



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



TFG del Grado en Ingeniería
Informática

**Sign2Text - Transcripción de
lenguaje de signos (a nivel de
palabra) mediante DL**



Presentado por Iván Ruiz Gázquez
en Universidad de Burgos — 2 de julio de 2022
Tutores: Dr. Daniel Urda Muñoz y Dr. Bruno
Baruque Zanon



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. nombre tutor, profesor del departamento de nombre departamento, área de nombre área.

Expone:

Que el alumno D. Iván Ruiz Gázquez, con DNI dni, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 2 de julio de 2022

Vº. Bº. del Tutor:

Vº. Bº. del co-tutor:

D. nombre tutor

D. nombre co-tutor

Resumen

En este primer apartado se hace una **breve** presentación del tema que se aborda en el proyecto.

Descriptores

Palabras separadas por comas que identifiquen el contenido del proyecto Ej: servidor web, buscador de vuelos, android ...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	vi
Introducción	1
Objetivos del proyecto	3
Conceptos teóricos	7
3.1. Machine Learning	7
3.2. Supervised Learning	15
3.3. Deep Learning	23
Técnicas y herramientas	31
4.1. Lenguajes	31
4.2. Metodologías	32
4.3. Herramientas	34
4.4. Miscelania	46
Aspectos relevantes del desarrollo del proyecto	47
5.1. Carga de datos	47
5.2. Tratamiento de los datos	50
5.3. Red Neuronal ResNet	52
5.4. Estructuras de la redes convolucionales	53
5.5. Entrenamiento e hiperparametrización	55
5.6. Dos salidas en la misma red	57

5.7. Exportación del modelo	58
5.8. Compresión del modelo	59
5.9. Repasemos un experimento	60
Trabajos relacionados	63
Conclusiones y Líneas de trabajo futuras	65
Bibliografía	67

Índice de figuras

3.1. Tipos de <i>machine learning</i> según datos, inferencia estadística y técnicas de aprendizaje	8
3.2. Ejemplo de datos antes y después del <i>clustering</i> para un algoritmo mutuamente excluyente <i>K-means</i> [48].	10
3.3. Visualización de la taxonomía de los métodos de aprendizaje semi-supervisado. En la rama correspondiente a los métodos basados en grafo podemos observar las distintas fases de clasificación [52]	15
3.4. Visualización de overfitting en un entrenamiento de una CNN (red neuronal convolucional). Podemos observar en rojo el punto exacto en el que la red comienza a sobre-ajustarse. Este es el momento en el que debemos parar el entranamiento.	17
3.5. Estructura de una red RNN y CNN	24
3.6. Estructura de una CNN. Figura de <i>Le Cun et al.</i> [30]	26
3.7. Ejemplo de convolución con <i>kernel 3x3</i> sin <i>strading</i> ni <i>padding</i>	27
4.1. Ejemplo de <i>transform</i> en PyTorch	36
4.2. Estructura de un modelo con CNNs en formato ONNX . Visualización con <i>Netron</i>	37
4.3. Ejemplo de código de detección de la anchura de un botón con <i>Svelte</i> (izquierda) y <i>HTML</i> (derecha)	44
5.1. Estructura de carpetas necesaria para la carga de datos con PyTorch	49
5.2. Estructura de una CNN exportada con <i>Netron</i>	54
5.3. Estructura de una CNN con doble salida exportada con <i>Netron</i>	55
5.4. La red es capaz de adivinar un signo hecho por un dibujo animado	58
5.5. Modelo exportado antes y después de cuantizar con ONNXRuntime	60
5.6. Concatenación de frames con la intención de representar un video (gesto es «Before»). Imágenes del dataset <i>WLASL</i> [31]	61

Índice de tablas

4.1. Diferencias entre Tensorflow y PyTorch	38
5.1. Accuracy de una red ResNet con un dataset pequeño de imágenes de animales [4]	52
5.2. Accuracy de una red ResNet con nuestro dataset original con 10 etiquetas, 30 videos por etiqueta, 1 solo frame por video	53

Introducción

Descripción del contenido del trabajo y del estructura de la memoria y del resto de materiales entregados.

Objetivos del proyecto

La principal motivación del proyecto es:

Mejorar la interacción entre personas que necesitan comunicación por signos y personas que no conocen el lenguaje de signos, intentando aumentar la calidad de vida de las primeras.

Objetivos teóricos

1. Creación de un modelo de DL¹ capaz de clasificar ASL² a nivel palabra.
2. Poner en producción un método para que los usuarios puedan probar el modelo de forma libre y transparente.
3. Generar una API de código abierto para que otros desarrolladores puedan crear plataformas de cliente, con el objeto de aumentar la audiencia y alcance del proyecto.
4. Liberar y contenedorizar el modelo para su libre distribución.
5. Creación de herramientas que permitan el tratamiento de datos y la transformación entre distintos formatos³.
6. Estudio del comportamiento, estructura e hiperparametrización de redes neuronales convolucionales.

¹*Deep Learning*

²*American Sign Language*

³i.e.: extracción de fotogramas de un video, concatenación de imágenes

7. Aprender a fondo el uso de PyTorch[32], un *framework* para acelerar la creación y prototipado de redes neuronales.
8. Exportación del modelo entrenado a un formato estándar que facilite la compatibilidad con el mayor número de librerías en distintos lenguajes de programación.
9. Mantenimiento de una *codebase* que favorezca la integración continua (*linting*⁴, *formatting*⁵ y *type-checking*⁶) de código, siguiendo así los estándares en contribuciones *Open Source*.

Estos objetivos representan en general la filosofía y los objetivos teóricos del proyecto. Si entramos más en detalle sobre los aspectos técnicos y las *features* que se esperan obtener al finalizar el proyecto, obtenemos los siguientes puntos:

***Features* esperadas**

- Debemos ser capaces de inferir resultados del modelo entrenado a tiempo real.
- Se desarrollará un modelo fácilmente escalable y adaptable a distintos dataset de cualquier tamaño.⁷
- Se realizará una fase de *data augmentation* sobre los datos iniciales.
- Se implementarán distintas redes neuronales para los distintos formatos de datos (imagen y video). Deben ser fácilmente refactorizables y mantenibles.
- Cada vez que se estudie el uso de un nuevo formato de dataset, se creará una nueva red, manteniendo la usabilidad de la anterior intacta.
- A lo largo de las pruebas y entrenamientos de la red, vamos a probar distintos *schedulers*, *optimizers* y *criteria*⁸ buscando el que mejor se adecúe al formato de los datos y la estructura de la red.

⁴Voz ingl. Estudio de limpieza, orden, calidad y redundancia

⁵Formatear: Correcta estructura y legibilidad

⁶Comprobación de tipados de variables y funciones

⁷El formato de los datasets debe ser el mismo. Un modelo entrenado para clasificación de imágenes no podrá clasificar en formato video o audio.

⁸Funciones de cálculo de pérdida

- Se mantendrán unas estadísticas a tiempo real de la fase de *training* y *test* mediante el uso de **Tensorboard**, una herramienta analítica que permite mantener un *log* de imágenes generadas en la ejecución; así como gráficas de costes y *accuracies*.

Conceptos teóricos

Este proyecto es un proyecto de *machine learning*. Dentro del *machine learning*, es de aprendizaje supervisado. Más concretamente intenta resolver un problema de clasificación mediante *deep learning*. Las redes neuronales aplicadas para la clasificación serán CNN (*Convolutional Neural Networks*), y más concretamente *ResNets*, una arquitectura sobre las CNN. Veamos en detalle todos estos terminos en los siguientes apartados.

3.1. Machine Learning

Según Tom Mitchell [35], un programa de ordenador aprende de una experiencia E con respecto a alguna tarea T si su medida de desempeño P (del inglés *performance*) mejora con E .

El uso del machine learning se origina en la búsqueda de soluciones a problemas que no podemos resolver programáticamente. Determinar las soluciones se centra en la recogida y análisis de datos⁹ con la finalidad de buscar relaciones entre ellos.

Para aclarar este concepto, veamos un ejemplo [5]: Imaginemos que debemos estimar el precio de un coche usado pero no tenemos la fórmula exacta para valorar su estado. Lo que si sabemos es que el precio del coche aumenta y disminuye dependiendo de sus propiedades, como la marca, el kilometraje, o el desgaste. No sabemos tampoco cuales de las propiedades

⁹A lo largo del documento nos referiremos a los datos indistintamente como «datos» o «instancias»

tiene más importancia. Sabemos seguro que cuantos más kilómetros tenga el coche, menor será su precio, pero no sabemos a que escala ocurre esto.

Para determinar la solución necesitamos estudiar el precio de coches en el mercado y anotar sus características. Estos datos son los que daremos a nuestro programa para que busque las relaciones entre propiedades e indique el precio óptimo de los coches.

Saliendo del ejemplo, el machine learning se divide en distintos tipos según la cantidad de datos (o supervisión). Hay más formas de dividir el machine learning, como puedes ver en 3.1 (nombraremos alguna de ella a lo largo de los siguientes apartados), pero nos vamos a centrar en *supervised*, *unsupervised* y *semi-supervised learning*.

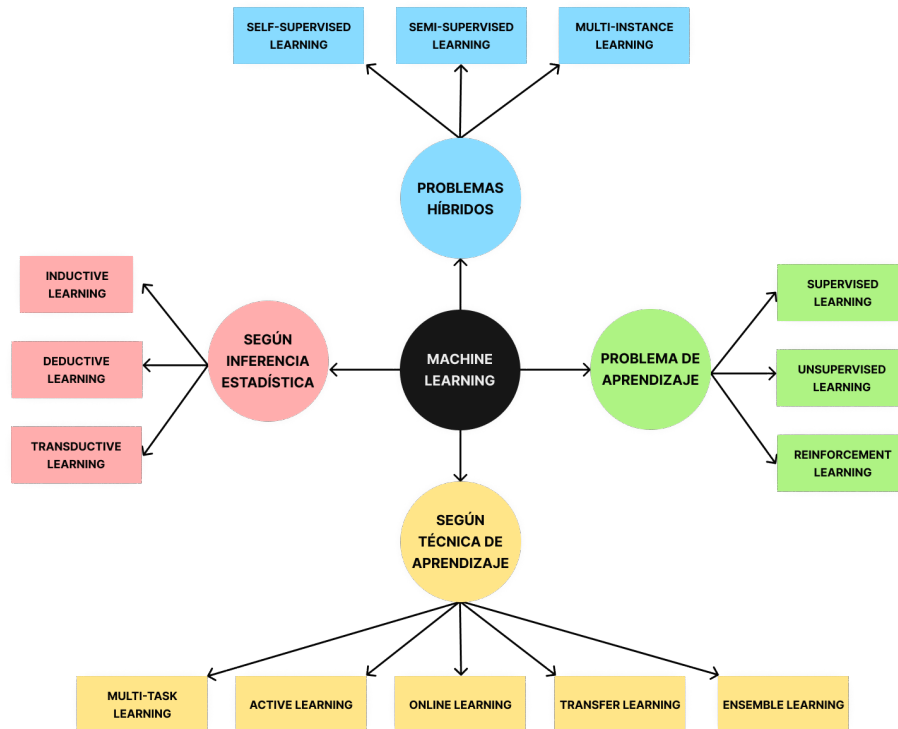


Figura 3.1: Tipos de *machine learning* según datos, inferencia estadística y técnicas de aprendizaje

Unsupervised Learning

El *unsupervised learning* o aprendizaje no supervisado, tiene su principal característica en que es capaz de detectar las relaciones entre los

datos sin ayuda de etiquetas o respuestas externas. Si volvemos al ejemplo de los coches, seríamos capaces de determinar el precio de venta sin necesidad de recibir datos de coches reales.

La técnica principal en el aprendizaje no supervisado es el agrupamiento de los datos (o *clustering*) según la similaridad de sus características. Pero hay más técnicas.

Clustering

El clustering consiste en el agrupamiento de datos no etiquetados basándonos en sus similitudes o diferencias. Los algoritmos de *clustering* son usados para procesar datos crudos sin alteraciones en grupos representados por patrones o estructuras de información. Estos algoritmos pueden clasificar en:

- **Mutualmente excluyentes:** En estos algoritmos se estipula que una instancia de datos solo puede existir en un y solo un *cluster* (o grupo). Este tipo de *clustering* también se denomina «*hard*» *clustering*. Un ejemplo de algoritmo mutualmente excluyente es el algoritmo de *K-means*. En *K-means*, la «K» indica el número de *clusters* basándonos en su centroide¹⁰. Los puntos, representando a un dato, caeran en el grupo con el centroide más cercano. Un valor de *K-means* mayor indicará mayor número de agrupamientos.

¹⁰O centro geométrico, es la posición media aritmética entre todos los puntos de una figura

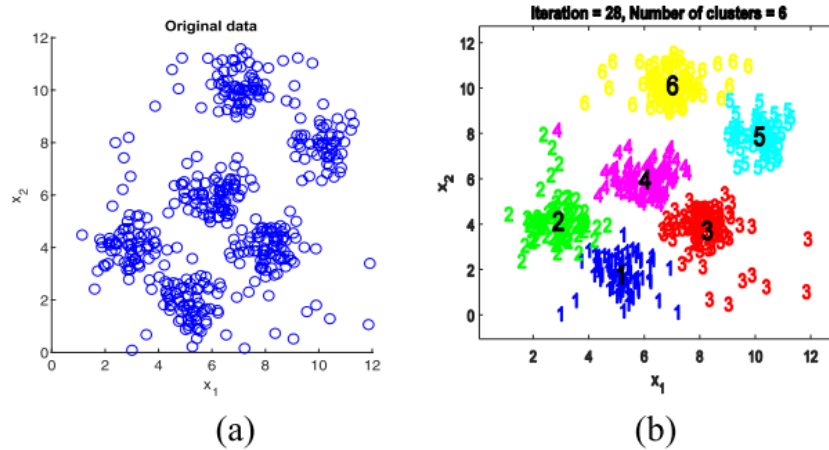


Figura 3.2: Ejemplo de datos antes y después del *clustering* para un algoritmo mutuamente excluyente *K-means* [48].

- **Superpuestos:** En el caso de los algoritmos superpuestos, la diferencia es que un dato puede pertenecer a más de un *cluster* de forma simultánea con distinto grado de pertenencia. Un ejemplo de algoritmo superpuesto es el «soft» *K-means*.
- **Jerárquicos:** También conocido como HCA (*hierarchical cluster analysis*), es un algoritmo no supervisado que se puede categorizar de dos modos, en aglomerativo o divisivo. El primero funciona con acercamiento «bottom-up». Los datos comienzan de forma separada y se van uniendo de forma iterativa según similitud hasta conseguir un único cluster. La similaridad se puede medir de las siguientes formas:
 1. Método de *Ward*: La distancia entre dos clusters se define por el aumento de la distancia cuadrática después de que los clusters sean fusionados.
 2. Promedio de distancias: Este método usa la distancia media entre dos puntos en cada *cluster*.
 3. Distancia máxima: La distancia usada es la distancia mayor entre dos puntos en distintos clusters.
 4. Distancia mínima: Se define por la distancia mínima entre dos puntos de dos clusters diferentes.

La medida más común a la hora de calcular estas distancias es la distancia euclídea, aunque otra muy común en la literatura es la distancia

Manhattan. Por otro lado, el acercamiento divisivo, o «*top-down*» funciona dividiendo un único *cluster* basándose en las diferencias entre instancias (puntos del cluster). Visualmente se parece a un dendrograma¹¹.

- **Probabilísticos:** En un algoritmo probabilístico, los puntos se agrupan basándose en la probabilidad de que pertenezcan a una distribución paramétrica¹². El GMM (*Gaussian Mixture Model*) es el método más usado en *clustering* probabilístico.
 - *Gaussian Mixture Model*: Este algoritmo se clasifica como mixto debido a que se compone de un número no especificado de funciones de densidad probabilística. Normalmente se encargará de comprobar si una instancia es parte de una distribución Gaussiana o normal. El algoritmo más usado para determinar la distribución es el E-M (Expectation-Maximization) [14].

Reglas de asociación

Las reglas de asociación es un método que busca buscar las relaciones entre variables de un *dataset*. Suelen ser usados en *marketing*, sistemas de recomendación, relación entre productos, estudio de hábitos de consumo [38], entre otros. El algoritmo más usado en reglas de asociación es el algoritmo Apriori.

- **Algoritmo Apriori:** Este algoritmo permite encontrar de forma frecuente «conjuntos de elementos frecuentes». Estos conjuntos sirven para generar reglas de asociación que permiten extender los sets a sets de mayor tamaño.

¹¹Representación gráfica de datos en forma de árbol para organizar datos en categorías y subcategorías hasta alcanzar el nivel de detalle deseado

¹²La estadística paramétrica es una rama de la estadística con la que se deciden valores basándose en distribuciones desconocidas. Estas se determinan según un número finito de parámetros.

Algoritmo 1: Algoritmo Apriori

Input: Para un dataset T , un umbral de soporte ε y un conjunto de datos C_k para el nivel k

```

1  $L_1 \leftarrow findFrequentOneItemsets(T)$ 
2  $k \leftarrow 2$ 
3 while  $L_{k-1}$  not empty do
    /* Generación de Apriori */
4    $C_k \leftarrow$  empty list
5   forall  $p \subseteq L, q \subseteq L$  do
6      $c \leftarrow p \cup q_{k-1}$ 
7     forall  $u \in L$  do
8       if  $u \subseteq c$  then
9          $C_k \leftarrow C_k \cup \{c\}$ 
10      end if
11    end forall
12  end forall
    /* Cálculo de candidatos */
13  for  $t \in T$  do
14     $D_t \leftarrow \{c \in C_k : c \subseteq t\}$ 
15    for  $c \in D_t$  do
16       $count[c] \leftarrow count[c] + 1$ 
17    end for
18  end for
19   $k \leftarrow k + 1$ 
20   $L_k \leftarrow \{c \in C_k : count[c] \geq \varepsilon\}$ 
21 end while
22 return  $Union(L_k)$ 

```

Reducción de la dimensionalidad

Normalmente cuando disponemos de una mayor cantidad de datos, tendemos a obtener mejores soluciones, aunque esto tiene impacto en el rendimiento de nuestros algoritmos y puede dificultarnos comprender de forma completa nuestro *dataset*. La reducción de la dimensionalidad se usa cuando el número de características o dimensiones de un *dataset* es muy grande. Reduce el número de instancias a un tamaño manejable manteniendo la integridad de los datos lo máximo posible. Algunos métodos de reducción

de la dimensionalidad son [50][19]: PCA (del inglés *Principal Component Analysis*), SVD (*Singular Value Decomposition*) y *Autoencoders*.

Semi-supervised Learning

El aprendizaje semi-supervisado se caracteriza por tener la capacidad de aprender de conjuntos de datos cuyas instancias no están completamente etiquetadas. En nuestro ejemplo 3.1 «cálculo de precio de un coche según sus propiedades», tendríamos precios reales solo para algunos de los coches, mientras que otros no tendrían una etiqueta establecida.

La característica principal que diferencia el aprendizaje semi-supervisado del aprendizaje no supervisado y del aprendizaje supervisado (que veremos a continuación) es que la capacidad de aprender del algoritmo se nutre del hecho de que algunos datos no estén etiquetados. A la hora de intentar averiguar las etiquetas faltantes en base a las presentes, podemos observar comportamientos de aprendizaje que no se verían en los otros dos tipos de *machine learning*.

De este modo, no es útil solo como algoritmo de aprendizaje si no como algoritmo de generación de nuevos datos en aprendizaje supervisado [55]. Si la obtención de datos y etiquetas nos supone un proceso costoso y/o el acceso a los datos es limitado ¹³, podemos crear un algoritmo de aprendizaje semi-supervisado para completar la falta de datos.

Algunos de los métodos usados en aprendizaje semi-supervisado son los siguientes:

- Modelos generativos: Con este modelo se busca estimar $p(x | y)$, la distribución de los puntos pertenecientes a cada clase. La probabilidad $p(x | y)$ de que un punto x pertenezca a una etiqueta y es proporcional a $p(x | y) \cdot p(y)$ por la Regla de Bayes.

Extrayéndolo de [55], los modelos generativos asumen que las distribuciones tienen la forma $p(x | y, \theta)$ para metrizada por el vector θ . Si esta suposición es incorrecta, los datos no etiquetados pueden incluso disminuir la capacidad de acierto de la solución. Pero, en el caso contrario, si la suposición es correcta, los datos etiquetados mejoraran el rendimiento del modelo.

¹³i.e.: La recopilación de datos médicos es algo compleja, ya que se debe tener permiso del paciente, así como hay enfermedades cuyos datos son muy escasos y difíciles de conseguir.

Los datos no etiquetados se suelen distribuir mediante una distribución Gaussiana mixta. La distribución de probabilidad conjunta se puede escribir como $p(x, y | \theta) = p(y | \theta) \cdot p(x | \theta)$ mediante el teorema de probabilidad compuesta. Cada vector θ se asocia con una función de decisión:

$$f_{\theta}(x) = \operatorname{argmax}_x p(y | x, \theta)$$

El parámetro se elige entonces basándonos en la adaptación a los datos etiquetados y no etiquetados, usamos un peso de balanceo λ :

$$\operatorname{argmax}_{\theta} \left(\log p \left(\{x_i, y_i\}_{i=1}^l | \theta \right) + \lambda \log p \left(\{x_i\}_{i=l+1}^{l+u} | \theta \right) \right)$$

- *Support Vector Machines* (SVM) : Según Van Engelen *et al.*[52] un SVM es un clasificador que intenta buscar la frontera de decisión que maximice el margen, que a su vez se define como la distancia entre la frontera de decisión y los puntos cercanos a ella¹⁴. En el escenario del aprendizaje semi-supervisado, un objeto adicional gana importancia: también queremos minimizar el número de datos «no etiquetados» que viola el margen. Como no podemos saber la etiqueta de los datos «no etiquetados», aquellos puntos que violan el margen son penalizados basándonos en su distancia a la frontera de decisión.
- Modelos basados en gráficos: Este modelo asume un grafo $G = V, E$ tal que los vértices V son las instancias de datos (etiquetados y no etiquetados) y E las aristas no dirigidas que conectan las instancias i, j con un peso w_{ij} . La construcción de un grafo se cumple en los siguientes pasos [49][52] 3.3:
 1. **Construcción del grafo:** Se suele asumir una instanciación inicial del grafo aleatoria
 2. **Weighting:** Cálculo de pesos de los datos etiquetados para la fase de inferencia.
 3. **Inferencia de etiquetas:** La tarea a cumplir aquí es propagar información de las instancias etiquetadas a las «no etiquetadas» incorporando la estructura de grafo creada en el paso anterior.

¹⁴A veces el margen también se refiere a la zona delimitada por la frontera de decisión en la que caen los puntos

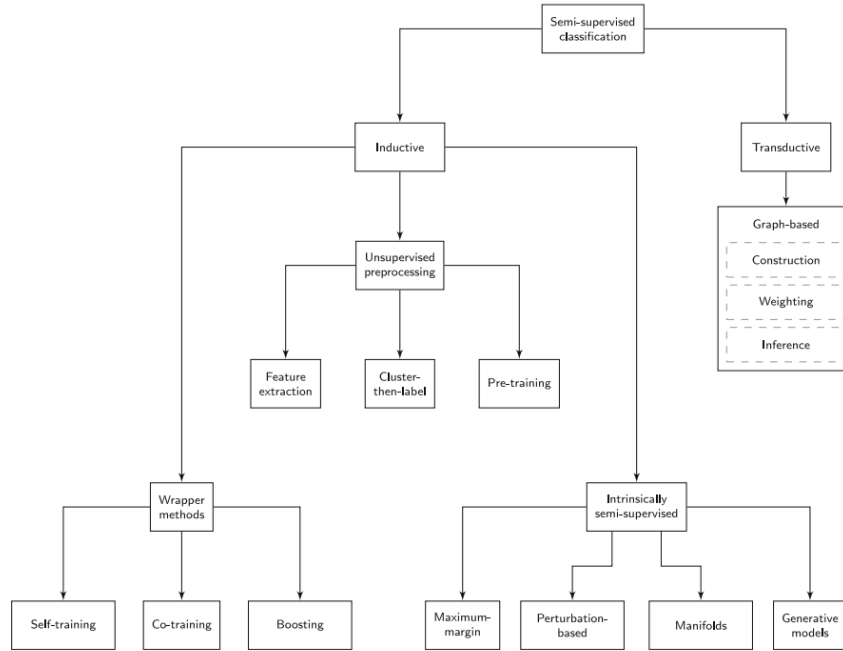


Figura 3.3: Visualización de la taxonomía de los métodos de aprendizaje semi-supervisado. En la rama correspondiente a los métodos basados en grafo podemos observar las distintas fases de clasificación [52]

3.2. Supervised Learning

Poco hace falta una definición de aprendizaje supervisado después de haber revisado aprendizaje no supervisado y aprendizaje semi-supervisado, pero por mantener el aislamiento de los apartados, diremos que: también llamado *classification*¹⁵ o *inductive learning* (recordemos fig. 3.1), el aprendizaje supervisado es aquel aprendizaje cuyas instancias de datos (usadas para que nuestro programa aprenda) están todas etiquetadas.

Según Christopher M. Bishop *et al.*[8], el objetivo último del aprendizaje supervisado es ser capaces de observar una variable t (*target*) junto con una variable de entrada x y predecir el valor de t para cualquier nuevo valor de x .

Para conseguir la predicción de nuevos datos debemos crear un modelo que va a ser «entrenado», «testado» y «validado». Para conse-

¹⁵A pesar que no solo trate problemas de clasificación

guir datos para las tres fases dividiremos nuestro *dataset* en tres grupos respectivamente.

Para que el modelo tenga un alto acierto en sus predicciones debe estar bien entrenado, por lo que la mayor fracción de los datos se destinará a esta fase¹⁶. Pero hay que tener cuidado, porque si asignamos un porcentaje demasiado alto de los datos a la fase de entrenamiento podemos generar un *overfitting* en el modelo.

El *overfitting* ocurre cuando un modelo se ha sobreentrenado con unos datos, por lo que a la entrada de nuevos datos (desconocidos) x , generará una predicción t sesgada a los datos de entrenamiento.

Para evitar esto, usamos otra fracción de los datos de entrada como fracción «de control». De este modo iremos comprobando la deriva de las predicciones entre los datos de entrenamiento y los datos (has adivinado) de validación.

Ahora podremos controlar la fase de entrenamiento y parar dicho entrenamiento, valga la redundancia, cuando detectemos este sobreajuste. Podemos ver esto en la figura 3.4. Gracias al *set* de validación, detectamos el momento en el que la red comienza a sobreajustarse. Esto no solo nos evita cometer errores en la fase de test, sino que mediante la eliminación de iteraciones inútiles sobre el modelo, disminuimos en gran cantidad el tiempo de entrenamiento.

¹⁶Normalmente un 70 % de los datos se asigna a la fase de entrenamiento, aunque depende mucho del problema específico al que nos enfrentemos

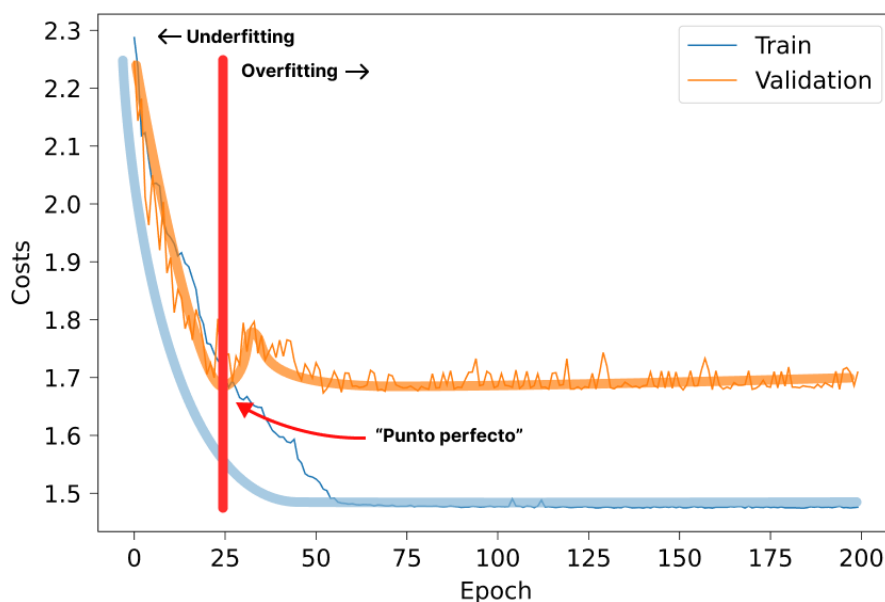


Figura 3.4: Visualización de overfitting en un entrenamiento de una CNN (red neuronal convolucional). Podemos observar en rojo el punto exacto en el que la red comienza a sobre-ajustarse. Este es el momento en el que debemos parar el entrenamiento.

En la imagen vemos que el valor del eje de ordenadas tenemos el valor de los costos sobre las iteraciones (*epochs* en inglés). Veremos más sobre esto en el apartado de *fine-tuning* e hiperparametrización de una red neuronal. Veamos ahora los distintos tipos de aprendizaje supervisado.

Regression

En la regresión, el objetivo es predecir un valor numérico y continuo. Uno de los usos principales de modelos de regresión es el relleno de espacios en los datos o la estimación de valores futuros. Por otro lado, la regresión también se puede utilizar para determinar relaciones casuales entre variables dependientes e independientes [33].

En los modelos de regresión, las variables independientes predicen a las variables dependientes. El análisis de regresión estima la variable dependiente y en base al rango de variables independientes x . En cuanto a los tipos de regresión, puede ser simple o múltiple.

- **Regresión simple:** En las regresiones simples solo tenemos una variable independiente y . Se define la dependencia de la variable como $y = \beta_0 + \beta_1 x + \epsilon$. De forma gráfica, la regresión simple se muestra como una línea entre los puntos cuyo objetivo es minimizar el error cuadrático de las distancias entre dicha línea y los puntos. Es común que en estos modelos tengamos puntos atípicos que se sitúen lejos de la mayoría. Estos puntos se denominan *Outliers* y al distanciarse del resto, provocan desviaciones importantes en la regresión [46].
- **Regresión múltiple:** El objetivo de los modelos múltiples es la predicción del conjunto de variables independientes y según el conjunto de variables dependiente x . El modelo básico es $y = \beta_0 + \beta_1 x_1 + \dots + \beta_m x_m + \epsilon$. La fórmula para determinar la matriz es:

$$\hat{\beta} = (X^T X)^{-1} X^T y \quad \text{donde}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix}, X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}, Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}$$

Classification

El segundo y último tipo de tipo de aprendizaje supervisado es también el más familiar. La clasificación consiste en etiquetar una variable discreta y dada una variable de entrada x . Según el número de etiquetas disponibles para la clasificación hablamos de clasificación binaria o multi-clase.

Aunque no listamos todos los métodos, a continuación se pueden observar los más ampliamente usados[41]:

1. **Árboles de decisión:** Los árboles de decisión son una forma de representar reglas sobre los datos de forma jerárquica con una estructura secuencial recursiva que particiona los datos. Los árboles de decisión se usan para aproximar funciones discretas. Un nodo hoja indica el valor de la etiqueta y , mientras que un nodo de decisión especifica una operación a realizar. La evaluación comienza evaluando el nodo raíz (un nodo de decisión) y moviendonos hasta encontrarnos con un

nodo hoja, con la etiqueta buscada. El algoritmo más común en la construcción de árboles de decisión es una extensión del algoritmo propuesto por J. Quinlan en 1993 *et al.*[39]: ID3 (*Iterative Dichotomiser 3*). El árbol se construye *top-down* de forma recursiva con la técnica de «divide y vencerás». Puedes ver el pseudocódigo a continuación.

Algoritmo 2: Algoritmo ID3 - TreeBuilding(X, Y)

Input: Para un set de entradas de entrenamiento X y una lista de etiquetas correspondientes Y

```

1   $N \leftarrow$  Nodo raíz
2  if  $\forall$  muestra = clase  $C$  then
3      | etiqueta  $N = C$ 
4      | return
5  end if
6  if  $Y = \text{vacío}$  then
7      | etiqueta  $N =$  clase más común  $C$  en  $X$  (voto mayoritario)
8      | return
9  end if
10  $y \leftarrow$  El atributo que mejor clasifica ejemplos donde  $y \in Y$ 
11 etiqueta  $N \leftarrow y$ 
12 forall  $v \in y$  do
13     | Haz crecer una rama desde  $N$  con condición  $y = v$ 
14     |  $X_v \leftarrow$  subset de instancias en  $Y$  con  $y = v$ 
15     | if  $X_v = \text{vacío}$  then
16     |     | añade un nodo hoja etiquetado con la clase más común en  $X$ 
17     |     | end if
18     |     | else
19     |     |     | añade el nodo generado por  $\text{TreeBuilding}(X_v, Y - v)$ 
20     |     |     | end if
21 end forall
```

2. **Bagging y boosting:** Es una técnica de ensamblado que combina varios métodos de *machine learning* en un único modelo predictivo en orden de disminuir la varianza (*bagging*) o el *bias*(*boosting*).

El *bagging* (*Bootstrap Aggregating*) comienza creando muestras aleatorias del *set* de entrenamiento. Tras esto, construye un clasificador para cada muestra. Finalmente, se combinan los resultados de los clasificadores y comienza una votación de mayoría. Al construir estos modelos al mismo tiempo, consideramos a Bagging un método de

ensamblado paralelo. Al aumentar el tamaño del *set* de entrenamiento, se disminuye la varianza del modelo final [6].

El *boosting* por otro lado se compone de dos pasos. En el primero usa *subsets* del *set* de entrenamiento para producir modelos con rendimiento medio. Tras esto estimula el rendimiento combinándolos usando el voto mayoritario. En el *boosting*, la creación de *subsets* no es aleatoria y depende del rendimiento de los modelos anteriores.

La principal diferencia entonces entre los dos, es que *bagging* aprende el *dataset* completo de forma paralela mientras que *boosting* aprende usando los resultados de los modelos anteriores. *Boosting* muestra una mejor *accuracy* que *bagging* [40], pero también tiene a producir *overfitting* en los modelos.

3. **Random Forest:** El *random forest* también se puede usar en tareas de regresión. El principio que se sigue es que un conjunto de clasificadores «débiles» se puede unir para crear un clasificador «fuerte». En este caso tenemos un *ensemble* de árboles de decisión que funciona mejor que *bagging* y *boosting* tanto en *accuracy* como en *overfitting*. Cada árbol del *ensemble* tiene un *set* de entrenamiento distinto. Debemos tener en cuenta que si necesitamos una descripción de las relaciones existentes entre los datos, los *random forest* no son la mejor opción, ya que al tener un conjunto de árboles, se hace difícil la comprensión de la estructura general.
4. **SVM (*Support Vector Machines*):** Los SVM también están presentes en la regresión. En el caso de la clasificación, tenemos un conjunto de puntos en un espacio de dimension N . Cada *set* perteneciendo a una de las posibles clases, el SVM busca los planos separados que maximicen los márgenes entre los *sets* de datos. Ya hemos visto en 22 un caso de SVM en aprendizaje semi-supervisado, solo que en este caso funciona distinto. A la hora de clasificar un dato, separamos dos planos y calculamos el error cuadrático de los puntos a dichos planos. En algunas ocasiones, los planos pueden no ser linealmente separables; en estos casos, debemos aumentar el número de dimensiones hasta que lo sean. Para esto se usa lo que se denomina función *kernel* [10]

$$K(x, y) = \langle f(x), f(y) \rangle$$

donde x e y son las entradas de dimensión n , y f es un mapa de paso de dimensión- n a dimensión- m . Normalmente nos encontraremos con que m es un número mucho mayor a n .

5. **Naive Bayes:** Este clasificador probabilístico se basa en el teorema de Bayes y asume que todas las características están independientemente condicionadas por la etiqueta de la clase [36]. Aunque esto no suele ocurrir, el modelo resultante funciona sorprendentemente bien, a veces compitiendo con técnicas mucho más sofisticadas [42]. Este modelo ha probado ser muy útil en tareas de clasificación de texto y diagnóstico médico, entre otros. En [15] se prueba que *naive Bayes* funciona de forma óptima en problemas con un alto grado de dependencia entre características.
6. **Redes Neuronales:** Por último, tenemos las redes neuronales. Las redes neuronales ocupan la mayoría de la literatura sobre inteligencia artificial y *machine learning* en la actualidad. Una red neuronal es un conjunto de neuronas artificiales colocadas en capas. Cada capa tiene la tarea de extraer características de los datos de entrada, actualizar sus pesos y «pasar» información a las siguientes capas.

Una neurona es una función f_j con una entrada $x = (x_1, \dots, x_d)$ y un vector de pesos $w_j = (w_{j,1}, \dots, w_{j,d})$, junto con un *bias* b_j y asociado a una función de activación ϕ [7]. La fórmula completa de esta función es

$$y_j = f_j(x) = \phi(\langle w_j, x \rangle + b_j)$$

Existen numerosas funciones de activación en una red neuronal, algunas son:

- Función identidad o lineal

$$\phi(x) = x$$

- *Binary Step*

$$\phi(x) = \begin{cases} 0 & \text{para } x < 0 \\ 1 & \text{para } x \geq 0 \end{cases}$$

- Función sigmoide

$$\phi(x) = \frac{1}{1 + \exp(-x)}$$

- Tangente hiperbólica

$$\phi(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1}$$

- **ReLU** (*Rectified Linear Unit*)

$$\phi(x) = \max(0, x)$$

- **Leaky ReLU**

$$\phi(x) = \max(0, 1 \cdot x, x)$$

- **ELU** (*Exponential Linear Unit*)

$$\phi(x) = \begin{cases} x & \text{para } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{para } x < 0 \end{cases}$$

- **Función softmax**: descrita como una combinación de múltiples sigmoides, calculamos su salida con

$$\phi(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

siendo x el conjunto de las salidas de una neurona en un problema multiclase.

Tanto la función de *binary step* como la función de identidad (ambas funciones lineales) no permiten *backpropagation* ya que la derivada de la función es una constante y no tiene relación alguna con la variable de entrada x (no aporta datos adicionales) [24].

Por otro lado, en regresión se usan funciones de activación lineales, mientras que en clasificación, usaríamos sigmoides en clasificación binaria, o softmax en clasificación multietiqueta.

Una vez tenemos nuestra función de activación elegida, podemos calcular la salida de la red dada una variable de entrada x . Y una vez calculada la salida de la red, podemos comparar el resultado y con la etiqueta de *ground truth* correspondiente con dicha entrada. Según sea igual o no, modificaremos el vector de pesos de nuestra neurona.

Repetiremos este proceso hasta que la *accuracy* de la neurona (o la red de neuronas) sea superior al *threshold* buscado [41].

Podemos medir el *accuracy* de una neurona dividiendo el número de predicciones correctas entre el número total de predicciones [18]. Pero el *accuracy* no es el único método de medida de acierto en las redes neuronales. También tenemos la precisión, el *recall*, el *F-measure*, la especificidad, el *fallout* o la Coeficiente de Correlación de Mathew (MCC) [11].

Con

$tp = \text{true positives}$

$tn = \text{true negatives}$

$fp = \text{false positives}$

$fn = \text{false negatives}$

- Accuracy: $Acc = \frac{tp+tn}{tp+tn+fp+fn}$
- Precisión: $P = \frac{tp}{tp+fp}$
- Recall: $R = \frac{tp}{tp+fn}$
- F-measure: $F = \frac{2P \cdot R}{P+R}$
- Especificidad: $S_p = \frac{tn}{tn+fp}$
- Fallout: $\frac{fp}{tn+fp}$
- Coeficiente de Correlación de Mathew: $MCC = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp+fp)(tn+fn)(tp+fn)(tn+fp)}}$

3.3. Deep Learning

Hemos visto en el último apartado las características de una red neuronal de forma individual: como se compone, que tipos de funciones de activación tenemos y como podemos evaluar y medir la salida de una neurona.

Las redes neuronales más básicas: FFNN (*Feed Forward Neural Netowrks*) y Perceptrones, se componen únicamente de una capa de entrada y salida, con ocasionalmente una o más capas ocultas entre ambas. El método para entrenar estas redes suele ser el *backpropagation*.

El término *backpropagation* viene tras ser empleado por *Rosenblatt* (1962) [9] a la hora de intentar generalizar el entrenamiento de un perceptrón a múltiples capas ocultas. No fué muy exitoso hasta que *Rumelhart* y *Williams* (1986) [45] extendieron la idea y la hicieron popular entre investigadores [44]. La idea del *backpropagation* es minimizar el error entre la salida correcta e incorrecta mediante el *gradient descent*. Para esto necesitamos calcular las derivadas parciales de nuestras funciones de activación ¹⁷. Calculamos la salida de la derivada en el *forward pass*, y con

¹⁷Recordemos que solo las funciones de activación no lineales pueden ser usadas para el *backpropagation*, ya que la derivada de una función lineal es una constante

el resultado obtenido, realizamos un *backward pass*, con el que propagamos y actualizamos los pesos de la red.

Esto que ahora nos parece básico, pero fué una «revolución» en su momento y es la base principal del *deep learning*. Pero, el *deep learning* no solo trata el aprendizaje supervisado, (también tenemos *deep reinforcement learning*, *unsupervised learning*¹⁸ y *semi-supervised learning*). Por simplificar vamos a tratar solo redes neuronales de aprendizaje supervisado o DNN (*Deep Neural Networks*). Una lista extensa de redes neuronales se puede ver en [26].

Las *deep neural networks* (DNN) se clasifican en dos tipos principales: Redes Neuronales Recurrentes (RNN) y Redes Neuronales Convolucionales (CNN). Se diferencia de las redes neuronales artificiales «clásicas» (ANR)¹⁹ en que en vez de tener una capa de entrada seguida de una o más capas escondidas, con una capa de salida a continuación; las DNN tienen más de una capa entre la salida y la entrada, de ahí que se denominen profundas [26].

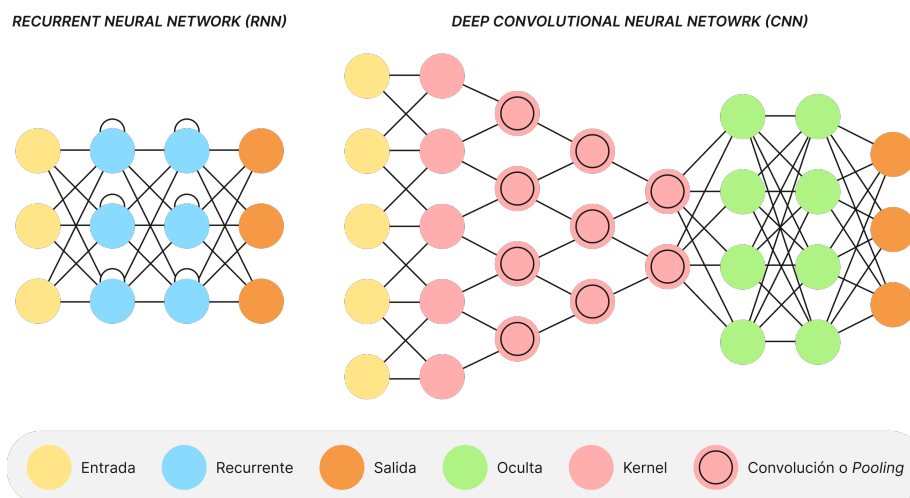


Figura 3.5: Estructura de una red RNN y CNN

En la figura 3.5, se puede observar la estructura de los dos tipos mencionados. La red recurrente debe su nombre a las neuronas que están conectadas consigo mismas (neuronas en azul).

¹⁸Una de las redes neuronales en *unsupervised learning* más habladas en la literatura son las redes GAN (*Generative Adversarial Neural Networks*). Estas redes usan la estructura de discriminador-generador [20] para extraer información de los datos de entrada

¹⁹Como FNNN o perceptrón

Las redes neuronales convolucionales son más complejas y su particularidad, como veremos más en profundidad en el siguiente apartado, es la extracción de características y reducción de la dimensionalidad. Vemos que la capa de entrada (en amarillo) tiene 5 neuronas, y que en las siguientes capas, el número de neuronas se va reduciendo hasta llegar a dos en la capa justo anterior a la oculta (en verde). Esto se produce mediante la convolución o el *pooling*. Vamos a estudiar estas dos redes y sus términos con un poco de profundidad a continuación.

- **RNN**: Las Redes Recurrentes son redes que tuvieron su auge en investigación en los años 90. *Fausett* (1994) [17] las define como redes neuronales con conexiones *feedback* (retroalimentación). Ejemplos de estas redes son las redes BAM (*Bidirectional associative memory*), redes *Hopfield* y máquinas de *Boltzmann* [34].

La característica principal de las redes recurrentes es que tienen un estado que mantienen entre distintas iteraciones de la red. Las neuronas no solo reciben información de la capa anterior si no que también reciben información de si mismas y de la iteración anterior [16]. Por esta razón, el orden con el que «alimentamos» datos a la red es importante y puede cambiar la salida generada de forma radical. Esta característica las hace muy útiles con formatos de datos como video o audio ²⁰. Su principal aplicación es el avance y completado de información, como por ejemplo el autocompletado de texto. En [43] podemos ver un ejemplo de una RNN aplicada para enseñar a la red a contar.

- **CNN**: Y llegamos a las redes convolucionales. Introducidas por *Le Cun* (1998) [28], estas redes son uno de los tópicos más hablados en la literatura actual en el mundo del *machine learning* y la inteligencia artificial. Estas redes se usan en muchísimas aplicaciones y dominios, sobre todo en procesamiento de video e imagen.

Las redes convolucionales tienen una estructura muy diferente a cualquier otra red neuronal (figura 3.6). Se componen de capas de convolución, capas de *subsampling* y capas «*fully-connected*».

²⁰Aunque también podemos tratar texto o imagen como concatenación de letras o píxeles en el tiempo

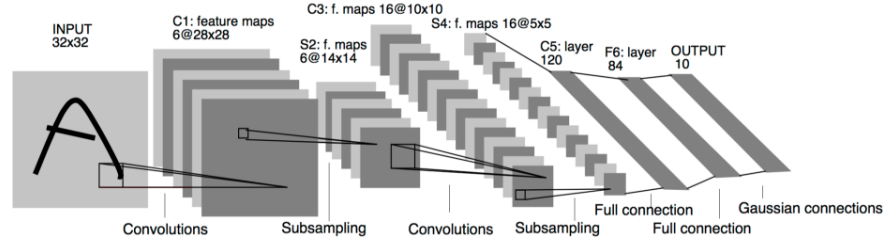


Figura 3.6: Estructura de una CNN. Figura de *Le Cun et al.*[30]

A continuación vamos a hablar de filtros o *kernels*, *pooling*, *padding*, *stride*, *ResNet*, entre otros términos.

1. **Extracción de características o convolución:** En esta fase se busca reducir la dimensionalidad de la entrada manteniendo las características espaciales de los datos. De esta forma no perdemos información y es más fácil procesarla en las siguientes capas. En redes neuronales tradicionales, transformaríamos una entrada (imagen o video) en un vector de dimensión uno. Al hacer esto perderíamos información de formas en la imagen, o de movimientos de objetos o personas en los videos. Por esa razón, la extracción de características de una imagen se hacía antes de forma manual en la fase de preparación de los datos.

Para eliminar esta fase manual, las CNN actúan directamente en matrices de datos multidimensionales (o tensores ²¹), en vez de actuar sobre vectores. Pero, ¿cómo extraemos información de un tensor?

Para extraer información de un tensor manteniendo sus características espaciales vamos a usar *kernels* o filtros. Un filtro es un tensor con las mismas dimensiones que la entrada, pero con tamaños mucho menores. i.e.: si tenemos como entrada una imagen de (200×200) píxeles, tendremos un filtro de dos dimensiones, por ejemplo, de (3×3) píxeles. Si en vez de una imagen, estamos clasificando un video con 10 frames $(10 \times 200 \times 200)$ tendremos un filtro que por ejemplo puede tener el tamaño $(2 \times 3 \times 3)$, siendo 2 el número de frames.

Una vez tenemos nuestro tensor de entrada y nuestro *kernel*, podemos iniciar el proceso de convolución. Para hacer esto, vamos a

²¹Un tensor es un objeto algebraico que representa un espacio vectorial. Puede mapear vectores y matrices multidimensionales [13]

«mover» nuestro *kernel* por nuestro tensor de entrada, generando un «mapa de caracterísitcas». Veamos un ejemplo:

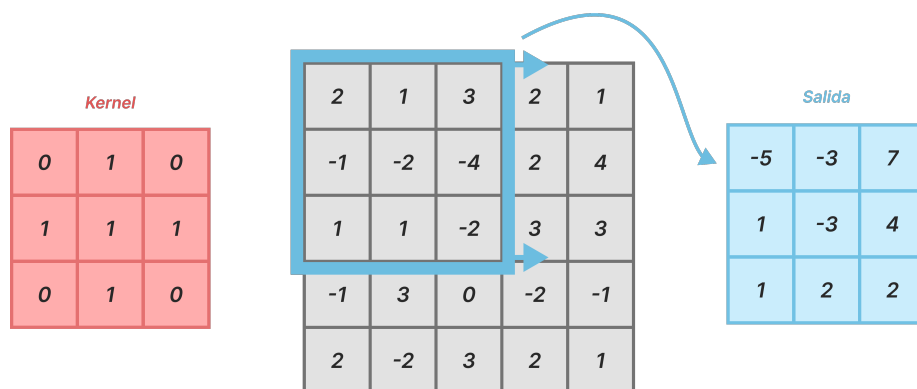


Figura 3.7: Ejemplo de convolución con *kernel* 3x3 sin *striding* ni *padding*

Vemos en la figura 3.7 que tenemos un tensor bidimensional (i.e. una imagen) de tamaño (5x5) con un *kernel* de tamaño (3x3). El mapa de caracterísitcas resultante tiene un tamaño de (3x3)²². En este caso no hemos tenido ningún problema y hemos podido «encajar» nuestro *kernel* en la matriz perfectamente. Esto no siempre es así, ya que si queremos adaptarnos a cualquier tipo de entrada, es posible que tengamos tamaños cuyo *kernel* no «encaje» de forma perfecta. Si nuestra matriz fuera de (4x4) en vez de (5x5), tendríamos que para el tercer cálculo, nuestro *kernel* se sitúa fuera de la matriz en el lado derecho. Para solucionar este problema usamos *padding* o *stride*.

- El *padding* se usa para añadir información a los bordes de nuestro tensor y así mantener el tamaño de salida deseado. Podemos elegir añadirlo en todos los bordes o solo en algunos. El valor óptimo del *padding* es 0. Esto es porque toda información que añadamos a los bordes, «contamina» la información importante ubicada en el tensor de entrada [37].
- El *stride* por otro lado indica la cantidad de unidades que movemos nuestro *kernel* sobre nuestra entrada en cada iteración de la convolución. Valores comunes de *stride* son 2 o 3 [21]. Tenemos que tener cuidado aquí, porque con un

²²Da la casualidad de que el tamaño de la salida es igual al tamaño del kernel, pero no es la norma. Si tuviéramos una entrada de (5x7) obtendríamos una salida de (3x5)

stride demasiado alto, podemos saltarnos extracciones de características importantes de nuestro tensor.

Podemos calcular el tamaño de salida con la siguiente fórmula [37]:

$$\delta = \frac{D + 2p - K}{s} + 1$$

donde D es cada una de las dimensiones un vector de tamaño d con las dimensiones de nuestro tensor de entrada, K es un vector de tamaño d con las dimensiones del *kernel*, y p y s son los valores de *padding* y *stride* respectivamente. La salida será un número δ . Si el número es un entero, entonces la convolución será correcta. Así en nuestro caso $\delta = [\frac{5+2\cdot0-3}{1} + 1, \frac{5+2\cdot0-3}{1} + 1] = [3, 3]$. Como vemos, en el proceso de convolución, «alteramos» las dimensiones del tensor de entrada con la finalidad de extraer las características espaciales del mismo.

2. **Pooling:** Muy similar a la fase de convolución, el pooling reduce el tamaño del mapa de características. El objetivo es disminuir la capacidad computacional necesaria para procesar gran cantidades de datos [12]. Hay dos tipos de *pooling* principales: *Max Pooling* y *Average Pooling*. El primero devuelve el valor máximo de la «porción» del tensor cubierta por el *kernel* mientras que el segundo devuelve la media de todos los valores en dicha «porción». *Max pooling* funciona normalmente mejor que *average pooling* porque no solo lleva a cabo la labor de reducción de la dimensionalidad, sino que también actúa como mecanismo de eliminación de ruido.

23

3. **Capa *fully connected*:** Una vez hemos extraído las características necesarias de nuestro tensor de datos, hemos terminado con un tensor con dimensiones reducidas. Esta fase se ve muy bien gráficamente en la figura 3.5. Las capas de color rosa son las capas en las que realizamos la convolución y el *pooling*. El tensor resultante debe ahora sí, ser convertido en un vector (tensor de dimensión uno), ya que este es la entrada ahora de una capa *fully connected*. Este proceso se denomina *flatten*.

Las capas *fully connected* se asemejan mucho a estructuras tradicionales de una red neuronal, con la diferencia de que las neuronas

²³Denotar que lo importante del *kernel* en el *pooling* son sus dimensiones ya que no vamos a realizar operaciones con él

que forman esta capa aplican una transformación lineal del vector de entrada primero, seguido de una transformación «no lineal».

La salida de esta capa se calcula como

$$y_{jk}(x) = f \left(\sum_{i=1}^n w_{jk}x_i + w_{j0} \right)$$

donde w_0 es el *bias*, w nuestra matriz de pesos y x el vector de entrada.

El nombre de esta capa ²⁴ se debe a que las neuronas están totalmente conectadas entre si entre las distintas capas (no entre neuronas de la misma capa ni consigo mismas) [25].

La salida de la capa *fully conneted* será la salida que produce la función de activación no lineal que se aplica tras las funciones lineales. Esta función puede ser una función *softmax* para clasificación multiclase o puede ser una sigmoide para otro tipo de salida. Por ejemplo, en este proyecto, se ha usado una *fully connected* con *softmax* para clasificar los signos y una *fully connected* con sigmoide para extraer un vector de poses de gestos en un video. Puedes avanzar a la sección «Aspectos relevantes» para más información.

Existen numerosas arquitecturas de redes neuronales convolucionales que han demostrado ser muy útiles en la resolución de distintos problemas. Algunas son: ALEXNET [27], LENET[29], VGGNET [47], GOOGLNET [51], ZFNET [54] o RESNET [23].

Todas estas no son ni mucho menos todas las redes neuronales del campo del *deep learning*. Cada pocos meses se publica un *paper* que muestra estructuras nuevas con oportunidades futuras impresionantes. En la actualidad, otro de los modelos de *deep learning* que más atención (nunca mejor dicho) genera, son los modelos *transformers*, expuestos por primera vez en el famoso *paper* «*Attention is all you need*», parte de un libro de (2017) [53]. Estos modelos usan el llamado «mecanismo de atención» que da pesos diferentes a las distintas partes de nuestro dato de entrada (llamados *tokens*). Los principales campos en los que se usan los *transformers* son el campo del *natural language processing* (NLP) y el campo de *computer vision* (CV).

²⁴También se suele llamar capa densa

Nota del autor: Los *transformers* se alejan un poco del tema principal de este proyecto, pero invito a leer el fantástico paper y descubrir sobre los modelos que usan esta red y que están revolucionando el mundo del *deep learning* (i.e.: GPT-3 o DALL-E)

Técnicas y herramientas

4.1. Lenguajes

Python

Hemos usado **Python** como lenguaje de programación en la creación del modelo. La versión utilizada ha sido la **3.9.8**. Se ha usado este lenguaje porque es el más común en *computer science* y es el lenguaje con más librerías de matemáticas, tensores y *machine learning* en general. En nuestro caso hemos usado **PyTorch**, una alternativa a **Tensorflow** con un paradigma más programático.

Por otro lado usar Python nos ha permitido crear scripts rápidos con muy poco código que han facilitado mucho el desarrollo y prueba de distintos *datasets*.

Y no solo eso, sino que por facilidad y reutilización de código, hemos creado el servidor y API con **FastAPI**, un *framework* para crear APIs, con rendimiento similar a **NodeJS** o **Go**.

HTML y CSS

A la hora de crear nuestro **front-end** para el usuario, hemos usado **HTML** para el marcado y **CSS** para los estilos. En este último lenguaje, hemos usado un *framework* de prototipado rápido llamado **Tailwind** (hablaremos de él un poco más adelante).

JavaScript

Con HTML y CSS podemos crear una *landing page* simple (aunque estilosa), pero ninguno es un lenguaje de programación. Con **JavaScript** podemos hacer peticiones HTTP, transformar el DOM en el navegador o renderizar la UI desde el servidor (SSR).

JavaScript es un lenguaje dinámicamente tipado y orientado a objetos que sigue el estandar **ECMAScript**. Es un lenguaje multi paradigma que soporta estilos de programación funcionales, basados en eventos e imperativos. El lenguaje nos ofrece distintas APIs para trabajar con el navegador: desde expresiones regulares, el DOM hasta *webworkers*, y una larga lista.

En el apartado Web veremos que librerías hemos usado para facilitar la creación de la UI así como la carga del modelo en este entorno. Allí hablaremos de dos frameworks muy interesantes para construcción de páginas estáticas y SSR: **Astro** y **Svelte**.

Markdown

Markdown es un lenguaje ligero de marcado que se ha usado para documentar la carpeta raíz o carpetas individuales del proyecto. En algunos de los **markdown** podemos observar guías de instalación o guías de uso, así como pasos para conseguir descargar el contenedor con el modelo, consumir de la API que hemos creado, entre otros.

L^AT_EX

Por último, LaTeX. Hemos usado LaTeX para generar esta memoria que estás leyendo, así como los anexos. Como nota personal, es la primera vez que lo hemos usado y nos ha sorprendido la velocidad con la que se escriben algunas estructuras de texto, ecuaciones y figuras. Desde luego seguiremos usándolo en el futuro.

4.2. Metodologías

Metodología SCRUM

La metodología SCRUM es un proceso de metodología ágil con el que se busca llevar a cabo un conjunto de tareas de forma colaborativa.

El objetivo es mejorar un producto en incrementos regulares, añadiendo nuevas características según el beneficio que aporten. Por esto es muy útil en proyectos complejos, con requisitos que cambian continuamente y en los que debemos ser flexibles a cambios repentinos.

Fases

1. Planificación (*Product Backlog*): En esta fase establecemos las tareas que se deben cumplir, aportando la información necesaria. Esta lista de tareas se formula con el equipo de trabajo y el *Product Owner*. Cada tarea recibe unos «puntos de historia» que priorizan unas u otras. Una vez tenemos el *product backlog* listo, comenzamos el primer *sprint*.
2. Ejecución: Un *sprint* es un intervalo del tiempo en el cual intentamos realizar todas las tareas marcadas en el *product backlog* con la intención de tener un «entregable» (o «producto mínimo viable» en el caso del primer *sprint*). Una vez hemos iterado sobre nuestro producto, debemos medir el progreso hecho.
3. Control: en la fase de control (o *burn down*) medimos el progreso realizado en el anterior *sprint*. El *scrum master* será el encargado de actualizar los gráficos con los *story points* de las tareas realizadas.

Continuous Integration / Continuous Development (CI/CD)

El desarrollo continuo es un término que engloba a muchos otros términos. Todos ellos teniendo en común la automatización de procesos, que se lanzan cuando cambiamos algo en nuestro código. Los términos que componen el desarrollo continuo son: la integración continua, el *continuous testing*, *continuous delivery* y el *continuous deployment*.

La integración continua es la práctica de automatizar la integración de cambio en el código con otro software «externo» o de terceros. Un ejemplo sería la comprobación de tipos de en nuestro código justo después de hacer un *commit* con *Git*. En nuestro proyecto hemos integrado tres herramientas que se ejecutaban justo después de realizar un *commit* de los cambios.

PyLint

PyLint es una herramienta de *linting* para **Python**, así como **ESLint** lo es para **JavaScript**. El *linting* es la práctica de revisar el código en busca de errores de sintaxis, código poco intuitivo, detección de «malas prácticas» o uso de estilos inconsistentes (i.e.: uso de variables estilo *snake case* con *camel case*). Las herramientas de *linting* pertenecen al grupo de herramientas de análisis estático.

Black

Black es un formateador de código opinado para **Python**. Nos permite mantener un estilo constante en toda nuestra *codebase*, entre proyectos, y entre distintos desarrolladores. Es muy útil en proyectos compartidos y permite mejorar la legibilidad del código. Podemos personalizar reglas como el tamaño máximo de una línea de código o el uso de comillas simples (') o compuestas (").

MyPy

MyPy es un comprobador de tipos estático para, lo has adivinado, **Python**. Esta herramienta nos ayuda a comprobar que todos los tipos de nuestra funciones y variables son correctos, o que en su defecto, no nos los hemos dejado vacíos. Es una herramienta opcional; pues al igual que **JavaScript**, **Python** es un lenguaje dinámicamente tipado y no necesita tipos para funcionar correctamente.

4.3. Herramientas

Git

Git es un controlador de versiones y manejo de código basado en la velocidad. Con **Git** podemos realizar *commits* con los cambios incrementales que hacemos en nuestro código y así mantener un historial. De esta forma podemos desarrollar código sabiendo que todas las iteraciones sobre el código están guardadas y si fuera necesario, podemos volver a ellas.

Git se usa como un CLI (*Command-Line Interface*), aunque hay muchas herramientas que lo integran con una UI simple e intuitiva (i.e.: **GitKraken**, **Github Desktop** o **Turtoise**)

GitHub

GitHub es una herramienta de *hosting* de repositorios de `Git` controlada por Microsoft. Nos proporciona una interfaz web en la que podemos ver nuestros repositorios o los de los demás (siempre que sean públicos).

Por otro lado nos ofrece herramientas para implementar CI/CD, así como una ventana en la que mostrar nuestras contribuciones en *Open Source*.

GitLab

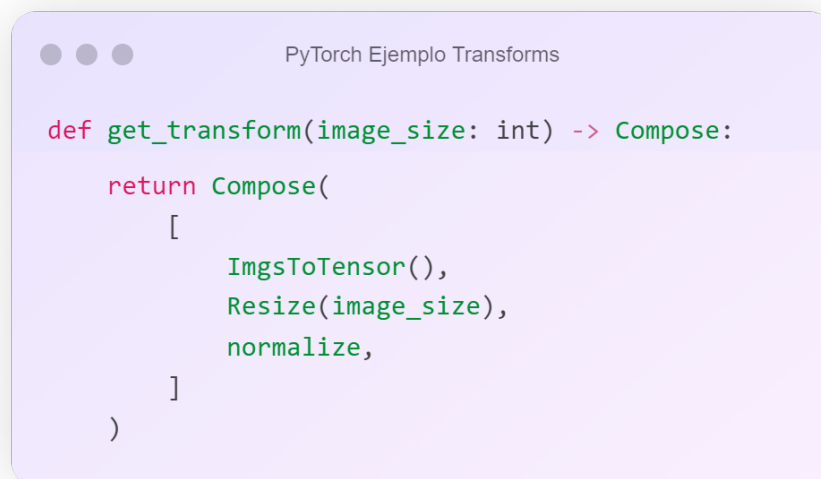
Gitlab es una alternativa a Github. Las características ofrecidas son prácticamente las mismas. Una de las diferencias, aunque es mínima, es que Gitlab ofrece mayor compatibilidad a la hora de hacer *continuous deployment* con los servicios de Google.

PyTorch

Pytorch es un framework de *machine learning* para `Python` que acelera la creación de prototipos y salida de modelos a producción. Fue creado por Facebook para rivalizar con `Tensorboard`. Pytorch nos ofrece un ecosistema con numerosas librerías y herramientas para acelerar el desarrollo. Algunas son PyTorch3D (computación 3D), ONNX Runtime (para ejecutar modelos en el formato estándar ONNX), Transformers (herramientas para procesamiento de lenguaje natural), PyTorch Lightning (para crear modelos de forma rápida, al estilo Keras) y Ensemble-PyTorch (para generar *ensembles* de modelos).

Por otro lado nos ofrece librerías para tratar con distintos formatos de datos: torchaudio (audio), torchtext (texto) y torchvision (imágenes y videos). También nos ofrece distintas clases para cargar datasets y transformar los datos que hemos cargado.

En el *snippet* de código 4.1 podemos ver un ejemplo de una transformación de datos usando el módulo «Compose». Vemos que dado un tamaño de imagen, vamos a convertir una lista de imágenes a tensor, tras esto vamos a ajustar el tamaño de cada imagen y normalizar los valores.



```
def get_transform(image_size: int) -> Compose:
    return Compose(
        [
            ImgsToTensor(),
            Resize(image_size),
            normalize,
        ]
    )
```

Figura 4.1: Ejemplo de *transform* en PyTorch

Pytorch nos otorga cientos de herramientas que hacen fácil cargar y transformar los datos, trabajar con tensores, crear y entrenar redes neuronales, exportarlas y cargarlas de nuevo; todo el proceso.

Tensorflow

Tensorflow es también un *framework* de **Python** de *machine learning*. Tiene la mayoría de herramientas que nos ofrece **Pytorch**, aunque existen varias diferencias entre ambos (ver tabla ?? para una lista más completa).

ONNX

ONNX (*Open Neural Network Exchange*) es un formato abierto contruido para representar modelos de *machine learning*. Busca ofrecer un formato estándar para poder compartir modelos entre distintos lenguajes, herramientas, *frameworks*, *runtimes* y compiladores. El proyecto es de código abierto con más de 200 *contributors* y está escrito en **C++**.

PyTorch tiene un módulo para exportar y cargar modelos en **ONNX**. Por otro lado **ONNX** nos ofrece una librería, **ONNXRuntime** para inferencia y serialización de los modelos, así como *quantization*.

Y por mantener el espíritu de la estandarización y reutilización entre lenguajes y ecosistemas, tenemos también una librería para usar los modelos en la web con NodeJS: `ONNX.js`.

Netron

Netron es una aplicación web en la que podemos subir neustros modelos con formato `ONNX`, `PTH` o similar para visualizar su estructura. En la figura 4.2 podemos ver la estructura de uno de los modelos entrenados en este proyecto.

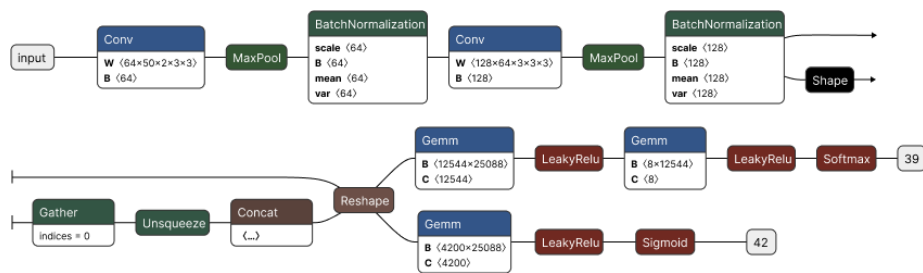


Figura 4.2: Estructura de un modelo con CNNs en formato `ONNX`. Visualización con Netron

	Tensorflow	PyTorch
Desarrollador	Google	Facebook
Adopción	Más usado	Relativamente nuevo
Definición de grafos	Estático. Definimos los grafos antes de ejecutar el modelo. Todas las comunicaciones se hacen usando el módulo <code>tf.Session</code>	Más imperativo y dinámico. Podemos definir y cambiar los nodos en plena ejecución
<i>Debugging</i>	Debemos usar una herramienta especializada: <code>tfdg</code> . Nos permite hacer log de tensores en la <i>session</i> en la que estamos	Podemos usar nuestras herramientas favoritas de <i>debugging</i> ya que se define en <i>runtime</i> . También podemos usar <code>print</code>
Visualización	Tenemos Tensorboard . Una herramienta de <i>log</i> y <i>debugging</i> que funciona a tiempo real ofreciendo una interfaz web. Ofrece tanto que hemos dedicado un pequeño apartado justo abajo	También es compatible con Tensorboard , lo que lo hace tan visual como Tensorflow
<i>Deployment</i>	Tensorflow nos ofrece Tensorflow Serving , un <i>framework</i> para hacer <i>deploy</i> de nuestros modelos en servidores especiales	No tenemos una herramienta especializada para hacer <i>deploy</i> . Debemos usar otros <i>frameworks</i> como Flask o Django (en nuestro caso hemos usado FastAPI)
Paralelismo	Es un proceso manual y se debe hacer de forma individual por modelo, aunque otorga un control más granular	El paralelismo es muy fácil de implementar. Simplemente llamamos a <code>torch.nn.DataParallel</code> y «mágicamente» funcionará sin ningún esfuerzo
<i>Framework</i> vs Librería	Tensorflow se parece más a una librería, ya que la mayoría de opciones son de «bajo nivel». Debes configurar muchas operaciones simples y produce algo de <i>boilerplate</i> . Es por esto que Tensorflow se suele usar junto a Keras , un <i>framework</i> de <i>deep learning</i> «diseñado para humanos»	PyTorch es un <i>framework</i> . Tiene muchas abstracciones con estructuras comunes disponibles de forma directa para un rápido prototipado. Además de herramientas de transformación, carga de distintos formatos y múltiples <i>datasets</i> famosos listos para ser descargados y usados

Tabla 4.1: Diferencias entre **Tensorflow** y **PyTorch**

Nos muestra de que «tipo» es cada capa de la red, el tamaño de entrada y salida, las funciones de activación usadas, la capa *flatten* y muchos datos más.

Docker

Docker es un PaaS que usa la virtualización a nivel de OS (sistema operativo) para distribuir software en «contenedores». Un contenedor está aislado del resto del sistema y consume muchos menos recursos que una máquina virtual convencional.

Los contenedores se construyen a través de imágenes. Las imágenes se crean normalmente usando un archivo de configuración **Dockerfile** (aunque hay más opciones). En este archivo añadimos las «instrucciones» para crear nuestra imagen, aunque no es hasta que instanciamos un contenedor, que sabemos si la imagen está correctamente construida.

Hemos usado **docker** a la hora de crear una imagen para nuestro *back-end* y modelo. Esto nos ha permitido asegurarnos que si la imagen funcionaba en local, iba a funcionar también en producción.

Tras tener la imagen, ya podemos subirla a distintos sitios y así poder generar instancias de contenedores y poner nuestro servicio en funcionamiento. Como veremos en el siguiente apartado, nosotros tenemos la imagen alojada para producción en *Google Cloud Registry*, un servicio de almacenamiento y registro de imágenes **Docker**.

Por otro lado y para poder compartir la imagen con la comunidad, también la hemos alojado en *DockerHub*, la plataforma de *hosting* propio de **Docker**.

- Accede a la imagen de Sign2Text-API en <https://hub.docker.com/repository/docker/gazquez/sign2text>

Deployment

En el proyecto se han considerado y probado numerosas opciones de *deploy* tanto para el modelo, el *back-end* y el *front-end*.

Google Drive

Comencemos con el modelo. El modelo está alojado en *Google Drive*. *Google Drive* no es el mejor CDN en el que alojar nuestros modelos, pero nos ha servido en nuestro caso. El tenerlo en *Google Drive* nos permite hacer iteraciones sobre el modelo sin tener que volver a construir el contenedor de **Docker** en local. Esto es útil para separar el desarrollo de la red neuronal de otros desarrollos como es el *front-end* y, sobre todo, el *back-end*.

De esta forma podemos descargar el modelo programáticamente cuando nos movemos al entorno de producción. Para esto usamos una pequeña librería de **Python**: **gdown** con la que ejecutamos la descarga del modelo en el **Dockerfile** en la fase de construcción de la imagen.

Esto nos ha ayudado mucho a reducir el tamaño del repositorio y la dependencia entre proyectos (**Sign2Text** y **Sign2Text-API**).

- Librería gdown: <https://github.com/wkentaro/gdown.git>
- Repositorio Sign2Text-API: <https://github.com/irg1008/Sign2Text-API.git>

Google Cloud

Moviéndonos a la API. La API está alojada en Google Cloud, más concretamente en **Cloud Run**.

Google Cloud es un **IAAS** (infrastructure as a service) y **PAAS** (Platform as a Service) que ofrece servicios que se ejecutan en la nube y que tratan *hosting*, computación, *big data*, **IoT** (*Internet Of Things*), *machine learning*, almacenamiento y bases de datos, entre otros.

En nuestro caso, necesitábamos alojar una API que consumiera pocos recursos. Hemos conseguido esto usando la tecnología *serverless* y **Cloud Run**.

En la tecnología *serverless* el servidor aloja solo los recursos necesarios para realizar una tarea, y lo hace a demanda. Es decir, que solo se ejecuta mientras un cliente necesite acceso al servicio. Esto hace que el tiempo que está ejecutándose se minimice, disminuyendo a su vez los costes. La pega con esto, es que si el servicio no se está utilizando, la instancia se apaga. Al apagarse, la próxima vez que un cliente quiera acceder al recurso,

deberá inicializarse de nuevo y asignar los recursos, y dependiendo de las necesidades de nuestro servicio; este tiempo puede ser mayor o menor.

En nuestro caso, y con el objetivo de probar estos tiempos, hemos asignado cero instancias mínimas al servicio (esto indica que puede apagarse en caso de que no reciba peticiones). Al asignar cero instancias y no recibir peticiones en un tiempo, el servicio se apaga. El tiempo medio de recuperación (o *up time*) es de 15s. Este tiempo es inutilizable en entornos de producción reales (ninguna persona va a esperar 15 segundos a que comience un servicio). Por esta razón, el número de servicios activos que hemos usado es «uno». Con un servicio activo, la instancia no se apaga y podemos responder a los clientes de forma rápida. El *drawback* de esto es que el precio de uso aumenta muchísimo, ya que la clave del *serverless* es consumir solo los recursos necesarios.

Google Cloud Run es la herramienta de Google para instancias *serverless*. Amazon y Azure tienen sus propias implementaciones de estas tecnologías.

- Se puede acceder a Cloud Run en <https://cloud.google.com/run>
- Se puede acceder a la API de Sign2Text (es posible que el tiempo de respuesta sea algo lento) en <https://api.sign2text.com/docs>

Azure

Azure es la alternativa de Microsoft a Google Cloud con herramientas muy similares. Se incluye en la lista de herramientas ya que se usó durante un tiempo para alojar el contenedor con el *back-end* y el modelo. Como nota personal, la experiencia y la DX (*developer experience*) con Azure es muy mala. La UI es muy desorganizada y no ofrece *logs* a tiempo real de la construcción de las imágenes o los contenedores.

AWS

Igual que Azure, AWS es la alternativa a Google Cloud. También tiene herramientas muy similares [1]. AWS es la plataforma de cloud más usada, seguido de Azure y Google [22].

En la realización de este proyecto, se han usado las tres herramientas para alojar el *back-end*. Si se desea una guía «paso a paso» de como subir el

contenedor y hacer *deploy* con el modelo descrito en el apartado anterior en cada una de ellas accédase en <https://github.com/irg1008/Sign2Text-API#readme>

Vercel

Hemos hablado de donde tenemos alojado el *back-end* y el modelo, pero, no menos importante, es el *front-end*, que permite a los usuarios una plataforma para usar y experimentar con nuestro modelo.

Vercel es un PAAS que busca la agilidad a la hora de desarrollar, previsualizar y expedir nuestro software. Es una plataforma que nos permite integrar fácilmente CI/CD, obteniendo el código necesario para producción de nuestros repositorios.

La DX es espectacular, ya que nos ofrece todas las herramientas que necesitamos (*serverless*, *edge functions*, dominios, etc) para desarrollar y llevar nuestro proyecto a producción de forma rápida. Además, el plan gratuito es suficiente para empezar con un proyecto medianamente grande.

De forma transparente, Vercel usa AWS y Cloudflare (Cloudflare Workers) para otorgar los servicios. Esto hace que el servicio ofrecido sea rápido y el *uptime* muy bajo. Por otro lado, Vercel es dueña también de un *framework* (Next.JS) escrito sobre la librería más usada de JavaScript: React. En este proyecto no se ha usado Next.JS, ya que se ha optado por otro *framework* (Astro) de prototipado rápido.

- Se puede acceder a Vercel en <https://vercel.com>
- Next.JS está disponible en <https://nextjs.org>

Front-end

El *front-end* es la interfaz que se ofrece a un «cliente» para acceder de forma sencilla a la plataforma y las características ofrecidas por una aplicación en el *back-end* (lado del servidor).

En nuestro caso, hemos realizado una web sencilla en la que los usuarios pueden arrastrar un video a la pantalla. Tras arrastrar el video, se muestra una *preview* con un botón. Al pulsar el botón se envía el video a nuestro *back-end* para ser procesado. El usuario recibe entonces el signo

identificado para el video. Se puede ver un análisis más en profundidad de este proceso en el anexo de «Diseño» y el «Manual del Usuario».

Veamos que herramientas se han usado para crear el *front-end*:

Tailwind

Tailwind es un *framework* de CSS para construir y estilar páginas de forma rápida y sin salirnos del HTML. Tailwind nos ofrece todas las herramientas disponibles en CSS:

- *Element states* como «:hover» o «:active».
- Media queries para construir sitios *responsive*
- Accesibilidad
 - Modo oscuro
 - *Reduced motion*
- Animaciones y transiciones
- Tipografía
- Sombras
- Y mucho más

Svelte

Svelte es un *framework* de *JavaScript* como React, Angular o Vue, solo que algo distinto.

En vez de usar el navegador y el denominado *virtual* DOM, Svelte solo cambia los elementos necesarios del DOM según cambien los estados internos de la aplicación.

Por otro lado, este *framework* nos permite escribir JS (**JavaScript**), CSS y HTML en el mismo archivo. Esto no es algo nuevo, pues podemos hacerlo en un archivo html normal. La diferencia con Svelte es que podemos acceder de forma programática sin tener que usar el DOM API, reduciendo muchísimo *boilerplate*, como podemos apreciar en la figura 4.3



Figura 4.3: Ejemplo de código de detección de la anchura de un botón con Svelte (izquierda) y HTML (derecha)

Astro

Astro es un constructor de sitios estáticos centrado en la DX y en la optimización del tamaño de la salida tras la compilación. **Astro** se basa en el concepto de «microislas».

Una isla o «microisla» se entiende como un fragmento de código que funciona por si mismo y no tiene dependencias de otros fragmentos o islas e nuestro código. También se conoce como *Partial Hydration* y permite rehidratar solo las partes necesarias de nuestra UI. De esta forma reducimos la carga en el sistema y los tiempos de actualización con el servidor, ya que también es SSR.

El SSR o *server side rendering* permite ejecutar todos los cambios sobre la UI (con JavaScript o CSS) en el servidor. De este modo el cliente solo recibe el código HTML a mostrar.

El sistema de islas es muy útil, porque al «aislar» todos los componentes por separado, podemos mezclar distintas tecnologías y *frameworks*, sin miedo a que interfieran entre sí. Es por esto por lo que hemos podido usar **Svelte** para hacer toda la lógica, y hemos mantenido la parte estática en HTML puro.

- Puedes acceder a la web de astro en: <https://astro.build>
- Svelte se puede visitar en <https://svelte.dev>
- También está disponible el *front* del proyecto proyecto en <https://sign2text.com>

Figma

Figma es una herramienta de diseño, del estilo de Adobe Illustrator, pero orientada a desarrollo web. En ella podemos manejar archivos vectoriales, crear animaciones, maquetar páginas webs, entre otras cosas. La comunidad es muy activa dentro del ecosistema de esta herramienta, por lo que se nos ofrecen cientos de *plugins* y *templates* para cubrir todas nuestras necesidades.

Las figuras y dibujos que se ven en esta memoria están hechos con esta herramienta.

Puedes acceder a figma en <https://figma.com>.

Back-end

El back-end es la parte más importante de una aplicación web, ya que es donde tenemos toda la lógica de negocio, los accesos a las bases de datos, llamadas a APIs externas, entre otras cosas. El código del back-end se ejecuta en servidores (o en nodos de borde) y el cliente no modifica ni accede a este código (como si pasa en el *front-end*).

En nuestro caso hemos usado un *framework* sobre Python para crear una API optimizada y documentada.

FastAPI

FastAPI es un *framework* sobre Python moderno y orientado a la velocidad. Gracias a sus amplias herramientas, nos permite crear *endpoints* documentados en base al estándar OpenAPI de forma rápida. Todo esto sumado a la cantidad de librerías que nos ofrece el ecosistema de Python, FastAPI es una opción perfecta para crear *endpoints* en nuestro *back-end* (Netflix usa FastAPI en producción).

Por otro lado, junto a FastAPI, usamos Uvicorn. Uvicorn es un ASGI (*Asynchronous Server Gateway Interface*) *server* usado para inicializar el servidor en desarrollo y en producción. Permite *hot reload* para un desarrollo más rápido.

- FastAPI: <https://fastapi.tiangolo.com/>
- Uvicorn: <https://www.uvicorn.org/>

4.4. Miscelania

Shields.io

Shields.io es una herramienta que permite la creación de *badges* personalizadas para nuestros archivos **markdown**. Podemos elegir colores y añadir etiquetas e iconos personalizados.

Github Copilot

Github copilot es una herramienta de IA que funciona como un asistente de código. Su función es sugerir *autocompletes* para el código que vamos escribiendo a tiempo real, facilitando mucho la labor de tareas sencillas.

Insomnia

Insomnia es un cliente de APIs. Nos permite hacer llamadas y tests sobre nuestros *endpoints*. Nos ayuda mucho a la hora de añadir *headers* personalizados y mantener una colección de peticiones ordenada.

IDE

VSCode

Se ha usado VSCode como IDE para el desarrollo tanto del modelo, como el *front-end*, el *back-end* y esta memoria. VSCode es un IDE multifacético de Microsoft y de código abierto. El ecosistema de *plugins* y la API interna del programa permite a la comunidad adaptar y soportar cientos de lenguajes y herramientas.

Este IDE está construido con **JavaScript** y portado a escritorio usando **Electron**, un *framework* para crear aplicaciones de escritorio.

T_EXMaker

Texmaker es un editor de L^AT_EX con un visor de PDF personalizado. Soporta Linux, macOS y Windows e integra todas las herramientas que necesitas para desarrollar documentos complejos y profesionales.

Podemos acceder en <https://www.xmlmath.net/texmaker/>

Aspectos relevantes del desarrollo del proyecto

En esta sección vamos a hablar de algunos de los problemas y retos con los que nos hemos enfrentado en el proyecto. Vamos a hablar de la carga de los datos y de como pasamos de clasificación de imágenes a clasificación de video. Seguiremos hablando un poco del tratamiento de los datos. Tras esto mencionaremos una de las estructuras de CNN usada, ResNet. Continuamos hablando de la estructura de nuestra red y problemas que hemos tenido, junto con la hiperparametrización y el entrenamiento. Para terminar hablaremos de como hemos usado una red con dos salidas para intentar mejorar el *accuracy* y de como se ha explorado la cuantización para comprimir el modelo y hacerlo «*deployment-ready*».

5.1. Carga de datos

El objetivo de este proyecto es la clasificación de lenguajes de signo a nivel palabra. El *dataset* usado [31] es WLASL (Word-Level American Sign Language) y está extraído de un paper que ganó mención honorable en el WACV (2020). Este *dataset* se compone de 21083 videos, que se clasifican en 2000 palabras distintas.

Al comienzo del proyecto, pensábamos que usando **PyTorch** íbamos a tener las herramientas necesarias para cargar los datos en formato video y procesar las transformaciones directamente. Esto resultó no ser así.

Nos dimos cuenta que no teníamos una forma directa de cargar un video con `Pytorch` y transformarlo a un tensor con el en el cual podíamos ejecutar transformaciones o usar como entrada.

Por eso, lo primero que hicimos, fue crear unos *scripts* con los que transformar los videos en *frames*. Con la idea de poder ser reutilizado lo máximo posible, nos pusimos a diseñar los *scripts* de forma dinámica, es decir; los *scripts* debían aceptar distintos argumentos para procesar distintos *datasets*, independientemente de que el más importante fuera el citado arriba.

Puedes ver más con detalle la estructura del script en el anexo «Manual del programador» pero básicamente el *script* se puede ejecutar con las siguiente opciones:

GLOBALES

- -i, -input: Carpeta con los videos a procesar (Opcional).
- -o, -output: Carpeta donde se guardaran los frames (Opcional).
- -l, -labels: Número de labels a procesar. Puede ser un número o “all” para procesar todos (Opcional).
- -c, -config: Archivo de configuración con las etiquetas (Opcional).
- -h, -help: Muestra esta ayuda.

EXTRACCIÓN DE FRAMES

- -m, -merge: Indica si se debe unir los frames en una imagen (Opcional).
- -f, -frames: Número de frames aleatorios del video (Opcional).

EXTRACCIÓN DE VIDEOS

- -v, -video: Extrae todos los frames del video en una carpeta única para ese video. Deshabilita las opciones -f y -m (Opcional).

Con estas opciones nos ahorramos tener que estar transformando los datos de forma «manual» o haciendo cambios a los *scripts* cada vez que queríamos usar unas u otras etiquetas.

Bien, una vez nos creamos estos *scripts* para transformar de video a *frames*, estabamos listos para cargarlos con **PyTorch** en un **DataLoader** (clase de **PyTorch** usada para cargar datos desde la estructura de archivos).

Como funciona el **DataLoader**, usa la estructura de archivos para cargar los datos, es decir, que carga los datos y los relaciona con la etiqueta correspondiente según el nombre de la carpeta en la que están.

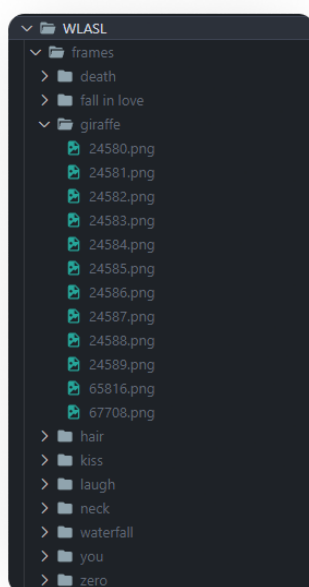


Figura 5.1: Estructura de carpetas necesaria para la carga de datos con **PyTorch**

En la figura 5.1 podemos observar que los *frames* exportados se sitúan en carpetas según la etiqueta que los clasifica. Al tenerlo de este modo, podemos cargar todo el dataset con **PyTorch** y hacer dentro la división en *set* de entrenamiento, test y validación.

Nota: Esta estructura de clases se mantiene para todos los dataset. La figura que acabamos de ver se corresponde con la red que clasifica únicamente *frames* y no videos (la primera red construida). La estructura de carpetas para el dataset con videos es muy similar, solo que en vez de haber un frame por video en cada clase, tenemos una carpeta por video con los frames de ese video, para cada clase (un nivel más).

Volvemos a unir los frames

Una vez tenemos la estructura lista para cargar los datos con PyTorch, debemos unir los frames en un tensor «video» (en el caso de que tengamos más de un frame por video). Para esto usamos un *DataLoader* personalizado, que va a concatenar las imágenes en un tensor único. De este modo pasamos de tener un tensor de tamaño $(3xHxW)$ a uno de tamaño $(Fx3xHxW)$ donde la H y la W representan la altura y la anchura de un *frame* respectivamente, y la F el número de frames cargados para un video en particular ²⁵.

Una vez hemos hecho la concatenación, ya tenemos nuestro dataset transformado en tensores de 4 dimensiones. Ahora debemos dividir los datos en *sets* de entrenamiento, test y validación, no sin antes, hacer las transformaciones necesarias en los datos y agrupar los videos en *batches*.

5.2. Tratamiento de los datos

Una vez hemos cargado los datos, comienza una de las partes más importantes en el *machine learning* y la minería de datos: la transformación de los datos y adaptación para el entrenamiento.

En nuestro caso, al usar CCNs, y como hemos visto en el apartado teórico, el tratamiento de datos no es tan importante. Esto se debe a que en la fase de convolución, el *kernel* tendrá en cuenta todas las dimensiones presentes en nuestro tensor de entrada, y se entrenará para sacar las características que más importen.

Aún así, hay una transformación en los datos que es muy importante, ya que debido al alto consumo de las CNNs, necesitamos un ajuste sobre el tamaño de las imágenes, pasando del tamaño original a $(224x224)$ (píxeles).

Una vez hecha esta transformación sobre las imágenes, normalizamos los datos.

Normalización

La normalización, que consiste en trasladar valores de su rango de valores original a otro controlado (normalmente entre 0 y 1), es muy útil

²⁵El 3 identifica los tres canales RGB de una imagen en color

cuando estamos tratando entardas con datos que se sitúan en rangos muy distintos ²⁶.

En nuestro caso la entrada de los datos está compuesta únicamente por imágenes de color (esto son 3 canales con píxeles que varían entre 0 y 255), por lo que al no mezclar con otros tipos de datos y rangos, podríamos proceder sin normalizar.

Pero no es la mejor opción. Normalizar los datos es una práctica común para conseguir acercar la media de los datos a 0 y acelera el aprendizaje de la red proporcionando una convergencia más rápida.

Data Augmentation

El *data augmentation* consiste en aumentar el tamaño del *dataset* duplicando datos del mismo y alterándolos para que parezcan distintos.

En nuestro caso no hemos incluido esta técnica debido a tratar con videos y no con imágenes. Al tener la transformación de los datos a nivel imagen (*resize* y normalización), se nos complicaba la aumentación de los datos ya que debíamos «alterar» los datos para todos los *frames* del mismo video, con tal de no alterar la información espacial de dicho video.

Esto es algo que nos gustaría marcar para un plan futuro en el proyecto, ya que el *dataset*, aunque muy extenso, tiene muy pocos videos por etiqueta (aproximadamente 40), y aumentar este número aunque sea en 10, sería una extensión notable del dataset.

Batching

A la hora de entrenar una red, no podemos usar el *dataset* completo como entrada, ya que nos quedaríamos sin memoria enseguida. Por esto agrupamos las entradas en *batches* o partes. Al hacer esto, tenemos que modificar nuestra red para aceptar los datos en grupos.

El *batching* realmente lo que está haciendo es aumentar la dimensión de nuestro tensor de entrada. En los puntos anteriores hemos visto que nuestro tensor de entrada era del tipo $(Fx3xHxW)$ y que representaba un video de nuestro dataset. Ahora añadimos una dimensión más B represen-

²⁶i.e.: un valor enumerado entre 0 y 5 y un valor continuo entre 100 y 200

tando el tamaño de cada *batch*: $(B \times F \times 3 \times H \times W)$, resultando en un tensor de entrada de 5 dimensiones.

Cuando entrenamos una CNN con un tensor de entrada de 5 dimensiones decimos que tenemos una red convolución 3D ²⁷.

Subsampling

En el subsampling divisimos los *batches* creados en *sets* de entrenamiento, test y validación.

En nuestro caso hemos usado un *subsampler* aleatorio proporcionado por PyTorch: `SubsetRandomSampler` con una división de 70 %, 20 % y 10 % para entrenamiento, test y validación.

5.3. Red Neuronal ResNet

Antes de comenzar a construir nuestra CNN, probamos el *dataset* con una *ResNet* ya construida. Una *ResNet* [23] es una estructura que usa numerosas capas de CNNs una detrás de la otra y conectadas entre sí cada dos o tres capas (conexión residual). Tenemos varios tipos identificadas por el número de capas que tienen: *ResNet18*, *ResNet34*, *ResNet50*, *ResNet101*, etc.

En nuestro caso, la prueba que hicimos con una *ResNet* fue usando un dataset con fotos de animales [4] con 10 etiquetas y con aproximadamente 50 ejemplos por etiqueta. El accuracy conseguido en entrenamiento y test (ya que en este punto inicial no teníamos validación) es:

Tipo de ResNet	Entrenamiento	Test
ResNet18	97,59 %	95,51 %

Tabla 5.1: Accuracy de una red ResNet con un dataset pequeño de imágenes de animales [4]

Por otro lado, la misma red con nuestro dataset compuesto por un único frame por video consiguió el siguiente *accuracy*:

²⁷Una CNN 1D se usa para procesar datos lineales, como la altura según el tiempo. Una CNN 2D se usa para imágenes y una 3D (que significa de 3 o más dimensiones) para videos u otros datos que necesitan imágenes en *stack*

Tipo de ResNet	Entrenamiento	Test
ResNet18	92,4 %	64,6 %

Tabla 5.2: Accuracy de una red ResNet con nuestro dataset original con 10 etiquetas, 30 videos por etiqueta, 1 solo frame por video

Como vemos, mucho menor. Pero esto es normal, ya que clasificar un signo completo de un video con un solo frame es imposible. A la hora de probar el modelo con una inferencia por webcam, los resultados eran prácticamente aleatorios.

Al ver esto, decidimos entonces pasar ya a crear nuestra propia red convolucional.

5.4. Estructuras de la redes convolucionales

La estructura de nuestra primera red convolucional (denominada «SimpleNet»), usada para clasificación de video, es mucho más simple que la que vemos en una red *ResNet*. Se compone de tres capas de convolución con *pooling* y una capa densa (o *fully connected*) con activación *softmax* al final. Podemos ver la estructura en la figura 5.2.

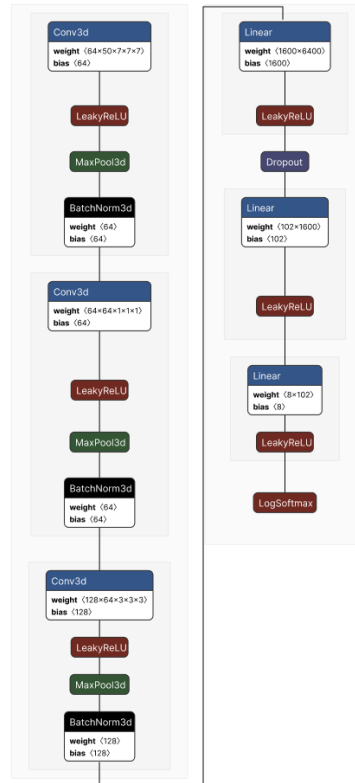


Figura 5.2: Estructura de una CNN exportada con *Netron*

La segunda red convolucional que tenemos, que hemos nombrado «TwoOutNet», tiene una estructura que comienza con dos capas convolucionales. Tras esto, tenemos una gran diferencia con la primera, y es que en este caso, generamos dos salidas (la clase y la pose detectada). Por eso tenemos dos capas densas, una por salida, con una función de activación de la capa oculta *softmax* y *sigmoid* respectivamente.

Podemos ver en la figura 5.3 la estructura mencionada con la división en dos tras la segunda convolucional.

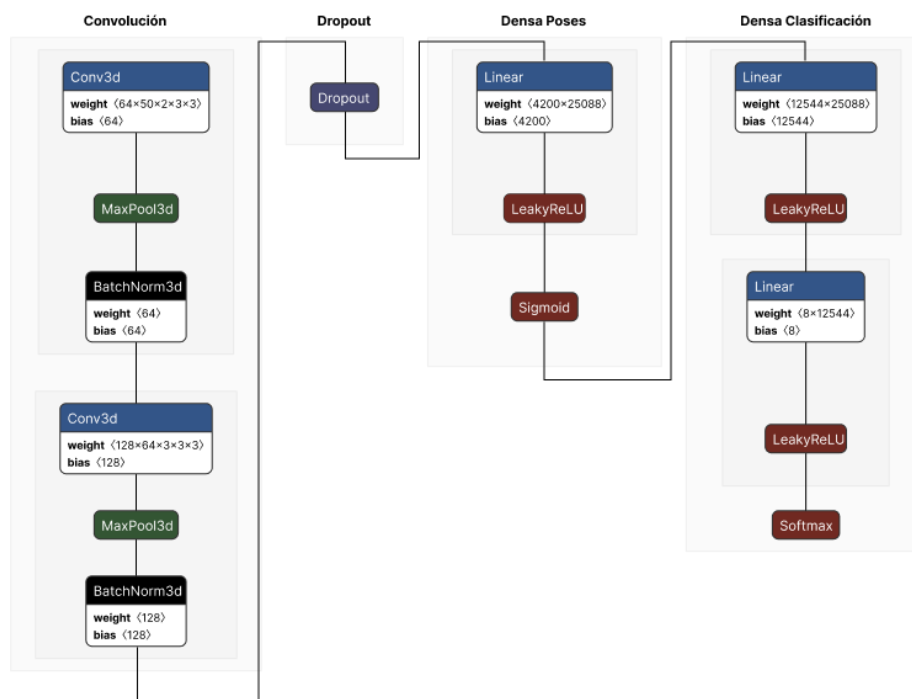


Figura 5.3: Estructura de una CNN con doble salida exportada con *Netron*

Para saber más sobre el porque de la doble capa de salida, adelanta al apartado de «Dos salidas en la misma red»

5.5. Entrenamiento e hiperparametrización

Los hiperparámetros que hemos cambiado en las pruebas son:

- Número de segmentos: Esto indica el número de partes en los que dividimos un video.
- *Frames* por segmento: Con este valor cambiamos el número de frames que seleccionamos en cada segmento.
- *Batch size*: El tamaño del *batch* de entrada de la red. Un número muy alto necesita mucha memoria de GPU.
- Tamaño de la imagen de entrada: En píxeles, aquí indicamos el tamaño que se debe usar en la fase de transformación de los datos antes de entrar en la red.

- Número de *epochs*: El número de iteraciones de entrenamiento de la red. Como vimos en la teoría, hay un punto en que el *accuracy* de validación de la red disminuye y comienza a sobreajustarse.
- *Learning rate*: El learning rate nos indica lo rápido que aprende la red.

Optimizer

Un optimizador es un algoritmo usado en *deep learning* para disminuir las funciones de pérdida. Son funciones que dependen de los parámetros de pesos, bias y en algunos casos, del *learning rate*. Hay muchos optimizadores como el descenso de gradiente, SGD (*Stochastic Gradient Descent*), ADAGRAD (*Adaptive Gradient Descent*), RMS-PROP (*Root Mean Square Propagation*) o ADAM (*Adaptive Moment Estimation*).

En nuestro caso hemos usado ADAM y SGD, siendo este último el que mejores resultados nos ha dado.

Scheduler

Criterion

O funciones de pérdida, nos ayudan a calcular la calidad del entrenamiento de la red. Esta medida es mucho más fiable que el *accuracy* global, ya que este puede estar contaminado por un valor que se está clasificando especialmente bien. A la hora de calcular la pérdida entre la salida de la red y el *ground truth*, buscamos que el valor sea el mínimo posible.

Las funciones de pérdida usadas son:

- *Cross Entropy Loss* [2]: Esta función forma parte de las llamadas funciones de pérdida NLL (*Negative Log-Likelihood*). Esta familia de funciones se usa únicamente en clasificación.
- MSLOSS (*Median Square Error Loss*) [3]: Esta función de pérdida se usa en la regresión de las poses. Compara el vector salido de la red con las poses correctas y calcula el error cuadrático medio.

5.6. Dos salidas en la misma red

En los apartados anteriores hemos hablado de clases y poses cuando nos hemos referido a la salida de la red. Esto es porque en nuestra última CNN creada, tenemos dos salidas, una con las probabilidades de cada clase y otra con un vector de poses sobre el gesto de entrada.

Esto significa que nos enfrentamos a dos problemas, uno de clasificación y uno de regresión. Es por esto que cada salida tiene una función de pérdida distinta.

La hipótesis que fundamenta la creación de esta estructura fue:

“A la hora de clasificar gestos, la red *SimpleNet* se centra más en los sujetos que hacen los gestos que en los propios gestos. Se comprueba que cuando el sujeto es completamente distinto, la salida de la red identifica el gesto por la persona más que por los movimientos de las manos. Por esto se piensa que añadiendo el factor movimiento con las poses, podemos «hacer olvidar» a la red del sujeto y conseguir centrarla en la detección de los gestos.”

Tras pensar esto, añadimos la segunda salida de la red, creamos la función de pérdida, y sumamos el valor del coste para dicha salida al ya existente de la clasificación. Tras esto, hacemos el *backpropagation* con la suma de las pérdidas. La mejora en *accuracy* es mínima pero vemos que es capaz de centrarse mucho mejor en los gestos que en la persona en concreto. Incluso es capaz de adivinar un gesto dada una entrada de un dibujo animado, como vemos en la figura 5.4.

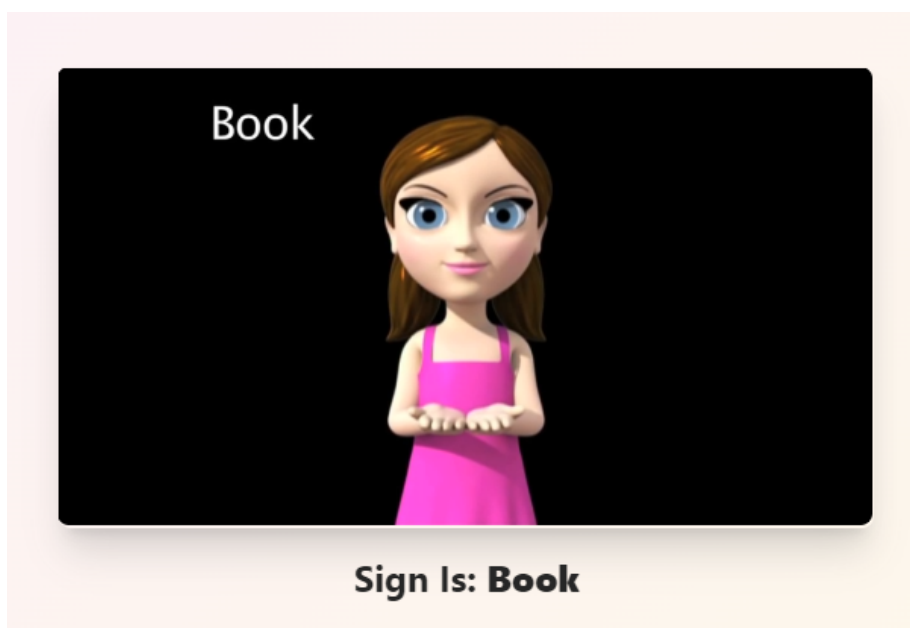


Figura 5.4: La red es capaz de adivinar un signo hecho por un dibujo animado

5.7. Exportación del modelo

A la hora de exportar el modelo, hemos usado la herramienta interna de PyTorch, que nos exporta modelos con formato «.pth». Este formato es perfecto si nos mantenemos en el ecosistema de PyTorch, pero si queremos usar el modelo con otros *frameworks* o librerías, debemos exportarlo a un formato estandar.

ONNX

El formato estandar más utilizado es ONNX. Puedes ir a la parte de herramientas en [4.3](#) para más información.

Con este formato, hemos podido usar el modelo con la librería de `ONNXRuntime` y así conseguir una inferencia mucho más rápida. Por otro lado, hemos usado esta misma librería para comprimir el modelo usando cuantización.

5.8. Compresión del modelo

Quantization

La cuantización es el proceso de reducir la precisión que tienen los pesos de las neuronas, así como los *bias*, con la intención de reducir el uso de memoria y el tamaño del modelo.

Para hacer esto transformamos el tipo de dato de los pesos, que normalmente es un *float* de 32 *bits*, a un tipo que ocupe mucho menos espacio, como por ejemplo puede ser un *int* de 8 *bits*.

En este movimiento de transformar de 32 a 8 *bits*, estamos reduciendo el tamaño 45 veces, reduciendo de forma significativa el consumo de memoria.

La pega que tiene esto, es que el *accuracy* se puede ver ligeramente rebajado. Aquí es donde tenemos que estudiar si merece la pena la reducción de tamaño y uso de memoria en contra de la pérdida de acierto.

En la figura 5.5 podemos observar la diferencia de tamaño entre el modelo en formato «.onnx» antes y después de cuantizar.

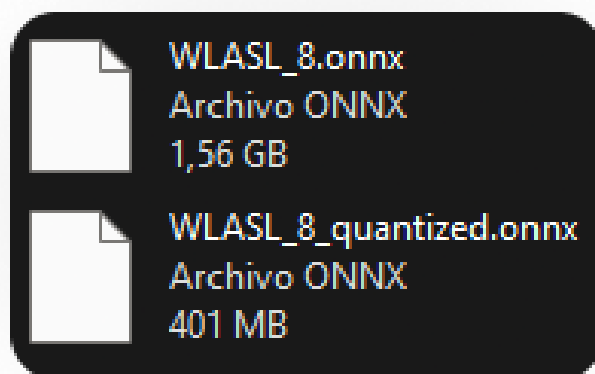


Figura 5.5: Modelo exportado antes y después de cuantizar con `ONNXRuntime`

Un ejemplo claro de esa pérdida de *accuracy* la vemos a la hora de inferir el mismo video del dibujo animado del apartado anterior. Esta vez en vez de inferir correctamente «Book», infiere «Before».

Por otro lado, la cuantización nos ha permitido poder subir el modelo contenedorizado de forma más sencilla y rápida.

5.9. Repasemos un experimento

Voy por último a hablar de uno de los experimentos realizados en el proyecto.

Antes de pasar a crear la CNN para clasificación de video, y cuando estaba en la fase en la que los datos de entrada eran únicamente *frames*; pensé que podía incluir la secuencia de *frames* de un video en una única imagen.

Podemos ver uno de los ejemplos en la figura 5.6.

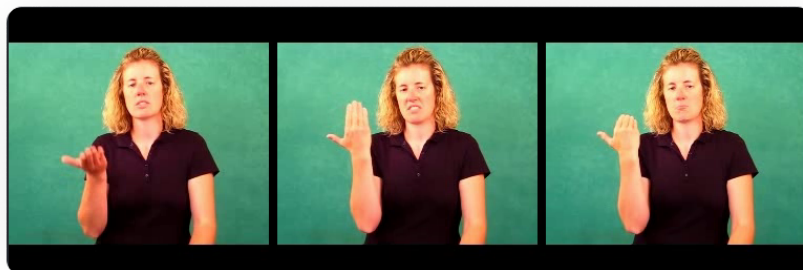


Figura 5.6: Concatenación de frames con la intención de representar un video (gesto es «Before»). Imágenes del dataset WLASL [31]

Esto fue un auténtico fracaso, el *accuracy* de la red era mínimo, y cuando concatenábamos números *frames* de una imagen, acabábamos con un tensor único enorme.

El problema principal es que una entrada de este estilo confunde mucho a una red convolucional, ya que esta intenta buscar características espaciales en la dimension, pero se está encontrando con muchos problemas:

1. Píxeles «muertos»: Para empezar, al concatenar la imagen, hemos añadido píxeles negros entre *frames*. El *kernel* va a pasar por ellos y no va a recibir ningún tipo de información espacial.
2. Demasiada información para una dimensión: Estamos dando información sobre las características espaciales de la imagen al mismo tiempo que buscamos encontrar el movimiento de los gestos entre *frames*. Para que esto funcione correctamente, el movimiento de los gestos entre *frames* debe estar en otra dimension, y que sea el *kernel* de esa dimensión el que produzca la convolución que detecte los movimientos.

Tras esto, pudimos comprender que necesitábamos pasarnos a la clasificación de un video, uniendo los *frames*, no como una imagen única, sino como una nueva dimensión en el tensor de entrada.

Trabajos relacionados

Este apartado sería parecido a un estado del arte de una tesis o tesina. En un trabajo final grado no parece obligada su presencia, aunque se puede dejar a juicio del tutor el incluir un pequeño resumen comentado de los trabajos y proyectos ya realizados en el campo del proyecto en curso.

Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

Bibliografía

- [1] Compara los servicios de aws y azure con google cloud | programa gratuito de google cloud. <https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison>. (Accessed on 07/02/2022).
- [2] 2022.
- [3] 2022.
- [4] Corrado Alessio. Animals-10, 2019.
- [5] Ethem Alpaydin. *Machine learning*. MIT Press, 2021.
- [6] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1):105–139, 1999.
- [7] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.
- [8] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.
- [9] H Dv Block, BW Knight Jr, and Frank Rosenblatt. Analysis of a four-layer series-coupled perceptron. ii. *Reviews of Modern Physics*, 34(1):135, 1962.
- [10] Steven Busuttil. Support vector machines. 2003.
- [11] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):1–13, 2020.

- [12] Yin Cui, Feng Zhou, Jiang Wang, Xiao Liu, Yuanqing Lin, and Serge Belongie. Kernel pooling for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2930, 2017.
- [13] Colaboradores de. Tensor, Feb 2004.
- [14] Chuong B Do and Serafim Batzoglou. What is the expectation maximization algorithm? *Nature biotechnology*, 26(8):897–899, 2008.
- [15] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2):103–130, 1997.
- [16] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- [17] L. Fausett and L.V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall international editions. Prentice-Hall, 1994.
- [18] Peter Flach. Performance evaluation in machine learning: the good, the bad, the ugly, and the way forward. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9808–9814, 2019.
- [19] Ali Ghodsi. Dimensionality reduction a short tutorial. *Department of Statistics and Actuarial Science, Univ. of Waterloo, Ontario, Canada*, 37(38):2006, 2006.
- [20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Nets*. Universite de Montréal, 2014.
- [21] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahrourdy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.
- [22] Bulbul Gupta, Pooja Mittal, and Tabish Mufti. A review on amazon web service (aws), microsoft azure & google cloud platform (gcp) services. 2021.
- [23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. arXiv, Dec 2015.

- [24] Geoffrey Hinton, Simon Osindero, Max Welling, and Yee-Whye Teh. Unsupervised discovery of nonlinear structure using contrastive back-propagation. *Cognitive science*, 30(4):725–731, 2006.
- [25] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
- [26] Tha Asimov Institute. the neural network zoo the asimov institute 2016, Sep 2016.
- [27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2012.
- [28] Yann Le Cun, Lawrence D Jackel, Brian Boser, John S Denker, Hans Peter Graf, Isabelle Guyon, Don Henderson, Richard E Howard, and William Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.
- [29] Yann Lecun, L Eon Bottou, Yoshua Bengio, and Patrick Haaner Abstract|. Gradient-based learning applied to document recognition. *PROC. OF THE IEEE*, 2006.
- [30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [31] Dongxu Li, Cristian Rodriguez, Xin Yu, and Hongdong Li. Word-level deep sign language recognition from video: A new large-scale dataset and methods comparison. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 1459–1469, 2020.
- [32] Pytorch main Mantainer. Pytorch developers guideline and design philosophy, 2022.
- [33] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(4):140–147, 2020.
- [34] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

- [35] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Number 9. McGraw-hill New York, 1997.
- [36] Kevin P Murphy et al. Naive bayes classifiers. *University of British Columbia*, 18(60):1–8, 2006.
- [37] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [38] Suprianto Panjaitan, Muhammad Amin, Sri Lindawati, Ronal Watrianthos, Hengki Tamando Sihotang, Bosker Sinaga, et al. Implementation of apriori algorithm for analysis of consumer purchase patterns. In *Journal of Physics: Conference Series*, volume 1255, page 012057. IOP Publishing, 2019.
- [39] J Quinlan. Building classification models: Id3 i c4. 5. *Dane udostepnione pod adresem: <http://yoda.cis.temple.edu>*, 8080, 1993.
- [40] J Ross Quinlan et al. Bagging, boosting, and c4. 5. In *Aaai/Iaai*, vol. 1, pages 725–730, 1996.
- [41] Shoba Ranganathan, Kenta Nakai, and Christian Schonbach. *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*. Elsevier, 2018.
- [42] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.
- [43] Paul Rodriguez, Janet Wiles, and Jeffrey L Elman. A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40, 1999.
- [44] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.
- [45] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.
- [46] Seldon. Outlier detection and analysis methods - seldon, Jul 2021.
- [47] Karen Simonyan and Andrew Zisserman. *Published as a conference paper at ICLR 2015 VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*. 2015.

- [48] Kristina P Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE access*, 8:80716–80727, 2020.
- [49] Zixing Song, Xiangli Yang, Zenglin Xu, and Irwin King. Graph-based semi-supervised learning: A comprehensive review. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
- [50] Carlos Oscar Sánchez Sorzano, Javier Vargas, and A Pascual Montano. A survey of dimensionality reduction techniques. *arXiv preprint arXiv:1403.2877*, 2014.
- [51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. *Going deeper with convolutions*. 2014.
- [52] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.
- [53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [54] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.
- [55] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.