Deusto
Facultad de Ingeniería
Ingeniaritza Fakultatea

Máster Universitario en Ingeniería Informática

Informatikako Ingeniaritzako Unibertsitate Masterra

Proyecto fin de máster

Master amaierako proiektua

Desarrollo de sistema de detección de líneas en tiempo real mediante la arquitectura LaneNet y Keras

Unai López Ansoleaga

Director: Antonio David Masegosa Arredondo

Bilbao, enero de 2021

Resumen

Este proyecto de fin de máster consiste en utilizar la arquitectura LaneNet descrita en el paper *Towards End-to-End Lane Detection: an Instance Segmentation Approach* [1] para la detección de carriles en tiempo real. Esta arquitectura aplica el *deep learning* y el *clustering* para lograr la *instance segmentation* de los carriles en tiempo real. Consiste en un *autoencoder* con una parte de codificación común y una parte de decodificación dividida en dos ramas: la rama de la segmentación binaria y la rama del *embedding*. Este modelo ha sido entrenando sobre el TuSimple *dataset* que es un conjunto de datos que se suele utilizar de *benchmark* en los problemas de percepción para vehículos autónomos y ha sido desarrollado en Python con Keras utilizando la computación en la nube de Google Colaboratory,

Los resultados de la rama de segmentación binaria han sido muy buenos logrando así resultados parecidos a los del *paper* [1]. Sin embargo, la rama del *embedding*, no han podido ser replicados a pesar de reproducir tanto la arquitectura como los hiperparámetros del *paper*. Esto se debe a una falta importante de detalle a la hora de describir dicha arquitectura.

Además, todo este proceso ha servido para detectar errores importantes en implementaciones de esta arquitectura disponibles en github muy bien valoradas, pero que no son fieles al contenido del *paper*.

Descriptores

- *Deep Learning*
- *Computer Vision*
- *Autonomous Driving*

ÍNDICE DE CONTENIDOS

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN

En los últimos 6 años, los vehículos autónomos han atraído una inmensa cantidad de capital por parte de inversores y grandes compañías tecnológicas y automovilísticas. Esto ha generado una carrera y una feroz competencia para buscar una tecnología que permita la conducción autónoma total que, a día de hoy, sigue muy disputada.

1.1 HISTORIA

Desde que se inventaron los vehículos, la tecnología se ha ido puliendo década tras década convirtiéndolos así en vehículos con mayor autonomía, más seguros, más fáciles de manejar y más inteligentes. Esta tendencia se ha seguido cumpliendo durante décadas pero fue en el año 2004 cuando se comenzó a pensar en alcanzar otro nivel que parecía imposible para la época. En 2004, el DARPA [2], Defense Advanced Research Projects Agency, organizó el primer DARPA Grand Challenge [3]. En esta competición, vehículos autónomos se enfrentaron en una carrera de 240 km a través del desierto de Mojave de los Estados Unidos atravesándolo desde Barstow (California) hasta pasada la frontera de California con Nevada. A pesar de que había muchos intereses puestos en esa competición por su premio de un millón de dólares, el vehículo qué más lejos llegó solo recorrió 11.78 km y por lo tanto se consideró que la competición había sido un fracaso. A pesar de ello, el siguiente año se volvió a realizar y esta vez cinco vehículos lograron terminar la carrera ganando un total de tres millones y medio de dólares en premios.

Durante el transcurso de los años y de las DARPA Grand Challenges, se ha ido investigando sobre el uso de algoritmos, RADAR, cámaras y LiDAR para lograr que los vehículos autónomos tuvieran una mejor percepción del entorno y así lograr detectar y evitar obstáculos. Además, gracias a la ley de Moore el poder computacional ha continuado aumentando y abaratándose dando pie a nuevos métodos como los algoritmos de *deep learning* para poder aplicarlos a la problemática de la conducción autónoma.

Esta reducción de los costes y mejora del software y del hardware, ha dado pie a diferentes empresas a abordar esta problemática y a tratar de crear su propio vehículo autónomo. Como por ejemplo, Waymo, Tesla, General Motors, Apple, Uber, Mercedes-Benz, etc.

A día de hoy, a pesar de que la carrera tecnológica por lograr el primer vehículo con autonomía total sigue en marcha, empresas como Tesla o Waymo llevan la delantera y actualmente tienen flotas de vehículos desplegados en las calles con capacidades muy cercanas a la autonomía total.

1.2 ESTADO DEL ARTE EN LA DETECCIÓN DE CARRILES

Hoy en día, los coches totalmente autónomos son el principal foco de atención de la investigación en la informática y la robótica, tanto a nivel académico como industrial. El objetivo de estas investigaciones es llegar a una comprensión plena del entorno que rodea al automóvil mediante el uso de diferentes sensores y módulos de percepción. La detección de carriles basada en cámaras es un paso importante para lograr esta percepción total del entorno, ya que permite al automóvil posicionarse adecuadamente dentro de los carriles de la carretera. Además, la detección de los carriles de la carretera también es crucial para una decisión posterior de salida del carril o de planificación de trayectoria.

Un factor clave para lograr coches totalmente autónomos es realizar una detección de carril precisa basada en cámaras en tiempo real y para esto se ha hecho muchísima investigación y se han logrado grandes avances. Las primeras aproximaciones para solucionar esta problemática, fueron los métodos tradicionales de detección de carril como [4], [5], [6], [7], [8] y [9]. Estos métodos tradicionales son una combinación de características artesanales y heurísticas altamente especializadas para identificar los segmentos de los carriles. Unas de las elecciones más populares entre los métodos artesanales son las características basadas en colores [10], el tensor de estructura [11], el filtro de barra [12] y las características de cresta [13], que normalmente se combinan con una transformada de Hough [14], [15] y filtros de partículas Kalman [16], [17], [12].

Después de identificar los segmentos del carril, se emplean técnicas de post-procesamiento para filtrar las detecciones erróneas y agrupar los segmentos para formar las líneas finales. Sin embargo, por lo general, estos enfoques tradicionales son propensos a problemas de robustez debido a las variaciones de la escena de la carretera que no pueden ser modeladas fácilmente por esos sistemas basados en reglas.

En los últimos años, los detectores artesanales han sido reemplazados por redes neuronales profundas. Estas redes hacen segmentaciones de los carriles a nivel de píxel. En [18] se utiliza un descriptor de características de jerarquía de píxeles para modelar la información contextual y un algoritmo de *boosting* para seleccionar características contextuales relevantes para la detección de marcas de carril. De una manera similar, en [19] se combina una red neuronal convolucional (CNN) con el algoritmo RANSAC para detectar carriles a partir de imágenes de bordes. Se ha de mencionar, que en este último método, la CNN se utiliza principalmente para mejorar la imagen y sólo si la escena de la carretera es compleja, por ejemplo, incluye árboles en los bordes de la carretera, vallas o intersecciones. En [20] se muestra como los modelos de CNN existentes pueden utilizarse para aplicaciones de conducción en autopistas, entre las que se encuentra una red neuronal convolucional de extremo a extremo que realiza la detección y clasificación de carriles. En [21] se introduce la Dual-View CNN (DVCNN) que utiliza una vista frontal y una vista superior de imágenes simultáneamente para excluir las detecciones

falsas y eliminar las estructuras sin forma de línea. En [22] se propone el uso de una red convolucional profunda multitarea que se centra en encontrar atributos geométricos de los carriles, como la ubicación y la orientación, junto con una red neuronal recurrente (RNN) que detecta los carriles. Más recientemente, en [23] se muestra cómo una red multitarea puede manejar conjuntamente la detección y el reconocimiento de carriles y marcas viales en condiciones de clima adverso y de poca iluminación. Además de la capacidad de las redes mencionadas de segmentar mejor las marcas de la carretera [20], su gran campo receptivo les permite también estimar los carriles incluso en los casos en que no hay marcas en la imagen. Sin embargo, en una etapa final, las segmentaciones de carril generadas todavía tienen que ser desenmarañadas en las diferentes instancias del carril. Para hacer frente a este problema, algunos enfoques han aplicado técnicas de post-procesamiento que se basan de nuevo en la heurística, generalmente guiada por las propiedades geométricas, como se hace en [19], [25] por ejemplo. Como se ha explicado anteriormente, estos métodos heurísticos son costosos desde el punto de vista computacional y propensos a problemas de robustez debido a las variaciones de la escena de la carretera. Otra línea de trabajo [26] plantea el problema de la detección de carriles como un problema de segmentación de clases múltiples, en el que cada carril forma su propia clase. De esta manera, la salida de la red contiene mapas binarios desenmarañados para cada carril y puede ser entrenada de una manera de extremo a extremo. A pesar de sus ventajas, este método se limita a detectar sólo un número predefinido y fijo de carriles. Además, como cada carril tiene una clase designada, no puede hacer frente a los cambios de carril.

En el paper *Towards End-to-End Lane Detection: an Instance Segmentation Approach* [1], que es en el que se va a basar este proyecto de fin de master, tratan de ir más allá de las limitaciones mencionadas y se propone plantear el problema de la detección de carriles como un problema de *instance segmentation*, en el que cada carril forma su propia instancia dentro de la clase de carril. Inspirado por el éxito de las redes de segmentación semántica [26],[27], [28], [29] y las tareas de segmentación de instancias [30], [31],[32], [33], [34], [35], se diseña una red ramificada y multitarea, como [27] para la segmentación de instancias de carril, que consiste en una rama de segmentación de carril y una rama de *embedding* de carril que puede ser entrenada de extremo a extremo. La rama de segmentación de carril tiene dos clases de salida, fondo o carril, mientras que la rama de *embedding* de carril desenreda aún más los píxeles de carril segmentados en diferentes instancias de carril. Al dividir el problema de detección de carriles en las dos tareas mencionadas, podemos utilizar plenamente la potencia de la rama de segmentación de carriles sin tener que asignar diferentes clases a los diferentes carriles. En cambio, la rama de *embedding* de carriles, que se entrena utilizando una función de pérdida de *clustering*, asigna una identificación de carril a cada píxel de la rama de segmentación de carriles, ignorando los píxeles de fondo. Al hacerlo, se alivia el problema de los cambios de carril y se puede manejar un número variable de carriles, a diferencia de [24].

2. ¿QUÉ ES EL COMPUTER VISION?

El *computer vision* o visión artificial en español, es un campo de estudio dentro de la inteligencia artificial que busca el desarrollo de técnicas para ayudar a los ordenadores a "ver" y comprender el contenido digital de imágenes y videos [36].

El problema de la visión por ordenador parece sencillo porque las personas lo resolvemos de una manera trivial. No obstante, sigue siendo en gran medida un problema sin resolver debido a nuestra limitada comprensión de la visión biológica y a la complejidad de la percepción en un mundo físico tan dinámico como el nuestro [36].

El algoritmo más utilizado para las aplicaciones que hacen uso de visión artificial, son las redes neuronales convolucionales o CNNs. Estos algoritmos de *deep learning* han sido utilizados para el reconocimiento de imágenes desde finales de la década de los 80 y surgieron a partir de un estudio del cortex visual del cerebro humano. [37]

2.1 REDES NEURONALES CONVOLUCIONALES

Las redes neuronales convolucionales o CNNs son un tipo de modelo de red neuronal que permite extraer representaciones de mayores dimensiones de una imagen. A diferencia de los métodos clásicos de reconocimiento de imágenes, en los que un programador debe de indicar todas las reglas de manera manual, las CNNs toman los pixeles de una imagen, entrenan un modelo y extraen características o patrones de la imagen logrando una mejor clasificación [38]. Además, la propia arquitectura de las CNNs, a diferencia de las redes neuronales clásicas, permite detectar patrones en las imágenes independientemente de su localización en las mismas. Es decir, el modelo detectará el patrón de las orejas de un gato independientemente de su posición en la imagen.

2.1.1 CAPA DE CONVOLUCIÓN

El bloque de construcción más importante de una red neuronal convolucional es la capa de convolución o *convolutional layer* en inglés. Estas capas se basan en una operación matemática llamada convolución:

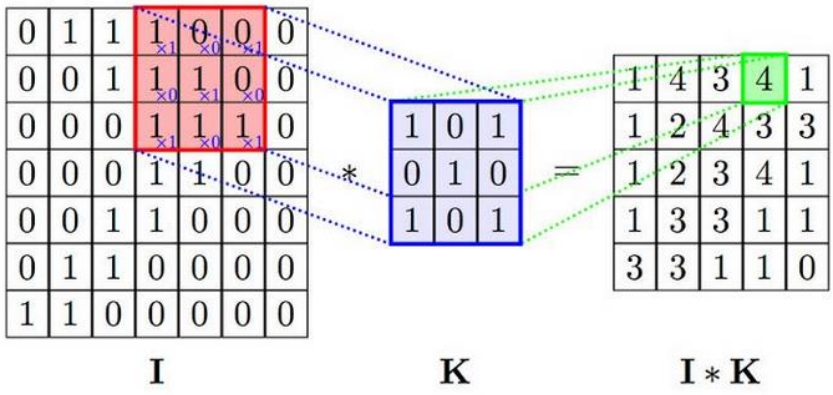


Ilustración 1: Operación de convolución

Fuente: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution
Autor: Petar Veličković

Como podemos observar en la ilustración 1, en una convolución una ventana barre la imagen inicial aplicándole un filtro. Con este filtro se calcula el producto lo que hace que se resalten las características más relevantes de la imagen [38]. La operación resaltada en color de la ilustración 1 la podemos definir así, donde * indica una convolución:

$$\begin{bmatrix} I_{14} & I_{15} & I_{16} \\ I_{24} & I_{25} & I_{26} \\ I_{34} & I_{35} & I_{36} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} = I_{14} \times K_{11} + I_{15} \times K_{12} + I_{16} \times K_{13} +$$

$$I_{24} \times K_{21} + I_{25} \times K_{22} + I_{26} \times K_{23} + I_{34} \times K_{31} + I_{35} \times K_{32} + I_{36} \times K_{33}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = 1 \times 1 + 0 \times 0 + 0 \times 1 + 1 \times 0 + 1 \times 1 + 0 + 0 +$$

$$1 \times 1 + 1 \times 0 + 1 \times 1 = 4$$

Con esta operación matemática, se puede detectar una característica o patrón concreto en una imagen y producir mapas de características los cuales enfatizan las características o patrones más importantes. Estos mapas de características siempre irán cambiando dependiendo de los filtros o *kernels* que cambiarán automáticamente acorde al descenso del gradiente para minimizar la función de pérdida de la predicción [38].

El potencial de este tipo de redes reside en su procesamiento en cascada. Es decir, cuantos más filtros se utilicen, más características extraerá la CNN permitiendo así que se encuentren más características pero con un costo mayor en el tiempo de entrenamiento [38]:

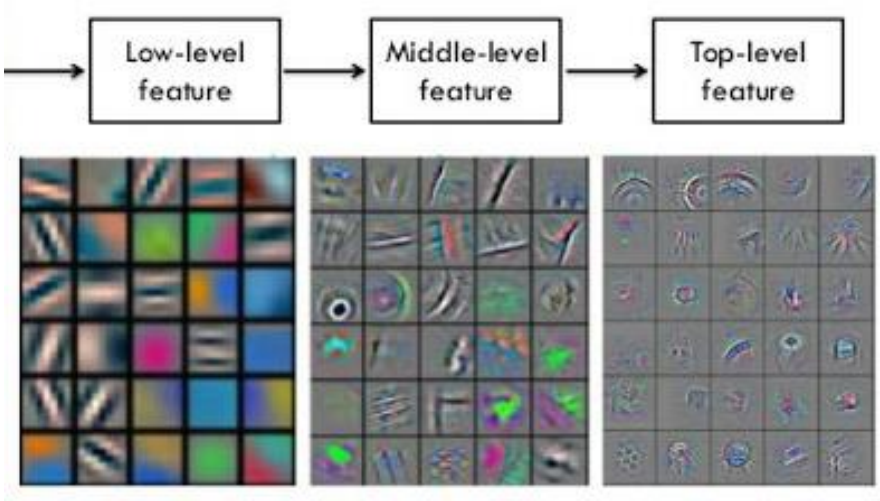


Ilustración 2: Mapas de características de CNN entrenada sobre ImageNet

Fuente: https://arxiv.org/pdf/1311.2901.pdf
Autor: Matthew D. Zeiler & Rob Fergus

Pero, ¿Qué sucede cuando la imagen no tiene los suficientes bloques adyacentes para deslizar el filtro sobre ella? ¿Deberían de ser eliminados estos bloques?
Cuando los bloques adyacentes son insuficientes para ajustar el filtro, normalmente no se eliminan porque se pierde información del input. Para estos casos se insertan 0s añadiendo así los bloques adyacentes que necesitemos [38]. Este proceso se llama *padding*.

2.1.2 CAPA DE POOLING

Las CNNs utilizan el *pooling* para reemplazar la salida de una convolución con una simplificación de los datos. Esto reduce el tamaño del *input* y el tiempo de procesamiento [38].

Estas operaciones reciben dos hiperparametros o *hyperparameters* en inglés: *stride* y tamaño.
El *stride* determina cuantos elementos debe saltarse la ventana al barrer las matrices mientras que el tamaño determina el tamaño de la ventana [38].

Existen varios tipos de *pooling*, como por ejemplo:

- ***Max pooling.*** Se queda con el elemento más grande de la ventana.
- ***Min pooling.*** Se queda con el elemento más pequeño de la ventana.
- ***Average pooling.*** Se queda con la media de todos los elementos de la ventana.

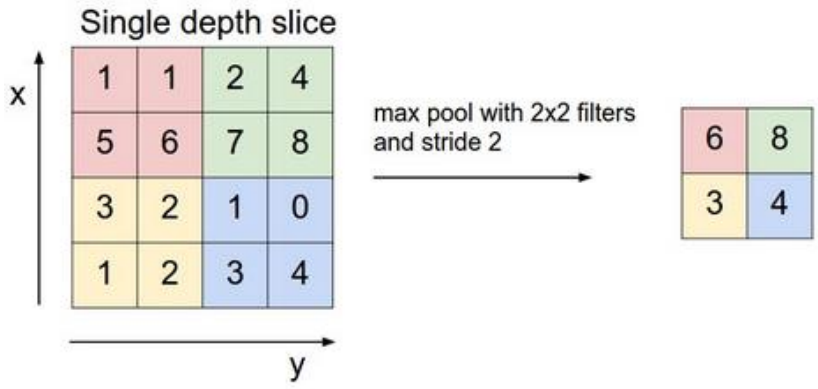


Ilustración 3: Operación de max pooling

Fuente: https://cs231n.github.io/convolutional-networks/
Autor: Stanford University

La operación *max pooling* de la ilustración 3 se podría definir de la siguiente manera:

$$\max\_pooling(filter\ size, stride, input) =$$

$$\max\_pooling\left(2x2, 2, \begin{bmatrix} Input_{11} & Input_{12} & Input_{13} & Input_{14} \\ Input_{21} & Input_{22} & Input_{23} & Input_{24} \\ Input_{31} & Input_{32} & Input_{33} & Input_{34} \\ Input_{41} & Input_{42} & Input_{43} & Input_{44} \end{bmatrix}\right) =$$

$$\begin{bmatrix} \max\{Input_{11}, Input_{12}, Input_{21}, Input_{22}\} & \max\{Input_{13}, Input_{14}, Input_{23}, Input_{24}\} \\ \max\{Input_{31}, Input_{32}, Input_{41}, Input_{42}\} & \max\{Input_{33}, Input_{34}, Input_{43}, Input_{44}\} \end{bmatrix}$$

$$\max\_pooling\left(2x2, 2, \begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}\right) =$$

$$\begin{bmatrix} \max\{1,1,5,6\} & \max\{2,4,7,8\} \\ \max\{3,2,1,2\} & \max\{1,0,3,4\} \end{bmatrix} =$$

$$\begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

2.1.3 FUNCIONES DE ACTIVACIÓN

Otro elemento fundamental para el correcto funcionamiento de cualquier red neuronal son las funciones de activación. Estas funciones de activación determinan si una neurona debe activarse o no [39]. Para saber si una neurona debe activarse, una neurona ha de calcular una *weighted sum* o suma balanceada sumándole un sesgo o *bias* y después pasarle el resultado a una función de activación:

$$Z = \sum (weight * input) + bias$$

$$A = \sigma(Z)$$

2.1.3.1 SIGMOIDE

$$\sigma(Z) = \frac{1}{1 + e^{-z}}$$

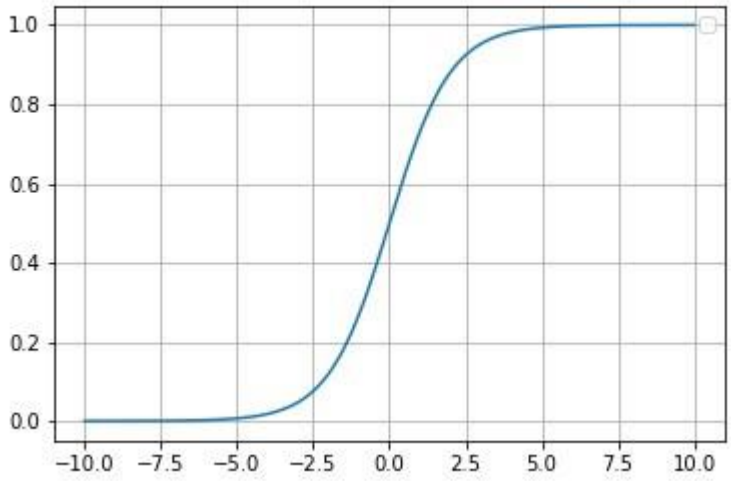


Ilustración 4: Función de activación sigmoide

Fuente: https://towardsdatascience.com/what-is-activation-function-1464a629cdca

- **Ventajas**
  - Funciona bien en problemas de clasificación binaria porque convierte todos los valores en el rango $(-\infty, \infty)$ a valores en el rango $(0,1)$ [39].
- **Desventajas**
  - No funciona en problemas de clasificación de multiples etiquetas (*multilabel classification*) [40].
  - La derivada para el cálculo del gradiente siempre es 0, por lo que es imposible actualizar los pesos [40].

2.1.3.2 TANH

La función de activación tanh es similar a la sigmoide. Sin embargo, la tanh es simétrica en 0, convierte todos los valores en el rango $(-\infty, \infty)$ a valores en el rango $(-1,1)$ y las derivadas son más pronunciadas por lo que entrena más rápidamente. Por este motivo, la función tanh suele ser mejor utilizarla en capas intermedias pero dado que es una función computacionalmente bastante costosa y suele generar problemas con el desvanecimiento del gradiente, se suele utilizar la función ReLU en su lugar [40].

$$\sigma(Z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

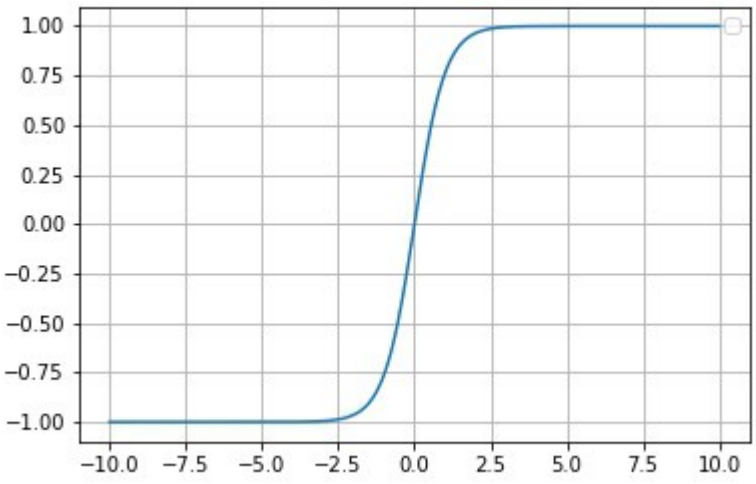


Ilustración 5: Función de activación tanh

Fuente: https://towardsdatascience.com/what-is-activation-function-1464a629cdca

- **Ventajas**
  - Funciona bien en capas intermedias por su simetría en el valor 0 y porque convierte todos los valores en el rango $(-\infty, \infty)$ a valores en el rango $(-1,1)$ [40].
  - El gradiente es más pronunciado que en la función sigmoide porque las derivadas son más pronunciadas [40].
- **Desventajas**
  - Al igual que la función sigmoide, la función de activación tanh sigue teniendo el problema de desvanecimiento del gradiente [40].

2.1.3.3 RELU

$$\sigma(Z) = \begin{cases} Z, Z > 0 \\ 0, Otherwise \end{cases}$$

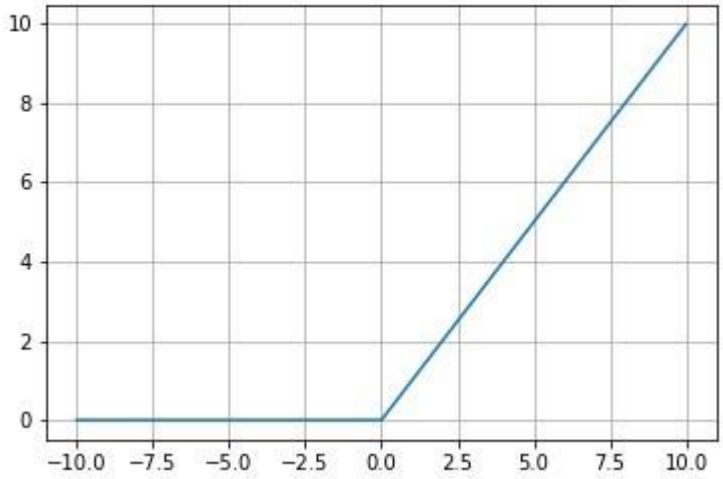


Ilustración 6: Función de activación ReLU

Fuente: https://towardsdatascience.com/what-is-activation-function-1464a629cdca

- **Ventajas**
  - Es fácil de implementar y muy ligero computacionalmente [40].
  - La optimización es fácil cuando la función de activación es lineal [40].
  - Es la función de activación más utilizada en las redes neuronales [40].
- **Desventajas**
  - Mayor facilidad de que la salida de las neuronas sea 0. Si la salida es 0, no hay gradiente y las neuronas no están activadas, por lo tanto, esto puede generar un deficiente rendimiento del modelo [40].
  - No es adecuada para las redes neuronales recurrentes o RNNs [40].

2.1.3.4 SOFTMAX

La función activación softmax es diferente a las demás. Esta función de activación computa una distribución de probabilidades por lo que se suele utilizar en la última capa de las redes neuronales cuando queremos predecir múltiples clases [40].

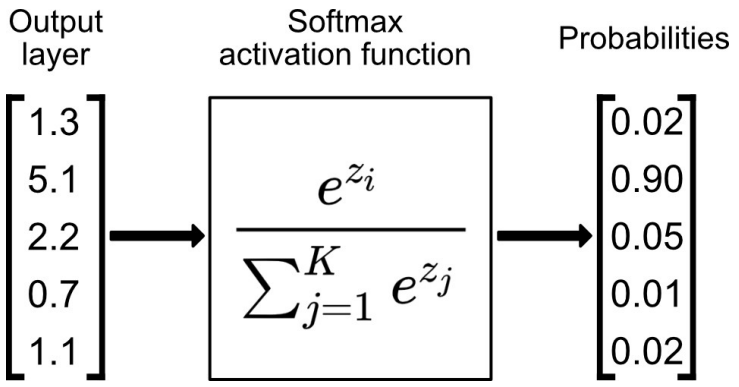


Ilustración 7: Ejemplo de funcionamiento de la función de activación softmax

Fuente: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60

Autor: Dario Radečić

2.1.4 ARQUITECTURA COMPLETA DE UNA RED NEURONAL CONVOLUCIONAL

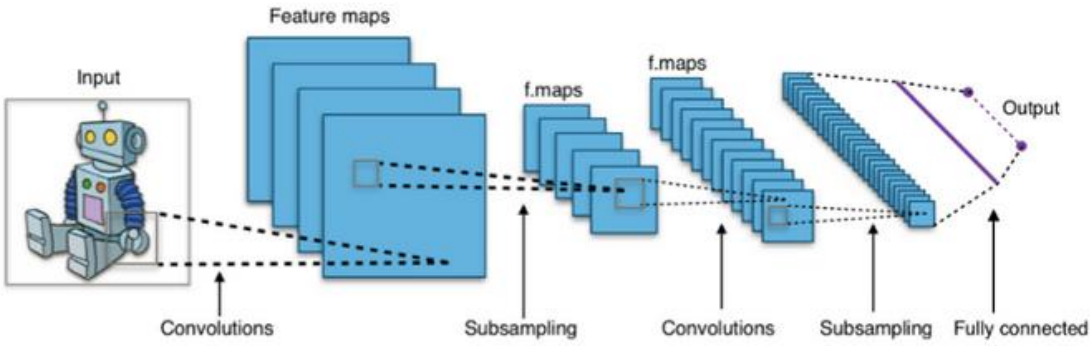


Ilustración 8: Arquitectura completa de una red neuronal convolucional

Fuente: https://towardsdatascience.com/understanding-cnn-convolutional-neural-network-69fd626ee7d4

Una vez entendidos todos los conceptos básicos que forman una red neuronal convolucional, podemos observar una arquitectura completa en la ilustración 8. Esta red está compuesta de dos capas convolucionales seguidas de sus capas de *pooling* y finalmente podemos ver como una capa *fully connected* reduce la dimensionalidad de todos los mapas de características generados por los procesos convolutivos a un vector unidimensional. Este vector, contiene todas las características o patrones más relevantes

de la imagen y si a este le aplicamos la función de activación softmax, nos devolverá una distribución de probabilidades. Para comprender mejor esta última fase en la que se hace la predicción, voy a inventarme unos valores para explicar este proceso de una manera más intuitiva:

$$Clases = \{\text{Dinosaurio}, Perro, Gato, Robot, Pato, Oso\}$$

$$Vector\ unidimensional\ de\ la\ capa\ fully\ connected = [1.4, 8.9, -11.1, 2.1, 3.0, 27.1]$$

$$Distribución\ de\ probabilidades\ después\ del\ softmax = [0.01, 0.02, 0.01\ , 0.8, 0.06, 0.1]$$

$$Confianza\ en\ las\ predicciones: \begin{cases} Confianza\ clase\ Dinosaurio: 1\% \\ Confianza\ clase\ Perro: 2\% \\ Confianza\ clase\ Gato: 1\% \\ Confianza\ clase\ Robot: 80\% \\ Confianza\ clase\ Pato: 6\% \\ Confianza\ clase\ Oso: 10\% \end{cases}$$

Según estos datos, esta red neuronal convolucional predeciría con un 80% de confianza que el input es un Robot

3. DEFINICIÓN DEL PROYECTO

3.1 OBJETIVOS

Para lograr un proyecto satisfactorio, se han planteado varios objetivos:

- Lograr un modelo basado en el paper de LaneNet [1] con unos resultados similares.
- Aportar una implementación en la API de alto nivel de Tensorflow, Keras. En la actualidad, no existe ningún repositorio disponible en internet con una implementación con esta API.
- Una vez finalizada la implementación del modelo, testearla usando *benchmarks* de referencia en el área.

3.2 ALCANCE

- Leer y comprender el paper de LaneNet [1] para poder hacer una implementación desde 0 y ajustar los hiperparámetros para lograr los mejores resultados posibles.
- Implementar el modelo completo utilizando Keras en todos los apartados en los que sea posible.
- Analizar el rendimiento del modelo mediante métricas y *datasets* de referencia en el área.

3.3 INNOVACIONES TECNOLÓGICAS

Como innovación tecnológica se propone crear una implementación de LaneNet en la API de alto nivel de Tensorflow, Keras. Esto es una innovación porque actualmente no existe ningún repositorio en internet que tenga una implementación con esta API.

Este modelo basado en el *deep learning*, es muy relevante en el campo de la conducción autónoma pero también para diferentes áreas en los que la detección de líneas sea relevante como por ejemplo en la detección de líneas de carril en los aeropuertos, aparcamientos, etc. La facilidad de este modelo para generalizar a partir de imágenes de entrenamiento lo convierte en una elección interesante para todas estas áreas en lugar de confiar en métodos heurísticos que no son capaces de generalizar ni son tan robustos como LaneNet.

4. DESCRIPCIÓN DEL PROYECTO

El proyecto se basa en crear una implementación del modelo LaneNet mencionado por primera vez en el paper del 2018 *Towards End-to-End Lane Detection: an Instance Segmentation Approach* [1]. Como no existe ninguna implementación en Keras, la API de alto nivel de Tensorflow, LaneNet se implementará con este framework.

4.1 LANENET

LaneNet es una arquitectura que logra resultados del estado del arte en la detección de carriles con una precisión del 96.4% y un F1 score de 94.80 en el *dataset* TuSimple. Estas métricas sitúan a LaneNet en la posición 6 y 13 respectivamente del ranking global sin apenas diferencia alguna con los valores del modelo que lidera el ranking en el *benchmark* TuSimple. [41]

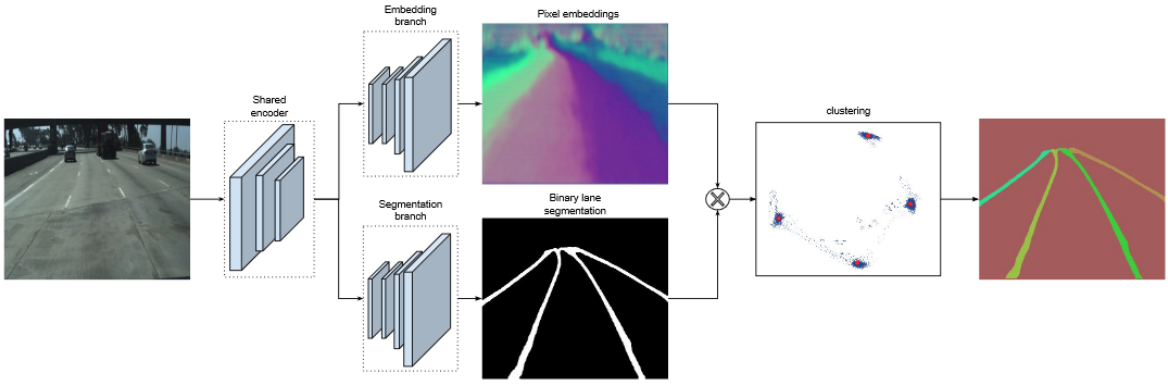


Ilustración 9: Arquitectura LaneNet

Fuente: https://arxiv.org/pdf/1802.05591.pdf
Autor: Davy Neven, Bert De Brabandere, Stamatios Georgoulis, Marc Proesmans & Luc Van Gool

En la ilustración 9, se puede ver la arquitectura de LaneNet. Está compuesta de dos ramas. La rama de segmentación (rama inferior) está entrenada para producir una máscara de carril binario y la rama de *embedding* (rama superior) genera un *embedding* N-dimensional por cada píxel de carril, de modo que los *embeddings* del mismo carril están cerca y las de diferentes carriles están lejos.

Para simplificar, se muestra un *embedding* bidimensional por píxel. Todos los píxeles se visualizan en un mapa de color en los ejes X e Y. Dentro de estos píxeles se diferencian los que pertenecen a un carril y los que no. Después de enmascarar los píxeles de fondo usando el mapa de segmentación binario de la rama de segmentación, los *embeddings* de carril (puntos azules) se clusterizan y se asignan a sus centros de *cluster* (puntos rojos).

4.1.1 BINARY SEGMENTATION

La rama de segmentación binaria, se basa en la arquitectura de E-Net [43]. Esta arquitectura basada en los principios encode-decode, es rápida y compacta, por lo que permite realizar la segmentación semántica en tiempo real. Como se ha dicho previamente, la detección en tiempo real es fundamental para lograr un vehículo de conducción autónoma dado que unos pocos milisegundos de retardo en una predicción pueden suponer el éxito o el fracaso de estos vehículos.

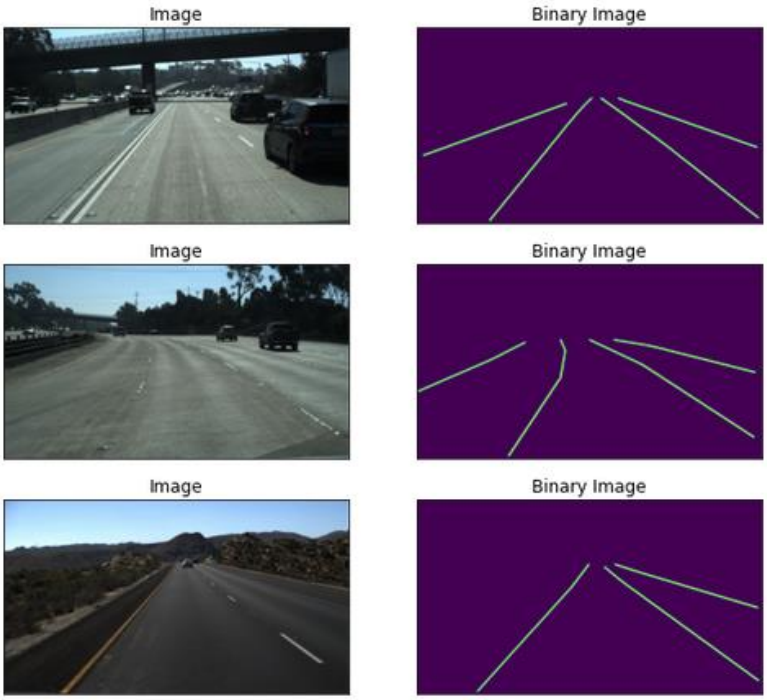


Ilustración 10: Input y output óptimo de la rama de segmentación binaria de LaneNet

En la ilustración 10, se pueden observar tanto las imágenes originales de input de LaneNet, como el output óptimo de la rama de segmentación binaria. En este contexto, óptimo se refiere a que son imágenes anotadas a mano, también conocido como *ground truth*.

Esta rama de LaneNet, tendrá como objetivo predecir las líneas de los carriles devolviendo una máscara binaria a partir de imágenes de carreteras reales.

4.1.2 EMBEDDING

La rama de *embedding*, también se basa en la arquitectura E-Net pero a diferencia de la rama de segmentación binaria, esta rama es entrenada para crear una representación N-dimensional. En el paper [1] utilizan 4 dimensiones o canales por lo que el output en este

caso es una representación de la misma altura y anchura que la imagen original pero con 4 canales de color.

Esta representación de mayores dimensiones facilita utilizar el *clustering* para la posterior segmentación de instancias.

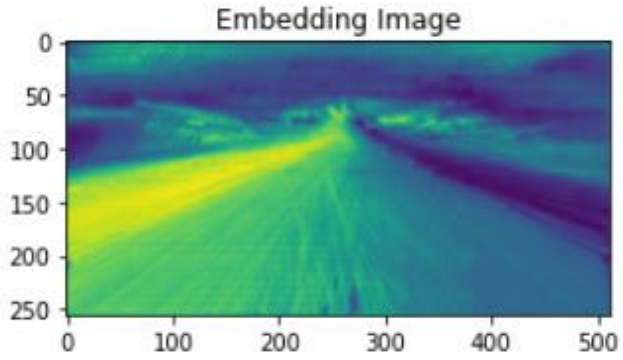


Ilustración 11: Ejemplo de *embedding*

Este es un ejemplo de lo que podemos esperar de un *embedding*. En esta ilustración se puede observar cómo hay valores diferentes en las zonas en las que se encuentran las líneas de carril. Estos diferentes valores facilitarán hacer un *clustering* una vez aplicada la máscara binaria sobre el *embedding*.

4.1.3 INSTANCE SEGMENTATION

En esta última parte, se tiene que identificar cada línea del carril añadiéndole un identificador o color diferente a cada una de las líneas. Para lograr esto, se hace uso de la máscara binaria, el *embedding* y algo de post-procesamiento junto con algunos algoritmos de *clustering*.

Este proceso debería de permitir lograr unos resultados parecidos a los que podemos ver en esta ilustración:

Ilustración 12: Ejemplo output final LaneNet

Fuente: https://github.com/MaybeShewill-CV/lanenet-lane-detection
Autor: MaybeShewill-CV

En la ilustración 12, podemos ver un ejemplo de lo que podemos esperar como output final de LaneNet. La máscara final de *instance segmentation* se superpondrá a la imagen de la carretera y se podrán reconocer las diferentes líneas del carril con diferentes colores.

4.2 DISEÑO EXPERIMENTAL

Este proyecto es puramente experimental y busca lograr resultados similares a los del paper original [1]. Por ello, se ha decidido utilizar el mismo *benchmark* propuesto en [1], el TuSimple *dataset*.

Además, la naturaleza experimental de este proyecto hace que tenga sentido desarrollarlo en un entorno en la nube como Google Colaboratory.

4.2.1 TUSIMPLE DATASET

El *dataset* TuSimple [42], es uno de los varios *benchmarks* que existen para el entrenamiento de algoritmos de inteligencia artificial con capacidades de conducción autónoma. Este *dataset* en concreto contiene datos para la detección de carriles y para la estimación de velocidad.

Me centraré en el *dataset* de detección de carriles dado que es el que he utilizado para este trabajo:

4.2.1.1 CARACTERÍSTICAS

- **Complejidad**
  - Condiciones climáticas buenas y medias
  - Diferentes horas del día
  - Autopistas de 2, 3 y 4 carriles
  - Diferentes condiciones de tráfico

- **Tamaño**
    - Los datos de entrenamiento contienen 3626 clips de video de 20 frames cada uno pero solamente hay un frame anotado por cada clip. Es decir, existen 3626 frames anotados.
    - Los datos de testeo contienen 2782 clips de video de 20 frames cada uno pero al igual que en los datos de entrenamiento, solamente hay un frame anotado por cada clip. Es decir, existen 2782 frames anotados.
    - Los frames tienen un tamaño de 1280 pixeles de ancho × 720 pixeles de alto.
- **Tipo de anotaciones**
    - Las anotaciones son coordenadas que marcan los carriles.

4.2.1.2 DETALLES DE ESTRUCTURA Y FORMATO

El *dataset* TuSimple no viene con los carriles dibujados por defecto y para poder entrenar al modelo, hay que preprocesar todos estos datos. Antes de este preprocesamiento debemos entender la estructura y el formato en el que se guardan los datos para poder manipularlos:

Dentro del directorio principal del *dataset*, existe un directorio con las muestras de entrenamiento y otro con las muestras de testeo. La estructura de ambos directorios es la siguiente:

- **Clips/.** Este directorio contiene todos los clips de video
    - **Clip_1/.** Este directorio contiene todos los frames que forman el primer clip de video.
        - **1.jpg.** Este es el primer frame del clip de video.
        - **…**
        - **20.jpg.** Este es el último frame del clip de video.
    - **…**
    - **Clip_n/.** Este directorio contiene todos los frames que forman el último clip de video.
        - **1.jpg.** Este es el primer frame del clip de video.
        - **…**
        - **20.jpg.** Este es el último frame del clip de video.
- **Label_data_(fecha).json.** Este fichero en formato JSON contiene toda la información de las anotaciones de los frames. Este es el fichero que nos otorga la información necesaria para poder dibujar los carriles.

Una vez conocida la estructura de directorios, vamos a analizar la estructura y el formato del fichero .json que contiene la información con las anotaciones:

{
‘lanes’: Lista que contiene la anchura a la que están las coordenadas de los carriles
‘h_sample’: Lista que contiene la altura a la que están las coordenadas de los carriles
‘**raw_file’:** String que contiene el *path* del frame número 20 del clip de video.
}

Como máximo habrá 5 marcas de carril. Se esperan 4 marcas de carril (carril actual y carril izquierdo/derecho) pero puede aparecer un carril adicional que se utilizará cuando se cambia de carril. De esta manera, no habrá confusión para saber cual es el carril actual.

Todas las marcas de los carriles tienen las mismas alturas (estas alturas están registradas en el campo ‘h_sample’). Esto significa que se puede emparejar cada elemento de un carril con los elementos del campo ‘h_sample’ para obtener la coordenada de la marca del carril en las imágenes.

Además, los carriles están alrededor del centro de la vista, lo que anima al vehículo de conducción autónoma a centrarse en el carril actual y en los carriles izquierdo/derecho. Estos carriles son esenciales para el control del coche.

4.2.1.3 PREPROCESADO

Una vez comprendida la estructura de los directorios del *dataset* TuSimple y la estructura y el formato de las anotaciones, es hora de hablar del preprocesado.

Para poder llevar a cabo el modelo de LaneNet, se necesita preprocesar las imágenes de una manera específica: hay que redimensionar las imágenes originales a 256 pixeles de altura × 512 píxeles de anchura, crear una máscara binaria en el que las líneas se marquen con 1s y todas las demás partes de la imagen se marquen con 0s y crear una máscara de *instance segmentation* en las que cada línea del carril tenga un identificador diferente (se identificará utilizando colores diferentes para cada línea).

Sin embargo, si simplemente dibujamos las marcas de nuestro archivo .json, lo que conseguiremos es esto:

Ilustración 13: Ejemplos de imágenes del *dataset* TuSimple sin preprocesar

Fuente: https://github.com/TuSimple/tusimple-benchmark/tree/master/doc/lane_detection
Autor: http://www.tusimple.ai

Como se puede ver en la ilustración 13, si solamente marcamos las imágenes con las coordenadas del archivo .json, solo conseguiremos algunos puntos sobre cada una de las líneas de la calzada, como se puede apreciar en verde. Estas imágenes preprocesadas no le serán útiles a LaneNet dado que busca otro tipo de preprocesado en sus imágenes para poder ser entrenada y testeada:

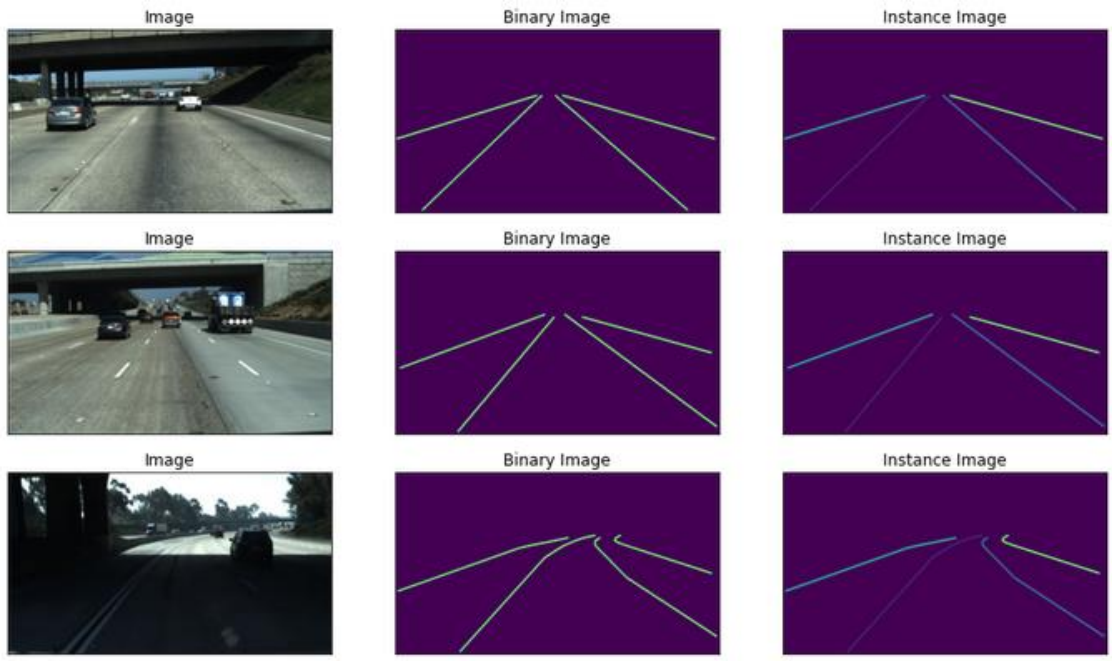


Ilustración 14: Input, máscaras binarias y máscaras de instancias

Sin embargo, como podemos observar en la ilustración 14, una vez obtenidas las coordenadas podemos ajustar una función polinómica de segundo grado sobre las coordenadas. Gracias a este método, las coordenadas pasan de ser únicamente unos pocos puntos distribuidos en las líneas de los carriles, a marcar todas las líneas de los carriles.

Si prestamos atención a la ilustración 14, podemos observar las imágenes originales redimensionadas a 256 píxeles de altura × 512 píxeles de anchura en la izquierda de la ilustración. En la columna central, podemos ver tres máscaras binarias creadas ajustando un polinomio de segundo grado a las coordenadas del archivo .json. Finalmente, en la derecha del todo, podemos observar cada uno de los carriles identificados con diferentes colores, también conocido como *instance segmentation*. Estas últimas imágenes con colores en cada línea de carril son las imágenes que queremos predecir a partir de las imágenes originales. Es decir, el input de nuestro modelo serán imágenes a color y el output será la *instance segmentation* de las líneas de los carriles.

4.2.2 GOOGLE COLABORATORY

Dado que no es un proyecto pensado para poner en producción, los notebooks de Google Colaboratory han sido la principal herramienta que se ha utilizado para desarrollar este proyecto. En un principio, se ha utilizado la versión gratuita, pero entrenar un modelo del calibre de LaneNet ha requerido de una mayor cantidad de memoria RAM y de computación en GPU. Por ello, también se ha utilizado la versión de pago Google Colaboratory Pro.

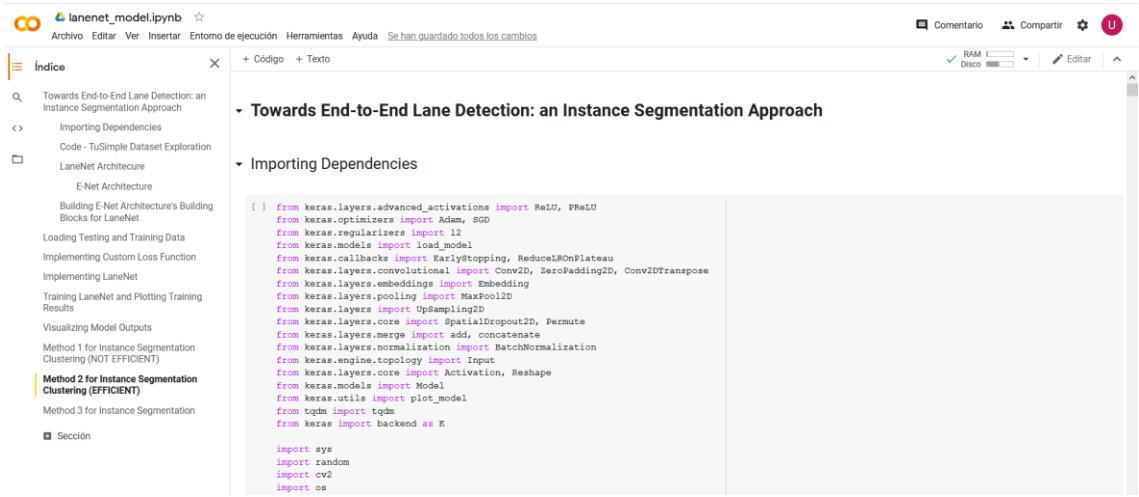


Ilustración 15: Interfaz de Google Colaboratory

5. DESARROLLO DEL PROYECTO

5.1 METODOLOGÍA REALIZADA

Debido a la magnitud y complejidad de LaneNet, se ha realizado el proyecto de manera incremental abordando las diferentes partes del proyecto en un orden secuencial empezando por la rama de segmentación binaria, siguiendo con la rama de *embedding* y finalizando con el post-procesado para lograr una correcta segmentación de instancias de los carriles.

5.1.1 E-NET

Como se ha comentado en el apartado 4, el modelo LaneNet hace uso de la arquitectura E-Net [43]. Este modelo es un autoencoder, por lo que su arquitectura de encode-decode es muy compacta y rápida a la hora de hacer inferencias. En el dominio de la percepción de los vehículos autónomos la velocidad de inferencia es de gran importancia y este motivo, junto con que es uno de los modelos liderando el estado del arte en la segmentación semántica en tiempo real han tenido un gran peso en su elección. Por ello, E-Net ha sido utilizado como *backbone* dentro de LaneNet.

Siguiendo el paper de LaneNet [1], la arquitectura de E-Net ha sido modificada para recibir un input de 256 de alto × 512 de ancho a diferencia del input que recibe en el paper original [43] de 512 de alto × 512 de ancho.

Name	Type	Output size
initial		$16 \times 256 \times 256$
bottleneck1.0	downsampling	$64 \times 128 \times 128$
$4\times$ bottleneck1.x		$64 \times 128 \times 128$
bottleneck2.0	downsampling	$128 \times 64 \times 64$
bottleneck2.1		$128 \times 64 \times 64$
bottleneck2.2	dilated 2	$128 \times 64 \times 64$
bottleneck2.3	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.4	dilated 4	$128 \times 64 \times 64$
bottleneck2.5		$128 \times 64 \times 64$
bottleneck2.6	dilated 8	$128 \times 64 \times 64$
bottleneck2.7	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.8	dilated 16	$128 \times 64 \times 64$
Repeat section 2, without bottleneck2.0		
bottleneck4.0	upsampling	$64 \times 128 \times 128$
bottleneck4.1		$64 \times 128 \times 128$
bottleneck4.2		$64 \times 128 \times 128$
bottleneck5.0	upsampling	$16 \times 256 \times 256$
bottleneck5.1		$16 \times 256 \times 256$
fullconv		$C \times 512 \times 512$

Ilustración 16: Arquitectura E-Net en el paper original

Autor: Abhishek Chaurasia, Sangpil Kim & Eugenio Culurciello
Fuente: 1606.02147v1.pdf (arxiv.org)

En la ilustración 16, se puede ver la arquitectura completa del paper original de E-Net [43]. Esta arquitectura está dividida en varias fases y se pueden distinguir mediante las líneas horizontales y el primer dígito después del nombre de cada bloque. A pesar de que en esta ilustración vemos que el output es de una dimensión de 512×512×el número de canales, LaneNet tiene dos ramas en las que tiene outputs de 256×512×1 para la rama de la máscara binaria y $256 \times 512 \times 4$ para la rama del *embedding*. Por esta razón, puede que algunos de los valores intermedios de la tabla tampoco se ajusten al *backbone* final utilizado en LaneNet.

Hay dos bloques de construcción principales en E-Net en los que se construye toda la red. El denominado bloque inicial y el *bottleneck module* o módulo de cuello de botella en español:

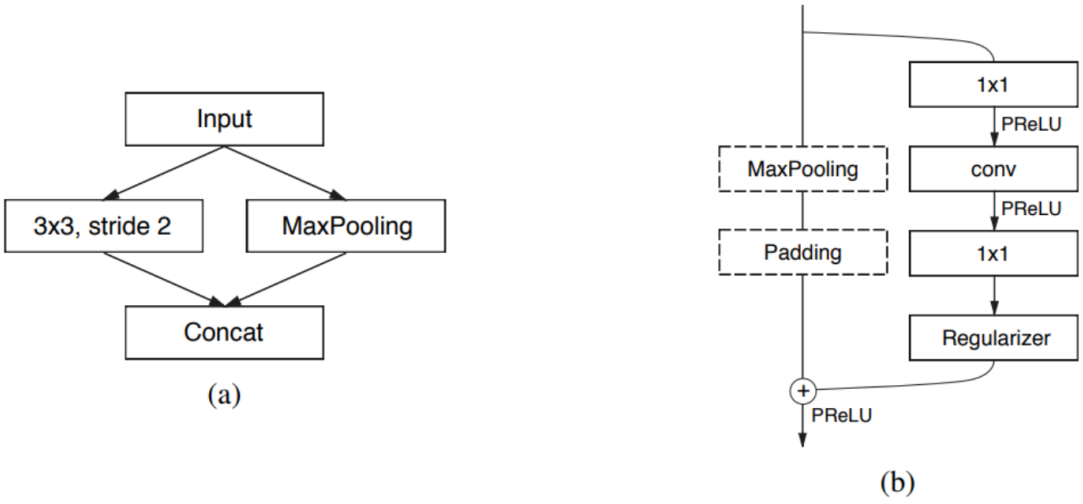


Ilustración 17: Bloques de construcción de E-Net

Autor: Abhishek Chaurasia, Sangpil Kim & Eugenio Culurciello
Fuente: 1606.02147v1.pdf (arxiv.org)

En el bloque (a), podemos ver el denominado bloque inicial. Este bloque, hace una operación de *max pooling* y una convolución 2D de 13 filtros con un *kernel* de 3×3 utilizando un *stride* de 2. Una vez hechas estas operaciones, los mapas de características resultantes se concatenan y suman un total de 16 mapas de características.

En el bloque (b), podemos ver el denominado *bottleneck module*. Existen variaciones de este módulo a lo largo de toda la arquitectura E-Net y en estas, las convoluciones 2D pueden ser regulares, dilatadas o convoluciones totales también conocidas como deconvoluciones. Estas convoluciones pueden tener filtros con *kernels* de 3×3 o de 5×5 descompuestas en 2 convoluciones asimétricas (una convolución de 5×1 y otra de 1×5).

Todos los *bottleneck modules* consisten en convoluciones 1×1 para reducir la dimensionalidad, una convolución principal de 3×3 que puede ser regular, dilatada o total, una expansión de 1×1 y cada una de las convoluciones mencionadas cuentan con una capa de *batch normalization* y una función de activación PReLU entre ellas.
Si el módulo está haciendo *downsampling*, se añade una capa de *max pooling* a la rama principal y la primera convolución 1×1 para reducir la dimensionalidad se reemplaza por una convolución 2×2 con un *stride* de 2.
En lo que a la regularización respecta, se utilizan capas de *spatial dropout* de p=0.01 antes del *bottleneck module* 2.0 y de p=0.1 después del *bottleneck module* 2.0.

Si juntamos todo lo mencionado anteriormente, nos queda lo siguiente:

1. El codificador o *encoder* en las fases 1, 2 y 3. Estas fases consisten en 5 bloques *bottleneck* con la excepción de que en la fase 3 no se hace *downsample*.

2. El decodificador o *decoder* en las fases 4 y 5. La fase 4 tiene 3 *bottleneck modules* y la fase 5 tiene 2.
3. Una vez implementadas estas 5 fases, E-Net se completa utilizando una capa *fullconv* también conocida como capa deconvolucional de las mismas dimensiones de ancho y de alto que el input pero con el mismo numero de canales a las clases que se quieren predecir. En nuestro caso el número de clases será 1 por lo que nuestro output de E-Net previo a implementar las dos ramas restantes será de 256×512×1.

Otras excepciones a tener en cuenta a la hora de implementar E-Net son las siguientes:

- No utilizan el valor del *bias* en ninguna de las proyecciones.
- Entre cada capa de convolución y función de activación se utiliza una capa de *batch normalization*.
- En el decodificador, se reemplaza la capa de *max pooling* por la capa de *max unpooling* y el *padding* también es reemplazado por la regularización *spatial convolution* sin utilizar el valor de *bias*.
- Se utiliza un *learning rate* de 5e-4.
- Se utiliza un optimizador Adam.
- Se utiliza un regularizador L2 a nivel de *kernel* de 2e-4

5.1.2 BINARY SEGMENTATION

Una vez comprendida la estructura y el *backbone* de LaneNet, procedo a explicar la rama de segmentación binaria:

Como se ha explicado en el apartado 4.1.1, esta rama devuelve una máscara binaria identificando las líneas del carril.

Para lograr que la red identifique las líneas de los carriles y no las pase por alto, es necesario crear una función de pérdida llamada *weighted categorical crossentropy*. Si nos fijamos en la ilustración 10, se puede observar que las líneas de los carriles y el *background* están claramente desbalanceados. Para ello, tenemos que dar mayor peso a los pixeles de los carriles consiguiendo así que el modelo se centre en predecir estas líneas. Para definir la función *weighted categorical crossentropy* primero es necesario invertir los pesos de los pixeles de los carriles y los pixeles del *background* y de esta manera forzar al modelo a dar mayor peso a los pixeles de los carriles. De la siguiente manera, se invertirían los pesos de una sola máscara binaria:

$$w = \frac{1}{\log\left(\left[\frac{pixels_{lane}}{pixels_{total}} \frac{pixels_{background}}{pixels_{total}}\right] + c\right)} = \left[w_{lane}\ w_{background}\right]$$

$pixels_{lane}$ equivale al número de pixeles de carriles que contiene la imagen binaria

$pixels_{background}$ equivale al número de pixeles de *background* que contiene la imagen binaria

C es una constante definida por el paper [1] con el valor 1.02

w_{lane} equivale al nuevo valor de los pesos de los carriles

$w_{background}$ equivale al nuevo valor de los pesos del *background*

Una vez invertidos los pesos, podemos calcular la función *weighted categorical crossentropy* de la siguiente manera*:*

$$loss = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) \cdot W_i$$

y_i equivale al *ground truth* de la imagen binaria en la posición i-ésima convertida a *one hot encoding*

$\hat{y}_i$ equivale a la distribución de probabilidades por píxel de la imagen binaria

W_i equivale a los pesos invertidos del carril y el *background* de la imagen en la posición i-ésima

n equivale al tamaño del *batch*

5.1.3 EMBEDDING

Otra de las ramificaciones es la rama del *embedding*. Esta rama, como se ha mencionado en el apartado 4.1.2, devuelve una representación N-dimensional del output de la máscara binaria. Este output N-dimensional junto con la máscara binaria de la otra ramificación, nos facilita utilizar un algoritmo de *clustering* para identificar con mayor facilidad cada línea del carril en el post-procesado para lograr una segmentación de instancias.

Para lograr una separación entre cada línea del carril y poder clusterizar cada línea con facilidad, muchos de los métodos de segmentación más utilizados no son adecuados para la segmentación de líneas de carril porque están hechos para segmentar elementos más compactos y las líneas no lo son. Por ello, se utiliza un método de aprendizaje por métrica de distancia propuesto por DeBrabandere et al. [35].

Aplicando la *discriminative loss function* o función de pérdida discriminativa en español, la rama de *embedding* es entrenada para que la distancia entre los píxeles del *embedding* pertenecientes a la misma línea del carril sea pequeña y la distancia entre los píxeles del *embedding* de diferentes líneas del carril sea grande. Gracias a esto, los píxeles de la misma línea se clusterizarán juntos formando clusters únicos por cada línea del carril.

Esta función de pérdida genera una fuerza de empuje entre *clusters* para alejarlos lo máximo posible y una fuerza de tracción para acercar los diferentes píxeles a su *cluster* más cercano:

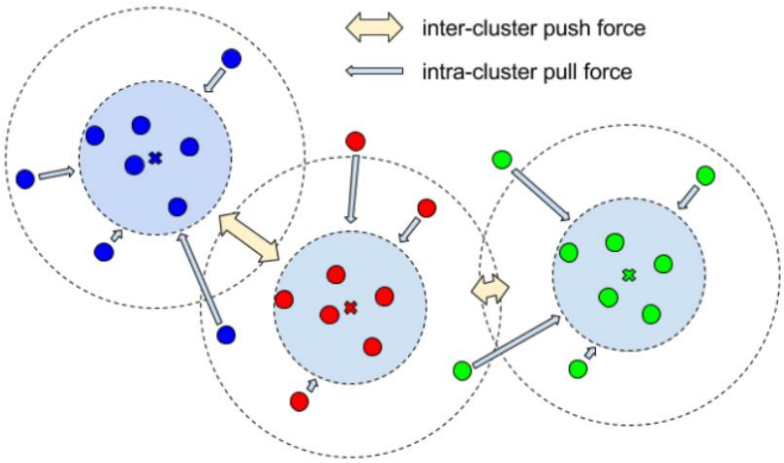


Ilustración 18: Visualización del resultado al aplicar la *discriminative loss function*

Fuente: https://arxiv.org/pdf/1708.02551.pdf
Autor: B. De Brabandere, D. Neven & L. Van Gool

Para lograr implementar esta función de pérdida, es necesario comprender sus 3 partes:

1. **Fuerza de tracción.** Se toman todos los píxeles de una instancia y se calcula su media. La fuerza de tracción atraerá a todos los píxeles del *embedding* de la misma instancia al mismo punto. En resumen, esta parte reduce la varianza del *embedding* por cada instancia.

$$L_{var} = \frac{1}{C}\sum_{c=1}^{C}\frac{1}{N_c}\sum_{i=1}^{N_c}[\|\mu_c - x_i\| - \delta_v]_+^2$$

C equivale al número de *clusters*

μ_c equivale al punto medio del *cluster* c

δ_v equivale al umbral de la fuerza de tracción

N_c equivale a todos los puntos que forman un *cluster* c

2. **Fuerza de empuje.** Se toman todos los puntos centrales del espacio del *embedding* y se empujan entre ellos para alejarlos.

$$L_{dist} = \frac{1}{C(C-1)} \sum_{\substack{c_A=1 \\ c_A \neq c_B}}^{C} \sum_{c_B=1}^{C} [2\delta_d - \|\mu_{c_A} - \mu_{c_B}\|]_+^2$$

C equivale al número de *clusters*

μ_c equivale al punto medio del *cluster* c

δ_d equivale al umbral de la fuerza de empuje

3. **Regularización.** Evita que los centros estén demasiado lejos del origen.

$$L_{reg} = \frac{1}{C} \sum_{c=1}^{C} \|\mu_c\|$$

C equivale al número de *clusters*

μ_c equivale al punto medio del *cluster* c

Una vez calculadas las 3 partes, el valor de la *discriminative loss* se calcularía de la siguiente manera:

$$L = \alpha \cdot L_{var} + \beta \cdot L_{dist} + \gamma \cdot L_{reg}$$

$$\alpha = \beta = 1$$

$$\gamma = 0.001$$

Si nos fijamos en [35], podemos ver como esta función de pérdida es capaz de agrupar puntos muy heterogeneos para lograr una segmentación de instancias:

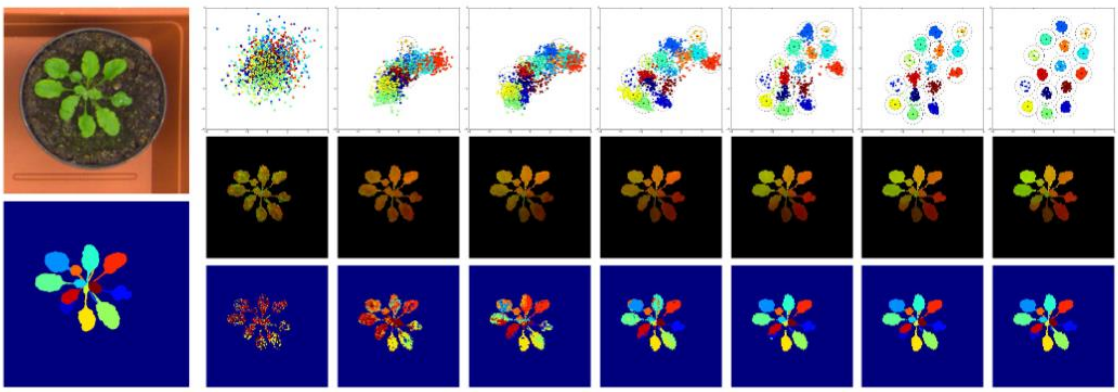


Ilustración 19: Resultado de convergencia en la función *discriminative loss*

Fuente: https://arxiv.org/pdf/1708.02551.pdf
Autor: B. De Brabandere, D. Neven & L. Van Gool

Como se puede observar en la ilustración 19, utilizando la función *discriminative loss*, partiendo de unos píxeles muy heterogeneos y sin un orden aparente se pueden agrupar facilitando el posterior *clustering* para conseguir la segmentación de instancias. De esta misma manera, se creará el embedding N-dimensional que separará los píxeles pertenecientes a líneas del carril diferentes facilitando su posterior *clustering* y segmentación de instancias.

5.1.4 INSTANCE SEGMENTATION

La *instance segmentation* se basa en identificar cada uno de los carriles en la imagen a nivel de píxel y saber diferenciarlos. Para ello, utilizamos los dos outputs de las ramas de E-Net aplicando la máscara binaria predecida a cada uno de los canales del *embedding*.

Una vez aplicada la máscara sobre el *embedding*, se puede aplicar un algoritmo de *clustering* como MeanShift o DBSCAN para conseguir la segmentación de instancias. Este método, aunque funciona, no es para nada eficiente ya que el algoritmo de *clustering* tiene que clusterizar 512×256×4 píxeles. Esto computacionalmente es muy lento en el caso de MeanShift y en el caso de DBSCAN, además de ser muy lento, la cantidad de memoria RAM necesaria se dispara haciéndolo inviable.

Por ello, en lugar de clusterizar todos los píxeles, se ha aplicado la máscara binaria al *embedding* por cada canal y posteriormente se ha aplicado MeanShift pero solamente a los pixeles cuyo valor no sea 0. Es decir, se ha aplicado el *clustering* a las líneas desechando todo el *background*.

Una vez hecho esto, a cada *cluster* se le ha aplicado un color diferente para poder diferenciar dichas líneas:

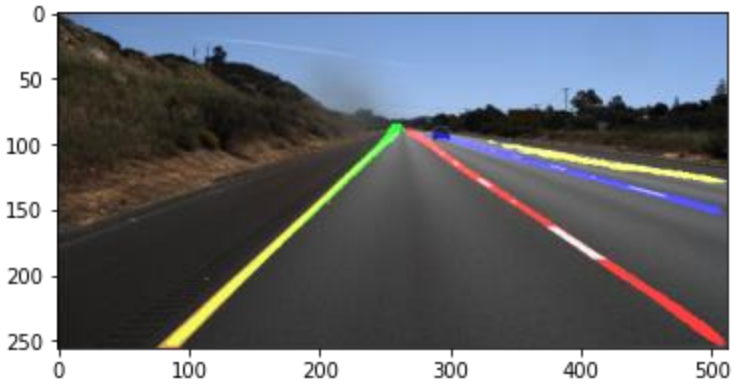


Ilustración 20: Imagen con máscara de instancias superpuesta

Como se puede ver en la ilustración 20, una vez aplicado el algoritmo MeanShift, se han dividido las líneas en *clusters* y se le ha aplicado un color diferenciador a cada una de las líneas.

5.2 TUSIMPLE DATASET

Como se ha explicado previamente en la sección 4 de la memoria, descripción del proyecto, el *dataset* a utilizar es el TuSimple *dataset*. Este es un conjunto de datos utilizado como *benchmark* en competiciones y algoritmos de visión artificial aplicada a la percepción de vehículos autónomos. Para una correcta ingesta de los datos en el modelo desarrolado, ha sido necesario crear dos scripts de preprocesado:

5.2.1 SCRIPTS DE PREPROCESADO

Los scripts desarrollados para el correcto preprocesamiento del *dataset* TuSimple son los siguientes:

5.2.1.1 PROCESS_TUSIMPLE_DATASET.PY

Este script tiene como objetivo preprocesar el *dataset* TuSimple creando a partir de las imágenes originales y las anotaciones, las imágenes binarias y las imágenes de instancia de carril.

Para ejecutar este script es necesario haber descargado previamente el *benchmark* TuSimple, descomprimir el archivo .zip y ejecutar el siguiente comando:

python process_tusimple_dataset.py --src_dir “path hasta el archivo descomprimido del *benchmark* TuSimple”

Una vez ejecutado este script, se crearán las carpetas test y train que contendrán a su vez 3 carpetas más: una carpeta con las imágenes originales, otra con las imágenes binarias y otra con las imágenes de instancia de carril. Ejemplos de estas imágenes pueden verse en la ilustración 13.

5.2.1.2 CONVERT_TUSIMPLE_DATASET_TO_BINARY.PY

Este script tiene como objetivo convertir todas las imágenes preprocesadas en arrays numpy en el que cada imagen estará redimensionada a 256 píxeles de altura × 512 píxeles de anchura.

Para ejecutar este script es necesario haber preprocesado previamente el *benchmark* TuSimple con el script anteriormente mencionado y ejecutar el siguiente comando:

python convert_tusimple_dataset_to_binary.py

Una vez ejecutado este script, se creará una carpeta adicional dentro de las carpetas test y train las cuales contendrán a su vez los 3 archivos binarios .npy que contendrán los arrays numpy con las imágenes preprocesadas.

6. RESULTADOS DEL PROYECTO

En esta sección expondré la progresión de los resultados del modelo LaneNet implementado desde 0 así como los problemas que se han ido encontrando durante su implementación.

Inicialmente, se implementó la rama de la segmentación binaria utilizando una sola clase, una función de activación sigmoide y una *weighted binary crossentropy loss*. Para implementar esta función de pérdida, primero era necesario calcular los mapas de pesos de la siguiente manera:

$$weight\_maps_{lane} = weight_{lane} * gt\_binary_{batch}$$
$$weight\_maps_{background} = weight_{background} * (1 - gt\_binary_{batch})$$

$weight_{lane}$ equivale al nuevo valor de los pesos de los carriles

$weight_{background}$ equivale al nuevo valor de los pesos del *background*

gt_binary_{batch} equivale al lote de imágenes binarias anotadas

Una vez calculados estos mapas de pesos, se aplica la función de pérdida *weighted binary crossentropy*:

$$loss = -\sum_{i=1}^{batch\ size} y_i \cdot \hat{y}_i \cdot weight\_maps_{lane_i} + (1 - y_i) \cdot (1 - \hat{y}_i) \cdot weight\_maps_{background_i}$$

y_i equivale a la imagen binaria en la posición i que se utiliza como ground truth

$\hat{y}_i$ equivale a la predicción de la imagen binaria generada por el modelo

$weight\_maps_{lane}$ equivale a una imagen con los nuevos pesos de los carriles

$weight\_maps_{background}$ equivale a una imagen con los nuevos pesos del *background*

Sin embargo, esta función de pérdida tardaba mucho en converger y sus resultados no acababan de diferenciar las líneas del *background* de una manera eficaz. Además, el entrenamiento y la *accuracy* de la segmentación binaria eran muy inconsistentes por lo que cada entrenamiento lograba resultados diferentes de entre un 70% y un 95% de *accuracy*. Este es el mejor resultado que se logró con este número de clases, función de activación y función de pérdida:

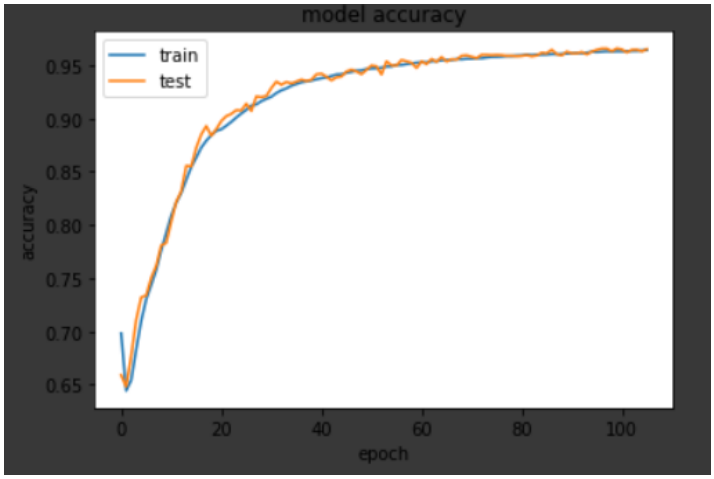


Ilustración 21: Mejor *accuracy* de la rama de segmentación binaria con *weighted binary crossentropy*

Como se puede ver en la ilustración 21, la rama de segmentación binaria logró unos muy buenos resultados en lo que a *accuracy* respecta. Sin embargo, necesitó al menos 100 *epochs* y muchísimos intentos de entrenamiento. Además, como podremos ver en la siguiente ilustración, el resultado no llegó a ser lo suficientemente bueno porque la predicción tenía muchísimo ruido:

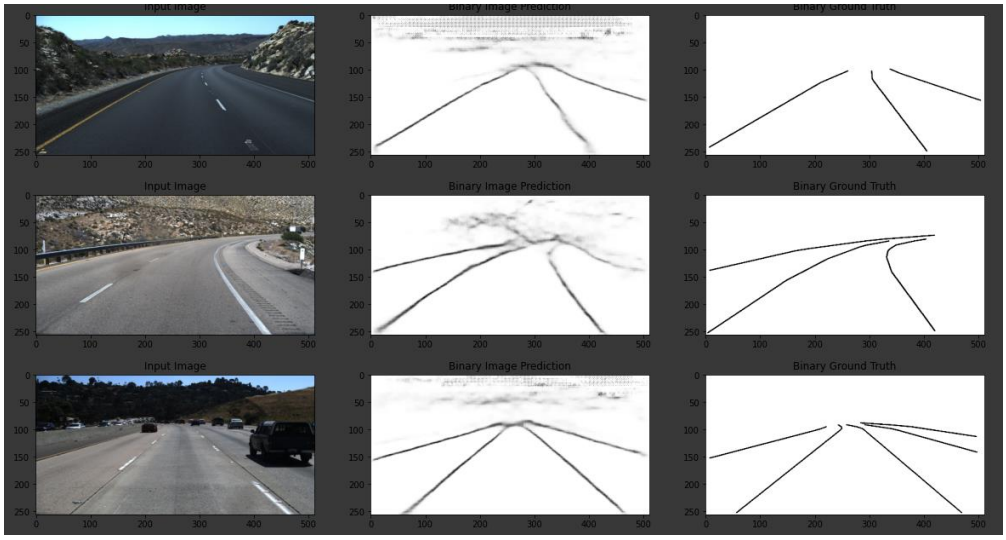


Ilustración 22: Imagen original, predicción binaria y *ground truth* binaria

Si prestamos atención a la columna central de la ilustración 22 podemos ver una gran cantidad de ruido en comparación con el *ground truth* por lo que es un resultado que no nos valdría para la posterior segmentación de instancias. Sin embargo, como he explicado previamente en el apartado 5.1.2, se cambió la función de pérdida de la *weighted binary crossentropy* a la *weighted categorical crossentropy* y los resultados mejoraron de una manera muy significativa:

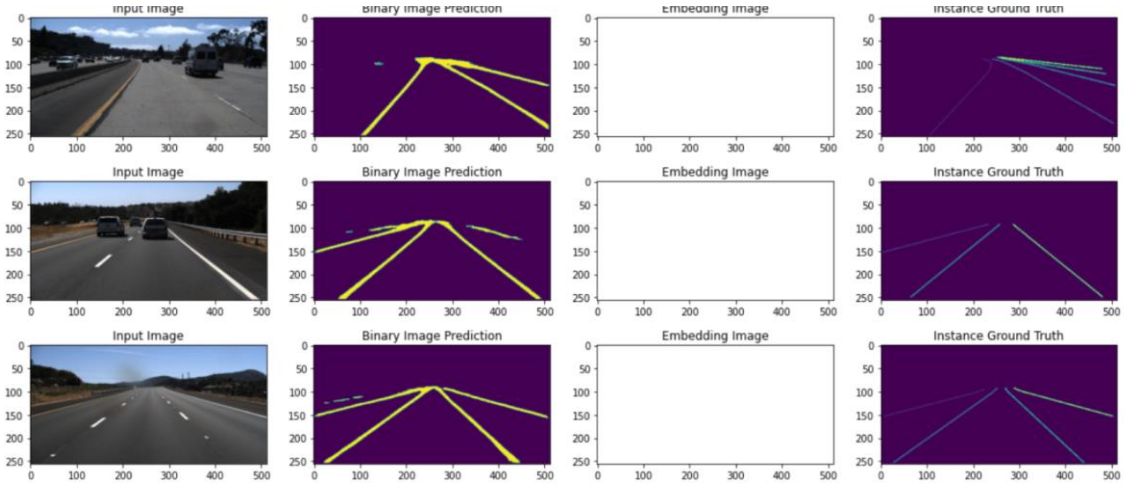


Ilustración 23: Imagen original, máscara binaria, *embedding* y *ground truth* de segmentación

En esta ilustración 23 podemos observar la gran mejora de la máscara binaria respecto a la ilustración 22. Ya no existe el ruido como el generado por la anterior función de pérdida y todos los valores de las líneas son 1s a diferencia de la anterior predicción que eran números entre 0 y 1. Sin embargo, podemos ver como en estos momentos del desarrollo la predicción del *embedding* no está funcionando bien.

Después de revisar la red durante varios días, se comprendió que en la arquitectura de E-Net, el *backbone* de LaneNet, no estaban bien implementadas las ramificaciones de la segmentación binaria y el *embedding*. En vez de empezar la ramificación una vez terminada la parte del *encode* y hacer la decodificación por separado, se decodificaban ambas en una misma rama y en la última capa se creaba la rama de la máscara binaria y la del *embedding*. Esto, junto con una mala implementación de la *discriminative loss function* estaba creando un problema porque no permitía que los *kernels* de la parte del *embedding* pudieran ajustarse para lograr un mejor resultado.

Muchos intentos después, no se consiguió arreglar el output del *embedding*. Este, siempre devolvía representaciones de 256×512×4 llenas de 0s en lugar de separar las diferentes líneas de carril en *clusters* para facilitar su *instance segmentation*. Hay varias hipótesis por las que se creía que el *embedding* no estaba funcionando bien:

1. El *paper* original [1] no ha explicado algunos detalles importantes para la implementación de esta rama.
2. La *discriminative loss function* utilizada no es correcta del todo. Esto implicaría, que los repositorios con mejores valoraciones de github no estarían siendo implementados de una manera fiel al *paper* [1].
3. Está ocurriendo el problema del desvanecimiento de los gradientes y por esto lo valores del *embedding* están convergiendo en 0.

Finalmente, se detectó el error en la *discriminative loss function*. Este, fué un error de codificación que no permitía que el *framework* de Keras actualizara los gradientes de algunas capas y por lo tanto no se lograban resultados. Una vez corregido este fallo de codificación los resultados fueron muy buenos:

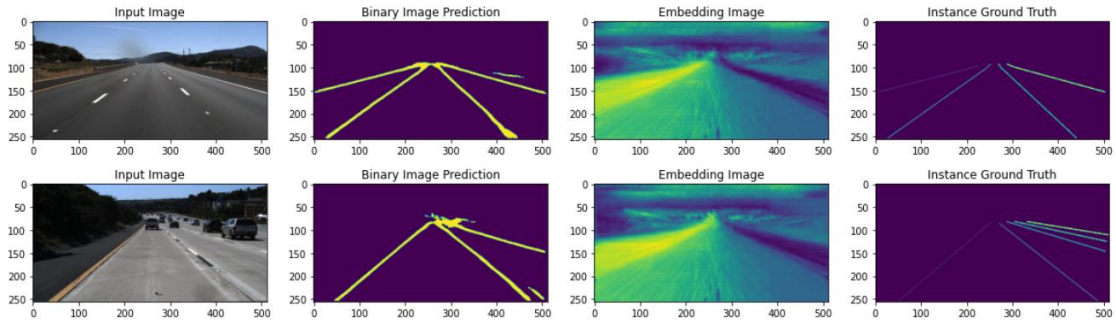


Ilustración 24: Resultados finales de segmentación binaria y *embedding*

En la ilustración 24, se puede observar el mejor resultado de mi implementación de LaneNet. A diferencia de la ilustración, 23, el *embedding* ha sido arreglado y si se observa fijamente, se puede apreciar como los pixeles tienen valores diferentes para cada una de las líneas. Esto, facilitará el *clustering* para posteriormente hacer la segmentación de instancias.

El resultado final a nivel de output con un *learning rate* de 0.0005, *batch size* de 8, regularización l2 a nivel de cada *kernel* convolucional de 0.0002, un valor de $\alpha = 1$, $\beta = 1$, $\gamma = 0.001$, $\boldsymbol{\delta}_{\mathrm{d}} = 3.0$, $\boldsymbol{\delta}_{\mathrm{v}} = 0.5$ y un optimizador Adam, es el resultado que se puede ver en la ilustración 24. En este resultado la máscara binaria llega a un 95.3% de *accuracy* y el *embedding* logra separar los valores de las líneas de carril para facilitar el posterior *cluster* y la segmentación de instancias.

En lo que a la *instance segmentation* se refiere, a pesar de la falta de detalles del *paper* de LaneNet [1] se han logrado resultados similares y sin lugar a dudas han cumplido tanto mis expectativas como las de mi director del proyecto.
Este sería un ejemplo del modelo LaneNet en acción puesto a prueba sobre diferentes imágenes:

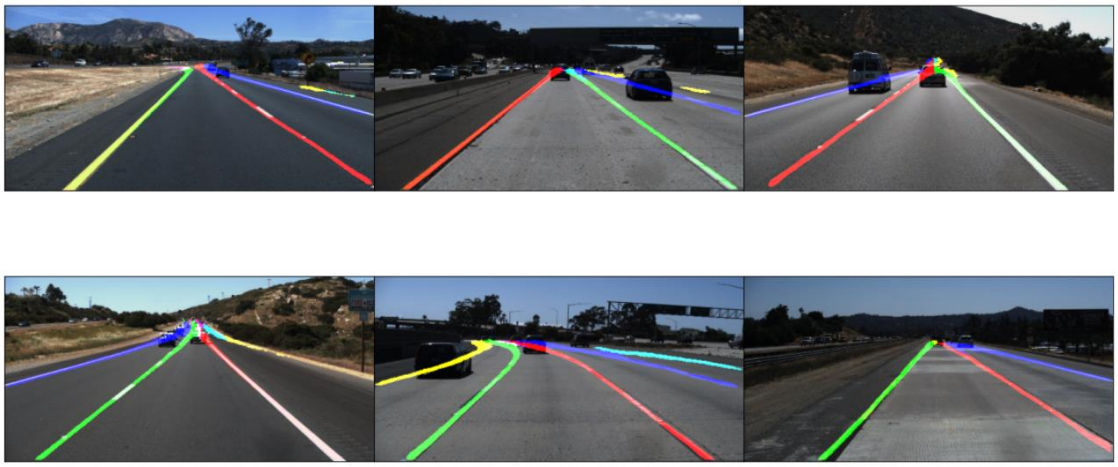


Ilustración 25: Aplicando LaneNet sobre imágenes reales

En la ilustración 25, se puede ver como LaneNet consigue diferenciar las líneas de carril con gran precisión. Esto podría mejorarse utilizando una transformación de perspectiva utilizando el modelo H-Net mencionado en el *paper* [1]. Sin embargo, este no era uno de los objetivos del proyecto por lo que queda pendiente para trabajos futuros.

Para lograr esta precisión han sido necesarias 41 *epochs* y alrededor de 6 horas de entrenamiento:

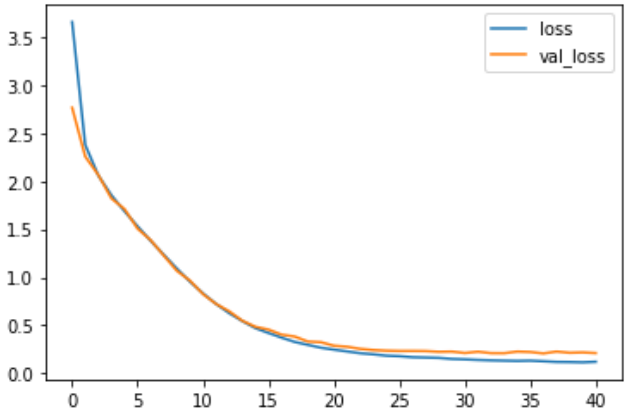
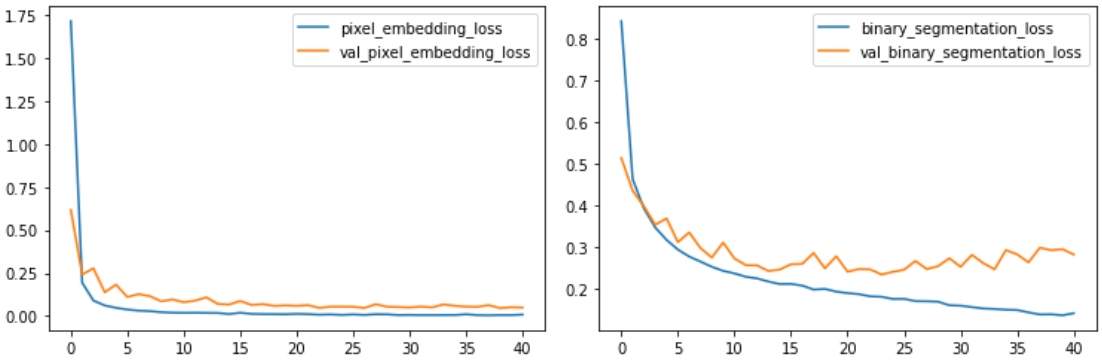


Ilustración 26: *Loss* general durante el entrenamiento de LaneNet



Ilustraciones 27 y 28: *Loss* de la rama del *embedding* y *loss* de la rama de segmentación

Si nos fijamos en las ilustraciones 26, 27 y 28, podemos visualizar como se ha ido minimizando tanto la función de pérdida general como las funciones de pérdida de cada rama de LaneNet. A simple vista, estos gráficos sugieren que LaneNet ha convergido bien como se proponían en el *paper* [1].

Para más información y poder conseguir un entendimiento más profundo sobre LaneNet, este es el enlace de mi notebook de Google Colaboratory con el que tendréis acceso como lectores: https://drive.google.com/drive/folders/1q_rV3KVk5fUe6bRlcCfItXaitmQWoeAi?usp=sharing

7. PLANIFICACIÓN DEL PROYECTO

Se ha planificado el proyecto de una manera incremental y modular. Estos han sido los pasos planificados a seguir en orden cronológico:

1. **Reunir información sobre algoritmos y métodos utilizados en el sector de la conducción autónoma**
    a. Leer papers
    b. Leer artículos

2. **Preparar el *dataset* TuSimple para la posterior ingesta del modelo**
    a. Comprender la naturaleza del *dataset* TuSimple
    b. Preprocesar el *dataset* TuSimple
    c. Convertir las imágenes preprocesadas a binario para facilitar la ingesta

3. **Implementación de LaneNet**
    a. Implementar arquitectura E-Net sin añadir las ramas de output
    b. Implementar rama de segmentación binaria
    c. Implementar rama de *embedding*
    d. Implementar el método de *clustering* para crear la *instance segmentation*

4. **Analizar el desempeño de LaneNet utilizando la *accuracy* como en el *paper***

5. **Crear clips de video poniendo el modelo a prueba en videos con escenarios reales**
    a. Fácil. Clip de conducción en autopista
    b. Intermedio. Clip de conducción en autopista
    c. Difícil. Clip de conducción en autopista
    d. Muy difícil. Clip de conducción en intersección de ciudad

8. PRESUPUESTO

CONCEPTO	MES 0	MES 1	MES 2	MES 3
Suscripción Mensual a Google Colab Pro	9.99 €	9.99 €	9.99 €	9.99 €
TOTAL	**39.96 €**			

Inicialmente, el presupuesto del proyecto iba a ser de 0 € pero debido a las limitaciones del hardware y de la computación en la nube gratuita disponible, se han gastado 9.99 € al mes en una suscripción a Google Colab Pro. Durante los 4 meses de duración del proyecto, este gasto ha supuesto un gasto total de 39.96 €.

9. CONCLUSIONES Y TRABAJOS FUTUROS

En este proyecto, se ha recreado la arquitectura que aparece en el *paper* de LaneNet [1] para lograr una segmentación de instancias en tiempo real sin indicar el número de carriles de antemano. Se han utilizado los mismos *hiperparámetros* que se indican en el *paper* [1] y las imágenes del *dataset* TuSimple. A pesar de la falta de detalles en la implementación por parte del *paper*, se ha conseguido implementar la arquitectura LaneNet con buenos resultados. Con ayuda de mi director del proyecto de fin de máster, me he dado cuenta de que los repositorios de github con mejores valoraciones de internet en la implementación de LaneNet están mal implementados. Estos repositorios, rondaban las 1500 estrellas en github lo cual indica que son muy populares pero, a pesar de su popularidad, su manera de computar el *embedding* utilizando la máscara binaria es errónea ya que no aplican la máscara binaria a los resultados del *embedding* para la posterior segmentación de instancias sino que aplican un postprocesado directamente sobre el *embedding* y la máscara binaria y añaden diferentes umbrales para eliminar el ruido y posteriormente poder hacer el *clustering*.

Volviendo a mi implementación, el modelo esta implementado al completo excepto por el apartado de *lane fitting* el cual no era un objetivo a cumplir para este proyecto. A pesar de los problemas que se han tenido durante toda la implementación, el proyecto ha sido un éxito.

Este proyecto de fin de máster me ha supuesto uno de los mayores retos a los que me he enfrentado tanto por su complejidad como por lo novel que soy en el campo del *deep learning* y la investigación. He podido experimentar una gran mejoría en mis habilidades para comprender *papers* y en el *deep learning* y he comprendido muchísimos conceptos que hace unos meses me resultaban muy difusos.

En trabajos futuros, se podría implementar la red H-Net al igual que en el *paper* original [1] para lograr un mejor ajuste de los carriles al crear una transformación de perspectiva y una vez logrado esto, se podrían utilizar técnicas de *data augmentation* y anotar más imágenes del TuSimple *dataset* para ver hasta donde es capaz de llegar este modelo.

Otro gran problema con los vehículos autónomos es la deficiente explicabilidad e interpretabilidad de los modelos de *deep learning* y esta podría ser una buena dirección a tomar para trabajos futuros. Un vehículo autónomo que toma decisiones sin que los humanos entendamos muy bien porque es algo muy peligroso ya que la conducción es una acción que en caso de hacerse mal puede causar tanto daños humanos como estructurales.

Para concluir, durante el desarrollo de este proyecto he podido observar el gran futuro y a la gran velocidad a la que evoluciona este campo y estoy seguro de que seguirá siendo

de esta manera y que seguirá atrayendo una gran inversión por parte de la academia pero sobre todo por parte de las empresas privadas.

10. BIBLIOGRAFÍA

1. D. Neven, B. De Brabandere, Stamatios G., Marc P.. Luc Van Gool, Towards End-to-End Lane Detection: an Instance Segmentation Approach. Aarxiv.org, 2018

2. "Defense Advanced Research Projects Agency". 2020. *Darpa.Mil*. Acceso el 19 de Noviembre. https://www.darpa.mil/.

3. "DARPA Grand Challenge". 2020. *Wikipedia.Org*. Acceso el 19 de Noviembre. https://en.wikipedia.org/wiki/DARPA_Grand_Challenge.

4. A. Borkar, M. Hayes, M. T. Smith, A Novel Lane Detection System With Efficient Ground Truth Generation. IEEE Trans. Intelligent Transportation Systems, vol. 13, no. 1, pp. 365-374, 2012.

5. H. Deusch, J. Wiest, S. Reuter, M. Szczot, M. Konrad, K. Dietmayer, A random finite set approach to multiple lane detection. ITSC, pp.270-275, 2012.

6. J. Hur, S.-N. Kang, S.-W. Seo, Multi-lane detection in urban driving environments using conditional random fields. Intelligent Vehicles Symposium, pp. 1297-1302, 2013.

7. H. Jung, J. Min, J. Kim, An efficient lane detection algorithm for lane departure detection. Intelligent Vehicles Symposium, pp. 976-981, 2013.

8. H. Tan, Y. Zhou, Y. Zhu, D. Yao, K. Li, A novel curve lane detection based on Improved River Flow and RANSA. ITSC, pp. 133-138, 2014.

9. P.-C. Wu, C.-Y. Chang, C.-H. Lin, Lane-mark extraction for automobiles under complex conditions. Pattern Recognition, vol. 47, no. 8, pp. 2756-2767, 2014.

10. K.-Y. Chiu, S.-F. Lin, Lane detection using color-based segmentation. Intelligent Vehicles Symposium, pp. 706-711, 2005.

11. H. Loose, U. Franke, C. Stiller, Kalman particle filter for lane recognition on rural roads. Intelligent Vehicles Symposium, pp. 60-65, 2009.

12. Z. Teng, J.-H. Kim, D.-J. Kang, Real-time Lane detection by using multiple cues. Control Automation and Systems, pp. 2334-2337, 2010.

13. A. L´opez, J. Serrat, C. Canero, F. Lumbreras, T. Graf, Robust lane markings detection and road geometry computation. International Journal of Automotive Technology, vol. 11, no. 3, pp. 395-407, 2010.

14. G. Liu, F. Wörgötter, I. Markelic, Combining Statistical Hough Transform and Particle Filter for robust lane detection and tracking. Intelligent Vehicles Symposium, pp. 993-997, 2010.

15. D. Neven, B. De Brabandere, S. Georgoulis, M. Proesmans, L.Van Gool, Fast Scene Understanding for Autonomous Driving. Deep Learning for Vehicle Perception, workshop at the IEEE Symposium on Intelligent Vehicles, 2017.

16. Z. Kim, Robust Lane Detection and Tracking in Challenging Scenar-ios. IEEE Trans. Intelligent Transportation Systems, vol. 9, no. 1, pp.16-26, 2008.

17. R. Danescu, S. Nedevschi, Probabilistic Lane Tracking in Difficult Road Scenarios Using Stereovision. IEEE Trans. Intelligent Transportation Systems, vol. 10, no. 2, pp. 272-282, 2009.

18. R. Gopalan, T. Hong, M. Shneier, R. Chellappa, A Learning Approach Towards Detection and Tracking of Lane Markings. IEEE Trans. Intelligent Transportation Systems, vol. 13, no. 3, pp. 1088-1098, 2012.

19. J. Kim, M. Lee, Robust Lane Detection Based On Convolutional Neural Network and Random Sample Consensus. ICONIP, pp. 454-461, 2014.

20. B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica,A. Coates, A. Y. Ng, An Empirical Evaluation of Deep Learning on Highway Driving. CoRR abs/1504.01716, 2015.

21. B. He, R. Ai, Y. Yan, X. Lang, Accurate and robust lane detection based on Dual-View Convolutional Neutral Network. Intelligent Vehicles Symposium, pp. 1041-1046, 2016.

22. J. Li, X. Mei, D. V. Prokhorov, D. Tao, Deep Neural Network forStructural Prediction and Lane Detection in Traffic Scene. IEEE Trans. Neural Netw. Learning Syst., vol. 28, no.3, pp. 690-703, 2017.

23. S. Lee, J.-S. Kim, J. S. Yoon, S. Shin, O. Bailo, N. Kim, T.-H. Lee, H.S. Hong, S.-H. Han, I. S. Kweon, VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition. CoRR abs/1710.06288, 2017.

24. J. Kim, C. Park, End-To-End Ego Lane Estimation Based on Sequential Transfer Learning for Self-Driving Cars. CVPR Workshops, pp.1194-1202, 2017.

25. A. Gurghian, T. Koduri, S. V. Bailur, K. J. Carey, V. N. Murali, Deep Lanes: End-To-End Lane Position Estimation Using Deep Neural Networks. CVPR Workshops, pp. 38-45, 2016.

26. J. Long, E. Shelhamer, T. Darrell, Fully convolutional networks for semantic segmentation. CVPR 2015.

27. H. Noh, S. Hong, B. Han, Learning deconvolution network for semantic segmentation. ICCV 2015.

28. O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation. International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 234-241, 2015.

29. L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, A. Yuille, Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. CoRR abs/1606.00915, 2016.

30. Z. Zhang, A. G. Schwing, S. Fidler, R. Urtasun, Monocular object instance segmentation and depth ordering with CNNs. ICCV, 2015.

31. Dai, K. He, and J. Sun. Instance-aware semantic segmentation via multi-task network cascades. In CVPR, 2016

32. B. Romera-Paredes, P. H. Torr, Recurrent instance segmentation. ECCV, 2016.

33. M. Bai, R. Urtasun, Deep watershed transform for instance segmentation. CoRR abs/1611.08303, 2016.

34. K. He, G. Gkioxari, P. Dollar, R. Girshick, Mask R-CNN. CoRRabs/1703.06870, 2017.

35. B. De Brabandere, D. Neven, L. Van Gool. CoRR abs/1708.02551, 2017.

36. Brownlee, Jason. 2019. "A Gentle Introduction To Computer Vision". *Machine Learning Mastery*. https://machinelearningmastery.com/what-is-computer-vision/.

37. Géron, Aurélien. 2020. *Hands-On Machine Learning With Scikit-Learn, Keras, And Tensorflow*. 2nd ed. O'Reilly.

38. Tatan, Vincent. 2019. "Understanding CNN (Convolutional Neural Network)". *Towards Data Science*. https://towardsdatascience.com/understanding-cnn-convolutional-neural-network-69fd626ee7d4.

39. Chen, Qiang. 2018. "Classification, Sigmoid Function". *Medium*. https://medium.com/machine-learning-and-math/classification-sigmoid-function-4f800363780c.

40. Pere, Christophe. 2020. "What Is Activation Function ?". *Towards Data Science*. https://towardsdatascience.com/what-is-activation-function-1464a629cdca.

41. "Towards End-To-End Lane Detection: An Instance Segmentation Approach". 2018. *Papers With Code*. https://paperswithcode.com/paper/towards-end-to-end-lane-detection-an-instance.

42. "Tusimple/Tusimple-Benchmark". 2017. *Github*. https://github.com/TuSimple/tusimple-benchmark/tree/master/doc/lane_detection.

43. A. Paszke, Abhishek C., S. Kim, E. Culurciello, ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation. Aarxiv.org, 2016.