Escuela Técnica Superior de Ingenieros
Industriales y de Telecomunicación

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Implementación de una Red Neuronal para Estimación de la Profundidad en Tiempo Real

(Implementation of a Deep Neural Network for Real-Time Depth Estimation)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Raquel De Esteban Ceballos

Octubre - 2018

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Raquel de Esteban Ceballos
Director del TFG: Pablo Pedro Sanchez Espeso
Título: "Implementación de una Red Neuronal para Estimación de la Profundidad en Tiempo Real"
Title: "Implementation of a Deep Neural Network for Real-Time Depth Estimation"

Presentado a examen el día: 10 de octubre de 2018

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Villar Bonet, Eugenio
Secretario (Apellidos, Nombre): Sánchez Espeso, Pablo Pedro
Vocal (Apellidos, Nombre): Alcalá Galán, Francisco

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado Nº
(a asignar por Secretaría)

Dedicado a
mi madre,
Irene.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Contexto

Las cámaras convencionales convierten el mundo que vemos en 3D a una imagen en 2D. Hasta ahora, las imágenes en 2D han sido consideradas lo suficientemente buenas para el consumidor, quitando importancia a la dimensión de profundidad, que se pierde en las imágenes 2D.

Con las mejoras en el campo de la visión artificial, combinado con el *Machine Learning*, muchas investigaciones han intentado que las máquinas entiendan nuestro mundo a través de una cámara para poder aumentar las capacidades humanas realizando nuevas tareas. Actualmente, en el campo de la visión artificial se pueden hacer cosas como reconocer escritura a mano, clasificar objetos, además de ser un componente crítico para el desarrollo de vehículos autónomos. En muchas de estas tareas en donde la información 2D es suficiente, los algoritmos de visión artificial han demostrado ser muy prometedores. Sin embargo, cuando analizan el mundo 3D real, las investigaciones en visión artificial se han encontrado con un cuello de botella. Los seres humanos tienen dos ojos que permiten percibir la profundidad de forma natural. Sin embargo, la mayoría de las aplicaciones de visión artificial dependen de una cámara para capturar e interpretar el mundo que la rodea, por lo que carecen de visión estéreo. Por lo tanto, la pérdida de la dimensión de profundidad limita significativamente el rendimiento [1].

Actualmente, las técnicas más utilizadas para obtener la profundidad de una imagen son los sistemas de luz estructurada, las cámaras estéreo y las cámaras de tiempo de vuelo.

Luz estructurada

Este método utiliza una fuente de luz (normalmente láser) para proyectar un patrón conocido. Una cámara detecta la distorsión del patrón reflejado para calcular el mapa de profundidad basado en la geometría del patrón. El sistema debe escanear todo el plano para obtener el mapa de profundidad y, aunque es muy preciso, lleva bastante tiempo. Además este método es sensible a la iluminación del ambiente, por lo que normalmente solo se implementa en zonas oscuras o de interior.

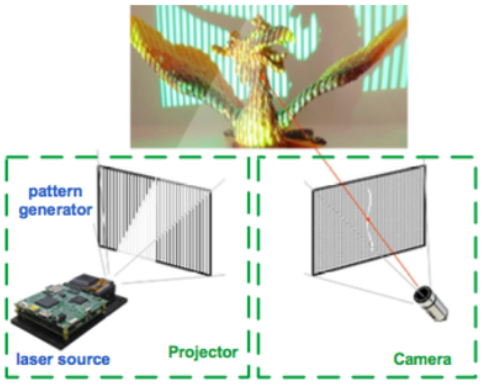


Figura 1.1: Sistema de luz estructurada.

Como se muestra en la Figura 1.1, un sistema de luz estructurada esta compuesto por un proyector y una cámara. El proyector ilumina un objeto y la cámara captura la luz reflejada.

Cámaras estéreo

El sistema de cámaras estéreo utiliza varias cámaras situadas en diferentes posiciones para capturar múltiples imágenes del mismo objetivo, a partir de las cuales se calcula el mapa de profundidad.

El tipo de cámara estéreo más utilizado es la cámara dual, mostrado en la Figura 1.2. En este sistema, se utilizan únicamente dos cámaras separadas una distancia similar a la que están separados los ojos humanos. Para cada punto del espacio, hay una pequeña diferencia de posición entre las dos imágenes que puede ser medida. Utilizando esta diferencia, se mide la profundidad por medio de geometría básica.

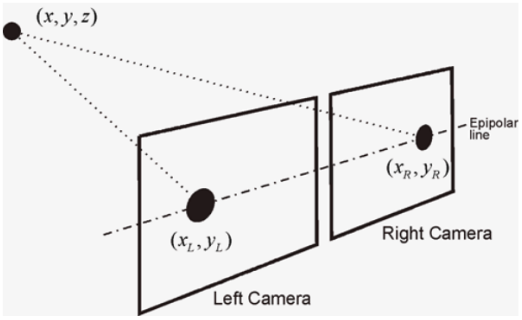


Figura 1.2: Visión estéreo.

Tiempo de vuelo

Esta técnica se basa en obtener la profundidad a partir del tiempo que tarda la luz en llegar a un objeto deseado. Para calcular este tiempo, una fuente de láser envía un pulso, y un sensor detecta la reflexión del pulso en el objeto objetivo para registrar su tiempo de vuelo. Sabiendo eso y la velocidad constante de la luz, el sistema puede calcular la distancia a la que se encuentra el objeto. Para garantizar una alta precisión, el período de pulso debe ser corto, lo que se traduce en un coste mayor.

1.2. Motivación

Los sistemas de cálculo de la profundidad presentados en la sección anterior necesitan varias cámaras o proyectores. Esto dificulta de manera considerable utilizar estos sistemas en aplicaciones médicas, vehículos autónomos, sistemas de realidad mixta, etc.

Este trabajo presenta un algoritmo de *Machine Learning* que calcula el mapa de profundidad a partir de una sola imagen, es decir, necesita solamente una cámara para obtener los valores de profundidad. Para hacer que este algoritmo funcione en tiempo real, y pueda ser utilizado para cualquiera de las aplicaciones citadas en el párrafo anterior, se ha implementado en un sistema embebido que acelera el algoritmo para que la profundidad pueda ser calculada en tiempo real.

Por lo tanto, las ventajas que presenta el algoritmo propuesto en este trabajo sobre los sistemas presentados en la sección 1.1 son que el primero solo utiliza una cámara y no necesita fuente de luz, por lo que puede ser utilizado en vehículos autónomos o sistemas de realidad mixta. Además, el algoritmo ha sido implementado en C++, lo que hace que pueda ser utilizado por diferentes dispositivos o plataformas.

1.3. Objetivos

El objetivo principal de este trabajo es explorar el uso de redes neuronales para obtener el mapa de profundidad de una escena a partir de una sola imagen RGB de la misma. Para ello, se ha implementado una Red Neuronal Convolucional en la plataforma embebida Jetson TX2 de NVIDIA, aprovechando su GPU para realizar la estimación en tiempo real.

Además de este objetivo general, los objetivos específicos son:

- Evaluar diferentes entornos de Machine Learning, como TensorFlow y Caffe.
- Estudiar técnicas de visión monocular para obtención de la profundidad.
- Evaluar redes neuronales para el procesado de imagen y vídeo.
- Implementar redes neuronales para procesado de vídeo en tiempo real en plataformas embebidas.

Capítulo 2

Machine Learning

2.1. ¿Qué es Machine Learning?

Machine Learning, traducido como aprendizaje automático, es un subcampo de las ciencias de la computación y una rama de la inteligencia artificial, cuyo objetivo es dotar a los computadores de la habilidad de aprender utilizando técnicas estadísticas [2].

Desarrollado a partir del estudio del reconocimiento de patrones y la teoría de aprendizaje computacional en inteligencia artificial, el aprendizaje automático explora el estudio y la construcción de algoritmos que pueden aprender y hacer predicciones sobre datos. El aprendizaje automático se emplea en una variedad de tareas en las que el diseño y la programación de algoritmos explícitos con un buen rendimiento es difícil o inviable [2].

2.1.1. Clasificación de los algoritmos

Los tipos de algoritmos utilizados en Machine Learning pueden clasificarse en función de la tarea que realizan, o en función de la aplicación a la que están destinados [2]. En función de las tareas del aprendizaje, los algoritmos de Machine Learning se clasifican en dos grandes categorías, dependiendo de si hay una "señal" o patrón de aprendizaje disponible para el sistema:

- **Aprendizaje supervisado:** Al computador se le proporcionan unas entradas y las salidas esperadas del sistema. El objetivo es que el sistema encuentre una regla general que mapee las entradas a las salidas.
    - **Aprendizaje semi-supervisado:** El computador recibe una señal de entrenamiento incompleta; es decir, se le proporciona un conjunto de estímulos de entrenamiento sin algunas (a menudo muchas) de las salidas esperadas.

  - **Aprendizaje por refuerzo:** Los datos de entrenamiento (en forma de fallos o aciertos) son proporcionados al sistema solo como retroalimentación a las acciones del mismo en un entorno dinámico, como conducir un vehículo o jugar un juego contra un oponente.

- **Aprendizaje no supervisado:** No se identifican las salidas deseadas con etiquetas. El objetivo es que el algoritmo de aprendizaje encuentre él solo un patrón en los datos de entrada.

Otra clasificación de los algoritmos de aprendizaje automático surge cuando se tiene en cuenta el resultado deseado de un sistema, es decir, la aplicación para la que va a ser utilizado. En este caso, los algoritmos se clasifican en:

- **Clasificación:** Las entradas se dividen en varias clases. El objetivo es producir un modelo que asigne entradas no proporcionadas durante el entrenamiento a una o más de estas clases. El entrenamiento generalmente se realiza de forma supervisada.

- **Regresión:** La regresión es un proceso estadístico que se utiliza para predecir la demanda a partir de una o más causas (variables independientes). A diferencia de los algoritmos de clasificación, cuya salida es el nombre del grupo al que pertenece la entrada, la regresión predice un número, como por ejemplo cuál va ser el precio de un artículo.

- **Agrupamiento:** El conjunto de entradas es dividido en grupos. A diferencia de la técnica de clasificación, los grupos no se conocen de antemano, por lo que esta suele ser una tarea no supervisada.

- **Reducción de dimensionalidad:** Simplifica las entradas al mapearlas en un espacio de dimensión inferior.

2.1.2. Tipos de algoritmos de aprendizaje

Actualmente existen más de 50 algoritmos o formas de aprendizaje utilizados en el campo de Machine Learning [3]. A continuación se presentan brevemente los más importantes.

Árboles de decisiones

Este tipo de aprendizaje usa un árbol de decisiones como modelo predictivo, en el que los datos son separados continuamente en función de un parámetro. El árbol esta compuesto por dos entidades: los nodos de decisión,

dónde se separan los datos, y las hojas, que son las decisiones (sí o no en el ejemplo de la Figura 2.1) o las respuestas finales. [4].

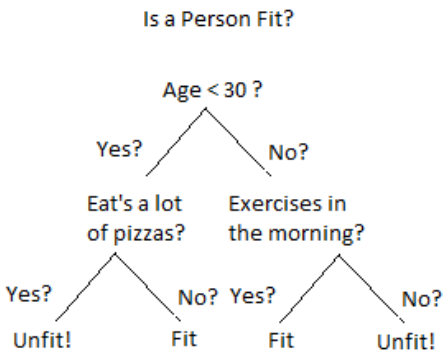


Figura 2.1: Árbol de decisión.

El aprendizaje se basa en dividir el conjunto de entradas en subconjuntos basándose en el valor de cada entrada. Este proceso se repite en cada subconjunto y termina cuando el todo el subconjunto en un nodo tiene el valor de la salida deseada o cuando la división ya no añade más valor a las predicciones.

Reglas de asociación

En aprendizaje automático, las reglas de asociación se utilizan para descubrir hechos que ocurren en común dentro de un determinado conjunto de datos y relaciones interesantes entre variables [5]. El problema de reglas de asociación se define como:

- Sea $I = \{i_1, i_2, ..., i_n\}$ un conjunto de n atributos binarios llamados items
- Sea $D = \{t_1, t_2, ..., t_m\}$ un conjunto de transacciones almacenadas en una base de datos.

Cada transacción en D tiene un identificador único y contiene un subconjunto de items de I. Una regla se define como:

$$X \Rightarrow Y \tag{2.1}$$

Donde X e Y pertenecen al conjunto de datos I. Un ejemplo práctico de este algoritmo pueden ser las ventas en un supermercado. El conjunto

de items para el ejemplo sería $I = \{Leche, Pan, Mantequilla, Cerveza\}$, mientras que la Tabla 2.1 muestra una pequeña base de datos que contiene los items, donde 1 significa que el item esta presente en la transacción y 0 que no esta presente.

ID	Leche	Pan	Mantequilla	Cerveza
1	1	1	0	0
2	0	1	1	0
3	0	0	0	1
4	1	1	1	0
5	0	1	0	0

Tabla 2.1: Base de datos con 4 items y 5 transacciones.

Un ejemplo de regla para el supermercado podría ser $\{Leche, Pan\} \Rightarrow \{Mantequilla\}$. Significaría que si el cliente compro leche y pan, también compró mantequilla.

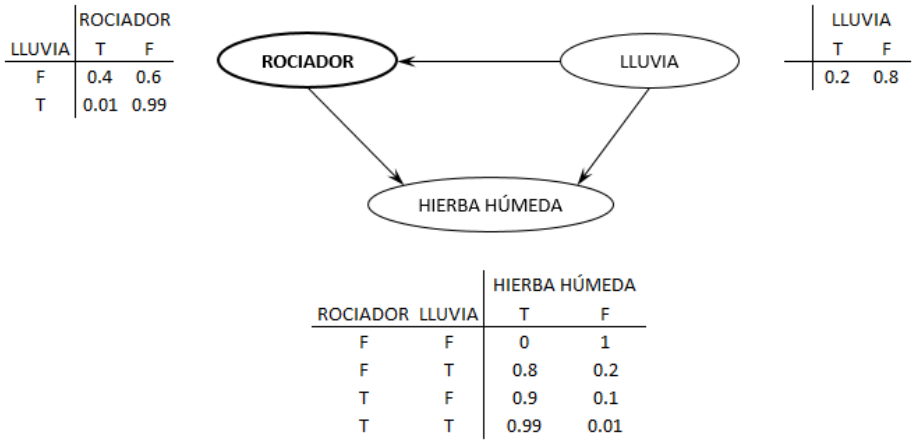
Redes Neuronales Artificiales

Las redes de neuronas artificiales se inspiran en las neuronas de los sistemas nerviosos de los animales. Se trata de un sistema de enlaces de neuronas que colaboran entre sí para producir un estímulo de salida. Las conexiones tienen pesos numéricos que se adaptan en función de si la salida es correcta o no. De esta manera, las redes neuronales se adaptan a un impulso y son capaces de aprender [6].

Esta forma de aprendizaje es la que ha sido utilizada en este trabajo y se explica con detalle en la Sección 2.2.

Redes bayesianas

Una red bayesiana, red de creencia o modelo acíclico dirigido es un modelo probabilístico que representa una serie de variables de azar y sus independencias condicionales a través de un grafo acíclico dirigido [8]. En la Figura 2.2 se muestra una red bayesiana con tres variables que representan tres sucesos diferentes: que haya llovido, que se haya activado el rociador y que la hierba este húmeda. Las tres variables tienen dos posibles valores, verdadero (T) o falso (F). En la Figura 2.2 se muestran además las probabilidades de cada suceso.



LLUVIA	ROCIADOR T	ROCIADOR F
F	0.4	0.6
T	0.01	0.99

LLUVIA T	LLUVIA F
0.2	0.8

ROCIADOR	LLUVIA	HIERBA HÚMEDA T	HIERBA HÚMEDA F
F	F	0	1
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

Figura 2.2: Red bayesina simple.

Hay dos eventos que provocan que la hierba este húmeda: que el rociador este activado o que este lloviendo. Generalmente el rociador se encuentra apagado cuando llueve, por lo que la lluvia tiene un efecto directo sobre el rociador y también hace que la hierba este húmeda. Esta situación se modela con la red de la Figura 2.2. El modelo puede responder mediante la ecuación de Bayes a preguntas como "¿Cuál es la probabilidad de que este lloviendo dado que la hierba esta húmeda?".

$$P(L|H) = \frac{P(H|L)P(L)}{P(H)} \tag{2.2}$$

Donde $P(H|L)$ es la probabilidad de que la hierba este húmeda dado que esta lloviendo, $P(L)$ es la probabilidad de que este lloviendo, y $P(H)$ es la probabilidad de que la hierba este húmeda.

2.2. Redes Neuronales Artificiales

Los conceptos presentados en esta sección han sido extraídos de [9]. Una red neuronal artificial es un algoritmo de aprendizaje inspirado en las neuronas biológicas. Las unidades básicas de procesamiento de datos del cerebro son las neuronas, cuyo funcionamiento general se basa en productos químicos (iones) que aumentan o disminuyen el potencial eléctrico dentro del cuerpo de la neurona. Si este potencial alcanza un umbral, la neurona se dispara y un pulso de intensidad y duración fijas se envía a través del axón. Los axones se dividen en conexiones a muchas otras neuronas, conectándose a cada una de ellas en una sinapsis. Cada neurona puede ser vista como un

procesador, el cual realiza una operación muy sencilla: decidir si activarse o no. Esto convierte al cerebro en un enorme computador compuesto por aproximadamente 10^{11} elementos de procesamiento. Por lo tanto es posible modelar este comportamiento dentro de una computadora y conseguir emular inteligencia animal o humana dentro de un ordenador. Esta es la visión de una inteligencia artificial robusta.

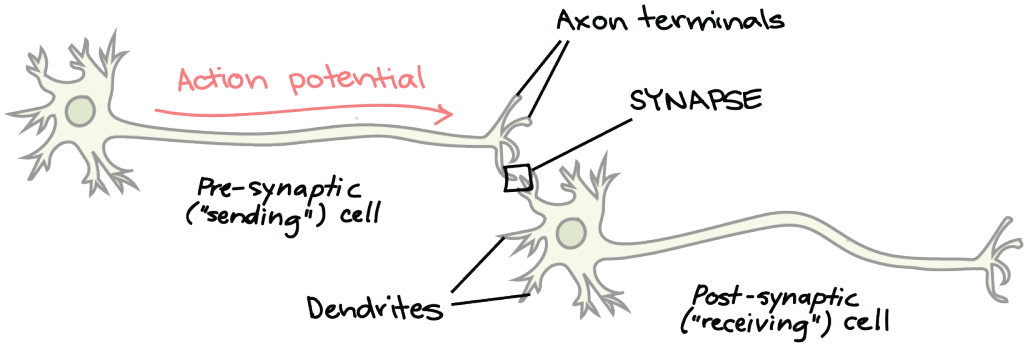


Figura 2.3: Estructura de una neurona.

2.2.1. ¿Cómo aprende el cerebro?

El concepto principal a tener en cuenta es la **plasticidad** o capacidad de modificar la intensidad de las conexiones sinápticas entre las neuronas y crear nuevas conexiones. No se conocen todos los mecanismos mediante los cuales se adapta la fuerza de estas sinapsis, pero un método que parece ser usado fue el postulado por Donald Hebb en 1949.

La regla de Hebb dice que los cambios en la fuerza de las conexiones sinápticas es proporcional a la correlación en la activación de dos neuronas conectadas. Por lo tanto, si dos neuronas se disparan a la vez constantemente, cualquier conexión entre ellas cambiará de intensidad, haciéndose más fuerte. Sin embargo, si las dos neuronas nunca se activan simultáneamente, la conexión entre ellas desaparecerá. La idea es que si dos neuronas responden a un evento, las mismas deberían estar conectadas.

Modelo matemático de una neurona

En 1943, McCulloch y Pitts introdujeron un modelo matemático de una neurona que consistía en:

- Un conjunto de entradas ponderadas con peso w_i, que corresponden a las sinapsis.

- Un sumador, para sumar las señales de entrada ponderadas, equivalente a la membrana de la célula que reune la carga eléctrica.
- Una función de activación, que decide si la neurona se activa para las entradas actuales.

Las entradas, x_i, son multiplicadas por los pesos, w_i, y la neurona suma sus valores. Si el resultado de esta suma es mayor que un umbral, θ, la neurona se activa.

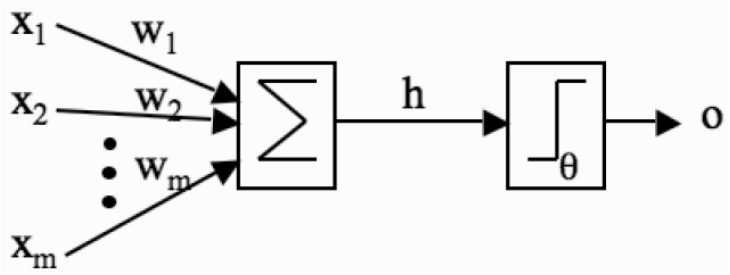


Figura 2.4: Modelo matemático de McCulloch y Pitts para una neurona.

El modelo se muestra en la Figura 2.4. En la parte izquierda del mismo se encuentra un conjunto de nodos de entrada, x_i. Para el siguiente ejemplo, se utilizaran tres entradas, $x_1 = 1$, $x_2 = 0$, $x_3 = 0{,}5$. En las neuronas reales, estas entradas provienen de las salidas de otras neuronas. Cada una de estas activaciones de la neurona anterior fluye a lo largo de una sinapsis para llegar a la neurona actual. Estas sinapsis tienen diferentes intensidades, llamadas pesos. La fuerza de la sinapsis afecta a la intensidad de la señal, por lo que cada entrada se multiplica por el peso de su sinapsis correspondiente ($x_1 \times w_1$, $x_2 \times w_2$, $x_3 \times w_3$). Todas estas entradas se suman al llegar a la neurona actual. Esto se puede escribir como:

$$h = \sum_{i=1}^{m} w_i x_i \tag{2.3}$$

Donde $m = 3$ para este ejemplo. Si los pesos sinápticos son $w_1 = 1$, $w_2 = -0{,}5$, $w_3 = -1$, entonces las entradas de la neurona serán: $h = 1 \times 1 + 0 \times (-0{,}5) + 0{,}5 \times (-1) = 0{,}5$. Ahora la neurona tiene que decidir si se activa o no. Para una neurona real, esta es una cuestión de si el potencial de membrana está por encima de un cierto umbral. En este ejemplo se ha seleccionado valor de umbral $\theta = 0$. Dado que $h = 0{,}5$ y $0{,}5 > 0$, entonces la neurona se activa, y produce un 1 como salida. Si la neurona no se activase, la salida sería 0.

En resumen, la neurona de McCulloch y Pitts es un dispositivo de umbral binario. Este suma las entradas (multiplicadas por los pesos) y se activa (produce salida 1) o no se activa (produce salida 0) dependiendo de si el valor de h está por encima de un umbral. La segunda mitad del trabajo de la neurona, lo que se conoce como una función de activación, se puede escribir como:

$$o = g(h) = \begin{cases} 1 & si \quad h \geq 0 \\ \\ 0 & si \quad h \leq 0 \end{cases} \tag{2.4}$$

2.2.2. El Perceptron

La cuestión que se plantea ahora es como pueden aprender las neuronas. Por lo general, se utiliza el *aprendizaje supervisado*, es decir, un algoritmo que aprende a base de ejemplos. En este caso, el conjunto de datos utilizado para entrenar a la red contiene un conjunto de entradas con sus salidas esperadas asociadas. De esta forma se intenta conseguir que la red obtenga un modelo o patrón y predecir correctamente otros ejemplos con los que no haya sido entrenada.

Volviendo a observar la neurona de McCulloch y Pitts y pensando en qué parte de ella puede variar, se observa que las entradas no se pueden variar (ya que proceden de una fuente externa) por lo que solo se pueden modificar los pesos y el umbral. Por lo tanto, la mayor parte del entrenamiento ocurre en los pesos, los cuales en realidad no forman parte de la neurona, si no que representan a las sinapsis. Por lo tanto, el aprendizaje ocurre entre las neuronas, en la forma en la que están conectadas.

El Perceptron no es más que una colección de neuronas de McCulloch y Pitts junto con un conjunto de entradas y algunos pesos para graduar las entradas a las neuronas. La red Perceptron genérica se muestra en la Figura 2.5. Los nodos de entrada, sombreados en gris, se encuentran a la izquierda de la figura. Los nodos de entrada no son neuronas, simplemente representan como las entradas son introducidas en la red y cuantos de estos valores de entrada hay. Las neuronas se muestran en la parte derecha de la figura, compuestas por la parte aditiva (el círculo negro) y la función umbral (caja con la función escalón).

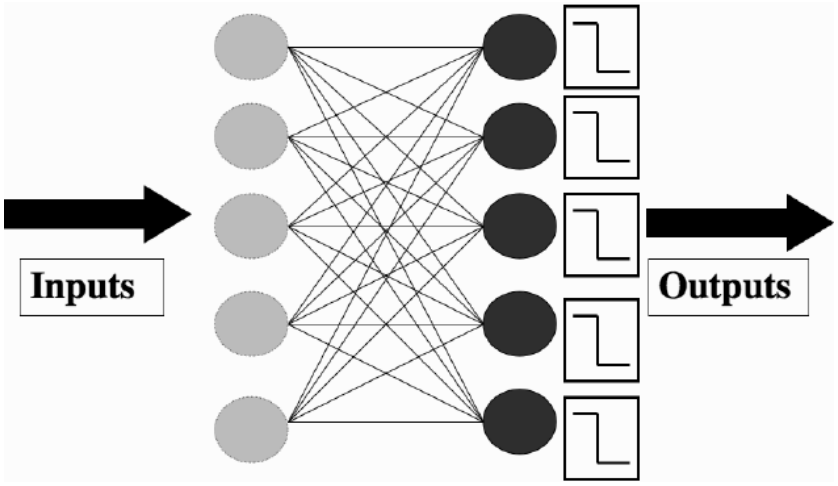


Figura 2.5: La red Perceptron.

Las neuronas en el Perceptron son independientes unas de otras: a ninguna neurona le importa lo que las otras están haciendo. Las neuronas deciden si activarse o no multiplicando sus propios pesos por las entradas, sumando los resultados y comparando el resultado con su propio umbral. Lo único que comparten las neuronas son la entradas, ya que cada neurona utiliza todas las entradas de la red. En la Figura 2.5 el número de entradas es el mismo que el número de neuronas, pero esto no tiene por que ser así siempre. Generalmente un Perceptron tiene m entradas y n neuronas.

En la primera versión de la neurona de McCulloch y Pitts, los pesos se etiquetaron como w_i, donde el índice i recorría el número de entradas. En este caso, también hay que determinar a qué neurona corresponde cada peso, por lo que los pesos se han etiquetado como w_{ij}, donde el índice j recorre el número de neuronas. Por ejemplo, w_{32} es el peso que conecta el nodo de entrada 3 con la neurona 2.

Para hacer que una neurona se active o no, hay que establecer los valores de los nodos de entrada para que coincidan con los elementos de un vector de entrada y luego usar las ecuaciones 2.3 y 2.4 para cada neurona. Si se hace esto para todas las neuronas el resultado es un patrón de neuronas activadas y desactivadas, que es representado por un vector de 0s y 1s. Por lo tanto, si hay 5 neuronas, como en la figura 2.5, un posible patrón de salida podría ser [0, 1, 0, 0, 1], que significa que la segunda y quinta neurona se han activado y las otras no. Este patrón se compara con la respuesta correcta para esta entrada con objeto de identificar qué neuronas obtuvieron la respuesta correcta y cuáles no.

Cualquier neurona que se haya activado cuando no debía, o que no se haya activado cuando sí debía, tiene que modificar los valores de sus pesos.

El problema se encuentra en que es imposible saber cuáles son los nuevos valores que deben ser asignados a los pesos de las neuronas que han fallado.

Para llegar a una solución, lo primero que hay que mirar es si cada peso es demasiado pequeño o demasiado grande. Algunos pesos pueden que sean muy grandes si la neurona se ha activado cuando no debía y muy pequeños si no se ha activado cuando debía hacerlo. Por lo tanto, se realiza la operación $y_k - t_k$ (la diferencia entre y_k, que representa la salida de la neurona, y t_k, que es el resultado que la neurona debería haber obtenido) donde k es el número de neuronas que han fallado. Si el resultado es negativo, la neurona debería haberse activado pero no lo hizo, así que se debe aumentar el valor de los pesos. Por el contrario, si el resultado es positivo, la neurona se activo pero no debería haberlo hecho, por lo que se debe disminuir los valores de los pesos. En caso de que la entrada fuese negativa, cambiaría los valores, por lo que si queremos que la neurona se active, los valores de los pesos también deberán ser negativos. Para ello, se obtiene un nuevo valor del peso: $\Delta w_{ik} = -(y_k - t_k) \times x_i$. El nuevo valor del peso sera el valor antiguo más este valor.

Ahora queda decidir cuanto se aumenta o se disminuye el valor del peso. Esto se hace multiplicando Δw_{ik} por un parámetro llamado *learning rate*, normalmente denotado por η. El valor del *learning rate* indica la velocidad con la que un modelo aprende, y normalmente se suele fijar su valor entre 0.1 y 0.4. Finalmente, el nuevo valor del peso vendrá dado por la siguiente expresión.

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{2.5}$$

La entrada de offset

Como se ha visto antes, cada neurona tiene asociada un umbral, θ, que determina el valor a partir del cual la neurona se activa. Este valor debe ser ajustable.

Cambiar el valor del umbral supone un problema, ya que requiere añadir una variable adicional para la que habría que escribir más código. Supongamos que se fija el valor del umbral para la neurona en cero. Ahora, se agrega un peso de entrada adicional a la neurona, con el valor de la entrada a ese peso siendo fijo siempre (para este ejemplo se ha usado el valor -1, pero cualquier valor distinto de cero es válido). Este peso es incluido en el algoritmo de actualización (como todos los demás pesos). El valor del peso cambiará para hacer que la neurona se active o no se active, según sea necesario.

Esta entrada se llama nodo de offset, y sus pesos son denominados normalmente con un subíndice 0. Por lo tanto el peso que lo conecta con la neurona

j es w_{0j}. La Figura 2.6 muestra la red Perceptron después de añadirle la entrada de offset.

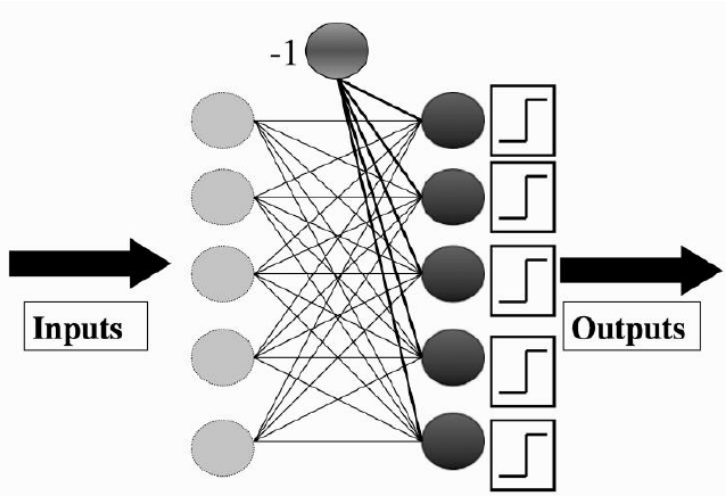


Figura 2.6: La red Perceptron con el nodo de offset.

El algoritmo de aprendizaje del Perceptron

El algoritmo de aprendizaje se divide en dos partes: una fase de entrenamiento y una fase de recuperación. La fase de entrenamiento usa la ecuación de recuperación, ya que tiene que calcular las activaciones de las neuronas antes de que se pueda calcular el error y entrenar los pesos. El algoritmo incluye los siguientes pasos.

1. Un vector de entrada se introduce en los nodos de entrada.
2. Las entradas y los pesos son utilizados parar decidir si las neuronas de salida se activan o no.
3. El error es calculado como la diferencia entre las salidas de la red y los objetivos.
4. Este error se utiliza para actualizar los pesos de la segunda capa.

A continuación, se presenta el algoritmo del Perceptron de forma detallada con las ecuaciones a utilizar en cada paso.

Algoritmo del Perceptron

- **Inicialización**
  - Establecer todos los pesos w_{ij} a números aleatorios pequeños (positivos y negativos)
- **Entrenamiento**
  - Para T iteraciones o hasta que todas las salidas sean correctas.
    - Para cada vector de entrada:
      - Calcular la activación de cada neurona j utilizando la función de activación g:

$$y_j = g\left(\sum_{i=1}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{si } \sum_{i=0}^{m} w_{ij}x_i \geq 0 \\ 0 & \text{si } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases} \tag{2.6}$$

      - Actualizar cada uno de los pesos individualmente utilizando:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{2.7}$$

- **Recuperación**
  - Calcular la activación de cada neurona j usando:

$$y_j = g\left(\sum_{i=1}^{m} w_{ij}ix_i\right) = \begin{cases} 1 & \text{si } \sum_{i=0}^{m} w_{ij}x_i \geq 0 \\ 0 & \text{si } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases} \tag{2.8}$$

Separabilidad lineal

Con objeto de mostrar lo que realmente calcula el Perceptron se ha utilizado la función lógica OR, cuya red se muestra en la Figura 2.7. El objetivo es separar los casos en los que la neurona debe activarse de los que no.

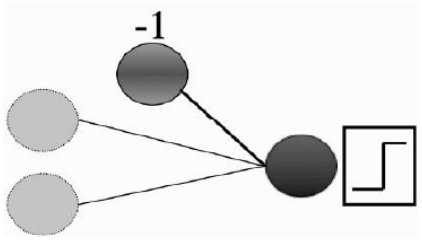


Figura 2.7: La red Perceptron para la función lógica OR.

Observando el gráfico en la parte derecha de la Figura 2.8, es posible dibujar una línea recta que separe los círculos de las cruces sin ningún problema. Esto es, de hecho, lo que hace el perceptron, intentar encontrar una línea recta donde las neuronas se activen a un lado de ella y no se activen al otro. Esta línea se llama *límite de decisión* o *función discriminante*.

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	1

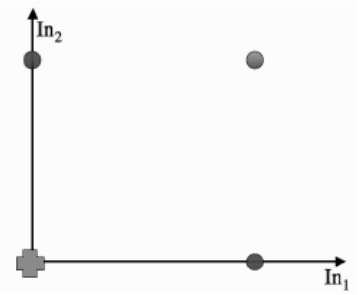


Figura 2.8: Representación de la tabla de verdad para la función lógica OR.

Por lo tanto, dados algunos datos de entrada, y los resultados objetivo asociados, el Perceptron simplemente trata de encontrar una línea recta que divida las situaciones donde cada neurona se activa de aquellas donde no lo hace. Los casos en los que es posible separar los resultados con una línea recta se llaman casos *linealmente separables*. Si hay más de una neurona de salida, los pesos para cada neurona describen por separado una línea recta, por lo que al juntar varias neuronas obtenemos varias líneas rectas que intentan separar diferentes partes del espacio. La Figura 2.9 muestra un ejemplo de límites de decisión calculados por un Perceptron con cuatro neuronas; al juntarlos se obtiene una buena separación de las clases.

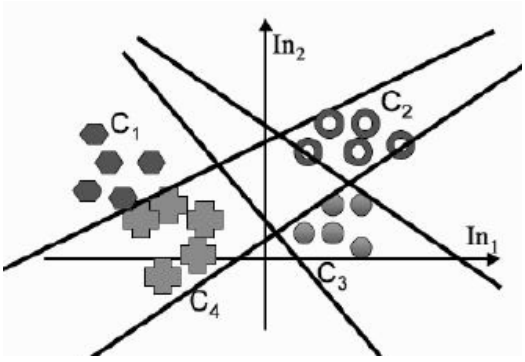


Figura 2.9: Diferentes límites de decisión calculados por un Perceptron con cuatro neuronas.

2.2.3. El Perceptron Multicapa

La mayoría de los problemas para los que se utiliza Machine Learning no son linealmente separables. Como se ha visto anteriormente, el aprendizaje en la red neuronal ocurre en los pesos. Entonces, para realizar más cálculos, parece lógico agregar más pesos. Hay dos cosas que se pueden hacer: agregar algunas conexiones hacia atrás (para que las neuronas de salida se conecten nuevamente a las entradas) o agregar más neuronas. El primer enfoque conduce a redes recurrentes. Mientras que el segundo consiste en agregar neuronas entre los nodos de entrada y las salidas, lo que crea redes neuronales más profundas y complejas, como la que se muestra en la Figura 2.10.

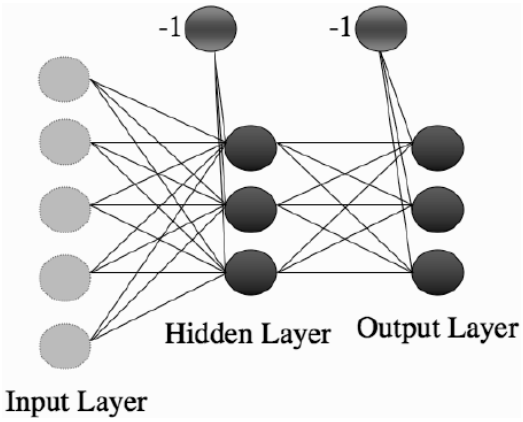


Figura 2.10: La red Perceptron multicapa.

Para comprobar que añadir más capas es una solución a los problemas no lineales, se ha tomado la función lógica XOR como ejemplo. La Figura 2.11 muestra los datos de entrada de la XOR representados en un gráfico. Se puede ver como es imposible dibujar una línea recta que separe los círculos de las cruces en la Figura 2.11.

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	0

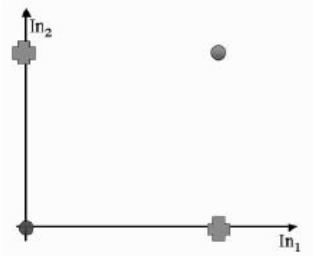


Figura 2.11: Representación de los datos para la función lógica XOR.

Un modelo apropiado para la función lógica XOR se muestra en la Figura 2.12. Para verificar que devuelve las respuestas correctas, todo lo que hay que hacer es introducir cada entrada y tratar la red como dos Perceptrones diferentes, primero calculando las activaciones de las neuronas en la capa intermedia (etiquetadas como C y D en la Figura 2.12) y luego usar esas activaciones como las entradas a la neurona individual en la salida.

La entrada (1, 0) corresponde a cuando el nodo A es 1 y el nodo B es 0. La entrada a la neurona C es por lo tanto $-1\times0{,}5+1\times1+0\times1 = -0{,}5+1 = 0{,}5$. Como está por encima del valor umbral, y la neurona C se activa, dando salida 1. Para la neurona D, la entrada es $-1 \times 1 + 1 \times 1 + 0 \times 1 = -1 + 1 = 0$, y por lo tanto no se dispara, dando salida 0. Por lo tanto, la entrada de la neurona E es $-1 \times 0{,}5 + 1 \times 1 + 0 \times -1 = 0{,}5$, por lo que la neurona E se activa. Comprobando el resultado para el resto de combinaciones de la tabla de verdad, se puede concluir que la neurona E se activa cuando las entradas A y B son diferentes entre sí, pero no se activa cuando son iguales, que es exactamente la función XOR.

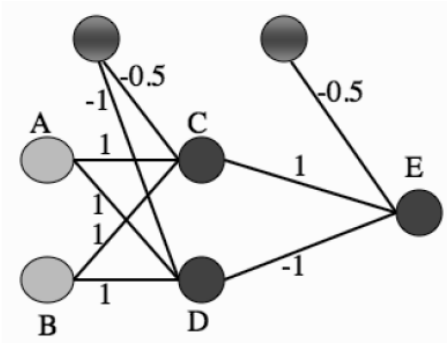


Figura 2.12: Red multicapa que resuelve el problema de la XOR.

Si para entrenar esta red se utiliza el método usado con el Perceptron de una sola capa, habría que calcular el error a la salida. Esto se puede realizar sin ningún problema, ya que se conocen los resultados objetivo, y por lo tanto es posible calcular la diferencia entre la salida obtenida y el objetivo deseado. Pero, sería imposible saber si los pesos que hay que modificar son los de la primera capa o los de la segunda. Esto hace que a las neuronas en mitad de la red se las llame *capas ocultas*, ya que es imposible examinar y corregir sus valores directamente.

Al igual que para el Perceptron de una sola capa, el entrenamiento del Perceptron multicapa consiste en dos partes: averiguar cuáles son las salidas para las entradas dadas y los pesos actuales, y luego actualizar los pesos de acuerdo con el error, que es una función de la diferencia entre las salidas y

los resultados objetivo. De esta forma, volviendo a observar la figura 2.12, se empezaría por la izquierda, insertando los valores de las entradas. Estas, junto con el primer nivel de pesos, se utilizan para calcular las activaciones de la capa oculta, y luego se usan esas activaciones y el siguiente conjunto de pesos para calcular las activaciones de la capa de salida. una vez se tienen los resultados de la red, se comparan con los resultados objetivo para calcular el error.

Entrada de offset

Es necesario incluir una entrada de offset a cada neurona. Esto se hace de la misma manera que se hizo para el Perceptron en la Sección 2.2.2: tener una entrada adicional que se establece permanentemente en -1, y ajustar los pesos de cada neurona como parte del entrenamiento. Por lo tanto, cada neurona de la red (ya sea de una capa oculta o de la capa de salida) tiene una entrada adicional, con un valor fijo distinto de 0.

Cáculo del error

En el Perceptron de una sola capa, los pesos eran modificados de forma que las neuronas se activasen cuando debían y no se activasen cuando no debían. El error para cada neurona k era calculado con la siguiente función: $E_k = y_k - t_k$. El objetivo era que cada iteración del algoritmo de aprendizaje disminuyese el valor del error. Como en aquel caso solo había una capa de pesos, esto era suficiente para entrenar la red.

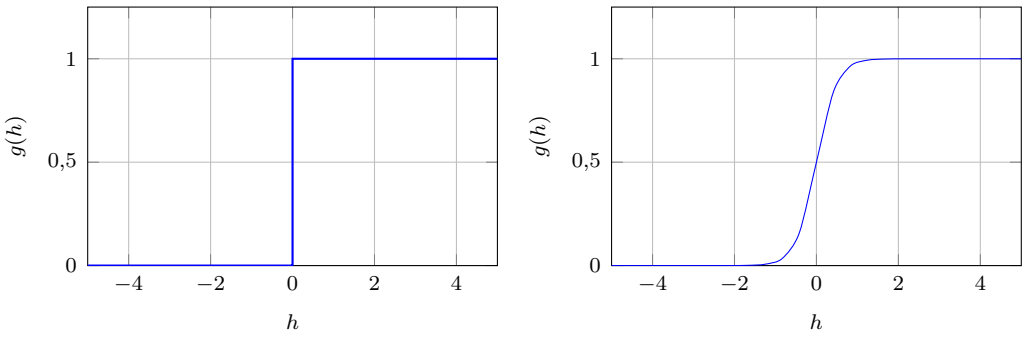
Al añadir más capas de pesos, calcular el error se vuelve más difícil. El problema se encuentra en que, al intentar adaptar los pesos del Perceptron multicapa, hay que averiguar qué capa de pesos ha causado el error. Este nivel puede ser la capa de pesos que conecta las entradas con la capa oculta, o la que conecta la capa oculta con la de salida.

La función de error utilizada en el Perceptron es: $\sum_{k=1}^{N} E_K = \sum_{k=1}^{N} y_k - t_k$, donde N es el número de nodos de salida. Se pueden producir dos tipos de errores: en el primero, el objetivo es más grande que la salida, mientras que en el segundo el resultado es más grande que el objetivo. Si estos dos errores tienen el mismo valor absoluto, entonces su suma vale 0, lo que significaría que no se ha cometido ningún error. Para evitar esto, hay que hacer que todos los errores tengan el mismo signo. La mejor manera de hacerlo es con la función de error de suma de cuadrados, que calcula el cuadrado de la

diferencia entre y y t para cada nodo, y los suma todos juntos:

$$E(t, y) = \frac{1}{2} \sum_{k=1}^{N} (y_k - t_k)^2 \tag{2.9}$$

El $\frac{1}{2}$ al principio de la ecuación 2.9 facilita la tarea de derivar la función. Al derivar una función, se obtiene su gradiente, lo que representa la dirección hacia la cual la función aumenta y disminuye más. Por lo tanto, si se deriva una función de error, se obtiene el gradiente del error. Debido a que el propósito del aprendizaje es minimizar el error, seguir la función de error en la dirección hacia la cual disminuye (es decir, la dirección del gradiente negativo), hará que el error disminuya hasta llegar a un mínimo local. Como lo único que se modifica durante el entrenamiento son los pesos, la función de error se deriva respecto de estos.



(a) Función umbral

(b) Función sigmoide

Figura 2.13: Funciones de activación

La función de activación utilizada en el Perceptron de una sola capa (Figura 2.13a) es discontinua en 0, por lo que es imposible de derivar en ese punto. El problema es que ese salto entre activarse o no activarse es necesario para que actúe como una neurona. Esto se puede resolver con una función de activación que se parezca a la función umbral pero que se pueda derivar, y de esta forma obtener el gradiente. Si se observa el gráfico de la función umbral mostrado en la Figura 2.13a, esta tiene una forma parecida a la de una S. Hay un conjunto de funciones matemáticas con forma de S, llamadas funciones *sigmoide* (ver Figura 2.13b). La forma más utilizada de esta función (donde β es un parámetro positivo) es:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)} \tag{2.10}$$

Por lo tanto, el Perceptron multicapa tiene una nueva forma de cálculo de errores (Ecuación 2.9) y una nueva función de activación (Ecuación 2.10), que decide si una neurona debe o no activarse. Para entrenar la red se debe calcular el gradiente de los errores y usarlos para decidir cuánto actualizar cada peso en la red. Esto se hará primero para los nodos conectados a la capa de salida, y después de haberlos actualizado, se trabajará hacia atrás a través de la red hasta llegar a las entradas. Esta aproximación tiene dos problemas:

- Para las neuronas de salida las entradas son desconocidas.
- Para las neuronas ocultas no se conocen los resultados objetivo.

Calcular el error en la salida es posible, pero como se desconocen las entradas que lo causaron, no se pueden actualizar los pesos de la segunda capa de la misma manera que en el Perceptron de una sola capa. Este problema se puede evitar utilizando la regla de la cadena de derivación. Aplicar la regla de la cadena implica que para saber cómo cambia el error a medida que varían los pesos, se puede estudiar cómo cambia el error a medida que cambian las entradas a los pesos, y multiplicar esto por cómo cambian esos valores de entrada cuando se varían los pesos de la capa anterior.

En resumen, los gradientes de los errores se calculan con respecto a los pesos, de modo que permitan actualizar los pesos siguiendo la dirección hacia la que el gradiente disminuye, lo que hace que los errores se vuelvan más pequeños. Para ello se deriva la función de error con respecto a los pesos, pero esto no se puede hacer directamente, por lo que hay que aplicar la regla de la cadena y derivar con respecto a las variables conocidas. Esto lleva a dos o más ecuaciones de actualización diferentes, una para cada uno de los conjuntos de pesos, que se aplican "hacia atrás" a través de la red, comenzando en las salidas y terminando en las entradas.

Algoritmo del Perceptron multicapa

Para describir el algoritmo del Perceptron multicapa, se han asumido L nodos de entrada, M nodos ocultos y N nodos de salida. Por lo tanto, si se incluye el offset, hay $(L + 1) \times M$ pesos entre la entrada y la capa oculta, y $(M + 1) \times N$ pesos entre la capa oculta y la salida. Las sumas descritas empiezan desde 0 si incluyen el nodo de offset y desde 1 si no lo incluyen, y llegan hasta L, M, o N, de manera que $x_0 = -1$ es el offset de entrada, y $a_0 = -1$ es el nodo de offset oculto. El algoritmo descrito es válido para cualquier número de capas ocultas. En este caso habrá varios valores para M y más conjuntos de pesos entre las capas ocultas. También se han

utilizado i, j, k, para indicar los nodos de cada capa en las sumas, y sus correspondientes letras griegas (ι, ζ, κ) para los índices fijados. El algoritmo incluye los siguientes pasos:

1. Un vector de entrada se introduce en los nodos de entrada.
2. Las entradas se "propagan" a través de la red.
   - Las entradas y los pesos de la primera capa (denominados con υ) son utilizados para decidir si los nodos ocultos se activan o no.
   - Las salidas de estas neuronas y los pesos de la segunda capa (denominados con w) se utilizan para decidir si las neuronas de salida se activan o no.
3. El error es calculado como la suma de cuadrados entre las salidas de la red y los objetivos.
4. Este error se propaga "hacia atrás" a través de la red para:
   - Actualizar los pesos de la segunda capa.
   - Después, actualizar los pesos de la primera capa.

A continuación, se explica el algoritmo del Perceptron multicapa detalladamente, indicando las ecuaciones a utilizar en cada paso.

Algoritmo del Perceptron Multicapa

- **Inicialización**
  - Establecer todos los pesos a números aleatorios pequeños (positivos y negativos)
- **Entrenamiento**
  - Repetir:
    - Para cada vector de entrada:

      Fase "hacia delante":

      · Calcular la activación de cada neurona j en la capa o capas ocultas:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{2.11}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{2.12}$$

      · Avanzar a través de la red hasta obtener las salidas, con las activaciones:

$$h_\kappa = \sum_{j} a_j w_{j\kappa} \tag{2.13}$$

$$a_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{2.14}$$

      Fase "hacia atrás":

      · Calcular el error a la salida utilizando:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (1 - y_\kappa) \tag{2.15}$$

      · Calcular el error en las capas ocultas utilizando:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{2.16}$$

      · Actualizar los pesos de la capa de salida:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta \tag{2.17}$$

      · Actualizar los pesos de la capa oculta:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{2.18}$$

- **Recuperación**
  - Utilizar las ecuaciones de la fase 'hacia delante".

2.3. Software

Actualmente existen más de 30 entornos, librerías o programas dedicados al desarrollo de aplicaciones de Machine Learning. Esta sección se centra en las librerías de TensorFlow y Caffe, ya que además de ser muy empleadas, han sido las utilizadas para realizar este trabajo.

2.3.1. Tensorflow

TensorFlow es una librería de código abierto para Machine Learning desarrollada por Google para satisfacer sus necesidades de implementar y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. Actualmente es utilizado tanto para la investigación como para la producción de productos de Google [10].

La librería de TensorFlow esta disponible en Python, C++, Java, Go, y Swift. En la versión actual, el único lenguaje en el que la librería es completamente estable y contiene todas las funciones es Python [11].

Tensores

TensorFlow es un entorno para definir y ejecutar operaciones con tensores. Un tensor es un conjunto de vectores y matrices. Internamente, TensorFlow representa los tensores como arrays de dimensión N [12]. Un tensor tiene las siguientes propiedades:

- El tipo de datos (*float32*, *int32*, o *string*, por ejemplo) de los elementos del tensor. Cada elemento del tensor tiene el mismo tipo de datos.
- El número de dimensiones y el tamaño de cada dimensión. El número de dimensiones define el rango del tensor, por lo que cada tipo de rango corresponde a una estructura de datos diferente

Rango	Entidad
0	Escalar (solo magnitud)
1	Vector (magnitud y dirección)
2	Matriz
3	Imagen
4	Vector de imágenes

Tabla 2.2: Estructuras de datos en función del rango del tensor.

Como se indica en la tabla 2.2, una imagen se almacenada en formato RGB y representa un tensor de rango 3. Es decir, tiene tres dimensiones: el alto, el ancho y los canales de la imagen, en ese orden. Si en vez de una sola imagen, se trata de un vector de imágenes, el tensor pasa a ser de rango 4. Por lo tanto, el número de dimensiones es 4 y el orden de las mismas es NHWC (número de imágenes, alto, ancho y canales).

Grafo

Un grafo computacional es una serie de operaciones de TensorFlow organizadas en un grafo [13]. Este se compone de dos tipos de objetos:

- **Operaciones:** Son los nodos del grafo. Las operaciones describen cálculos que computan y devuelven tensores.
- **Tensores:** Representan los valores que fluirán a través del grafo.

A continuación se muestra un grafo computacional simple donde se realiza la operación más básica, la suma. Los tensores *a* y *b* son las entradas, y *total* es la suma de ambos.

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0) # also tf.float32 implicitly
total = a + b
print(a)
print(b)
print(total)
```

Las tres funciones *print* producen el mismo resultado:

```
Tensor("Const:0", shape=(), dtype=float32)
Tensor("Const_1:0", shape=(), dtype=float32)
Tensor("add:0", shape=(), dtype=float32)
```

Como se puede observar, la función *print* no devuelve los valores 3.0, 4.0 y 7.0 como sería de esperar. Las instrucciones anteriores construyen el grafo computacional pero no ejecutan sus acciones. Estos tensores solamente representan las operaciones que serán ejecutadas.

Para visualizar el grafo, TensorFlow cuenta con una herramienta llamada TensorBoard [14]. Esta aplicación permite ver todas las conexiones entre tensores y operaciones. En la figura 2.14 se puede ver el grafo correspondiente a la operación de la suma.

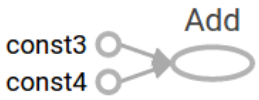


Figura 2.14: Grafo correspondiente a la operación suma.

Capas

Un modelo de red neuronal debe modificar los valores del grafo para obtener nuevas salidas con la misma entrada. Una forma de mejorar la capacidad de modelado de comportamientos y capacidad de aprendizaje es utilizar capas de forma que se añaden parámetros entrenables al grafo [15].

Las capas agrupan las variables y las operaciones que actúan sobre ellas. Por ejemplo, la capa *Dense* realiza una suma ponderada en todas las entradas para cada salida y aplica una función de activación opcional. Los pesos y bias son gestionados por la capa.

El siguiente código de ejemplo crea una capa *Dense* que toma un conjunto de vectores de entrada y produce un solo valor para cada uno. Para aplicar una capa a una entrada, se llama a la misma como si fuera una función.

```
x = tf.placeholder(tf.float32, shape=[None, 3])
linear_model = tf.layers.Dense(units=1)
y = linear_model(x)
```

Por lo tanto, un modelo es definido por una interconexión de capas y operaciones a través de los tensores de entrada y salida de cada capa u operación.

La librería de TensorFlow cuenta con más de 100 capas y operaciones, además de disponer de dos tipos de librerías diferentes, una para entrenar redes neuronales y otra de "runtime", con la que poder ejecutar o implementar redes ya entrenadas.

2.3.2. Caffe

Desarrollado por el Vision and Learning Centre de la universidad de Berkeley (BVLC) y usuarios de la comunidad, Caffe es un entorno para Machine Learning diseñado para ser rápido y modular. Fue creado por Yangqing Jia mientras realizaba su doctorado en Berkeley [16]. Esta escrito en C++, y tiene interfaces para Python, MATLAB y C++. [17]

Los modelos y la optimización se definen mediante la configuración con una codificación flexible. Se puede cambiar entre la CPU y la GPU configurando un solo flag para entrenar en una máquina en GPU y luego implementar en clusters de productos básicos o dispositivos móviles, por ejemplo [16].

Caffe puede procesar más de 60 millones de imágenes por día con una sola GPU NVIDIA K40, lo que se traduce en 1 ms/imagen para la implementación y 4 ms/imagen para el aprendizaje. Por ello, Caffe se encuentra entre las implementaciones de redes convolucionales más rápidas actualmente disponibles [16].

Almacenamiento de datos

Caffe almacena sus datos en arrays de 4 dimensiones llamados *blobs* (binary large object). Los *blobs* proporcionan una interfaz de memoria unificada, almacenando una o más imágenes (u otros datos), parámetros o actualizaciones de parámetros [17].

La primera dimensión es el número de imágenes (N) que hay en el *blob*, la segunda los canales (C) de la imagen, la tercera el alto (H) de la imagen y la última el ancho (W) de la imagen. De esta manera se define la estructura de un *blob* como NCHW, a diferencia de la del *tensor* de TensorFlow que es NHWC, como se ha visto en la sección 2.3.1.

Las imágenes son almacenadas en formato BGR, debido a que Caffe utiliza OpenCV para leer imágenes, y dicha librería guarda las imágenes en este formato. Para evitar la conversión del orden de los canales de la imagen, Caffe utiliza el mismo formato que OpenCV.

Capas

Las capas son la esencia de un modelo, y la unidad fundamental de computación. Cada capa toma uno o más *blobs* como entrada y devuelve uno o más *blobs* como salida [17, 18]. Las capas tienen generalmente tres funciones implementadas en el archivo C++ que las describe:

- **Setup:** Inicializa la capa y sus conexiones durante la inicialización del modelo.
- **Forward:** Calcula la salida de la capa actual y la envía a la capa siguiente.
- **Backward:** Calcula el error de la salida de la capa a partir del error de la capa siguiente y lo envía a la capa anterior.

Más específicamente, habrá dos funciones *Forward* y *Backward* implementadas por cada capa, una para la CPU y otra para la GPU. Si la capa no tiene implementada una versión de GPU, la capa usará las funciones de la CPU [18].

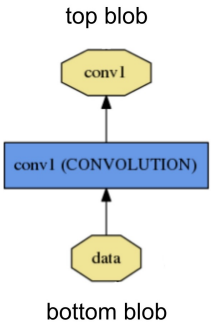
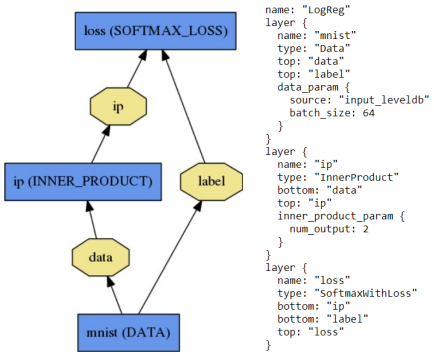


Figura 2.15: Las capas en Caffe toman las entradas a través de conexiones inferiores y devuelven las salidas hacia capas superiores.

Caffe proporciona un conjunto completo de tipos de capas que incluyen, entre otros: convolución, *pooling*, productos internos, no linealidades como rectificado lineal y logística, normalización de respuesta local, operaciones elemento a elemento y pérdidas como *softmax*. Además, Caffe posibilita desarrollar capas personalizadas: basta con escribirlas [17]. En total Caffe proporciona más de 50 tipos de capas diferentes.

Definición de la red

En Caffe, una red es definida como un conjunto de capas y sus conexiones en un modelo de texto plano. La Figura 2.16 muestra una red clasificadora y su definición. Esta red toma los datos (data) de entrenamiento de un dataset (mnist) y se los pasa a la capa *ip* que realiza un producto interno. El resultado se envía a la capa *loss*, donde se calcula el error utilizando los resultados esperados (label) y el resultado obtenido (ip).



(a) Red (b) Definición

Figura 2.16: Ejemplo de una red simple

Los modelos de Caffe están definidos con *protocol buffers*. Los *protocol buffers* son un mecanismo de Google para serializar datos estructurados y que sea posible utilizar estos datos entre diferentes lenguajes de programación [19]. Las redes de Caffe se definen en *protocol buffers* de texto plano (.prototxt), como se muestra en la Figura 2.16b. Por otro lado, los pesos aprendidos por el modelo son almacenados en *protocol buffers* binarios (binaryproto) de tipo .caffemodel [20].

Capítulo 3

Redes Neuronales para Procesado de Imagen

La red neuronal convolucional es el tipo de red neuronal más utilizada para el reconocimiento y procesamiento de imágenes. Este tipo de red es la que ha sido implementada en este trabajo.

3.1. Redes Neuronales Convolucionales

Las redes convolucionales son redes con cierta profundidad que están inspiradas en los procesos biológicos, en los que el patrón de conectividad entre neuronas se asemeja a la organización de la corteza visual de los seres vivos [21].

Las redes neuronales convolucionales son una variante del perceptron multicapa y usan relativamente poco preprocesamiento en comparación con otros algoritmos de clasificación de imágenes. Esto significa que la red procesa la información mediante filtros que en los algoritmos tradicionales eran diseñados a mano [21]. La red neuronal tiene capacidad de aprender, es decir, de calcular automáticamente los pesos de los filtros.

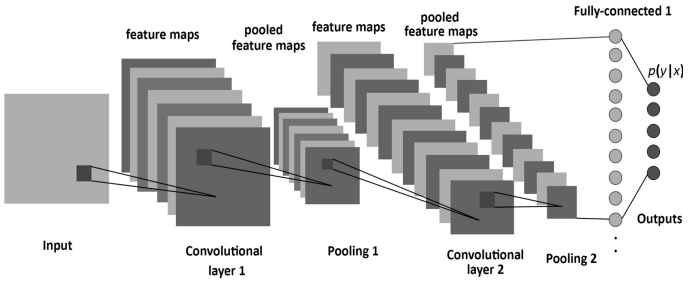


Figura 3.1: Arquitectura típica de una red neuronal convolucional.

Las capas ocultas de una red neuronal convolucional suelen integrar capas de convolución, *pooling*, normalización y capas completamente conectadas. En la Figura 3.1 se observa una posible arquitectura de una red neuronal convolucional [21].

Convolución

La capa de convolución es el elemento básico de una red convolucional. Los parámetros de la capa consisten en un conjunto de filtros (o núcleos) que son aprendidos por la misma durante el entrenamiento. Cada filtro se convoluciona con el elemento de entrada, calculando el producto escalar entre el filtro y la entrada y produciendo un mapa de activación bidimensional de ese filtro. Como resultado, la red calcula filtros que se activan cuando detecta algún tipo específico de función en alguna posición espacial en la entrada [21].

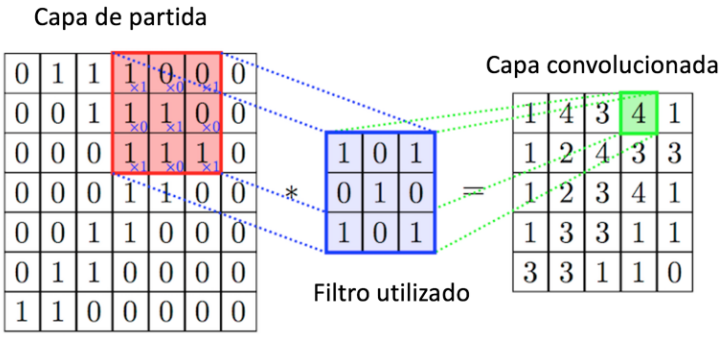


Figura 3.2: Capa de convolución.

Los mapas de activación de cada filtro, se apilan a lo largo de la dimensión de profundidad para formar el elemento de salida completo de la capa de convolución. La salida de cada capa convolucional se calcula como:

$$Y_j = g(b_j + \sum_i K_{ij} \otimes Y_i) \tag{3.1}$$

Donde la salida Y_j de la neurona j es una matriz que se calcula por medio de la combinación lineal de las salidas Y_i provenientes de las neuronas de la capa anterior, con el núcleo de convolución K_{ij}. A esta cantidad se le suma un offset, b_j, y finalmente se le aplica una función de activación $g()$ no lineal. Generalmente, esta función de activación se aplica mediante una capa de tipo ReLU.

ReLU es la abreviación de *Rectified Linear Units*. Esta capa incrementa las propiedades no lineales de la función de decisión y de la red completa sin

afectar a los campos receptivos de la capa de convolución [21]. La función en sí es la siguiente:

$$f(x) = max(0, x) \tag{3.2}$$

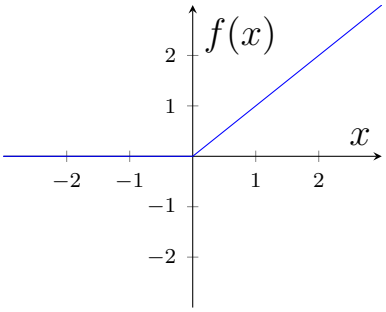


Figura 3.3: Función de activación ReLU

Pooling

La agrupación o *pooling* es una forma de muestreo descendente no lineal. La función más común para implementar el agrupamiento es *max pooling*. Esta función divide la imagen de entrada en un conjunto de rectángulos no superpuestos y, para cada una de estas subregiones, devuelve el máximo [21].

La capa de *pooling* sirve para reducir progresivamente el tamaño de la imagen, reduciendo así el número de parámetros y la cantidad de cálculos necesarios en la red. Es común insertar periódicamente una capa de *pooling* entre capas convolucionales sucesivas en una red neuronal convolucional.

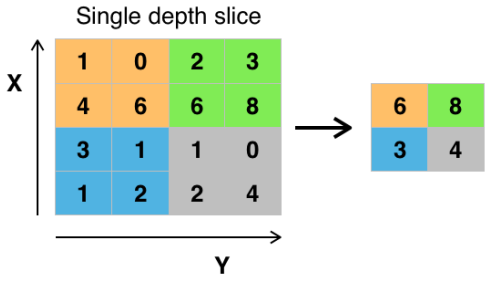


Figura 3.4: Max pooling con filtro 2x2 y paso de 2 muestras.

Esta capa opera de forma independiente en cada segmento de la dimensión de profundidad de la entrada y lo cambia de tamaño espacialmente. La forma más común, mostrada en la Figura 3.4, es una capa de *pooling* con filtros

de tamaño 2x2 con un paso de 2 muestras descendentes cada segmento de profundidad de la imagen de entrada por 2 a lo largo del ancho y alto de esta. En este caso, cada operación del máximo tiene más de 4 números. La dimensión de profundidad permanece sin cambios [21].

Capa completamente conectada

Finalmente, después de varias capas convolucionales y de *max pooling*, la clasificación de alto nivel en la red neuronal se realiza a través de capas totalmente conectadas. Las neuronas en una capa completamente conectada tienen conexiones con todas las activaciones en la capa anterior, como se ve en las redes neuronales comunes. Por lo tanto, sus activaciones pueden calcularse con una multiplicación de matrices seguida de un offset [21].

3.2. Redes Neuronales Residuales

La profundidad de una red es de crucial importancia para que el sistema pueda modelar un comportamiento complejo. El problema es que estas redes profundas son más difíciles de entrenar que una red estándar. La solución propuesta en [22], y resumida en esta sección, es el uso de redes neuronales residuales. Las redes residuales son mucho más profundas que sus redes convolucionales equivalentes, sin embargo, requieren definir una cantidad similar de parámetros (pesos) durante el entrenamiento.

Al incrementar la profundidad de una red, la precisión de esta se satura (alcanza un máximo) y luego se degrada rápidamente. Esta degradación no se debe a un ajuste de parámetros excesivo, si no al hecho de agregar más capas a un modelo con suficiente profundidad, lo que conduce a un mayor error de entrenamiento.

Para solucionar este problema, se introducen las redes neuronales residuales. Partiendo de una red neuronal convolucional, se define $H(x)$ como un mapeado subyacente para ser ajustado por unas pocas capas apiladas (no necesariamente toda la red), donde x representa las entradas a la primera de estas capas. Si se supone que múltiples capas no lineales pueden aproximarse de forma asintótica a funciones complicadas, entonces es equivalente a suponer que pueden aproximarse de forma asintótica a las funciones residuales, es decir, a $H(x) - x$ (suponiendo que la entrada y la salida tienen las mismas dimensiones). Entonces, en lugar de esperar que las capas apiladas se aproximen a $H(x)$, explícitamente se deja que estas capas se aproximen a una función residual $F(x) := H(x) - x$. La función original se convierte así en F (x) + x.

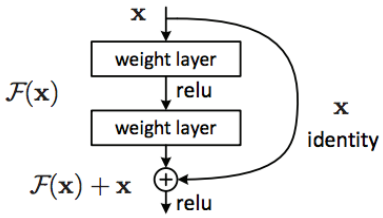


Figura 3.5: Bloque residual.

El aprendizaje residual se aplica cada pequeño grupo de capas apiladas. En la figura 3.5 se muestra un bloque residual, que se define como:

$$y = F(x, W_i) + x \tag{3.3}$$

Donde x e y son los vectores de entrada y salida de las capas consideradas. La función $F(x, W_i)$ representa el mapeado residual que ha de ser aprendido. Para el ejemplo de la figura 3.5 en el que hay dos capas, $F = W_2\sigma(W_1x)$. Donde σ denota la capa de activación de tipo ReLU y los offsets son omitidos para simplificar la notación.

En resumen, la base de una red residual consiste en añadir, cada cierto número de capas, la salida de una capa anterior a la salida de última de ellas. La cual a su vez se añadirá a la salida del siguiente grupo de capas. En la figura 3.6 se puede observar un ejemplo del comienzo de una red residual de 34 capas.

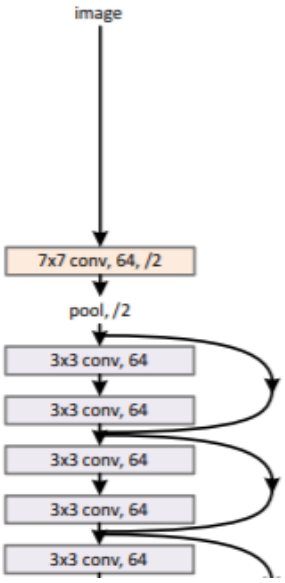


Figura 3.6: Red residual.

3.3. ResNet-50

La red ResNet-50, es una red neuronal residual que cuenta con 50 capas. Esta red gano el ILSVRC (Imagenet Large Sacle Visual Recognition Challenge) de 2015.

Al aumentar el número de capas, también aumenta el tiempo que es necesario emplear en el entrenamiento de la red. Para disminuirlo, se ha modificado el bloque residual fundamental de la figura 3.5 de forma que tenga un diseño de "cuello de botella". Para cada función residual F, se utiliza un grupo de tres capas en vez de dos. Las tres capas son convoluciones de 1x1, 3x3 y 1x1, donde las capas de 1x1 se encargan de reducir y después restaurar las dimensiones de la imagen de entrada. De esta manera, la convolución de 3x3 opera con una entrada más pequeña, lo cual resulta en un entrenamiento y una ejecución más rápidos. En la figura 3.7 se puede observar como el bloque de tres capas recibe una entrada de dimensión 256, la cual es reducida por la primera capa a 64 para aplicarle la convolución de 3x3, y finalmente la tercera capa aumenta la dimensión para que vuelva a ser 256 [22].

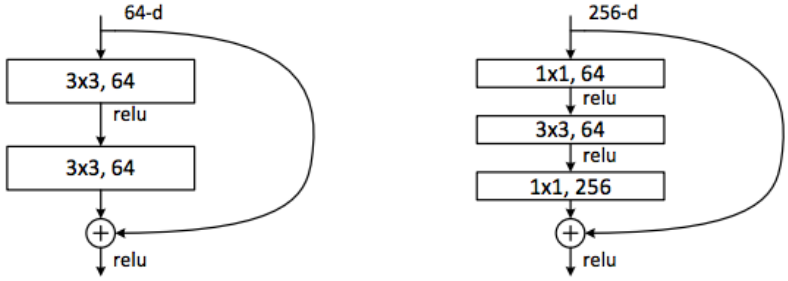


Figura 3.7: Comparación entre el bloque residual básico y el bloque residual cuello de botella.

Para formar la ResNet-50, se han sustituido los bloques residuales básicos de la red residual de 34 capas, mostrada en la figura 3.6 por los bloques residuales de tres capas, resultando así una red residual de 50 capas. La estructura completa de la red se muestra en [23].

Capa de normalización

La ResNet-50 introduce una capa de normalización [24] después de cada capa de convolución. La razón de normalizar la salida de cada convolución, se debe a que la normalización permite a cada capa de la red a aprender independientemente de otras capas.

Para incrementar la estabilidad de una red, se normaliza la salida de la función de activación de la capa anterior restando la media (μ) y dividiendo por la varianza (σ). Además, se le aplica un escalado (γ) y se le suma un offset (β).

$$\frac{\gamma(\mu - x)}{\sigma} + \beta \tag{3.4}$$

Capítulo 4

Algoritmo de estimación de la profundidad

En este capítulo se presenta el algoritmo que se ha implementado en la plataforma hardware. Después, se detalla el proceso que se ha seguido para realizar la implementación y se analiza la plataforma hardware que se ha utilizado.

4.1. Modificación de la ResNet-50

La implementación realizada se basa en el artículo *Deeper Depth Prediction with Fully Convolutional Residual Networks* [25]. En él, se propone una ResNet-50 modificada con el fin de estimar el mapa de profundidad de una escena utilizando una sola imagen RGB.

Casi todas las redes neuronales convolucionales actuales contienen una parte que reduce progresivamente la resolución de la imagen de entrada a través de una serie de convoluciones y operaciones de *pooling*. En los problemas de regresión en los que la salida deseada es una imagen de alta resolución, se requiere alguna forma de aumento de resolucion al final de la red para obtener un mapa de salida más grande.

La arquitectura propuesta, mostrada en la Figura 4.1, se basa en la ResNet-50, en la cual se ha sustituido la capa completamente conectada por cuatro bloques de proyecciones "hacia arriba", seguidos de una última capa de convolución. La modificación realizada se muestra encuadrada en rojo en la Figura 4.1. Si en vez de estos bloques se hubiese utilizado una capa completamente conectada del mismo tamaño, esta habría introducido 3.3 billones de parámetros, que ocuparían alrededor de 12.6 GB en memoria, haciendo imposible implementarlo en el hardware actual. En cambio, estos bloques

contienen una menor cantidad de pesos a la vez que mejoran la exactitud de los mapas de profundidad predichos. Después de los bloques de proyección, se aplica una última capa de convolución, que devuelve la predicción final.

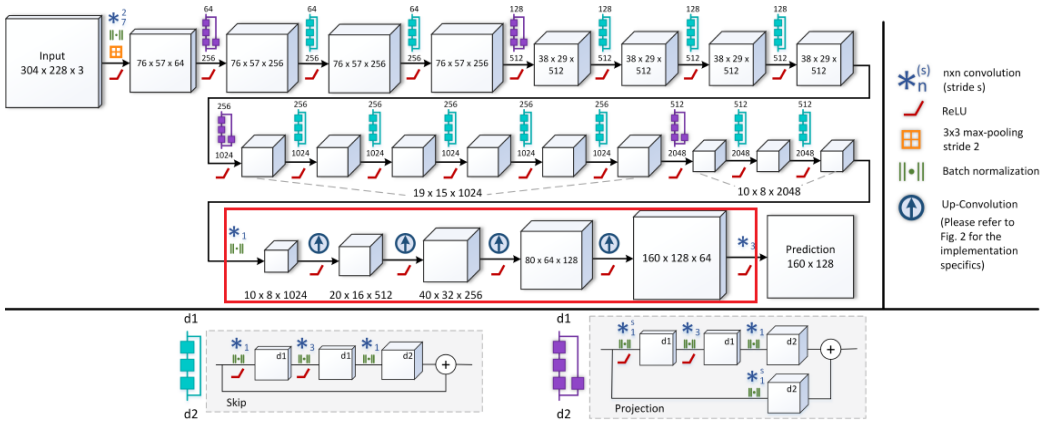


Figura 4.1: Arquitectura de la ResNet-50 modificada.

Bloques de proyección

Las capas de *unpooling* realizan la operación inversa al *pooling*, incrementando la resolución espacial de los mapas de características. En la red propuesta, se han implementado estas capas de *unpooling* con el fin de aumentar el tamaño de la salida. A cada una de estas capas le sigue otra de convolución con un filtro de 5 x 5 y sucesivamente de una función de activación ReLU. Este bloque compuesto por las capas de *unpooling*, convolución y ReLU, se define como bloque de convolución "hacia arriba" y se muestra en la Figura 4.3a. Cada uno de estos bloques aumenta en dos el tamaño de la entrada. En la arquitectura propuesta, se han conectado 4 de estos bloques, haciendo que la salida sea 16 veces mayor, y consiguiendo el mejor *trade-off* entre el consumo de memoria y la resolución de la imagen.

Para hacer el bloque de convolución más rápido, se ha reformulado la operación de convolución "hacia arriba". Esta modificación se puede ver en la Figura 4.2. En la parte superior, al mapa de características se le aplica *unpooling* seguido de una convolución con un filtro de 5 x 5. Observando el mapa después de haberle aplicado el *unpooling*, se ve que, dependiendo de la localización del filtro, solo ciertos pesos son multiplicados por valores potencialmente distintos de cero. Estos pesos se clasifican en cuatro grupos no superpuestos, indicados por colores diferentes y por las letras A, B, C, D en la figura.

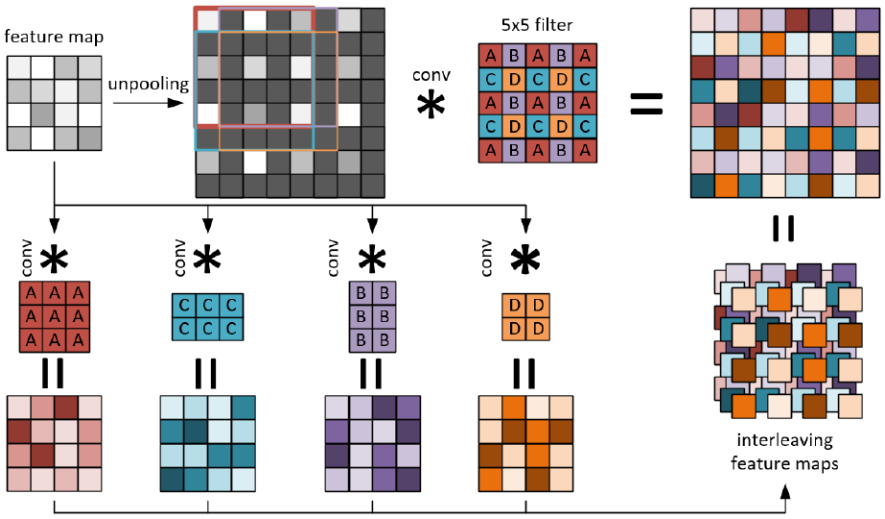


Figura 4.2: Convoluciones "hacia arriba" más rápidas.

Basandose en los grupos de filtros, el filtro de 5 x 5 ha sido convertido en cuatro nuevos filtros de tamaños 3 x 3 (A), 3 x 2 (B), 2 x 3 (C) y 2 x 2 (D). Exactamente la misma salida que utilizando las operaciones originales (*unpooling* y convolución), puede ser obtenida entrelazando los cuatro mapas de características obtenidos al aplicar cada filtro como se muestra en la Figura 4.2.

Posteriormente, los bloques de convolución "hacia arriba" han sido ampliados para crear un bloque residual que aumente la resolución. Para ello, se ha introducido una convolución de 3 x 3 después del bloque original y se ha añadido una conexión que proyecta la salida de la capa de *unpooling* hacia el resultado, tal y como se muestra en la Figura 4.3c. Debido a los diferentes tamaños, el mapa más pequeño necesita ser aumentado de tamaño usando otra convolución "hacia arriba" en el hilo de la proyección. Ya que el *unpooling* tiene que ser aplicado en ambos hilos, se aplica simplemente la convolución de 5 x 5 por separado en ambos hilos. A este nuevo bloque se le ha llamado bloque de proyección "hacia arriba".

El encadenamiento de bloques de proyección permite que la información de alto nivel se transmita de forma más eficiente en la red al tiempo que aumenta progresivamente el tamaño de los mapas de características. Esto permite la construcción de la propuesta red coherente y completamente convolucional para la predicción de la profundidad. La Figura 4.3 muestra las diferencias entre un bloque de convolución y un bloque de proyección, así como sus versiones más rápidas. Finalmente, el bloque de proyección más rápido mostrado en la Figura 4.3d ha sido el implementado en la arquitectura propuesta.

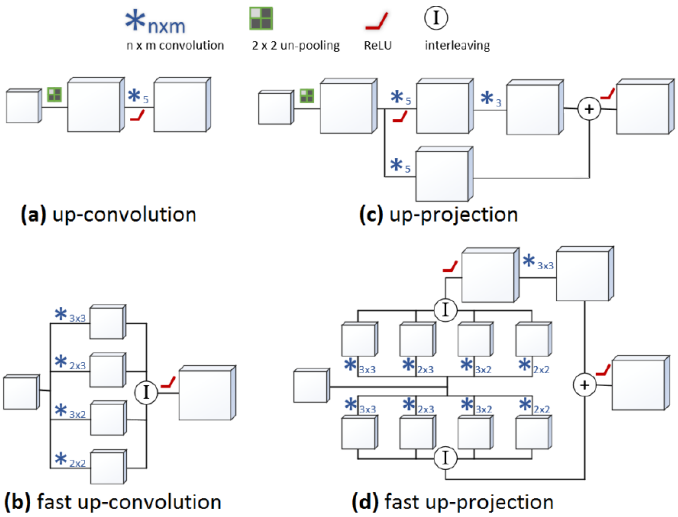
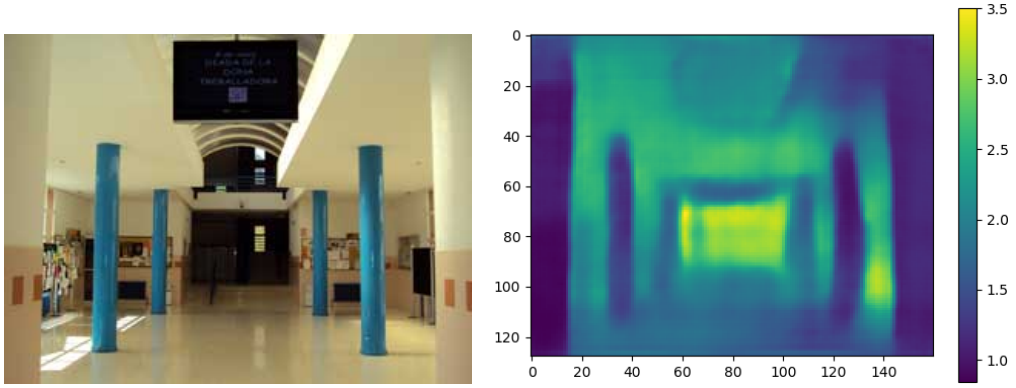


Figura 4.3: De convoluciones a proyecciones "hacia arriba". (a) Convolución "hacia arriba" estándar. (b) Equivalente más rápido de (b). (c) Bloque de proyección. (d) Versión equivalente más rápida de (c).

Entrenamiento y predicción

La ResNet-50 modificada ha sido entrenada por los autores de [25] con el dataset *NYU Depth v2* [26], el cual es uno de los datasets RGB más grandes actualmente para la reconstrucción de escenas interiores. El modelo y los pesos finales correspondientes a cada capa se encuentran disponibles para los entornos de TensorFlow [27] y Matlab [28]. En la Figura 4.4 se muestra la predicción de una escena utilizando el modelo en TensorFlow.



(a) Imagen RGB (b) Mapa de profundidad

Figura 4.4: Predicción utilizando TensorFlow

4.2. Características del algoritmo

El algoritmo descrito utiliza como entrada una imagen RGB de 304 x 228 píxeles y devuelve como salida una imagen de 160 x 128 píxeles y un solo canal, dónde cada píxel contiene un valor de profundidad. Cuanto mayor sea este valor, mayor sera la profundidad en esa región de la imagen.

4.3. Implementación en Caffe

Dado que uno de los objetivos de este trabajo es implementar en tiempo real la arquitectura descrita en la sección 4.1, es necesario un archivo ejecutable que utilice la GPU de la plataforma embebida.

Ya que la plataforma embebida en la que se va a realizar la implementación no soporta la librería de TensorFlow, se ha reescrito el modelo utilizando Caffe, librería para C++ y a partir de la cual se puede generar un archivo ejecutable.

Como se explicó en la sección 2.3.2, las redes neuronales en Caffe se definen en archivos de texto plano. Para definir la ResNet-50 modificada en Caffe, primero se ha descargado el archivo *.prototxt* que define la ResNet-50 original [30]. En este archivo, se han modificado las opciones de la capa de entrada de modo que la red modifique las dimensiones de la imagen de entrada a 304 x 228 píxeles. Además, se han eliminado las tres últimas capas (*pooling*, capa completamente conectada y activación de tipo *softmax*) y se han sustituido por los 4 bloques de proyección descritos en la sección 4.1, seguidos de una capa de *dropout* y una capa final de convolución.

4.3.1. Modificaciones en el código fuente de Caffe

Una de las ventajas que tiene Caffe, es que los códigos C++ que describen la función de cada capa se pueden modificar. Además, también es posible añadir capas nuevas creadas por el usuario. Debido a que hay funciones de TensorFlow que no están en Caffe, ha sido necesario realizar modificaciones en el código de alguna de las capas, y añadir dos capas nuevas para poder implementar los bloques de proyección.

Modificación de la capa de convolución

Antes de cada convolución, se le aplica un *padding* a la imagen, que dependerá del tamaño del filtro. Esta operación rellena el mapa de características con ceros, añadiendo filas o columnas de ceros arriba y abajo, a derecha e izquierda, o solo por una de las cuatro partes. En TensorFlow, el *padding* es

una operación primitiva, independiente de cualquier capa. Sin embargo, en Caffe, se encuentra integrado en la capa de convolución y se puede especificar la cantidad de filas y/o columnas de ceros que se quieren añadir, con la restricción de que se añade la misma cantidad de filas por arriba que por abajo, y la misma cantidad de columnas por la izquierda que por la derecha. El problema se encuentra en que, a la hora de aplicar los filtros A, B, C y D descritos en la sección 4.1, el *padding* que hay que realizar antes es irregular, es decir, no se añade la misma cantidad de ceros por arriba que por abajo, ni por la izquierda que por la derecha de la imagen.Para poder hacer esto en Caffe, se ha modificado la capa de convolución, de forma que se pueda especificar el *padding* en cada uno de los lados de la imagen.

En primer lugar, se ha editado el archivo *caffe.proto* que se encuentra en el directorio *caffe/src/caffe/proto*. En este archivo, se han modificado los parámetros de la capa de convolución (*ConvolutionParameter*) y se ha sustituido el parámetro *pad_h* por *pad_ht* y *pad_hb* que indican el número de filas de ceros a añadir encima y debajo de la imagen respectivamente. El parámetro *pad_w* se ha sustituido por *pad_wl* y *pad_wr*, que indican el número de columnas de ceros a añadir a izquierda y derecha de la imagen respectivamente.

A continuación se ha modificado el archivo *base_conv_layer.cpp* del directorio *caffe/src/caffe/layers*, el cual describe la capa base de las capas de convolución y convolución inversa. En la función *LayerSetUp*, en la sección *Setup pad dimensions*, se han realizado los siguientes cambios. En la condición del módulo, se ha comprobado que se haya especificado cualquiera de los parámetros *pad_ht*, *pad_hb*, *pad_wl* o *pad_wr*, en vez de comprobar *pad_h* y *pad_w* como se hacía originalmente. Dentro de las acciones de la condición, se ha modificado la asignación de valores al array *pad_data* como se muestra a continuación.

```
pad_data[0] = conv_param.pad_ht();
pad_data[1] = conv_param.pad_hb();
pad_data[2] = conv_param.pad_wl();
pad_data[3] = conv_param.pad_wr();
```

En el bucle *for*, se ha modificado la condición de fin de bucle, cambiando la variable *num_spatial_axes_*, que debe valer siempre 2, por el número 4, ya que ahora el array *pad_data* contiene cuatro elementos en vez de dos. Esta parte *else* contiene las operaciones a realizar en caso de que se haya utilizado una sola variable llamada *pad* en los parámetros de la capa de convolución para realizar el mismo *padding* por los cuatros lados de la imagen. Por lo tanto, hay que rellenar los cuatro elementos del array *pad_data* con el mismo valor indicado en el parámetro *pad*.

En el mismo directorio, se ha modificado también el archivo, *conv_layer.cpp*, que describe la capa de convolución. En la función *compute_output_shape*, dentro del bucle *for*, el cuál tiene dos iteraciones, una por la altura de la imagen y otra por la anchura, se ha sustituido la siguiente línea:

```
const int output_dim = (input_dim + 2 * pad_data[i] -
    kernel_extent) / stride_data[i] + 1;
```

Por el siguiente código:

```
int output_dim
if (i == 0) {
    output_dim = (input_dim + pad_data[0] + pad_data[1] -
        kernel_extent) / stride_data[i] + 1;
}
else {
    output_dim = (input_dim + pad_data[2] + pad_data[3] -
        kernel_extent) / stride_data[i] + 1;
}
```

De esta forma, en la primera iteración del bucle se sumará el *padding* correspondiente a las filas y en la segunda iteración al de las columnas. Y por lo tanto se tendrá en cuenta si se esta añadiendo o no la misma cantidad de filas o columnas a cada lado de la imagen.

Finalmente, se ha entrado en el directorio *caffe/src/caffe/util* y se ha editado el archivo *im2col.cpp*, el cual describe funciones utilizadas por la capa base de convolución. En este archivo, se ha sustituido *pad_h* por *pad_ht* y *pad_hb* y *pad_w* por *pad_wt* y *pad_wb* en todas las declaraciones de funciones en las que aparecen *pad_h* y *pad_w*. Cada vez que aparece $2 * pad\_h$, se ha sustituido por $pad\_ht + pad\_hb$, y cada vez que aparece $2 * pad\_w$, se ha sustituido por $pad\_wt + pad\_wb$. Estos mismos cambios se han realizado también en la versión CUDA del archivo, *im2col.cuda*, además de modificar las declaraciones de funciones en el archivo *im2col.hpp* que se encuentra en el directorio *caffe/include/caffe/util* para que coincidan con las modificadas en el archivo *.cpp*.

Capa de redimensionado

Una vez realizadas las cuatro convoluciones A, B, C y D, hay que entrelazar sus resultados. En el modelo descrito en TensorFlow, esta operación se realiza con la siguiente función.

```
def interleave(tensors, axis):
    old_shape = get_incoming_shape(tensors[0])[1:]
    new_shape = [-1] + old_shape
    new_shape[axis] *= len(tensors)
```

```
    return tf.reshape(tf.concat(tensors, axis + 1), new_shape)
```

La función recibe un array (*tensors*) con dos tensores de las mismas dimensiones y un índice (*axis*) que indica a lo largo de qué dimensión se realiza el entrelazado. Su propósito es concatenar los tensores que se le han pasado como argumento a lo largo del índice indicado más uno. El resultado de la concatenación se redimensiona de forma que tenga las dimensiones de los tensores de entrada con la dimensión indicada en *axis* multiplicada por dos.

Esta función es utilizada para entrelazar primero las convoluciones A y B (left) y después las convoluciones C y D (right). A continuación, se entrelazan los dos mapas de características resultantes (Y). El código siguiente se ejecuta justo después de realizar la convolución D.

```
# Interleaving elements of the four feature maps
left = interleave([outputA, outputB], axis=1)  # columns
right = interleave([outputC, outputD], axis=1)  # columns
Y = interleave([left, right], axis=2) # rows
```

Para implementar la función *interleave* en Caffe se han utilizado dos capas. La capa *Concat*, que al igual que la función de TensorFlow concatena dos tensores a lo largo de la dimensión indicada. Y una capa llamada *MyReshape* que ha sido creada para cambiar las dimensiones de la misma forma que en la función *interleave*. La capa recibe dos *Blobs* como entrada, y cuenta con un solo parámetro llamado *axis*, que se utiliza igual que en la función *interleave* descrita anteriormente. El código C++ que describe a esta capa y su cabecera correspondiente se encuentran en el apéndice A.

Para poder utilizar la capa creada en la descripción de la red se ha incluido el archivo *myreshape_layer.cpp* en el directorio *caffe/src/caffe/layers* y el archivo *myreshape_layer.hpp* en el directorio *caffe/include/caffe/layers*. Además, se ha editado el archivo *caffe.proto* que se encuentra en el directorio *caffe/src/caffe/proto*. Se ha añadido la siguiente línea dentro de *LayerParameter*.

```
message LayerParameter {
    //...
    optional MyReshapeParameter myreshape_param = 148;
    //...
}
```

Finalmente, se ha incluido el siguiente código, que crea los parámetros de la capa. En este caso solo hay uno, *axis*.

```
message MyReshapeParameter {
  optional int32 axis = 1 [default = 0];
}
```

Capa para permutar la imagen

En la tercera operación de entrelazado (Y), se concatena a lo largo de la dimensión de los canales, lo que hace que aumente en dos el tamaño de dicha dimensión. A continuación, se redimensionan los datos de forma que la dimensión de los canales se reduzca a la mitad, y la de la altura (filas) aumente en dos.

A la hora de aplicar esta redimensión, el resultado obtenido en Caffe no coincide con el de TensorFlow. Esto se debe a que los datos almacenados en los tensores de TensorFlow tienen la estructura NHWC y los *Blobs* de Caffe utilizan NCHW. Por lo tanto, los datos se distribuyen de distinta manera al hacer la redimensión.

Para corregir este problema, se ha utilizado una capa que cambia el orden de las dimensiones. Esta capa de permutación se ha añadido después de la de concatenación, para cambiar el orden de las dimensiones de NCHW a NHWC. A continuación, se ha hecho el redimensionado y finalmente se ha vuelto a utilizar la capa de permutación para volver a poner las dimensiones en formato NCHW.

Esta capa no esta incluida en el código fuente original de Caffe. Sin embargo, se encuentra disponible en *GitHub* [31] para su uso. Para incluirla en Caffe y poder utilizarla, se han descargado los códigos *permute_layer.cpp* y *permute_layer.hpp* y se han guardado en los directorios *caffe/src/caffe/layers* y *caffe/include/caffe/layers* respectivamente. Además, se ha modificado el archivo *caffe.proto* que se encuentra en el directorio *caffe/src/caffe/proto*. Se ha añadido la siguiente línea dentro de *LayerParameter*.

```
message LayerParameter {
  //..
  optional PermuteParameter permute_param = 149;
  //..
}
```

El código que se muestra a continuación, ha sido añadido para crear los parámetros de la capa. Esta capa tiene un solo parámetro llamado *order*, el cual se puede especificar una vez por cada dimensión.

```
message PermuteParameter {
  repeated uint32 order = 1;
}
```

Es decir, si solo se quiere cambiar de orden la primera dimensión, solo se especificará *order* una vez, indicando la nueva posición de esa dimensión (de 0 a 3). En el siguiente ejemplo se muestra como especificar el parámetro *order* para cambiar de NCHW a NHWC.

```
layer {
    bottom: "layer2x_br1_concat_Y"
    top: "layer2x_br1_permute_Y"
    name: "layer2x_br1_permute_Y"
    type: "Permute"
    permute_param {
        order: 0
        order: 2
        order: 3
    }
}
```

4.3.2. Bloque de proyección en Caffe

Las tres últimas capas de la ResNet-50 original (*pooling*, capa completamente conectada y activación de tipo *softmax*), han sido sustituidas por una convolución con su posterior normalización correspondiente, seguida por los cuatro bloques de proyección. Cada bloque se ha descrito en Caffe utilizando la siguiente disposición de capas:

1. **Capas de convolución con los filtros A, B, C y D.** Estas capas reciben como entrada la salida del bloque de proyección anterior, o en el caso del primer bloque, la salida de la capa de convolución mencionada en el párrafo anterior.

2. **Capa de concatenación.** Para concatenar los resultados de las convoluciones A y B.

3. **Capa *MyReshape*.** Utilizada para redimensionar el resultado de la concatenación anterior.

4. **Capa de concatenación.** Para concatenar los resultados de las convoluciones C y D.

5. **Capa *MyReshape*.** Para redimensionar el resultado de la concatenación anterior.

6. **Capa de concatenación.** Cuyas entradas son las salidas de las dos capas *MyReshape* anteriores.

7. **Capa de permutación.** Utilizada para cambiar el orden de las dimensiones de la imagen de NCHW a NHWC.

8. **Capa de redimensionado.** La entrada de esta capa es la salida de la capa de permutación, a la cuál se le aplica un redimensionado de forma que los canales se reduzcan a la mitad y las filas aumenten en dos.

9. **Capa de permutación.** Para cambiar el orden de las dimensiones de la imagen de NHWC a NCHW de nuevo.

10. **Capas de normalización y escalado.** Cuya entrada es la salida de la permutación anterior.

11. **Capa de activación ReLU.**

12. **Capa de convolución.** Con un filtro de 3x3, que toma como entrada la salida de la capa de activación.

13. **Capas de normalización y escalado.** Cuya entrada es la salida de la convolución anterior.

A continuación, se vuelven a repetir las capas de los pasos 1 a 10 manteniendo la misma entrada para las convoluciones con los filtros A, B, C y D. La salida de la última capa se suma con la salida de la capa de escalado del paso 13. A este resultado se le aplica una activación de tipo ReLU, cuya respuesta representa la salida del bloque de proyección, y constituye la entrada del siguiente bloque. A la salida del cuarto bloque se le aplica un *dropout* seguido de una convolución y una capa de activación, siendo la salida de esta última el resultado final, es decir, el mapa de profundidad predicho.

4.4. Fichero de pesos para Caffe

Para implementar el modelo descrito en Caffe, hace falta el archivo con los pesos correspondientes a cada capa. Por lo tanto, se han exportado los pesos pertenecientes al modelo desarrollado en TensorFlow a un archivo de tipo *.caffemodel* que pueda ser utilizado por Caffe. Hay que tener en cuenta que no todas las capas tienen pesos. En el modelo que se ha implementado, solamente las capas de convolución y normalización llevan pesos asociados.

Para pasar los pesos de TensorFlow a Caffe, se ha utilizado Python, las librerías de TensorFlow y Caffe, y la librería *NumPy* para Python. Esta última librería contiene, entre otros aspectos, la posibilidad de trabajar con arrays de N dimensiones y guardarlos en ficheros de tipo .npy.

Diferencias entre TensorFlow y Caffe a tener en cuenta

Antes de realizar la conversión, hay que saber que en TensorFlow, la capa *BatchNormalization* tiene cuatro variables entrenables asociadas: media, varianza, escalado y offset. En Caffe, esta capa se divide en dos capas: *BatchNorm* y *Scale*. La primera tiene asociadas las variables media y varianza, así como una tercera variable llamada promedio móvil, que no se encuentra en TensorFlow y que habrá que fijar a 1 para todas las capas de normalización. La capa *Scale* tiene asociadas las variables de escalado y offset.

Además, los pesos de los filtros de convolución, están en diferente orden en TensorFlow que en Caffe. En TensorFlow el orden es: alto, ancho, profundidad, número de filtros. Mientras que en Caffe es: número de filtros, profundidad, alto, ancho.

4.4.1. Generación del fichero NumPy

En el PC, y utilizando TensorFlow, se ha escrito y ejecutado un script de Python que genera un fichero NumPy. Este contiene un array con los pesos de las capas de convolución y normalización de la red.

El array principal se compone de un array para cada capa de convolución que a su vez contiene dos arrays, uno para los pesos y otro para los offsets. Y de otro array para cada capa de normalización, que a su vez contiene cuatro arrays para la media, varianza, escalado y offset.

Como se muestra a continuación, se ha inicializado la red y se han cargado los pesos del archivo *checkpoint* (.ckpt).

```
import tensorflow as tf
import numpy as np
import models

# Load network architecture and parameters
# Create a placeholder for the input image
height = 228
width = 304
channels = 3
batch_size = 1
input_node = tf.placeholder(tf.float32, shape=(None, height,
    width, channels))
net = models.ResNet50UpProj({'data': input_node}, batch_size,
    1, False)

# Use to load from ckpt file
sess = tf.Session()
saver = tf.train.Saver()
saver.restore(sess, 'NYU_FCRN.ckpt')
```

Una vez hecho esto, se ha generado el vector que contiene todas las variables entrenables. Para ello, se han recorrido todas las capas de la red que tienen pesos asociados y se han ido escribiendo en el vector los pesos obtenidos de cada una de estas capas. A continuación se muestra un ejemplo en el que se cargan los pesos de las capas *BatchNormalization*.

```
with tf.variable_scope(layer, reuse = True):
    mean = sess.run(tf.get_variable("mean"))
with tf.variable_scope(layer, reuse = True):
    variance = sess.run(tf.get_variable("variance"))
with tf.variable_scope(layer, reuse = True):
    scale = sess.run(tf.get_variable("scale"))
with tf.variable_scope(layer, reuse = True):
    offset = sess.run(tf.get_variable("offset"))
#combine layer parameters in an array
layer_pair = [mean, variance, scale, offset]
#append array to data file
data_file.append(layer_pair)
```

Finalmente, se ha escrito el vector *data file* en un archivo de tipo .npy con la instrucción *save* del módulo NumPy.

4.4.2. Generación del fichero caffemodel

Una vez generado el fichero NumPy con todos los pesos, este se ha pasado a la plataforma embebida, donde se ha utilizado para generar un archivo de pesos que pueda ser leído por el entorno de Caffe.

Para ello, se ha escrito otro script de Python, en el que se vuelve a utilizar el módulo NumPy y se sustituye la librería de TensorFlow por la de Caffe. A continuación se muestra el principio del código que realiza la conversión.

```
import caffe
import numpy as np

#load the data file
data_file = np.load('ResNet50UpProj.npy')
#define architecture
net = caffe.Net('ResNet-50UpProj-deploy.prototxt', caffe.TEST)

#load parameters
net.params['conv1'][0].data[...] =
    data_file[0][0].transpose((3,2,0,1))
net.params['conv1'][1].data[...] = data_file[0][1]

net.params['bn_conv1'][0].data[...] = data_file[1][0]
net.params['bn_conv1'][1].data[...] = data_file[1][1]
net.params['bn_conv1'][2].data[...] = 1
```

```
net.params['scale_conv1'][0].data[...] = data_file[1][2]
net.params['scale_conv1'][1].data[...] = data_file[1][3]
```

En primer lugar, se carga el array que hay en el fichero NumPy creado anteriormente en una variable llamada *data file*. A continuación, se define la red a la vez que se la hace un test para comprobar que no ha habido ningún error a la hora de definirla.

Una vez hecho esto, se asocian los pesos del array con la variable correspondiente de cada capa. El código muestra un ejemplo para las tres capas que utilizan pesos en esta red (convolución, normalización, y escalado).

Para la primera capa de convolución, se asocia el elemento 0 de dicha capa con el elemento (0,0) del array, el cuál representa los pesos de los filtros de la convolución. Además, como se ha explicado anteriormente, hay que trasponerlo ya el formato en el que están los pesos es diferente en TensorFlow que en Caffe. El elemento 1 de esta capa, que son los offset, se carga con el elemento (0,1) del array.

El caso de la capa de normalización sería igual, con la única diferencia de que esta tiene una variable más que, como se ha dicho antes, hay que fijar a 1 en todas las capas de normalización.

Finalmente, se cargan las variables de la primera capa de escalado, que corresponden a los elementos (1,2) y (1,3) del array principal, ya que se guardaron en el mismo subarray que las variables de la capa de normalización.

El código continua asociando cada variable de cada capa con el elemento del array que contiene los pesos correspondientes a dicha variable. Una vez finalizada la asignación, se crea el fichero .caffemodel con la siguiente instrucción:

```
#save caffemodel
net.save('ResNet50UpProj.caffemodel')
```

4.5. Kit de desarrollo Jetson TX2

La plataforma en la que se ha implementado la red descrita es el módulo Jetson TX2, integrado en el kit de desarrollo Jetson TX2 de NVIDIA. Este kit de desarrollo cuenta con todas las funciones necesarias para computación visual. Además, esta preparado para aplicaciones que requieran alto rendimiento computacional en un entorno de baja potencia [29].

Hardware

El módulo Jetson TX2, integrado en el kit de desarrollo y mostrado en la figura 4.5, cuenta con:

- GPU: NVIDIA Pascal con 256 cores de CUDA
- CPU: Dos núcleos NVIDIA Denver 2 de 64 bits, acompañados de cuatro núcleos Cortex-A57
- Memoria LPDDR4 de 8GB
- Almacenamiento flash eMMC 5.1 de 32GB
- Conectividad: 802.11ac WLAN, Bluetooth
- 10/100/1000BASE-T Ethernet

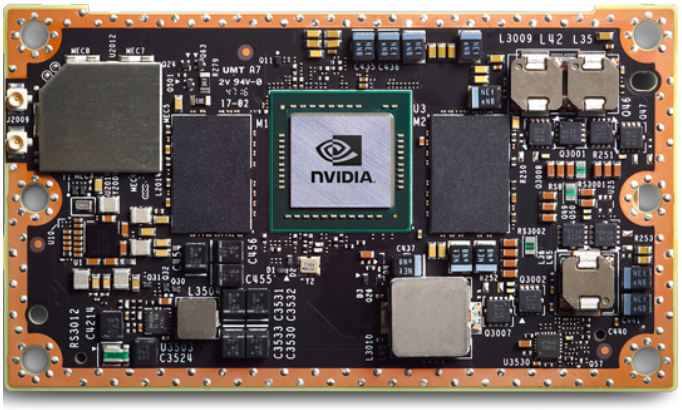


Figura 4.5: Modulo Jetson TX2.

Software

A través de un host, se ha instalado la última versión del JetPack. Este es un instalador que automatiza la instalación de un entorno de desarrollo para la plataforma embebida, lo que facilita arrancar el módulo con la última imagen del sistema operativo. El cual es un sistema simple de ficheros derivado de Ubuntu. El JetPack incluye, entre otras, las siguientes librerías:

- CUDA Toolkit.
- OpenCV.

- VisionWorks: Paquete de desarrollo de software para Computer Vision y procesado de imagen.
- cuDNN: Librería de CUDA para redes neuronales.
- TensorRT: Librería para inferenciar modelos de deep learning.

4.6. Ejecución de la red

Una vez definida la red y obtenido el fichero con los pesos para cada capa, se ha escrito un código en C++ que ejecute la red en tiempo real. Este código, descrito en el apéndice B.1, utiliza las librerías de Caffe y OpenCV además de las propias de C++.

El programa inicializa la red descrita en el fichero *ResNet-50UpProj-deploy.prototxt* con los pesos correspondientes, los cuales se encuentran almacenados en el fichero *ResNet50UpProj.caffemodel*.

A continuación, se abre la cámara utilizando las funciones descritas en el módulo *imag.cpp*, el cual se encuentra en la sección B.2 del apéndice. Una vez el programa ha abierto la cámara, entra en un bucle que termina cuando se pulse cualquier tecla. En cada iteración del bucle, se lee un *frame* del vídeo capturado por la cámara y se guarda en un objeto *Mat*. Este objeto es preprocesado para que sea de las mismas dimensiones que la entrada de la red y, después, se carga en el *Blob* correspondiente a la capa de entrada de la red para ejecutarla.

La salida de la red es un *Blob* cuyos datos son copiados en un vector de tipo *float* y representan el valor de profundidad para cada píxel de la imagen. Cuanto más grande sea un valor, mayor será la profundidad para dicho píxel. Estos datos se escalan de forma que el valor más grande valga 1 y el más pequeño valga 0, mientras se van almacenando en un objeto *Mat* de 160x128 con un solo canal (blanco y negro). Esto se debe a que la función *imshow* de OpenCV mapea los valores flotantes entre 0 y 1 a una escala de grises. De esta forma, cuando a continuación, se ejecuta la función *imshow* el mapa de profundidad se verá en blanco y negro. Seguidamente, el programa saltará a la siguiente iteración del bucle para procesar el próximo *frame* del vídeo.

En la figura 4.6b se puede observar el mapa de profundidad de la imagen mostrada en la figura 4.6a. Cuanto más oscura esté una zona de la imagen más cerca significa que está. Mientras que, cuanto más blanca esté una zona, más lejos significa que está.

(a) Imagen del video congelado.

(b) Mapa de profundidad en blanco y negro.

Figura 4.6: Resultado de la predicción en tiempo real

Capítulo 5

Evaluación

En este capítulo se ha evaluado la red neuronal propuesta en [25] e implementada en la placa embebida Jetson TX2. Para ello ha sido necesario generar un modelo Caffe específico. La evaluación de la implementación ha permitido obtener los tiempos de ejecución en dos plataformas diferentes (ordenador portátil y placa embebida), y calcular el suelo de error de la implementación en la Jetson TX2.

5.1. Tiempos

Los tiempos de ejecución han sido calculados en un ordenador portátil ASUS UX330UA con un procesador Intel i7-7500 CPU @ 2.70 GHz y en la placa Jetson TX2. La versión para TensorFlow del modelo descrita por [25], ha sido evaluada en el portátil, mientras que su versión para Caffe, realizada en este trabajo, ha sido evaluada en la Jetson TX2.

Para cada plataforma, el tiempo que tarda el modelo en devolver el mapa de profundidad de una imagen de 2,85MB y 4032 $\times$ 3024 píxeles ha sido medido 100 veces para obtener el valor medio.

Plataforma	Librería	Lenguaje	Tiempo de Ejecución
Jetson TX2	Caffe	C++	123.87 ms
ASUS UX330UA	TensorFlow	Python	714.16 ms

Tabla 5.1: Tiempos de ejecución en las distintas plataformas.

Como se puede observar en la Tabla 5.1, el modelo en la plataforma embebida utilizando Caffe procesa aproximadamente 10 frames por segundo y es 7 veces más rápido que en el ordenador portátil utilizando TensorFlow.

5.2. Suelo de error

Para calcular el suelo de error de la implementación, se ha utilizado la imagen de la Figura 5.1 y su correspondiente mapa de profundidad (Figura 5.2b), los cuales pertenecen al dataset disponible en [32].

Figura 5.1: Imagen RGB utilizada para calcular el suelo de error.

En la Figura 5.2 se muestra la predicción del mapa de profundidad de la imagen de la Figura 5.1 en comparación con el mapa de profundidad real proporcionado por el dataset de [32].

(a) Predicción (b) Mapa de profundidad esperado

Figura 5.2: Comparación del mapa de profundidad predicho con el esperado.

El suelo de error ha sido calculado restando las dos imágenes de la Figura 5.2 mediante el comando *absdiff* de OpenCV. Observando la Figura 5.3, se ve como las zonas más oscuras indican que hay un menor porcentaje de error, mientras que las zonas más claras indican un mayor porcentaje de error. Un buen ejemplo es la lámpara que se encuentra en el escritorio a la derecha de la imagen. La lámpara no se aprecia en la predicción (Figura 5.2a) debido a que no tiene perspectiva, y por lo tanto en la Figura 5.3 la zona alrededor de la lampara se muestra de un color próximo al blanco.

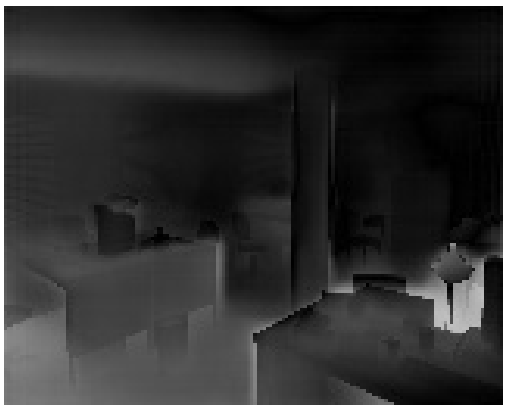


Figura 5.3: Suelo de error.

Finalmente, se ha calculado el error que introduce la implementación al calcular el mapa de profundidad utilizando la fórmula del error cuadrático medio.

$$x_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{5.1}$$

Donde x_i es cada píxel de la imagen de la Figura 5.3 y N el número de píxeles de la imagen. Ya que la fórmula se aplica a una imagen en blanco y negro, el máximo valor de error posible que podría obtenerse sería 255 (blanco), que significa un 100 % de error, mientras que el mínimo sería 0 (negro), que significa un 0 % de error. El valor obtenido a partir de la Figura 5.3 es 44,69. Este valor se divide por 255 y multiplica por 100 para obtener el porcentaje de error, el cual es 17,53 %.

Capítulo 6

Conclusiones

Este trabajo presenta un algoritmo basado en redes neuronales profundas que, a partir de la imagen capturada por una sola cámara, obtiene su mapa de profundidad. Dicho algoritmo se ha implementado en una plataforma embebida.

Para ello, se ha realizado un estudio de las diferentes técnicas disponibles para extraer la profundidad de una escena, llegando a la conclusión de que el uso de técnicas de *Machine Learning* es una de las mejores formas de obtener el mapa de profundidad utilizando nada más que una cámara.

Dentro del campo de *Machine Learning*, se han estudiado los algoritmos basados en redes neuronales, y dentro de estos, las redes neuronales convolucionales, que son las más utilizadas para el reconocimiento y procesamiento de imágenes. Para conseguir redes neuronales convolucionales con cierta profundidad sin que aumente significativamente el número de pesos durante el entrenamiento, se utilizan las llamadas redes neuronales residuales. En este trabajo concretamente, se ha estudiado la ResNet-50, una red neuronal residual con 50 capas.

Teniendo en cuenta el artículo [25] se ha modificado la ResNet-50, sustituyendo su última capa por cuatro bloques de proyección, cuya función es acelerar la red, y disminuir el número de parámetros (pesos) durante el entrenamiento. El algoritmo propuesto en [25] obtiene la profundidad de una escena con prometedores resultados.

Este algoritmo se encontraba ya descrito en Python mediante la librería TensorFlow, y entrenado con el conjunto de datos *NYU Depth v2* [26]. Los ficheros que describen al modelo y los valores finales de los pesos asociados a cada capa se pueden descargar de *GitHub* [27]. Ya que la placa embebida en la que se ha realizado la implementación no soporta la librería de TensorFlow, ha sido necesario reescribir el modelo de la ResNet-50 modificada utilizando Caffe, librería para C++.

Para conseguir en Caffe la misma red que la descrita en [25], se ha partido del fichero Caffe que describe la ResNet-50 original [30]. En este fichero, se ha eliminado la última capa y se han añadido los cuatro bloques de proyección presentados en [25]. Para implementar estos bloques en Caffe, ha sido necesario crear y añadir nuevas capas a la librería de Caffe, así como modificar algunas ya existentes. Además, el fichero con los pesos para el modelo de TensorFlow ha sido convertido al formato de fichero de pesos utilizado por Caffe.

La plataforma en la que se ha implementado esta red neuronal es la placa Jetson TX2 de NVIDIA. Con ella, se obtienen unos tiempos de ejecución de 123,87ms de media, es decir, el algoritmo procesa aproximadamente unas 10 imágenes por segundo. El mapa de profundidad que se obtiene con esta implementación presenta un suelo de error del 17,53 %.

Este trabajo ha precisado programar utilizando lenguajes como Python y C++ y librerías como TensorFlow, Caffe y OpenCV.

Apéndice A

Capa MyReshape

A.1. Código C++ que describe la capa

```
#include <vector>
#include "caffe/layers/myreshape_layer.hpp"

namespace caffe {

template <typename Dtype>
void MyReshapeLayer<Dtype>::Reshape(const vector<Blob<Dtype>*>&
    bottom, const vector<Blob<Dtype>*>& top) {

    int inferred_axis_ = 0;
    int constant_count_ = 1;
    int input_axis =
        this->layer_param_.myreshape_param().axis();
    int inferred_dim;

    vector<int> top_shape(bottom[0]->num_axes());

    for (int i = 0; i < bottom[0]->num_axes(); i++) {
        if(i==inferred_axis_) {
            top_shape[i] = -1;
        }
        else if (i==input_axis) {
            top_shape[i] = 2*bottom[0]->shape(i);
            constant_count_ *= top_shape[i];
        }
        else {
            top_shape[i] = bottom[0]->shape(i);
            constant_count_ *= top_shape[i];
        }
    }
```

```
    CHECK_EQ(0, bottom[1]->count() % constant_count_) <<
        "bottom count (" << bottom[1]->count() << ") must be
        divisible by the product of the specified dimensions ("
        << constant_count_ << ")";
    inferred_dim = bottom[1]->count() / constant_count_;

    top_shape[inferred_axis_] = inferred_dim;

    top[0]->Reshape(top_shape);
    CHECK_EQ(top[0]->count(), bottom[1]->count()) << "output
        count must match input count";
    top[0]->ShareData(*bottom[1]);
    bottom[1]->ShareDiff(*top[0]);
}

INSTANTIATE_CLASS(MyReshapeLayer);
REGISTER_LAYER_CLASS(MyReshape);

}  // namespace caffe
```

A.2. Cabecera de la capa

```
#ifndef CAFFE_MYRESHAPE_LAYER_HPP_
#define CAFFE_MYRESHAPE_LAYER_HPP_

#include <vector>

#include "caffe/blob.hpp"
#include "caffe/layer.hpp"
#include "caffe/proto/caffe.pb.h"

namespace caffe {

template <typename Dtype>
class MyReshapeLayer : public Layer<Dtype> {
 public:
  explicit MyReshapeLayer(const LayerParameter& param)
      : Layer<Dtype>(param) {}
  virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);

  virtual inline const char* type() const { return "MyReshape";
     }

 protected:
```

```
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {}
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const
          vector<Blob<Dtype>*>& bottom) {}
};

}  // namespace caffe

#endif  // CAFFE_MYRESHAPE_LAYER_HPP_
```

Apéndice B

Código C++ para ejecutar la red

B.1. Módulo principal

```
#define USE_OPENCV 1

#include <caffe/caffe.hpp>
#ifdef USE_OPENCV
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#endif  // USE_OPENCV
#include <algorithm>
#include <iosfwd>
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>
#include "image.hpp"

using namespace caffe;  // NOLINT(build/namespaces)
using std::string;

class ResNet {
 public:
  ResNet(const string& model_file, const string& trained_file);
  std::vector<float> Predict(const cv::Mat& img);

 private:
```

```
  void WrapInputLayer(std::vector<cv::Mat>* input_channels);
  void Preprocess(const cv::Mat& img, std::vector<cv::Mat>*
      input_channels);

 private:
  shared_ptr<Net<float> > net_;
  cv::Size input_geometry_;
  int num_channels_;
};

ResNet::ResNet(const string& model_file, const string&
    trained_file) {
    #ifdef CPU_ONLY
      Caffe::set_mode(Caffe::CPU);
    #else
      Caffe::set_mode(Caffe::GPU);
    #endif

  /* Load the network. */
  net_.reset(new Net<float>(model_file, TEST));
  net_->CopyTrainedLayersFrom(trained_file);

  CHECK_EQ(net_->num_inputs(), 1) << "Network should have
      exactly one input.";
  CHECK_EQ(net_->num_outputs(), 1) << "Network should have
      exactly one output.";

  Blob<float>* input_layer = net_->input_blobs()[0];
  num_channels_ = input_layer->channels();
  CHECK(num_channels_ == 3 || num_channels_ == 1) << "Input
      layer should have 1 or 3 channels.";
  input_geometry_ = cv::Size(input_layer->width(),
      input_layer->height());

  Blob<float>* output_layer = net_->output_blobs()[0];
}

std::vector<float> ResNet::Predict(const cv::Mat& img) {

  Blob<float>* input_layer = net_->input_blobs()[0];
  input_layer->Reshape(1, num_channels_,
      input_geometry_.height, input_geometry_.width);

  /* Forward dimension change to all layers. */
  net_->Reshape();

  std::vector<cv::Mat> input_channels;
  WrapInputLayer(&input_channels);
  Preprocess(img, &input_channels);
```

```
  net_->Forward();

  /* Copy the output layer to a std::vector */
  Blob<float>* output_layer = net_->output_blobs()[0];
  const float* begin = output_layer->cpu_data();
  const float* end = begin +
      output_layer->width()*output_layer->height();
  return std::vector<float>(begin, end);
}

void ResNet::WrapInputLayer(std::vector<cv::Mat>*
    input_channels) {

  Blob<float>* input_layer = net_->input_blobs()[0];

  int width = input_layer->width();
  int height = input_layer->height();
  float* input_data = input_layer->mutable_cpu_data();
  for (int i = 0; i < input_layer->channels(); ++i) {
    cv::Mat channel(height, width, CV_32FC1, input_data);
    input_channels->push_back(channel);
    input_data += width * height;
  }
}

void ResNet::Preprocess(const cv::Mat& img,
    std::vector<cv::Mat>* input_channels) {
  /* Convert the input image to the input image format of the
     network. */
  cv::Mat sample;
  if (img.channels() == 3 && num_channels_ == 1)
    cv::cvtColor(img, sample, cv::COLOR_BGR2GRAY);
  else if (img.channels() == 4 && num_channels_ == 1)
    cv::cvtColor(img, sample, cv::COLOR_BGRA2GRAY);
  else if (img.channels() == 4 && num_channels_ == 3)
    cv::cvtColor(img, sample, cv::COLOR_BGRA2BGR);
  else if (img.channels() == 1 && num_channels_ == 3)
    cv::cvtColor(img, sample, cv::COLOR_GRAY2BGR);
  else
    sample = img;

  cv::Mat sample_resized;
  if (sample.size() != input_geometry_)
    cv::resize(sample, sample_resized, input_geometry_);
  else
    sample_resized = sample;

  cv::Mat sample_float;
```

```
  if (num_channels_ == 3)
    sample_resized.convertTo(sample_float, CV_32FC3);
  else
    sample_resized.convertTo(sample_float, CV_32FC1);

  /* This operation will write the separate BGR planes directly
     to the input layer of the network because it is wrapped by
     the cv::Mat objects in input_channels. */
  cv::split(sample_float, *input_channels);

  CHECK(reinterpret_cast<float*>(input_channels->at(0).data) ==
     net_->input_blobs()[0]->cpu_data()) << "Input channels are
     not wrapping the input layer of the network.";
}

int main() {

  string model_file   = "ResNet-50UpProj-deploy.prototxt";
  string trained_file = "ResNet50UpProj.caffemodel";

  ResNet model(model_file, trained_file);

  if(openVideo(1)!=0) {
    cout << "Camera ???" << endl;
  }

  do {
      cv::Mat img = nextFrame();

      CHECK(!img.empty()) << "Unable to decode image" <<
          std::endl;
      std::vector<float> depth = model.Predict(img);

      float max_depth = 0.0;
      float min_depth = 100.0;

      for(int i = 0; i < depth.size(); i++) {
        if (depth[i] > max_depth) {
            max_depth = depth[i];
        }
        if (depth[i] < min_depth) {
            min_depth = depth[i];
        }
      }
      cv::Mat depth_map(128, 160, CV_32FC1);

      for(int i = 0; i < 128; i++) {
        for(int j = 0; j < 160; j++) {
```

```
                depth_map.at<float>(i,j) =
                    (depth[i*160+j]-min_depth)/(max_depth-min_depth);
            }
        }
        cv::imshow("Depth Map", depth_map);
    } while(isKeyPressed()== -1);
}
```

B.2. Módulo auxiliar para abrir la cámara

```
// Include openCV library
#include <opencv2/opencv.hpp>
#include <opencv2/videoio.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <iostream>

// Define opencv namespace
using namespace cv;
using namespace std;

static VideoCapture cap;     // Video Source

cv::Mat nextFrame() {
    cv::Mat image;
    // Check camera/video has been opened
    if(!cap.isOpened()) {  // Check if if is opened
        cout << "Cannot access video frame" << endl ;
    }

    // Read image
    if(!cap.read(image)) {
        cout <<  "There are not more frames" << endl ;
    }

    // Show image
    imshow("Display Video window", image );
    return image;
}

int openVideo(int dev) {

    cap = VideoCapture(dev);

    if(!cap.isOpened()) {  // Check if if is opened
        cout <<  "Could not open camera" << endl ;
```

```
                return -1;
    }
        namedWindow( "Display Video window", WINDOW_AUTOSIZE
            );// Create a window for display.
    return 0;
}

void closeVideo() {
    cap.release();
}

int isKeyPressed() {
    return waitKey(10);
}
```

Bibliografía

[1] Comet Labs Research Team, "Depth sensors are the key to unlocking next level computer vision applications", https://blog.cometlabs.io/depth-sensors-are-the-key-to-unlocking-next-level-computer-vision-applications-3499533d3246

[2] Wikipedia, "Machine Learning", https://en.wikipedia.org/wiki/Machine_learning

[3] Wikipedia, "Outline of Machine Learning", https://en.wikipedia.org/wiki/Outline_of_machine_learning

[4] Wikipedia, "Decision tree', https://en.wikipedia.org/wiki/Decision_tree

[5] Wikipedia, "Association Rule Learning", https://en.wikipedia.org/wiki/Association_rule_learning

[6] Wikipedia, "Artificial Neural Network", https://en.wikipedia.org/wiki/Artificial_neural_network

[7] Wikipedia, "Cluster Analysis", https://en.wikipedia.org/wiki/Cluster_analysis

[8] Wikipedia, "Bayesian Network", https://en.wikipedia.org/wiki/Bayesian_network

[9] S. Marsland, *Machine Learning, An Algoritmic Perspective*. Boca Ratón, FL: CRC Press, 2015

[10] Wikipedia, "TensorFlow" https://es.wikipedia.org/wiki/TensorFlow

[11] TensorFlow webpage, "API Documentation", https://www.tensorflow.org/api_docs/

[12] TensorFlow webpage, "Tensors",
`https://www.tensorflow.org/guide/tensors`

[13] TensorFlow webpage, "Graph",
`https://www.tensorflow.org/guide/low_level_intro#graph`

[14] TensorFlow webpage, "TensorBoard",
`https://www.tensorflow.org/guide/low_level_intro#tensorboard`

[15] TensorFlow webpage, "Layers",
`https://www.tensorflow.org/guide/low_level_intro#layers`

[16] Caffe webpage, "Caffe",
`http://caffe.berkeleyvision.org/`

[17] Y. Jia, E. Shelhamer, J.Donahue, S.Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *Proceedings of the 22nd ACM international conference on Multimedia*, ACM, 2014. pp. 675-678.

[18] Caffe webpage, "Layer computation and connections",
`http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`

[19] Protocol Buffers, "Developer guide",
`https://developers.google.com/protocol-buffers/docs/overview`

[20] Caffe webpage, "Net definition and operation",
`http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`

[21] Wikipedia, "Convolutional Neural Network"
`https://en.wikipedia.org/wiki/Convolutional_neural_network`

[22] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, 2016, pp. 770-778.

[23] GitHub, "ResNet50 Netscope",
`http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006`

[24] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In *ICML*, 2015.

[25] I. Laina, C. Rupprecht, V. Belagiannis, F. Tombari and N. Navab, "Deeper Depth Prediction with Fully Convolutional Residual Networks," *2016 Fourth International Conference on 3D Vision (3DV)*, Stanford, CA, 2016, pp. 239-248.

[26] P. K. Nathan Silberman, Derek Hoiem and R. Fergus. Indoor segmentation and support inference from RGBD images. In *ECCV*, 2012.

[27] ResNet-50 Up-Projection model for TensorFlow, `http://campar.in.tum.de/files/rupprecht/depthpred/NYU_FCRN-checkpoint.zip`

[28] ResNet-50 Up-Projection model for Matlab, `http://campar.in.tum.de/files/rupprecht/depthpred/NYU_ResNet-UpProj.zip`

[29] NVIDIA, *Jetson TX2 Developer Kit, User Guide*, NVIDIA Corporation, 2017

[30] GitHub, "ResNet-50 Deploy Model for Caffe", https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-50-deploy.prototxt

[31] GitHub, "Permute Layer for Caffe", `https://github.com/BVLC/caffe/commit/b68695db42aa79e87429607192753636 3fe1efbf`

[32] "Stereo and Depth-from-Defocus Dataset 2: The average office", `http://devernay.free.fr/vision/focus/office/`