Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Detección y clasificación de señales de tráfico basado Deep Learning

Autor: Juan José Morales Calvo

Tutor: Noelia Hernández Parra

2021/2022

Universidad de Alcalá

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado
Detección y clasificación de señales de tráfico basado Deep Learning

Autor: Juan José Morales Calvo

Tutor: Noelia Hernández Parra

TRIBUNAL:

Presidente: D. Ignacio Parra Alonso

Vocal 1º: D. Melquiades Carbajo Martín

Vocal 2º: Dña. Noelia Hernández Parra

FECHA: 20/06/2022

Nunca te rindas...

"Los errores no están en el arte, sino en los artífices."
Isaac Newton.

Agradecimientos

En nuestros locos intentos, renunciamos a lo que somos por lo que esperamos ser.

William Shakespeare

La realización de este trabajo ha sido fruto del trabajo y sacrificio, por supuesto del autor, sino también de mi familia, amigos y profesores.

En primer lugar me gustaría agradecer a mis padres por siempre buscar mi bien por encima del suyo, perdiendo horas de su tiempo en facilitarme a mi las cosas.

Dar gracias a mis amigos, que el haber pasado momentos difíciles juntos ha fortificado nuestros lazos. Les llevaré siempre en la memoria a donde me depare el futuro.

Finalmente agradecer a Noelia que desde primero ya supe por su dedicación que quería realizar el TFG con ella. Sin duda la universidad necesita profesores como ella que te hacen interesarte por la asignatura y querer ir a las 8:00 de la mañana a una clase de programación.

Índice general

Índice de figuras

Índice de tablas

Resumen

El desarrollo del trabajo constará en la explicación y comprensión de las diferentes partes que forman un sistema autónomo, basado en Deep Learning, capaz de detectar diferentes tipos de señales de tráfico para su correcto cumplimiento en la vía pública.

Se llevará a cabo un estudio teórico general de la redes neuronales convolucionales y del software utilizado para su implementación. Se explicarán las diferentes partes del programa, implementado en el lenguaje Python, entrando en detalle de cada una de sus funciones y resultados que puedan producir.

Finalmente se realizará un estudio de los parámetros más importantes del programa con el objetivo de buscar los mejores resultados durante el entrenamiento de la red.

Palabras clave: Redes Neuronales Convolucionales, Python, Pytorch, Modelos con pesos pre-entrenados.

Abstract

The development of the work will consist of the explanation and understanding of the different parts that make up an autonomous system, based on Deep Learning, capable of detecting different types of traffic signals for their correct compliance on public roads.

A general theoretical study of convolutional neural networks and the software used for their implementation will be carried out. The different parts of the program, implemented in the Python language, will be explained, going into detail about each of its functions and the results they can produce.

Finally, a study of the most important parameters of the program was carried out in order to find the best results during network training.

Keywords: Convolutional Neuronal Network, Python, Pytorch, Dataset, Pre-trained models.

Resumen extendido

El objetivo principal del trabajo será el de crear un sistema capaz de clasificar diferentes tipos de señales de tráfico basado en Deep Learning. El trabajo se centra en el uso del módulo *Pytorch* para poder crear una red capaz de diferenciar entre diferentes tipos de señales de tráfico dispuestas en diferentes posiciones y ángulos.

Se realiza el estudio de los conceptos teóricos de las redes neuronales convolucionales, las capas que las forman y la función de cada una. También se realizará el estudio del módulo *Pytorch*, comprendiendo la función y la correcta utilización de cada una de las funciones del módulo para poder obtener los mejores resultados.

Se realiza el estudio y se hará uso del *dataset* de imágenes utilizado en GTSRB (The German Traffic Sign Benchmark) que nos permite clasificar más de 43 tipos de clases de señales de tráfico contando con más de 40 mil imágenes.

Se realiza el estudio de las diferentes redes neuronales convolucionales con pesos pre-entrenados proporcionadas por el módulo Pytorch explicando también su implementación en nuestro sistema. En concreto se analizarán y usarán ResNet, Inception y AlexNet.

Se implementan en lenguaje Python los diferentes bucles de entrenamiento y validación a través del espacio de trabajo Google Colab que permite, de manera gratuita, acceder a estaciones de trabajo de alto rendimiento para el entrenamiento de la red.

Se implementa, a partir del módulo *Matplotlib*, funciones que permiten comparar los diferentes resultados obtenidos, durante el entrenamiento y validación, de manera gráfica para facilitar la comprensión de los propios resultados.

Posteriormente, a través del espacio de trabajo Google Colab se realiza el estudio de los hiperparámetros más importantes que afecten al rendimiento de la red. También se experimenta con las diferentes redes con pesos pre-entrenados expuestas. Se parte de un sistema base y a partir de los resultados obtenidos con los diferentes hiperparámetros y redes se llegará a concluir cual será el sistema final que nos proporciona los mejores resultados.

Finalmente se exponen las conclusiones de la experimentación a partir de la comprensión de los resultados obtenidos y las líneas futuras que se podrán seguir partiendo de este trabajo como base.

Capítulo 1

Introducción

1.1 Motivación

La Organización Mundial de la Salud (OMS) estima que mueren al año 1,3 millones de personas en las carreteras, siendo la primera causa de muerte de las personas de entre 5 y 29 años y estiman que alrededor de 50 millones de personas sufren lesiones graves. El problema en las carreteras es evidente causando daños humanos, materiales, sanitarios y administrativos.

Poniendo un ejemplo, una de las causas principales en la sucesión de accidentes en las carreteras interurbanas es la velocidad, inadecuada, a la que circulan los conductores de dichas vías, no siendo conscientes del mal que pueden causar. Sin embargo, la motivación de este libro no es buscar las causas ni culpables a este problema evidente en la sociedad, sino a aportar una solución que pudiese prevenirlo con ayuda de la tecnología.

Es por ello que se implementa una solución para que una máquina detecte e identifique las señales de tráfico de manera que sea posible tomar las acciones necesarias para proteger al conductor y su alrededor.

Para cumplir con nuestro objetivo se hará uso de la tecnología Deep Learning que otorga a una máquina la capacidad de pensar como si estuviese provista de un cerebro humano. Deep Learning es un subconjunto de machine learning muy utilizado en la visión computacional, que se basa en el estudio y desarrollo de maquinas que son capaces de aprender.

El origen del aprendizaje profundo fue en 1959 cuando se encontraron células simples y células complejas en la corteza visual del gato (Wiesel y Hubel, 1959). Estas células se disparan en respuesta a ciertas propiedades de las entradas sensoriales visuales, como la orientación de los bordes. Las células complejas se mostraban con una mayor variación espacial que las células simples. Se considera que este descubrimiento fue el que inspiró las redes neuronales profundas. Fue en 1965 cuando Ivakhnenko y Lapa idearon la primera red, Networks trained by the Group Method of Data Handling (GMDH), que se considera el primer sistema basado en Deep Learning [16].

Las redes neuronales profundas entre 1990 y el año 2000 muestran un gran avance captando la atención en el ámbito de inteligencia artificial debido a su mejor rendimiento frente a otros métodos de Machine Learning como las Máquinas de soporte vectorial (Vapnik, 1995; Scholkopf et al., 1998).

Fue en 2011 cuando se gana la competición internacional de reconocimiento de patrones logrando los primeros resultados de reconocimiento de patrones visuales en dominios limitados [16].

En la actualidad el Deep Learning es usado en múltiples sectores como por ejemplo la medicina, permitiendo un diagnóstico temprano de enfermedades potencialmente mortales mejorando los resultados en el tratamiento de dichas enfermedades, o en automóviles autónomos, garantizando una conducción segura detectando cualquier peligro y llevando a cabo la acción más adecuada [17].

Es por ello que esta tecnología y la inteligencia artificial en general, están adquiriendo tanta importancia en su necesidad de desarrollarla para el bien común del ser humano.

1.2 Objetivos

El objetivo principal de este TFG es desarrollar una aplicación capaz de identificar las señales de tráfico con un error lo mas pequeño posible. Se utilizarán distintos parámetros, redes convolucionales y algoritmos de optimización para estudiar la mejor solución y combinación de estos. Los siguientes puntos a estudiar y valorar son:

- El estudio del lenguaje de programación Python.
- El estudio de la librería Torch y Torchvision para la creación de Datasets y uso de las redes convolucionales.
- El estudio de la librería Numpy para la manipulación de los arrays y los datos de las imágenes.
- El estudio de la librería Matplotlib para poder mostrar los datos de una manera gráfica facilitando así su exposición y comprensión.
- El estudio del funcionamiento de Google Colab para el uso de GPU potentes para facilitar el entrenamiento de la red.
- El estudio y comprensión de las diferentes redes convolucionales ya pre-entrenadas.
- El estudio y comprensión de los diferentes parámetros modificables para su uso en el entrenamiento.
- El estudio y comprensión de los diferentes algoritmos de optimización para su uso en el entrenamiento.
- Análisis y comparación de los datos obtenidos.
- Búsqueda de la perfección en la detección de las señales de tráfico.

1.3 Organización de la memoria

La memoria se encuentra dividida en cinco capítulos:

1. Introducción. Capítulo en el que se expone la motivación del trabajo realizado.
2. Estado del arte. Descripción teórica de los elementos a usar y modificar para su comparación de los resultados.
3. Descripción del sistema. Explicación del funcionamiento del sistema.
4. Experimentación del sistema. Parte más extensa en la que se muestra los diferentes valores de los entrenamientos haciendo modificaciones en los parámetros explicados antes y comparando dichos resultados
5. Conclusiones y líneas futuras. Se expone la explicación del porqué se elige un sistema u otro en base a los datos obtenidos en los resultados. También se expondrán posibles mejoras futuras del trabajo realizado.

Capítulo 2

Estado del arte

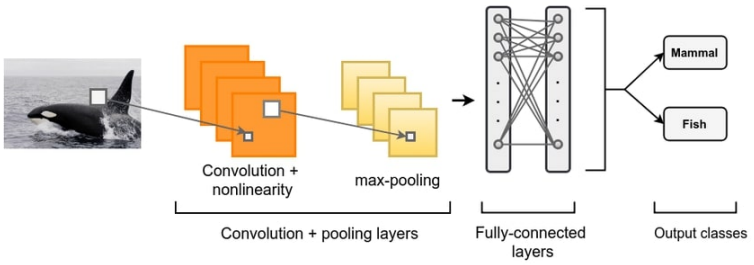
2.1 Redes neuronales convolucionales

Las redes neuronales convolucionales, también conocidas por su abreviación CNN, son redes neuronales jerárquicas en las que se alternan capas de convolución y capas de agrupación (pooling) equivaliendo a las células simples y complejas en la corteza visual primaria de nuestro cerebro biológico. Son perfectas para la clasificación de imágenes debido a que se especializan en procesar información que se pueda dividir en diferentes celdas equivaliendo a los píxeles de la imagen.

Fue en 1959 cuando D.H Hubel y T.N Wiesel sugirieron un nuevo modelo de como los mamíferos percibían el mundo visualmente, describiendo dos tipos de células:

- S cells: Se activaban cuando detectaban formas básicas como líneas en un área fija [18].
- C cells: tienen campos receptivos más grandes y su salida no es sensible a la posición específica [18].

Estas redes están formadas en múltiples capas convergiendo todas en una final (fully conected). A continuación se realiza una explicación general de todas las capas.



Redes Convolucionales

Figura 2.1: Estructura general de las redes convolucionales [5].

2.1.1 Capa convolucional

La capa de convolución se basa en la reducción de la dimensión y obtención de las características de las imágenes de entrada a tratar a través, como su nombre indica, de operaciones de convolución.

Cada capa convolucional estará formada por un campo receptivo, a la entrada, de tamaño M (M_x, M_y) que es la región que obtendríamos del *input* para realizar la operación de convolución con el filtro convolucional, también llamado *kernel*, de tamaño K (K_x, K_y) que nos permite obtener siempre el mismo número de características a la salida de la capa.

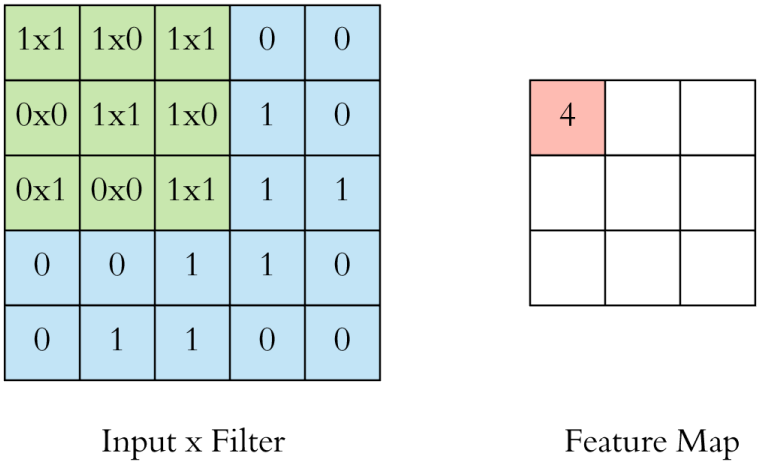


Figura 2.2: Color verde muestra el filtro 3x3 utilizado, el color azul el *input* y el color rojo el resultado de la convolución [6].

Los *skipping factors* S (S_x, S_y) definen cual será el tamaño de salto del filtro kernel en cuanto a píxeles (desplazamiento de la matriz de color verde en la figura 2.2) en dirección X y dirección Y entre operaciones convolucionales.

Por lo que habiendo definido todos los factores, el tamaño de la salida sería:

$$M_x^n = \frac{M_x^{n-1} - K_x^n}{S_x^n + 1} + 1; M_y^n = \frac{M_y^{n-1} - K_y^n}{S_y^n + 1} + 1;$$

donde n indica la capa convolucional a la que nos referimos [19].

Los puntos a tener en cuenta en una capa convolucional son [20]:

- La profundidad de la imagen de entrada debe ser la misma que la del filtro utilizado.
- Las capas iniciales de convolución utilizarán un menor número de filtros para extraer características de alto nivel.
- Las capas más profundas obtendrán mayor número de características pero serán computacionalmente más costosas.
- Se deben introducir filtros no lineales para hacer más simple la suma.

2.1.2 Función ReLU

La unidad lineal rectificada, también conocida como ReLU, puede ser considerada como una parte de las operaciones convolucionales o considerarla como una parte apartada de la capa de convolución. En este caso la definiremos aparte debido a su importancia en el tratamiento de las imágenes.

El principal objetivo de la utilización de este rectificador es incrementar la no linealidad en las imágenes de entrada que por su propia naturaleza no son lineales. El rectificador elimina la parte negativa definiendo la parte positiva del argumento:

$$f(x) = x^+ = max(0, x)$$

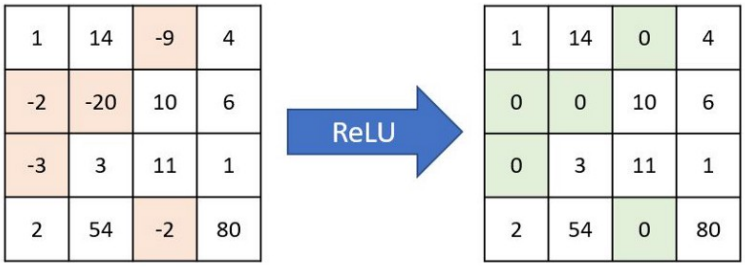


Figura 2.3: Función rectificadora ReLU [6].

Un ejemplo del uso de ReLU sería en el caso de una imagen en blanco y negro, que estaría dispuesta en 3 colores: blanco, negro y gris, siendo el gris de distintas tonalidades. El hecho de utilizar el rectificador eliminaría el color negro al ser negativo y dispondría la imagen en color blanco y gris en distintas tonalidades.

La diferencia entre la imagen rectificada y la que no se le ha aplicado la rectificación es la progresión de los colores. En el caso de la versión sin rectificar la progresión sería lineal observando tonos blancos seguidos de grises y finalmente de un color negro. Sin embargo, la versión rectificada (no lineal) tiene cambios más abruptos sin signos de linealidad [7].

Figura 2.4: Non-ReLU vs ReLU [7].

2.1.3 Capa de agrupación

La capa de agrupación tiene como principales objetivos los siguientes puntos:

- La reducción de la dimensión de la salida de la capa de convolución que nos permite realizar las operaciones computacionales en un menor espacio de tiempo.
- Reducir la probabilidad de sobre ajuste (*overfitting*).
- Permite que frente a imágenes en diferentes ángulos, escenarios y iluminación la red neuronal pueda clasificar correctamente la imagen, es decir, reducir las invarianzas.

Al contrario que en la capa de convolución, la capa de agrupación no tiene parámetros de entrada.

A la salida de la capa de convolución *feature map* se le aplicará un filtro/ventana normalmente de dimensiones 2x2, aunque al igual que en la capa de convolución se definirá la ventana y el salto de dicha ventana por el *feature map*.

La capa toma los cuatro valores de dicha ventana y realiza la operación determinada. Existen diferentes tipos de operaciones que podemos realizar en la agrupación, entre ellos: agrupación máxima, agrupación promedio y agrupación suma, que se explican a continuación.

2.1.3.1 Agrupación máxima

Se obtienen los valores máximos de la ventana, permitiendo una convergencia más rápida, seleccionar características invariantes superiores, y mejorar la generalización [19].

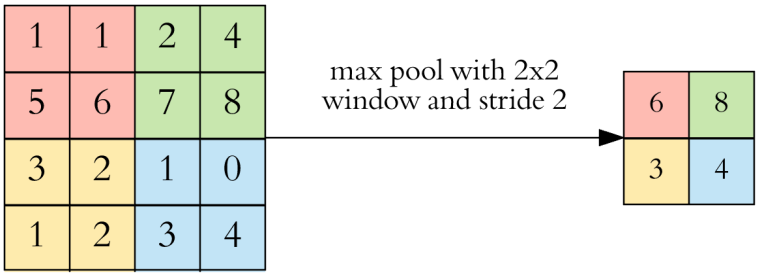


Figura 2.5: Funcionamiento de la capa de agrupación máxima [8].

Se observa en la figura 2.5 como se utiliza una ventana de 2x2, tomando cuatro muestras, y un salto de 2 celdas. Se toman, como ya se mencionó en la explicación de la capa de agrupación máxima, los valores mas altos de la ventana.

2.1.3.2 Agrupación promedio

Se obtienen la media de los valores de la ventana. Reduce la muestra no preservando las características máximas obtenidas y introduce invarianza en la traslación. Extrae características de forma más suave y no tan pronunciado como la capa de agrupación máxima, mientras que la capa de agrupación máxima extrae características más pronunciadas como bordes [21].

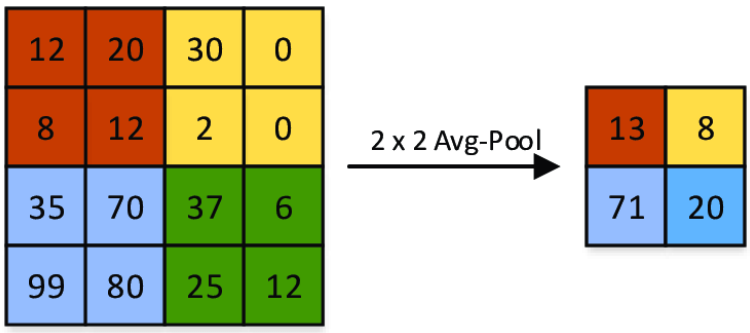


Figura 2.6: Funcionamiento de la capa de agrupación promedio [9].

Se observa en la figura 2.6 como se utiliza una ventana de 2x2, tomando cuatro muestras, y un salto de 2 celdas. Se toman, como ya se mencionó en la explicación de la capa de agrupación promedio, la media formada por las cuatro muestras de la ventana.

2.1.3.3 Agrupación suma

Derivado de la agrupación máxima, se obtienen los valores de la suma de los cuatro valores de la ventana. Menos utilizado que la agrupación máxima y promedio.

Figura 2.7: Funcionamiento de la capa de agrupación suma.

Se observa en la Figura 2.7 como se utiliza una ventana de 2x2, tomando cuatro muestras, y un salto de 2 celdas. Se toman, como ya se mencionó en la explicación de la capa de agrupación promedio, la suma formada por las cuatro muestras de la ventana.

2.1.4 Capa de aplanado

Una vez recibido el *feature map* de la capa de agrupación seleccionada, tendremos que prepararla para la capa final de clasificación. Tendremos que convertir dicha matriz de ya sea 3x3 o 2x2 a una única columna, de ahí la palabra *flattened*, que sirva como entrada a la propia capa *fully-connected*.

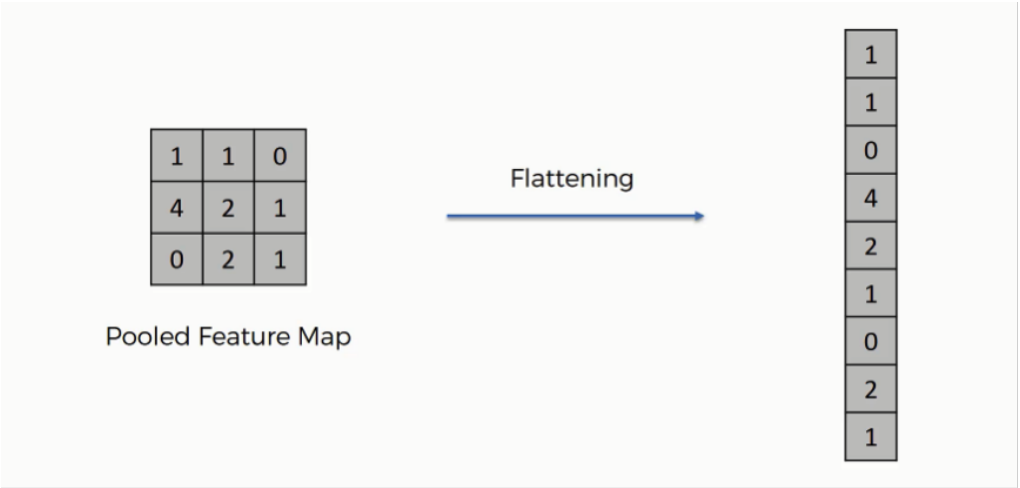


Figura 2.8: Funcionamiento de la capa de aplanamiento [10].

2.1.5 Capa completamente conectada

Una vez realizadas las operaciones convolucionales, rectificaciones, agrupaciones y aplanamiento es la capa completamente conectada la encargada de conectar todas las características y por lo tanto, realizar la clasificación de las imágenes.

La capa completamente conectada estará formada por 3 partes:

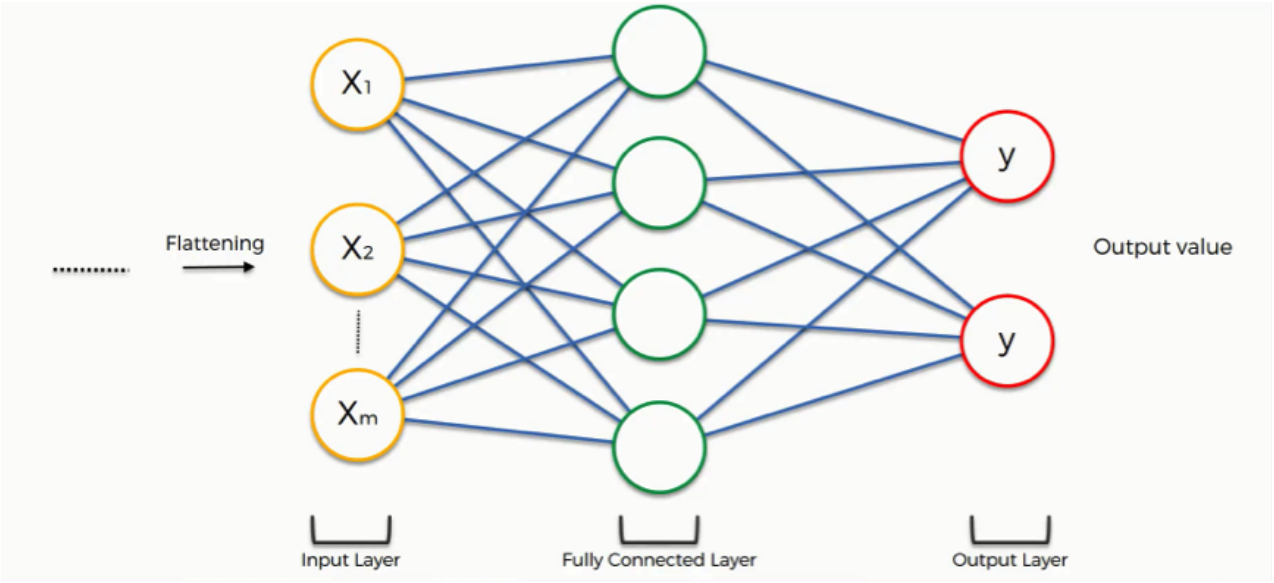


Figura 2.9: Funcionamiento de la capa totalmente conectada [10].

- *Input Layer*: Capa de entrada con el número de neuronas igual al número de características del vector de 1 dimensión aplanado.
- *Fully Connected Layer*: la disposición de un número determinado de neuronas que conectan todas las entradas con cada una de las salidas.
- *Output Layer*: Capa de salida con el número de neuronas igual al número de posibles resultados o tipos de imágenes.

Las características de la imagen introducida a la entrada convergerán todas en la capa *fully connected*, en la que se realizarán los cálculos relativos a la predicción de la clase a la que pertenece la imagen. A partir de esta predicción obtendremos el valor del error también llamada como *loss function*. A partir de este error la red deberá evolucionar para mejorar la efectividad en sus predicciones, modificando los pesos de determinadas características y atributos gracias al entrenamiento realizado con la conocida como propagación hacia atrás o *backpropagation*.

En la figura 2.9 las líneas azules, conocidas como sinapsis debido a su analogía con neuronas, tienen un peso determinado entre su par. Por lo que cada neurona de la capa completamente conectada tendrá un porcentaje de cada valor de la entrada y dependiendo de dichos valores, la red llegará a una conclusión de cual es la clase de la imagen de entrada. Es por ello que si realiza una mala predicción, mediante diferentes optimizadores podremos modificar los pesos de dichas sinapsis para que la predicción sea la correcta.

También se deberán modificar los detectores de características debido a que muchas veces no se centran en las características correctas. Es por ello que podrá ignorar ciertas sinapsis y poner toda su atención en las conexiones adecuadas.

Este entrenamiento nos permite ajustar los pesos que vamos a otorgar a cada neurona y sinapsis de la red para que una vez ya entrenado, podamos clasificar las imágenes con un error mínimo.

2.1.5.1 Entropía Cruzada para función de pérdida

Cross-entropy, es la función mayormente utilizada para realizar la función de pérdida también conocida como *loss function*. Cuanto mayor sea la divergencia entre el valor real y el predicho aumentará el valor de la entropía entre 0 y 1.

En la clasificación binaria la entropía cruzada se puede calcular como:

$$-(y * logp + (1 - y) * log1 - p))$$

Siendo y la indicación si la predicción es correcta y p la predicción realizada según los pesos.

2.2 PyTorch

PyTorch es un *framework* automático de código abierto que acelera el camino desde la creación de prototipos de investigación hasta la implementación de producción [22].

Sus principales capacidades y características son [22]:

- Listo para producción.
- Entrenamiento distribuido.
- Ecosistema robusto.
- Soporte de la nube.

Librería muy usada en el ámbito de la inteligencia artificial y gracias a su simplicidad en el uso y fácil manipulación de parámetros, será la opción a utilizar.

2.2.1 Torch

El paquete torch contiene estructuras de datos para tensores/arrays multidimensionales, como es el caso en el tratamiento de las imágenes y define operaciones matemáticas sobre estos tensores/arrays. Además, proporciona muchas utilidades para la serialización eficiente de tensores y tipos arbitrarios, y otras utilidades [23].

De todas sus funcionalidades, se destacan las más utilizadas en el entrenamiento de la red:

- Torch.no_grad(): Durante el entrenamiento de la red, en el momento de validación no será necesario calcular el gradiente ni hacer uso de la función backward(), todo relacionado con el entrenamiento y actualización de los pesos calculando la función pérdida. También nos permitirá un menor consumo de computación de la máquina, realizando los cálculos con una mayor velocidad.

  No dispone de parámetros.

- Torch.backward(): Cálculo del gradiente de la pérdida de todos los *input* introducidos siempre que el parámetro requires_grad = True. Podremos acceder a el valor computado a partir de x.grad(), siendo x el *input* deseado.

  Dispone de los siguientes parámetros [24]:

  - tensors: Los tensores/arrays en los que se realiza el cómputo de la función pérdida.
  - grad_tensors: Argumento opcional relacionado con el producto del vector jacobiano.
  - retain_graph: Variable booleana, dependiendo de su valor se libera o no el gráfico utilizado para calcular el gradiente.
  - create_graph: Variable booleana, si su valor es True, se construirá el gráfico de la derivada, lo que permitirá calcular productos derivados de orden superior. El valor predeterminado es False.
  - inputs: entradas en las que se acumulará el gradiente .grad.

2.2.1.1 Torch.cuda

Módulo de Torch que permite mediante *Torch.cuda.device()* decidir si la tarea computacional será realizada por la GPU o CPU del ordenador, siendo la primera la más indicada para tareas de cómputo. Principalmente la diferencia entre ambas será en el tiempo necesario para completar el entrenamiento de la red.

2.2.1.2 Torch.nn

Módulo de Torch que será principalmente usado para crear una red neuronal convolucional especificando cada una de las capas que el programador desee. Se podrán hacer uso de diferentes contenedores, siendo el más utilizado el secuencial, en el que se establecerá en una lista cual serán las capas por las que pasarán las características de manera secuencial como su propio nombre indica.

- Torch.nn.Sequential(): Contenedor secuencial en el que se le irán pasando las capas en orden al que se les pasa al constructor. No tiene un número máximo de argumentos siendo estos las propias capas deseadas, poniendo un ejemplo como la capa ReLU, una capa convolucional, o una capa de Pooling.

Este módulo es una opción a tener en cuenta a la hora de utilizar una red neuronal convolucional que el propio usuario pueda manipular y controlar, pero se optará por el uso de redes ya creadas y pre-entrenadas que poseen una efectividad demostrable.

2.2.1.3 Torch.optim

Módulo de Torch que contiene varios algoritmos de optimización. Para usarlo se debe construir un objeto optimizador, que mantendrá el estado actual y actualizará los parámetros en función de los gradientes calculados y los valores otorgados al learning rate y momentum [25].

Cada uno de los optimizadores del módulo, implementan la función step() *optimizer.step()* utilizado generalmente en cada paso por el bucle de entrenamiento, actualizando los valores de los parámetros. Sin embargo, solo se podrá hacer uso de step() cuando previamente se hayan calculado los gradientes con la ya mencionada función backward() [25].

Se observa en la documentación de PyTorch que actualmente están disponibles 13 algoritmos de optimización para utilizar. Solo se explicarán los utilizados en la fase de experimentación, que son los más comúnmente utilizados en el estado del arte:

- Adam:

  Presentado por Diederik Kingma y Jimmy Ba en 2015 basado en gradientes de primer orden de funciones objetivos estocásticas que estima momentos de orden menor [26].

  Esto es debido a que contiene ventajas de ambos como el uso de una tasa de aprendizaje fija por parámetro que mejora el rendimiento en el cómputo de gradientes dispersos implementado en AdaGrad y también mantiene tasas de aprendizaje que se adaptan según los gradientes previos permitiendo un buen funcionamiento en problemas muy ruidosos implementado en RMSprop [27].

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

---

return θ_t

Figura 2.10: Algoritmo Adam [11].

Se explican únicamente los parámetros que introduciremos en el optimizador:

- Params: Parámetros que van a ser sometidos al proceso de optimización.
- LR: *learning rate*, porcentaje o peso que se le otorga al nuevo valor del parámetro.

- SGD:

  Implementa el algoritmo de optimización de descenso de gradiente estocástico, como sus siglas indican, reduciendo la carga computacional gracias a la aleatoriedad en la elección de los puntos de los datos que se escogen para calcular las derivadas [28].

  Se hará uso del parámetro momentum que nos permite obtener estimaciones más lineales reduciendo la oscilación debido a que el gradiente calculado siempre depende del anterior.

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov, maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_{t-1} + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

Figura 2.11: Algoritmo SGD [12].

Se explican únicamente los parámetros que introduciremos en el optimizador:

- Params: Parámetros que van a ser sometidos al proceso de optimización.
- LR: *learning rate*, porcentaje o peso que se le otorga al nuevo valor del parámetro.
- *Momentum*: Valor del *momentum*, que es el peso del gradiente previo con el actual.

- Adadelta:

  El algoritmo de optimización Adadelta es una extensión del algoritmo ya comentado SGD. Disminuye la carga computacional respecto SGD acelerando el propio proceso de optimización debido a que se necesitan menos número de evaluaciones para alcanzar el valor óptimo [29].

  El método no requiere ajuste manual de una tasa de aprendizaje y es robusto a datos ruidosos, diferentes arquitecturas de redes neuronales convolucionales, varias modalidades de datos y selección de hiperparámetros como por ejemplo learning rate [29].

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), ρ (decay), λ (weight decay)
initialize : $v_0 \leftarrow 0$ (square avg), $u_0 \leftarrow 0$ (accumulate variables)

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$if\ \lambda \neq 0$
$g_t \leftarrow g_t + \lambda\theta_{t-1}$
$v_t \leftarrow v_{t-1}\rho + g_t^2(1-\rho)$
$\Delta x_t \leftarrow \frac{\sqrt{u_{t-1}+\epsilon}}{\sqrt{v_t+\epsilon}} g_t$
$u_t \leftarrow u_{t-1}\rho + \Delta x_t^2(1-\rho)$
$\theta_t \leftarrow \theta_{t-1} - \gamma\Delta x_t$

return θ_t

Figura 2.12: Algoritmo Adadelta [13].

  Se explican únicamente los parámetros que introduciremos en el optimizador:

  - Params: Parámetros que van a ser sometidos al proceso de optimización.
  - LR: *learning rate*, porcentaje o peso que se le otorga al nuevo valor del parámetro.

El módulo optim también permite ajustar la tasa de aprendizaje *learning rate* gracias al uso de torch.optim.lr_scheduler(). Nos proporciona diferentes maneras de modificar el valor del learning rate según el número de *epoch* y el tipo de esquema que utilicemos.

A continuación se explican los que se van a utilizar en la parte de experimentación:

- MultiStepLR: Decae la tasa de aprendizaje según el valor de gamma a partir de una serie de *epochs* límite.
- ConstantLR: Decae la tasa de aprendizaje de cada grupo de parámetros por gamma hasta una iteración determinada [25].
- ReduceLROnPlateau: reducirá el *learning rate* en el momento que no obtengamos mejoras del resultado introducido en la función *step*.

2.2.1.4 Torch.utils

Enfocamos este módulo en su función *Torch.utils.data.Dataloader* debido a que es primordial para el tratamiento y carga de datos en las redes neuronales. Se puede obtener y cargar datos de manera iterativa permitiendo [30]:

- Conjunto de datos *datasets* con estilo de mapa e iterables.
- Modificar el orden de carga de los datos.
- *Batch* automático.
- Carga de datos con uni y multi-proceso.
- Fijación de memoria automática.

En cuanto a los parámetros utilizados para Dataloader serán:

- Dataset: Conjunto de datos de los que obtendremos las imágenes a clasificar.
- Batch_size: Cantidad de imágenes que cargaremos.
- Shuffle: Variable booleana que permite o no la aleatoriedad en la toma de datos.
- Num_workers: Número de procesos.

Es crucial la organización del conjunto de datos *dataset* para que el Dataloader pueda acceder sin problemas. Se propone en la fase de experimentación la división de las imágenes en 3 directorios que contendrán las imágenes utilizadas para entrenamiento, validación y test. En cada uno de estos directorios cada señal de tráfico tendrá su propia carpeta/directorio en el que se contendrán todas sus imágenes.

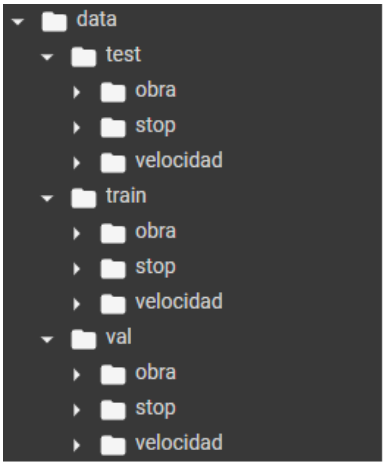


Figura 2.13: Ejemplo de estructura de dataset con 3 tipos de señales.

2.2.2 Torchvision

El módulo Torchvision cuenta con paquetes de datos, arquitecturas modelo, redes neuronales convolucionales pre-entrenadas y la posibilidad del tratamiento de imágenes y su modificación en sus características para mejorar el rendimiento y entrenamiento de la red, ya sea modificando sus colores, iluminación, altura y sentido [31].

2.2.2.1 Torchvision.transforms

La clase *Transforms* nos permite modificar las características de la imagen ya sea para poder representarla mejor aumentando su tamaño, o para modificar ciertos parámetros y que la red sea entrenada en casos, por ejemplo, de poca visibilidad. Denominado *Data Augmentation*, permite modificar la posición de la imagen para poder usarla en el entrenamiento con distintos "puntos de vista" y mejorar la capacidad de generalización de la red.

Las imágenes pueden ser tratadas como PIL (Python Imaging Library) o como tensores siendo ciertas funciones de *Transforms* solo accesibles en un formato u el otro.

La función *Compose*, dentro del paquete *Transforms*, nos permite juntar varias transformaciones que se aplicarán sobre las imágenes de un determinado *dataset*, asemejándose a *Torch.nn.Sequential*, incluso siendo esta última posible de utilizar en este apartado.

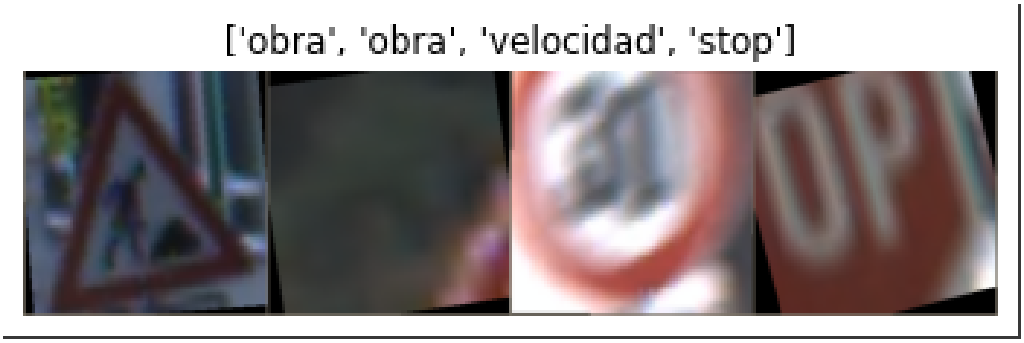


Figura 2.14: Diferentes señales de tráfico transformadas.

Las imágenes de las señales de tráfico en la figura 2.14 no son las originales de la base datos, sino una versión modificada debido a que en la vida real la condiciones no siempre son las mismas. Sin embargo, estas modificaciones siempre tienen que permitir diferenciar una señal con otra debido a que podría llevar a la red a entrenarse de una manera equivocada.

Las modificaciones de las imágenes deberán ser diferentes entre el estado de entrenamiento y validación, siendo necesario utilizar funciones que tengan el factor de aleatoriedad *Random* durante en el entrenamiento y dejando un tamaño y centrado fijo durante la validación.

2.2.2.2 Torchvision.dataset

Torchvision nos permite crear nuestro propio conjunto de imágenes y vincularlas/guardarlas en una determinada dirección de nuestro sistema. Brinda la posibilidad de usar dataset ya creados, o crear los nuestros propios con la función que se utiliza *ImageFolder*.

Posteriormente introduciremos en nuestro primer parámetro de *Torch.utils* la dirección creada con nuestras imágenes, creando así el *dataset*.

2.2.2.3 Torchvision.models

Torchvision nos permite utilizar modelos de CNN ya creados con pesos pre-entrenados para la clasificación de las imágenes, segmentación semántica por píxeles, detección de objetos, segmentación de instancias, detección de puntos clave de personas, clasificación de vídeo y flujo óptico [32].

Modelos que serán comparados en la fase de experimentación y explicados en la descripción del sistema. Se centra el estudio en modelos como ResNet, AlexNet y Inception.

Todos los modelos esperan la entrada de imágenes normalizadas de 3 canales de color RGB y de un tamaño de 224 píxeles tanto de alto como de ancho.

Capítulo 3

Descripción del Sistema

En nuestros locos intentos, renunciamos a lo que somos por lo que esperamos ser.

William Shakespeare

En este capítulo se exponen las partes que componen el sistema de detección de señales de tráfico, explicando la base de datos de la que extraemos las imágenes, las funciones utilizadas, el espacio de trabajo utilizado y las diferentes modificaciones posibles que se han realizado a partir del modelo pre-entrenado.

3.1 Google Colab

Se utiliza como espacio de trabajo Google Colab [33] debido a su interfaz intuitiva y fácil de usar, capacidad de compartir el trabajo simplemente compartiendo el enlace del proyecto, sin necesidad de creación de un ambiente Python con la correspondiente descarga de las librerías necesarias y debido a que para realizar las operaciones computacionales durante el entrenamiento se necesita una GPU lo suficientemente potente y Google Colab nos proporciona un equipo remoto para ello de manera gratuita.

3.2 GTRSB

Utilizaremos la base de datos proporcionada por INI (INSTITUT FÜR NEUROINFORMATIK) [34] formada por más de 50.000 imágenes de señales de tráfico alemanas de 43 tipos diferentes de una única señal por imagen sin ruido introducido en la base de datos.

3.2.1 Estructura de la base de datos

La estructura de la base de datos es la siguiente [34]:

- Cada directorio/carpeta representa un tipo de señal por el número del directorio.

- Dentro de cada directorio habrá hasta 30 imágenes diferentes de la misma señal de tráfico disponibles para el entrenamiento, validación o para el test.
- Cada directorio/carpeta contendrá un archivo CSV con anotaciones e información sobre la propia imagen.
- La imagen solo contiene la señal de tráfico sin su entorno o ruido que hiciese necesario establecer la posición de la señal, el archivo CSV solo indicará las medidas de dicha señal en la imagen.
- Cada imagen solo contiene una señal de tráfico.
- Las imágenes tienen un formato PPM (Portable PixMax).
- Las imágenes tienen un tamaño que varia entre 15x15 y 250x250 píxeles.
- Las imágenes no tienen porque ser cuadradas.

Cada fichero CSV contiene la siguiente información separadas por ";" [34]:

- Filename: Nombre/id de la imagen.
- Width: Anchura de la imagen.
- Height: Altura de la imagen.
- X1: Coordenada X del extremo izquierdo superior de la señal.
- Y1: Coordenada Y del extremo izquierdo superior de la señal.
- X2: Coordenada X del extremo derecho inferior de la señal.
- Y2: Coordenada Y del extremo derecho inferior de la señal.

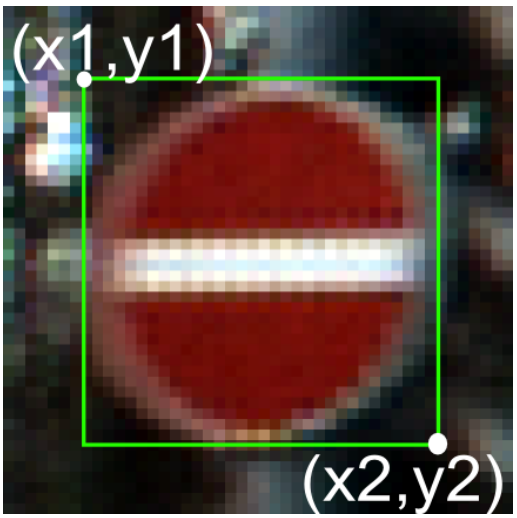


Figura 3.1: Posición de la señal guardada en el fichero CSV.

3.3 Creación del Dataset y el Dataloader [1]

3.3.1 Estructura del Dataset

Una vez descargadas las imágenes del dataset, se dispondrá a la identificación de cada una de ellas. Como ya se explicó, cada señal está contenida en un directorio con un ID determinado, por ejemplo, la señal de STOP está contenida en el directorio con ID igual a 14. Es por ello que habrá que relacionar cada señal que queramos identificar con su ID para poder guardar correctamente en cada directorio la señal que corresponde.

La disposición será con un peso de 70 por ciento para el entrenamiento, 20 por ciento para la validación y un 10 por ciento restante reservado para probar el sistema.

Es por ello que el árbol jerárquico del sistema será dispuesto de la siguiente manera:

- Data
  - Test
    - Señales 10 %
  - Train
    - Señales 70 %
  - Validation
    - Señales 20 %

Esta estructura nos permite una mayor facilidad para crear los *Dataloaders* de una manera atómica y controlando en todo momento en donde se ubican las entradas a la red otorgando una mayor robustez.

3.3.2 Transformaciones realizadas al Dataset

Como se explicó previamente se van a realizar una serie de transformaciones en las imágenes para que puedan ser tratadas por los modelos pre-entrenados de Torchvision como ya se mencionó en el apartado [2.2.2.3]. Mencionar que las imágenes deberán tener un tamaño de 224 píxeles de alto y ancho y la normalización será realizada con los valores que nos recomiendan en la documentación de Pytorch [32] que son los siguientes:

$$mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]$$

Se dividen las transformaciones realizadas en 2 partes:

- Entrenamiento y Validación:

  En el caso del entrenamiento nos interesa que las imágenes introducidas no sean iguales ni monótonas, por lo que gracias a la utilización de las funciones *Random* de *Torchvision.transform* nos permite transformar la imagen de una manera aleatoria en su tamaño, iluminación, girándolas un determinado ángulo. Todo esto es debido a las condiciones cambiantes que supondrían

la utilización del sistema en la vida real. También se hará uso para la validación debido a que sino los valores no serían reales ya que se haría uso de un dataset más sencillo recibiendo valores de pérdidas y precisión mejores.

Es por ello que realizaremos la siguiente secuencia de transformaciones con la función *Compose*:

$$train\_transforms = torchvision.transforms.compose([$$

torchvision.transforms.RandomResizedCrop(size=256),

torchvision.transforms.RandomRotation(degrees=15),

torchvision.transforms.RandomHorizontalFlip(),

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize(mean_nums, std_nums)])

Como podemos observar en la función se realizan las siguientes transformaciones [31]:

- *RandomResizedCrop*: Recorta una parte aleatoria de la imagen y cambia su tamaño al tamaño pasado por parámetro, siendo en este caso 256 píxeles.
- *RandomRotation*: Gira la imagen un ángulo determinado de manera aleatoria, siendo en este caso la máxima rotación de 15 grados.
- *RandomHorizontalFlip*: Giro horizontal aleatorio de la imagen.
- *ToTensor*: Convierte la imagen en un Tensor.
- *Normalize*: Normaliza la imagen con los valores que se introduzcan.

- Test:

  En este caso no necesitamos realizar diferentes modificaciones en las imágenes debido a que el único sentido de hacerlo es proporcionarle diferentes modificaciones para entrenar la red. Por lo que en este caso se le otorga valores fijos con la secuencia de transformaciones con la función *Compose*:

$$val\_test\_transforms = torchvision.transforms.compose([$$

torchvision.transforms.Resize(size=256),

torchvision.transforms.CenterCrop(size=224),

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize(mean_nums, std_nums)])

  Como podemos observar en la función se realizan las siguientes transformaciones [31]:

  - *Resize*: Cambia el tamaño por el valor introducido, siendo este caso 256 píxeles.
  - *CenterCrop*: Recorta la imagen dada al tamaño introducido, siendo este caso 224 píxeles.

- *ToTensor*: Convierte la imagen en un Tensor.
- *Normalize*: Normaliza la imagen con los valores que se introduzcan.

3.3.3 Aplicación de las transformaciones y agrupación con ImageFolder

Procedemos a la creación de un diccionario *dict* en Python con 3 componentes que representan la parte de entrenamiento, validación y test. Como se explicó con anterioridad haremos uso del módulo *ImageFolder* para indicar la dirección en la que se encuentran las imágenes y pasando como segundo parámetro las transformaciones realizadas sobre dichas imágenes.

A continuación se muestra un ejemplo con el *ImageFolder* de la sección de entrenamiento:

$$Image\_Folder = \{'train' : ImageFolder(f'data\_dir/train', train\_transforms)\}$$

Finalmente, se crea el *DataLoader* para la inyección en cada una de sus partes de entrenamiento de la red y prueba posterior. Para la creación del *DataLoader*, como se explicará posteriormente, se podrán introducir diferentes hiperparámetros como el *Batch* o *Num_workers* donde se explorarán diferentes valores y sus consecuencias en los resultados en la parte de experimentación.

A continuación se muestra un ejemplo con el *DataLoader* de la sección de entrenamiento:

$$Data\_Loaders = \{DataLoader(Image\_Folder, batch\_size = 4, shuffle = True, num\_workers = 4)\}$$

A partir de este momento se dispone de los datos de entrada bien organizados y preparados para su introducción en cada una de las fases.

3.4 Modelos con pesos pre-entrenados

Se facilita la creación de un sistema capaz de clasificar imágenes gracias al uso de un modelo con pesos pre-entrenados. Para su utilización en Python haremos uso del, ya explicado, módulo *Models* seguido del nombre de la red, siempre indicándoles que queremos hacer uso de los pesos ya pre-entrenados:

$$modelo = models.\text{“}Nombre\ del\ modelo\text{”}(pretrained = True)$$

A continuación se explican los diferentes modelos que se van a utilizar en el fase de experimentación.

3.4.1 Resnet [2]

Presentado en la competición ImageNet Large Scale Visual Recognition Challenge's (ILSVRC) por el grupo de Investigación de Microsoft Asia (MRSA) formado Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Su nombre proviene del concepto de Red Residual (Residual Net) y su motivación de la dificultad del entrenamiento de Redes Neuronales Convolucionales.

Los autores de Resnet se hacían la pregunta:

"¿Cuánto más capas tiene la red, más profunda, se obtienen mejores resultados?"

La respuesta es negativa pues se encontraron el problema de la desaparición/explosión de los gradientes que dificultan la convergencia desde el principio. Existe un problema de degradación en redes muy profundas dificultando su exactitud, siendo esta saturada. Es por ello que el hecho de añadir bloques de capas no es una opción recomendable.

Si consideramos *H(x)* como la salida de una serie de capas siendo x la entrada de dichas capas, si consideramos que una serie de capas no lineales pueden de manera asintomática aproximar funciones complicadas, podemos pensar que pueden de manera asintomática aproximar funciones residuales *H(x)* - x siempre asumiendo que tanto la salida como la entrada son de las mismas dimensiones. Por lo que en vez de obtener la salida *H(x)* buscamos obtener la función residual *F(x):= H(x)* - x por lo que la entrada al siguiente bloque vendría dado por *F(x)* + *X*. Ambos métodos llegan a la misma conclusión pero la facilidad de uso del residual hace decantarse por esta opción.

Esa es la principal diferencia entre la red residual y las demás redes neuronales, funcionando en la red residual de manera que si el valor de identidad es óptimo, el peso del bloque siguiente será igual a 0 evitando así el problema de la degradación.

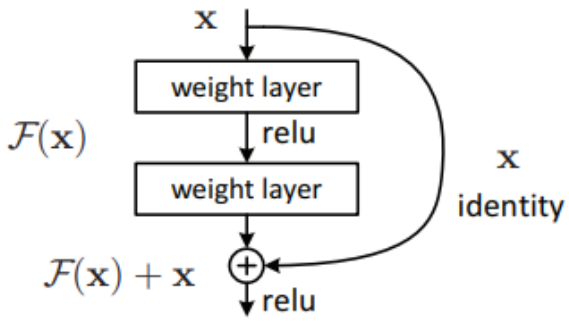


Figura 3.2: Bloque de la red ResNet [2].

El bloque mostrado en la figura 3.2 es definido como:

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

Siendo x la entrada e y la salida del bloque. $F(x, \{W_i\})$ representa la salida residual a aprender. Siendo las dimensiones de x y F las mismas.

3.4.1.1 Arquitectura del modelo Resnet

La arquitectura de ResNet está basada en las redes VGG [35] añadiéndole ese valor de entrada a la salida de los bloques/capas. La mayoría de capas convolucionales están formadas por filtros de dimensiones 3x3 que siguen 2 reglas de diseño:

- El número de filtros de las capas será igual al tamaño de salida (características).
- Si el tamaño de la salida (características) se reduce a la mitad el número de filtros se duplica para preservar la complejidad de la capa.

Las capas convolucionales tendrán un tamaño de paso de 2 por lo que conseguiremos reducir las muestras de entrada. Finalmente se convergerá en una sola capa de agrupamiento y una capa totalmente conectada formada por la capa *Softmax* como se puede apreciar en la figura 3.3.

Cabe mencionar la arquitectura de cuello de botella utilizada por los investigadores, en la que introducen antes y después de la capa de convolución de 3x3 una capa de convolución 1x1 que reduce y después restaura las dimensiones, dejando un cuello de botella a la capa de convolución de 3x3 con salidas y entradas de menos dimensiones.

ResNets más profundos sin cuello de botella ganan en precisión pero no son tan ligeros, en cuanto a hardware necesario, como las arquitecturas ResNets con cuello de botella. Entonces, el uso de diseños de cuello de botella se debe principalmente a consideraciones prácticas.

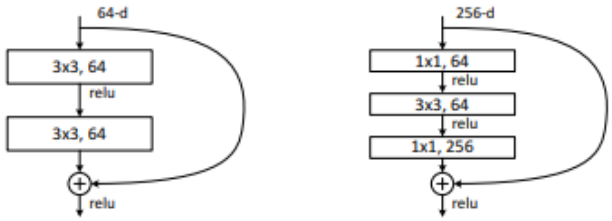


Figura 3.3: Figura de la izquierda: Resnet sin cuello de botella. Figura de la derecha: Resnet con cuello de botella [2].

3.4.1.2 Implementación ideal del modelo Resnet

Es importante tener en cuenta que este modelo ha sido creado y entrenado para trabajar con la base de datos de imágenes de ImageNet y sus pesos han sido entrenados con diferentes transformaciones en las imágenes y determinados valores de hiperparámetros por lo que habrá que adaptar las imágenes de entrada a los requisitos impuestos por la red para que los resultados sean aceptables.

La imagen se cambia de tamaño a un rango de [256,480] para un aumento de escala. Recibe un recorte de 224x224 y diferentes modificaciones aleatorias en la imagen para mejorar el entrenamiento, como giros con diferentes ángulos y desplazamientos. Después de cada convolución se realiza la normalización del *Batch*. Utiliza el optimizador SGD con un *learning rate* de 0.1 como valor inicial, una caída de peso *weight decay* de 0.0001 y un *momentum* de 0.9.

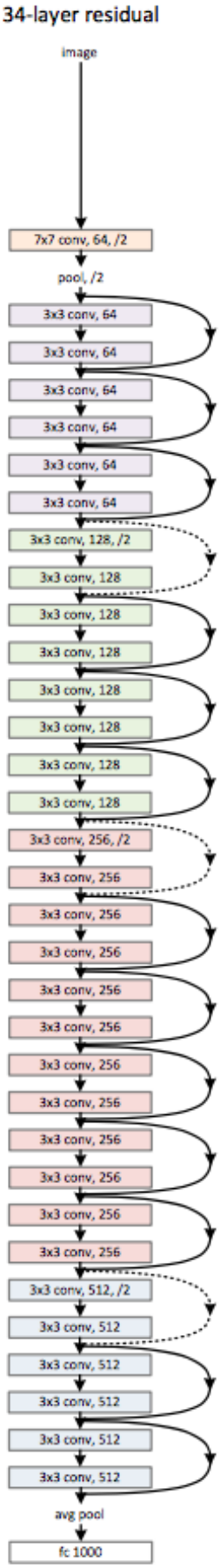


Figura 3.4: Estructura de ResNet34 [2].

3.4.2 AlexNet [3]

Presentado en 2012 en la competición LSVRC (Large Scale Visual Recognition Challenge) por su creadores Alex Krizhevsky, en colaboración con Ilya Sutskever y Geoffrey E. Hinton. En los orígenes del *Machine Learning* las principales bases de datos de imágenes etiquetadas, como NORB o CIFAR, utilizadas para el entrenamiento tenían una capacidad de miles de imágenes que no suponían un problema de rendimiento para la redes neuronales. El problema llegaba en el momento que aparecieron bases de datos de imágenes etiquetadas como ImageNet con más de 15 millones de imágenes y con una resolución alta de imagen que dificultaba el trabajo de las redes neuronales convolucionales.

La motivación era obtener buenos resultados en entrenamientos extensos en un tiempo que no fuera excesivo, y fue gracias a la implementación del multiproceso GPU altamente optimizado en las tareas de convolución de la red.

El *dataset* utilizado es el ya mencionado ImageNet, que contiene más de 15 millones de imágenes de más de 22,000 diferentes tipos de clases. Mientras ImageNet tiene imágenes de diferentes tipos de resolución, AlexNet necesita entradas de las mismas dimensiones por lo que las imágenes serán re-escaladas y recortadas a una resolución común de 256x256.

3.4.2.1 Arquitectura del modelo AlexNet

La arquitectura de la red está formada por 8 capas, 5 capas convolucionales con componentes que aportan no linealidad y 3 capas completamente conectadas.

A continuación se explican las características de cada una de ellas:

- ReLU no linealidad:

  En el momento de creación de la red, era común utilizar como componente que aportase la característica de no linealidad la función de tangente hiperbólica *tanh*:

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

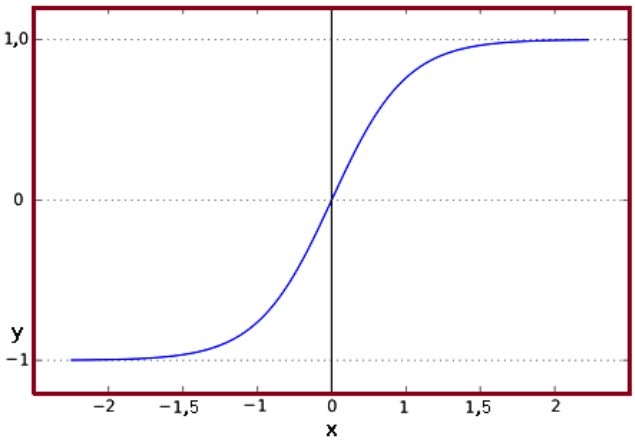


Figura 3.5: Función tangente hiperbólica [14].

Pero los creadores se decantan por el uso de la función ReLU debido a que en términos de tiempo de entrenamiento de la red, los métodos saturadores de no linealidad como la función *tanh* son mucho más lentos que los métodos no saturadores de no linealidad como ReLU, por lo que en busca de la optimización del tiempo de entrenamiento se emplea ReLU para aportar la no linealidad.

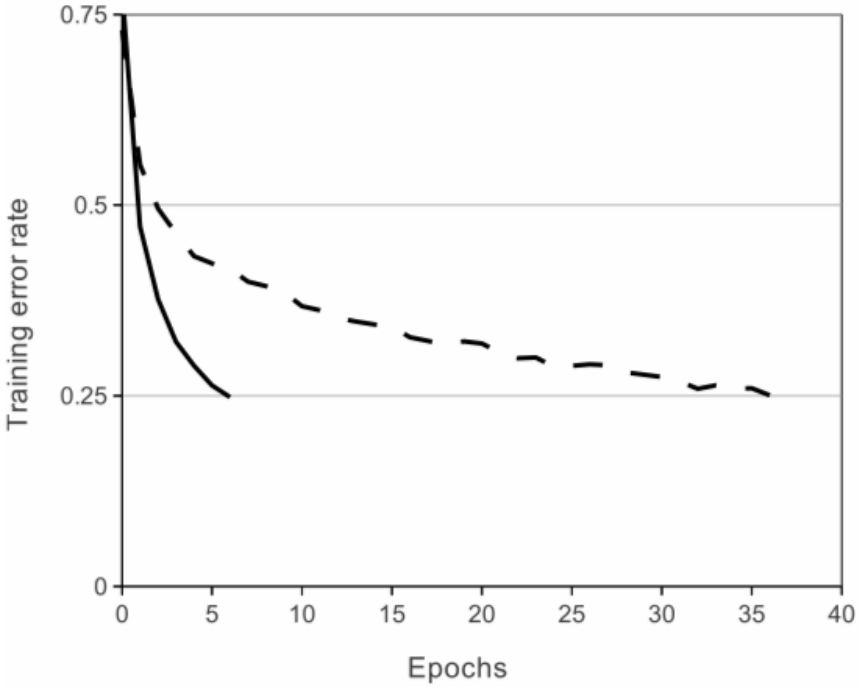


Figura 3.6: Tasa de Error entre la utilización de la función ReLU (línea continua) y tanh (línea discontinua) [3].

Como se observa en la figura 3.6 si optamos por el uso de la función *tanh* para aportar la no linealidad es necesario llegar hasta la *epoch* 35 para llegar a la misma tasa de error de la *epoch* 5 si optamos por el uso de la función ReLU para aportar la no linealidad.

Es por ello que se puede concluir con que la función ReLU es hasta 6 veces más rápida que la función *tanh*.

- Entrenamiento con múltiples GPU:

  Actualmente existen GPU con memorias lo suficientemente grandes para llevar a cabo la tarea de entrenamiento por si solas. Eso no sucedía en el momento de creación de la red, siendo la memoria de las GPUs más potentes, accesibles en el mercado por un precio razonable, de 3GB. Esto suponía un problema debido a la inmensa cantidad de imágenes de los *dataset* ya mencionados.

  Es por ello que se propone la utilización de 2 GPUs de 3 GB cada una para poder repartir a la mitad el trabajo realizado por cada una de ellas y por lo tanto reducir el tiempo de entrenamiento.

  El método utilizado por los creadores es el dividir la cantidad de neuronas para cada GPU permitiendo a cada GPU solo comunicarse con determinadas capas de convolución reduciendo

así la tasa de error comparado con la utilización de una sola GPU.

- Agrupación superpuesta:

  Las capas de agrupación tradicionales agrupan las neuronas vecinas de un mismo mapa de características sin usar la superposición. Los creadores observaron que el hecho de introducir la superposición reducía en un 0.4 por ciento.

  La agrupación superpuesta se basa en la utilización de un salto menor que la dimensión de la salida de la agrupación. En el apartado de la capa de agrupación observábamos como mediante un salto de 2 posiciones en una matriz de 4x4 se obtenían una reducción de la dimensión de la matriz a 2x2 utilizando el método que se quisiese. Si se opta por utilizar un salto de 1 en el que la matriz de salida sea mayor que el filtro utilizado produce la superposición en la que utilizaremos los mismos píxeles en varias ventanas como se muestra en la figura 3.7 que ilustra como de un filtro de 2x2 se obtiene una matriz 3x3.

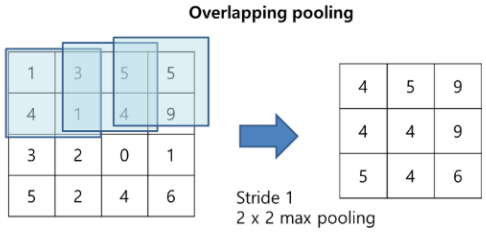


Figura 3.7: Agrupación superpuesta [15].

La arquitectura global de AlexNet como se explica con anterioridad está formada por 8 capas con pesos, siendo las 5 primeras capas convolucionales y las 3 últimas capas completamente conectadas. La salida de la última capa totalmente conectada alimentará una capa *softmax*.

Los *kernel* de las capas convolucionales segunda, cuarta y quinta están conectados solo a las salidas de la capa anterior que residen en la misma GPU. Los kernel de la tercera capa están conectados a ambas salidas de la segunda capa de convolución. Las capas de agrupación se aplicarán a la salida de la primera, segunda y quinta capa de convolución. La capa ReLU es aplicada a la salida de cada capa incluyendo la capa completamente conectada. Finalmente, todas las neuronas están conectadas, como su nombre indica, en la capa *fully-connected*.

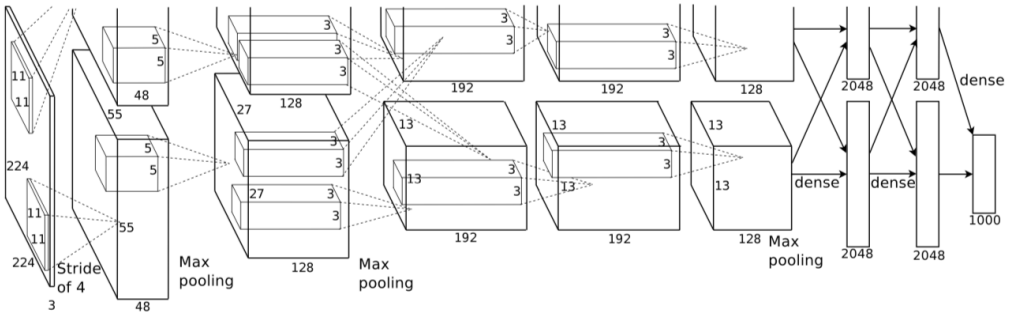


Figura 3.8: AlexNet [15].

3.4.3 Inception [4]

Creado por los investigadores de Google, Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe y Jonathon Shlens en colaboración con Zbigniew Wojna. Actualmente funcionando en su versión más actualizada, la versión 3. En su primera versión (figura 3.9) ya introdujo una de sus principales características: el uso de capas convolucionales de diferentes dimensiones debido a que la información de las imágenes no siempre es del mismo tamaño, ya sea por ejemplo una imagen con mucho ruido al rededor del objeto que deseamos clasificar o siendo el objeto mismo el que ocupa toda la dimensión de la imagen. Es por ello que a mayor es la información, mayor deberá ser el filtro y a menor información menor deberá ser el filtro.

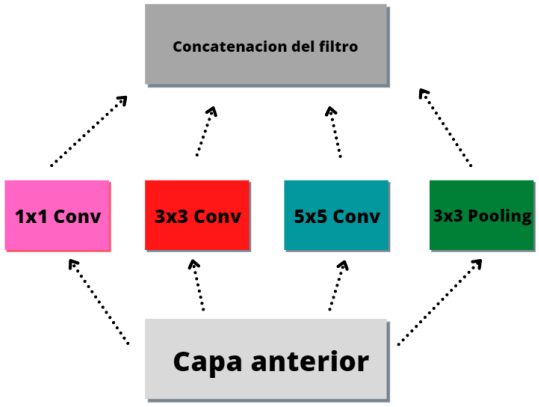


Figura 3.9: Bloque inception. Versión 1.

Posteriormente, los investigadores reducirían el número de parámetros de entrada para así reducir la tarea computacional del entrenamiento introduciendo una nueva estructura. Se consiguió reducir la dimensión de la entrada mediante la reducción de los canales introduciendo previamente a las capas de convolución 3x3 y 5x5 una capa de convolución 1x1. También posteriormente a la capa de 3x3 de agrupación se añadiría una capa de convolución 1x1 adicional. Se observan las modificaciones en la figura 3.10.

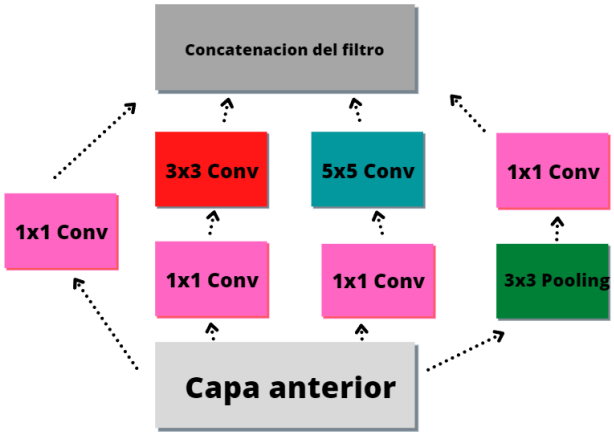


Figura 3.10: Bloque inception. Versión 2.

El hecho de añadir las capas 1x1 permiten reducir el tamaño de profundidad de la entrada previamente a la convolución ya sea de 3x3 y 5x5 permitiéndonos realizar la misma tarea pero con una carga computacional mucho menor.

Llegamos a la versión 3 del modelo Inception (Figura 3.11) en el que observamos dos modificaciones:

- La factorización a convoluciones más pequeñas:

  Mediante la modificación del bloque de convoluciones 5x5 por dos bloques de convolución 3x3. Esto es debido a que se observa que la convolución de 5x5 funciona como una capa *fully-connected* en tamaño reducido y en la clasificación de imágenes se debe aprovechar la invarianza de traducción por lo que se sustituirá la componente de estar completamente conectados por una arquitectura de dos capas convolucionales 3x3 en cascada.

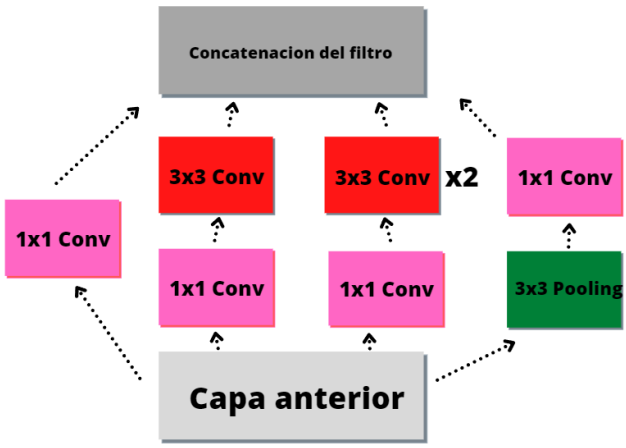


Figura 3.11: Bloque inception. Versión 3.

- La factorización espacial con convoluciones asimétricas:

  No se utilizan convoluciones mayores de 3x3 debido a que estas siempre podrán ser factorizadas pero también nace la pregunta si es posible factorizar las capas 3x3 en capas de 2x2 incluso 1x1. La respuesta es que se obtiene mejores resultados no solo reduciendo la dimensión del filtro sino mediante el uso de convoluciones asimétricas seguidas de nx1 y 1xn siendo n la dimensión de la capa convolucional a replicar nxn. Este hecho permite obtener una mejora del 33 por ciento en cuanto a esfuerzo computacional respecto al 11 por ciento que supondría sustituir la capa convolucional de 3x3 por dos capas convolucionales en cascada de 2x2. Mencionar que los creadores experimentan mayor tasa de error cuando aplican esta factorización en las capas iniciales de la red por lo que su uso sería en las capas intermedias.

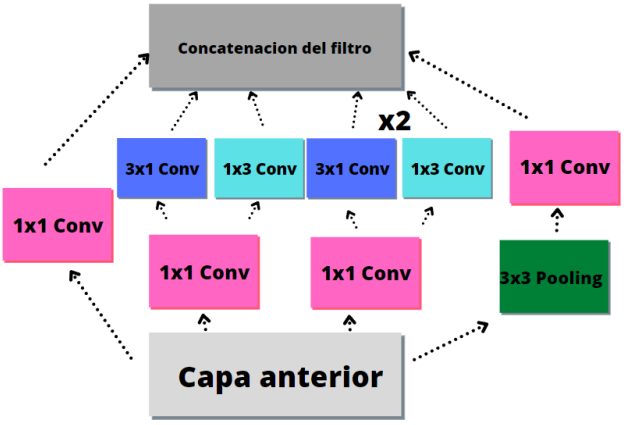


Figura 3.12: Bloque inception. Uso de convolución asímetrica.

3.4.3.1 Arquitectura del modelo Inception

La arquitectura global del modelo Inception v3 está formada por seis capas de convolución de 3x3 con diferentes valores de salto, una capa de agrupación de 3x3 y otra de 8x8, 3 bloques Inception como en la figura 3.11, 5 bloques Inception como en la figura 3.12 con valor n=7, 2 bloques como en la figura 3.12 con valor n=3 y finalmente las capas completamente conectadas y la capa softmax.

Observamos la figura 3.13 los valores de entrada de cada capa y sus valores de salto:

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception		35×35×288
5×Inception		17×17×768
2×Inception		8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Figura 3.13: Tabla con la arquitectura del modelo Inception [4].

3.5 Entrenamiento y validación [1]

Para realizar un correcto entrenamiento, se han definido dos funciones dentro de la rutina de entrenamiento, la primera función será empleada para el aprendizaje de la red y la actualización de sus pesos mediante el cálculo de los gradientes y función de pérdida y la segunda función será empleada para la validación que nos permite mejorar la precisión de la red indirectamente, actualizando los valores de los hiperparámetros y al final evaluar, como su nombre indica, la efectividad de la red. Estableceremos un ratio de 70 % para el dataset del entrenamiento y un 20 % para el dataset de validación, mientras el resto del porcentaje lo utilizaremos para probar a la red con datos no vistos con anterioridad. Sin embargo, es importante destacar que el ratio deberá variar dependiendo del número de hiperparámetros que introduzcamos en la red debido a que a mayor número de hiperparámetros a configurar mayor deberá ser el tamaño y tiempo de validación.

3.5.1 Entrenamiento

Se define la función para realizar un paso de entrenamiento, con nombre *train_epoch*, como:

$$train\_epoch(model, data\_loader, loss\_fn, optimizer, device, scheduler, n\_examples)$$

Sus parámetros representan las siguientes funciones:

- model: Representa el modelo de pesos pre-entrenados, ya sea ResNet, AlexNet o Inception.
- data_loader: Representa la carga de los dataset configurados previamente con los ratios de cantidad de imágenes ya mencionados. En este caso será el dataset de entrenamiento el que introduciremos en esta función.
- loss_fn: Representa la función de pérdidas. En este caso emplearemos la entropía cruzada ya explicada previamente.
- optimizer: Representa el algoritmo de optimización. La efectividad de dichos algoritmos será expuesta en la fase de experimentación.
- device: Representa al motor de procesado, es decir, ya sea la tarjeta gráfica GPU o el procesador CPU dependiendo de lo seleccionado por el sistema en la condición establecida con el módulo *Torch.cuda*. En este caso la variable a introducir será siempre la misma.
- scheduler: Representa el módulo *Lr_scheduler* que nos permite ajustar la tasa de aprendizaje. Se utilizarán varios métodos, expuestos previamente.
- n_examples: Representa el número de imágenes de cada tipo diferente de señal de tráfico que utilizaremos en el entrenamiento. Se utilizará para realizar bucles con el tamaño correcto y para obtener la media en los resultados de los entrenamientos y validación.

Una vez en la función se comunica al modelo que se procede a entrenarlo mediante *model.train()* debido a que el modelo funcionará de manera diferente en determinadas capas si estamos entrenando o evaluando la red. Todos los modelos tendrán un atributo que determinará si está entrenando o evaluando por lo que será importante establecerlo previamente.

A continuación se muestra el cuerpo de la función:

```
losses = []
correct_predictions = 0
model = model.train()
for inputs, labels in data_loader:
  inputs = inputs.to(device)
  labels = labels.to(device)

  outputs = model(inputs)

  _, preds = torch.max(outputs, dim=1)

  loss = loss_fn(outputs, labels)

  correct_predictions += torch.sum(preds == labels)

  losses.append(loss.item())

  loss.backward()
  optimizer.step()
  optimizer.zero_grad()

scheduler.step()

return correct_predictions.double() / n_examples, np.mean(losses)
```

Figura 3.14: Función de entrenamiento.

Como se puede apreciar en la figura 3.14 se procede a realizar el bucle de entrenamiento en el que recorremos el *data_loader* mediante las variables *inputs* y *labels*. El *data_loader* nos otorga un número, establecido por el programador, de lote de imágenes *batch* y los cargamos al modelo junto a los *labels* que será un array/tensor de tamaño igual al *batch* con valores de 0 a x siendo x el número de tipos de señales de tráfico menos la unidad.

A la salida del modelo obtenemos unos resultados (*outputs*) determinados que equivalen a la predicción de cada imagen realizada por el modelo siendo el de mayor valor el que el modelo estima que tiene un mayor porcentaje de ser la señal de tráfico de la imagen dada. Mediante el módulo *torch.max* se obtiene el índice del valor máximo del tensor de cada imagen por lo que la variable *preds* equivaldrá a un array del mismo tamaño que *labels* cuyos valores determinan la señal que la red ha predicho. Si la red no se ha equivocado entonces el valor de *preds* será igual que *labels* pero como observaremos más a delante habrá ciertos valores de ambos arrays que diferirán entre ellos. Se aprecia el funcionamiento de *torch.max*, para una red capaz de diferenciar 3 tipos de señales, en la

figura 3.15 obteniendo los índices cuyos valores han resultado mayores:

```
Outputs: tensor([[-0.9284,  0.1817,  1.4765],
        [-0.9582,  0.6803,  0.5736],
        [ 1.8122, -0.3480, -1.3331],
        [-1.3766,  1.6297,  0.6005]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
Prediccion: tensor([2, 1, 0, 1], device='cuda:0')
```

Figura 3.15: Funcionamiento de *Torch.Max*.

La función pérdida se calcula con la entropía cruzada pasando como parámetros la salida del modelo y la variable *labels*.

Una vez calculada la pérdida se actualiza la variable *correct_predictions* que equivale al número de valores iguales entre las predicciones realizadas por el modelo y el valor real *labels* de las imágenes. En la figura 3.16 se muestra un ejemplo del funcionamiento de las predicciones:

```
Labels: tensor([0, 0, 2, 2], device='cuda:0')
Prediccion: tensor([0, 0, 2, 2], device='cuda:0')
Nº de prediccion correcta: 4
Labels: tensor([2, 1, 0, 1], device='cuda:0')
Prediccion: tensor([2, 2, 0, 0], device='cuda:0')
Nº de prediccion correcta: 6
```

Figura 3.16: Número de predicciones correctas.

Como se puede apreciar en la figura se muestran primero el valor/etiqueta del lote de imágenes, siendo en este caso el valor de *Batch=4*. Posteriormente se muestra el valor de las predicciones siendo en este caso 100 % correctas. Es por ello que el número de predicciones se incrementa en 4 debido a que en el primer paso por el bucle el modelo ha sido capaz de predecir correctamente las 4 imágenes. Sin embargo, no es el caso en el segundo paso por el bucle, fallando en la predicción del modelo en la posición de los arrays 1 y 3. Es por ello que en este caso el modelo solo ha sido capaz de predecir la mitad de las imágenes correctamente por lo que el número de predicciones correctas solo se incrementará en 2.

También se obtienen los valores de la media de pérdida del *Batch* mediante la función *loss.item()*. Dicho valor se almacenará en un array del que posteriormente se calculará la media haciendo uso de la función, del módulo NumPy como np, *np.mean*.

Finalmente, se calculan los gradientes de la función pérdida y realizamos el paso en el optimizador y del planificador del learning rate. Destacar la importancia de dejar a cero los gradientes mediante *optimizer.zero_grad()*, para no acumular los gradientes entre épocas del bucle de entrenamiento. También mencionar la necesidad de calcular la retropropagación de la función de pérdidas previamente al paso del optimizador debido a que dentro de la función *Step* utiliza los valores calculados en la función *Backward*.

Cuando finalice el entrenamiento retornaremos de la función el porcentaje de aciertos del modelo y la media de pérdidas por motivos meramente estadísticos e ilustrativos del buen funcionamiento o no de la red.

3.5.2 Validación

Se define la función para realizar un paso de validación, con nombre *eval_model*, como:

$$eval\_model(model, data\_loader, loss\_fn, device, n\_examples)$$

Sus parámetros representan las siguientes funciones:

- model: Representa el modelo de pesos pre-entrenados, ya sea ResNet, AlexNet o Inception.
- data_loader: Representa la carga de los dataset configurados previamente con los ratios de cantidad de imágenes ya mencionados. En este caso será el dataset de validación el que introduciremos en esta función.
- loss_fn: Representa la función de pérdidas. En este caso emplearemos la entropía cruzada ya explicada previamente.
- device: Representa al motor de procesado, es decir, ya sea la tarjeta gráfica GPU o el procesador CPU dependiendo de lo seleccionado por el sistema en la condición establecida con el módulo *Torch.cuda*. En este caso la variable a introducir será siempre la misma.
- n_examples: Representa el número de imágenes de cada tipo diferente de señal de tráfico que utilizaremos en la validación. Se utilizará para realizar bucles con el tamaño correcto y para obtener la media en los resultados de los entrenamientos y validación.

Una vez en la función, al igual que sucedía en el modelo de entrenamiento, se debe comunicar al modelo que se desea realizar la validación mediante *model.eval()*.

A continuación se muestra el cuerpo de la función:

```
def eval_model(model, data_loader, loss_fn, device, n_examples):
  model = model.eval()

  losses = []
  correct_predictions = 0

  with torch.no_grad():
    for inputs, labels in data_loader:
      inputs = inputs.to(device)
      labels = labels.to(device)

      outputs = model(inputs)

      _, preds = torch.max(outputs, dim=1)

      loss = loss_fn(outputs, labels)

      correct_predictions += torch.sum(preds == labels)
      losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

Figura 3.17: Función de validación.

Se puede observar como la función de validación es prácticamente idéntica a la de entrenamiento teniendo de diferencias la mencionada respecto al *model.eval()* y que la fase de validación no contiene ningún cálculo de gradiente, paso de optimización o del planificador del learning rate ni la retropropagación de la función pérdida.

3.5.3 Bucle de entrenamiento y validación

Se define el bucle de entrenamiento y validación, con nombre *train_model*, como:

$$train\_model(model, data\_loaders, dataset\_sizes, device, n\_epochs)$$

Sus parámetros representan las siguientes funciones:

- model: Representa el modelo de pesos pre-entrenados, ya sea ResNet, AlexNet o Inception.
- data_loader: Representa la carga de los dataset configurados previamente con los ratios de cantidad de imágenes ya mencionados. En este caso será el diccionario/lista que incorpora tanto la de entrenamiento como la de validación o de test.
- dataset_sizes: Representa la variable *n_examples* del modelo de entrenamiento y validación. En este caso contiene el número de imágenes de cada clase.
- device: Representa al motor de procesado, es decir, ya sea la tarjeta gráfica GPU o el procesador CPU dependiendo de lo seleccionado por el sistema en la condición establecida con el módulo *Torch.cuda*. En este caso la variable a introducir será siempre la misma.
- n_epochs: Representa el número epochs de entrenamiento. Cuanto mayor sea este número mejor serán los resultados, siempre que no se produzca *overfitting*, pero más tiempo durará el proceso de computación.

A continuación se muestra el cuerpo de la función:

```python
def train_model(model, data_loaders, dataset_sizes, device, n_epochs=3):
  optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
  scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
  loss_fn = nn.CrossEntropyLoss().to(device)

  history = defaultdict(list)
  best_accuracy = 0

  for epoch in range(n_epochs):

    print(f'Epoch {epoch + 1}/{n_epochs}')
    print('-' * 10)

    train_acc, train_loss = train_epoch(model, data_loaders['train'],loss_fn,optimizer, device, scheduler, dataset_sizes['train'])

    print(f'Train loss {train_loss} accuracy {train_acc}')

    val_acc, val_loss = eval_model(model, data_loaders['val'],loss_fn,device,dataset_sizes['val'])

    print(f'Val   loss {val_loss} accuracy {val_acc}')
    print()

    history['train_acc'].append(train_acc)
    history['train_loss'].append(train_loss)
    history['val_acc'].append(val_acc)
    history['val_loss'].append(val_loss)

    if val_acc > best_accuracy:
      torch.save(model.state_dict(), 'best_model_state.bin')
      best_accuracy = val_acc

  print(f'Best val accuracy: {best_accuracy}')

  model.load_state_dict(torch.load('best_model_state.bin'))

  return model, history
```

Figura 3.18: Función de bucle de entrenamiento y validación.

Se puede apreciar en la figura 3.18 que se declaran previamente al entrenamiento el algoritmo de optimización, el método de planificación de learning rate y la función de pérdidas que será siempre la de entropía cruzada.

Se comienza el bucle de un número de épocas pasado por parámetro, obteniendo los valores del modelo de entrenamiento y posteriormente los valores de validación, ambos ya explicados en anteriores secciones.

En un diccionario con nombre *history* se procederá a guardar los valores obtenidos en cada época del bucle para realizar posteriormente el análisis de los resultados. Se compararán los valores de validación con una variable *best_accuracy* inicializada a 0 que se actualizará cuando se obtengan porcentajes mayores a los previos. Posteriormente guardaremos los pesos utilizados en dicha validación debido a que representan los mejores resultados obtenidos durante el entrenamiento y validación.

3.5.4 Gráficas de evolución del entrenamiento

Además, se ha diseñado otra función para mostrar una serie de gráficos con los valores recogidos en la variable *History* a lo largo de las épocas para observar la mejoría o no mejoría del sistema en la fase de experimentación.

Haremos uso de la función siguiente para mostrar los valores de pérdidas y precisión de la red a lo largo de las épocas tanto en la fase de entrenamiento como en la de validación.

```python
def estadisticas_plot(history):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

  ax1.plot(history['train_loss'], label='train loss')
  ax1.plot(history['val_loss'], label='validation loss')

  ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax1.set_ylim([-0.05, 1.05])
  ax1.legend()
  ax1.set_ylabel('Loss')
  ax1.set_xlabel('Epoch')

  ax2.plot(history['train_acc'], label='train accuracy')
  ax2.plot(history['val_acc'], label='validation accuracy')

  ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax2.set_ylim([-0.05, 1.05])
  ax2.legend()

  ax2.set_ylabel('Accuracy')
  ax2.set_xlabel('Epoch')

  fig.suptitle('Training history')
```

Figura 3.19: Función para la obtención de estadísticas.

A la izquierda se disponen las pérdidas tanto de la fase de entrenamiento como de validación y a la derecha se dispone la precisión en la predicción tanto en la fase de entrenamiento como en la fase de validación.

A modo de ejemplo, en la figura 3.23 se visualiza el resultado de una simulación de un entrenamiento de la red de duración 3 épocas con tres tipos de señales.

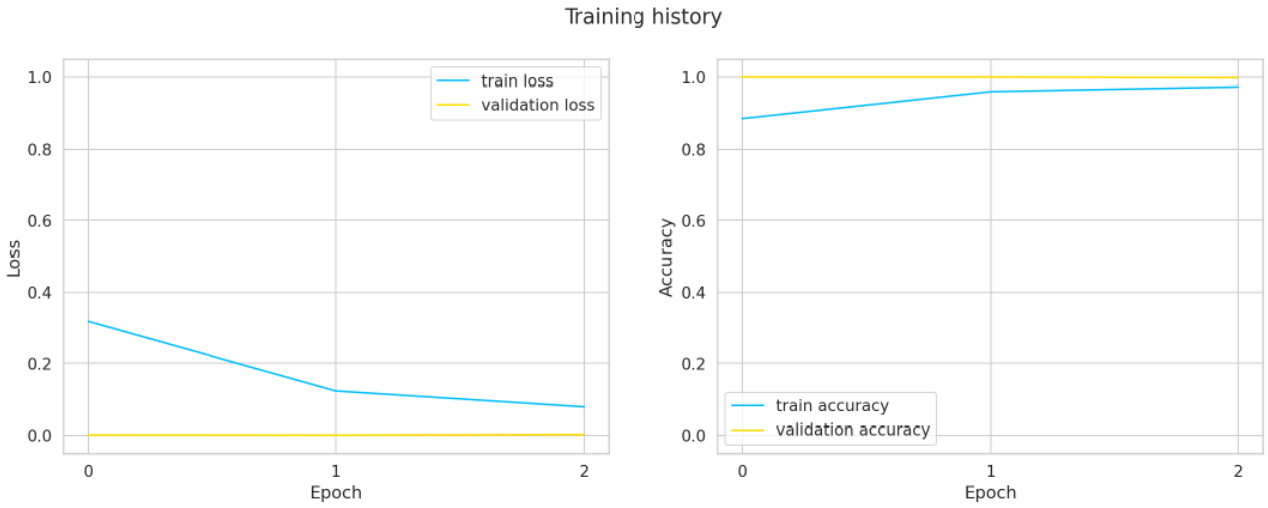


Figura 3.20: Tablas de entrenamiento de ejemplo.

3.6 Test de la red entrenada [1]

Se va a definir una función para probar la calidad de la red mediante la evaluación del 10 % de las imágenes del total. También se hace uso de una función que permita mostrar en una gráfica, mediante el uso del módulo Matplotlib, los valores guardados en la variable *history* de la pérdida y precisión en el entrenamiento y validación. También se realizan estudios estadísticos del acierto en las pruebas realizadas en la red.

3.6.1 Test

Se procede a utilizar el *dataloader* reservado para el test siendo un porcentaje bastante más pequeño que el entrenamiento por lo que se realizará en un menor periodo de tiempo. La función de test será muy parecida a la función de validación pues habrá que comunicar a la red que estamos realizando la evaluación mediante *model.eval* y cargar el *dataloader* en el bucle para finalmente comparar las etiquetas introducidas al modelo con las predicciones obtenidas del mismo para observar la precisión de la red.

```
def test_model(model):
  correct_predictions=0
  model.eval()
  with torch.no_grad():
    for inputs, labels in data_loaders['test']:
      inputs = inputs.to(device)
      labels = labels.to(device)

      outputs = model(inputs)

      _, preds = torch.max(outputs, dim=1)

      correct_predictions += torch.sum(preds == labels)
  return correct_predictions.double() / dataset_sizes['test']

predicciones=test_model(resnet_model)
print(f"Accuracy: {predicciones}")
```

Figura 3.21: Función de test.

También se define una segunda función para observar las predicciones con la imagen a la que hace referencia para que los resultados sean más visuales y se puedan observar los resultados de forma cualitativa.

```
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_handeled = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(data_loaders['test']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_handeled += 1
                ax = plt.subplot(num_images//2, 2, images_handeled)
                ax.axis('off')
                ax.set_title('predicted: {}'.format(class_names[preds[j]]))
                imshow(inputs.cpu().data[j])

                if images_handeled == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)

visualize_model(resnet_model)
plt.show()
```

Figura 3.22: Función de visualización de resultados.

El sistema de evaluación es el mismo que en la función anterior, añadiendo el uso del módulo Matplotlib para mostrar junto a la predicción realizada la señal a la que hace referencia. En este caso se añade un bucle adicional que posicionará las imágenes en un *plot* sucedido de la predicción. La predicción es obtenida una a una de la variable *preds* y gracias al establecimiento previo de la variable *class_names* con los nombres de las señales de tráfico en orden según entrada en el *dataloader* seremos capaces de mostrar el nombre real de la predicción. En la variable *num_images* se establece el número de imágenes que se desean mostrar por lo que establecemos una condición que, en el caso de que se cumpla, finalice la función siendo el tiempo de ejecución mínimo.

Un ejemplo de la disposición de los resultados es la mostrada en la figura 3.21 que muestra las predicciones de seis señales de tráfico:

Figura 3.23: Ejemplo de salida de la función visual de pruebas.

Capítulo 4

Experimentación del sistema

Ninguna cantidad de experimentación puede probar definitivamente que tengo razón; pero un solo experimento puede probar que estoy equivocado.

Albert Einstein

En este capítulo se realizarán modificaciones en la configuración del sistema, ya sea en el valor de los hiperparámetros, optimizadores o modelos CNN de pesos pre-entrenados. Previamente al estudio se definirá el sistema base con el que compararemos los resultados obtenidos y medio por el cual sacaremos las conclusiones sobre el rendimiento del sistema en diferentes puntos.

Se mostrarán tablas y cálculos relacionados con el comportamiento y efectividad del sistema en cada fase o parámetro modificado para realizar con argumentos las conclusiones finales.

4.1 Sistema base

Como sistema base se define un sistema capaz de clasificar por 43 tipos diferentes de señales de tráfico con un ID/*label* para cada tipo. En la Tabla 4.1 se muestra los diferentes tipos de señales con su ID/*label* y la cantidad de imágenes de cada tipo.

El sistema constará de las siguientes partes:

- *DataLoaders* formados por un valor de *Batch* de cuatro imágenes mezcladas mediante la función *Shuffle*.
- Modelo con pesos pre-entrenados ResNet en su versión de 34 capas.
- Algoritmo de optimización SGD con un valor de learning rate de 0.001 y con un valor de momentum de 0.9.
- Planificador de learning rate con un valor de salto de 7 y un valor de gamma de 0.1.
- Función de pérdida de entropía cruzada.

- El número de *epochs* será de diez.

ID/label	Señal	Cantidad
0	Límite velocidad 20Km/h	210
1	Límite velocidad 30Km/h	2220
2	Límite velocidad 50Km/h	2250
3	Límite velocidad 60Km/h	1410
4	Límite velocidad 70Km/h	1980
5	Límite velocidad 80Km/h	1860
6	Límite velocidad 90Km/h	420
7	Límite velocidad 100Km/h	1440
8	Límite velocidad 120Km/h	1410
9	Prohibido adelantar	1470
10	Camiones prohibido adelantar	2010
11	Prioridad frente al resto de carriles	1320
12	Prioridad	2100
13	Ceda el paso	2160
14	Stop	780
15	Circulación prohibida	630
16	Prohibido camiones	420
17	Prohibido	1110
18	Peligro	1200
19	Curva izquierdas peligrosa	210
20	Curva derechas peligrosa	360
21	Varias curvas peligrosa	330
22	Perfil irregular	390
23	Peligro deslizante	510
24	Estrechamiento hacia la izquierda	270
25	Obra	1500
26	Semáforo	600
27	Peatones	240
28	Tramo escolar	540
29	Peligro ciclos	270
30	Peligro nieve	450
31	Peligro animales silvestres	780
32	Fin de prohibiciones	240
33	Obligación giro derecha	689
34	Obligación giro izquierda	420
35	Obligación seguir adelante	1200
36	Obligación girar derecha o seguir delante	390
37	Obligación girar izquierda o seguir delante	210
38	Obligación seguir por la derecha	2070
39	Obligación seguir por la izquierda	300
40	Rotonda	360
41	Fin prohibición de adelantar	240
42	Fin prohibición de adelantar para camiones	240
Total		39 209

Tabla 4.1: Distribución del tipo de señales.

Los resultados obtenidos en el entrenamiento son los siguientes:

Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
1	52,28 %	84,39 %	37,43 %	88,66 %
2	41,91 %	87,28 %	32,21 %	89,74 %
3	36,85 %	88,63 %	29,03 %	90,73 %
4	33,53 %	89,31 %	29,61 %	90,95 %
5	31,39 %	89,93 %	25,86 %	91,59 %
6	29,15 %	90,86 %	25,30 %	91,80 %
7	28,69 %	90,97 %	25,03 %	91,76 %
8	22,40 %	92,72 %	**17,01 %**	**94,50 %**
9	20,26 %	93,45 %	18,97 %	94,12 %
10	**19,16 %**	**93,73 %**	18,02 %	94,08 %

Tabla 4.2: Resultados del sistema base.

Se percibe un patrón en el que tanto la pérdida como la precisión son mejores en la validación y eso es debido a que la cantidad de imágenes tomadas para la validación en relación con el entrenamiento es mucho menor. Dicho fenómeno se apreciará la mayoría de las veces en las primeras fases del entrenamiento. Sin embargo, no significa que los resultados sean malos, debido a que puede ser por factores como la facilidad de clasificación de las imágenes o el número de capas de regularización y/o la característica *dropout* que no está habilitada durante la validación.

Cabe mencionar que la mejora más grande se produce entre la época 7 y 8. En parte es debido al planificador del learning rate que actúa en dicho momento disminuyéndolo cada 7 épocas y permitiendo un mejor resultado en la estimación de los pesos.

Los resultados son representados en la siguiente tabla:

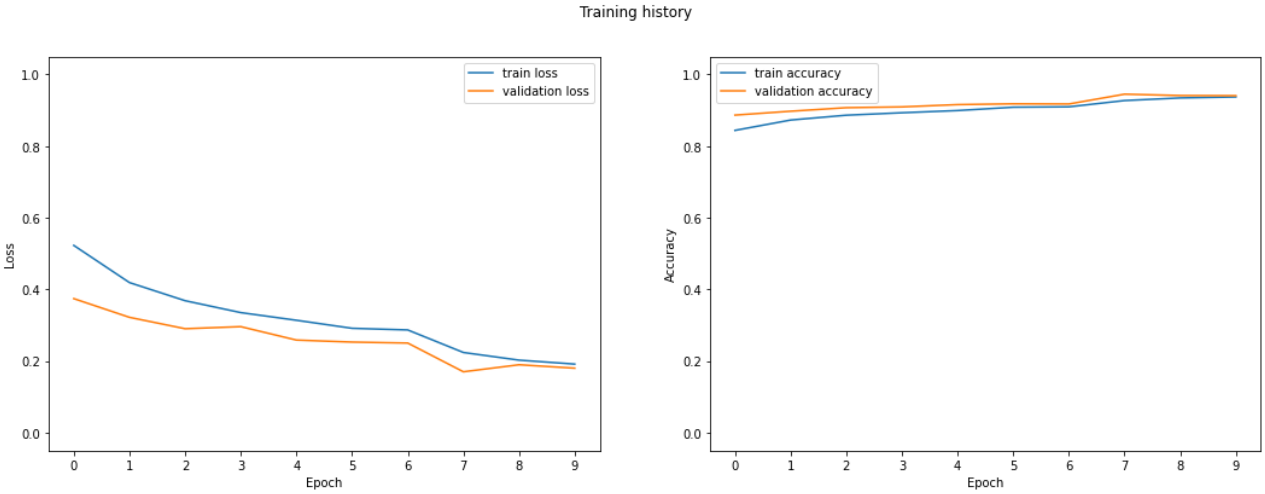


Figura 4.1: Gráfica de los resultados del sistema base.

El objetivo final es que los valores de pérdida de validación y entrenamiento sean lo más similares posibles y que no se produzca ni *Overfitting*, en el que la pérdida de la validación es mucho mayor que la de entrenamiento, ni *Underfitting*, que ambas pérdidas son muy grandes.

Se puede observar el fenómeno previamente comentado en el que la precisión y pérdidas son mejores en la validación en las primeras estancias del entrenamiento de la red. Se obtienen unos

valores de precisión lo suficientemente buenos y con una progresión bastante pronunciada. El tiempo de ejecución del entrenamiento, al haber introducido una gran cantidad de imágenes, ha sido de 2 horas y 30 minutos.

A continuación se evalúa a la red para comprobar la eficacia a la hora de realizar las predicciones una vez la red ha sido entrenada con los datos de test.

Gracias al entrenamiento de la red con imágenes en diferentes situaciones mucho más difíciles de diferenciar, el test tiene una precisión del 97.68 % ya que como se estudia en la descripción del sistema, se usan imágenes que no reciben ningún tipo de transformación.

4.2 Estudio de los hiperparámetros

En esta sección se estudiará el efecto de modificar los hiperparámetros que componen el sistema, mencionados en la sección 4.1, buscando la mejora en cuanto a rendimiento, efectividad y mejora en general del sistema. En cada sección relativa al estudio de los hiperparámetros, solo se modificarán los parámetros referenciados dejando los valores del resto de parámetros como los establecidos en el sistema base. Esto significa que los valores de los hiperparámetros por defecto cuando no sean referenciados son los siguientes:

Épocas	10
Algoritmo de optimización	SGD
Learning Rate	0,001
Momentum	0,9
Batch	4
Modelo	ResNet34
Tamaño de paso	7
Gamma	0.1
Función de pérdida	Entropía cruzada

Tabla 4.3: Valores de hiperparámetros por defecto.

4.2.1 Número de epochs

El primero de los hiperparámetros a estudiar es el del número de epochs que corresponde al número de veces que realizaremos el bucle de entrenamiento en el *dataset*. Por sentido común se podría afirmar que realizar un número mayor de épocas siempre contribuirá a obtener un mejor resultado en cuanto a precisión y pérdida. Sin embargo, esto no es así debido a que si el número de epochs es mayor de lo necesario puede producir el problema de *Overfitting* que produce que la red neuronal aprenda patrones fijos del *dataset* utilizado produciendo su mal funcionamiento en *datasets* completamente nuevos.

El *Overfitting* o sobre-ajuste será diagnosticado cuando la precisión durante el entrenamiento sea relativamente buena y la precisión durante la validación sea relativamente mala.

Se procede a mostrar en la tabla los resultados obtenidos junto al número de épocas seleccionado para cada prueba. Se mostrarán los mejores valores de pérdida y precisión tanto de entrenamiento

como de validación junto a la época en el que se consiguió. Finalmente se muestra el resultado final del entrenamiento:

Número de Epochs	10	15	20	30
Mejor pérdida entrenamiento / epoch	19,16 % / 10	17,01 % / 15	16,85 % / 17	**16,25 %** / 24
Mejor precisión entrenamiento / epoch	93,73 % / 10	94,37 % / 15	94,56 % / 17	**94,82 %** / 24
Mejor pérdida validación / epoch	17,01 % / 8	15,72 % / 13	**14,41 %** / 19	14,41 % / 19
Mejor precisión validación / epoch	94,50 % / 8	94,84 % / 13	**95,06 %** / 19	95,06 % / 19
Valor final de pérdida entrenamiento	19,16 %	**17,10 %**	17,68 %	17,32 %
Valor final de pérdida validación	18,02 %	17,54 %	**14,84 %**	15,69 %
Valor final de precisión entrenamiento	93,73 %	94,37 %	94,36 %	**94,77 %**
Valor final de precisión validación	94,08 %	94,52 %	**95,01 %**	94,34 %

Tabla 4.4: Resultados hiperparámetro epoch.

Figura 4.2: Gráfica de los resultados de la tabla 4.4.

Se puede apreciar que a partir de aproximadamente la época número 20 el sistema se queda en un estado de "meseta". Esto quiere decir que el sistema no está avanzando ya sea debido al valor del *learning rate* que es muy pequeño debido a la utilización del planificador o que el sistema ya no es capaz de mejorar sus predicciones y va a producir el ya mencionado *Overfitting*, significando que su aprendizaje es contraproducente. Es por ello que la conclusión es que el número de épocas perfecto para el sistema es el intervalo de 15 a 25 épocas. Incluso se podrá observar más adelante que podremos aumentar el número de épocas sin perder rendimiento y sin ser contraproducente gracias al uso de diferentes estrategias en la planificación del *learning rate* y diferentes tamaños de *Batch*.

4.2.2 Valor del Learning Rate

La modificación del *learning rate*, para muchos programadores el hiperparámetro más importante de configurar, permite otorgar el tamaño del cambio cada vez que se calcula la pérdida. Se puede decir que determina la velocidad con la que el modelo aprende de los problemas y errores. Cada vez que se calcula el gradiente de error, durante el cálculo de la pérdida, mediante el algoritmo de optimización seleccionado se actualizarán los pesos en función del peso anterior y el valor del *learning rate*. Es por

ello que se podría pensar que un valor relativamente bajo del *learning rate* nos otorgaría un mejor valor, pero se puede producir lo contrario si nos tomara mucho tiempo converger, especialmente si nos quedamos atascados en una región de meseta [36].

A favor de un valor relativamente grande de *learning rate*, nos permitiría un entrenamiento mucho más rápido pues se considera que encuentra el peso indicado con una mayor rapidez.

Es por ello que se realiza el estudio de 3 posibles valores siendo el más grande 0,1 y el más pequeño 0,001. Como observamos en la sección 4.2.1, en el que experimentamos con el número de epochs, el fenómeno meseta sucede a partir de la época 20. Por ello, se realizan las pruebas con 3 valores del *learning rate* durante 30 épocas para observar los posibles efectos de *Overfitting* y meseta.

Valor del Learning Rate	0,1	0,01	0,001
Número de épocas	30	30	30
Mejor valor de pérdida entrenamiento / epoch	26,53 % / 16	21,82 % / 28	**16,25 %** / 24
Mejor valor de precisión entrenamiento / epoch	91,73 % / 16	92,83 % / 28	**94,82 %** / 24
Mejor valor de pérdida validación / epoch	25,82 % / 23	20,83 % / 25	**14,41 %** / 19
Mejor valor de precisión validación / epoch	91,62 % / 23	93,27 % / 25	**95,06 %** / 19
Valor final de pérdida entrenamiento	26,73 %	22,52 %	**17,32 %**
Valor final de pérdida validación	26,02 %	22,34 %	**15,69 %**
Valor final de precisión entrenamiento	91,12 %	92,67 %	**94,77 %**
Valor final de precisión validación	91,22 %	92,63 %	**94,34 %**

Tabla 4.5: Resultados modificando el hiperparámetro Learning Rate.

La conclusión de los resultados de la tabla es que el valor adecuado para el *Learning Rate* es 0,001. Esto puede ser debido a que el modelo con pesos pre-entrenados no necesita una actualización grande en los pesos por lo que el uso de un valor de *learning rate* alto simplemente generaría un valor de acierto y pérdida peores.

Al final el modelo utilizado ha sido entrenado y diseñado para llevar a cabo una determinada tarea, es por ello que no debemos modificar en exceso los pesos, dando menor importancia al gradiente como se explica previamente en esta sección.

En las tres pruebas se puede observar el fenómeno meseta a partir de la época 20 y no se aprecia *Overfitting* en ninguna de ellas.

4.2.3 Tamaño del Batch

El hiperparámetro *Batch* equivale al lote de imágenes que introducimos en el bucle de cada entrenamiento para que el sistema realice sus predicciones. Para poner un ejemplo en el que se entienda el significado de *Batch*, si se dispone de un *Dataset* formado por 100 imágenes y un tamaño de *Batch* de 4 imágenes, se dispondría de 25 lotes de imágenes que se introducirán en el modelo durante el entrenamiento. El tamaño del *Batch* dependerá de la memoria de la GPU utilizada ya que para introducir en el modelo un número grande de imágenes la GPU tendrá que tener una gran cantidad de memoria. También habrá que tener en cuenta el tamaño de cada una de las muestras para calcular el tamaño del *Batch* óptimo.

Es importante determinar el tamaño del *Batch* óptimo debido a que:

- Si es relativamente pequeño se pueden obtener buenas soluciones en las primeras épocas del entrenamiento debido a que el sistema aborda las imágenes con un entrenamiento y actualización de pesos mayor que con tamaños de batch mayores. Sin embargo, el tiempo del entrenamiento será mayor para lotes de imágenes más pequeños debido que para recorrer el Dataset habrá que realizar un mayor número de iteraciones.

- Si es relativamente grande se podrán realizar entrenamientos en un menor tiempo posible reduciendo la carga computacional que tiene que realizar el sistema gracias al paralelismo de las GPU. Sin embargo, como podemos observar en el artículo [37] el uso de tamaños de lotes grandes puede generar un problema de *Generalization Gap* que consiste en la degradación de predicción del sistema en imágenes nuevas. Este fenómeno esta relacionado con el overfitting que consiste en la diferencia de pérdida y precisión del entrenamiento y validación.

Como se menciona previamente es preciso calcular el tamaño máximo del *batch* según la memoria de la gráfica y el tamaño de las entradas. Hacemos uso de la gráfica Tesla P100 con una memoria de 16 GB y haciendo uso de la función summary del módulo *Torchsummary* podemos obtener el tamaño de la entrada a la red de cada imagen. Se llega a la conclusión que el tamaño máximo de *batch* es de 64 muestras.

Tamaño del Batch	4	12	24	64
Número de épocas	10	10	10	10
Duración aprox.(min)	38	33	28	22
Mejor pérdida entrenamiento / epoch	19,16 % / 10	17,80 % / 10	**17,63 %** / 10	20,81 % / 10
Mejor precisión entrenamiento / epoch	93,53 % / 10	94,17 % / 10	**94,46 %** / 10	93,54 % / 10
Mejor pérdida validación / epoch	17,01 % / 7	**15,45 %** / 9	15,67 % / 10	19,77 % / 10
Mejor precisión validación / epoch	94,50 % / 7	**94,91 %** / 10	94,67 % / 10	93,71 % / 10
Valor final de pérdida entrenamiento	19,16 %	17,80 %	**17,63 %**	20,81 %
Valor final de pérdida validación	18,02 %	**15,56 %**	15,67 %	19,77 %
Valor final de precisión entrenamiento	93,52 %	94,17 %	**94,46 %**	93,54 %
Valor final de precisión validación	94,08 %	**94,91 %**	94,67 %	93,71 %

Tabla 4.6: Resultados modificando el hiperparámetro Batch.

En los resultados dispuestos en la Tabla 4.6 se puede apreciar como a medida que vamos aumentando el tamaño del batch los resultados son similares pero con el hecho de que al utilizar un batch mayor, el tiempo de entrenamiento se reduce. Es por ello que aun sin haber conseguido un resultado mucho mejor en cuanto a precisión y pérdida, un buen compromiso entre precisión y el tiempo del entrenamiento hace llegar a la conclusión de que el tamaño del *batch* óptimo es de 24 muestras.

4.2.4 Planificador del Learning Rate

Se podría llegar a pensar que el sistema no es capaz de obtener una precisión mayor del 95 % debido a que todas las pruebas realizadas hasta el momento no han evitado el problema del fenómeno de meseta. Ese es el principal problema pues a partir de la época [15-20] en la mayoría de las pruebas el sistema obtiene predicciones muy similares sin seguir con la mejora progresiva. No muestra *Overfitting*

excesivo pero el hecho de seguir entrenando la red para seguir obteniendo los mismos valores es contraproducente.

Uno de los mecanismos estudiados para abordar el fenómeno de meseta es el de planificar el *learning rate* y modificarlo cuando se produzca el estado de meseta para poder llegar a converger en la progresión de la pérdida. Es por ello que se procede a estudiar diferentes métodos de planificación del *learning rate* para poder llegar a abordar el problema principal del sistema.

Aunque el fin es el mismo, se realizan las siguientes pruebas con los tres planificadores explicados en la sección 2.2.1.3 en el que se explicaba el módulo *Torch.optim*. A continuación se explica el funcionamiento de cada uno y el valor del *learning rate* en cada momento:

- ConstantLR: si a partir de la época 15 se detecta el problema de meseta, vamos a introducir un *learning rate* fijo de 0,001 hasta que se llegue a la época 15 que aumentamos el *learning rate* a un valor de 0,01. Los valores para conseguir el anterior funcionamiento será el observado en la siguiente figura:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# lr = 0.001    if 0 < epoch  < 15
# lr = 0.01    if epoch > 15
scheduler = lr_scheduler.ConstantLR(optimizer, factor=0.1, total_iters=15)
```

Figura 4.3: Configuración del planificador ConstantLR.

Se puede observar como partimos de un valor de *learning rate* de 0,01 que el planificador reducirá multiplicando por un *factor* de 0,1 durante las primeras 15 épocas para tratar de converger con un valor de *learning rate* mayor.

Los resultados obtenidos no son favorables pues si bien salimos del estado de meseta para converger en un valor peor, no se llega a valores previamente obtenidos por lo que el sistema no se debe usar.

Podemos observar los resultados en la siguiente gráfica:

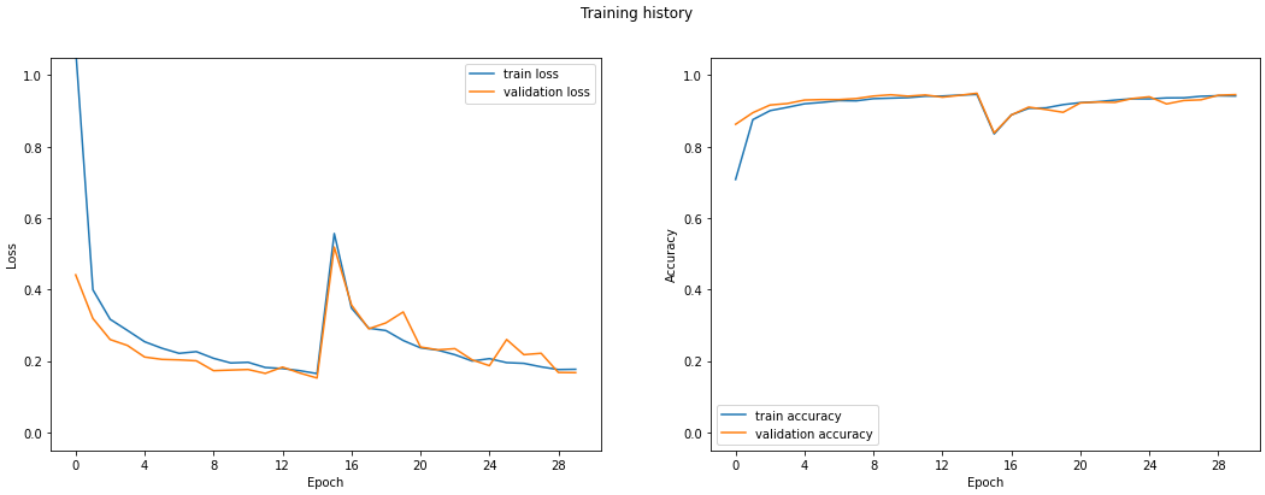


Figura 4.4: Gráfica planificador ConstantLR.

Como se puede apreciar llegamos en la época número 15 a valores de precisión de 94,45 % y de pérdida de 15,54 %, pero una vez introducida la modificación del *learning rate* no se vuelve a aproximar a los valores máximos previamente obtenidos.

- MultiStepLR: se va a realizar una estrategia diferente al planificador ConstantLR, en este caso se va a proceder a disminuir dos veces el valor del *learning rate* como se dispone en la siguiente figura:

```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
# lr = 0.001 if 0 < epoch < 10
# lr = 0.0001 if 10 < epoch < 15
# lr = 0.00001 if 15 < epoch < 20
scheduler = lr_scheduler.MultiStepLR(optimizer, milestones=[10,15], gamma=0.1)
```

Figura 4.5: Configuración del planificador MultiStepLR.

En este caso se comienza con un *learning rate* de 0,001 para que cuando se detecte el estado de meseta se realice una reducción del *learning rate* en un valor *gamma* de 0,1 en la época 10 y 15.

Los resultados obtenidos son positivos pues se observa una ruptura, entre la época 10 y 15, del valor de la precisión y pérdida con una ganancia de casi un 3 %. Es en la época 16 cuando llegamos al valor máximo y mínimo, de la precisión y pérdida respectivamente, desde que se comenzó a experimentar con el sistema. El resultado es 95,78 % en cuanto a precisión y 12,84 en cuanto a la pérdida ambos en la fase de validación.

Los resultados se observan en la siguiente gráfica :

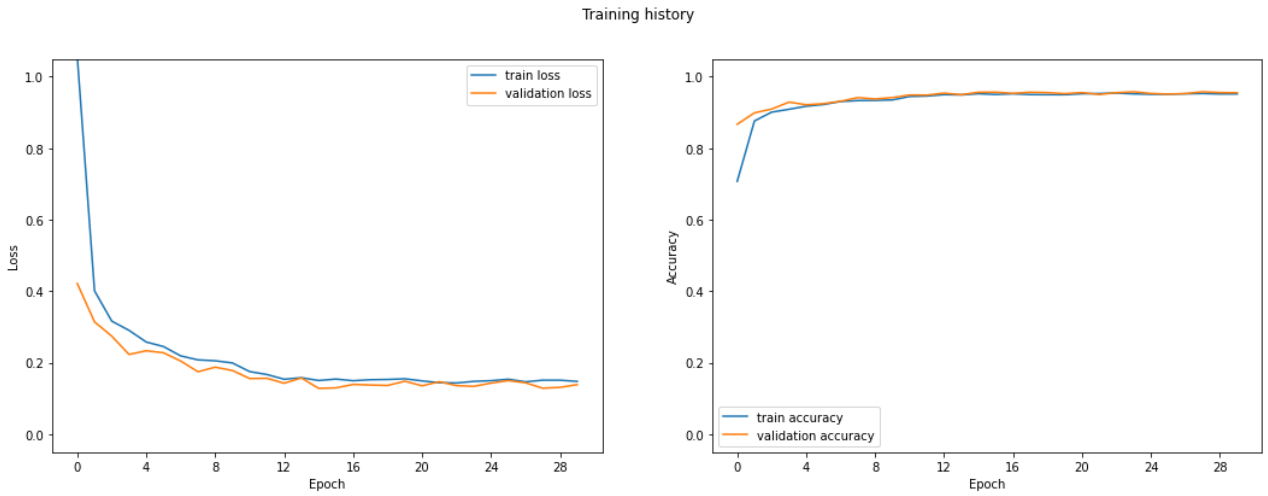


Figura 4.6: Configuración del planificador MultiStepLR.

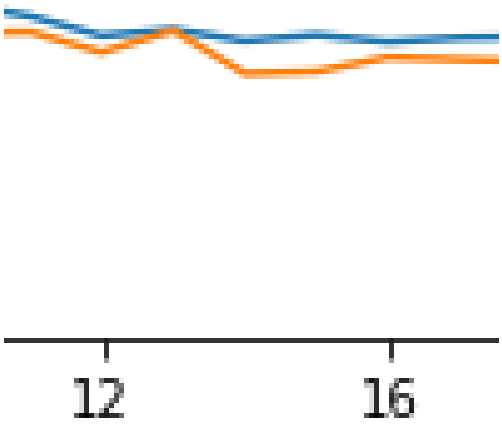


Figura 4.7: Zoom al valor de pérdida entre epoch 10 a epoch 19.

El cambio del entrenamiento es muy gradual sin realizar cambios bruscos mientras que, como se observa en la figura 4.7, el valor de pérdida si sufrirá dichos cambios bruscos debido al cambio del *learning rate* llegando a disminuir entre la epoch 12 y 16 de un valor del 20 % al 16 %.

Se puede observar como obtenemos ese máximo y mínimo y a partir de la época 19 como los resultados de la validación van empeorando produciendo el ya comentado *Overfitting* por lo que refuerza la teoría de que el número de épocas ideal oscilaría entre 15 y 20 épocas.

- ReduceLROnPlateau: reducirá el learning rate en el momento que no obtengamos mejoras del resultado introducido en la función step durante dos épocas según los valores que se muestran en la siguiente figura:

```
# lr = lr * 0.1 if val_loss_new < val_loss
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min',patience=1)
```

Figura 4.8: Configuración del planificador ReduceLROnPlateau.

Los valores seleccionados son el modo igual a 'min' debido a que buscamos que si el valor de pérdida de la validación es inferior al mínimo ya establecido tendrá una paciencia *patience* de 1 época adicional en la que podría suceder dicho suceso. En el momento en el que dos veces seguidas se produzca un valor menor que el establecido previamente el planificador considera que estamos en un estado de meseta y reduce el *learning rate* establecido en el planificador multiplicándolo por 0,1 que es establecido por *default*.

Debido a que en este caso el *learning rate* no tiene modificaciones fijas, se va a mostrar en la tabla su valor a lo largo del entrenamiento:

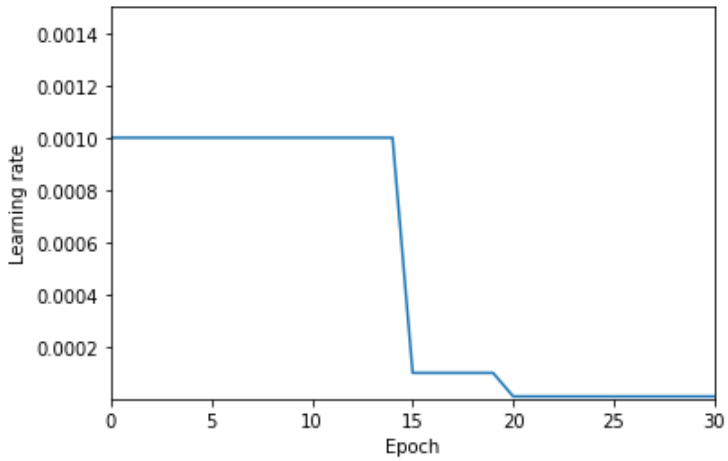


Figura 4.9: Gráfico del valor Learning Rate.

Los resultados se pueden observar en la siguiente gráfica:

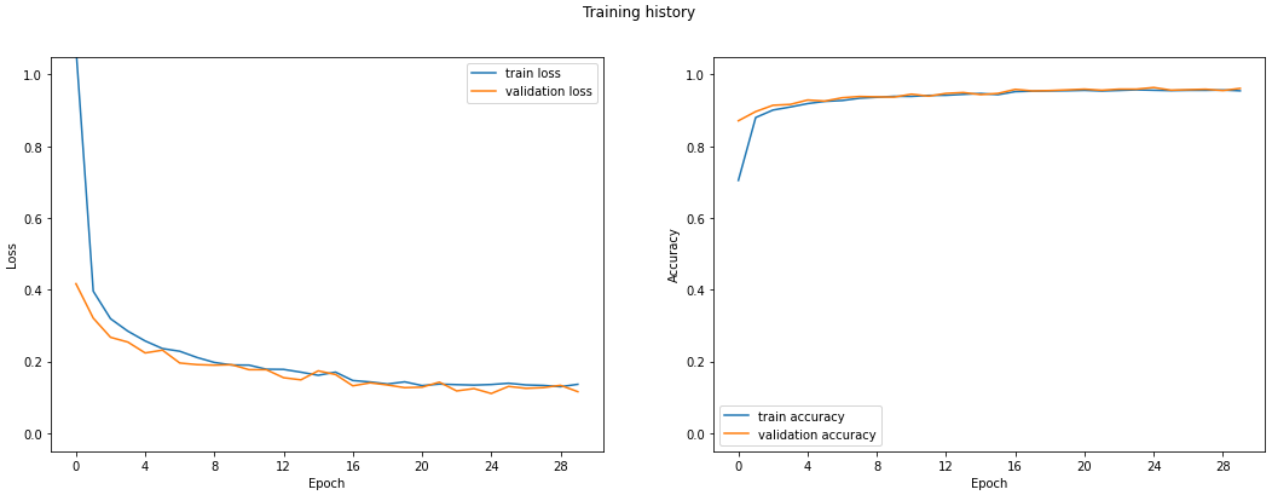


Figura 4.10: Gráfico del planificador ReduceLROnPlateau.

En la gráfica se observan valores positivos pues se producen dos situaciones de violación del mínimo valor en la época 15 y 19 por lo que el valor se reduce 100 veces entre dicho intervalo, resultando en la época 23 el mejor resultado hasta el momento que equivale a 96,39 % de precisión y 11,08 % de pérdida ambos en la fase de validación. Se demuestra que el funcionamiento del planificador es vital para evitar esas situaciones no deseadas en la que el sistema no está aprendiendo del entrenamiento.

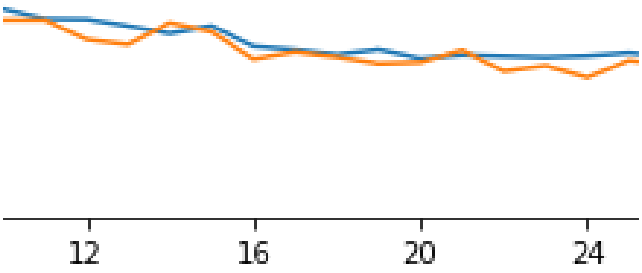


Figura 4.11: Zoom al valor de pérdida entre epoch 12 a epoch 24.

Se pueden observar dichos cambios bruscos del valor de pérdida en los momentos en el que se realiza la disminución del valor del *learning rate* cuando el planificador detecta el estado de meseta siendo las epochs 15, 20 y 24.

La conclusión final es que se debe reducir, a medida que el sistema se establece en un determinado valor fijo, el valor del *learning rate* para ayudar a evitar el efecto de meseta. Esta función es realizada de la mejor manera por el planificador del módulo de la librería Pytorch creado para evitar dicho problema que es el planificador *ReduceLROnPlateau*.

4.3 Algoritmo de optimización

Una vez estudiados los hiperparámetros que más convienen para el entrenamiento de la red, se procede a hacer el estudio de los tres tipos de algoritmos de optimización expuestos en la sección 2.2.1.3 habiendo ya obtenido los resultados del algoritmo de optimización SGD. Se va a realizar el estudio durante 30 epochs, aunque haya resultado ser en cierto modo un número de épocas superior al óptimo, debido a que se quiere observar indicios de *Overfitting* y estados de meseta.

En la tabla 4.7 se disponen los resultados obtenidos:

Algoritmo de optimización	Adam	Adadelta	SGD
Tamaño del Batch	24	24	24
Número de épocas	30	30	30
Mejor valor de pérdida entrenamiento / epoch	13,43 % / 27	31,10 % / 30	**12,71 %** / 27
Mejor valor de precisión entrenamiento / epoch	95,59 % / 27	91,11 % / 30	**95,95 %** / 27
Mejor valor de pérdida validación / epoch	13,02 % / 23	27,76 % / 30	**11,08 %** / 23
Mejor valor de precisión validación / epoch	95,63 % / 23	91,54 % / 30	**96,39 %** / 23
Valor final de pérdida entrenamiento	13,37 %	31,10 %	**13,01 %**
Valor final de pérdida validación	13,46 %	27,76 %	**12,65 %**
Valor final de precisión entrenamiento	95,53 %	91,11 %	**95,77 %**
Valor final de precisión validación	95,60 %	91,54 %	**96,03 %**
Learning rate inicial	1e-3	1e-3	1e-3
Época/s en la que se produce meseta	16/26/29	-	15/19
Learning rate final	1e-6	1e-3	1e-5

Tabla 4.7: Resultados Algoritmos de optimización.

Los resultados se muestran gráficamente en comparación unos con otros en la figura 4.12.

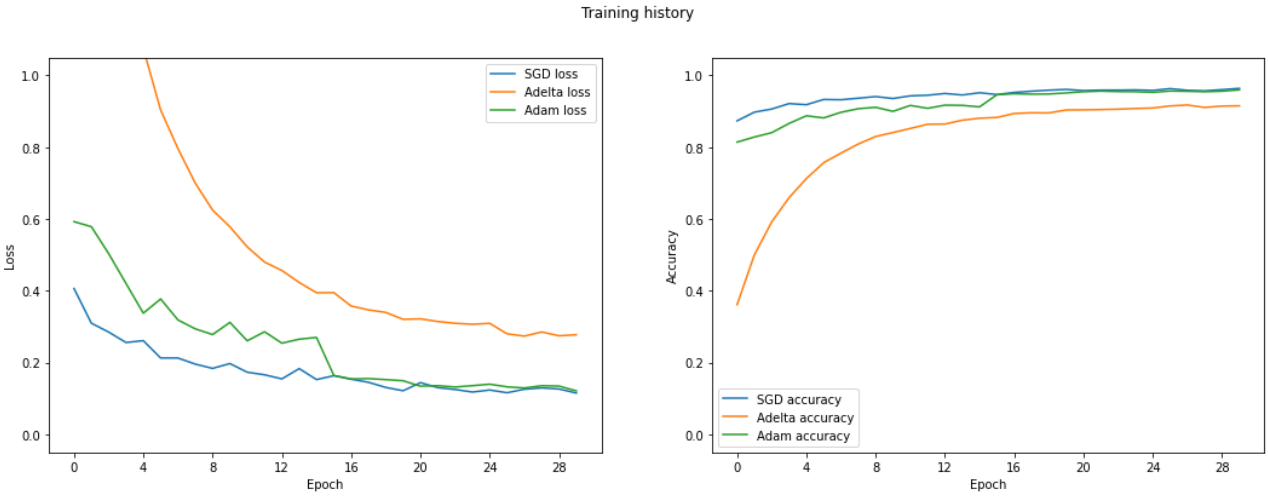


Figura 4.12: Comparación de resultados entre algoritmos de optimización.

El desarrollo del valor del *Learning rate* a lo largo de las 30 epochs se muestra en la figura 4.13.

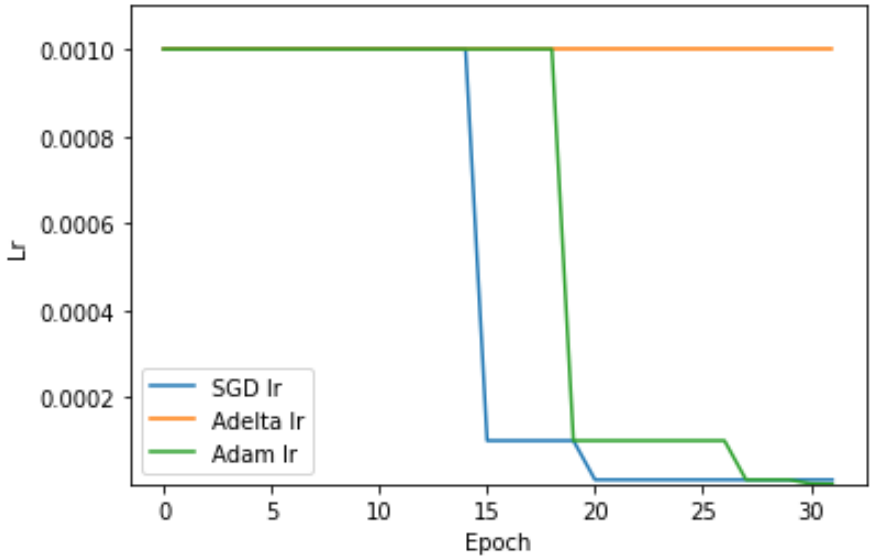


Figura 4.13: Comparación de resultados entre algoritmos de optimización.

Las conclusiones respecto a cada uno de los algoritmos de optimización son:

- Adam: Los resultados obtenidos se pueden calificar de buenos pero no llegan al nivel obtenido por el algoritmo de optimización SGD. Se observa un estado de meseta al rededor del valor 91 % en cuanto a la precisión de validación entre la época 11 y 15, consiguiendo encadenar dos resultados peores en la época 15 reduciendo el *learning rate* a 1e-4. En ese momento se obtiene una mejora sustancial de 3 % llevando al sistema a establecerse en una precisión de casi del 95 % en la época 17. Finalmente el sistema realiza mejoras muy pequeñas hasta que se alcanzan dos estados de meseta tanto en la época 26 como en la 29 siendo la proximidad entre ambas un indicio de que el mínimo máximo ya se ha alcanzado. Se observa que el algoritmo Adam en comparación con el algoritmo SGD es muy propenso a establecerse en estados de meseta produciendo mejoras bruscas y no progresivas como el algoritmo SGD. Los resultados

son mejores utilizando el algoritmo SGD por lo que en comparación con el algoritmo Adam se seguirá usando el primero.

- Adadelta: Los resultados obtenidos son muy pobres, siendo su mejor resultado conseguido en la última época el equivalente al resultado de la época 11 de los otros dos algoritmos de optimización estudiados. El planificador no actúa en ningún momento debido a que el aprendizaje es progresivo pero muy pequeño. No se hará uso de este algoritmo de optimización debido a su rendimiento bajo respecto al algoritmo Adam y SGD.
- SGD: Su explicación y resultados son explicados en secciones anteriores. Desde el principio los valores son los más altos, obteniendo en un menor tiempo posible buenos valores de precisión y pérdida. Genera menos estados de meseta que el algoritmo de optimización Adam.

Tanto Adam como SGD tienen resultados muy positivos, mientras Adadelta no cumple con los requisitos mínimos impuestos para el sistema. SGD será el utilizado en el resto de la experimentación del sistema debido a que llega, en comparación con Adam y Adadelta, a valores altos de precisión y bajos de pérdida en las épocas iniciales.

4.4 Modelos con pesos pre-entrenados

Se procede al estudio y comparación de los tres modelos explicados en la sección 3.4. Se tendrán en cuenta para precisar conclusiones no solo los resultados de precisión y pérdida sino también la velocidad del entrenamiento.

Las configuraciones que se deben realizar en cada uno de los modelos es diferente:

- Alexnet: En este caso Alexnet es recibido, por parte de la librería Pytorch, con una capa fully-connected que puede llegar a etiquetar 1000 diferentes tipos de clases.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=1024, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1024, out_features=43, bias=True)
  )
)
```

Figura 4.14: Conjunto del modelo Alexnet.

Es por ello que para evitar errores no reales en la red se deben establecer en la capa linear [4] una entrada de 4096 y una salida de 1024 y en la última capa linear [6] una entrada igual a la

salida de la anterior capa y una salida de 43 equivale al tipo de clases de señales de tráfico que clasificamos en nuestro sistema. El formato de la capa linear [4] y [6] se observan en la figura 4.12. Es por ello que se debe acceder al atributo *classifier* para poder modificar dichas capas como se observa en la figura 4.13.

```
alexnet_model.classifier[4] = nn.Linear(4096,1024)
alexnet_model.classifier[6] = nn.Linear(1024, 43)
```

Figura 4.15: Configuración y modificación del modelo AlexNet.

- Inception: En este caso Inception es recibido, por parte de la librería Pytorch, con una capa *fully-connected* que puede llegar a etiquetar 1000 diferentes tipos de clases. Por lo que se configurará de la misma manera que se realiza con ResNet obteniendo el número de entradas a la capa *fully-connected* y modificando el atributo *fc* para poder establecer 43 tipos de señales de tráfico como salida.

```
num_ftrs = inception_model.fc.in_features
inception_model.fc = nn.Linear(num_ftrs,43)
inception_model.aux_logits=False
```

Figura 4.16: Configuración y modificación del modelo Inception.

Inception contará con dos peculiaridades que en caso de no tener en cuenta llevará a sendos errores que no permitirán ejecutar el código de entrenamiento. El primero de ellos es debido a que el modelo Inception en su versión 3, como establece C.Szegedy en [4] y en la documentación de Pytorch en [38], ha sido diseñado para recibir imágenes de 300x300 píxeles, por lo que se deberá modificar las transformaciones realizadas en el *Dataset*, explicadas en la sección 3.3.2, para establecer un tamaño de corte de 300 píxeles.

```
train_transforms = T.Compose([
  T.RandomResizedCrop(size=299),
  T.RandomRotation(degrees=15),
  T.RandomHorizontalFlip(),
  T.ToTensor(),
  T.Normalize(mean_nums, std_nums)
])
val_transforms = T.Compose([
  T.RandomResizedCrop(size=299),
  T.RandomRotation(degrees=15),
  T.RandomHorizontalFlip(),
  T.ToTensor(),
  T.Normalize(mean_nums, std_nums)
])
```

Figura 4.17: Modificación de las transformaciones del Dataset para poder operar con el modelo Inception.

El segundo de ellos es debido a que la salida del modelo introduce un atributo llamado *.logits* que no podrá ser tratado por la función *Torch.max* en el momento de obtener las predicciones.

Es por ello que se deberá deshabilitar la salida del atributo negando el atributo del modelo *aux_logits*, como se observa en la figura 4.14, para poder proceder con normalidad.

- ResNet: Se hará uso del modelo ResNet de 34 capas también conocido como ResNet34. En este caso ResNet es recibido, por parte de la librería Pytorch, con una capa *fully-connected* que puede llegar a etiquetar 1000 diferentes tipos de clases. Es por ello que se configurará de la misma manera que Inception en el que se obtienen el número de características de entrada a la capa *fully-connected* y se establece la salida en 43 que es el número de tipos de señales de tráfico.

```
num_ftrs = resnet_model.fc.in_features
resnet_model.fc = nn.Linear(num_ftrs, 43)
```

Figura 4.18: Configuración y modificación del modelo ResNet.

En este caso no habrá que realizar ninguna modificación relacionada con las salidas y entradas del modelo como sucedía con el modelo de Inception. Se volverán a introducir imágenes con un tamaño de 256 píxeles.

Los resultados obtenidos se pueden observar en la tabla 4.8.

Modelo	ResNet	Inception	AlexNet
Algoritmo de optimización	SGD	SGD	SGD
Tamaño del Batch	24	24	24
Número de épocas	30	30	30
Duración del entrenamiento (min)	84	134	24
Mejor valor de pérdida entrenamiento / epoch	**11,82 %** / 27	12,71 % / 30	24,01 % / 30
Mejor valor de precisión entrenamiento / epoch	95,72 % / 27	**95,87 %** / 28	92,26 % / 30
Mejor valor de pérdida validación / epoch	**11,08 %** / 23	11,96 % / 25	22,02 % / 29
Mejor valor de precisión validación / epoch	**96,39 %** / 23	96,22 % / 25	93,01 % / 28
Valor final de pérdida entrenamiento	13,76 %	**12,71 %**	24,01 %
Valor final de pérdida validación	13,46 %	**12,05 %**	22,52 %
Valor final de precisión entrenamiento	95,53 %	**95,83 %**	92,26 %
Valor final de precisión validación	95,56 %	**95,80 %**	93,00 %
Learning rate inicial	1e-3	1e-3	1e-3
Época/s en la que se produce meseta	16/26/29	24/29	16/22/25/27/29
Learning rate final	1e-6	1e-5	1e-8

Tabla 4.8: Resultados: Modelos con pesos pre-entrenados.

La gráfica con los resultados enfrentados entre los tres modelos se muestra en la figura 4.19.

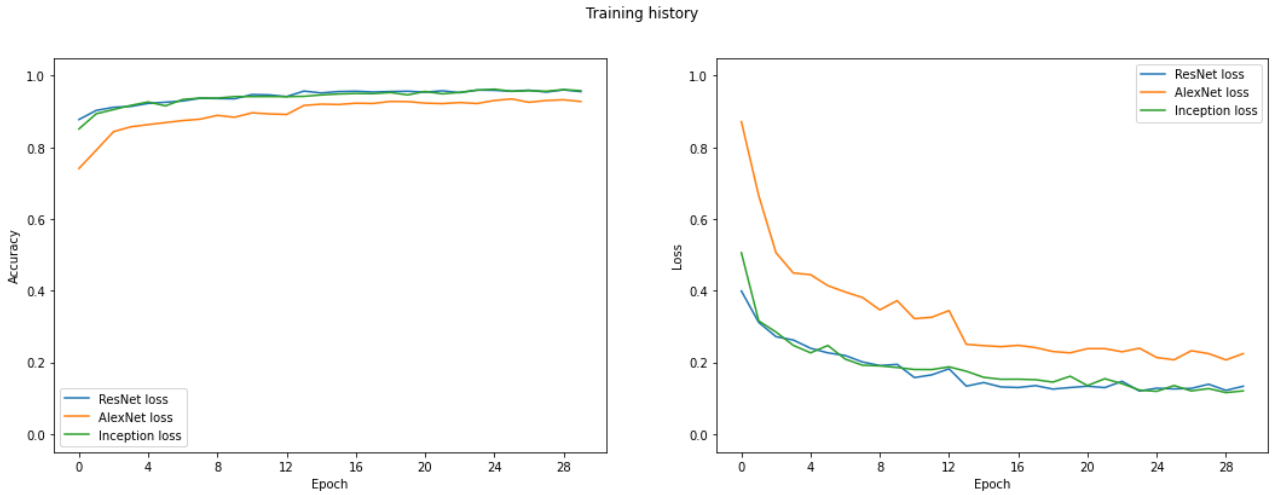
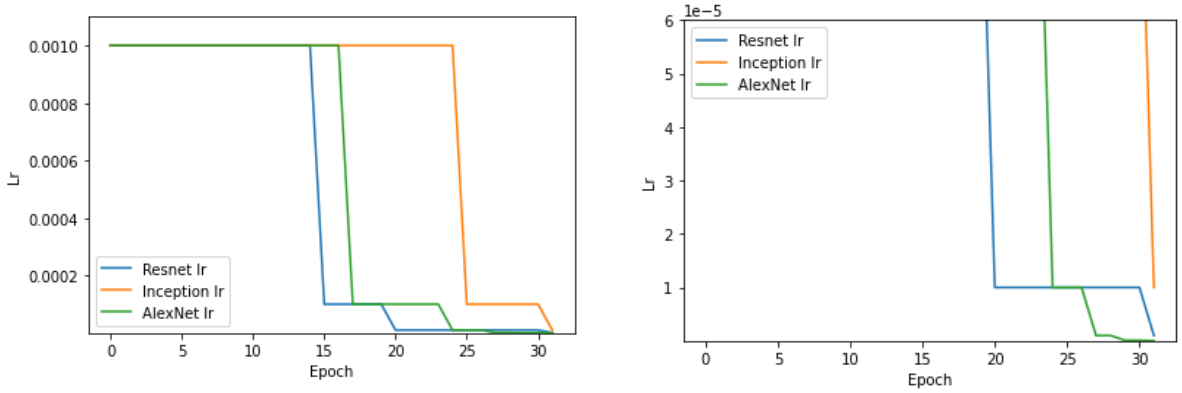


Figura 4.19: Comparación entre modelos de precisión y pérdidas durante validación.

La gráfica con los valores del *learning rate* a lo largo del entrenamiento se muestra en la figura 4.20.



(a) (b)

Figura 4.20: Figura (a) enfrenta los valores de learning rate en cada modelo ; Figura (b) representa la misma gráfica que (a) pero ampliando la vista en las últimas épocas.

Tras la exposición de los resultados se realizan las conclusiones respecto a cada uno de los modelos:

- AlexNet: Si bien los resultados obtenidos no son igual de buenos que los obtenidos en Inception y ResNet, AlexNet destaca por su velocidad en el entrenamiento siendo su duración hasta cinco veces más baja que Inception y cuatro veces más baja que ResNet. Uno de los objetivos de los creadores era reducir el tiempo de entrenamiento mediante el uso de menos capas convolucionales y reducir el overfitting mediante la introducción de la capa de regularización *dropout*. Sin embargo, se observan muchos estados de meseta pudiendo ser debido a que el *learning rate* seleccionado no era el adecuado para este modelo. Se recomendará el uso de AlexNet en el momento en el que el usuario no disponga de una máquina con la suficiente capacidad computacional para realizar entrenamiento en tiempos no razonables. Pero en el caso de este sistema

se debe buscar la mayor precisión y menor pérdida por lo que los otros dos modelos serán las opciones a elegir.

- Inception: Los resultados obtenidos son muy parecidos a los obtenidos mediante el modelo ResNet y mejores que los obtenidos mediante el modelo AlexNet. Sin embargo, el modelo Inception es el que mayor duración ha tenido el entrenamiento debido a su número de capas superior a los otros dos modelos. Los valores de ResNet llegan a ser mínimamente mejores pero eso dependerá en cierto modo de lo acertado que esté en dicho momento el sistema, por lo que se considera que ambos modelos alcanzan el mismo valor de precisión de validación. Uno de los puntos positivos del modelo Inception es su capacidad de evitar estados de meseta hasta las épocas finales del entrenamiento, alcanzando dicho estado únicamente dos veces siendo sus valores de precisión y pérdida progresivos. Si bien es cierto que los valores obtenidos tanto en ResNet como Inception son similares, se optará por el uso del modelo ResNet principalmente por su diferencia en cuanto a duración del entrenamiento.

- ResNet: Tanto los valores como conclusiones ya han sido expuestas durante toda la experimentación pues esta ha sido realizada con la red ResNet. Se considera la mejor opción debido a que se obtienen los resultados más altos junto a una duración aceptable.

La conclusión será el uso del modelo con pesos pre-entrenados ResNet debido a sus buenos resultados y duración aceptable. Si bien cabe mencionar la posibilidad de uso de la red AlexNet para sistemas con menor capacidad computacional.

4.5 Sistema Final

Tras haberse evaluado los efectos de variar los diferentes hiperparámetros, redes y optimizadores, haber experimentado con ellas y haber sacado las conclusiones finales, se llega al sistema final considerado la mejor opción en cuanto a rendimiento y resultados:

- *DataLoaders* formados por un valor de *Batch* de veinticuatro imágenes mezcladas mediante la función *Shuffle*.

- Modelo con pesos pre-entrenados ResNet en su versión de 34 capas.

- Algoritmo de optimización SGD con un valor de *learning rate* inicial de 0.001 y con un valor de momentum de 0.9.

- Planificador de learning rate *ReduceLROnPlateau* reduciendo el valor del *learning rate* multiplicándolo por un valor 0,1 *gamma* a partir del valor límite de la pérdida de validación con una paciencia de 1 evento.

- Función de pérdida de entropía cruzada.

- El número de épocas será de veinticinco.

Se muestra en la tabla 4.9 la comparación entre los resultados obtenidos en el sistema base y en el sistema final.

Sistema	Base	Final
Mejor valor de pérdida entrenamiento / epoch	19,16 % / 10	**11,85 %** / 24
Mejor valor de precisión entrenamiento / epoch	93,73 % / 10	**95,62 %** / 24
Mejor valor de pérdida validación / epoch	17,01 % / 8	**11,08 %** / 23
Mejor valor de precisión validación / epoch	94,50 % / 8	**96,39 %** / 23
Valor final de pérdida entrenamiento	19,16 %	**11,87 %**
Valor final de pérdida validación	18,02 %	**11,54 %**
Valor final de precisión entrenamiento	93,73 %	**95,54 %**
Valor final de precisión validación	94,08 %	**96,04 %**

Tabla 4.9: Sistema base vs sistema final.

Los resultados obtenidos son muy positivos, pues los modelos con pesos pre-entrenados no necesitan una configuración excesivamente complicada para que obtengan buenos resultados y será complicado mejorar los resultados ya obtenidos.

Capítulo 5

Conclusiones y líneas futuras

Para finalizar se exponen las conclusiones obtenidas durante el desarrollo del trabajo y las posibilidades futuras que otorga el sistema.

5.1 Conclusiones

A modo de resumen, en este trabajo se han realizado las siguientes aportaciones:

- Un sistema capaz de identificar una serie de señales de tráfico.
- Estudio de los componentes de las redes neuronales convolucionales.
- Estudio de la librería Pytorch para establecer el sistema.
- Estudio del sistema utilizado para el cometido de identificación de las señales de tráfico.
- Estudio de las diferentes modificaciones a realizar en busca del sistema perfecto.
- Aumentar el número de epochs puede mejorar los resultados, siempre que no se excede dicho número debido a que puede causar *Overfitting*. Se ha identificado el número ideal de epochs entre 20 y 25.
- Se deberá disminuir el valor del *learning rate* en los momentos en los que se detecten estados de meseta de lo contrario no se conseguirá converger en los valores de pérdida y precisión.
- Los modelos con pesos pre-entrenados otorgan valores muy buenos en epochs tempranas pero modelos como AlexNet priorizan un tiempo de entrenamiento pequeño mientras que Inception y ResNet priorizan mejores valores de entrenamiento y validación.

5.2 Líneas Futuras

Como se ha comentado en las conclusiones del trabajo, se han cubierto los objetivos planteados al comienzo del mismo. Sin embargo, en el futuro se podrán realizar diferentes extensiones y mejoras:

- Implementación de una base de datos más extensa que contenga señales de tráfico de otros países para que la implementación del sistema pueda ser utilizada en cualquier país.
- Implementación de una red neuronal convolucional propia sin la necesidad de hacer uso de un modelo con pesos pre-entrenados.
- Realizar una experimentación del sistema más extensa en la búsqueda del mayor rendimiento y precisión, estableciendo por ejemplo un número mayor de épocas o un mayor número de planificadores de *learning rate*.
- Implementación de un sistema que sea capaz de detectar una señal de tráfico en una imagen con un mayor ruido e información de su entorno que no represente únicamente la propia señal.
- Implementación de un ejemplo real mediante el montaje de un vehículo a escala mediante un sistema empotrado, con una cámara capaz de pasar imágenes a la red y que en tiempo real adapte el funcionamiento del vehículo según la predicción del sistema ya creado.

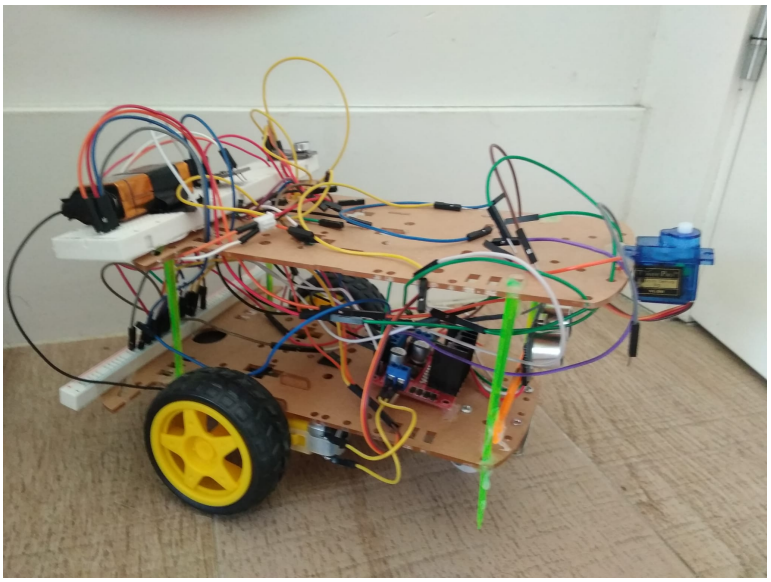


Figura 5.1: Ejemplo de vehículo para implementación de sistema real. Desarrollado para la asignatura Sistemas Electrónicos Digitales Avanzados.

Bibliografía

[1] Venelin Valkov, "Transfer learning for image classification with pytorch and python," 2020, [Online], https://colab.research.google.com/drive/1Lk5R4pECDxDhd1uXcv26YRQ02fb_mrL9?usp=sharing#scrollTo=X-ZxBZO4ouW9, Último Acceso en 2022-05-15.

[2] K. He, X. Zhang, S. Ren, y J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] A. Krizhevsky, I. Sutskever, y G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, y K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Disponible en: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, y Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[5] G. L. Santos, P. T. Endo, K. H. d. C. Monteiro, E. d. S. Rocha, I. Silva, y T. Lynn, "Accelerometer-based human fall detection using convolutional neural networks," *Sensors*, vol. 19, no. 7, p. 1644, 2019.

[6] PyLessons, "Convolutional neural networks step by step introduction," 2019, [Online], https://pylessons.com/CNN-tutorial-introduction, Último Acceso en 2022-03-15.

[7] SuperDataScience Team, "Convolutional neural networks (cnn): Step 1(b) - relu layer," 2018, [Online], https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1b-relu-layer, Último Acceso en 2022-03-12.

[8] CS231n, "Convolutional neural networks for visual recognition," [Online], https://cs231n.github.io/convolutional-networks/, Último Acceso en 2022-03-16.

[9] Juan Pedro Dominguez Morales, "Average pooling," [Online], https://www.researchgate.net/figure/Average-pooling-example_fig21_329885401, Último Acceso en 2022-03-16.

[10] S. Moncada, "The ultimate guide to convolutional neural networks (cnn)," 2018, [Online], https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn, Último Acceso en 2022-03-16.

[11] "Adam optimizer," [Online], https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, Último Acceso en 2022-03-18.

[12] "Sgd optimizer," [Online], https://pytorch.org/docs/stable/generated/torch.optim.SGD.html, Último Acceso en 2022-03-18.

[13] "Adadelta optimizer," [Online], https://pytorch.org/docs/stable/generated/torch.optim.Adadelta.html, Último Acceso en 2022-03-18.

[14] "Función tangente hiperbólica," [Online], https://numerentur.org/funcion-de-activacion-tangente-hiperbolica/, Último Acceso en 2022-03-25.

[15] "Visión artificial, red neuronal convolucional, alexnet," [Online], https://oi.readthedocs.io/en/latest/computer_vision/cnn/alexnet.html, Último Acceso en 2022-03-25.

[16] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[17] Tokio School, "Aplicaciones-del-deep-learning," 2021, [Online], https://www.tokioschool.com/noticias/aplicaciones-del-deep-learning/, Último Acceso en 2022-03-12.

[18] L. M. Martinez y J.-M. Alonso, "Complex receptive fields in primary visual cortex," *The neuroscientist*, vol. 9, no. 5, pp. 317–331, 2003.

[19] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, y J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Twenty-second international joint conference on artificial intelligence*, 2011.

[20] Renu Khandelwal, "Convolutionalneuralnetwork: Feature map and filter visualization," 2020, [Online], https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filter-visualization-f75012a5a49c#:~:text=Feature%20maps%20are%20generated%20by,Convolutional%20layers%20in%20the%20model., Último Acceso en 2022-03-20.

[21] Paperswithcode, "Average pooling," 2020, [Online], https://paperswithcode.com/method/average-pooling, Último Acceso en 2022-03-28.

[22] Open source Pytorch library, "Pytorch," 2022, [Online], https://pytorch.org/, Último Acceso en 2022-04-02.

[23] Open source Pytorch library Torch Module, "Torch," 2022, [Online], https://pytorch.org/docs/stable/torch.html, Último Acceso en 2022-04-02.

[24] Torch Backward Function, "Torch.backward," 2022, [Online], https://pytorch.org/docs/stable/generated/torch.autograd.backward.html, Último Acceso en 2022-03-29.

[25] Torch Optim Function, "Torch.optim," 2022, [Online], https://pytorch.org/docs/stable/optim.html, Último Acceso en 2022-03-29.

[26] D. P. Kingma y J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[27] Kingma, Diederik P and Ba, Jimmy, "Adam optimizer," 2022, [Online], https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, Último Acceso en 2022-03-29.

[28] A. V. Srinivasan, "Stochastic gradient descent-clearly explained," *Medium. url: https://towardsdatascience. com/stochastic-gradient-descentclearly-explained-53d239905d31*, 2019.

[29] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[30] Torch Utils Function, "Torch.utils," 2022, [Online], https://pytorch.org/docs/stable/data.html, Último Acceso en 2022-03-29.

[31] Open source Pytorch library Torchvision Module, "Torchvision," 2022, [Online], https://pytorch.org/vision/stable, Último Acceso en 2022-04-02.

[32] Torchvision Models Function, "Torchvision.models," 2022, [Online], https://pytorch.org/vision/stable/models.html, Último Acceso en 2022-04-02.

[33] Google, "Google colab," [Online], https://colab.research.google.com/, Último Acceso en 2022-05-28.

[34] J. Stallkamp, M. Schlipsing, J. Salmen, y C. Igel, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *IEEE International Joint Conference on Neural Networks*, 2011, pp. 1453–1460.

[35] K. Simonyan y A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[36] H. Zulkifli, "Understanding learning rates and how it improves performance in deep learning," *Towards Data Science*, vol. 21, no. 23, 2018.

[37] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, y P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[38] Szegedy, Christian and Vanhoucke, Vincent and Ioffe, Sergey and Shlens, Jon and Wojna, Zbigniew, "Inception," 2016, [Online], https://pytorch.org/vision/stable/generated/torchvision.models.inception_v3.html#torchvision.models.inception_v3, Último Acceso en 2022-05-02.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá