UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Máster Oficial en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Trabajo Final de Master

Aplicación de Fully Convolutional Neural Networks para Segmentación Semántica de imágenes utilizadas para aplicaciones de Conducción Autónoma

Daniel Hormigo Ruiz

Tutores: Dr. Cayetano Guerra Artal

Fecha

Dedicado a mis padres, sin los que este trabajo nunca habría podido ser realidad. A los doctorandos del Lab5 por sus constantes enseñanzas, y a Cayetano Guerra, por inspirarme a trabajar en el terreno de la Inteligencia Artificial.

Gracias a todos por guiarme y brindarme todo un mundo de posibilidades.

Agradecimientos

Quiero agradecer a Cayetano Guerra Artal su ayuda, motivación e inspiración durante todo el proceso de este Trabajo.

Gracias a los profesores del máster. Me han enseñado un nuevo mundo antes desconocido.

Gracias a los doctorandos del Lab5 por sus constantes ayudas, ofrecimientos y discusiones que llevan a la reflexión y al entendimiento.

Gracias a mi familia y amigos por estar siempre presentes y ser un remanso de paz incluso en los peores momentos.

Resumen

La localización de la carretera dentro de una imagen para aplicaciones de conducción autónoma es la tarea principal para poder definir un espacio de búsqueda de señales de marcas viales preciso y fiable.

La segmentación semántica de imágenes es una herramienta que permite categorizar píxel a píxel toda una imagen, otorgando una información sin precedentes para los sistemas de visión por computador.

Para la realización de la segmentación semántica de imágenes es necesaria la implantación de un algoritmo que categorice los píxeles de la imagen de manera precisa y fiable. En este Trabajo presentamos un sistema de visión por computador basado en técnicas de *Deep Learning* que permitirá categorizar los píxeles de una imagen para poder posicionar la carretera dentro de la misma.

Índice general

Índice de figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Motivación

La conducción es una actividad que el ser humano empezó a realizar usando animales como medio de transporte hasta que el vehículo fue inventado. Desde entonces, la forma de conducir no ha cambiado: un ser humano sentado frente a un volante y unos pedales mecánicos que permiten acelerar o frenar el vehículo. Durante el S. XX el ser humano empezó a automatizar procesos industriales, pero la conducción siguió siendo manual. Sin embargo, hoy, en pleno S. XXI, la Era de la Información, el ser humano ha empezado a automatizar el proceso de la conducción, pues es la principal actividad que el ser humano realiza casi diariamente: conducir para realizar actividades rutinarias.

Unos de los principales campos de la conducción autónoma es la visión por computador, que dota al sistema de visión del entorno: marcas viales, señales de tráfico, peatones, y un sinfín de otros objetos que son de vital importancia a la hora de tomar decisiones cuando conducimos. Dentro de la actividad de conducir, la operación más crítica es mantener al vehículo dentro de su carril, así como detectar posibles señales de tráfico que puedan estar en la carretera y no como señales verticales.

En este Trabajo Fin de Máster se aplicarán técnicas avanzadas de visión por computador para la segmentación de imágenes, para determinar en qué lugar de la imagen se encuentra la carretera, ya que es vital determinar su localización para poder definir un espacio de búsqueda preciso y se pueda determinar la localización de marcas viales; es decir, determinar las líneas que delimitan el carril sobre el que se encuentra el vehículo, la existencia

de una señal de stop horizontal, etc, con el fin de aumentar la precisión de los sistemas que suceden a éste y aumentar la seguridad de los vehículos autónomos.

1.2. Objetivos

El objetivo de este proyecto es segmentar una imagen semánticamente para determinar la localización de la carretera en la misma. Para ello, será necesario estudiar las diferentes técnicas disponibles en el ámbito de la Inteligencia Artificial y determinar cuál de ellas es la idónea para segmentar semánticamente la escena.

De esta manera, los objetivos secundarios serían la obtención de un dataset de imágenes representativas del problema a estudiar, el modelo preentrando VGG-16, entrenamiento de diferentes arquitecturas de clasificadores y estudiar su precisión.

1.3. Redes Neuronales: matemática y tipos

En este capítulo se abordará qué son las redes neuronales, la matemática asociada y se describirán las diferentes arquitecturas que entrarán en juego a lo largo de este Trabajo Fin de Máster.

Inicialmente se describe la recta como el clasificador lineal más básico y, a continuación, se describe la recta como una operación matricial. Finalmente, se explica el modelo biológico de una neurona y la conexión existente entre la neurona y la recta descrita como una operación matricial, lo que dará lugar al nacimiento de las Redes Neuronales.

1.3.1. La recta como clasificador más básico

Previa explicación y exposición de qué son las Redes Neuronales, es necesario explicar las ideas básicas que dieron lugar a su creación. Supóngase un conjunto de datos cuya distribución es como en la Figura 1.1 izquierda. Sabiéndose que los datos se corresponden a: azul - clase 1, y rojo - clase 2; entonces, ¿cuál es la forma geométrica más sencilla que se puede construir y que permite separar ambos conjuntos de manera que se puedan clasificar fácilmente? La respuesta correcta es una recta (Figura 1.1 derecha), ya que

una recta que dividiese el espacio en dos subespacios permitiría alojar y diferenciar cada uno de ellos, de manera que se podría realizar una clasificación fácilmente.

$$Ax + By + C = 0 \tag{1.1}$$

$$y = mx + n \tag{1.2}$$

La recta se puede describir matemáticamente siguiendo la Ecuación General de la Recta (Ec. 1.1), o en su forma más conocida representada en la Ec. 1.2, donde m es la pendiente, o gradiente, de la recta y n la intersección con el eje y. Siguiendo estas ecuaciones, se observa que la recta permite dividir un único espacio es tres subespacios diferentes, como se puede observar en la Figura 1.2, de manera que estos son:

- **Subespacio 1:** donde $Ax + By + C = 0$,
- **Subespacio 2:** donde $Ax + By + C > 0$, y
- **Subespacio 3:** donde $Ax + By + C < 0$.

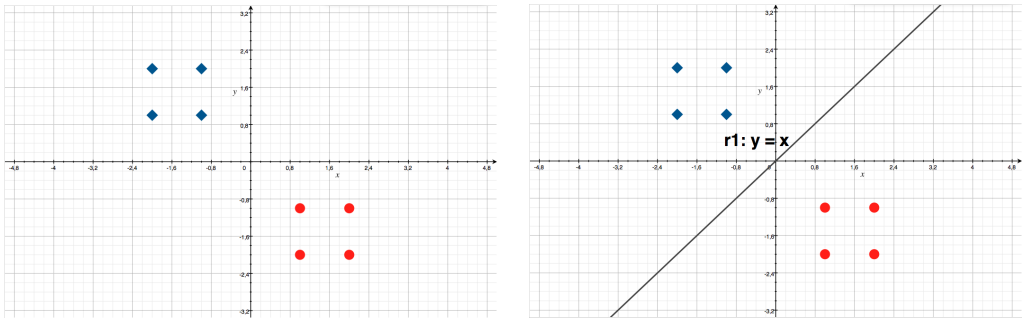

Figura 1.1: **Izquierda:** Conjunto de datos. **Derecha:** División del espacio mediante una recta.

Por lo tanto, la clasificación de un conjunto de datos se realizaría determinando en qué subespacio se encuentran. Es decir, se reduce a calcular una suma y determinar si es mayor o menor a 0, de manera que si el resultado es mayor a 0 se clasifica el dato como de la clase '1', y, en caso contrario, de la clase '2' (Ec. 1.3).

$$clase = \begin{cases} 1 & \text{si } Ax + By + C > 0 \\ 2 & \text{si } Ax + By + C < 0 \end{cases} \tag{1.3}$$

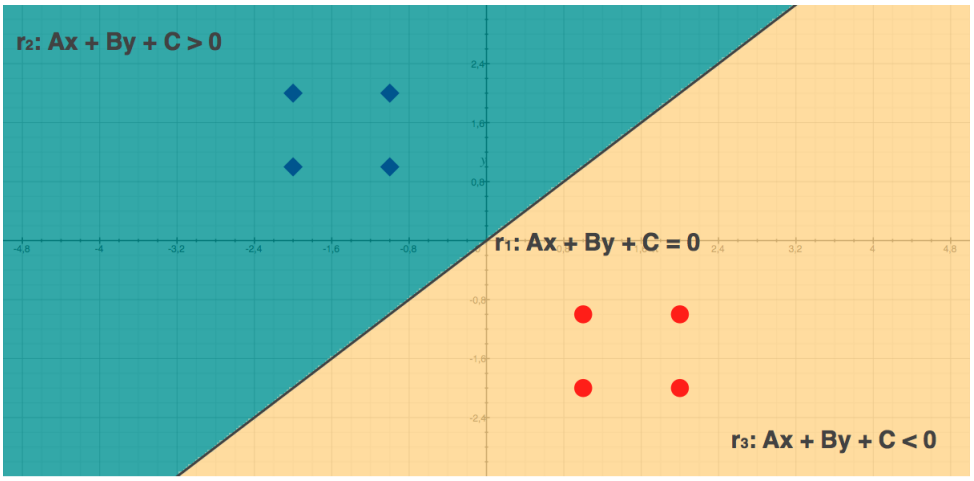

Figura 1.2: Subespacios generados mediante una recta

No obstante, para explicar la relación existente entre la recta y las Redes Neuronales, es necesario formular la recta como una operación matricial. Esta representación queda formulada a través de la Ec. 1.4,

$$W^T x + b = 0 \tag{1.4}$$

donde:

- W es la matriz que contiene los valores de A y B (Ec. 1.5).

$$W = \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \tag{1.5}$$

- x es la matriz que contiene los valores de x e y (Ec. 1.6).

$$x = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{1.6}$$

- b es la matriz que contiene el valor C (Ec. 1.7).

$$b = \begin{bmatrix} C \end{bmatrix} = \begin{bmatrix} b_1 \end{bmatrix} \tag{1.7}$$

Y, finalmente, su representación gráfica mediante un esquema de bloques está representado en la Figura 1.3, donde:

- x_i representan las entradas del sistema,

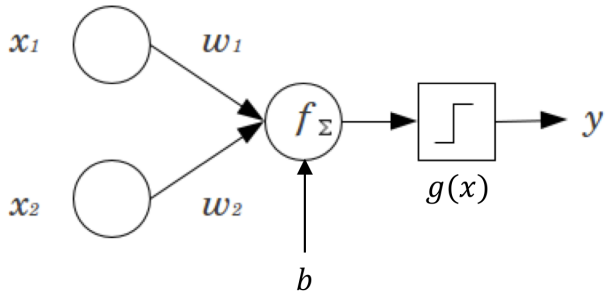

Figura 1.3: Esquema de bloques de una recta como clasificador

- w_i representa los pesos del sistema,
- b representa el *offset* de activación,
- $f_{\sum}$ representa el módulo de cómputo de la Ec. 1.4, y
- $g(x)$ representa la función signo, cuya salida es negativa si el valor de x es negativo y positiva en caso contrario.

Hasta este punto, se ha explicado el modelo matemático que describe la recta como una operación matricial. A continuación, se detallará el modelo biológico de una neurona y su conexión con el modelo matemático descrito a lo largo de la presente sección.

1.3.2. Modelo biológico y virtual de una neurona

En el sentido biológico, una neurona es el elemento computacional más básico del cerebro. La parte izquierda de la Figura 1.4 representa el esquema biológico de una neurona: cada neurona recibe una señal de entrada por sus dendritas (entradas del sistema), procesa la información recibida en su núcleo (núcleo de proceso) y produce una señal de salida por su axón (salida del sistema). Sin embargo, las neuronas no están aisladas en el cerebro, sino que están unidas mediante sinapsis a otras neuronas, de manera que la información se procesa y se propaga, y además, no todas las sinapsis son igual de fuertes, sino que unas lo son más que otras de manera que el cerebro define qué entrada de información es más importante. La unión de unidades de procesamiento permite que la información de las capas más externas sea información básica y simple, pues la información que procesan procede directamente del entorno, mientras que en las capas más profundas la información es más abstracta y compleja, ya que reciben información preprocesada. En

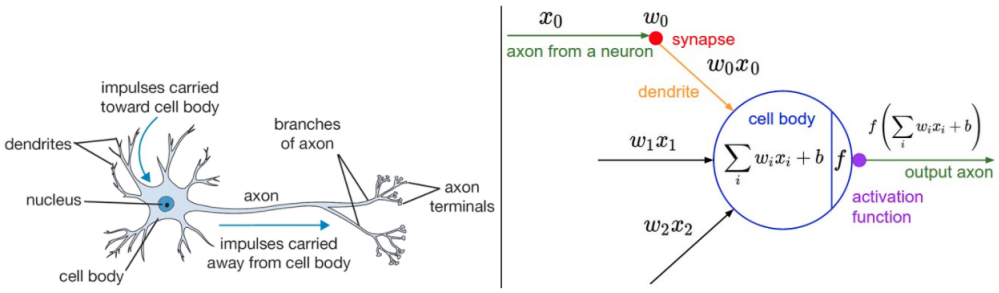

Figura 1.4: Modelo biológico de una neurona

otras palabras, las neuronas de las capas más externas se activarán ante patrones de información poco complejos, mientras que las más profundas lo harán ante patrones complejos, patrones que incluso el ser humano no es capaz de detectar a simple vista.

En cuanto al modelo virtual, éste debe reflejar el mismo esquema presentado anteriormente, y es el que se puede observar en la parte derecha de la Figura 1.4: las entradas al sistema son las denominadas x_i, y pueden ser desde píxeles de una imagen hasta datos numéricos para regresiones lineales, las sinapsis son descritas como ponderaciones de las entradas, y comúnmente son denominadas pesos (w_i), un *offset* b_i que determinará si la neurona se activa o no dada una serie de entradas, y finalmente el núcleo está constituido por una función de activación f cuyo resultado es la salida enviada a través del axón (y) [1].

Como se puede observar, la Figura 1.4 derecha coincide exactamente con el esquema de bloques representado en la Figura 1.3, lo que permite concluir que una neurona computa la Ec. 1.4.

En definitiva, lo que ha quedado plasmado a lo largo de estos párrafos es que una neurona, concebida como unidad simple de cómputo, actúa como un clasificador, de manera que si la operación de cómputo (Ec. 1.4) es mayor a un *offset*, o *bias*, la neurona se activará. No obstante, el uso de una única neurona como clasificador es un caso casi improbable ya que, habitualmente, la distribución de los datos requiere de formas geométricas más complejas para separarlos en diferentes conjuntos clasificables. Y es por este motivo por el que hablamos de Redes Neuronales: sistemas básicos de cómputo lineal interconectados que generan formas geométricas no lineales que permiten modificar el espacio de representación, y convertirlo en un espacio fácilmente clasificable. Un ejemplo de lo aquí mencionado es la Figura 1.5, donde el

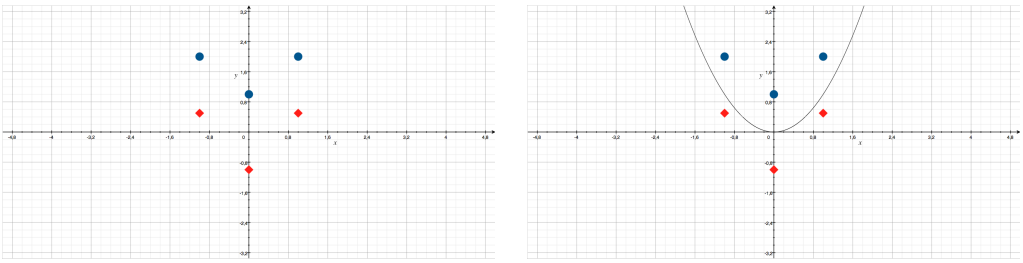

Figura 1.5: **Izquierda:** Conjunto de datos. **Derecha:** División del espacio mediante una parábola.

patrón que sigue el conjunto de datos es un polinomio de segundo grado, por lo que sería necesario crear una red de neuronas que permitiese generar un patrón de clasificación similar.

1.3.3. Entrenamiento de una Red Neuronal

Hasta ahora se ha hablado de que una Red Neuronal clasifica, pero la pregunta más importante es: ¿cómo aprende?¿Cómo debemos realizar su entrenamiento? A priori estas preguntas pueden parecer difíciles, pero en la realidad la respuesta es muy fácil y sigue el siguiente razonamiento: como se puede observar en la Figura 1.3, la red está compuesta por las entradas x_i, que son dadas al sistema y por lo tanto de valor fijo, y los pesos w_i y los bias b_i, que son valores que determinan cómo de importante es una entrada y cómo ha de ser el resultado de la operación de cómputo (Ec. 1.1) para que la neurona se active. Es decir, estos dos últimos parámetros son modificables, ajustables, de manera que son los parámetros que se modificarán a lo largo del entrenamiento, hasta alcanzar un conjunto de valores que maximicen la precisión del clasificador. Por lo tanto, es necesario definir un proceso matemático que refleje dicho objetivo. Este proceso se denomina *backpropagation*, y permite actualizar los pesos de la red, de manera que estos se dirijan hacia unos valores óptimos que permitan optimizar la tasa de acierto de clasificación [2].

Existen múltiples métodos ara el entrenamiento de una Red Neuronal, y de otros tipos de clasificadores, siendo los más usados [3]:

- **Resustitución:** consiste en utilizar todas las muestras del conjunto de datos tanto para entrenar, como para testear y validar. Es un método optimista, que ofertará un 100 % de acierto en los datos existentes, pero una nula tasa de generalización del modelo (1.6 a)).

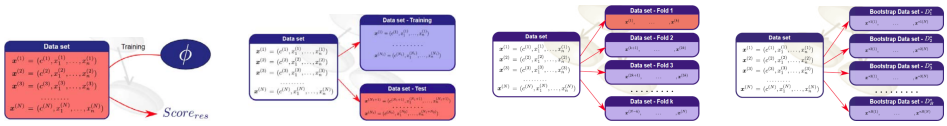

Figura 1.6: **a)** Resustitución. **b)** Holdout. **c)** Validación cruzada. **d)** Bootstraping.

- **Hould-Out:** consiste en dividir el conjunto de datos en dos, o tres, subconjuntos: entrenamiento, validación y test, y representan el 70 %, 15 % y 15 % del tamaño del conjunto de datos respectivamente. Genéricamente, la estimación obtenida es menor a la precisión real de clasificación (1.6 b)). .
- **Validación cruzada:** consiste en dividir el conjunto de datos en k subconjuntos, de manera que se usan $k-1$ datasets para entrenar, y el sobrante para testear, y se itera hasta obtener k resultados de clasificación. La tasa final se calcula como la media de los resultados obtenidos (1.6 c)).
- **Bootstraping:** se submuestrea el conjunto de datos con reemplazo, de manera que se obtienen n datasets de entrenamiento y 1 de test, y se obtiene la tasa de acierto de cada uno de los n modelos generados usando el mismo conjunto de test para todos ellos, siendo la tasa de acierto la tasa promedio de las n tasas de acierto (1.6 d)).

El proceso de entrenamiento viene definido por los siguientes pasos:

1. Definir la técnica de entrenamiento a usar, y dividir el conjunto de datos en función a las características del método.
2. Obtener de la clasificación dada una red en un estado j, denominada $\hat{y}$.
3. Calcular el error asociado a la clasificación deseada (y) respecto de la obtenida ($\hat{y}$). La función de pérdidas, o función de coste, no es una función definida, sino que se define según el problema a tratar.
4. Actualizar los pesos usando la técnica de *backpropagation*. El *backpropagation* es una técnica que permite actualizar de manera iterativa los pesos de la red a través de la búsqueda del mínimo de la función de coste f definida, siendo el valor actualizado el valor anterior más un pequeño incremento en la dirección del mínimo (Ec. 1.8), siendo ΔW_{t-1}

definida en la Ec. 1.9 [4, 5, 6]. En ella se observa la existencia de varios parámetros importantes:

- α**:** es el *learning rate* de la red. Este parámetro determina cómo de rápido aprende la red, llegando a ser uno de los parámetros más importantes dentro de la misma, ya que un *learning rate* demasiado bajo haría que la red generalizase correctamente el modelo pero el proceso de aprendizaje sería muy lento, en el caso de un *learning rate* alto la red aprendería rápido pero quedaría estancada en unos valores de pesos no óptimos, y en caso de ser muy alto podría hacer que la red se ajustase a su set de entrenamiento y no generalizase el modelo (Figura 1.7). A este último caso se le denomina *overfitting* [7].
- x_i**:** es el valor actual de las entradas a la red, procedentes del set de entrenamiento.
- o_j**:** es la clasificación obtenida dada la entrada i para la clase j.
- t_j**:** es la clasificación objetivo, la clasificación real de la entrada i para la clase j.
- f**:** es la función de pérdidas descrita.
- w_{ij}**:** es el conjunto de pesos dada una entrada i asociados a una clase j.

5. Repetir hasta conseguir la tasa de acierto objetivo o alcanzar el número de iteraciones deseadas.

$$W_t = W_{t-1} + \Delta W_{t-1} \tag{1.8}$$

$$\Delta W_{t-1} = \alpha * [x_i * (o_j - t_j) * o_j * (1 - o_j)] = \alpha * \frac{\partial f}{\partial w_{ij}} \tag{1.9}$$

Una vez entendido el proceso de entrenamiento, lo siguiente es definir los tipos de redes existentes para clasificación en imágenes así como sus características.

1.4. Tipos de Redes Neuronales para clasificación

Llegados a este punto, el lector ha adquirido una noción básica sobre lo que es una neurona, su función, cómo funcionan y cómo se entrenan, pero

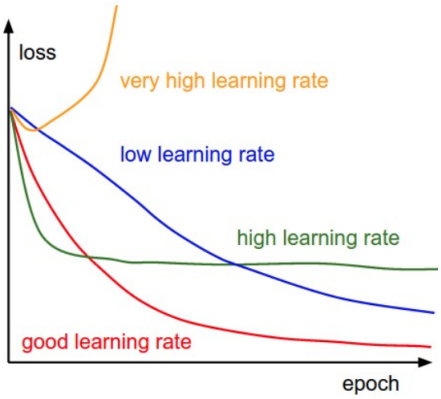

Figura 1.7: Efecto del *learning rate* sobre el aprendizaje de la red

no se ha descrito cómo se organizan dentro de una red, así como qué tipo de redes existen. En los siguientes puntos se detallarán las arquitecturas más conocidas, así como sus principales ventajas y desventajas, de manera que el lector pueda discernir sus diferencias y aplicaciones.

1.4.1. Arquitectura de las Redes Neuronales

Antes de iniciar la descripción de las arquitecturas más conocidas, es necesario clarificar la nomenclatura que se usará de aquí en adelante. Dada la red de la Figura 1.8, se distinguen tres partes, o capas de aquí en adelante, que son:

- **Capa de entrada:** es la capa conectada a las entradas del sistema. En caso de ser una imagen, cada entrada sería un píxel de la imagen.
- **Capa oculta:** es, o son en caso de ser más de una, la capa intermedia; la capa entre la entrada y la salida. Esta capa es la que permite dotar de abstracción a la red, y permitir la clasificación de conjuntos de datos que a priori no podrían ser separables.
- **Capa de salida:** es la última capa de la red, y se corresponde con la capa de clasificación. En ella se computa la clasificación de la entrada en una, o varias, de las n clases disponibles.

Además, cada neurona de la red tiene una función de activación, que permite determinar si una neurona se activa o no dependiendo de sus entradas y su bias asociado. Esta activación, idealmente, debería ser 0 cuando la neurona no se activa y 1 cuando sí; es decir, una función escalón (Figura 1.9 a), Ec.

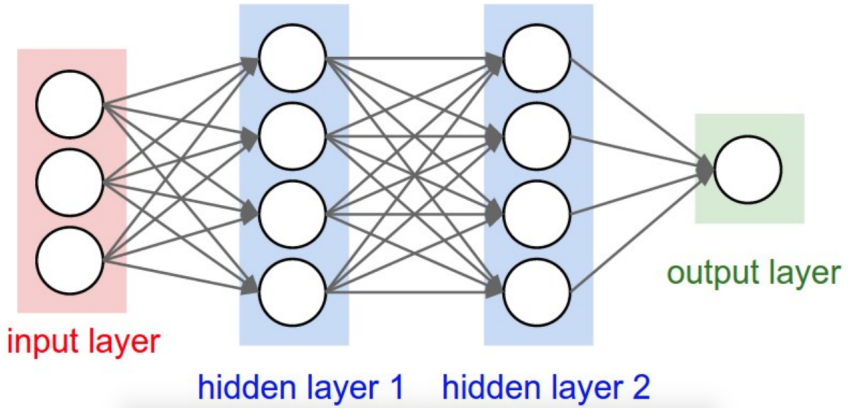

Figura 1.8: Capas de una Red Neuronal

1.10). No obstante, este tipo de funciones tienen gradiente infinito en el punto (0, 0), ya que el ángulo de la pendiente es 90º. Para solventar este problema, se definieron funciones que emulasen dicho comportamiento, pero con una transición más suave que la de una función escalón, de manera que aportan gradiente a la función y permiten el entrenamiento de la red. Las más conocidas son ReLu (Figura 1.9 b), Ec. 1.11), Sigmoide (Figura 1.9 c), Ec. 1.12) y Tanh (Figura 1.9 d), Ec. 1.13), además de la función softmax para clasificar en la última capa (Ec. 1.14) [8]. La función de activación comúnmente usada es la ReLu, ya que otorga un mayor gradiente a la función, y no lo desvanece, como sí pasa con la función sigmoide.

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases} \tag{1.10}$$

$$f(x) = \text{máx}(0, x) \tag{1.11}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.12}$$

$$f(x) = \tanh(x) \tag{1.13}$$

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \text{ for } j = 1, ..., K \tag{1.14}$$

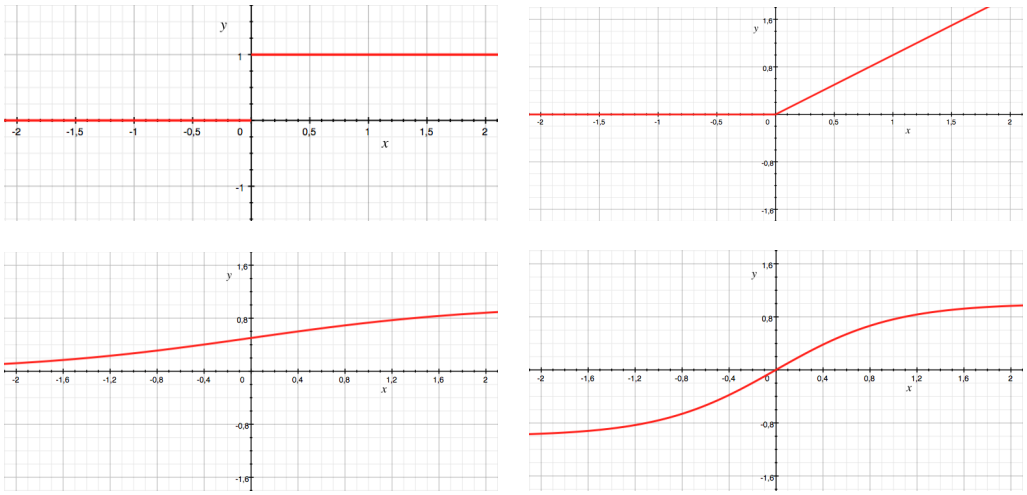

Figura 1.9: Función de activación **a)** Escalón. **b)** ReLu. **c)** Sigmoid. **d)** Tanh.

1.4.2. El Perceptrón

El perceptrón es la arquitectura que comúnmente es denominada Red Neuronal, ya que su estructura está constituida única y exclusivamente por neuronas. Su origen data de los años 50 aproximadamente por Minsky y Papert, y nació de la idea de dotar de inteligencia a las máquinas. La idea era entrenar a las máquinas de manera que pudieran aprender a reconocer y clasificar emulando la forma en la que los seres humanos lo hacen. Es decir, mediante descubrimiento de patrones siguiendo el proceso de prueba y error, siendo este aprendizaje un aprendizaje supervisado. Los seres humanos tienen dos formas de aprender: aprendizaje supervisado, donde otro ser humano determina si la acción tomada es correcta o no y se lo hace saber al humano que la ha llevado a cabo (un ejemplo es cuando un niño aprende a distinguir los colores), y no supervisado, donde son las recompensas asociadas a acciones las que permiten determinar si la acción, o conjunto de acciones, tomada es correcta o no (un ejemplo es cuando un niño está aprendiendo a caminar). En el caso propuesto por Minsky y Papert, el perceptrón, el aprendizaje es supervisado; es decir, se entrenará la red usando datos de entrada con una etiqueta asociada, de manera que al final de cada clasificación se calcula el error obtenido y se actualizan los pesos de la red (ver sección 1.3.3). La estructura típica de un perceptrón es la presente en la Figura 1.8, denominándose perceptrón multicapa a aquel que posee una o más capas ocultas, y, en caso contrario, perceptrón monocapa.

Este tipo de redes tienen la ventaja de que permiten crear hiperplanos

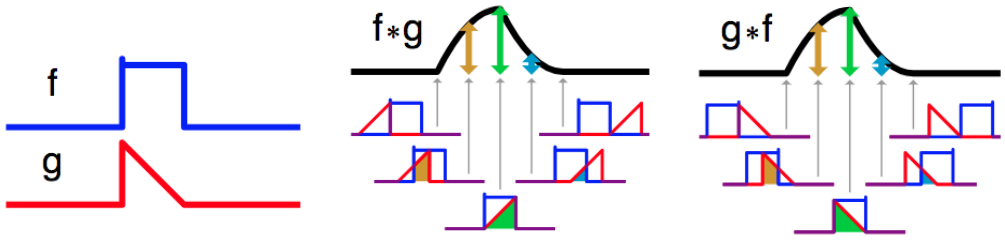

Figura 1.10: Proceso de convolución. **a)** Señales a convolucionar. **b)** Convolución de f y g. **c)** Convolución de g y f.

que son capaces de seccionar el conjunto de datos y clasificarlos fácilmente, pero no son las óptimas para la extracción de características de imágenes que permitan su clasificación. Es decir, son redes cuya finalidad es clasificar los datos, y no extraer características de ellos. Es por ello que, en aras a mejorar la tasa de acierto de clasificación de imágenes, se inventaron las Redes Neuronales Convolutivas, que, como su propio nombre avanza, son redes constituidas por una Red Neuronal y una parte convolutiva.

1.4.3. Redes Neuronales Convolutivas

La convolución es una operación matemática definida por la Ec. 1.15, que transforma dos funciones f y g en una tercera función h que refleja la magnitud en la que se suponen f y una versión trasladada e invertida de g (Figura 1.10) [9, 10].

$$h(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \tag{1.15}$$

Aunque la definición matemática es la descrita en el párrafo anterior, existen muchas otras formas de explicar la convolución: desde el punto de vista de la Teoría de la Señal es una herramienta que permite describir a un sistema lineal, en el campo de la acústica permite definir un eco como la convolución del sonido original con una función que represente los objetos variados que lo reflejan, etc. En el campo que concierne a este Trabajo, las imágenes, la convolución se define como una operación a realizar sobre una imagen para obtener un efecto deseado. Es decir, la convolución se usa para lograr efectos en la imagen de entrada, tales como enfatizar los bordes de la imagen, enfocar, etc.. La forma en la que se aplican los filtros en procesado digital de imagen es definiendo el filtro de convolución como una matriz de tamaño $n \times m$, de manera que se calcula el valor de un pixel central como la suma ponderada de k vecinos al mismo, denominándose *kernel* de con-

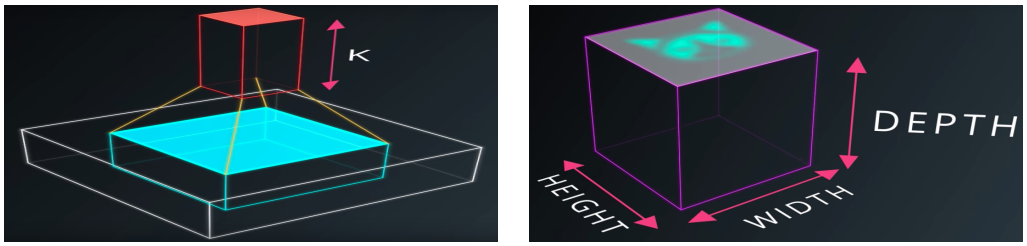

Figura 1.11: La convolución: **Izq.** Representación en formato FC de un kernel de convolución. **Der.** Imagen obtenida tras la convolución.

volución al filtro a aplicar [11]. Dada esta característica de la convolución para con las imágenes, se observa que su inclusión en las Redes Neuronales conlleva un gran potencial a la hora de descubrir patrones que definan a un objeto dentro de una imagen.

En las CNN, el kernel de convolución se añade como una capa más de la red. Es decir, se incluye como una capa más cuyos valores son modificables durante el proceso de entrenamiento, de manera que se modificarán hasta encontrar un conjunto de pesos óptimos que activen un conjunto de neuronas cuando un patrón ha sido descubierto dentro de la imagen. Además, a los *kernels* también se les considera neuronas dentro de la red, aunque son un tipo de neuronas especiales. En las FC las neuronas tenían pesos asociados a cada una de sus entradas, y cada neurona tenía su propio conjunto de pesos. Sin embargo, en las CNN esto no es así: cada *kernel* se considera como una neurona en sí que será aplicada a toda la imagen, de manera que cada *kernel* tiene asociados un conjunto de pesos que serán compartidos. Para una visualización y mejor entendimiento de lo mencionado anteriormente, lo ideal es imaginar el *kernel* como una red FC de K salidas que se aplicará a lo largo de toda la imagen, tal y como se representa en la Figura 1.11 izquierda. Tras su aplicación sobre la imagen, se obtiene una nueva imagen de tamaño más reducido a la original, pero con una profundidad K que será mayor a la original, que se puede entender como una imagen de K colores, como se representa en la Figura 1.11 derecha. De esta manera, capa tras capa la complejidad de la información con la que trabajan las capas más profundas la red es mayor, debido a que el mapa de características se aleja de la información del entorno y pasa a ser una información ya procesada [12, 13].

Para el correcto uso y entendimiento de las CNN, es necesario explicar algunos términos:

- ***Stride:*** es el número de píxeles que se desplaza la ventana de convolución entre operación y operación. Por ejemplo, un *stride* de 1 genera una imagen de salida del mismo tamaño que la imagen de entrada, mientras que uno de *stride* 2 genera una imagen de tamaño la mitad.

- ***Padding:*** tiene como objetivo modificar o no el tamaño de la imagen de salida.

  - **Same:** tiene como objetivo generar una salida de la mismas dimensiones que la entrada, y para ello rellena con ceros en los bordes (Figura 1.12 izquierda).
  - **Valid:** no realiza el procesado anteriormente descrito, y por tanto las dimensiones de la salida pueden verse reducidas respecto a las de la entrada (Figura 1.12 derecha) [14].

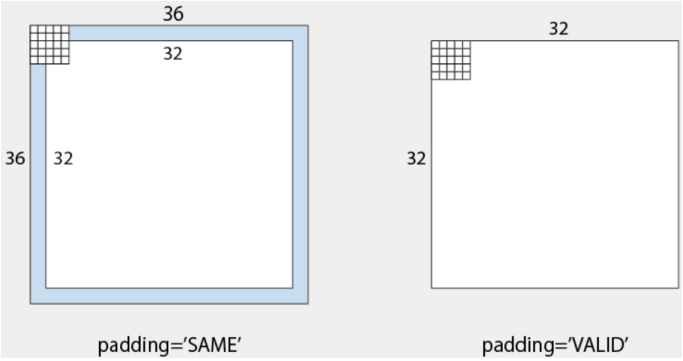

Figura 1.12: Efecto de la aplicación de los diferentes padding disponibles

- **Mapa de características:** el término mapa de características hace referencia al vector resultante de la aplicación de una capa convolutiva a la imagen. En el caso explicado anteriormente, el mapa de características iniciales de una imagen RGB de tamaño 512×512 es de tamaño $512 \times 512 \times 3$, y tras la aplicación del *kernel* de dimensiones $2 \times 2 \times K$ es de tamaño $256 \times 256 \times K$. Como se puede observar, las CNN generan mapas de características sucesivos en forma de pirámide, cuya información aumenta en complejidad (Figura 1.13).

No obstante, las CNN pueden necesitar una capa de *pooling* cuyo objetivo es disminuir el tamaño del mapa de características. Estas capas suceden a la capa convolutiva y computan el máximo o media de una celda de tamaño $p \times q$ dada una matriz de tamaño $n \times m$, de manera que se obtiene una matriz de salida de tamaño $\frac{n}{p} \times \frac{m}{q}$ cuyos elementos son el máximo o media

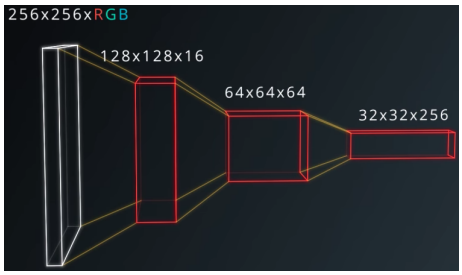

Figura 1.13: Mapas de características resultantes.

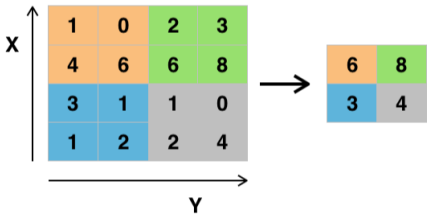

Figura 1.14: Ejemplo de aplicación de max pooling 2×2 a una matriz 4×4

de las celdas [15] (Figura 1.14). Además, como su propio nombre indica, las CNN están compuestas también por una FC, por lo que el clasificador estaría compuesto por: una parte convolutiva, encargada de extraer el mapa de características necesario para la clasificación de las imágenes, y la FC, encargada de clasificar. La representación de la arquitectura típica de una CNN es la Figura 1.15, en la que fácilmente se puede distinguir las capas convolutivas, de *pooling* y FC, constituyentes todas ellas de la CNN.

En cuanto a la mejora introducida por este tipo de arquitectura, un ejemplo claro de su clara mejora sobre las FC es el trabajo presentado por Yann LeCun, Corinna Cortes y Christopher J.C. Burges, donde reflejan la precisión obtenida clasificando el dataset MNIST (conjunto de imágenes de números escritos a mano) usando diferentes métodos. En él, se refleja que la tasa de

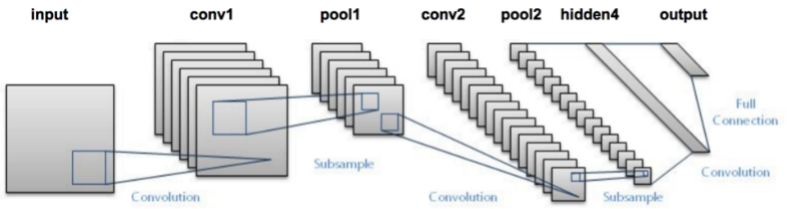

Figura 1.15: Arquitectura de una CNN.

error para una red FC de 2 capas con 300 unidades ocultas es de 4.7 %, mientras que con una red CNN de 3 capas ésta es de 1.7 % [16]. Por lo tanto, este trabajo concluye que las CNN permiten una mejor clasificación cuando de imágenes se trata, y es por ello por lo que es la arquitectura base usada en este Trabajo Fin de Máster.

Capítulo 2

Estado del arte

En este capítulo se describirá el estado del arte en la definición de espacios de búsqueda de carreteras en imágenes usadas en aplicaciones de conducción autónoma, y está dividido de manera que al comienzo se detallará el método de selección de espacios de búsquedas manuales, para finalmente definir las técnicas más modernas que han permitido la automatización de este proceso.

2.1. Definición del espacio de búsqueda manualmente

Esta técnica se basa en determinar una región de interés usando técnicas heurísticas, tales como suponer dónde se encontraran normalmente los objetos a buscar dentro de una imagen y recortar la imagen en esas zonas. Esta suposición solamente es válida cuando la cámara está sujeta en una posición fija e invariante, de manera que, por ejemplo, la carretera siempre esté en la misma zona de la imagen.

La desventaja de esta técnica reside en la suposición de que la imagen se encontrará siempre centrada respecto del mismo punto de referencia, y descartar zonas de la imagen que pudieran ser de interés. Su principal ventaja es la que la complejidad algorítmica es mínima, por lo que su impacto sobre la velocidad de procesamiento del software realizado es mínimo. Un ejemplo de esta aplicación puede observarse en la Figura 2.1, donde en la imagen de la izquierda se ha definido manualmente la región de interés, y la imagen derecha es el resultado obtenido.

Figura 2.1: **Izq.:** Definición del espacio de búsqueda manualmente. **Der.:** Región obtenida.

2.2. Definición del espacio de búsqueda automáticamente

En aras de evitar suposiciones a la hora de diseñar software para una aplicación crítica como es la conducción autónoma, ya que siempre conllevan cierto riesgo e inexactitud, se desarrollaron técnicas de definición automática de espacios. La principal ventaja de estos métodos es que definen las zonas de búsqueda teniendo en cuenta elementos que deben encontrarse dentro de la imagen para su definición, y su principal desventaja es que ocupan tiempo de cómputo y por lo tanto afectan a la velocidad de procesamiento.

Dentro de la búsqueda de la carretera dentro de una imagen, existen dos métodos: definición del espacio de búsqueda mediante la búsqueda y definición de las líneas que definen la carretera, y la segmentación semántica de escenas, técnica innovadora dentro de la Visión por Computador y que será la utilizada en este Trabajo.

2.2.1. Mediante búsqueda de líneas

La definición del espacio de búsqueda de carreteras dentro de una imagen mediante la búsqueda de líneas, permite definir de manera fiable el espacio de búsqueda, ya que define la región teniendo en cuenta unos de los elementos principales que constituyen una carretera: las líneas que definen los carriles. Para ello es necesario la aplicación de técnicas de búsqueda de líneas, para después buscar los vértices de las líneas que son de interés y recortar la imagen en esa zona.

Existen diversas técnicas que permiten buscar líneas dentro de una imagen, siendo las más usadas las siguientes: el filtrado por color, ya que las líneas de una carretera suelen seguir el mismo patrón de colores, y mediante la aplicación de filtros que permitan filtrar aquellas zonas donde hay líneas.

2.2.1.1. Búsqueda de líneas mediante filtrado por colorimetría

Como ya es conocido, una imagen es una matriz n-dimensional constituida por píxeles, los cuales definen el color del mismo. La forma en la que el color del píxel es codificado se denomina espacio de color del píxel, existiendo múltiples y diferentes espacios de color (Figura 2.2), así como la posibilidad de cambiar de un espacio a otro, ya que dependiendo de la aplicación será más útil usar uno que otros [17, 18]. Dada esta característica de las imágenes, una forma de determinar la localización de un objeto en una imagen es realizando un filtrado mediante el color que defina a tal objeto [18]. En el caso que nos concierne, las líneas de una carretera siguen un patrón de color, siendo generalmente blancas, por lo que se podría definir dónde están filtrando la imagen por el color blanco. Un ejemplo de aplicación de esta técnica es la Figura 2.3, donde la imagen izquierda se observa que el color que define las líneas es blanca (RGB [200, 200, 200]) y la imagen derecha es la imagen filtrada usando el amarillo como filtro.

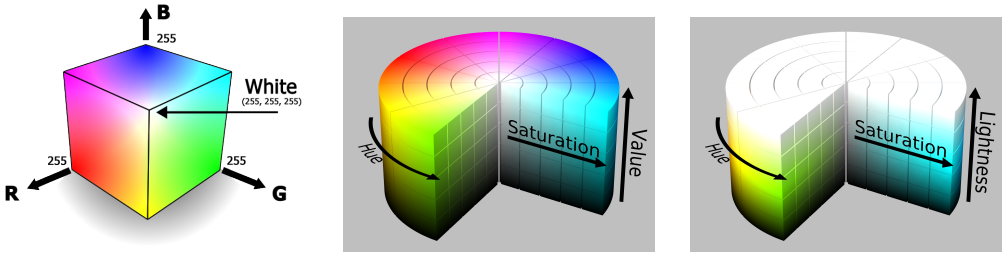

Figura 2.2: Espacio de color **Izq.:** RGB. **Centro:** HSV. **Der.:** HLS

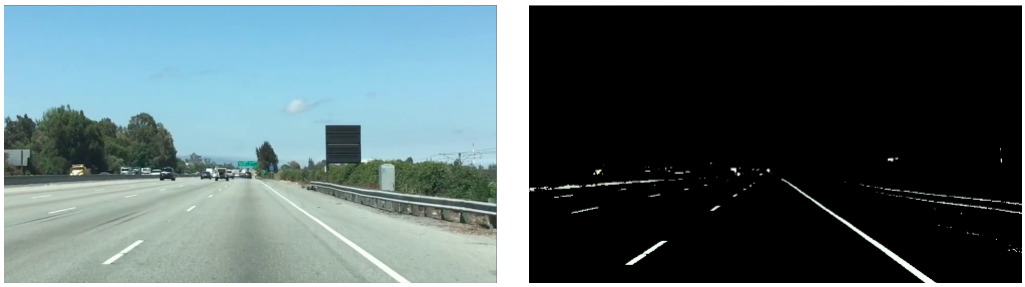

Figura 2.3: **Izq.:** Imagen de la carretera. **Der.:** Resultado tras filtrar la imagen por el color RGB [200, 200, 200].

No obstante, el color que define las líneas de una carretera no es el mismo en todos los países, por lo que el filtrado por color se antoja una herramienta

ineficiente para ser aplicada en el ámbito de la conducción autónoma, pues su aplicación no es global, siendo esta su principal y mayor desventaja.

2.2.1.2. Búsqueda de líneas mediante aplicación de filtros

Otra característica de las líneas en las imágenes es que son regiones en las que el color cambia a alta frecuencia. Un ejemplo gráfico de lo anteriormente mencionado, es la Figura 2.4, donde se puede observar la existencia de cambios en la intensidad de la imagen en aquellas regiones donde existen, o podrían existir, líneas. Entonces, determinar la posición en la que se dan esos cambios en la imagen es una herramienta útil para determinar la existencia de líneas en la carretera. Una herramienta que permite determinar dónde se encuentran los cambios en una función f, es la derivada, tal y como se puede apreciar en la Figura 2.5. Su forma analítica viene expresada por la Ec. 2.1.

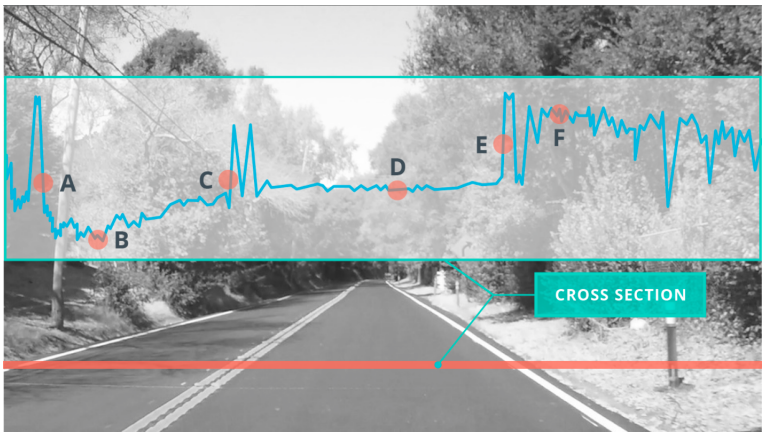

Figura 2.4: Gráfica de cambios en la intensidad de la imagen en zonas donde existen líneas

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} G_x & G_y \end{bmatrix} \tag{2.1}$$

No obstante, en procesamiento de imagen la derivada no existe como una función analítica a resolver, sino son matrices cuyos valores imitan el efecto de la derivada; es decir, son matrices que emulan una función que permite aproximar la derivada por la diferencia de dos píxeles contiguos de la imagen [19, 20]. Dado que los bordes pueden estar inclinados, ha de definirse dos matrices de convolución: uno para el eje x y otro para el eje y, siendo la matriz genérica la presenten en la Figura 2.6 a). Los operadores más utilizados son:

- **Roberts:** Obtiene buena respuesta ante bordes diagonales y ofrece buenas prestaciones en cuanto a localización. No obstante, es extre-

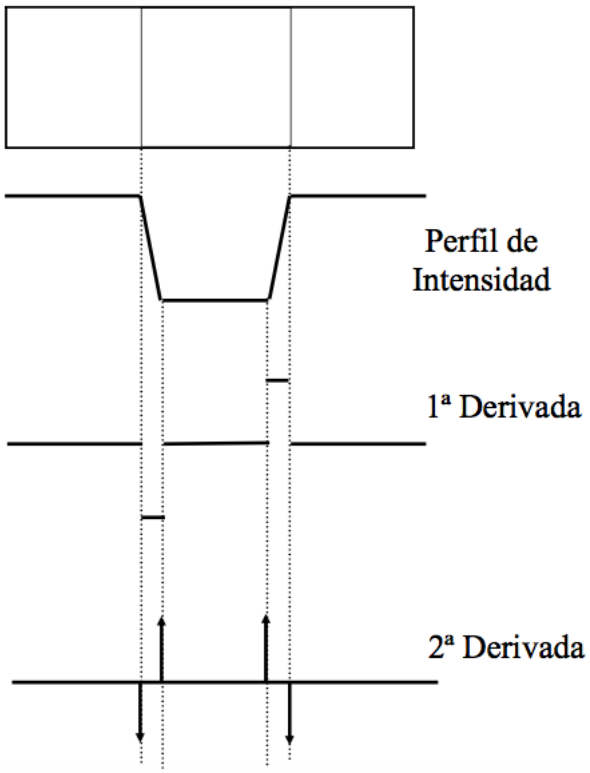

Figura 2.5: Detección de bordes usando operadores derivada.

madamente sensible al ruido y por tanto tiene pobres cualidades de detección Su matriz de convolución está definida en la Figura 2.6 b), y el resultado obtenido dada una imagen está descrito en la Figura 2.7 b).

- **Prewitt:** En el operador Prewitt se involucran a los vecinos de filas / columnas adyacentes para proporcionar mayor inmunidad al ruido y está basado en la idea de diferencia central. Su matriz de convolución está definida en la Figura 2.6 c), y el resultado obtenido dada una imagen está descrito en la Figura 2.7 c).

- **Sobel:** En el operador Sobel explota la misma idea que el de Prewitt, pero en este caso pondera más los píxeles centrales en el promediado, y en teoría es más sensible a los bordes diagonales que el de Prewitt, aunque en la práctica la diferencia no es significativa. Su matriz de convolución está definida en la Figura 2.6 d), y el resultado obtenido dada una imagen está descrito en la Figura 2.7 d).

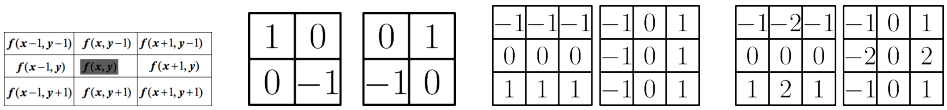

Figura 2.6: Máscaras de convolución **a)** genérica. **b)** de Roberts. **c)** de Prewitt. **d)** de Sobel.

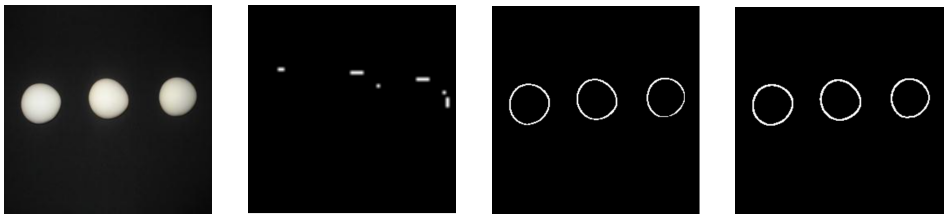

Figura 2.7: **a)** Imagen original. **b)** Bordes mediante Roberts. **c)** Bordes mediante Prewitt. **d)** Bordes mediante Sobel.

Además, debido a la definición de derivada, se puede obtener parámetros tales como la dirección y la magnitud del gradiente de la imagen, lo que aportaría un extra de información valiosa a la hora de filtrar la imagen [20]:

- **Dirección del gradiente:** permite filtrar aquellos bordes detectados en la imagen que no cumplen con la dirección deseada. En el caso de buscar líneas a 45º, este método ayudaría a eliminar todas aquellos bordes que no cumplan con dicho requisito.
- **Magnitud del gradiente:** permite filtrar aquellos bordes detectados en la imagen cuya intensidad de cambio está fuera de los umbrales deseados. Este método ayudaría a eliminar bordes detectados debidos a pequeños cambios de intensidad en la imagen y que no son parte de las líneas de la carretera.

Esta información, analíticamente, vendría descrita por las ecuaciones **??**, y su efecto en la detección de bordes puede contemplarse en la Figura 2.8, donde, como se puede observar, las líneas de la carretera quedan bien definidas tras la aplicación de los métodos anteriormente descritos con sus correspondientes umbrales de decisión.

$$|\nabla f\,| \approx |\nabla G_x\,| + |\nabla G_y\,| \tag{2.2}$$

$$\alpha(x, y) = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{2.3}$$

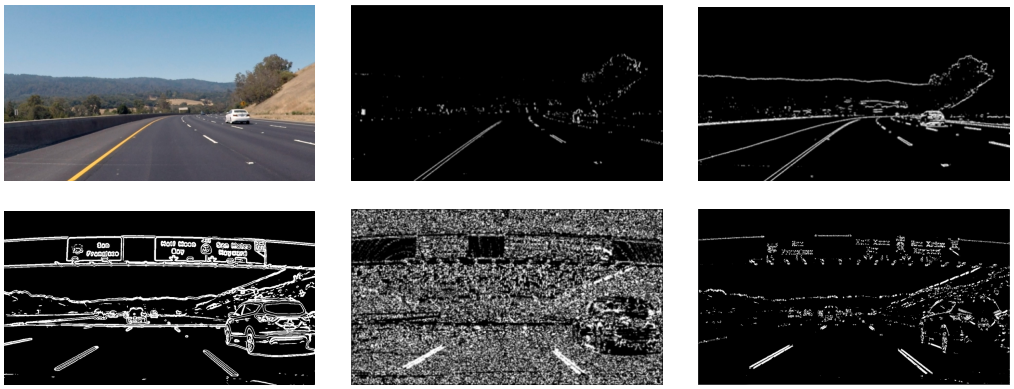

Figura 2.8: **a)** Imagen original. **b)** Bordes en eje x. **c)** Bordes en eje y. **d)** Bordes mediante cálculo de la magnitud del gradiente. **e)** Bordes mediante cálculo de la dirección del gradiente. **f)** Resultado final.

Tras aplicar la detección de bordes mediante operadores derivada, se antoja necesaria la aplicación de un método que obtenga la posición de las líneas dentro de la imagen, como pueden ser la Transformada de Hough [21] o transformación de perspectiva a vista de pájaro combinada con medición de histogramas para obtener la posición de las líneas [22]. Este último método es el que mayor precisión presenta hasta el momento, y permite obtener las líneas y su curvatura, tal y como se puede observar en la Figura 2.9.

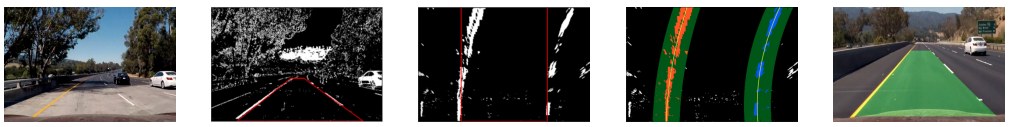

Figura 2.9: **a)** Imagen original. **b)** Imagen procesada y ROI **c)** Vista de pájaro de la ROI **d)** Líneas detectadas **e** Posición de la carretera

No obstante, necesita de la existencia de marcas viales para determinar la existencia de la carretera, lo cual es su principal desventaja. En aras a evitar la necesidad de la existencia de marcas viales que determinen dónde se encuentra la carretera, así como evitar umbrales que puedan ser válidos solo en algunas situaciones y no en todas, nacieron las técnicas de segmentación semántica de escenas. Estas técnicas permiten segmentar una imagen mediante la aplicación de técnicas de Inteligencia Artificial, mejorando en bastante los resultados obtenidos usando herramientas basadas en técnicas clásicas de procesado de imagen, como las anteriormente descritas.

A continuación, se presentará la técnica de segmentación semántica de escenas, que es la técnica en la que se basa este Trabajo Fin de Máster

2.2.2. Segmentación semántica de escenas

La segmentación semántica es "la tarea de asignar significado a un objeto". Esto es: asociar cada píxel de una imagen a una clase determinada, como se puede observar en la Figura 2.10, de manera que se dota de significado a la escena. Además, esta técnica permite localizar objetos (coches, peatones, etc.) dentro de la imagen con una mayor precisión que usando técnicas de *bounding box*, ya que el etiquetado de la imagen es píxel a píxel y no mediante porciones de imagen [23, 24]. De esta manera, los sistemas de control de los vehículos autónomos se nutren de una información más precisa y de mayor calidad, mejorando su eficiencia y seguridad. No obstante, hasta la aparición del *deep learning*, se usaban técnicas como *Random Forest based classifiers* para segmentar semánticamente escenas, aportando tasas de acierto cercanas al 70 % [25, 26]. Con la aparición del *deep learning* y de las CNN, la tasa de acierto aumentó hasta alcanzar aproximadamente un 94 % [27], lo que refleja la importancia de las CNN en este proceso.

Figura 2.10: Ejemplo de imagen segmentada semánticamente. **Izq.** Imagen original. **Der.** Imagen segmentada.

Llegados a este punto, el lector posee conocimientos sobre CNN y la segmentación semántica de escenas, que son los pilares básicos de este Trabajo Fin de Máster. A continuación, explicaremos el desarrollo de este Trabajo Fin de Máster, que incluye una explicación más profunda sobre la segmentación semántica de imágenes usando CNN.

Capítulo 3

Desarrollo

En este capítulo se presenta cómo se ha realizado este Trabajo: desde la descripción de la herramienta a usar hasta su programación, entrenamiento y testeo.

3.1. Justificación del proyecto

A lo largo de las secciones anteriores se ha visto que existen múltiples herramientas que permiten posicionar la carretera dentro de una imagen, pero todas ellas con algunas desventajas: determinando manualmente la posición de la carretera corremos el riesgo de definir una zona en la que no hay carretera, y por búsqueda de líneas sin la existencia de líneas no funcionaría el algoritmo y por ende no podríamos posicionar la carretera. Debido a ello, en este Trabajo vamos a desarrollar un modelo de Inteligencia Artificial cuyo objetivo es segmentar semánticamente una imagen, de manera que podamos localizar la carretera en ella de manera automática y fiable, ya que este tipo de algoritmos no están basados en buscar unas determinadas características en la imagen para definir otras, sino que aprenden y obtienen una inteligencia sobre la aplicación en la que son aplicadas.

3.2. Recursos

Este Trabajo ha sido desarrollado en el lenguaje de programación Python debido a que existe una gran comunidad de desarrolladores de Inteligencia Artificial basada en este lenguaje de programación. Además, se ha decidido desarrollar este Trabajo en Python ya que la librería más famosa y usada actualmente para las aplicaciones de *Deep Learning* es Tensorflow, desarrollada principalmente para Python.

En cuanto a los recursos necesarios para el desarrollo del Trabajo, ha sido necesaria la obtención del modelo pre-entrenado VGG 16 [28] ya que es el modelo sobre el que basaremos el desarrollo de este Trabajo, así como la obtención y descarga del dataset de Kitti para segmentación semántica de carreteras, que nos proveerá de las imágenes necesarias para el desarrollo de nuestra aplicación [29].

3.3. Programación de Redes Neuronales usando Tensorflow

En esta sección se detallará la API de Tensorflow para construir Redes Neuronales en Python.

3.3.1. ¿Qué es Tensorflow?

Tensorflow es librería software de código abierto para computación numérica usando grafos de flujo de datos. Los nodos en el grafo representan operaciones matemáticas, y las conexiones representan arrays de datos multidimensionales, llamados tensores (de ahí el nombre Tensorflow). Tensorflow fue originalmente desarrollado por investigadores del *Google Brain Team* dentro de la organización de investigación *Google's Machine Intelligence* [30]. Un ejemplo de grafo es la Figura 3.1, donde el nodo verde es la entrada al sistema, los nodos azules son variables internas, los nodos lilas son operaciones matemáticas, y las uniones representarían los tensores.

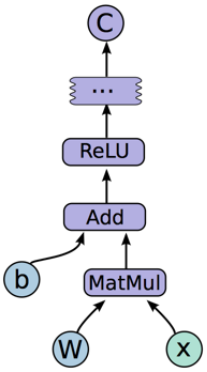

Figura 3.1: Grafo computacional

A continuación, se describirán los tipos de datos y funciones usados durante el desarrollo de este Trabajo para programar las arquitecturas aquí

propuestas. Solo se presentarán los parámetros de las funciones que se consideran de vital importancia para su correcto funcionamiento.

3.3.2. Constantes

Las constantes en Tensorflow definen datos del grafo que no variarán su valor a lo largo de la ejecución del grafo. Las constantes en Tensorflow se pueden crear de diversas maneras [31], siendo una de las más usadas la creación a través de la llamada a la función *tf.constant(value, dtype, name)* [32], donde:

- **Value:** representa el valor de la constante, y puede ser un valor constante o una lista de valores de tipo *dtype*.
- **Dtype:** representa el tipo de dato con el que se representará la constante.
- **Name:** es el nombre asociado dentro del grafo a la constante creada.

3.3.3. Variables

Las variables en Tensorflow definen datos del grafo que pueden variar su valor a lo largo de la ejecución del mismo. Las variables en Tensorflow se crean mediante la llamada al constructor *tf.Variable(initial_value, trainable)* [33], donde:

- **Initial_value:** representa el valor inicial de la variable. Su valor puede ser asignado mediante un método *initializer* de Tensorflow, como puede ser el inicializador truncado aleatorio [34].
- **Trainable:** siendo *True* indica que la variable es entrenable, y por lo tanto ha de ser incluida en la lista de variables del optimizador (ver apartado 3.3.10).

3.3.4. Placeholders

Los *placeholder* definen las entradas a un grafo descrito en Tensorflow, y siempre ha de ser alimentado con algún tipo de dato [35]. Por ejemplo, en la Figura 3.1, el nodo verde es un *placeholder* en el grafo, y es el que alimentará al sistema con los datos de entrada. La forma de crear un *placeholder* en Tensorflow es llamando a la función *tf.placeholder(datatype, shape, name)*, donde:

- **Datatype:** representa el tipo de dato que se alojará en el *placeholder*. Éstos pueden ser enteros, números en coma flotante de precisión simple y doble, entre otros muchos [36].
- **Shape:** representa la dimensionalidad del tensor a crear. Por ejemplo, si el dato de entrada es una imagen de 1024×1024 píxeles y en color, la dimensionalidad del tensor deberá ser de 1024×1024×n, siendo n el número de canales del color usado.
- **Name:** representa el nombre asociado al tensor.

3.3.5. Inicialización de variables

Durante el desarrollo y programación de un grafo usando Tensorflow, las constantes y variables son representaciones simbólicas, por lo que a la hora de ejecutar el grafo es necesario inicializarlas. Existen diversas maneras de inicializarlas, pero la más extendida es usando la función *tf.global_variables_initializer()* [37], y ha de ser llamada tras crear la sesión (ver apartado 3.3.11).

3.3.6. Suma matricial

La suma matricial se refleja en Tensorflow mediante la llamada a la función *tf.add(a, b)*, donde *a* y *b* son matrices creadas como objetos de Tensorflow [38].

3.3.7. Multiplicación matricial

La multiplicación matricial se refleja en Tensorflow mediante la llamada a la función *tf.matmul(a, b)*, donde *a* y *b* son matrices creadas como objetos de Tensorflow [39].

3.3.8. Convolución

La convolución se realiza mediante la llamada a la función *tf.layers.conv2d(input, filters, kernel_size, strides, padding)* [40], donde:

- **Input:** representa el tensor de entrada.
- **Filters:** representa la dimensionalidad del tensor de salida.
- **Kernel_size:** es un entero o lista de dos enteros que especifica las dimensiones de una ventana de convolución 2D.

- **Strides:** es un entero o lista de dos enteros que especifica los pasos de una ventana de convolución 2D; es el número de píxeles que se desplaza la ventana de convolución entre operación y operación.
- **Padding:** puede ser *same* o *valid*, donde:
  - **Same:** tiene como objetivo generar una salida de la mismas dimensiones que la entrada, y para ello rellena con ceros en los bordes (Figura 1.12 izquierda).
  - **Valid:** no realiza el procesado anteriormente descrito, y por tanto las dimensiones de la salida pueden verse reducidas respecto a las de la entrada (Figura 1.12 derecha) [14].

3.3.9. Convolución transpuesta

La convolución transpuesta se realiza mediante la llamada a la función *tf.layers.conv2d_transpose(input, filters, kernel_size, strides, padding)* [41], donde los parámetros de la función son los mismos que los descritos en el apartado 3.3.8.

3.3.10. Optimización de la función de coste

La optimización de la función de coste se puede realizar usando diferentes tipos de optimizadores, como pueden ser el *Gradient Descent Optimizer* o *Adam Optimizer*. Los constructores se encuentran dentro de la clase *train* de Tensorflow, y ha de indicarse el *learning rate*, entre otros parámetros, y para optimizar la función de coste ha de usarse la función *minimize* de los optimizadores [42]. Un ejemplo de aplicación se representa en la Figura 3.2.

```
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy_loss)
```

Figura 3.2: Ejemplo de un optimizador para la función de coste

3.3.11. Sesiones

Las sesiones en Tensorflow encapsulan el entorno en el que se ha ejecutado un grafo y los tensores asociados, de manera que, en caso de estar entrenando una red, la sesión contiene los pesos de la red y éstos pueden ser almacenados mediante un objeto *saver* para ser usados en producción. Las sesiones se crean en Tensorflow mediante el constructor *tf.Session()* [43].

3.4. Fully Convolutional Neural Networks

En la sección 1.4 vimos que las Redes Convolutivas son una herramienta muy potente para la extracción de características y clasificación de imágenes, pues generan múltiples filtros especializados en detectar diferentes patrones en la imagen, de menor a mayor grado de complejidad a medida que se profundiza en la red. Además, preservan información espacial de la imagen ya que la operación de convolución lo permite. No obstante, esta información se pierde debido a la capa *Fully Connected*, por lo que las Redes Convolutivas permiten responder a la pregunta "¿qué tipo de objeto es?". Sin embargo, el objetivo de este Trabajo Fin de Máster es responder a la pregunta "¿dónde está este objeto en la imagen?" Por lo que cabría preguntarse si habría alguna forma de preservar la información espacial de las Redes Convolutivas, de manera que pudiese responderse a dicha pregunta.

Para responder a la pregunta planteada anteriormente se diseñaron las Redes Neuronales Totalmente Convolutivas (FCN, por sus siglas en inglés) en el año 2014. En el año 2014 los investigadores Jonathan Long, Evan Shelhamer, Trevor Darrell, de la Universidad de Berkeley, propusieron la realización de segmentación semántica de imágenes usando FCN, un nuevo y novedoso tipo de red basada única y exclusivamente en CNN, y en una arquitectura codificador-decodificador [27]. La idea subyacente a la idea de usar solo CNNs es que las CNNs permiten preservar la información espacial, por lo que permitirían responder a la pregunta "dónde está el objeto".

A continuación se explicarán las principales características y la arquitectura de la red usada para el desarrollo de este Trabajo: las *Fully Convolutional Neural Networks*

3.4.1. Características de las Fully Convolutional Neural Networks

Las FCN son redes, que, como su propio nombre determina, están compuestas básicamente por operaciones convolutivas. Las Redes Convolutivas permiten clasificar una imagen de entrada, pero no preservan información espacial sobre dónde están localizadas las texturas que definen a un objeto dentro de una imagen, ya que la capa FC no preserva este tipo de información. Sin embargo, la principal característica de las FCN es que permiten preservar información espacial y responder a la pregunta ¿dónde está el objeto? [44] Esta característica es una característica elemental en el ámbito de la conducción autónoma, ya que los sistemas de visión han de determinar dónde están de manera precisa los diferentes objetos que pueden entrar en

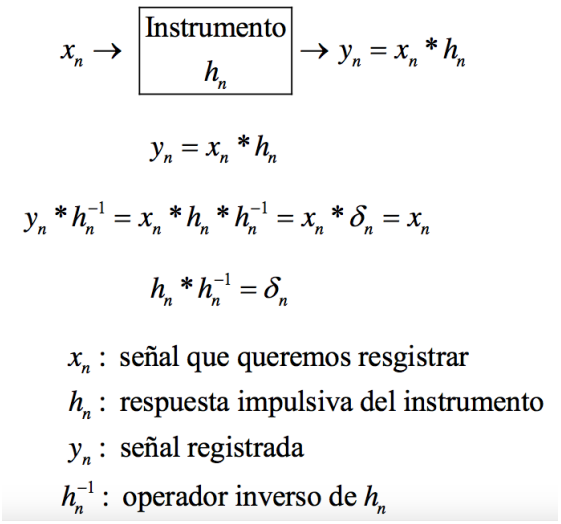

Figura 3.3: Procedimiento matemático de la deconvolución

juego, como son coches, personas, obstáculos, y la carretera, que es sobre la que se basará este Trabajo Fin de Máster.

Además, las redes FCN hacen uso de la deconvolución. La deconvolución es una técnica usada en varios campos, como en procesado digital de la señal, física, etc. en los que su objetivo es extraer una determinada característica (un objeto, etc.) dados unos datos alterados, y que enmascaran a dicha característica. Por ejemplo, en astronomía se usa para determinar la forma de un objeto sabiendo que la imagen tomada contiene a dicho objeto pero deformado. Este proceso puede llevarse a cabo considerando que la imagen tomada es el resultado de la convolución del objeto a buscar y un determinado filtro, por lo que si se conoce el filtro y la imagen tomada, y se aplica la operación inversa (la deconvolución), entonces se puede obtener la forma del objeto original. Matemáticamente este hecho se refleja en la Figura 3.3 [45].

En las redes FCN este proceso permite realizar el *upsamplig* de la imagen entre las diferentes capas deconvolutivas, permitiendo aumentar la dimensionalidad de la imagen y obtener finalmente una imagen de salida con el mismo tamaño que la imagen de entrada. La deconvolución en las redes FCN es la combinación de una operación de *padding* que aumenta la dimensionalidad de la imagen de entrada, y un kernel convolutivo, de manera que la imagen de salida resultante tiene mayor dimensionalidad que la de entrada; es decir, se ha realizado el *upsumpling* de la imagen de entrada. De esta manera, el proceso de *upsampling* es aprendible, ya que está constituido por capas

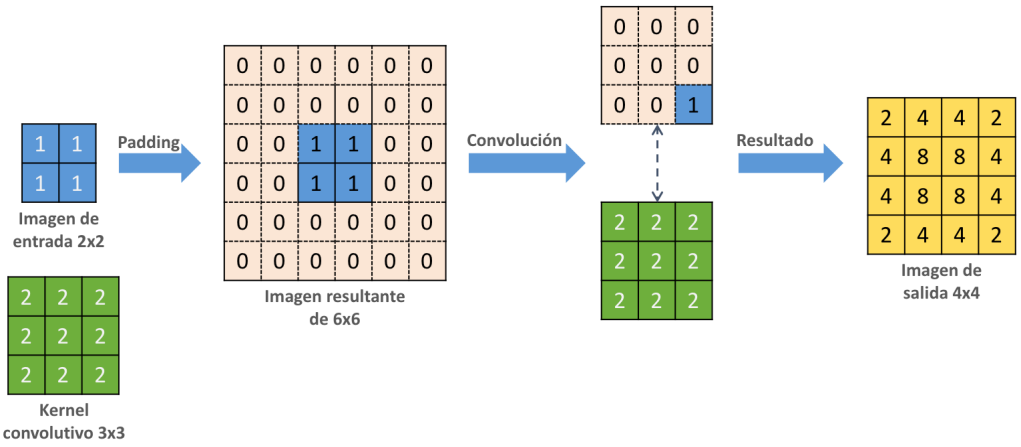

Figura 3.4: Procedimiento de deconvolución

convolutivas, que, como ya se ha explicado anteriormente, están basadas en pesos que son actualizables a través del proceso de *backpropagation*. Por lo tanto, los kernels convolutivos de las capas deconvolutivas generan filtros que se activarán ante determinados patrones en el mapa de características generado tras la codificación de la imagen, de manera que finalmente se aprende a asociar dichos patrones con la presencia de un objeto en la imagen (segmentación semántica). Un ejemplo gráfico de lo aquí descrito es la Figura 3.4: supóngase que el mapa de características obtenido tras la codificación de la imagen tiene un tamaño de 2×2. Para aumentar su tamaño a una imagen de 4×4 mediante el uso de un kernel convolutivo de 3×3, es necesario añadir 2 filas y 4 columnas (2 en la parte izquierda y 2 en la parte derecha) más para rellenarla con ceros, de manera que el resultado tras aplicar la operación convolutiva sobre dicha imagen sea de 4×4.

Dado que las redes FCN están compuestas únicamente por operaciones convolutivas y deconvolutivas, se elimina la restricción asociada a que el tamaño de la matriz de salida esté restringida al tamaño de la red FC, ya que cambiar el tamaño de la imagen de entrada significa cambiar el tamaño de las matrices a usar en las capas FC, que son de tamaño fijo. Por el contrario, en las redes FCN el tamaño de la imagen de entrada no está limitada debido a que la operación de convolución no está sujeta a restricciones de tamaño, lo que permite entrenar un único clasificador y usarlo para escalas de imágenes diferentes [44, 27].

En resumen, las características principales de una red FCN son:

1. Son redes compuestas básicamente por operaciones convolutivas,

2. permiten clasificar y localizar espacialmente, y

3. la dimensionalidad de la entrada no está condicionada, de manera que un único clasificador puede ser usado para diferentes escalas de imágenes.

3.4.2. Arquitectura de las Fully Convolutional Neural Networks

Una vez explicadas las características de las FCN, lo siguiente es explicar su arquitectura. Como ya se ha avanzado en la sección 3.4.1, las Redes FCN están compuestas básicamente por operaciones convolucionales, y su arquitectura se estructura en tres bloques: codificador (CNN pre-entrenada), capa convolutiva de 1×1 y decodificador (Red Deconvolutiva) [46, 47], descritos a continuación y observables en la Figura 3.5.

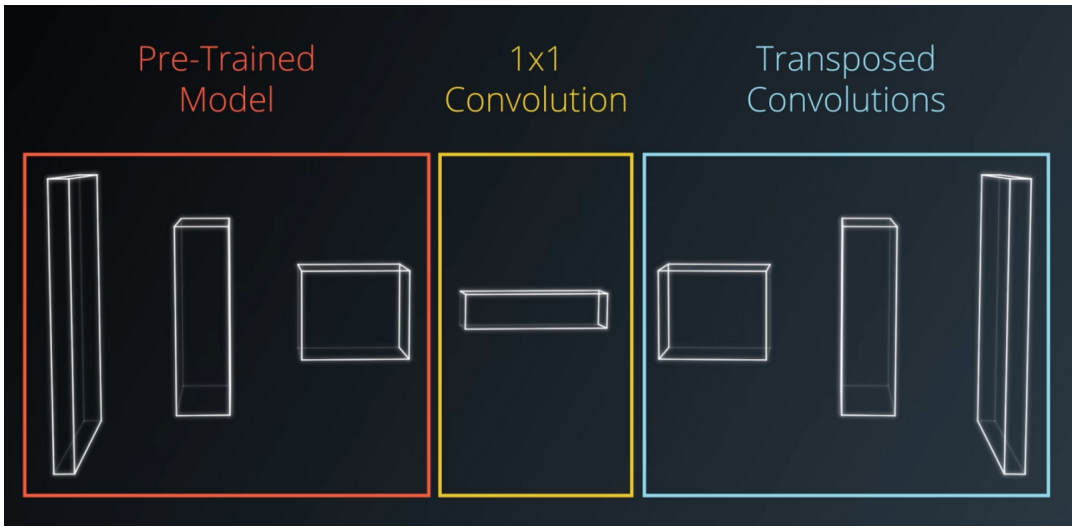

Figura 3.5: Arquitectura de las redes FCN

3.4.2.1. Codificador

El codificador es el bloque que permite obtener un mapa de características dada una imagen de entrada, cuyo nivel de abstracción es mayor al presente en la información visual de la imagen. El codificador es, normalmente, una CNN que extrae las características de una imagen de entrada, y suelen ser modelos pre-entrenados que permiten clasificación multiclase. En la Figura 3.5 se encuentra representado en la parte encuadrada en rojo.

3.4.2.2. Conversión de FC a CNN de 1x1

El segundo bloque dentro de las Redes FCN es la capa convolutiva de 1×1. Este bloque consiste en sustituir la capa FC de la CNN por una capa convolutiva de kernel 1×1, lo que permite generar una clasificación de la imagen al mismo tiempo que se preserva la información espacial [44]. El resultado de colocar esta capa es un tensor 4D que nos permitirá, tras un proceso deconvolutivo, crear una imagen de salida de la misma dimensionalidad que la imagen de entrada [46]. En la Figura 3.5 se encuentra representado en la parte encuadrada en amarillo.

3.4.2.3. Decodificador

Finalmente, el tercer y último bloque de una FCN es el decodificador. El decodificador, como su propio nombre indica, permite decodificar el mapa de características obtenido, de manera que se pueda localizar espacialmente las texturas asociadas a los patrones descubiertos mediante el proceso convolutivo. Este bloque es el que permite generar una imagen de salida del mismo tamaño de la imagen de entrada, y para ello hace uso de capas deconvolutivas. La deconvolución "se usa en restauración de señales para recuperar datos que han sido degradados por un proceso físico que puede describirse mediante la operación inversa, una convolución" [48]. En el caso de las FCN, la deconvolución permite extraer qué característica, de las extraídas durante el proceso convolutivo en el codificador, pertenece a cada píxel, de manera que a cada píxel se le asocia una o más características. La matemática detrás de una capa deconvolutiva es exactamente la misma que hay tras una convolutiva, aunque con una ligera diferencia: las capas convolutivas se basan en multiplicar una matriz A de entrada por un *kernel* B convolutivo y sumar los resultados del vector resultante, de manera que se obtiene un único valor c por cada $n \times m$ píxeles de imagen (Ec. 3.1) (Figura 3.6). Por el contrario, en el caso de las deconvoluciones, se busca obtener una matriz A como resultado de resolver un sistema de ecuaciones. Este proceso es el denominado proceso de *up-sampling*, y permite generar la imagen de salida mediante el encadenamiento de varias capas deconvolutivas [49]. En la Figura 3.5 se encuentra representado en la parte encuadrada en azul.

Para representar la imagen segmentada, las redes FCN cuentan con un código de colores RGB asociados a cada una de las clases del problema, de manera que analizando qué clase está asociada a cada píxel podemos colorearlo usando el código RGB de la misma, obteniéndose, por tanto, una representación gráfica y comprensible.

$$c = \sum_{i=0}^{j} a_i \cdot b_i \tag{3.1}$$

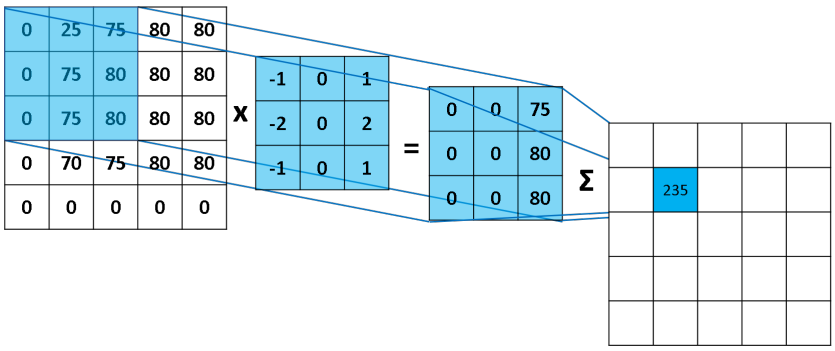

Figura 3.6: Ejemplo de proceso convolutivo

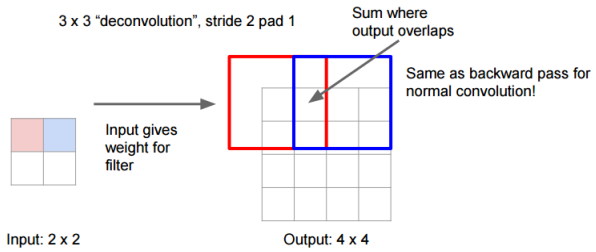

Figura 3.7: Ejemplo de proceso deconvolutivo

3.5. Segmentación de carreteras usando Fully Convolutional Neural Networks

En este apartado se abordará el problema propuesto como Trabajo Fin de Máster. Para una total comprensión del problema, se ha optado por dividir su explicación en diferentes sub-bloques que permitirán ir explicando paulatinamente su desarrollo. Inicialmente se explicará el escenario sobre el que se desarrollará este Trabajo, y a continuación definir las arquitecturas propuestas. Finalmente, los resultados se muestran y analizan en el Capítulo 4.

3.5.1. Escenario

El escenario en el que se desarrolla este Trabajo es en el ámbito de las imágenes aplicadas a la conducción autónoma. Las imágenes aquí usadas se corresponden con las obtenidas usando una cámara RGB en el frontal de un vehículo, siendo un ejemplo de este tipo de imágenes la Figura 3.8 izquierda.

El *dataset* usado para la realización de este Trabajo es el *Kitti Dataset* desarrollado por el Karlsruhe Institute of Technology, y está formado por imágenes RGB (Figura 3.8 izquierda) y etiquetadas para el entrenamiento y test (Figura 3.8 derecha), y ficheros de texto para calibración [29]. En el caso que nos concierne, solo se hará uso del set de imágenes y no de los archivos de calibración.

A continuación, se explicarán las arquitecturas propuestas para la consecución de los objetivos de este Trabajo.

Figura 3.8: **Izq.:** Imagen RGB. **Der.:** *Ground truth*

3.5.2. Arquitecturas de segmentadores propuestas

A fin de poder extraer conclusiones de este Trabajo Fin de Máster y de las posibles arquitecturas a usar en futuros trabajos, en esta sección se presentan las arquitecturas usadas para experimentación en este Trabajo. Ambas usan el mismo codificador, el modelo pre-entrenado VGG16, siendo el único cambio apreciable entre ellas la existencia o no de conexión entre el codificador y el decodificador.

3.5.2.1. Codificador: Modelo pre-entrenado VGG16

El codificador usado en este Trabajo Fin de Máster es el modelo pre-entrenado VGG16. El modelo VGG16 es un modelo creado dentro del marco VGG (es una Red Neuronal Convolutiva propuesta por K. Simonyan y A. Zisserman de la Universidad de Óxfod, en la publicación "*Very Deep Convolutional Networks for Large-Scale Image Recognition*". Este modelo obtuvo un

92.7 % de precisión en el dataset ImageNet, un dataset compuesto por aproximadamente 14 millones de imágenes pertenecientes a 1000 clases [50, 28]. Sin embargo, este modelo es el modelo CNN, es decir, el modelo con las capas FC todavía en el modelo. En el caso que concierne a este Trabajo, el modelo es el mismo, solo que con la diferencia de que las capas FC han sido sustituidas por capas convolutivas de 1×1×4096, tal y como se representa en la Figura 3.10.

En cuanto a la arquitectura del modelo CNN original, ésta puede observarse gráficamente en la Figura 3.9. Durante el entrenamiento, el tamaño de la imagen de entrada se fijó en 224×224, su formato era RGB y se le substrajo la media del valor RGB a cada píxel de las imágenes de entrenamiento. Sucediendo a la entrada, se encuentra una pila de capas convolutivas de tamaño 3×3, con un número de características a extraer comenzando por 64 y aumentándose el doble tras la aplicación de las capas de *pooling*, *stride* de tamaño 1 y *padding* ajustado para mantener la resolución espacial de la imagen entre capas convolutivas adyacentes. Para disminuir el tamaño espacial de la imagen, la red está provista de 5 capas de *max pooling* de tamaño 2×2 y *padding* de tamaño 2, lo que permite disminuir el tamaño de la imagen entre capas a la mitad. Sucediendo a la pila convolutiva se encuentra una red FC de 3 capas: las dos primeras con 4096 neuronas cada una (una por clase), y la tercera con 1000 neuronas. Y, por último, se encuentra la capa softmax que computa la clasificación de la imagen entre las 1000 clases disponibles. No obstante, como ya se mencionó en el párrafo anterior, en el caso de este Trabajo la arquitectura usada sustituye a las capas FC por capas convolutivas de tamaño 1×1×4096. El código usado para la obtención de este modelo se encuentra en el Apéndice A.1.

Tras obtener el modelo y los tensores necesarios para construir las arquitecturas necesarias, lo siguiente es determinar las dos arquitecturas para el Decodificador desarrolladas en este Trabajo Fin de Máster, siendo el Codificador el mismo para ambos casos.

3.5.2.2. Arquitectura 1: Sin connexionado entre Codificador y Decodificador

En esta arquitectura Codificador y Decodificador no están conectados de ninguna forma, de manera que el Decodificador solo recibe información de las capas predecesoras. En este tipo de arquitecturas la información no es multiescala ni polisemántica debido a que no existe conexionado entre el Codificador (información más cercana al entorno, aunque procesada) y

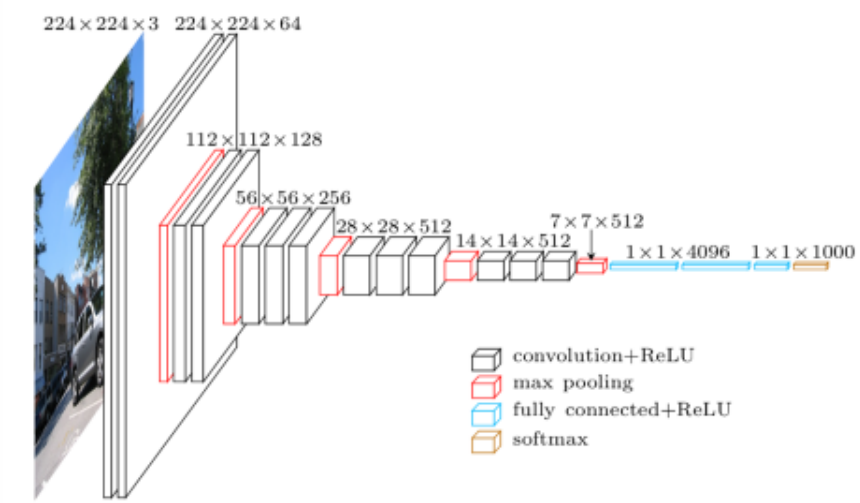

Figura 3.9: Arquitectura del modelo pre-entrenado VGG16

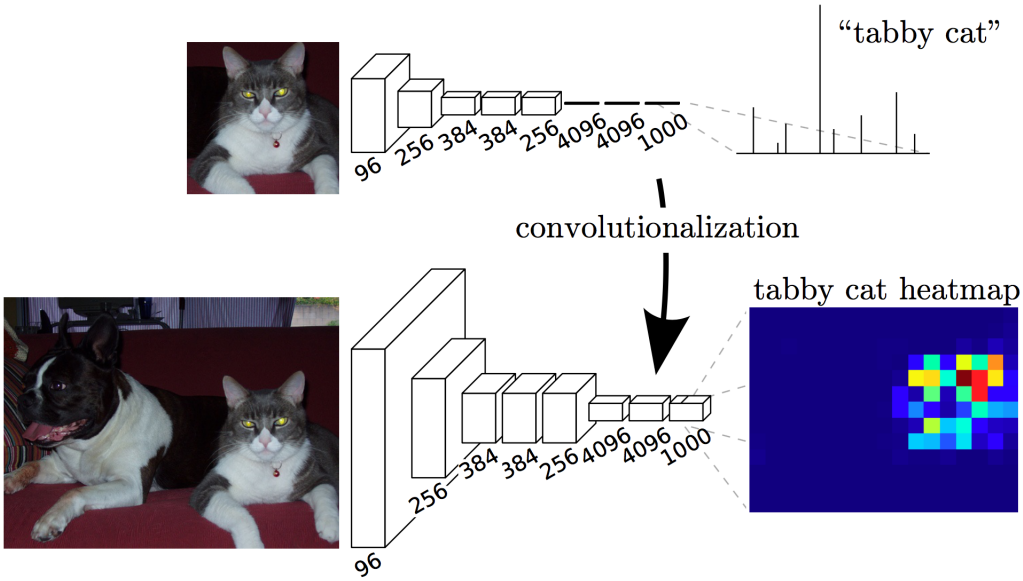

Figura 3.10: Transformación de la arquitectura del modelo pre-entrenado VGG16 CNN a un modelo FCN

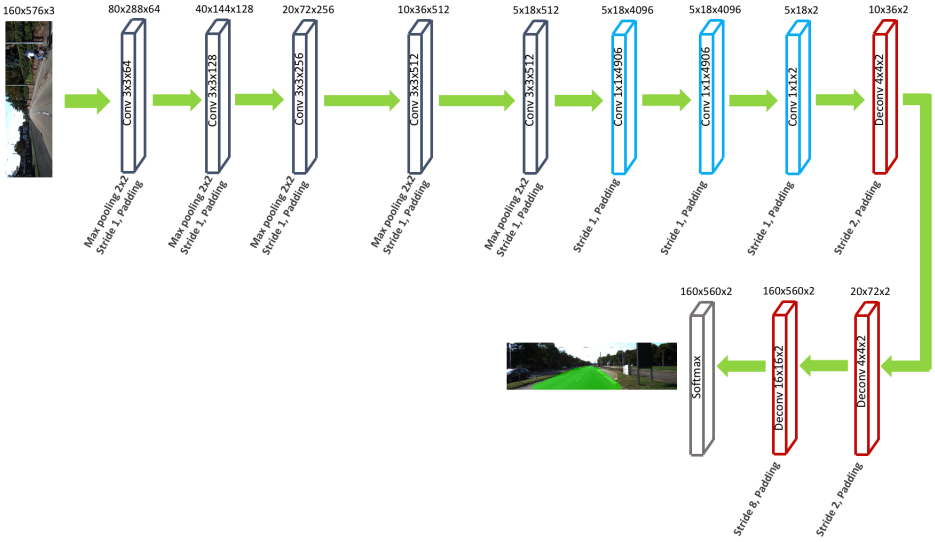

Figura 3.11: Arquitectura 1 de las redes FCN propuestas

el Decodificador (información totalmente procesada, abstracta y alejada del entorno), por lo que la precisión esperada es menor que en el caso de la existencia de dicha interconexión. La arquitectura propuesta puede observarse en la Figura 3.11, donde se observa la existencia de tres capas deconvolutivas consecutivas, de tamaño 4×4×2 con *stride* 2 y *padding* las dos primeras y tamaño 16×16×2, *stride* 8 y *padding* la última. En cuanto al código que permite generar esta arquitectura puede encontrarse en el Apéndice A.2.

3.5.2.3. Arquitectura 2: Con connexionado entre Codificador y Decodificador

En este caso, el Codificador y el Decodificador sí están conectados, de manera que el Decodificador recibe información de capas precedesoras, y además de capas con información cercana al entorno, por lo que la información será multiescala y polisemántica. Gracias a este conexionado, la red poseerá diferentes fuentes de información que la ayudarán a mejorar la precisión de su clasificación, y por ende, a ofrecer una mejor y más precisa localización de la carretera dentro de la imagen.

La arquitectura propuesta puede observarse en la Figura 3.12, donde se observa la existencia de capas convolutivas de 1×1×2 con *stride* 1 previas a la de información entre capas, tres capas deconvolutivas de tamaño 4×4×2 con *stride* 2 las dos primeras y *padding*, y tamaño 16×16×2, *stride* 8 y padding la

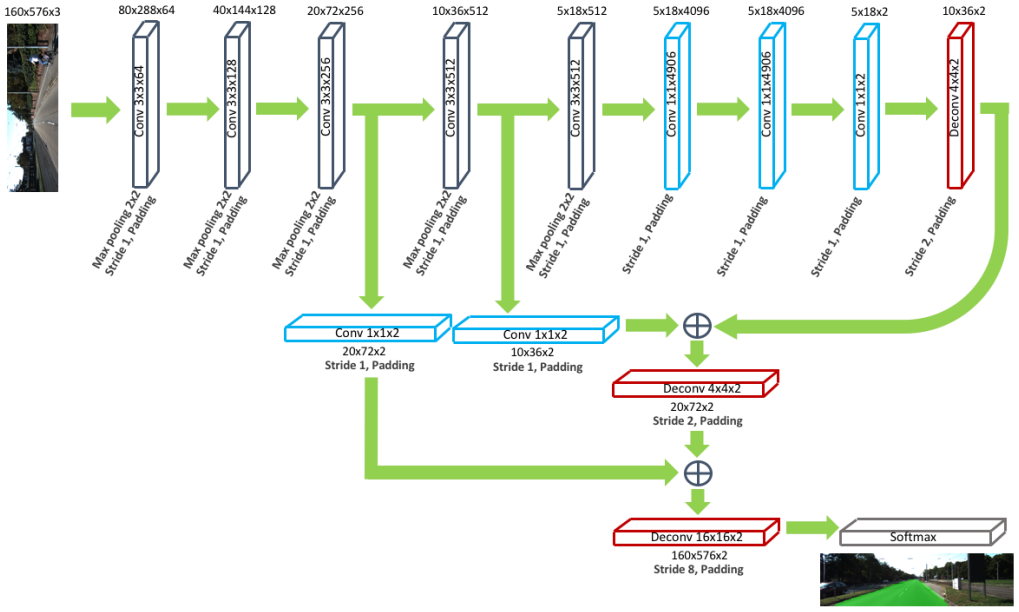

Figura 3.12: Arquitectura 2 de las redes FCN propuestas

última, así como dos adiciones de tensores, lo que permite tener información multiescala y polisemántica. El código usado para la construcción de esta arquitectura se encuentra en el Apéndice A.3.

Una vez definidas las dos arquitecturas propuestas en este Trabajo Fin de Máster, lo siguiente es realizar su entrenamiento.

3.5.3. Entrenamiento de las Redes FCN

Como se ha avanzado a lo largo de este documento, las redes FCN generan un tensor de tamaño igual a la imagen de entrada y con profundidad igual al número de clases a clasificar, de manera que aplicando la conversión a *one-hot* (codificar un conjunto de datos como un vector binario, donde solo puede haber un '1' y que corresponde con la posición del mayor valor del conjunto de datos) se puede comparar la clasificación realizada con la clasificación real (también codificada en condificación *one-hot*, y calcular la función de pérdidas asociada a dicha clasificación. En cuanto a la obtención de las etiquetas (clasificación real de la imagen), éstas se obtienen haciendo uso de un código de colores RGB que permite determinar qué píxeles están asociados a cada clase en la imagen etiqueta de entrenamiento.

En el caso que concierne a este Trabajo Fin de Máster, el entrenamiento

de la red se realizó mediante el uso de un set de 289 imágenes procedentes del Kitti-dataset para segmentación semántica de carreteras [29], compuestas por el *ground truth* de la imagen y la imagen en sí, tal y como se muestra en la Figura 3.8, y de un código de colores predefinido. El set de entrenamiento está compuesto por 239 imágenes (el 80 % del dataset) y el set de test por 58 imágenes (el 20 % del dataset), de manera que tras el entrenamiento de las diferentes arquitecturas se ha podido evaluar la precisión de las mismas.

No obstante, como ya se ha avanzado a lo largo de este documento, lo primero para llevar a cabo el entrenamiento de una red es definir su función de pérdidas, lo cual se realiza usando la función *optimize()* de los archivos *main.py* y *main_ FCN_ sin_ conexion.py*. Esta función (definida en el Apéndice A.4.1), define que la función de pérdidas es la media de la entropía cruzada de las etiquetas con la clasificación realizada (línea 22), así como que la función optimizadora es o *SGD* (Descenso por el Gradiente Estocástico, por sus siglas en inglés) o la *ADAM* (líneas 25 - 31). En el caso que concierne a este Trabajo, el optimizador elegido fue *ADAM* (línea 12) ya que tras diversas pruebas se determinó que el *SGD* era demasiado lento.

Por último, tras definir el optimizador, hay que obtener los *batches* de imágenes y sus etiquetas para el entrenamiento (obtener n imágenes del dataset, siendo n menor al tamaño total del dataset) y ejecutar el entrenamiento (código en Apéndice A.4.2. En el caso que concierne a este Trabajo, los parámetros de entrenamiento han sido:

- **Batch size:** 4
- **Número de épocas:** 4
- **Tamaño de la imagen de entrada:** 160×576
- **Learning rate:** 10^{-3}
- **Equipo:**
  - PC: MacBook Pro mediados de 2012
  - Procesador: 2.5 GHz Intel Core i5
  - Memoria: 8 GB 1600 MHz DDR3
  - Gráficos: Intel HD Graphics 4000 1536 MB

3.5.4. Test de las Redes FCN

Una vez realizado el entrenamiento de las redes, lo siguiente es testearlas, de manera que se pueda obtener la precisión media de la red, y observar gráficamente cómo de buena es la red así como calcular la precisión del modelo. En el caso que concierne a este Trabajo, el dataset de test está compuesto por 58 imágenes, relativas al 20 % del dataset completo.

Capítulo 4

Resultados

En este capítulo se presentarán los resultados obtenidos con las diferentes arquitecturas definidas en la sección 3.5.2. Para ello, ha sido necesario realizar el entrenamiento de las diferentes arquitecturas propuestas y guardar sus modelos (almacenados en la carpeta *modelos*). Inicialmente se presentarán los resultados por arquitectura, y finalmente se realizará una valoración conjunta de los resultados.

4.1. Arquitectura 1

La gráfica presente en la Figura 4.1 muestra la precisión de la arquitectura 1 (sin conexión entre las diferentes capas del modelo) para cada una de las muestras del dataset de test. En ella se observa que la precisión de esta arquitectura oscila entre 81.79 % y 98.06 %, sus valores estadísticos los recogidos en la Tabla 4.1, y el valor medio de la precisión de 93.86 %. Analizando su varianza, se observa que la precisión de este modelo es variable, lo que permite determinar que la fiabilidad de este modelo es baja y que, por tanto, no puede ser utilizada para la aplicación que persigue este Trabajo. En cuanto a la representación gráfica de la segmentación realizada por esta arquitectura, la Figura 4.2 muestran dos filas compuestas por la imagen real y la inferencia realizada. Como se puede observar en ella, la red es capaz de realizar una buena inferencia, pero incluye zonas de la imagen que no son carretera como si lo fuera y/o no infiere zonas de la carretera como tal, lo cual reafirma que no es asequible para la aplicación que se persigue en este Trabajo Fin de Máster. Por último, el tiempo de entrenamiento para esta arquitectura fue de 2.16 horas.

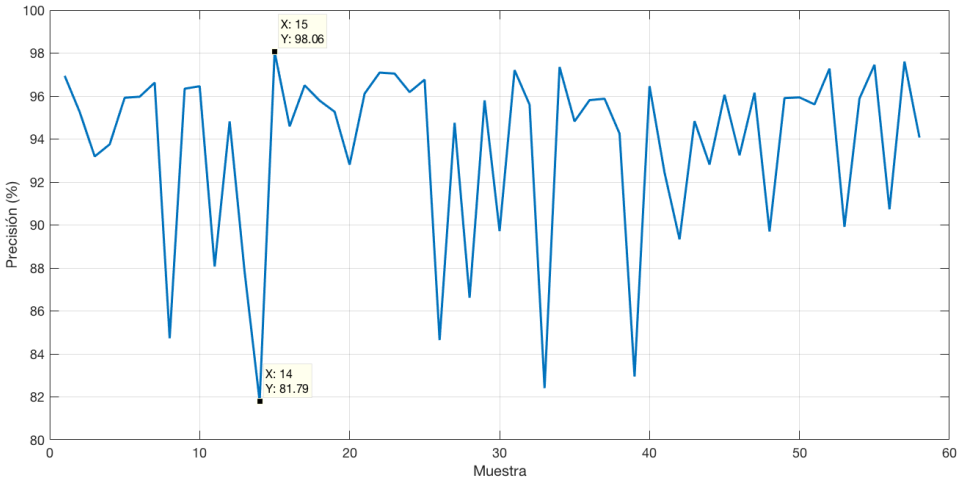

Figura 4.1: Gráfica de precisión de la arquitectura 1

Media	**Mediana**	**Varianza**
93.68 %	81.79 %	4.18

Tabla 4.1: Estadísticos asociados a la precisión de la arquitectura 1

Figura 4.2: Resultados de la segmentación de la arquitectura 1. **Izq.:** Escena real. **Der.:** Segmentación de carretera realizada

4.2. Arquitectura 2

En cuanto a la arquitectura 2, la gráfica presente en la Figura 4.3 muestra la precisión de dicha arquitectura (con conexión entre las diferentes capas del modelo) para cada una de las muestras del dataset de test. En ella se observa que la precisión de esta arquitectura oscila entre 92.82 % y 99.26 %, sus valores estadísticos los recogidos en la Tabla 4.2, y el valor medio de la precisión de 97.23 %. En cuanto a su varianza, se observa que la precisión de este modelo es prácticamente estable, lo que permite determinar que la fiabilidad de este modelo es alta y que, por tanto, puede ser utilizada para la aplicación que persigue este Trabajo. En cuanto a la representación gráfica de la segmentación realizada por esta arquitectura, la Figura 4.4 muestran dos filas compuestas por la imagen real y la inferencia realizada. Como se puede observar en ella, la red es capaz de realizar una inferencia precisa aunque incluye pequeñas zonas de la imagen que no son carretera como si lo fuera y/o no infiere zonas de la carretera como tal. No obstante, dado que la aparición de este fenómeno es mínimo, se puede afirmar que es asequible para la aplicación que se persigue en este Trabajo Fin de Máster. Por último, el tiempo de entrenamiento para esta arquitectura fue de 3.84 horas.

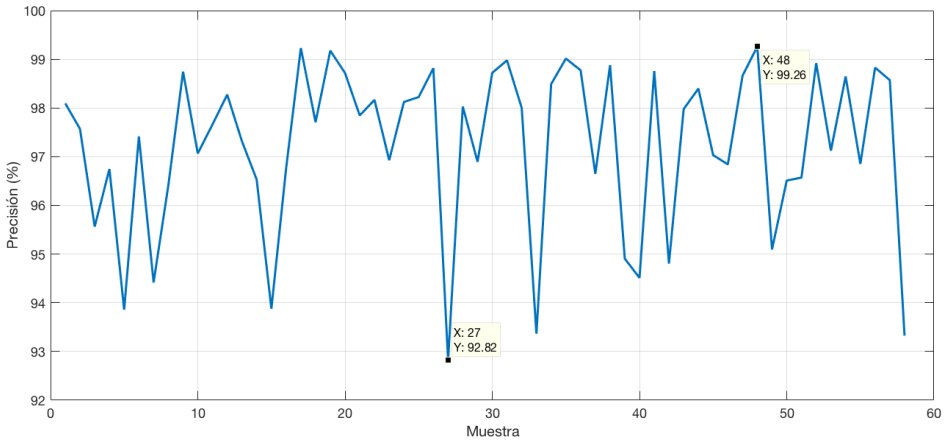

Figura 4.3: Gráfica de precisión de la arquitectura 1

Media	Mediana	Varianza
97.23 %	92.82 %	1.71

Tabla 4.2: Estadísticos asociados a la precisión de la arquitectura 2

Figura 4.4: Resultados de la segmentación de la arquitectura 2. **Izq.:** Escena real. **Der.:** Segmentación de carretera realizada

Capítulo 5

Conclusiones

En este capítulo presentaremos las conclusiones asociadas a este Trabajo Fin de Máster, así como las posibles líneas futuras de investigación y/o desarrollo posibles.

5.1. Conclusiones

En este Trabajo Fin de Máster se han propuesto dos arquitecturas diferentes: la primera, sin conexionado entre el codificador y el decodificador, por lo que la información que posee no es multisemántica, y la segunda, con dicho conexionado, y por tanto con información polisemántica (para más información ver sección 3.5.2). Tras el entrenamiento de ambas arquitecturas, se ha construido una gráfica (Figura 5.1), tabla (Tabla 5.1) y figura (Figura 5.2) comparativas, de las que se desprende la existencia de una gran diferencia entre ambas arquitecturas:

1. La precisión media de la arquitectura 2 es un 3.37 % mayor, obteniendo, como consecuencia directa, una mejor segmentación de la carretera.

2. La variación de la precisión de la arquitectura 2 es mucho menor, lo que traducido a fiabilidad indica que la red más es estable.

Del análisis de las imágenes presentes en la figura 5.2 se desprende que la arquitectura 2 premite:

Arquitectura	**Media**	**Mediana**	**Varianza**
Arquitectura 1	93.68 %	81.79 %	4.18
Arquitectura 2	**97.23 %**	92.82 %	1.71

Tabla 5.1: Comparativa de estadísticos entre arquitecturas

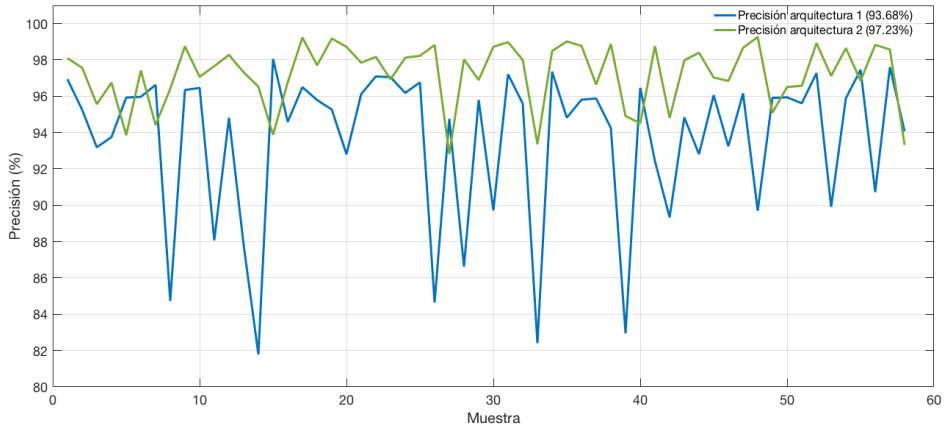

Figura 5.1: Gráfica de comparación de precisión entre arquitecturas

1. Delimitar mejor las zonas de la carretera.

2. Bordear mejor a los vehículos.

3. Una menor tasa de error a la hora de clasificar pavimento muy similar a la carretera como tal.

Por lo tanto, se puede concluir que los sistemas de visión basados en redes FCN con conexionado entre las diferentes capas del modelo son un claro candidato a ser una de las herramientas más usadas para aplicaciones de conducción autónoma, pues nutre de información de alta calidad a los sistemas que lo suceden, de manera que éstos podrían ser más seguros, fiables y rápidos ya que no sería necesario la utilización de sistemas redundantes debido a que la información ofrecida es veraz y fiable.

5.1.1. Revisión de objetivos

En cuanto a los objetivos de este Trabajo Fin de Máster, su objetivo principal era investigar sobre las técnicas existentes en el campo de la visión por computador para la segmentación semántica de carreteras en una imagen, realizado en el capítulo 2, implementar las FCN para segmentación semántica de carreteras (capítulo 3) y comprobar su eficacia (capítulo 4). Por lo tanto, tras la ejecución y redacción de este Trabajo se observa que queda patente el cumplimiento de los objetivos de este Trabajo Fin de Máster.

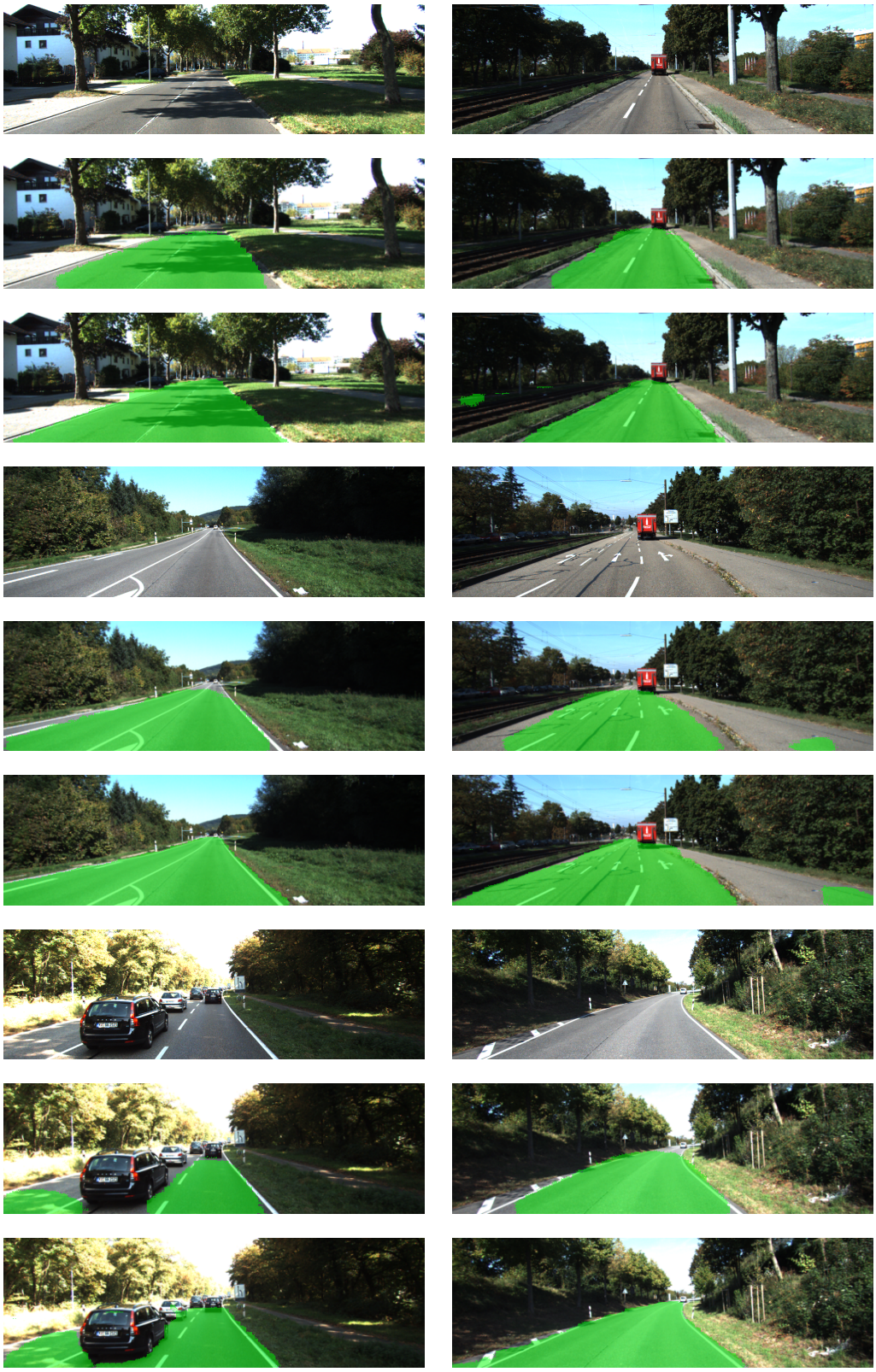

Figura 5.2: Comparativa de segmentaciones. **Primera, cuarta y séptima filas:** Escenas reales. Segmentación de carretera realizada por **segunda, quinta y octava filas:** arquitectura 1. **Tercera, sexta y novena filas:** arquitectura 2

5.2. Líneas futuras

En cuanto a las posibles líneas futuras asociadas a este Trabajo Fin de Máster, entre otras muchas, se encuentran:

- Implementar un segmentador semántico de n clases: Esta línea permitiría clasificar n clases usando segmentación semántica de escenas, de manera que el espectro de información que reciben en los sistemas que suceden a éste es mayor, permitiendo implementar sistemas más seguros.
- Implementar un controlador de posición del vehículo: Con la información visual que el sistema de visión propuesto en este Trabajo ofrece, implementar un sistema que busque y determine la posición de las líneas de la carretera y calcule la posición del vehículo, permitiendo corregir y predecir su rumbo.
- Diseño e implementación de un sistema de visión que permita la lectura de la información en los paneles de tráfico.
- Diseño e implementación de un sistema de visión que permita predecir el rumbo de otros vehículos en la vía.
- Desarrollo e implementación de un sistema de control de movimiento del vehículo.

Como se observa, existen múltiples líneas asociadas a la conducción autónoma: sistemas de visión, sensórica, control del vehículo, etc., por lo que las arriba mencionadas son solo un extracto de las muchas posibles.

Bibliografía

[1] University of Standford. *Convolutional Neural Networks for Visual Recognition*, September 2017.

[2] S. Fernando. Entrenamiento de redes neuronales: mejorando el gradiente descendiente, 2017.

[3] J. Lorenzo. Técnicas de estimación. Technical report, Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería, 2017.

[4] Backpropagation, intituitons.

[5] Michael Nielsen. How the backpropagation algorithm works, 2017.

[6] Matt Mazur. A step by step backpropagation example, 2015.

[7] Neural networks part 3: Learning and evaluation.

[8] Neural networks part 1: Setting up the architecture.

[9] S. H. Nawab A. V. Oppenheim, A. S. Willsky. *Signal and Systems*, chapter 3, pages 69–125. Pearson new international ed., 2ª edition, 2014.

[10] F. J. Acevedo. Ejemplo de convolución en tiempo continuo, October 2010.

[11] J. Ludwig. Image convolution. Technical report, Portland State University, 2013.

[12] Udacity Self-Driving Car Engineer Nanodegree. Convolutional neural networks, February 2017.

[13] Convolutional neural networks: Architectures, convolution and pooling layers.

[14] What is the difference between 'same' and 'valid' in padding tensorflow, September 2017.

[15] Udacity Self-Driving Car Engineer Nanodegree. Explore the design space, February 2017.

[16] C. Burges Y. Lecun, C. Cortes. The mnist database.

[17] División de Ingeniería de Sistemas y Automática. Color en imágenes digitales. Technical report, Universidad Miguel Hernández, 2011.

[18] B. Acha & C. Serrano R. Rangayyan. Color image processing with biomedical applications. Technical report, Universidad de Calgary & Universidad de Sevilla, 2011.

[19] División de Ingeniería de Sistemas y Automática. Detección de bordes en una imagen. Technical report, Universidad Miguel Hernández, 2011.

[20] *Detección de bordes en una Imagen.*

[21] *La Transformada de Hough. Detección de Líneas y Círculos*, 2015.

[22] T. 1. Udacity Self-Driving Car Engineer Nanodegree. Finding lane pixels by histogram and sliding window. Video, November 2016.

[23] S. Chilamkurthy. A 2017 guide to semantic segmentation with deep learning, July 2017.

[24] T. 3 Udacity Self-Driving Car Engineer Nanodegree. Scene understanding, July 2017.

[25] M. Verma R. Jingar. Semantic segmentation of an image using random forest and single histogram class model, 2012.

[26] H. Bischof P. Kontschieder, S. R. Bulò. Structured class-labels in random forests for semantic image labelling, November 2011.

[27] S. Evan & D. Trevor L. Jonathan. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[28] A. Zisserman K. Simonyan. Very deep convolutional networks for large-scale image recognition. *Computer Vision and Pattern Recognition*, September 2014.

[29] Karlsruge Institute of Technology. Benchmark for road segmentation, 2013.

[30] About tensorflow, September 2017.

[31] *Different ways to create constants in Tensorflow*, 2017 October.

[32] *Constants in Tensorflow*, October 2017.

[33] *Variables in Tensorflow*, October 2017.

[34] *Truncated random initializer in Tensorflow*, October 2017.

[35] *Placeholders in Tensorflow*, September 2017.

[36] *Data types in Tensorflow*, September 2017.

[37] *Variables and constants initialization in Tensorflow.*

[38] *Matrix addition in Tensorflow*, September 2017.

[39] *Matrix multiplication in Tensorflow*, September 2017.

[40] *Matrix convolution in Tensorflow*, September 2017.

[41] *Matrix deconvolution in Tensorflow*, September 2017.

[42] *Cost function optimizers in Tensorflow*, September 2017.

[43] *Sessions in Tensorflow*, September 2017.

[44] Term 3 Udacity Self-Driving Car Engineer Nanodegree. Semantic segmentation, October 2017.

[45] R. Rebollo. Filtros inversos y deconvolución.

[46] Term 3 Udacity Self-Driving Car Engineer Nanodegree. Architecture of fully convolutional neural networks, October 2017.

[47] K. Alex & C. Roberto B. Vijay. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, October 2015.

[48] Scientific Volume Imaging. Deconvolución.

[49] Vdumoulin, May 2017.

[50] D. Frossard. Vgg in tensorflow, June 2016.

Apéndice A

Códigos

A.1. Obtención del modelo pre-entrenado VGG16

El código realizado para la descarga del modelo VGG16 puede observarse a continuación (ficheros *helper.py* y *helper_ FCN_ sin_ conexion.py* en la función *maybe_ download_ pretrained_ vgg()*) y permite: descargarse el modelo en caso de que no lo encuentre en el directorio descrito por la entrada *data_ path* (en nuestro caso *data_ path* es la carpeta *data*) y lo almacena en dicho directorio (líneas 5 - 19). Tras descargarlo, o determinar que ya existía en el directorio el archivo en formato *zip*, lo extrae del mismo creando un archivo denominado *vgg*, de manera que se obtiene un archivo y una carpeta relativos al modelo (*saved_ model.pb*) y a las variables del mismo (*variables*) respectivamente (líneas 20 - 31), y elimina el archivo *zip* (línea 34).

```
1  def maybe_download_pretrained_vgg(data_dir):
2    ## Download and extract pretrained vgg model if it doesn't
         exist
3    # :param data_dir: Directory to download the model to
4
5    vgg_filename = 'vgg.zip'
6    vgg_path = os.path.join(data_dir, 'vgg')
7    vgg_files = [
8      os.path.join(vgg_path, 'variables/variables.data-00000-of
           -00001'),
9      os.path.join(vgg_path, 'variables/variables.index'),
10     os.path.join(vgg_path, 'saved_model.pb')]
11
12   missing_vgg_files = [vgg_file for vgg_file in vgg_files if not
         os.path.exists(vgg_file)]
13   if missing_vgg_files:
14     # Clean vgg dir
15     if os.path.exists(vgg_path):
```

```
16          shutil.rmtree(vgg_path)
17      os.makedirs(vgg_path)
18
19      # Download vgg
20      print('Downloading pre-trained vgg model...')
21      with DLProgress(unit='B', unit_scale=True, miniters=1) as
            pbar:
22          urlretrieve(
23              'https://s3-us-west-1.amazonaws.com/udacity-
                    selfdrivingcar/vgg.zip',
24              os.path.join(vgg_path, vgg_filename),
25              pbar.hook)
26
27      # Extract vgg
28      print('Extracting model...')
29      zip_ref = zipfile.ZipFile(os.path.join(vgg_path,
            vgg_filename), 'r')
30      zip_ref.extractall(data_dir)
31      zip_ref.close()
32
33      # Remove zip file to save space
34      os.remove(os.path.join(vgg_path, vgg_filename))
35  \label{codigo_load_VGG}
```

Una vez descargado y extraído el codificador, lo siguiente es obtener los tensores necesarios para la creación de la red FCN, que son:

- **image_ input:0** tensor que aloja la imagen de entrada.
- **keep_ prob:0** tensor que aloja la probabilidad para las capas de *dropout*.
- **layer3_ out:0** tensor que aloja la salida de la capa convolutiva número 3.
- **layer4_ out:0** tensor que aloja la salida de la capa convolutiva número 4.
- **layer7_ out:0** tensor que aloja la salida de la capa convolutiva número 7, que es la capa previa a la capa de clasificación.

Para extraer estos tensores lo primero es cargar el modelo usando la función *tf.saved_ model.loader.load(sess, [model_ tag], model_ path)* de Tensorflow (línea 31). Y, una vez cargado el modelo, lo siguiente es obtener los tensores usando la función *tf.get_ default_ graph().get_ tensor_ by_ name(tensor_ name)* (línea 32 - 38).

```
1  def load_vgg(sess, vgg_path):
2    ## Load Pretrained VGG Model into TensorFlow.
3    ## :param sess: TensorFlow Session
4    ## :param vgg_path: Path to vgg folder, containing "variables
         /" and "saved_model.pb"
5    ## :return: Tuple of Tensors from VGG model (image_input,
         keep_prob, layer3_out, layer4_out, layer7_out)
6
7    ## TODO: Implement function
8    ##   Use tf.saved_model.loader.load to load the model and
         weights
9
10   ## 1. Check if exists the .pb file for VGG16 on folder "VGG-
         model". In case it is not, then we need to transform it.##
11   ##
12   files, nfiles = helper.find_files_with_format(format = "pb",
         dir_to_search = vgg_path)
13   if nfiles == 0:
14     warnings.warn("No pretrained VGG16 model found. Downloading
           pretrained VGG16 model ...")
15     helper.maybe_download_pretrained_vgg(vgg_path)
16     print("Downloading process done !")
17   else:
18     print("Pretrained model already exists on dir", vgg_path)
19
20   ## 2. Define the names of the different fields to get from the
         model.                                ##
21   ##                                                          ##
22   vgg_tag             = 'vgg16'
23   vgg_input_tensor_name      = 'image_input:0'
24   vgg_keep_prob_tensor_name    = 'keep_prob:0'
25   vgg_layer3_out_tensor_name   = 'layer3_out:0'
26   vgg_layer4_out_tensor_name   = 'layer4_out:0'
27   vgg_layer7_out_tensor_name   = 'layer7_out:0'
28
29   ## 3. Get the fields from the model.
                        ##
30   ## The tag to introduce on the <c> tf.saved_model.loader.load
         () function is the model 'vgg16'
                                              ##
31   tf.saved_model.loader.load(sess, [vgg_tag], vgg_path)
32   vgg_input_tensor     = tf.get_default_graph().
         get_tensor_by_name(vgg_input_tensor_name)
33   vgg_keep_prob_tensor   = tf.get_default_graph().
         get_tensor_by_name(vgg_keep_prob_tensor_name)
34   vgg_layer3_out_tensor    = tf.get_default_graph().
         get_tensor_by_name(vgg_layer3_out_tensor_name)
35   vgg_layer4_out_tensor    = tf.get_default_graph().
```

```
    get_tensor_by_name(vgg_layer4_out_tensor_name)
36  vgg_layer7_out_tensor    = tf.get_default_graph().
    get_tensor_by_name(vgg_layer7_out_tensor_name)
37
38  return vgg_input_tensor, vgg_keep_prob_tensor,
    vgg_layer3_out_tensor, vgg_layer4_out_tensor,
    vgg_layer7_out_tensor
```

A.2. Creación de la arquitectura 1

La función que permite diseñar la arquitectura 1 (sin conexión codificador-decodificador) se encuentra en el fichero *main_ FCN_ sin_ conexion.py*, y se denomina *layers()*. Esta función define las nuevas capas de la arquitectura FCN a diseñar, siendo la capa convolutiva 1×1×2 generada en la línea 8, las dos capas deconvolutivas de tamaño 4×4×2 en las líneas 11 y 14, y la capa deconvolutiva de tamaño 16×16×2 en la línea 17.

```
1  def layers(vgg_layer3_out, vgg_layer4_out, vgg_layer7_out,
     num_classes):
2    ## Define the Kernel initializer
3    kernel_initializer = tf.random_normal_initializer(stddev=0.01)
4    ## Define the padding
5    padding = 'SAME'
6
7    ## Convolutional layer number 7
8    conv_layer_7 = tf.layers.conv2d(vgg_layer7_out, num_classes,
       1, 1, kernel_initializer = kernel_initializer)
9
10   ## Deconvolutional layer number 1
11   deconv_layer_1 = tf.layers.conv2d_transpose(conv_layer_7,
       num_classes, (4, 4), (2, 2), padding, kernel_initializer =
       kernel_initializer)
12
13   ## Deconvolutional layer number 2
14   deconv_layer_2 = tf.layers.conv2d_transpose(deconv_layer_1,
       num_classes, (4, 4), (2, 2), padding, kernel_initializer =
       kernel_initializer)
15
16   ## Deconvolutional layer number 3
17   deconv_layer_3 = tf.layers.conv2d_transpose(deconv_layer_2,
       num_classes, (16, 16), (8, 8), padding, kernel_initializer
       = kernel_initializer)
18
19   return deconv_layer_3
```

A.3. Creación de la arquitectura 2

La función que permite diseñar la arquitectura 2 (con conexión codificador-decodificador) se encuentra en el fichero *main.py*, y se denomina *layers()*. Esta función define las nuevas capas de la arquitectura FCN a diseñar, siendo las capas convolutivas 1×1×2 generadas en las líneas 19, 24 y 31, las dos capas deconvolutivas de tamaño 4×4×2 en las líneas 22 y 29, y la capa deconvolutiva de tamaño 16×16×2 en la línea 36. En cuanto a las adiciones entre capas, estas están definidas en las líneas 26 y 33 respectivamente.

```
1  ## Define the Kernel initializer
2    kernel_initializer = tf.random_normal_initializer(stddev=0.01)
3    ## Define the padding
4    padding = 'SAME'
5
6    ## Define the kernel_size and stride for the convolutional
         layers
7    conv_kernel_size = 1
8    conv_stride = 1
9
10   ## Define the kernel_size and stride for the deconvolutional
         layers
11   deconv_kernel_size = (4, 4)
12   deconv_stride = (2, 2)
13
14   ## Define the kernel_size and stride for the last
         deconvolutional layer
15   deconv_last_kernel_size = (16, 16)
16   deconv_last_stride = (8, 8)
17
18   ## Convolutional layer number 7
19   conv_layer_7 = tf.layers.conv2d(vgg_layer7_out, num_classes,
         1, 1, kernel_initializer = kernel_initializer)
20
21   ## Deconvolutional layer number 1
22   deconv_layer_1 = tf.layers.conv2d_transpose(conv_layer_7,
         num_classes, (4, 4), (2, 2), padding, kernel_initializer =
         kernel_initializer)
23   ## Convolutional layer number 4
24   conv_layer_4 = tf.layers.conv2d(vgg_layer4_out, num_classes,
         1, 1, kernel_initializer = kernel_initializer)
25   ## Skip union 1
26   skip_dec_1_layer_4 = tf.add(deconv_layer_1, conv_layer_4)
27
28   ## Deconvolutional layer number 2
29   deconv_layer_2 = tf.layers.conv2d_transpose(skip_dec_1_layer_4
         , num_classes, (4, 4), (2, 2), padding, kernel_initializer
```

```
    = kernel_initializer)
30  ## Convolutional layer number 3
31  conv_layer_3 = tf.layers.conv2d(vgg_layer3_out, num_classes,
        1, 1, kernel_initializer = kernel_initializer)
32  ## Skip union 2
33  skip_dec_2_layer_3 = tf.add(deconv_layer_2, conv_layer_3)
34
35  ## Deconvolutional layer number 3
36  deconv_layer_3 = tf.layers.conv2d_transpose(skip_dec_2_layer_3
        , num_classes, (16, 16), (8, 8), padding,
        kernel_initializer = kernel_initializer)
37
38  return deconv_layer_3
```

A.4. Entrenamiento de los modelos

A.4.1. Definición de la función de pérdidas

```
1  def optimize(nn_last_layer, correct_label, learning_rate,
       num_classes):
2  ##  Build the TensorFLow loss and optimizer operations.
3  ## :param nn_last_layer: TF Tensor of the last layer in the
       neural network
4  ## :param correct_label: TF Placeholder for the correct label
       image
5  ## :param learning_rate: TF Placeholder for the learning rate
6  ## :param num_classes: Number of classes to classify
7  ## :return: Tuple of (logits, train_op, cross_entropy_loss)
8  ## TODO: Implement function
9
10 ## ==========================================================
11 ## Define here your train_function
12 train_function = "ADAM"
13 ## ==========================================================
14
15 ## Logits: Reshape from 4D to 2D tensor
16 logits = tf.reshape(nn_last_layer, (-1, num_classes))
17
18 ## Labels: Reshape from 4D to 2D tensor
19 labels = tf.reshape(correct_label, (-1, num_classes))
20
21 ## Loss function --> optimize this
22 cross_entropy_loss = tf.reduce_mean( tf.nn.
       softmax_cross_entropy_with_logits( logits = logits, labels
       = labels ))
23
```

```
24
25    ## Training function definition
26    if train_function == "SGD":
27      train_op = tf.train.GradientDescentOptimizer(learning_rate).
          minimize(cross_entropy_loss)
28    elif train_function == "ADAM":
29      train_op = tf.train.AdamOptimizer(learning_rate).minimize(
          cross_entropy_loss)
30    else:
31      print("The trainning function [{}] is not available.".format
          (train_function))
32      exit(0)
33
34    ## Accuracy operation
35    y = tf.nn.softmax(logits)
36    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(
        labels, 1))
37    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.
        float32))
38
39    return logits, train_op, cross_entropy_loss, accuracy
```

A.4.2. Entrenar la red

Esta función está definida en los archivos *main.py* y *main_ FCN_ sin_ conexion.py*, y se denomina *train_ nn* (presente a continuación). En ella se define inicialmente si queremos guardar los pesos del modelo mediante el *flag save* (línea 22 - 23), así como se van obteniendo los diferentes *batches* usando la función *get_ batches_ fn()* (definida en los archivos *helper.py* y *helper_ FCN_ sin_ conexion.py*) que devuelve dos *arrays*: uno que contiene las imágenes y otro con sus etiquetas (línea 33). Tras obtener los *batches*, lo siguiente es lanzar la operación de entrenamiento y calcular el error asociado (línea 36), de manera que, en caso de que sea menor que el error obtenido en la iteración anterior y el *flag save* esté activo, se almacena el estado del modelo en el archivo *model.ckpt*, mostrándose en pantalla que se ha guardado dicho estado (líneas 40 - 53). Además, para una visualización del estado de la red se ha incluido un *print* que muestra la época (número de veces que se ha recorrido el dataset entero; valor definido por el usuario), la iteración (número de *batch*), el error y el tiempo que se ha tardado en ejecutar el entrenamiento para ese *batch* en concreto (línea 46). El entrenamiento termina cuando se halla alcanzado el número de épocas definida por el usuario, o en caso de recibir el comando *CTRL + C*.

```
1  def train_nn(sess, epochs, batch_size, get_batches_fn, train_op,
       cross_entropy_loss, input_image,
```

```
2         correct_label, keep_prob, learning_rate):
3     ##  Train neural network and print out the loss during
          training.
4     ## :param sess: TF Session
5     ## :param epochs: Number of epochs
6     ## :param batch_size: Batch size
7     ## :param get_batches_fn: Function to get batches of training
          data.  Call using get_batches_fn(batch_size)
8     ## :param train_op: TF Operation to train the neural network
9     ## :param cross_entropy_loss: TF Tensor for the amount of loss
10    ## :param input_image: TF Placeholder for input images
11    ## :param correct_label: TF Placeholder for label images
12    ## :param keep_prob: TF Placeholder for dropout keep
          probability
13    ## :param learning_rate: TF Placeholder for learning rate
14    ## TODO: Implement function
15
16    ## Import needed packages
17    import matplotlib.pyplot as plt
18    import numpy as np
19
20    # Create the model saver
21    global save
22    if save:
23      saver = tf.train.Saver()
24
25    ## min_loss and loss vector definitions
26    min_loss = 2**34
27    tloss = np.array([])
28
29    ## Training loop
30    try:
31      for epoch in range(epochs):
32        it = 0
33        for image, gt_image in get_batches_fn(batch_size):
34          start = time()
35          ## Train and get the loss value
36          _, train_loss = sess.run([train_op, cross_entropy_loss],
                feed_dict = {input_image: image,
37                                        correct_label: gt_image,
38                                        keep_prob: 0.5,
39                                        learning_rate: 0.001})
40          stop = time()
41
42          ## Add the training loss to the vector
43          tloss = np.array(tloss, train_loss)
44
45          ## Print the loss
46          print("[Epoch.", epoch, "] [It.", it, "] Loss:",
```

```
            train_loss, " -->", str(stop - start), "s")
47
48          ## Update the min_loss
49          if save:
50            if train_loss < min_loss:
51              min_loss = train_loss
52              saver.save(sess, 'model.ckpt')
53              print ("Checkpoint saved")
54
55          ## Increment the iterator
56          it += 1
57    except KeyboardInterrupt:
58      exit(0)
```

A.5. Test del modelo

Para testear la red se han definido las funciones *save_ inference_ samples()* y *gen_ test_ output()* en los archivos *main.py* y *main_ FCN_ sin_ conexion.py*. La primera de ellas permite crear un directorio (línea 3 - 6), definido por *runs_ dir*), donde almacenar las imágenes resultantes de ejecutar el modelo (línea 10 - 13), y la segunda de ellas permite cargar las imágenes de test (línea 12 - 14), lanzar el modelo (línea 15 - 17), determinar a qué clase pertence cada píxel (línea 18 - 19) y colorear de verde la imagen original aquellos píxeles que han sido clasificados como "carretera" (línea 20 - 23), mediante el uso la matriz de clasificación.

```
1  def save_inference_samples(runs_dir, data_dir, sess, image_shape
       , logits, keep_prob, input_image, correct_label, accuracy_op)
       :
2    # Make folder for current run
3    output_dir = os.path.join(runs_dir, str(time.time()))
4    if os.path.exists(output_dir):
5      shutil.rmtree(output_dir)
6    os.makedirs(output_dir)
7
8    # Run NN on test images and save them to HD
9    print('Training Finished. Saving test images to: {}'.format(
         output_dir))
10   image_outputs = gen_test_output(
11     sess, logits, keep_prob, input_image, os.path.join(data_dir,
           'data_road/testing_2'), image_shape, correct_label,
         accuracy_op)
12   for name, image in image_outputs:
13     scipy.misc.imsave(os.path.join(output_dir, name), image)
```

```
1  def gen_test_output(sess, logits, keep_prob, image_pl,
       data_folder, image_shape, correct_label, accuracy_op):
2
3    ## Generate test output using the test images
4    # :param sess: TF session
5    # :param logits: TF Tensor for the logits
6    # :param keep_prob: TF Placeholder for the dropout keep
         robability
7    # :param image_pl: TF Placeholder for the image placeholder
8    # :param data_folder: Path to the folder that contains the
         datasets
9    # :param image_shape: Tuple - Shape of image
10   # :return: Output for for each test image
11   file = open("/Volumes/MACINTOSH H/Users/danielhormigoruiz/
         AnacondaProjects/TFM/Semantic Segmentation/
         resultados_no_conexion.txt", 'w')
12   get_batches_fn = gen_batch_function(data_folder, image_shape)
13   #for image_file in glob(os.path.join(data_folder, 'image_2',
         '*.png')):
14   for image, gt_image, name in get_batches_fn(1):
15     #image = scipy.misc.imresize(scipy.misc.imread(image_file),
           image_shape)
16     print("Running Softmax...")
17     im_softmax = sess.run(
18       [tf.nn.softmax(logits)],
19       {keep_prob: 1.0, image_pl: [image[0]]})
20
21     im_softmax = im_softmax[0][:, 1].reshape(image_shape[0],
           image_shape[1])
22     segmentation = (im_softmax > 0.5).reshape(image_shape[0],
           image_shape[1], 1)
23     print("Softmax done :)")
24     print("Running accuracy op with: Image name -> ", name, "
           Image shape -> ", np.shape(image), " GT shape -> ", np.
           shape(gt_image))
25     accuracy = sess.run(accuracy_op, feed_dict = {image_pl:
           image, correct_label: gt_image, keep_prob: 1.0})
26
27     msg = "Image: " + name[0] + " Accuracy: " + str(accuracy) +
           "\n"
28     print(msg)
29     file.write(msg)
30
31     mask = np.dot(segmentation, np.array([[0, 255, 0, 127]]))
32     mask = scipy.misc.toimage(mask, mode="RGBA")
33     street_im = scipy.misc.toimage(image[0])
34     #street_im = image[0]
35     street_im.paste(mask, box=None, mask=mask)
```

```
36
37      #yield os.path.basename(image_file), np.array(street_im)
38      yield name[0], np.array(street_im)
```