



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 deep
learning**



Presentado por Iván Ruiz Gázquez
en Universidad de Burgos — 6 de julio de 2022
Tutores: Dr. Daniel Urda Muñoz y Dr. Bruno
Baruque Zanon



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



Dr. Daniel Urda Muñoz, profesor del área de Ciencia de la Computación e Inteligencia Artificial, del departamento de Ingeniería Informática.

y

Dr. Bruno Baruque Zanon, profesor del área de Ciencia de la Computación e Inteligencia Artificial, del departamento de Ingeniería Informática.

Exponen:

Que el alumno D. Iván Ruiz Gázquez, con DNI 71311599F, 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, 6 de julio de 2022

Vº. Bº. del Tutor:

Vº. Bº. del co-tutor:

Dr. Daniel Urda Muñoz

Dr. Bruno Baruque Zanon

Resumen

En la actualidad, no existe ningún traductor de lenguaje de signos a texto que funcione a tiempo real y sea capaz de traducir palabras complejas u oraciones extensas. En este proyecto se busca crear un modelo que mediante las técnicas de *deep learning* comunent empleadas, transcriba el lenguaje de signos que se visualiza en un video a texto. La transcripción se realizará a tiempo real y nos apoyaremos en *datasets* de dominio público para mapear los gestos a palabras individuales. Preprocesaremos los datos de vídeo para adaptarlos a la entrada del modelo de *deep learning*.

El objetivo principal es habilitar la comunicación entre personas que usan lenguaje de signos y personas que no lo conocen. Adicionalmente, se pretende crear un modelo que pueda ser fácilmente adaptable a otros *datasets* de vídeo y que permita la futura traducción de signos en múltiples idiomas.

Tras esto, queremos implementar una plataforma o demo web en la que los usuarios puedan probar el modelo con vídeos propios. Se habilitarán medios para que cualquier usuario y desarrollador pueda usar y mejorar el proyecto con el objetivo de que Sign2Text llegue al mayor número de manos posibles.

Descriptores

aprendizaje automático, aprendizaje supervisado, aprendizaje profundo, redes neuronales convolucionales, transcripción de lenguaje de signos, procesado de imagen y vídeo, pytorch

Abstract

Currently, there is no sign language to text translator that works in real time and is capable of translating complex words or long sentences. In this project we seek to create a model that, by means of the deep learning techniques commonly used, transcribes the sign language displayed in a video to text. The transcription will be performed in real time and we will rely on public domain datasets to map gestures to individual words. We will preprocess the video data to adapt it to the input of the deep learning model.

The main goal is to enable communication between people who use sign language and people who do not know it. Additionally, we aim to create a model that can be easily adapted to other video datasets and that allows the future translation of signs from multiple sign languages.

After this, we want to implement a web platform or demo in which users can test the model with their own videos. Means will be provided so that any user and developer can use and improve the project with the aim of Sign2Text reaching the largest number of hands possible.

Keywords

machine learning, supervised learning, deep learning, convolutional neural networks, sign language transcription, image and video processing, pytorch

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	vi
Introducción	1
1.1. Estructura del trabajo	2
Objetivos del proyecto	5
2.1. Objetivos teóricos	5
2.2. Características esperadas	6
Conceptos teóricos	9
3.1. Machine Learning	9
3.2. Supervised Learning	11
3.3. Deep Learning	18
3.4. Data Augmentation	25
Técnicas y herramientas	29
4.1. Lenguajes	29
4.2. Metodologías	31
4.3. Herramientas	32
4.4. Miscelánea	44
Aspectos relevantes del desarrollo del proyecto	45
5.1. Carga de datos	45
5.2. Tratamiento de los datos	48

5.3. Red Neuronal ResNet	50
5.4. Pruebas de arquitecturas	51
5.5. Experimento: Concatenación directa de imágenes	53
5.6. Aplicando clasificación con regresión	55
5.7. Exportación del modelo	56
5.8. Compresión del modelo	57
5.9. Estado actual del modelo	58
Trabajos relacionados	61
6.1. Sign2Text - <i>Letter level translation</i>	62
6.2. ASL Translator	62
6.3. <i>ASL to english</i> - traducción a nivel palabra	64
6.4. How2Sign	65
Conclusiones y Líneas de trabajo futuras	67
7.1. Conclusiones	67
7.2. Líneas futuras	68
7.3. Experiencia personal	69
Bibliografía	73

Índice de figuras

3.1. Tipos de <i>machine learning</i> según datos, inferencia estadística y técnicas de aprendizaje	10
3.2. Visualización de <i>overfitting</i> 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.	12
3.3. Estructura de una red RNN y CNN	20
3.4. Estructura de una CNN. Figura de <i>Le Cun et al.</i> [30]	21
3.5. Ejemplo de convolución con <i>kernel 3x3</i> sin <i>striding</i> ni <i>padding</i>	22
3.6. Ejemplo de <i>data augmentation</i> aplicando <i>adversarial neural networks</i> , más concretamente <i>CycleGANs</i> . Extraído de [54]	26
4.1. Ejemplo de <i>transform</i> en PyTorch	34
4.2. Estructura de un modelo con CNNs en formato ONNX . Visualización con Netron	35
4.3. Ejemplo de código de detección de la anchura de un botón con Svelte (izquierda) y HTML (derecha)	42
5.1. Estructura de carpetas necesaria para la carga de datos con PyTorch	47
5.2. Estructura de una CNN exportada con Netron	52
5.3. Estructura de una CNN con doble salida exportada con Netron	53
5.4. Concatenación de <i>frames</i> con la intención de representar un vídeo (gesto es «Before»). Imágenes del <i>dataset</i> WLASL [31]	54
5.5. La red es capaz de adivinar un signo hecho por un dibujo animado	56
5.6. Modelo exportado antes y después de cuantizar con ONNXRuntime	58

Índice de tablas

4.1. Diferencias entre Tensorflow y PyTorch	36
5.1. Accuracy de una red ResNet con un dataset pequeño de imágenes de animales [3]	50
5.2. Accuracy de una red ResNet con nuestro dataset original con 10 etiquetas, 30 videos por etiqueta, 1 solo frame por video	51
5.3. Accuracy actual tras la combinación de clasificación y regresión	58

Introducción

Antes de empezar, me gustaría agradecer: A todos los tutores, la paciencia que han tenido conmigo. Desde el primer día me han explicado lo más básico de la mejor manera posible, aportando fuentes y ayudando en todo. Muchas gracias a Bruno y Daniel, los tutores de la UBU, y muchas muchísimas gracias a María, Alejandro y Pablo, los tutores de HP SCDS, no se que habría hecho sin vosotros.

En este proyecto vamos a implementar una transcripción automática de lenguaje de signos a texto a nivel palabra usando técnicas de inteligencia artificial.

El lenguaje de signos es una forma de comunicación por gestos gracias a la cual más de 72 millones de personas con discapacidad auditiva pueden comunicarse. Hay más de 300 lenguas de signo distintas y dentro de un mismo lenguaje hay varios gestos que pueden significar palabras distintas dependiendo del contexto.

En la actualidad existen transcriptores a nivel letra o nivel palabra, solo que estos transcriptores solo detectan gestos estáticos, que no tienen movimiento. No existe ningún traductor de lenguaje de signos que sea capaz de transcribir gestos con movimiento incluido.

Sign2Text tiene el propósito de crear un transcriptor a tiempo real usando redes neuronales, *deep learning* y procesamiento de imagen y vídeo. Para esto se ha usado un *dataset* muy completo de ASL (*American Sign Language*) [31] compuesto por secuencias cortas de vídeo e información

de poses de distintos sujetos. Se van a aplicar técnicas de *deep learning* como redes neuronales convolucionales, *transformers* y procesado 3D (dos dimensiones para la imagen y otra para el tiempo).

Tras crear la red que transcriba el texto, se pretende crear una demo *web* y *API* para que cualquier usuario alrededor del mundo pruebe probarlo. También se busca poder encapsular el modelo en un contenedor y dejarlo disponible para que cualquier desarrollador pueda descargarlo y crear su propia aplicación.

1.1. Estructura del trabajo

- **Objetivos del trabajo:** En este apartado vamos a listar los objetivos que queremos cumplir en la realización del proyecto, así como las características que esperamos cumpla nuestro modelo de *deep learning*.
- **Conceptos teóricos:** En este apartado vamos a hacer un *overview* de *machine learning*, comenzando por el aprendizaje no supervisado, siguiendo del semi-supervisado y terminando en lo que más no interesa, el supervisado y el *deep learning*. Veremos las principales técnicas en cada tipo de aprendizaje para conocer cuales son las mejores opciones en el desarrollo del proyecto.
- **Técnicas y herramientas:** Aquí listaremos y explicaremos todas las herramientas usadas en la realización de **Sign2Text**. También comentaremos un poco como las hemos usado nosotros y que alternativas tenemos en cada caso. Hablaremos de las herramientas con las que desarrollaremos nuestro modelo y con las que crearemos nuestro *front-end* y *back-end*.
- **Aspectos relevantes del desarrollo:** En este apartado hablaremos sobre experimentos y experiencias del proyecto. Aquí contaremos las mejoras que se han hecho a medida que iteramos con el modelo, así como curiosidades y problemas con los que nos hayamos encontrado.
- **Trabajos relacionados:** Esta sección pretende mostrar otras herramientas y proyectos similares a **Sign2Text** disponibles en la comunidad y el mercado.
- **Conclusiones y líneas futuras:** Por último haremos un repaso de las mejoras que se pueden realizar en el proyecto. Hablaremos de

las conclusiones que hemos sacado al finalizarlo y repasaremos que porcentaje de los objetivos hemos cumplido al finalizarlo.

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.

2.1. Objetivos teóricos

1. Crear 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. Hacer disponible y encapsular el modelo en un contenedor Docker para su libre distribución.
5. Crear de herramientas que permitan el tratamiento de datos y la transformación entre distintos formatos³.
6. Estudiar del comportamiento, estructura e hiperparametrización de redes neuronales convolucionales.

¹*Deep Learning*

²*American Sign Language*

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

7. Aprender a fondo el uso de PyTorch[32], un *framework* para acelerar la creación y prototipado de redes neuronales.
8. Exportar del modelo entrenado a un formato estándar que facilite la compatibilidad con el mayor número de librerías en distintos lenguajes de programación.
9. Mantener 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:

2.2. Características esperadas

- el sistema debe ser capaz 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 vídeo). 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 vídeo 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 *ResNets*, una arquitectura sobre las CNN. Veamos en detalle todos estos términos en los siguientes apartados.

3.1. Machine Learning

Según Tom Mitchell [36], 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 la experiencia 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 [4]: 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»

tienen 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) que otorgamos al «programa»: *supervised*, *unsupervised* y *semi-supervised learning*. 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 únicamente en *supervised learning*.



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

3.2. Supervised Learning

También llamado *classification*¹⁰ 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.*[7], 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 conseguir datos para las tres fases dividiremos nuestro *dataset* en tres grupos respectivamente. Con la intención de 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 fase¹¹. Pero hay que tener cuidado, porque asignar un porcentaje demasiado alto de los datos a la fase de entrenamiento puede derivar en *overfitting* en el modelo.

El *overfitting* ocurre cuando un modelo se ha sobre-entrenado con unos datos, por que 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»: los datos de validación. De este modo iremos comprobando la deriva de las predicciones entre los datos de entrenamiento y los datos de validación.

Ahora podremos controlar la fase de entrenamiento y parar dicho entrenamiento, valga la redundancia, cuando detectemos este sobre-ajuste. Podemos ver esto en la figura 3.2. Gracias al *set* de validación, detectamos el momento en el que la red comienza a sobre-ajustarse. 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.

¹⁰A pesar que no solo trate problemas de clasificación

¹¹Normalmente un 70 % de los datos se asigna a la fase de entrenamiento, aunque depende mucho del problema específico al el que nos enfrentemos

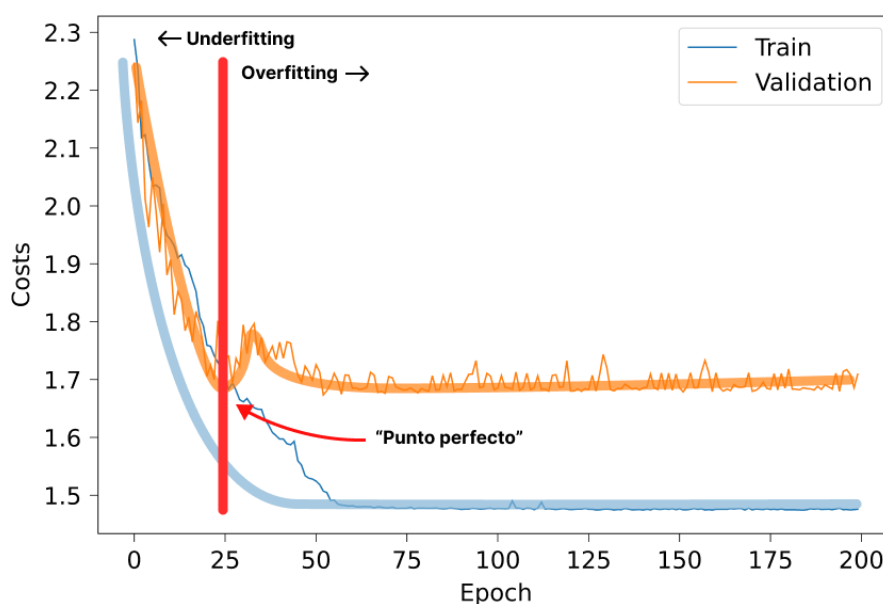


Figura 3.2: Visualización de *overfitting* en un entrenamiento de una CNN (red neuronal convolucional). Podemos observar en rojo el punto exacto en el que la red comienza a sobre-ajustarse. Este es el momento en el que debemos parar el entrenamiento.

En la imagen vemos que el valor del eje de ordenadas tenemos el valor de los costes 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 [34].

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 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 (2 etiquetas) 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 moviéndonos 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 1: 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 [5].

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. **Support Vector Machines (SVM):** Según Van Engelen *et al.* [50] 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 ella¹².

Los SVM también están presentes en la regresión. En el caso de la clasificación, tenemos un conjunto de puntos en un espacio de dimensión 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. 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* [9]

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

¹²A veces el margen también se refiere a la zona delimitada por la frontera de decisión en la que caen los puntos

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 [37]. 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 [14] 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 [6]. 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 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 el *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 [19]. 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 Networks*) 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) [8] 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 ¹³. Calculamos la salida de la derivada en el *forward pass*, y con el resultado obtenido, realizamos un *backward pass*, con el que propagamos y actualizamos los pesos de la red.

Esto ahora nos parece básico, pero fue 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* ¹⁴ 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) ¹⁵ 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].

¹³Recordemos 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

¹⁴Una 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

¹⁵Como FNNN o perceptrón

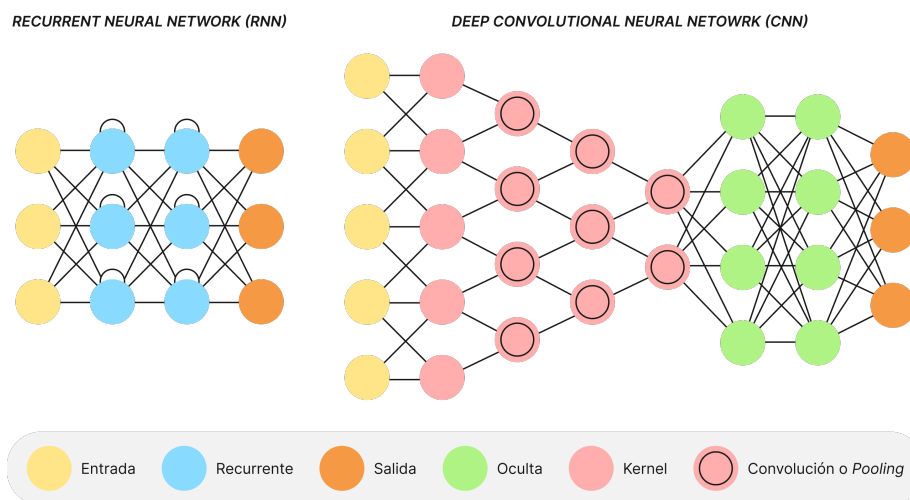


Figura 3.3: Estructura de una red RNN y CNN

En la figura 3.3, 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).

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 más de profundidad a continuación.

- **RNN:** Las Redes Recurrentes son redes que tuvieron su auge en investigación en los años 90. *Fausett* (1994) [18] 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* [35].

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 vídeo o audio ¹⁶. 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 vídeo e imagen.

Las redes convolucionales tienen una estructura muy diferente a cualquier otra red neuronal (figura 3.4). Se componen de capas de convolución, capas de *subsampling* y capas «*fully-connected*».

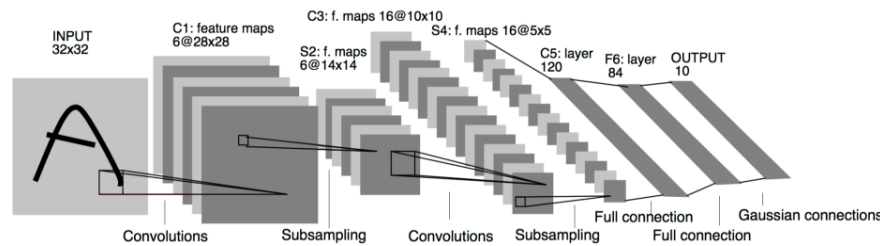


Figura 3.4: Estructura de una CNN. Figura de *Le Cun et al.*[30]

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 vídeo) 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 vídeos. Por esa razón, la extracción de características de una imagen se hacía antes de forma manual en la fase de preparación de los datos.

¹⁶Aunque también podemos tratar texto o imagen como concatenación de letras o píxeles en el tiempo

Para eliminar esta fase manual, las CNN actúan directamente en matrices de datos multidimensionales (o tensores ¹⁷), 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 el tensor de entrada de la red, 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 vídeo 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 «mover» nuestro *kernel* por nuestro tensor de entrada, generando un «mapa de características». Veamos un ejemplo:

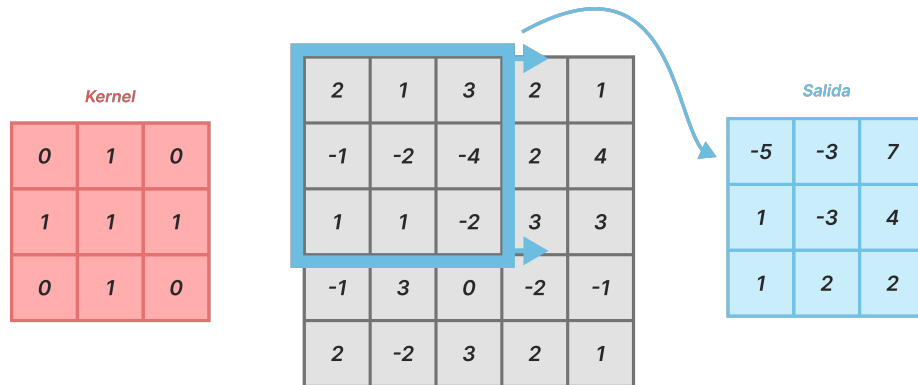


Figura 3.5: Ejemplo de convolución con *kernel* 3×3 sin *striding* ni *padding*

Vemos en la figura 3.5 que tenemos un tensor bidimensional (i.e. una imagen) de tamaño (5×5) con un *kernel* de tamaño (3×3) . El mapa de características resultante tiene un tamaño de (3×3) ¹⁸. 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

¹⁷Un tensor es un objeto algebraico que representa un espacio vectorial. Puede mapear vectores y matrices multidimensionales [13]

¹⁸Da 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 (5×7) obtendríamos una salida de (3×5)

de entrada, es posible que tengamos tamaños cuyo *kernel* no «encaje» de forma perfecta.

Por ejemplo, si nuestra matriz fuera de (4×4) en vez de (5×5) , 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» en 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 [38].
- 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 *stride* demasiado alto, podemos saltarnos extracciones de características importantes de nuestro tensor.

Podemos calcular el tamaño de salida con la siguiente fórmula [38]:

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

donde D es 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 $\delta = \lceil \frac{5+2 \cdot 0 - 3}{1} + 1, \frac{5+2 \cdot 0 - 3}{1} + 1 \rceil = [3, 3]$. 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 cantidad 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.¹⁹

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 3.3. 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 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, x el vector de entrada y f la función de activación no lineal.

El nombre de esta capa²⁰ se debe a que las neuronas están totalmente conectadas entre sí entre las distintas capas (no entre neuronas de la misma capa ni consigo mismas) [25].

La salida de la capa *fully connected* 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 vídeo. 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

¹⁹Denotar que lo importante del *kernel* en el *pooling* son sus dimensiones ya que no vamos a realizar operaciones con él

²⁰También se suele llamar capa densa

problemas. Algunas son: ALEXNET [27], LENET[29], VGGNET [48], GOOGLNET [49], ZFNET [53] 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) [51]. 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).

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)

3.4. Data Augmentation

Las redes neuronales convolucionales, como hemos visto en los apartados anteriores, tienden a funcionar extraordinariamente bien en tareas de clasificación de vídeo e imagen, pero, en contrapartida, necesitan gran cantidad de datos para evitar *overfitting* [47].

Desgraciadamente, son muchas las ocasiones en las que no podemos usar datos de entrenamiento suficientes para evitar *overfitting*, como en el caso de datos médicos. En estos casos, aplicamos aumento de datos.

Data augmentation o aumento de los datos, agrupa un conjunto de técnicas con las cuales buscamos extender el conjunto de datos de entrenamiento de nuestros modelos, en caso de que sean insuficientes.

Las principales técnicas de aumento de datos son la transformación geométrica, las transformaciones en el espacio de color, aplicación de filtros *kernel* (como hemos visto en la convolución), mezcla de distintas imágenes o vídeos, borrado aleatorio, aumento del espacio de características, transferencia de estilo y redes generativas adversarias (GANs).

El aumento de datos busca un acercamiento al *overfitting* desde la raíz principal del problema: el *dataset*. Esto es asumiendo que el *dataset* tienen datos que pueden ser aumentados.

Estas aumentaciones artificiales inflan el *dataset* en tamaño de dos formas: por deformación de datos (*data warping*) o por *oversampling* (sobremuestreo).

Las deformaciones de datos transforman los datos existentes preservando su etiqueta. Algunos tipos son:

1. Transformación geométrica y/o de color
2. Eliminación parcial aleatoria de datos espaciales
3. Transferencia de estilo (mediante redes adversarias generativas (GANs)).
Podemos ver un ejemplo en la figura 3.6.

Por otro lado, el supersampleo consiste en mantener los datos actuales y añadir nuevas instancias al *set* de entrenamiento. La aplicación de *oversampling* y *data warping* no es una dicotomía mutuamente exclusiva.



Figura 3.6: Ejemplo de *data augmentation* aplicando *adversarial neural networks*, más concretamente *CycleGANs*. Extraído de [54]

Lo que hace el reconocimiento de imágenes y vídeo un problema tan complejo en *deep learning* es que el modelo de reconocimiento debe

sobrellevar problemas del punto de visión, la luz, la oclusión ambiental, el color del fondo, la escala de los elementos, entre otros. El aumento de datos consiste en internalizar estas pequeñas variaciones de movimiento y color en los datos, de tal modo que, una vez entrenado el modelo, funcione en todas las situaciones posibles.

Pero el *data augmentation* no es la única técnica que se puede aplicar para evitar *overfitting*. Otra técnica bastante común (y que nosotros hemos aplicado) es el denominado *batch normalization*. El *batch normalization* es una técnica de regularización de los datos que normaliza los *sets* de las capas de activación de una red neuronal. Esta normalización funciona restando la media de cada activación y dividiendo entre la desviación estándar de todos los valores del *batch* [47].

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 ciencia de datos 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 un rendimiento similar a **NodeJS** o **Go** [2].

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).

- Accede al *framework* tailwind desde <https://tailwindcss.com/>

JavaScript

Con **HTML** y **CSS** podemos crear una *landing page* simple (aunque con mucho estilo), 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 estándar **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**.

- Puedes acceder a la web de astro en: <https://astro.build>
- Svelte se puede visitar en <https://svelte.dev>

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. Estas son:

PyLint

PyLint es un 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 opinionado 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 inspector de tipos estático para **Python**. Esta herramienta nos ayuda a comprobar que todos los tipos de nuestras 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 Tortoise)

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 de *Open Source*.

GitLab

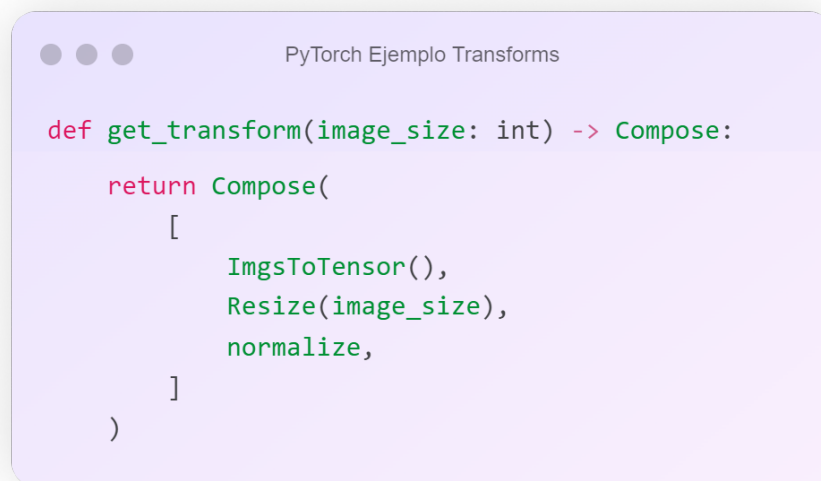
Gitlab es una alternativa a Github. Las características ofrecidas son prácticamente las mismas. Una de las diferencias, aunque es mínima, es que Gitlab ofrece mayor compatibilidad a la hora de hacer *continuous deployment* con los servicios de Google.

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 `Tensorflow`. `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 vídeos). 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 4.1 para una lista más completa).

ONNX

ONNX (*Open Neural Network Exchange*) es un formato abierto construido 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 nuestros 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.

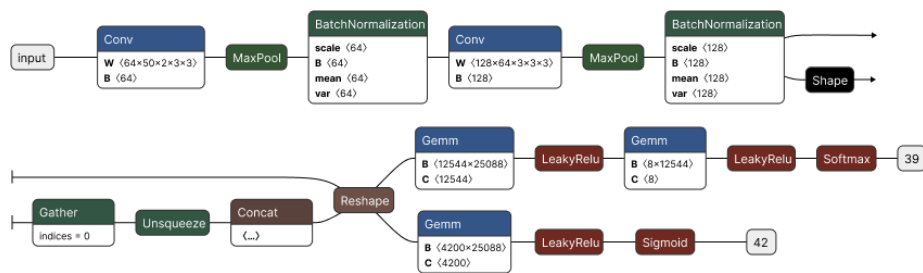


Figura 4.2: Estructura de un modelo con CNNs en formato ONNX. Visualización con Netron

	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.

- Accede a la web de ONNX en <https://onnx.ai/>
- Al app de Netron está disponible en: <https://netron.app/>

Docker

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

Los contenedores se construyen a traves 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>
- ¿Deseas probarlo directamente? Accede en <https://api.sign2text.com/docs>
- La demo web se encuentra disponible en <https://sign2text.com>

- Videos para probar disponibles en <https://bit.ly/3bPWve6>

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.

- Librería *gdown*: <https://github.com/wkentaro/gdown.git>
- Repositorio *Sign2Text-API* con la 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 inútil 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 (Cliente)

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 vídeo a la pantalla. Tras arrastrar el vídeo, se muestra una *preview* con un botón. Al pulsar el botón se envía el vídeo a nuestro *back-end* para ser procesado. El usuario recibe entonces el signo identificado para el vídeo. 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 (Servidor)

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. Miscelánea

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 polifacético 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 vídeo. 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
- Extracción de frames
- Extracción de video

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*, estábamos 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.

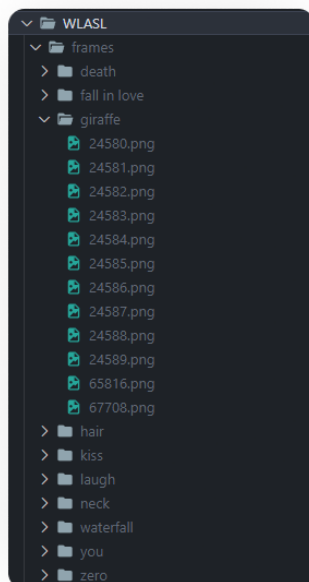


Figura 5.1: Estructura de carpetas necesaria para la carga de datos con PyTorch

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

($F \times 3 \times H \times W$) 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 ²¹.

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 parte más importantes en el aprendizaje automático 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 (224×224) (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 cuando estamos tratando entardas con datos que se sitúan en rangos muy distintos ²².

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

²¹El 3 identifica los tres canales RGB de una imagen en color

²²i.e.: un valor enumerado entre 0 y 5 y un valor continuo entre 100 y 200

255), por lo que al no mezclar con otro 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*, como hemos visto en los conceptos teóricos, 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 tratando con videos. Al tener la transformación de los datos a nivel imagen (*resize* y normalización), se nos complicaba el aumento 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.

En el caso de *SimpleNet*, hemos aplicado *data augmentation* con un *random crop*.

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, marcaría una extensión notable.

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 $(F \times 3 \times H \times W)$ y que representaba un video de nuestro dataset. Ahora añadimos una dimensión más B representando 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 ²³.

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 [3] 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 [3]

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

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

²³Una 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

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. Pruebas de arquitecturas

A media que avanzaba el proyecto, probamos con redes preentrenadas *ResNet*, que daban buenos resultados al ser aplicadas con imágenes. Esto funcionaba a la perfección, pero llegaría el momento en el que tendríamos que comenzar a clasificar entradas de vídeo.

En este punto, decidimos crear nuestra propia red convolucional desde cero y aplicar una estructura personalizada. La estructura de nuestra primera red convolucional (denominada «SimpleNet»), usada para clasificación de video, es algo 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.

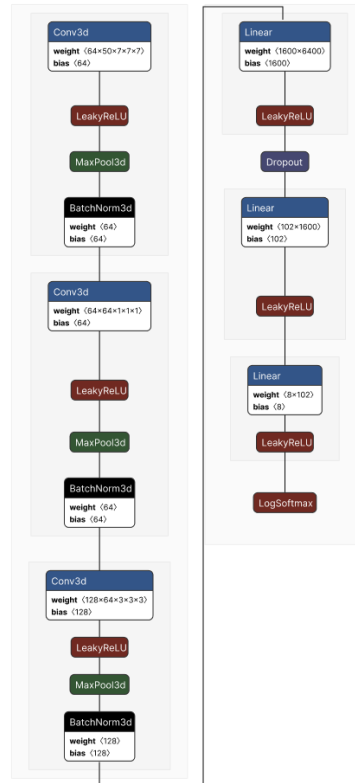


Figura 5.2: Estructura de una CNN exportada con *Netron*

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.

5.5. EXPERIMENTO: CONCATENACIÓN DIRECTA DE IMÁGENES

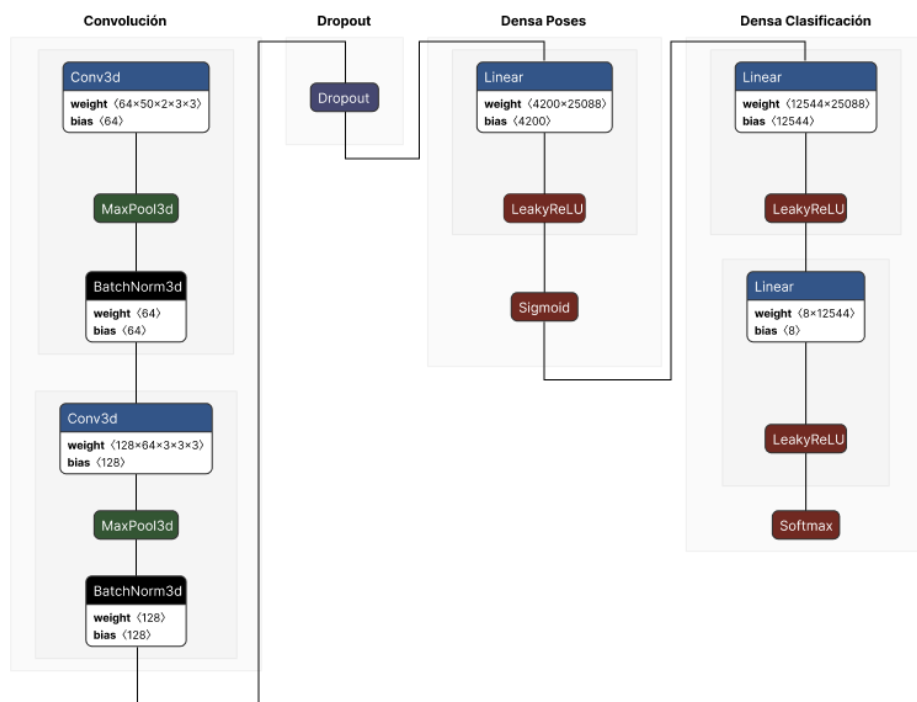


Figura 5.3: Estructura de una CNN con doble salida exportada con *Netron*

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

5.5. Experimento: Concatenación directa de imágenes

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

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

Podemos ver uno de los ejemplos en la figura 5.4.

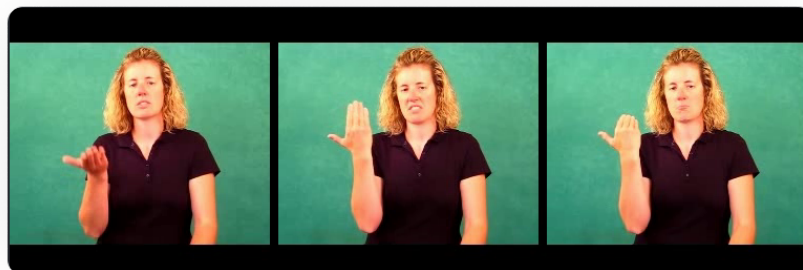


Figura 5.4: Concatenación de *frames* con la intención de representar un vídeo (gesto es «Before»). Imágenes del *dataset* WLASL [31]

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 dimensión, 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 dimensión, 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 vídeo, uniendo los *frames*, no como una imagen única, sino como una nueva dimensión en el tensor de entrada.

5.6. Aplicando clasificación con regresión

Las redes descritas en los apartados anterior estaban compuestas de una salida con *softmax* de clasificación de signos y una salida con sigmoide para regresión de cálculo de poses y marcas de movimiento.

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.

Para comprobar que la mejora en *accuracy* proviene de la nueva funcionalidad y no del factor aleatorio del entrenamiento, vamos a probar con videos que se salgan de lo normal ²⁴, como videos de dibujos animados o videos distorsionados, como vemos en la figura 5.5.

²⁴Con normal nos referimos a videos extraidos del *set* de test del *dataset*

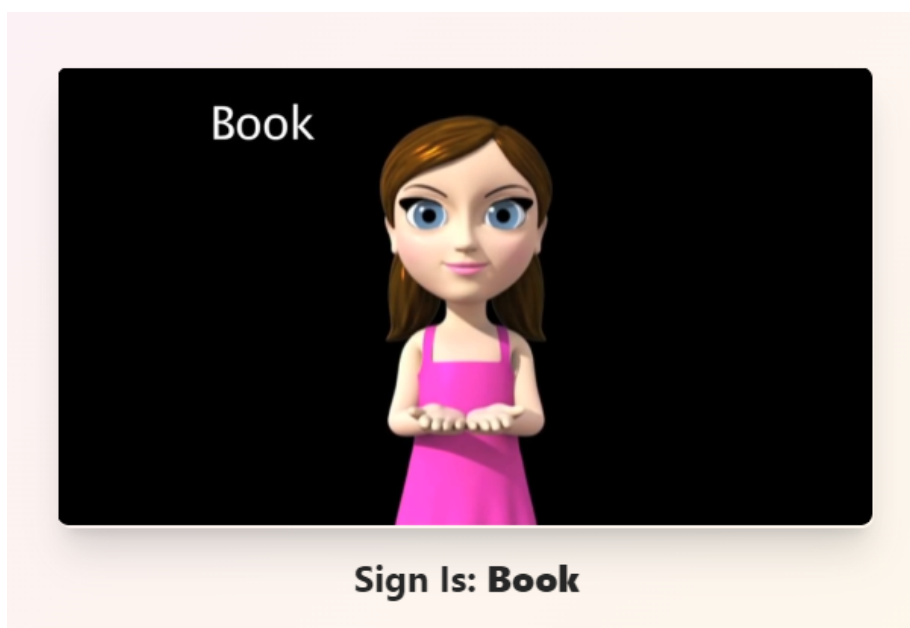


Figura 5.5: La red es capaz de adivinar un signo hecho por un dibujo animado

Una idea muy parecida se puede ver en [33]. En este *paper* proponen usar numerosas entradas de la red junto con datos de poses. Proponen introducir la imagen completa, junto a la imagen por separado de las manos y la cara y por otro lado pretenden también introducir información de poses del cuerpo completo.

Por otro lado, en [17] podemos ver el acercamiento basado únicamente en poses. En este caso usa una DRNN (*Deep Recurrent Neural Network*).

Nota: El *dataset* de las poses está proporcionado por los mismo investigadores de WLASL [31]. Este *dataset* está a su vez generado usando *PoseNet*, un modelo de generación de poses bastante renombrado [10][52].

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 estándar.

ONNX

El formato estándar 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 4 veces, disminuyendo 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 5.6 podemos observar la diferencia de tamaño entre el modelo en formato «.onnx» antes y después de cuantizar.

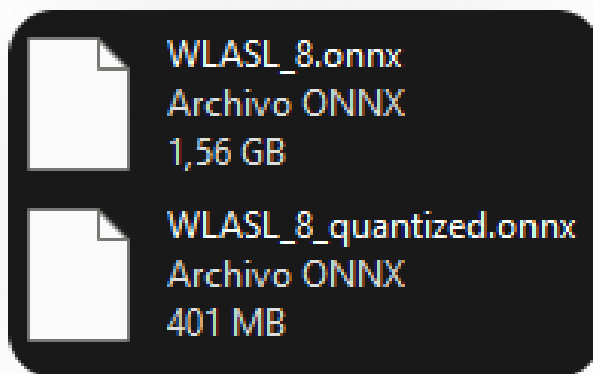


Figura 5.6: Modelo exportado antes y después de cuantizar con `ONNXRuntime`

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. Estado actual del modelo

El modelo ha sido entrenado con 8 etiquetas (*all*, *before*, *book*, *drink*, *help*, *no*, *walk*, *yes*) y mantiene los siguientes *accuracies* (tabla tabla:accs)

Entrenamiento	Test	Validación
99,3 %	79,88 %	78,65 %

Tabla 5.3: Accuracy actual tras la combinación de clasificación y regresión

Este modelo es accesible en DockerHub. Se puede consumir desde la API abierta o desde la demo en la web. El código es *Open Source* en Gitlab y Github.

- **Dockerhub:** <https://hub.docker.com/repository/docker/gazquez/sign2text>
- **API Abierta:** <https://api.sign2text.com/docs>
- **Demo Web:** <https://sign2text.com/>
- **Repositorio del modelo:** <https://github.com/irg1008/Sign2Text.git>
- **Repositorio de la demo:** <https://github.com/irg1008/Sign2Text-Astro.git>
- **Repositorio del servidor:** <https://github.com/irg1008/Sign2Text-API.git>

Trabajos relacionados

En esta sección vamos a repasar trabajos similares al realizado en este proyecto. Se buscan redes neuronales con similar propósito, así como proyectos cuya idea central similar, usen *datasets* similares, etc.

Las características de este proyecto y las que vamos a nombrar para el resto por comparación son:

- **Nombre:** Sign2Text
- **Tipo (Red/*Dataset*/Otro):** Red
- **Nivel de traducción:** nivel palabra
- **Detección de movimiento en gestos:** sí
- **Lenguaje de entrada:** ASL
- **Lenguaje de salida:** texto (inglés)
- ***Dataset:*** *Dataset* de dominio público extraído de [31]. Compuesto de más de 20000 vídeos con +2000 palabras
- **Librería o *framework*:** PyTorch
- **Técnicas adicionales aplicadas:** *data augmentation*
- **Estructura o red neuronal aplicada:** *CNN*, propia

6.1. Sign2Text - *Letter level translation*

Este proyecto, con el mismo nombre que este proyecto, tiene 125 estrellas en Github y se trata de un traductor a tiempo real de lenguaje de signos ASL a texto a nivel de letra. Es capaz de traducir las letras del abecedario usando inferencia por webcam. Usa un modelo preentrenado RESNET50 y tiene el modelo alojado en una instancia en AWS.

Datos sobre el proyecto

- **Nombre:** Sign2Text
- **Tipo (Red/*Dataset*/Otro):** Red
- **Nivel de traducción:** nivel letra (letras del abecedario)
- **Detección de movimient en gestos:** no
- **Lenguaje de entrada:** ASL
- **Lenguaje de salida:** texto (inglés)
- ***Dataset:*** *Dataset* con imágenes de signos referenciando las letras del abecedario. Dataset no referenciado
- **Red implementada:** No implementada desde cero. Reentrenamiento sobre RESNET50
- **Librería o *framework*:** TensorFlow, Keras
- **Técnicas adicionales aplicadas:** Ninguna, *dataset* usado ya preprocesado
- **Estructura o red neuronal aplicada:** *ResNet* (preentrenado)

Podemos acceder al repositorio en <https://github.com/BelalC/sign2text.git>

6.2. ASL Translator

Esta aplicación, subida al App Store ya que usa el mismo *dataset* que hemos usado en este proyecto, es una *app* de traducción de texto a signos, es decir, todo lo contrario a la nuestra. Asegura:

1. Traducir más de 30000 palabras
2. Ser capaz de traducir frases complejas
3. Usar un algoritmo que mezcla los vídeos con los signos de forma que muestran oraciones complejas de forma suavizada y sin cortes.

Las características técnicas del proyecto son:

- **Nombre:** ASL Translator
- **Tipo (Red/*Dataset*/Otro):** Aplicación
- **Nivel de traducción:** nivel oración
- **Detección de movimiento en gestos:** no
- **Lenguaje de entrada:** texto (inglés)
- **Lenguaje de salida:** ASL compuesto de varios *clips* fusionados para crear un *video stream* sin cortes.
- ***Dataset*:** No se especifica, pero por las imágenes de la aplicación se puede observar el uso de *dataset*
- **Librería o *framework*:** No se especifica
- **Técnicas adicionales aplicadas:** Desconocido
- **Estructura o red neuronal aplicada:** Desconocido

No compete directamente con la idea de este proyecto pero parece un proyecto interesante y con un hueco en el mercado sin competencia.

Se puede acceder a la página de la *app* en <https://apps.apple.com/us/app/asl-translator/id421784745>

6.3. *ASL to english* - traducción a nivel palabra

Este proyecto, que cogió mucha fuerza después de aparecer en algún artículo y hacerse ligeramente viral en LinkedIn, tiene 674 estrellas en Github.

En este proyecto se traducen «6» palabras de lenguaje de signo a texto; eso sí, ninguna de las palabras se representa con un gesto en movimiento, todas son estáticas. Usa también una red preentrenada y una webcam como entrada para la inferencia.

La ficha técnica de este proyecto es la siguiente:

- **Nombre:** ASL to english
- **Tipo (Red/*Dataset*/Otro):** Red
- **Nivel de traducción:** nivel palabra
- **Detección de movimient en gestos:** no
- **Lenguaje de entrada:** texto (inglés)
- **Lenguaje de salida:** ASL
- ***Dataset*:** Propio
- **Librería o *framework*:** Tensorflow
- **Técnicas adicionales aplicadas:** Desconocido
- **Estructura o red neuronal aplicada:** *MobileNet* (preentrenado)

Como curiosidad, al comienzo de este proyecto, se mantuvo una conversación con la autora por *e-mail* para ver si se podría hacer una colaboración.

Se puede acceder al repositorio del proyecto en https://github.com/priyaanjalii0611/ASL_to_English.git

6.4. How2Sign

How2Sign [15] es un *dataset* de ASL no solo compuesto de imágenes y vídeos de gestos, sino que también aporta audios, transcripciones, mapas de profundidad y datos de poses.

Todos los vídeos están grabados usando un *chroma key* para mayor facilidad de cambio de fondo y las poses han sido calculadas en 3D en un estudio dedicado. Por esta razón el tamaño del *dataset* aumenta a 290GB.

- **Nombre:** How2Sign
- **Tipo (Red/*Dataset*/Otro):** Dataset
- **Nivel de traducción:** nivel palabra
- **Detección de movimiento en gestos:** sí
- **Lenguaje de *dataset*:** ASL
- **Técnicas adicionales aplicadas:** Grabación con *chroma key*, estudio de pose en 3D, transcripción de audio.

Con una licencia CC y sin permiso para uso comercial, el *dataset* se puede acceder en <https://how2sign.github.io/>

Conclusiones y Líneas de trabajo futuras

Vamos a analizar las conclusiones extraídas de la realización de este proyecto y ver que mejoras se pueden realizar de cara al futuro.

7.1. Conclusiones

De forma general, se han cumplido las características buscadas al inicio del proyecto.

- Disposición del modelo neuronal a la comunidad. Hemos podido crear un contenedor docker con el modelo listo para ser usado. También hemos creado una API abierta para que cualquier desarrollador la consuma en su aplicación de cliente. Y también hemos creado una web con una demo para que cualquier usuario lo use de forma sencilla.
- Se ha aprendido a usar el *framework* **PyTorch** de forma general, así como en detalle los módulos de *supervised learning* y tratamiento y procesamiento de imagen y vídeo.
- Hemos conseguido implementar una red convolucional desde cero, comprendiendo de forma profunda su estructura y las conexiones entre las distintas capas, así como la modificación de los vectores de salida para adaptarse a los distintos problemas.
- Se ha mejorado considerablemente la lectura de artículos científicos relacionados con el campo de la ciencia de datos y el aprendizaje

automático. Al finalizar, eramos capaces de analizar y comprender lo principal de un artículo y de este modo sustraer la información útil en el desarrollo del proyecto.

- Se ha ampliado el número de tecnologías conocidas en tanto en el campo de la ciencia de datos como en el desarrollo *software* en general.
- Hemos podido usar las últimas herramientas y servicio de *cloud* con las que hemos podido hacer *deploy* de un modelo que consume sustanciales recursos. Se ha aprendido a usar plataformas como AWS, Azure y Google Cloud, entre otras.

Y, aunque el modelo actual basado en CNNs funciona y es medianamente escalable, no es nada comparado con el *State of the Art* [17] [31] en la actualidad. Por esto, veamos como podemos orientar el proyecto en el futuro.

7.2. Líneas futuras

Ha habido pasos en la creación del modelo, en los que podría haber tomado mejores decisiones. Como en la carga y transformación de los datos, en la creación de la estructura de la CNN o en la elección de *schedulers* y *optimizers*.

Por lo tanto, quedan muchos aspectos que mejorar. Algunas son los siguientes:

- Cambio de *dataset*: Uno de los grandes aspectos que ha ralentizado las mejoras e iteraciones de la red ha sido el *dataset*. Aunque el número de vídeos del *dataset* es correcto, la cantidad de vídeos por etiqueta es algo corta. Por otro lado, los datos de poses tenían muchas veces datos nulos o valores a 0, que no aportaban información en el entrenamiento. En algunas ocasiones en las que he probado el modelo con *datasets* distintos, el *accuracy* ha subido una media de 25 % en test y validación. Por todo esto, el primer cambio que haría sería el *dataset*. el recambio perfecto es How2Sign, un *dataset* mucho más extenso con datos de profundidad, gestos en 3D y transcripciones de audio [15].
- Mejorar la estructura de la red: En las redes y estructuras sobre CNNs ya establecidas observamos estructuras de decenas e incluso cientos

de capas convolucionales, mezcladas con *pooling*, seguidas de capas densas o incluso otras estructuras complejas formando un *ensemble*. La idea de futuro es estudiar y completar una estructura más compleja de convolucionales, que permita escalar el modelo a un número mucho mayor de etiquetas.

- Probar otro tipo de redes de *deep learning*, como RNN (*Recursive Neural Networks*)
- Mejorar la documentación a nivel *codebase* y repositorio. Me gustaría haber podido documentar mucho mejor los resultados obtenidos. Es una pena que descubrí herramientas como **Tensorboard** en las últimas etapas del proyecto.
- Creación de una *landing page* de presentación, mejoría de la demo subida para inferencia con *webcam*. Me gustaría poder mejorar el *front-end* para incluir inferencia a tiempo real con el modelo en el *back-end*. Es algo que puedo hacer muy pronto, con el estado actual del proyecto, pero que se ha quedado fuera de la iteración final.
- Utilización de *criteria*, *optimizers* y *schedulers* distintos. No he podido probar tantas opciones como me hubiera gustado. Pretendo investigar en la literatura cuales son las que ofrecen mejor resultado en los distintos escenarios, y ver como puedo aplicarlo en este proyecto.
- Buscar la forma de compartirlo en la comunidad *Open Source*. Me gustaría buscar la forma de expandir el proyecto, y así encontrar personas que estén interesadas en continuar con el proyecto o en colaborar de algún modo.
- Y mucho más. Me encantaría poder seguir mejorando el modelo e intentar terminar creando un producto viable con el que las personas se puedan beneficiar. Me parece que el objetivo es muy noble y con un poco de impulso por mi parte, o por cualquier desarrollador que siga con el proyecto; podemos conseguir algo que funcione genial.

7.3. Experiencia personal

Este proyecto ha sido una aventura y me ha abierto los ojos al mundo del *deep learning* y la investigación. Cuando comencé pensaba que iba a ser capaz de crear una red neuronal con un *accuracy* alto y que iba a poder crear una red *transformers* con la que sintetizar las salidas de la

CNN. También pensaba que todo esto lo iba a hacer en los cuatro primeros *sprints*. Tras esto, la idea era poder crear una aplicación web que usará el modelo en los dos últimos meses.

Nada más lejos de la realidad.

Este proyecto ha sido una prueba a la paciencia: prueba tras prueba, cambio tras cambio. El avance era lento, y había semanas en las que me quedaba atascado cambiando la estructura de la red, o probando una y otra vez con distintos hiperparámetros.

El hecho de tratar con datos que producían un tensor de entrada de la red de 5 dimensiones ha hecho que las iteraciones y mejoras sobre el modelo sean lentas, muy lentas. En ocasiones el modelo ha estado entrenando durante casi un día completo (¡y en GPU!), inhabilitando cualquier posibilidad de trabajar en él.

Aun así, este proyecto me ha hecho comprender mucho más a fondo el comportamiento de las redes neuronales en general, y sobre todo, de las redes neuronales convolucionales. Comencé sin saber lo que era una convolución y un *kernel*, con miedo a averiguar lo que significaba *ResNet*; y he terminado creando una *CNN* con un *accuracy* alto (aunque con relativas pocas etiquetas) y he perdido el miedo a investigar y leer artículos.

Por otro lado, he podido realizar una demo web para que cualquier usuario pueda probar el modelo con tan solo subir un vídeo. Esto me ha permitido aprender tecnologías web que todavía no había descubierto. También he creado una API en **Python**, con lo que he aprendido un *framework* más.

Y además me he tenido que enfrentar a los tres grandes de la computación en la nube hasta que he conseguido subir el pesado modelo a una de ellas. De esta experiencia salió una guía que se puede leer en el repositorio del servidor.

En general, la experiencia ha sido genial y el aprendizaje ha sido inmenso. Me ha hecho ver que el campo del *machine learning* es el futuro en todos los ámbitos tanto en investigación como cotidianos. El uso de tecnologías que usen *deep learning* está creciendo enormemente en la actualidad, y esta experiencia me impulsa a seguir investigando en este campo e intentar trabajar con y de esto en el futuro.

Tanto me ha gustado el proyecto, tanto he aprendido, y tanto me he dado cuenta que no se, que lo único que quiero hacer es seguir, y hacerlo mejor, con mejor información.

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] Nikhil Akki. Bench-marking restful apis, Aug 2021.
- [3] Corrado Alessio. Animals-10, 2019.
- [4] Ethem Alpaydin. *Machine learning*. MIT Press, 2021.
- [5] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1):105–139, 1999.
- [6] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.
- [7] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.
- [8] 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.
- [9] Steven Busuttil. Support vector machines. 2003.
- [10] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

- [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] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2):103–130, 1997.
- [15] Amanda Duarte, Shruti Palaskar, Lucas Ventura, Deepti Ghadiyaram, Kenneth DeHaan, Florian Metze, Jordi Torres, and Xavier Giro-i Nieto. How2Sign: A Large-scale Multimodal Dataset for Continuous American Sign Language. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.
- [16] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- [17] Biyi Fang, Jillian Co, and Mi Zhang. Deepasl: Enabling ubiquitous and non-intrusive word and sentence-level sign language translation. In *Proceedings of the 15th ACM conference on embedded network sensor systems*, pages 1–13, 2017.
- [18] L. Fausett and L.V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall international editions. Prentice-Hall, 1994.
- [19] 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.
- [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] Mizuki Maruyama, Shuvojit Ghose, Katsufumi Inoue, Partha Pratim Roy, Masakazu Iwamura, and Michifumi Yoshioka. Word-level sign language recognition with multi-stream neural networks focusing on local regions. *arXiv preprint arXiv:2106.15989*, 2021.
- [34] 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.
- [35] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.
- [36] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Number 9. McGraw-hill New York, 1997.
- [37] Kevin P Murphy et al. Naive bayes classifiers. *University of British Columbia*, 18(60):1–8, 2006.
- [38] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [39] J Quinlan. Building classification models: Id3 i c4. 5. *Dane udostepniona pod adrese: <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] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.
- [48] Karen Simonyan and Andrew Zisserman. *Published as a conference paper at ICLR 2015 VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*. 2015.
- [49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. *Going deeper with convolutions*. 2014.
- [50] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.
- [51] 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.
- [52] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *CVPR*, 2016.
- [53] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.
- [54] Xinyue Zhu, Yifan Liu, Zengchang Qin, and Jiahong Li. Data augmentation in emotion classification using generative adversarial networks, 2017.