.array(Y_no_carretera_train + Y_carretera_train)
```

Figura 3.7: Etiquetado del conjunto de datos.

En el caso de las redes neuronales, la salida de las mismas es una probabilidad para cada una de las dos posibles clases (se explicará más detalladamente en la sección 3.4.4), por lo que las etiquetas ***Y*** deben de ser diferentes, siguiendo el patrón que se muestra a continuación:

- ***Carretera***: [0, 1]
- ***No carretera***: [1, 0]

Para crear esta estructura, se usará la función *to_categorical*, ofrecida por *Keras* en el módulo *utils*.

3.3.4. Guardar las variables localmente

El objetivo de esta tarea es guardar el conjunto de datos (***X, Y***) localmente, para poder utilizarlo siempre que se requiera; este cometido se llevara a cabo mediante el módulo *pickle*, tal y como se muestra en la figura 3.8:

```
pickle_out_train = open('entrenamiento.pickle', 'wb')
pickle.dump([X_train, X_train_red, Y_train, Y_train_red], pickle_out_train)
pickle_out_train.close()
```

Figura 3.8: Código para guardar las variables localmente.

3.3.5. Procesar imagen entera (por ventanas)

En este apartado se trabaja con una imagen entera, obteniendo una ventana por cada píxel de la misma; este proceso puede llegar a ser muy largo, ya que por cada píxel de una imagen de 1024x1024 se extrae una ventana (cuanto más grande sea su tamaño, más tardará en llevarse a cabo).

Elección de la imagen

La imagen se podrá elegir manualmente o aleatoriamente: para el primer caso, simplemente se ha de especificar cual es su ruta; para el segundo, en cambio, se ha implementado una función llamada *get_entire_image*, el cual elige una imagen aleatoriamente haciendo uso del módulo *random*.

Obtener ventanas para cada píxel de la imagen

Para esta tarea se ha definido la función *get_X_Y_for_entire_image* (figura 3.9), la cual recibe como argumento una imagen, su máscara y la dimensión de las ventanas. En ella se procesa cada píxel individualmente; primero, mediante la función *get_array_of_pixels* (explicada anteriormente en la sección 3.3.2) se obtendrá la ventana de la dimensión deseada, para después añadirla a la variable que almacenará el conjunto de datos ($\boldsymbol{X}$). Finalmente, haciendo uso de la función *check_if_there_is_road*, se analizará el valor del píxel en la máscara para comprobar que es carretera, y se añadirá a la variable que almacenará las etiquetas ($\boldsymbol{Y}$). A continuación se enumeran las variables que devuelve la función:

- $\boldsymbol{X}$: El conjunto de datos (X) aplanado, en un *array* unidimensional.
- ***X_red***: El conjunto de datos (X) en un *array* tridimensional, con la misma configuración que las ventanas.

- ***Y***: Las etiquetas (Y) en un *array* unidimensioanl.

- **Y_red**: Las etiquetas de las ventanas (Y) en dos categorías diferentes (una por cada clase), procesadas por la función *to_categorical*.

```
def get_X_Y_for_entire_image(image, image_mask, ventana_real):
    X = []
    X_red = []
    Y = []
    for i in range(0, 1024):
        for j in range(0, 1024):
            array_of_image = get_array_of_pixels(image, [i, j], ventana_real)
            X.append(array_of_image.flatten())
            X_red.append(array_of_image)
            Y.append(check_if_there_is_road(i, j, image_mask))
    X = np.array(X)
    X_red = np.array(X_red)
    Y = np.array(Y)
    Y_red = to_categorical(Y)
    return(X, X_red, Y, Y_red)
```

Figura 3.9: Función para el preprocesado de una imagen completa.

3.3.6. Guardar las variables localmente

Tal y como se hizo anteriormente en la sección 3.3.4, el conjunto de datos (***X, Y***) que representa una imagen en su totalidad también se guardará localmente mediante el módulo *pickle*.

3.4. Modelos computacionales y entrenamiento

En esta sección, además de especificar como se ha realizado el entrenamiento, se analizarán los diferentes modelos computacionales que han sido empleados en el proyecto: el *Support Vector Machine*, el *Random Forest* y las redes neuronales. Se estudiarán la implementación y las características más importantes de cada de una de ellas, y en el caso de las redes neuronales también se hará un desglose por capas con el fin de realizar un análisis más profundo.

3.4.1. *Support Vector Machine*

El *Support Vector Machine* es una técnica de aprendizaje supervisado muy concurrida para las tareas de clasificación. Una de las características más relevantes es que es muy efectivo para datos de alta dimensión, además de que es bastante robusta cuando la cantidad de variables es más alta que la cantidad de datos. Asimismo, el poder elegir el *kernel* ofrece cierta versatilidad a este método.

Para este proyecto se ha decidido usar un *kernel* **lineal**, el cual esta implementado con el nombre *LinearSVC* en la librería *scikit-learn* dentro del módulo *svm*; como ya se ha comentado en la sección de descripción teórica 2.1.3, el *SVM* con *kernel* lineal es conocido como *Support Vector Classifier*. Después de inicializar el modelo con el método *LinearSVC*, el entrenamiento se ha efectuado mediante el método *fit*, recibiendo como entrada el conjunto de datos ***X*** (aplanado, en un *array* unidimensional) y sus etiquetas ***Y*** (figura 3.10). Finalmente, a continuación se mostrarán los parámetros más relevantes de la instanciación del modelo:

- **Penalización**: *penalty* : *l2*

  Especifica la norma que se utilizará para la penalización. Normalmente, por defecto se utiliza la norma **euclideana** denotada como ***l2***, el cual se calcula de la siguiente manera:

$$||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2} \tag{3.1}$$

- **Función de coste**: *loss* : *squared_hinge*

  Especifica la función de coste. Por defecto, en este tipo de modelos se utiliza la función de coste *Hinge*, o también una variación que se consigue elevándola al cuadrado. Este último es el que se ha utilizado para el *SVM* y se formaliza de la siguiente manera:

$$L(y, \hat{y}) = \sum_{i=1}^{n} \text{máx}(0, 1 - y_i \cdot \hat{y}_i)^2) \tag{3.2}$$

- **Tipo de problema**: *dual* : *False*

  Especifica el tipo de algoritmo que se ha de utilizar para resolver el problema (*True* $\rightarrow$ dual; *False* $\rightarrow$ primal). Para este caso, al tener más variables que número de muestras, es necesario resolver el problema como primal, ya que si no el ajuste puede llegar a no converger.

- **Parámetro de regularización**: *C : 1.0*

  Especifica el valor del parámetro de regularización. Tal y como se ha explicado en la sección teórica 2.1.2, este parámetro sirve para controlar la cantidad de violaciones que el margen puede sufrir, y su valor influye directamente en el resultado final.

```
clf = svm.LinearSVC(dual = False)
clf.fit(X_train, Y_train)
```

Figura 3.10: Inicialización y entrenamiento del *SVM*.

3.4.2. *Random Forest*

Otra de las técnicas más populares para problemas de clasificación es el *Random Forest* (explicación teórica en la sección 2.2), en el cual los resultados se obtienen mediante el promedio de una larga colección de árboles no correlacionados. En cuanto a características se refiere, este modelo tiene el poder de manejar grandes cantidades de datos, además de poder estimar qué variables tienen mayor importancia sobre el modelo; además, en los conjuntos de datos incompletos es capaz de lograr unos buenos resultados.

Al igual que para el *SVM*, el entrenamiento se ha llevado a cabo utilizando la función *fit*, recibiendo como entrada el conjunto de datos ***X*** (aplanado, en un *array* unidimensional) y sus respectivas etiquetas ***Y*** (figura 3.11). Para la instanciación se ha utilizado la función *RandomForestClassifier* de la librería *scikit-learn* (módulo *ensemble*), con los siguientes parámetros:

- **Número de árboles**: *n_estimators : 100*

  Especifica el número de árboles del bosque. Cuanto más grande sea el número, más mejoran los resultados; aun así, tampoco se ha de extralimitarse con la cantidad de árboles, ya que a partir de un número el modelo deja de mejorar.

- **Criterio**: *criterion : gini*

  Especifica la función que medirá la calidad de las particiones de los datos en los nodos. El modelo ofrece dos funciones: la ganancia de información y la impureza de *Gini*; esta última es la que se utilizará, y se calcula de la siguiente manera:

$$G = \sum_{i=1}^{C} p(i) \cdot (1 - p(i)) \tag{3.3}$$

donde C es la cantidad de clases y $p(i)$ es la probabilidad de coger una muestra de la clase i.

- **Bootstrap (o Bagging)**: *bootstrap : True*

  Especifica si las muestras del modelo serán obtenidas mediante la técnica *Bootstrap* (también conocida como *Bagging*). Si el valor es *False*, todo el conjunto de datos será utilizado para crear cada árbol.

- **Aleatoriedad**: *random_state : None*

  Especifica la aleatoriedad de las muestras obtenidas con la técnica *Bootstrap*. Además, también define la aleatoriedad a la hora de elegir las muestras para decidir la mejor división de los nodos.

- **Cantidad de información**: *verbose : 0*

  Especifica la cantidad de información que se muestra en pantalla a la hora de entrenar el modelo o hacer predicciones. Cuanto más grande sea el número, más información se proyectará.

```
clf_2 = RandomForestClassifier(n_estimators = 100)
clf_2.fit(X_train, Y_train)
```

Figura 3.11: Inicialización y entrenamiento del *Random Forest*.

3.4.3. Red Neuronal: Arquitectura 1

La *arquitectura 1* se trata de una red neuronal convolucional muy básica que ha servido para entender y aplicar los diferentes conceptos y características de una red de este tipo.

Por ello, el primer paso ha sido documentarse sobre *Keras*, una librería sobre redes neuronales escrita en *Python* y que es conocida por ser la más utilizada para este tipo de tareas. En cuanto a la implementación se refiere, lo primero fue decidir cual sería el modelo que se iba a utilizar: el secuencial o el funcional. En el modelo secuencial se crean modelos capa por capa, es decir, una capa solo puede recibir información de la anterior y transmitírsela al siguiente; en cambio, el modelo funcional es más flexible, ya que una capa

puede conectarse con cualquier otra, y es por eso que finalmente se haya decantado por este modelo. Después de elegir el tipo de modelo, se ha tenido que decidir que capas se usarán; en una red convolucional, la estructura más básica se trata de unas pocas capas de convolución seguidas por capas de *MaxPooling* (figura 3.12). Además, al tratarse de un problema de clasificación, al final de la red se utiliza una capa totalmente conectada (*FCL*) que servirá para llevar a cabo esta tarea; asimismo, como la entrada de esta última capa tiene que ser unidimensional, la penúltima capa se trata de una capa aplanadora (*Flatten*). Como detalle final, también se hace uso de una capa de normalización del *Batch*, que renormaliza los pesos tras cada *batch* de entrenamiento y sirve para que la implementación sea más eficiente. A continuación se ofrecerá una explicación más detallada de la implementación:

- **Capa de convolución**: *Conv2D*:

  Capa de convolución en el cual se emplearán 64 filtros y un kernel de 3x3. La entrada será la ventana en su totalidad. Además, la activación empleada para esta capa será *ReLU*.

- **Normalización del *Batch***: *BatchNormalization*

  Capa de normalización del *Batch*. La entrada será la salida de la capa previa de convolución.

- **Capa de *MaxPooling***: *MaxPooling2D*

  Capa de *MaxPooling*, donde la dimensión de la imagen se reducirá a la mitad debido a que se utiliza un *pool* de 2x2. La entrada será la salida de la capa previa de normalización del *Batch*.

- **Capa de convolución**: *Conv2D*

  Capa de convolución en el cual se emplearán 32 filtros y un kernel de 3x3. La entrada será la salida de la capa previa de *MaxPooling*. Además, la activación empleada para esta capa será *ReLU*.

- **Capa de aplanamiento**: *Flatten*

  Capa de aplanamiento que servirá para transformar los datos a las necesidades de la capa totalmente conectada (*FCL*). La entrada será la salida de la capa previa de convolución.

- **Capa totalmente conectada**: *Dense*

  Capa totalmente conectada que realizará la clasificación. Se especifican el número de clases sobre los que se efectuará la predicción y el tipo de activación de la capa; el valor del primero será **2** (carretera y no carretera), mientras que el valor del segundo será *softmax*, el cual determina la probabilidad para cada una de las clases. La entrada será la salida de la capa previa de aplanamiento.

```
def crear_arquitectura1(input_size):
    inputs = Input(input_size)

    c1 = Conv2D(64, 3)(inputs)
    b1 = BatchNormalization()(c1)
    a1 = Activation('relu')(b1)
    p1 = MaxPooling2D((2, 2))(a1)
    c2 = Conv2D(32, 3, activation = 'relu')(p1)

    f2 = Flatten()(c2)
    outputs = Dense(2, activation = 'softmax')(f2)
```

Figura 3.12: Implementación de la *arquitectura 1*.

Información sobre las capas

- ***Conv2D***

  Este capa efectúa varias convoluciones sobre la información (imagen) que recibe como entrada. Los parámetros más importantes son los siguientes:

  - **Número de filtros**: Determina la cantidad de filtros que utilizara la capa para el aprendizaje de características; además, también especifica el número de filtros (dimensión) de salida.
  - **Dimensión del *kernel***: Determina la altura y anchura del *kernel* que efectuará la convolución.
  - ***Paso* del kernel**: Determina cual será la cantidad de unidades que se moverá el *kernel* en el proceso convolucional.

- ***Padding***: En algunos casos, en la fase de convolución las dimensiones de las imágenes pueden llegar a reducirse; si el valor de este parámetro es *valid*, las imágenes se reducirán, en cambio, si su valor es *same*, las imágenes se rellenarán manteniendo las mismas dimensiones.
- **Tipo de activación**: Determina que tipo de activación se usará después de efectuar las convoluciones.
- **Inicialización del *kernel***: Define cual es la función que determinará los valores iniciales de los pesos del *kernel*.

- ***BatchNormalization***

  La capa de normalización del *Batch* ([Ioffe and Szegedy, 2015]) es empleada para mejorar la velocidad, eficacia y estabilidad de las redes neuronales. Principalmente, su función es hacer frente al problema del *covariance shift*, el cual se produce debido a que la distribución de los datos de entrenamiento no siempre es la misma. Este fenómeno ocurre mucho en las capas ocultas, donde los valores de activación de la capa previa cambian constantemente; por ello, mediante la normalización del *Batch*, estos valores se normalizan como *z-scores* calculando el promedio y la desviación típica.

- ***MaxPooling2D***

  Esta capa (tal y cómo se explica en el apartado teórico 2.4.3) se ocupa de reducir el tamaño de las imágenes; este paso es totalmente necesario, ya que sin esta capa el número de neuronas que la siguiente capa de convolución tendría que procesar sería excesivamente grande. En este caso, los parámetros más importantes son los siguientes:

  - **Tamaño del *pool***: Determina cuanto se reducirá la imagen. Normalmente, las imágenes se suelen reducir a la mitad, utilizando un *pool* de (2,2).
  - **Paso del *pool***: Determina cuanto se moverá el *pool* en cada paso del mismo.
  - ***Padding***: Si el valor de este parámetro es *same* y la división que se efectúa en el *pooling* no da un resultado exacto, se rellenan los píxeles que se llegarían a perder; en cambio, si el valor es *valid*, no se produce ningún rellenado.

- ***Flatten***

  La única función de esta capa es aplanar la información de entrada; es decir, transformar todos los datos a una única dimensión.

- ***Dense***

  La capa *Dense* es una capa totalmente conectada (*FCL*), y su objetivo principal es realizar la clasificación. Asimismo, implementa la siguiente operación:

$$\text{salida} = \text{activación}(\text{entrada} \cdot \text{kernel} + \text{bias}) \tag{3.4}$$

  Los parámetros más importantes son los siguientes:

  - **Número de clases**: Determina cual será la dimensión de la salida; es decir, cual será el número de clases sobre los que se efectuará la predicción.
  - **Tipo de activación**: Determina cual sera la función de activación de la capa; su elección es realmente importante, ya que se ha de elegir un tipo u otro dependiendo de la tarea que haya que resolver.

3.4.4. UNet

Luego de tratar con una red neuronal convolucional más básica, se implementó una red que tuviera cierta similitud con las arquitecturas ganadoras del reto [Zhou et al., 2018] y pudiera imitar los resultados obtenidos: la red *UNet* (explicada en le sección 2.4.4). Para realizar esta tarea, ha sido muy útil y valiosa esta entrada de la plataforma *Kaggle*, donde se efectúa un análisis realmente interesante de la arquitectura.

En cuanto a la estructura de la red se refiere, se divide en dos partes principales: la parte codificadora (camino de convolución o extracción) y la parte descodificadora (camino de convolución transpuesta o expansión). Además, se puede distinguir una parte central que se coloca entre las dos secciones principales. A continuación se procederá a explicar estas partes detalladamente:

Codificador

La parte codificadora o camino de convolución es la primera sección de la arquitectura, donde se realizan las convoluciones. En este proceso, se obtienen las características más importantes de la imagen mediante los filtros, además de reducir su tamaño considerablemente. En esta parte de la arquitectura se sigue un patrón especifico que esta formado por dos capas convolucionales (cada una seguida por una capa de normalización del *Batch*,

figura 3.13), una capa de *MaxPooling* y una capa final de *Dropout*. Cabe destacar que este patrón **se repite 4 veces** (figura 3.14), y a continuación será explicado detalladamente:

- **Capa de convolución**: *Conv2D*:

  Capa de convolución en el cual se empleará siempre un *kernel* de 3x3, aunque el número de filtros irá variando: en la primera repetición del patrón se utilizaran 64 filtros, en la segunda 128, en la tercera 256. y finalmente, en la cuarta 512. Además, la función de activación que se utilizará será siempre *ReLU*, mientras que el *padding* tendrá el valor *same* (se rellenarán los píxeles restantes) y los valores del *kernel* se inicializarán siguiendo una distribución normal centrada en 0. La entrada será la ventana en su totalidad.

- **Normalización del *Batch***: *BatchNormalization*

  Capa de normalización del *Batch*. La entrada será la salida de la capa previa de convolución.

- **Repetición** de la estructura **capa de convolución + normalización del *Batch*** con la misma configuración.

- **Capa de *MaxPooling***: *MaxPooling2D*

  Capa de *MaxPooling*, donde la dimensión de la imagen se reducirá a la mitad debido a que se utiliza un *pool* de 2x2. La entrada será la salida de la capa previa de normalización del *Batch*.

- **Capa de *Dropout***: *Dropout*

  Capa de *Dropout*, donde algunas neuronas son expulsadas del modelo con el fin de evitar el *overfitting*. La cantidad de neuronas expulsadas se regula mediante el parámetro *rate*; por ejemplo, si su valor el 0.5, es 50 % de las neuronas serán expulsadas. En este caso, en la primera repetición del patrón se expulsarán el 20 % de las neuronas (ya que se trata de la capa de entrada), mientras que en las demás repeticiones se expulsarán el 40 %.

```
def bloque_conv2d(input_tensor, n_filters):
    x1 = Conv2D(n_filters, 3, activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(input_tensor)
    b1 = BatchNormalization()(x1)
    x2 = Conv2D(n_filters, 3, activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(b1)
    b2 = BatchNormalization()(x2)
    return b2
```

Figura 3.13: Bloque de dos capas de convolución (cada una seguida por una capa de normalización del *Batch*).

```
inputs = Input(input_size)
# ENCODER
c1 = bloque_conv2d(inputs, 64)
p1 = MaxPooling2D((2, 2))(c1)
d1 = Dropout(0.2)(p1)

c2 = bloque_conv2d(d1, 128)
p2 = MaxPooling2D((2, 2))(c2)
d2 = Dropout(0.4)(p2)

c3 = bloque_conv2d(d2, 256)
p3 = MaxPooling2D((2, 2))(c3)
d3 = Dropout(0.4)(p3)

c4 = bloque_conv2d(d3, 512)
p4 = MaxPooling2D((2, 2))(c4)
d4 = Dropout(0.4)(p4)
```

Figura 3.14: Parte codificadora de la red *UNet*.

Central

Esta es la parte más baja de la arquitectura, donde se efectúan las convoluciones con mayor cantidad de filtros; asimismo, es la parte donde las dimensiones de la imagen están más reducidas. Se aplican dos capas de convolución (cada una seguida por una capa de normalización del *Batch*), con la misma configuración que las capas convolucionales de la parte codificadora (figura 3.13), con la diferencia de que se utilizan **1024 filtros** (imagen 3.15).

```
# CENTRAL
c5 = bloque_conv2d(d4, 1024)
```

Figura 3.15: Parte central de la red *UNet*.

Descodificador

La parte descodificadora o camino de convolución transpuesta es la última sección de la arquitectura, donde se realizan las convoluciones transpuestas. En este proceso, después de haber sacado las características más importantes de la imagen, se recupera su estado inicial para finalmente efectuar la clasificación. En esta parte de la arquitectura se sigue un patrón especifico que esta formado por una capa de convolución traspuesta, una capa de concatenación, una capa de *Dropout* y, finalmente, dos capas convolucionales (cada una seguida por una capa de normalización del *Batch*, figura 3.13). Cabe destacar que este patrón **se repite 4 veces** (figura 3.16), y a continuación será explicado detalladamente:

- **Capa de convolución transpuesta**: *Conv2DTranspose*

  Capa de convolución transpuesta en el cual se empleará siempre un *kernel* de 2x2, aunque el número de filtros irá variando: en la primera repetición del patrón se utilizarán 512 filtros, en la segunda 256, en la tercera 128, y finalmente, en la cuarta 64; es decir, justo lo contrario a lo realizado en la parte codificadora. Además, el valor del *stride* o paso del *kernel* sera de (2,2), definiendo la altura y anchura de cada paso. Finalmente, al igual que en las capas de convolución, la función de activación será *ReLU*, el valor del *padding* será *same* (se rellenarán los píxeles restantes) y los valores del *kernel* se inicializarán siguiendo una distribución normal centrada en 0. La entrada será la salida del bloque central.

- **Capa de concatenación**: *concatenate*

  Capa de concatenación, donde diferentes entradas de la misma altura y anchura pero diferente (no necesariamente) profundidad son concatenados; por ejemplo, concatenando la ventana de (8, 8, 256) de la parte codificadora con la ventana de (8, 8, 256) de la parte descodificadora, se obtiene una ventana de (8, 8, 512). Además, la concatenación normalmente se efectúa en el último eje, y el objetivo de esta tarea es combinar la información de las capas previas para obtener más precisión. La entrada será la salida de la capa de convolución transpuesta.

- **Capa de *Dropout***: *Dropout*

  Capa de *Dropout*, donde algunas neuronas son expulsadas del modelo con el fin de evitar el *overfitting*. La cantidad de neuronas expulsadas se regula mediante el parámetro *rate*; por ejemplo, si su valor es 0.5, el 50 % de las neuronas serán expulsadas. En este caso, en todas las repeticiones del patrón se expulsan el 40 % de las neuronas; de esta manera, además de evitar el *overfitting*, se agiliza muchísimo el proceso de entrenamiento. La entrada será la salida de la capa de concatenación.

- **Capa de convolución**: *Conv2D*:

  Capa de convolución en el cual se empleará siempre un *kernel* de 3x3, aunque el número de filtros irá variando: en la primera repetición del patrón se utilizarán 512 filtros, en la segunda 256, en la tercera 128, y finalmente, en la cuarta 64; es decir, se emplearán en cada repetición la misma cantidad de filtros que en la capa de convolución transpuesta. Además, la función de activación que se utilizará será siempre *ReLU*, mientras que el *padding* tendrá el valor *same* (se rellenarán los píxeles restantes) y los valores del *kernel* se inicializarán siguiendo una distribución normal centrada en 0. La entrada será la salida de la capa de *Dropout*.

- **Normalización del *Batch***: *BatchNormalization*

  Capa de normalización del *Batch*. La entrada será la salida de la capa previa de convolución.

- **Repetición** de la estructura **capa de convolución + normalización del *Batch*** con la misma configuración.

```
# DECODER
u6 = Conv2DTranspose(512, 2, strides = (2, 2), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c5)
u6 = concatenate([u6, c4])
d6 = Dropout(0.4)(u6)
c6 = bloque_conv2d(d6, 512)

u7 = Conv2DTranspose(256, 2, strides = (2, 2), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c6)
u7 = concatenate([u7, c3])
d7 = Dropout(0.4)(u7)
c7 = bloque_conv2d(d7, 256)

u8 = Conv2DTranspose(128, 2, strides = (2, 2), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c7)
u8 = concatenate([u8, c2])
d8 = Dropout(0.4)(u8)
c8 = bloque_conv2d(d8, 128)

u9 = Conv2DTranspose(64, 2, strides = (2, 2), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c8)
u9 = concatenate([u9, c1])
d9 = Dropout(0.4)(u9)
c9 = bloque_conv2d(d9, 64)
```

Figura 3.16: Parte descodificadora de la red *UNet*.

Final (clasificación)

En esta parte final se efectuará la clasificación de las ventanas, etiquetándolas como *carretera* o *no carretera* (imagen 3.17). El patrón que se seguirá será el mismo que en la arquitectura previa (3.4.3): primero se aplanará la información mediante la capa *Flatten*, para que después utilizando la capa *Dense* se lleve a cabo la clasificación; asimismo, los parámetros de esta última capa serán **2** (la cantidad de clases sobre los que se efectuará la clasificación) y ***softmax*** (la función de activación, el cual define una probabilidad sobre cada una de las clases).

```
f10 = Flatten()(c9)
outputs = Dense(2, activation = 'softmax')(f10)
```

Figura 3.17: Parte final de la red *UNet*.

Información sobre las capas

Además de las capas anteriormente analizadas en la sección 3.4.3, a continuación se muestra información más detallada acerca de las capas utilizadas en la arquitectura *UNet*:

- ***Dropout***

  El *dropout* ([Srivastava et al., 2014]) es una técnica de regularización que tiene como objetivo reducir el *overfitting*. Para ello, en la fase de entrenamiento se descartan aleatoriamente varias neuronas de la red. Esto es, cada neurona recibirá valores de diferentes entradas en cada iteración, para que de esta manera el modelo *dude* de los valores que recibe y mantenga los pesos más repartidos y distribuidos. El objetivo es conseguir cierta capacidad de generalización, reduciendo el *overfitting*. El parámetro más importante de esta capa es sin duda el **ratio**, el cual tiene un valor entre [0, 1] y define el porcentaje de neuronas que se expulsarán de la red en cada iteración.

- ***Conv2DTranspose***

  Esta capa efectúa varias convoluciones transpuestas sobre la información (imagen) que recibe como entrada. Las convoluciones transpuestas son una forma de realizar el *upsampling* mediante la operación *convolución*; tal y como se puede observar en la imagen 3.18, en verde se representa el resultado final (*upsampling*), mientras que los cuadrados sombreados representan el *kernel* utilizado para la convolución transpuesta. Además, los cuadrados blancos representan los valores nulos que se añaden al mapa original (azul) para aumentar la dimensionalidad.

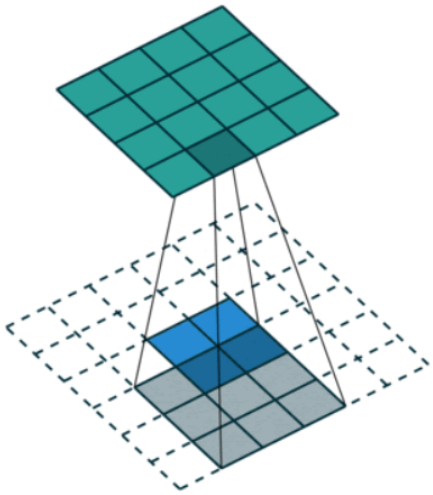


Figura 3.18: Convolución transpuesta.

Fuente: http://deeplearning.net/software/theano/_images/no_padding_no_strides_transposed.gif

Los parámetros más importantes son las siguientes:

- **Número de filtros**: Determina la cantidad de filtros que utilizara la capa para el aprendizaje; además, también especifica el número de filtros (dimensión) de la salida.
- **Dimensión del *kernel***: Determina la altura y anchura del *kernel* que efectuará la convolución transpuesta.
- ***Paso* del kernel**: Determina cual será la cantidad de unidades que se moverá el *kernel* durante la ejecución del proceso.
- ***Padding***: Si su valor es *valid*, no se efectúa el rellenado; en cambio, si su valor es *same*, se lleva a cabo un relleno uniforme hacia arriba/abajo o izquierda/-derecha, de modo que la salida tendrá la misma dimensión que la entrada.
- **Tipo de activación**: Determina que función de activación se usará después de efectuar las convoluciones transpuestas.
- **Inicializado del *kernel***: Define cual es la función que determinará con que pesos se inicializa el *kernel* empleado en las convoluciones transpuestas.

- ***Concatenate***

  La función de esta capa, como bien lo define su nombre, es concatenar las capas que recibe como entrada. En este caso, tiene como objetivo mejorar la precisión de la imagen, ya que la información pérdida durante las convoluciones es recuperada en la fase descodificadora juntando cada capa de esta última con su respectiva capa de la fase codificadora. Asimismo, el parámetro más importante es el **eje** sobre el que se realiza la concatenación, siendo el último eje el que se suele utilizar por defecto (en este caso, la profundidad de la imagen).

3.4.5. Compilación de las redes

A la hora de compilar una red, se han de definir cuales serán el optimizador, la función de coste y la métrica de evaluación (imagen 3.19); en este caso, los valores utilizados para las dos redes han sido los mismos.

```
model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

Figura 3.19: Compilación de la red *UNet* (misma configuración para la *arquitectura 1*).

La función de coste que se ha utilizado para llevar a cabo el proyecto ha sido la **entropía cruzada binaria**, el cual ha sido explicado y analizado en la sección teórica 2.3.1. Además, en este caso la entropía cruzada es binaria porque la clasificación se efectúa sobre dos clases: la clase que contiene *carretera* y la que no.

Tal y como se explicará después en la sección 3.4.6, el *batch* o lote de datos será de **32**; es decir, el modelo trabajará paralelamente con 32 ventanas a cada momento. De esta manera, después de pasar por el codificador y efectuar otros pasos, se obtendrá un vector de 32x1, donde cada elemento será la predicción realizada por el modelo para cada ventana. Después, se creará un vector de la misma dimensión con los valores reales, y finalmente los vectores de las predicciones y valores reales serán utilizados por la función de coste para calcular el error.

De esta forma, después de calcular el error de cada uno de los lotes se aplica la función de optimización de la función de coste; en este caso, tal y como se ha explicado en la sección teórica 2.3.1, se ha utilizado el optimizador ***Adam***. El optimizador se ocupará de ajustar los pesos de la mejor manera posible en cada iteración, efectuando actualizaciones del gradiente para obtener una mejoría.

Finalmente, la métrica es una función utilizada para evaluar el rendimiento del modelo (su rol es muy parecido al de la función de coste). En este caso, la métrica utilizada ha sido el ***accuracy***, y su valor es obtenido calculando con que frecuencia la predicción es igual a su etiqueta.

3.4.6. Entrenamiento de las redes

La función para llevar a cabo el entrenamiento ha sido *fit* (figura 3.20), y los parámetros utilizados para ambas redes han sido iguales: los datos para el entrenamiento ***(X_train, Y_train)***, los datos para la validación ***(X_valid, Y_valid)***, y finalmente, la cantidad de épocas. El valor para este último parámetro ha sido **30**, y define el número de veces que se iterará (con el fin de reducir el error) sobre la totalidad de los datos. Asimismo, aunque no se especifique en la invocación de la función ya que se utilizará el valor por defecto, el *batch* o lote de datos será de **32**.

```
unet.fit(X_train_red, Y_train_red, epochs = 30, validation_data = (X_valid_red, Y_valid_red))
```

Figura 3.20: Entrenamiento de la red *UNet* (misma configuración para la *arquitectura 1*).

3.4.7. Guardar las variables localmente

Tal y como se hizo con el conjunto de datos en la parte del preprocesado (sección 3.3.4), los diferentes métodos computacionales se guardarán localmente. El *SVM* y el *Random Forest* se guardarán mediante el módulo *pickle*; las redes neuronales, en cambio, se guardarán utilizando el método *save* ofrecido por la librería *Keras*.

3.5. Test

Después de finalizar el entrenamiento, los modelos se testearán sobre diferentes conjuntos de datos utilizando la métrica *Intersection Over Union* (*IoU*). Primero, el test se realizará sobre un conjunto de ventanas elegidas aleatoriamente, y finalmente se efectuará un testeo real recorriendo todos los píxeles de las imágenes completas.

3.5.1. Métrica de evaluación

Información principalmente extraída de la siguiente página: [1]

El *Intersection Over Union* (*IoU*), también conocido como *índice Jaccard* y publicado por primera vez en el artículo [Jaccard, 1901], ha sido la métrica de evaluación que se ha utilizado para valorar la calidad de los modelos del proyecto. Esta métrica evalúa la precisión de una segmentación propuesta para una imagen, realizando una comparación con una segmentación sobre *ground truth*. Para tareas de clasificación de imágenes, es preferible usar el *IoU* a usar el *accuracy*, ya que si el problema está mal balanceado el *IoU* da medidas menos sesgadas por la clase mayoritaria; por ejemplo, si en un problema el 90 % de las píxeles son de fondo (*no carretera*), un modelo que clasifique todos los píxeles como tal obtendrá un 90 % de *accuracy*, mientras que conseguirá un *IoU* del 10 %.

El valor del *IoU* se obtiene dividiendo el área de intersección (entre la predicción y los valores reales) por el área de unión (figuras 3.21 y 3.22); además, basándose en una matriz de confusión, la función se puede escribir de la siguiente manera:

$$IoU(Y,\hat{Y}) = \frac{Y \cup \hat{Y}}{Y \cap \hat{Y}} = \frac{TP}{TP+FN+FP} \tag{3.5}$$

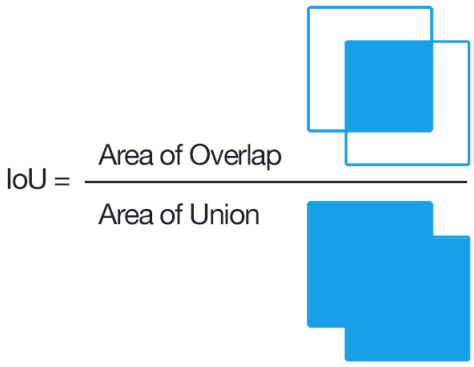


Figura 3.21: Ecuación del *Intersection over Union*.

Fuente:
https://www.pyimagesearch.com/wp-content/uploads/2016/09/iou_equation.png

```
def IoU(valorReal, prediccion):
    interseccion = np.logical_and(valorReal, prediccion)
    union = np.logical_or(valorReal, prediccion)
    iou = np.sum(interseccion) / np.sum(union)
    return iou
```

Figura 3.22: Implementación del *Intersection over Union*.

3.5.2. Cargar datos y modelos

Para cargar los conjuntos de datos (ventanas e imágenes completas), el *SVM* y el *Random Forest* se utilizará la función *load* del módulo *pickle*; para las redes, en cambio, se empleará el método *load_model* ofrecido por la librería *Keras*.

3.5.3. Testear en ventanas

Una vez cargado el conjunto de datos para el test de las ventanas ***(X_test, Y_test***), se ha de utilizar la función *predict* para llevar a cabo la predicción (la entrada será *X_test*). Después de realizar este cómputo, la salida de la misma y el vector de valores reales (*Y_test*) serán procesados por la función *IoU* para obtener el resultado final (imagen 3.23).

En el caso de las redes, ya que la predicción es una probabilidad sobre cada una de las dos clases, se ha de crear otro vector binario unidimensional (imagen 3.24), donde el valor de un píxel será **0** si la clase *no carretera* tiene más porcentaje que la clase *carretera*, y **1** si se da el caso opuesto.

```
SVM_prediction = clf.predict(X_test)
iou_SVM_ventanas = IoU(Y_test, SVM_prediction)
print('IoU para SVM en clasificación de ventanas:', iou_SVM_ventanas)
```

Figura 3.23: Testeo sobre las ventanas.

```
unet_prediction_to_reshape = np.zeros(len(unet_prediction), dtype = np.int32)
for i in range(len(unet_prediction_to_reshape)):
    if (unet_prediction[i][1] > unet_prediction[i][0]):
        unet_prediction_to_reshape[i] = 1
```

Figura 3.24: Transformación de las predicciones de la red para adecuarlas a las necesidades de la función *IoU*.

3.5.4. Testear en imágenes completas

El proceso de testeo en imágenes completas será el mismo que en las ventanas, pero utilizando el conjunto de datos ***(X_entire, Y_entire***). Además, mediante el conjunto de funciones disponibles en *pyplot* (de la librería *matplotlib*), se obtendrá una representación gráfica de la máscara. Finalmente, por medio de la librería *cv2* (versión de *OpenCV* para *Python*) se procesará la máscara para guardarla localmente (figura 3.25).

```
unet_prediction_entire_reshaped = np.reshape(unet_prediction_entire_to_reshape, (1024, 1024))
plt.figure(figsize = (7.5, 7.5))
plt.imshow(unet_prediction_entire_reshaped, cmap ='gray')
plt.title('Unet prediction')
plt.show()

unet_prediction_entire_reshaped_to_save = cv2.convertScaleAbs(unet_prediction_entire_reshaped, alpha=(255.0))
cv2.imwrite(folder_path_save_results + 'unet_no_transformations.png', unet_prediction_entire_reshaped_to_save)
```

Figura 3.25: Código para obtener la representación gráfica de la máscara.

3.6. Transformaciones morfológicas

En esta sección se aplicarán dos de las transformaciones morfológicas más conocidas (explicadas en la sección teórica 2.5.1): la apertura (*opening*) y el cierre (*closing*). Principalmente, el objetivo de estas transformaciones es mejorar el resultado obtenido en las predicciones aplicando operaciones simples sobre los valores de la imagen.

El parámetro a definir más importante en estas transformaciones es el **objeto estructural**, el cual decide la naturaleza de la operación y se desliza sobre los píxeles de la imagen. La

diferencia entre estas operaciones es cómo se procesa el resultado después de aplicar el objeto estructural; por ejemplo, en el caso de la erosión (operación básica en la apertura y el cierre), un píxel tendrá el valor **1** (*carretera*) si todos los píxeles que caen dentro del objeto estructural tienen también el valor **1**. En este caso, se ha utilizado un objeto estructural con forma cuadrada (muy utilizado en las transformaciones morfológicas), de tamaño 5x5 y compuesto totalmente por elementos de valor **1** (figura 3.26).

```
kernel = np.ones((5,5), dtype = np.uint8)
```

Figura 3.26: Implementación del *kernel*.

Las transformaciones se realizarán mediante la función *morphologyEx* (figura 3.27) de la librería *cv2*, siendo los siguientes parámetros los más importantes:

- La imagen sobre el que se va a realizar la transformación.
- El tipo de transformación.

  En este caso, las funciones utilizadas han sido *cv2.MORPH_OPEN* (apertura) y *cv2.MORPH_CLOSE* (cierre).
- El *kernel*.

```
unet_closing = cv2.morphologyEx(unet_prediction_entire_reshaped_uint8, cv2.MORPH_CLOSE, kernel)
```

Figura 3.27: Función para realizar las transformaciones morfológicas.

4. CAPÍTULO

Resultados y evaluación

Una vez explicado como se ha desarrollado el proyecto, se debe evaluar el trabajo realizado para ver como se desenvuelven los diferentes modelos implementados dependiendo de diversos parámetros y contextos. Para ello, la evaluación se dividirá en tres partes:

- Se analizará la fase de entrenamiento sobre las ventanas seleccionadas, comparando los tiempos de aprendizaje y, en el caso de las redes, observando cómo varía la función de coste en diferentes contextos.
- Se llevara a cabo la fase de test, la cual estará dividida en dos partes principales:
  - Testear sobre ventanas.

    Se elegirán ventanas aleatoriamente (distribuidas uniformemente entre las dos clases) y se evaluarán los modelos sobre ellas.
  - Testear sobre imágenes completas.

    Dos imágenes totalmente diferentes serán elegidas para que los modelos puedan evaluarse en diferentes contextos. La primera imagen será realmente sencilla, con una sola carretera y con muy pocas características destacables; la segunda, en cambio, tendrá múltiples carreteras, además de tener muchos elementos más (casas, árboles...) que dificultan la detección de carreteras en la imagen.
- Se realizarán dos transformaciones morfológicas (apertura y cierre) sobre cada una de las predicciones, con el fin de mejorar los resultados obtenidos.

Además, para analizar la importancia de la dimensión de las ventanas sobre los modelos implementados, toda la evaluación se realizará utilizando dos dimensiones diferentes:

- Ventanas de **16x16**.
- Ventanas de **32x32**.

4.1. Entrenamiento

El primer paso será definir con cuantos datos se entrenaran los modelos; además, ya que (en el caso de las redes) la función *fit* ofrece efectuar la validación a la par que el entrenamiento, también se definirán aquí los datos que se utilizarán para la validación:

- Datos para el entrenamiento.

  Los modelos se entrenarán utilizando **300000** ventanas, obtenidas de 3000 imágenes diferentes. Las ventanas estarán uniformemente distribuidas entre las clases *carretera* y *no carretera*, ya que de cada imagen se obtendrán 50 ventanas por clase.

- Datos para la validación.

  Se seguirá el mismo proceso para obtener los datos de validación, exceptuando que en vez de 3000 imágenes se utilizarán 1000, obteniendo en total **100000** ventanas uniformemente distribuidas entre las dos clases.

Una vez definidos los datos, se llevarán a cabo los procesos de entrenamiento y validación. Los tiempos para realizar estas determinadas tareas están definidas en la tabla 4.1; asimismo, se ha efectuado una comparación de los tiempos de ejecución en términos del uso del *GPU*.

Tiempos de entrenamiento				
Modelo	16x16		32x32	
	Con *GPU*	Sin *GPU*	Con *GPU*	Sin *GPU*
SVM	-	50 min	-	8 h 30 min
Random Forest	-	20 min	-	30 min
Arquitectura 1	13 min	1 h 30 min	40 min	5 h 30 min
UNet	4 h	4 días 7 h	5h 30 min	8 días 18 h

Tabla 4.1: Tiempos de entrenamiento.

En el caso de las redes, se dispone de la opción de analizar como varían los errores de entrenamiento (función de coste) y validación sobre todas las iteraciones; por ello, en la tabla 4.2 se mostrarán los resultados finales, acompañados por una gráfica (figura 4.1) donde se puede observar como cambian los errores de entrenamiento en función de las iteraciones.

Errores de entrenamiento y validación				
Medida	16x16		32x32	
	Arquitectura 1	*UNet*	Arquitectura 1	*UNet*
Error de entrenamiento	0.42	0.27	0.35	0.14
Accuracy (entrenamiento)	0.8	0.88	0.84	0.94
Error de validación	0.46	0.38	0.38	0.28
Accuracy (validación)	0.78	0.83	0.83	0.89

Tabla 4.2: Errores de entrenamiento y validación.

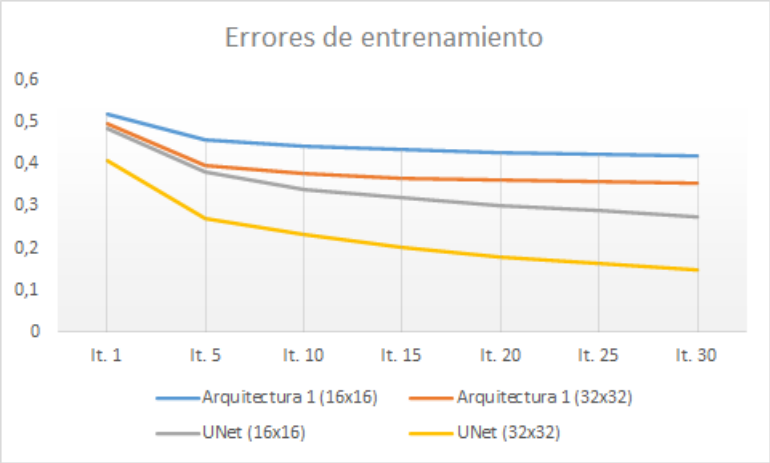


Figura 4.1: Representación gráfica de los errores de entrenamiento.

4.1.1. Discusión

Discusión acerca del tiempo de entrenamiento

Tal y cómo se puede intuir, cuanto más grande sea el tamaño de las ventanas más tiempo llevará la fase de entrenamiento. En el caso del *Random Forest*, la diferencia no es demasiado grande; en cambio, en el *SVM* y las redes, la desigualdad es muy notable. Cabe

destacar que en el caso del *SVM*, debido a que los cálculos incrementan exponencialmente con la dimensión de la entrada, el tiempo de ejecución es 7-8 veces mayor con las ventanas de 32x32.

Otro aspecto a tener en cuenta es el uso del *GPU* ofrecido por *Google Colaboratory*. En el caso de la librería *scikit - learn* (al cual pertenecen las implementaciones del *SVM* y *Random Forest*) no se puede utilizar esta herramienta ya que está diseñada únicamente para utilizarla en un entorno de aprendizaje profundo; en cambio, en el caso de las redes neuronales esta función sí está habilitada, y gracias a ella se puede realizar el entrenamiento en un tiempo razonable (por ejemplo, el tiempo de ejecución de la arquitectura *UNet* utilizando ventanas de 32x32 **sin *GPU*** es absolutamente inviable).

Discusión acerca de los errores de entrenamiento y validación

Habitualmente no se reportan los resultados de entrenamiento y validación, pero en este caso se realiza puesto que el proceso de aprendizaje dista mucho de ser perfecto. En cuanto a las medidas de las redes, se puede observar que los resultados mejoran a medida de que el tamaño de las ventanas aumenta. Además, la red *UNet aprende* mejor que la *arquitectura 1*, obteniendo mejores resultados incluso utilizando un tamaño de ventana menor ($0.27 < 0.35$). El mejor de los resultados se consigue utilizando la red *UNet* con ventanas de 32x32, obteniendo un error realmente bajo (**0.14**).

4.2. Test

Tal y cómo se ha mencionado anteriormente, la fase de test se dividirá en dos partes: el testeo sobre ventanas y el testeo sobre imágenes completas.

4.2.1. Testear sobre ventanas

Para esta fase se han utilizado **100000** ventanas, obtenidas de 1000 imágenes diferentes extrayendo de cada una de ellas 50 ventanas por cada clase. Los resultados se muestran en la tabla 4.3.

Testear sobre ventanas		
Modelo	IoU	
	16x16	32x32
SVM	0.5	0.52
Random Forest	0.64	0.67
Arquitectura 1	0.66	0.72
UNet	0.73	0.81

Tabla 4.3: Testeo sobre ventanas.

Discusión acerca del testeo sobre ventanas

La conclusión principal de este testeo es que las ventanas de mayor dimensión obtienen mejores resultados que las de menor dimensión. Además, las redes obtienen, en cualquiera de los casos, valores de *IoU* más altos que el *SVM* y *Random Forest*; asimismo, como es de esperar, los mejores resultados se obtienen mediante la arquitectura *UNet*, obteniendo un **0.81** de *IoU* cuando se trabaja con ventanas de 32x32.

4.2.2. Testear con imágenes completas

Esta es la parte más importante del testeo, donde los modelos se pondrán a prueba en una situación real. La información a procesar es muy grande, ya que las imágenes son de 1024x1024 y por cada píxel se creará una ventana de 16x16 o 32x32; por ello, ha sido imprescindible el uso de los recursos ofrecidos por *Google Colaboratory* (25GB de *RAM* y el *GPU*). Asimismo, cabe destacar que la cantidad de píxeles evaluados incrementa considerablemente con la dimensión de las ventanas (en este caso, 768 píxeles de diferencia por cada ventana procesada), y el problema se convierte inmanejable a partir de las dimensiones utilizadas en el proyecto por la manera en la que se han tratado, almacenado y recuperado las imágenes del *dataset*.

En cuanto a la elección de las imágenes, se ha optado por escoger una imagen sencilla y otra más compleja, con el fin de evaluar como se desenvuelven los diferentes modelos en cada uno de los casos:

- Imagen ***sencilla***.

  En la máscara de la imagen (figura 4.2b) se representa una única carretera situada en medio de un prado o bosque; a simple vista, no hay elementos que pueden llevar a la confusión, aunque existen varios senderos visibles en la imagen *RGB* (parte inferior izquierda, por ejemplo) donde los modelos pueden llegar a equivocarse en la predicción.

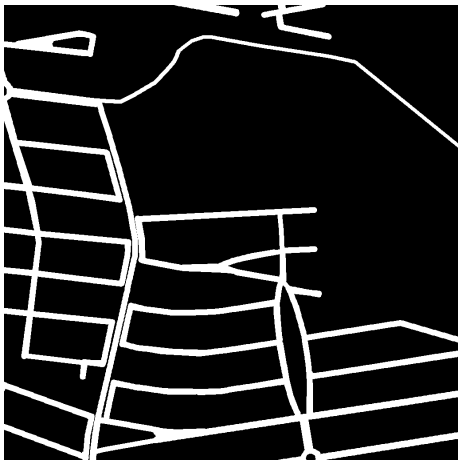
(a) Imagen *sencilla* original. **(b)** Máscara original de la imagen *sencilla*.

Figura 4.2: Imagen *sencilla*.

- Imagen ***compleja***.

  A diferencia del caso anterior, en la máscara (figura 4.3b) se pueden distinguir múltiples carreteras; asimismo, tal y como se puede observar en la imagen *RGB* (figura 4.3a), existen varios elementos externos (edificios, árboles, cruces...) que hay que tener muy en cuenta a la hora de efectuar la predicción, ya que se corre el riesgo de obtener muchos *Falsos Positivos* si no se logra diferenciarlos bien.

(a) Imagen *compleja* original.



(b) Máscara original de la imagen *compleja*.

Figura 4.3: Imagen *compleja*.

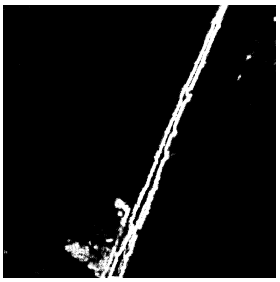
Imagen *sencilla*

A continuación, en la tabla 4.4 se mostrarán los resultados numéricos obtenidos en la imagen *sencilla*, y después, en las figuras 4.4 y 4.5 se proyectarán las máscaras predichas.

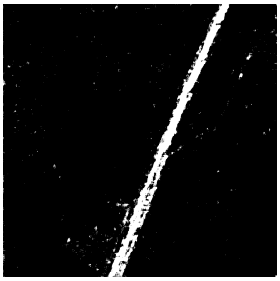
Testear sobre imagen *sencilla*		
Modelo	IoU	
	16x16	32x32
SVM	0.51	0.53
Random Forest	0.26	0.35
Arquitectura 1	0.44	0.52
UNet	0.44	0.48

Tabla 4.4: Testeo sobre imagen *sencilla*.

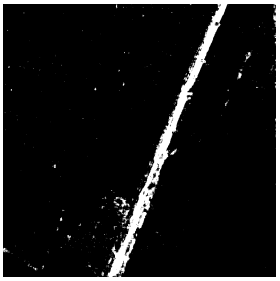
(a) Predicción del *SVM*.



(b) Predicción del *Random Forest*.

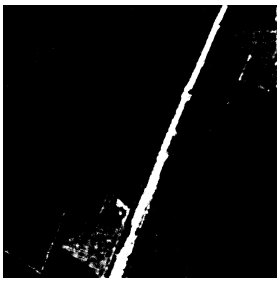


(c) Predicción de la *arquitectura 1*.

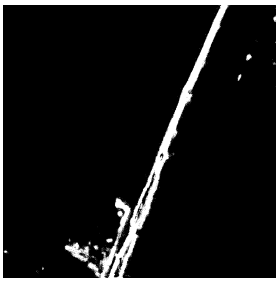


(d) Predicción de la red *UNet*.

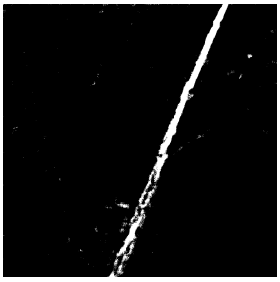
Figura 4.4: Predicción de la imagen *sencilla* con ventanas de 16x16.



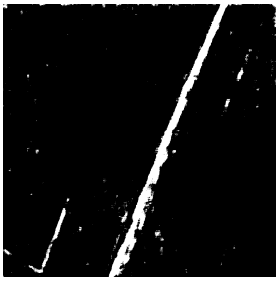
(a) Predicción del *SVM*.



(b) Predicción del *Random Forest*.



(c) Predicción de la *arquitectura 1*.



(d) Predicción de la red *UNet*.

Figura 4.5: Predicción de la imagen *sencilla* con ventanas de 32x32.

Discusión acerca de las predicciones de la imagen *sencilla*

Tal y cómo ha sucedido hasta ahora, los mejores resultados se han obtenido utilizando ventanas de 32x32, mejorando en todos los casos los valores del *IoU* conseguidos con ventanas de 16x16. En cuanto a las modelos, se puede observar que el *Random Forest* obtiene los peores resultados, logrando un valor de 0.35 (figura 4.5b) en el mejor de los casos. Asimismo, el *SVM* obtiene los mejores resultados, alcanzando un 0.53 en la predicción con ventanas de 32x32; estos buenos resultados probablemente se deben a la poca complejidad de la imagen (una única carretera fácilmente reconocible), y por ello la aproximación lineal efectuada por esta técnica obtiene resultados más que aceptables. Finalmente, en cuanto a las redes, se puede contemplar que los resultados obtenidos son parecidos entre sí (un poco peores que las del *SVM*), siendo ligeramente favorables a la *arquitectura 1*; esto sucede a causa de que la red *UNet*, al ser una arquitectura más sofisticada y capacitada para detectar más detalles, localiza pequeños fragmentos de senderos (ver figura 4.5d) que no están definidas como *carretera* en la máscara original (por ejemplo, parte inferior izquierda de la imagen original). En cambio, la *arquitectura 1* no detecta estos pequeños fragmentos, por lo que obtiene mejores resultados que la red *UNet*.

Imagen *compleja*

A continuación, en la tabla 4.5 se mostrarán los resultados numéricos obtenidos en la imagen *compleja*, y finalmente en las figuras 4.6 y 4.7 se proyectarán las máscaras predichas.

Testear sobre imagen *compleja*		
Modelo	IoU	
	16x16	32x32
SVM	0.28	0.3
Random Forest	0.24	0.26
Arquitectura 1	0.24	0.35
UNet	0.41	0.51

Tabla 4.5: Testeo sobre imagen *compleja*.

(a) Predicción del *SVM*.

(b) Predicción del *Random Forest*.

(c) Predicción de la *arquitectura 1*.

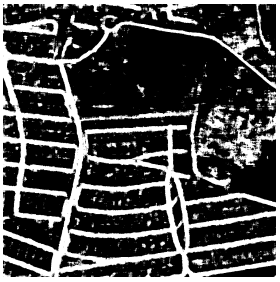
(d) Predicción de la red *UNet*.

Figura 4.6: Predicción de la imagen *compleja* con ventanas de 16x16.

(a) Predicción del *SVM*.

(b) Predicción del *Random Forest*.

(c) Predicción de la *arquitectura 1*.



(d) Predicción de la red *UNet*.

Figura 4.7: Predicción de la imagen *compleja* con ventanas de 32x32.

Discusión acerca de las predicciones de la imagen *compleja*

Siguiendo con la misma dinámica, los mejores resultados se obtienen con las ventanas de 32x32. En cambio, en esta imagen sucede justo lo contrario a lo ocurrido en la anterior: al ser más compleja, donde tienen presencia más carreteras y elementos externos (edificios, cruces...), solo la red *UNet* es capaz de distinguir las carreteras correctamente (figura 4.7d), obteniendo un muy buen resultado en el caso de las ventanas de 32x32 (0.51). Los resultados del *SVM* bajan drásticamente (0.3) debido a que la aproximación linear del problema no es nada bueno; lo mismo sucede con la *arquitectura 1*, la cual es incapaz de reconocer únicamente las carreteras y define como *carretera* casi todos los elementos externos. El peor de los casos es el *Random Forest* (0.26 en su mejor predicción), el cual determina muchísimos *Falsos Positivos* detectando como *carretera* todos los edificios.

4.3. Transformaciones morfológicas sobre las predicciones de imágenes enteras

En esta sección, se aplicarán la **apertura** (*opening*) y el **cierre** (*closing*) sobre las predicciones previamente efectuadas. Estas operaciones no requieren de demasiados recursos, ya que realizan operaciones muy sencillas sobre los píxeles de las máscaras.

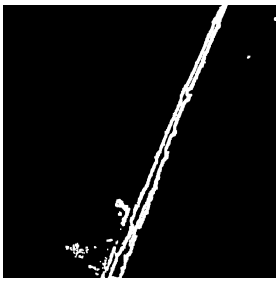
4.3.1. Apertura (*Opening*)

A continuación, en la tabla 4.6 se mostrarán los resultados numéricos obtenidos mediante la apertura, y después se proyectarán las máscaras transformadas de la imagen *sencilla* (figuras 4.8 y 4.9) y de la *compleja* (figuras 4.10 y 4.11).

Apertura (*opening*)				
Modelo	IoU			
	Imagen *sencilla*		Imagen *compleja*	
	16x16	32x32	16x16	32x32
SVM	0.57	0.59	0.3	0.33
Random Forest	0.29	0.38	0.29	0.31
Arquitectura 1	0.5	0.56	0.27	0.41
UNet	0.5	0.54	0.49	0.59

Tabla 4.6: Apertura (*opening*).

(a) Apertura (*SVM*).



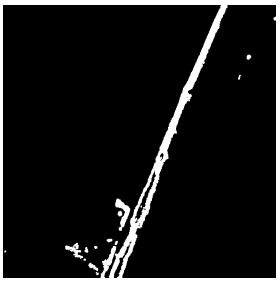
(b) Apertura (*Random Forest*).

(c) Apertura (*arquitectura 1*).

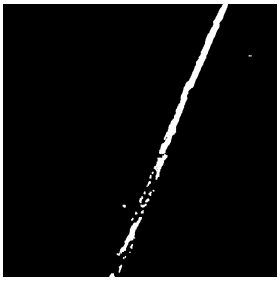
(d) Apertura (*UNet*).

Figura 4.8: Apertura (*opening*) de la imagen *sencilla* con ventanas de 16x16.

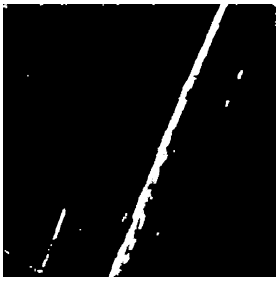
(a) Apertura (*SVM*).



(b) Apertura (*Random Forest*).



(c) Apertura (*arquitectura 1*).



(d) Apertura (*UNet*).

Figura 4.9: Apertura (*opening*) de la imagen *sencilla* con ventanas de 32x32.

(a) Apertura (*SVM*).

(b) Apertura (*Random Forest*).

(c) Apertura (*arquitectura 1*).

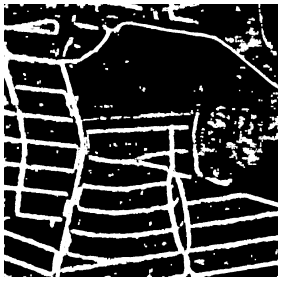
(d) Apertura (*UNet*).

Figura 4.10: Apertura (*opening*) de la imagen *compleja* con ventanas de 16x16.

(a) Apertura (*SVM*).

(b) Apertura (*Random Forest*).

(c) Apertura (*arquitectura 1*).



(d) Apertura (*UNet*).

Figura 4.11: Apertura (*opening*) de la imagen *compleja* con ventanas de 32x32.

Discusión acerca de la apertura (*opening*)

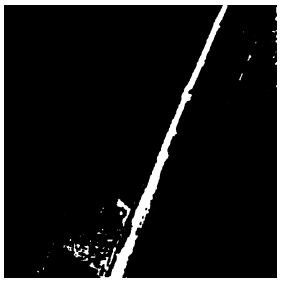
Los resultados de la apertura son realmente buenos, ya que todos las valores mejoran con respecto a las predicciones originales; esto se debe a que la mayoría de *Falsos Positivos* que están aislados acaban siendo eliminados, ya que esta técnica se ocupa de expulsar estos pequeños salientes. Asimismo, aunque también se hayan obtenido resultados favorables con las ventanas pequeñas, los mejores resultados han sido logrados con ventanas de 32x32. Cabe destacar que los valores máximos del *IoU* de ambas imágenes (0.59 en los dos casos) se han obtenido mediante esta técnica, y observando las máscaras (4.9a para la imagen *sencilla* y 4.11d para la imagen *compleja*) se puede declarar que los resultados se asemejan correctamente a la máscara original. Finalmente, es preciso señalar que la mejoría ha estado en todos los casos entre [0.2 - 0.8], siendo el más bajo el caso del *SVM* con ventanas de 16x16 y el más alto el caso de la red *UNet* con ventanas de 32x32 (con la imagen *compleja* en ambos casos).

4.3.2. Closing

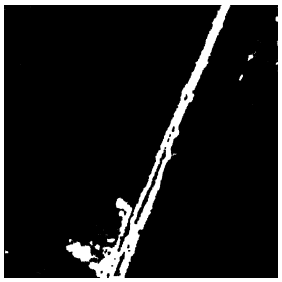
A continuación, en la tabla 4.7 se mostrarán los resultados numéricos obtenidos mediante el cierre (*closing*), y finalmente se proyectarán las máscaras transformadas de la imagen *sencilla* (figuras 4.12 y 4.13) y de la *compleja* (figuras 4.14 y 4.15).

Cierre (*closing*)				
Modelo	IoU			
	Imagen *sencilla*		Imagen *compleja*	
	16x16	32x32	16x16	32x32
SVM	0.5	0.45	0.27	0.26
Random Forest	0.25	0.33	0.22	0.22
Arquitectura 1	0.4	0.5	0.22	0.3
UNet	0.4	0.43	0.37	0.46

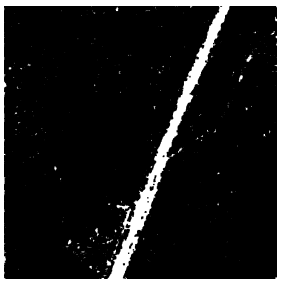
Tabla 4.7: Cierre (*closing*).



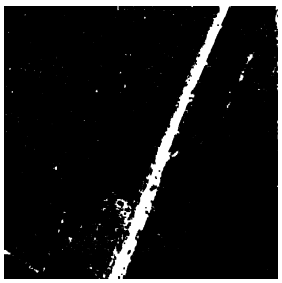
(a) Cierre (*SVM*).



(b) Cierre (*Random Forest*).

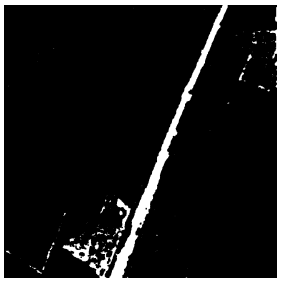


(c) Cierre (*arquitectura 1*).

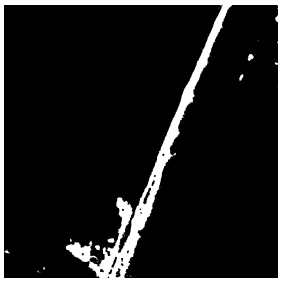


(d) Cierre (*UNet*).

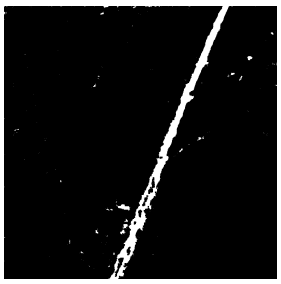
Figura 4.12: Cierre (*closing*) de la imagen *sencilla* con ventanas de 16x16.



(a) Cierre (*SVM*).



(b) Cierre (*Random Forest*).



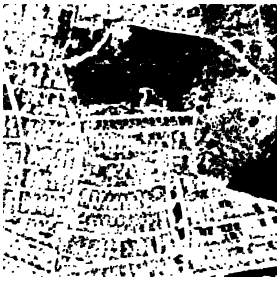
(c) Cierre (*arquitectura 1*).

(d) Cierre (*UNet*).

Figura 4.13: Cierre (*closing*) de la imagen *sencilla* con ventanas de 32x32.

(a) Cierre (*SVM*).

(b) Cierre (*Random Forest*).



(c) Cierre (*arquitectura 1*).

(d) Cierre (*UNet*).

Figura 4.14: Cierre (*closing*) de la imagen *compleja* con ventanas de 16x16.

(a) Cierre (*SVM*).

(b) Cierre (*Random Forest*).

(c) Cierre (*arquitectura 1*).

(d) Cierre (*UNet*).

Figura 4.15: Cierre (*closing*) de la imagen *compleja* con ventanas de 32x32.

Discusión acerca del cierre (*closing*)

Si bien la técnica de la apertura ha servido para mejorar los resultados, el cierre ha empeorado las predicciones originales; debido a que esta operación elimina pequeños huecos clasificándolos como *carretera*, el número de los *Falsos Positivos* aumenta. Este fenómeno se puede ver claramente reflejado en la figura 4.15b, donde los píxeles incorrectamente clasificados cómo *carretera* ocupan la mayoría de la superficie de la máscara; por ello, el peor resultado de todo el proyecto se ha obtenido en esta transformación, obteniendo un pobre 0.22. Finalmente, el rango de empeoramiento ha sido [0.1 - 0.8], ambas obtenidas por el *SVM* en la imagen *sencilla* (con ventanas de 16x16 y 32x32, respectivamente).

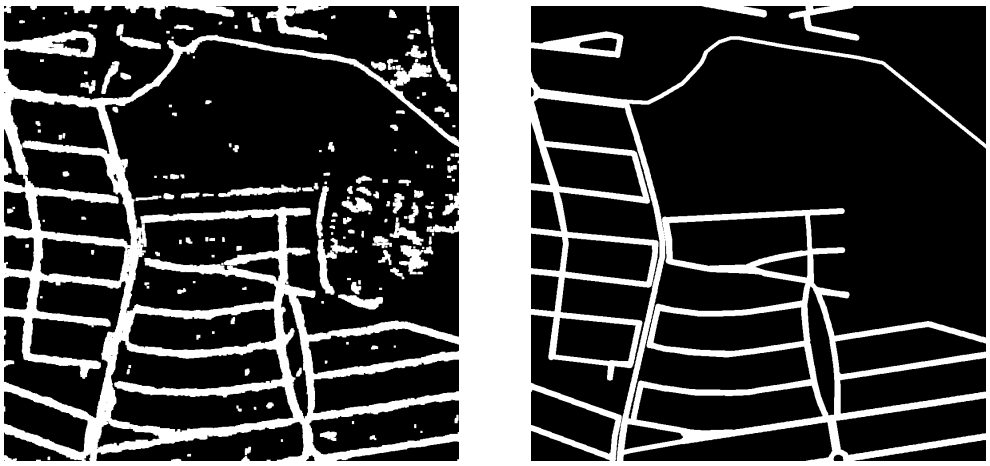
4.4. Evaluación general

La primera evaluación general (y una de las más importantes) es que los resultados mejoran utilizando ventanas más grandes, obteniendo mejoras de hasta 12 % en las predicciones originales; esto se debe a que es más fácil reconocer ciertos patrones o características disponiendo de una superficie más grande, ya que utilizando ventanas más pequeñas hay menos probabilidad de que el modelo llegue a reconocer el área que está clasificando.

En cuanto a los modelos se refiere, se han podido observar los pros y los contras de cada uno de ellos. Para empezar, analizando los resultados del ***Random Forest*** se puede declarar que no es muy efectivo para este tipo de problemas; esto se produce porque, aunque teóricamente sí pueda llegar a dar buenos resultados, en la práctica no es capaz de llevar a cabo la extracción de características de una manera eficiente (permitiendo el aumento del conjunto de características en tiempo real). En cuanto al ***SVM*** se refiere, se ha podido contemplar que sólo responde bien ante un tipo determinado de imagen donde la clasificación se pueda aproximar linealmente y no tenga demasiadas características; aun así, este tipo de imagen sólo constituye una muy pequeña parte del *dataset*, por lo que el *SVM* no se puede considerar cómo una buena solución al problema. Finalmente, se ha podido observar que la solución a este problema está en las redes convolucionales: la ***arquitectura 1***, al ser muy básica (se debe recordar que está compuesta por un par de capas de convolución y *pooling*), no tiene la capacidad de aprender correctamente las diferentes características que se pueden encontrar en una imagen y no logra buenos resultados en la mayoría de los contextos; en cambio, la red ***UNet*** sí está capacitada para dar una buena respuesta a este problema. A pesar de que en ciertos contextos, cómo en la imagen *sencilla*, pueda llegar

a confundir ciertos fragmentos prediciendo *Falsos Positivos* (debido a que la extracción de características se hace de una forma muy detallada), el resultado global que obtiene es realmente bueno; además, se podría debatir si estas *confusiones* (los senderos de la imagen *sencilla*, por ejemplo) son fragmentos mal clasificados o si realmente son errores en la creación de la máscara (es decir, que estos fragmentos no se hayan detectado cómo carretera al realizar el etiquetado original). Tal y cómo se ha podido experimentar en la imagen *compleja*, la red *UNet* ha sido el único modelo capaz de separar correctamente los elementos externos (cómo los edificios o cruces) de la carretera, obteniendo un resultado realmente superior a todos los demás.

Teniendo esto en cuenta, y sabiendo que la mayoría de las imágenes del *dataset* se asemejan a la imagen *compleja* (múltiples carreteras y elementos externos), se puede concluir que el mejor modelo es la red *UNet* y que los resultados obtenidos son más que aceptables. Prueba de ello es el mejor *IoU* obtenido con dicha imagen, logrando un valor de 0.59, el cual está muy cerca de la media obtenida por los ganadores del reto (0.63); además, esto se puede respaldar gráficamente, realizando una comparación visual entre la máscara obtenida y la original (figura 4.16).



(a) Predicción de la red *UNet*. **(b)** Máscara original.

Figura 4.16: Comparación entre la máscara obtenida por la red *UNet* y la original.

Para concluir con las evaluaciones generales, se llevará a cabo una breve comparación entre la **apertura** y el **cierre**. Se tratan de dos operaciones opuestas: mientras que la primera intenta eliminar los salientes (clasificados cómo *carretera*) que estén aislados, el segundo pretende cerrar huecos (clasificados como *no carretera*) juntando píxeles de *carretera*. Cómo bien se ha explicado anteriormente, este problema está caracterizado por el gran

número de *Falsos Positivos*, por lo que la operación más interesante y efectiva será la apertura, ya que eliminará muchos de estos casos. En cambio, en el caso del cierre, se crearán más *Falsos Positivos* de los que había en la predicción, debido a que se establecerán más casos cerca de los que ya estaban mal clasificados, obteniendo (cómo es de esperar) un peor resultado.

5. CAPÍTULO

Conclusiones y objetivos para el futuro

En este capítulo se explicarán las conclusiones (tanto del proyecto cómo personales), además de exponer cuales serían los siguientes pasos a dar para seguir adelante con el proyecto.

5.1. Conclusiones

El objetivo de esta sección es expresar que conclusiones se han podido obtener del proyecto y trabajo realizado, al igual que explicar cómo ha sido la experiencia personal de llevarla a cabo.

5.1.1. Conclusiones del proyecto

En general, los objetivos propuestos al iniciar el proyecto se han cumplido, y así lo demuestran los conocimientos y resultados obtenidos; aun así, ha habido ciertas tareas que se han complicado y se ha tenido que invertir más tiempo de lo esperado en ellos. Primero se llevó a cabo la tarea de la documentación, después se probaron y analizaron los cuatro modelos, y finalmente se ha concluido el proyecto comparando los resultados obtenidos. En general, aún teniendo en cuenta las limitaciones en términos de recursos (computacionales y de tiempo), los resultados logrados son buenos y lógicos; es decir, que los mejores resultados se obtengan mediante la red *UNet* tiene sentido, al igual que las ventanas más

grandes funcionen mejor que las pequeñas. A continuación, se explicarán las conclusiones generales acerca del proyecto.

Primero, para poder entender y trabajar con los diferentes modelos computacionales, se ha invertido un gran tiempo y esfuerzo en la documentación, entendiendo y analizando a fondo las bases teóricas. De esta manera, se han aprendido muchos conceptos nuevos, a la par que se han podido revisar los conocimientos ya adquiridos.

Después, se han definido cuatro modelos diferentes para llevar a cabo la tarea de detección de carreteras, analizando las potencialidades y puntos débiles de cada uno de ellos. Además, se han podido comparar en un entorno práctico, donde se ha podido observar cómo respondía cada uno de ellos a diferentes contextos; para llevar a cabo toda esta tarea, se ha utilizado el conjunto de datos ofrecido por los creadores del reto *DeepGlobe Road Extraction Challenge*, aunque solo se ha utilizado una parte de él ya que se ha tenido que reducirlo por las necesidades del proyecto.

Asimismo, se ha analizado el impacto que tienen los diferentes parámetros sobre los modelos, comparando los resultados obtenidos variando sus valores; por ello, se ha experimentado con diferentes tamaños de ventana, capas de las redes, tipos de *kernel*... hasta obtener los modelos y resultados finales.

Aun así, han ocurrido varios problemas a la hora de llevar a cabo las experimentaciones prácticas, sobre todo relacionadas con las limitaciones de memoria y tiempo; por ejemplo, el *SVM* no se podía testear localmente ya que superaba la memoria *RAM* disponible, al igual que entrenar localmente la red *UNet* era totalmente inviable (se necesitan 8 días para realizar esta tarea). Por ello, ha sido totalmente necesario el uso de la plataforma *Google Colaboratory*, la cual esta dotada de 25GB de *RAM* y de un *GPU* muy eficiente; aunque es cierto que a la hora de almacenar archivos está limitado (15GB en la versión básica), esta herramienta es realmente útil para llevar a cabo este tipo de proyectos.

Resumiendo, las conclusiones generales acerca del proyecto son realmente positivas, ya que se ha logrado tanto un gran conocimiento teórico acerca del tema como la habilidad para desenvolverse en el ámbito práctico del proyecto.

5.1.2. Conclusiones generales

Personalmente, el proyecto ha sido muy enriquecedor en todos los aspectos: se ha obtenido un gran conocimiento acerca del tema, y este conocimiento también se ha puesto en práctica donde se ha tenido que hacer frente a situaciones que no eran esperadas; asimis-

mo, se ha desarrollado la habilidad de detectar diferentes desviaciones y problemas, a la par que se ha logrado la capacidad para resolverlos.

Antes de empezar el proyecto, el nivel de conocimiento acerca de las técnicas de aprendizaje profundo y de la clasificación de imágenes era realmente bajo. Por ello, el primer paso fue informarse sobre los diferentes modelos comúnmente usados para hacer frente a este tipo de problema, examinando a fondo el ámbito de la redes neuronales (convolucionales). De esta manera, comparando la situación de entonces y la actual, se puede declarar que la cantidad de conceptos nuevos adquiridos es realmente grande.

Una parte muy importante del conocimiento adquirido es la habilidad de programación, sobre todo en el campo de las redes neuronales. Aún teniendo nociones básicas sobre la programación en *Python*, este proyecto ha servido para inicializarse con librerías como *Keras*, especializado en programación de redes neuronales; además, debido a que el procesamiento de las imágenes ha sido una parte fundamental del proyecto, librerías cómo *cv2* (*OpenCV*) han sido realmente importantes.

Asimismo, el conocimiento previamente obtenido en diferentes asignaturas ha sido realmente útil para la elaboración del proyecto. Las asignaturas *Machine Learning* e *Inteligencia Artificial* han servido para tener unas nociones mínimas acerca de las diferentes técnicas de aprendizaje profundo, al igual que asignaturas más matemáticas cómo *Estadística* o *Cálculo* han valido para entender las bases sobre las que están construidas los modelos.

En cuanto a la memoria, la tarea más difícil ha sido aclarar las ideas y decidir correctamente que es lo que se debía escribir. Es realmente clave explicar la información de una forma clara y sencilla, ya que la lectura de la misma tiene que ser fácil y ligera. Para ello, es necesario ordenar los conceptos antes de empezar a escribir, ya que invertir un esfuerzo en esta tarea puede ahorrar muchas pérdidas de tiempo.

La conclusión personal más importante ha sido el desarrollo de la habilidad para afrontar los problemas que han ido surgiendo durante el proyecto. Además de los problemas previamente planteados y previstos, ha habido incontables casos de desviaciones inesperadas que han tenido que ser resueltas de la mejor manera posible. Con este tipo de casos, en los que difícilmente se puede ver la solución al instante, es cuando más se pone uno a prueba y desarrolla la capacidad de solventar estos contratiempos.

Para finalizar con las conclusiones, y haciendo un análisis global del proyecto, se puede expresar que este último ha sido totalmente positivo en todos los aspectos. Además de poder nutrirse de nuevos conocimientos, también ha servido para el crecimiento personal,

afrontando por primera vez una experiencia de este tipo. Asimismo, el proyecto ha creado en el estudiante una gran curiosidad en cuanto al ámbito del aprendizaje profundo y clasificación de imágenes, abriendo nuevas puertas para un futuro cercano.

5.2. Objetivos para el futuro

En esta sección se analizarán las tareas que no se han podido realizar en el proyecto por diferentes limitaciones (computacionales, de tiempo...) y se identificarán múltiples aspectos que pueden ser interesantes para llevarlos a cabo en un futuro:

- Mejoras en los resultados y experimentaciones.

  Tal y cómo se ha mencionado anteriormente, varias mejoras relacionadas con la dimensión de los datos y la exploración de los parámetros pueden ser realizadas si se dispone de los recursos necesarios. Es decir, teniendo una capacidad computacional que pueda soportar grandes dimensiones de datos, los resultados obtenidos mejorarán ya que los modelos podrán ser entrenados con más contundencia. Además, si se dispusiera de más tiempo para llevar a cabo el proyecto, más pruebas podrían haberse realizado cambiando los diferentes parámetros que caracterizan los modelos, obteniendo así mejores resultados.

- Realizar un testeo más completo de los modelos.

  Debido a que la cantidad de cálculos que se deben realizar en la predicción de una imagen completa es realmente grande, se decidió que el test se haría individualmente sobre cada imagen. En cambio, si los recursos disponibles lo permiten, llevar a cabo este testeo sobre una cantidad considerable de imágenes y obtener el promedio de todas las predicciones sería una gran mejoría, ya que de este manera se obtendrían unos resultados más generalizados.

- Creación de nuevas arquitecturas.

  Además de intentar imitar (y mejorar) arquitecturas que ya hayan sido creadas, el siguiente objetivo sería crear una nueva arquitectura que sea capaz de mejorar los resultados obtenidos hasta ahora. Esta tarea podría llevarse a cabo partiendo desde cero (creando capa por capa una arquitectura entera) o combinando arquitecturas ya creadas; aun así, este cometido lleva consigo un gran tiempo y dedicación, el cual está totalmente fuera del alcance de este proyecto.

Anexos

A. ANEXO

Documento de los objetivos del proyecto

A.1. Descripción y objetivos del proyecto

El objetivo de este proyecto es entender, examinar a fondo y reproducir los diferentes métodos que se utilizan para resolver los problemas de detección de carreteras. Para ello, es necesario reforzar las bases teóricas y practicar con diferentes modelos computacionales.

A.2. Planificación del proyecto

A.2.1. Diagrama *EDT*

Mediante el diagrama de la *Estructura de la Descomposición del Trabajo* (*EDT*) se ha descompuesto el trabajo realizado durante el proyecto (figura A.1).

A.2.2. Paquetes de trabajo

Los paquetes de trabajo constituyen la parte más baja del diagrama *EDT* (figura A.1) y definen las actividades y tareas principales en las cuales se divide el proyecto. A continuación se dará una explicación detallada de cada uno de ellos, y en la tabla A.1 se definirá cuanto tiempo se ha invertido en cada paquete.

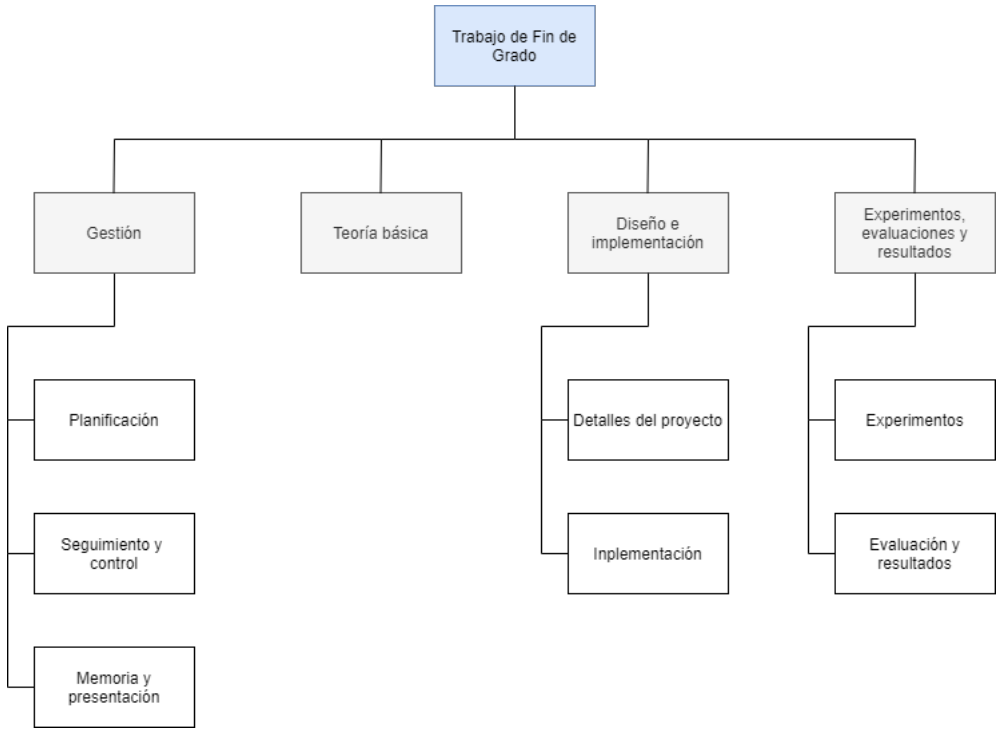


Figura A.1: Diagrama *EDT*.

Planificación

En esta sección se ha efectuado la planificación del proyecto. Para ello, se han especificado y definido los objetivos del proyecto, las tareas a realizar y el método de trabajo. Además, también se ha llevado a cabo un análisis de los peligros y riesgos del proyecto.

Seguimiento y control

El seguimiento y control será una sección que se gestionará durante todo el proyecto. Primero, se garantizará que la planificación principal y las fechas límite establecidas se cumplan; además, se asegurará de que el proyecto seguirá adelante aún produciéndose cualquier desviación (documentando estas últimas). Finalmente, será necesario efectuar reuniones de supervisión de una manera continua.

Memoria y presentación

Aquí se llevarán a cabo la memoria y la presentación, que se ocuparán de explicar como se ha desarrollado el proyecto.

- Memoria: Documento que recogerá los detalles del proyecto.

- Presentación: Documento que ayudará a efectuar la defensa del proyecto. Aquí, se mostrarán las partes más importantes del proyecto de una manera clara y ordenada.

Teoría básica

En esta sección se recogerán todos los conceptos básicos necesarios para llevar a cabo el proyecto; de esta manera, se analizarán múltiples métodos utilizados en problemas de detección de carretera, empezando desde los modelos más esenciales y llegando hasta las arquitecturas más complejas.

Detalles del diseño

Aquí se definirán todos los detalles de los diferentes modelos computacionales empleados para el proyecto.

Implementación

En esta sección se definirán las implementaciones de los modelos computacionales y se explicarán los cambios realizados respecto a las arquitecturas originales.

Experimentos

Se efectuarán experimentos para, sobre todo, ver y entender las diferencias entre los diferentes métodos computacionales; asimismo, se hará una exploración de los parámetros para analizar como afectan al funcionamiento de los modelos.

Evaluación y resultados

Se evaluarán los experimentos y se hará un análisis de los resultados. De esta manera, se obtendrán unas conclusiones finales que resumirán todo el trabajo realizado.

Paquete de trabajo	Duración
Gestión	**120**
Planificación	10
Seguimiento y control	20
Memoria y resultados	90
Teoría básica	**65**
Diseño e implementación	**130**
Detalles del diseño	40
Implementación	90
Experimentos, evaluaciones y resultados	**85**
Experimentos	65
Evaluación y resultados	20
TOTAL	**400**

Tabla A.1: Tiempo invertido en cada paquete de trabajo.

A.2.3. Entregables

Los entregables son productos o resultados que sirven para dar por finalizado una fase del proyecto; de esta manera, los entregables que han sido desarrollados durante el proyecto han sido los siguientes:

- El código.
- La memoria.
- La presentación.

A.2.4. Fechas límite

En la tabla A.2 se han fijado las fechas de finalización de cada uno de los entregables:

Entregable	Fecha de entrega
Código	06 / 09 / 2020
Memoria	06 / 09 / 2020
Presentación	14 - 18 / 09 / 2020

Tabla A.2: Fechas límite.

A.2.5. Diagrama de *Gantt*

En esta sección se muestra el diagrama de *Gantt* (A.2), en el cual se definen las fechas de comienzo y finalización de cada tarea, exponiendo el tiempo invertido en cada una de ellas.

Paquete de trabajo		Comienzo	Final	2020					
				Abril	Mayo	Junio	Julio	Agosto	Septiembre
Gestión	Planificación	01/04/2020	07/04/2020						
	Seguimiento y control	03/04/2020	05/09/2020						
	Memoria y presentación	02/08/2020	05/09/2020						
Teoría basíca		03/04/2020	22/04/2020						
Diseño e implementación	Detalles del proyecto	18/04/2020	27/04/2020						
	Inplementación	25/04/2020	30/07/2020						
Experimentos, evaluaciones y resultados	Experimentos	20/06/2020	28/08/2020						
	Evaluación y resultados	29/07/2020	28/08/2020						

Figura A.2: Diagrama de *Gantt*.

A.3. Metodología de trabajo

Principalmente, el proyecto se ha desarrollado **localmente** utilizando ***Jupyter Notebook***, un entorno de trabajo interactivo que permite desarrollar código en ***Python*** de manera dinámica, a la vez que integrar en un mismo documento tanto bloques de código como texto, gráficas o imágenes.

Hacía el final del proyecto, también se ha empezado a utilizar ***Google Colaboratory***, ya que ofrece una tarjeta *GPU* y mucha más memoria RAM de la que se obtiene trabajando localmente, siendo una gran herramienta para trabajar con redes neuronales.

A.3.1. Reuniones

El número de reuniones entre el estudiante y el tutor ha sido elevado. Las reuniones se han efectuado en función de las necesidades del proyecto y de la situación; es decir, si la tarea asignada se ha resuelto en pocos días, se ha realizado otra reunión lo antes posible para poder seguir desarrollando el proyecto. Dicho esto, normalmente se han efectuado 1-2 reuniones por semana; asimismo, el objetivo de las mismas ha sido valorar las tareas

asignadas en la reunión anterior y acordar los cometidos para la siguiente, con el fin de dar continuidad al proyecto.

Debido al *Covid-19*, todas las reuniones se han efectuado telemáticamente, utilizando la aplicación *Skype*.

A.3.2. Horarios planificados

El estudiante no ha tenido horarios predefinidos y ha tenido la libertad de elegir el ritmo de trabajo; aun así, ha intentando mantener una rutina y llevar a cabo una planificación.

A.4. Viabilidad

Una vez fijadas las diferentes secciones del proyecto, es imprescindible garantizar su viabilidad. Para ello, se definirán las siguientes pautas:

- **Coste de los recursos**

  Se garantizará que todos los recursos utilizados durante el proyecto sean gratuitos y que se tenga total accesibilidad a ellos.

- **Garantía del funcionamiento de los recursos**

  Se garantizará que todos estos recursos funcionen adecuadamente y que estén disponibles en cualquier momento y situación.

- **Tiempo**

  Se garantizará que se tenga suficiente tiempo para llevar a cabo el proyecto, teniendo en cuenta las desviaciones y los imprevistos que puedan llegar a suceder.

- **Comunicación entre el estudiante y el tutor**

  Se garantizará que la comunicación sea eficiente y sencilla, a la par que cualquier duda o mensaje se responda lo antes posible (por ambas partes).

A.5. Riesgos y prevención

Es común que en el desarrollo del proyecto haya riesgos o inconvenientes que lleguen a cambiar totalmente el rumbo del proyecto. Debido a esto, es muy importante reconocer

los riesgos antes de empezar el proyecto y tomar las medidas necesarias para afrontarlos. A continuación se muestran los riesgos principales, pero se ha de tener en cuenta que probablemente haya algún riesgo más que no se haya documentado.

A.5.1. Riesgos

- Uno de los riesgos más grandes del proyecto ha sido el tiempo, ya que debido al *Erasmus* en Italia y el no saber con que universidad debía de ser desarrollado el proyecto, el comienzo del mismo ha sido realmente tardío (principios de abril).
- En proyectos de este calibre es muy común que haya grandes desviaciones respecto a la planificación principal.
- Durante el desarrollo del proyecto puede darse la pérdida de datos realmente necesarios como el código, las tablas de resultados, las imágenes...
- Debido a que en el proyecto se han utilizado métodos de aprendizaje profundo y se han realizado múltiples pruebas con ellos, se ha empleado mucho tiempo para el entrenamiento de los mismos. Además, hay que tener en cuenta que los recursos en cuanto a procesador y memoria son limitados (sea localmente o sea en *Google Colaboratory*).

A.5.2. Prevención

Para afrontar los riesgos previamente mencionados y los inconvenientes que puedan llegar a suceder, se ha creado el siguiente plan de prevención:

- Se ha acelerado, en la mayor medida posible, el ritmo de trabajo. Además, para que el desarrollo del proyecto fuera constante y continuo, la interacción entre el estudiante y el tutor ha sido muy activa.
- A la hora de crear la planificación se ha tenido en cuenta el riesgo de que se produjese alguna desviación. Por ello, se ha creado una planificación flexible, con el objetivo de poder modificarlo más fácilmente.
- Para evitar la pérdida de los ficheros, además de guardar todos los archivos localmente, también se han guardado en la nube; de esta manera, si algún problema con el ordenador tuviese lugar, los ficheros seguirían estando disponibles.

- Con el fin de sacar mayor rendimiento al tiempo de espera (mientras los modelos estaban siendo entrenados), se ha invertido dicho tiempo en adelantar el trabajo de otras secciones.

Bibliografía

[Bottou, 2010] Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. *COMPSTAT*, pages 177–186.

[Breiman, 1996] Breiman, L. (1996). Bagging Predictors. *Machine Learning*, pages 123–140.

[Breiman, 2001] Breiman, L. (2001). Random Forests. *Machine Learning*, pages 5–32.

[Cleeremans et al., 1989] Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite State Automata and Simple Recurrent Networks. *Neural Computation*, pages 372–381.

[Demir et al., 2018a] Demir, I., Hughes, F., Raj, A., Tsourides, K., Ravichandran, D., Murthy, S., Dhruv, K., Garg, S., Malhotra, J., Doo, B., Kermani, G., and Raskar, R. (2018a). Generative Street Addresses from Satellite Imagery. *ISPRS International Journal of Geo-Information*.

[Demir et al., 2018b] Demir, I., Koperski, K., Lindenbaum, D., Pang, G., Huang, J., Basu, S., Hughes, F., Tuia, D., and Raskar, R. (2018b). DeepGlobe 2018: A Challenge to Parse the Earth through Satellite Images. *cs.CV*.

[Fukushima, 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, pages 193–202.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arVix*.

[Jaccard, 1901] Jaccard, P. (1901). Distribution de la Flore Alpine dans le Bassin des Dranses et dans quelques régions voisines. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, pages 241–272.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. L. (2015). Adam: A Method for Stochastic Optimization. *cs.LG*.

[Máttyus et al., 2017] Máttyus, G., Luo, W., and Urtasun, R. (2017). DeepRoadMapper: Extracting Road Topology from Aerial Images. *IEEE*.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of ICML*, pages 807–814.

[Ronneberger et al., 2015] Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *cs.CV*.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 386–408.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, pages 323–533.

[Schapire, 1990] Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, pages 197–227.

[Serra, 1983] Serra, J. (1983). *Image Analysis and Mathematical Morphology*. Academic Press, Inc.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, pages 1929–1958.

[Vapnik and Lerner, 1963] Vapnik, V. and Lerner, A. (1963). Pattern Recognition Using Generalized Portrait Method. *Automation and Remote Control*, pages 774–780.

[Zhou et al., 2018] Zhou, L., Zhang, C., and Wu, M. (2018). D-linknet: Linknet with pretrained encoder and dilated convolution for high resolution satellite imagery road extraction. *IEEE*.