eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

RECONOCIMIENTO DE GESTOS MANUALES MEDIANTE RED NEURONAL ARTIFICIAL

Estudiante *San Martín Garaluce, Jon*
Director/Directora *Espinosa Acereda, Jon Koldobika*
Departamento de Ingeniería de Comunicaciones
Curso académico *2019/2020*

Bilbao, 1 de Septiembre, 2020

Resumen

Las redes neuronales artificiales forman parte del conjunto de tecnologías que conocemos como Inteligencia Artificial y que, hoy en día, son usadas en infinidad de aplicaciones y además poseen un gran potencial de cara al futuro. Ellas se basan en modelos matemáticos que intentan asemejarse a las neuronas biológicas, proviniendo de ahí su nombre.

En este trabajo se ha desarrollado una red neuronal artificial capaz de detectar y reconocer gestos manuales. Para ello, se han estudiado los principios de estas redes, así como las mejores tecnologías existentes para su implementación. Acto seguido, usando estas tecnologías, se han codificado varias aplicaciones para crear, entrenar y usar la red neuronal artificial objetivo. El entrenamiento de la red neuronal, por su parte, ha sido un proceso iterativo, como es normal en el desarrollo de las redes neuronales artificiales. Se ha partido de una arquitectura de red inicial, y tras numerosas iteraciones y cambios de parámetros, se ha llegado a una arquitectura óptima y eficiente, como se ha demostrado con una serie de pruebas realizadas finalmente.

Laburpena

Sare neuronal artifizialak, Adimen Artifiziala bezala ezagutzen ditugun teknologien parte dira eta, gaur egun, teknologia hauek hainbat aplikazioetan erabiliak izaten dira, eta gainera potentzial handia daukate etorkizunari begira. Sare neuronal artifizialak neurona biologikoen antza daukaten eredu matematikoetan oinarritzen dira, hortik datorkie izena.

Lan honetan, esku keinuak detektatzeko eta sailkatzeko gai den sare neuronal artifizial bat sortu da. Horretarako, sare hauen printzipioak ikasi dira, baita beren implementazioa lortzeko dauden teknologia onenak ere. Ondoren, teknologia hauen bidez, sare neuronal artifiziala sortzeko, entrenatzeko eta erabiltzeko hainbat aplikazio kodetu dira. Sare neuronal entrenamendua iterazio-prozesu bat izan da, sare neuronal artifizialak sortzean normala den moduan. Hasierako arkitektura batetik abiatu da, eta hainbat iterazio eta parametro aldaketa ondoren, arkitektura optimo eta eraginkor batera heldu da, egindako proba multzo batekin frogatu den moduan.

Abstract

Artificial neural networks are part of the group of technologies that we know as Artificial Inteligence, which are used nowdays in lots of applications and have a great potential looking to the future. They are based on mathematical models that try to resemble biological neurons, hence its name.

In this project, an artificial neural network capable of detecting and recognizing hand gestures has been developed. To that end, the principles of this networks have been studied, as well as the best existing technologies for its implementation. Thereupon, using these technologies, various applications have been codified to create, train and use the targeted artificial neural network. For its part, the training of the neural network has been an iterative process, as is normal in the development of artificial neural networks. An initial architecture has been the starting point, and from there, after a series of iterations and parameter changes, an optimal and efficient architecture has been reached, as shown in a series of tests.

Índice de contenidos

Tabla de ilustraciones

Índice de tablas

1. Introducción

Vivimos en una sociedad en la que la tecnología ocupa nuestro día a día. Smartphones, ordenadores, tablets, smartwatches, GPSs… incluso nuestro coche o frigorífico ya no son meras máquinas para transportarnos o para conservar nuestra comida. Actualmente, casi todo tiene una parte "inteligente" que complementa su función principal, se conecta a internet, nos envía avisos al móvil, etc.

Y entre tanto avance tecnológico que vemos estos días, oímos de forma repetida palabras como "industria 4.0", "5G", "Smart grid", e "inteligencia artificial" o simplemente IA. Esta última puede que sea la más repetida, ya que no sólo se utiliza en la industria y ambientes de investigación, sino que también ha llegado al público en general.

Actualmente, todo el mundo posee una IA en su teléfono móvil que ayuda a buscar información en internet, programar recordatorios, encontrar un restaurante cercano, mandar mensajes y correos electrónicos, etc., todo con simples comandos de voz. Siri en iPhone, Google Assistant en Android, o Alexa en los dispositivos Echo de Amazon, son ejemplos de estas IAs que cada vez ocupan más bolsillos y hogares. También han llegado a la industria donde ayudan a procesar cantidades enormes de datos (big data).

Y con el término "Inteligencia Artificial" también han llegado a nosotros otros términos relacionados como "machine learning", "deep learning", "redes neuronales" y otros tantos que forman el conjunto de las disciplinas científicas del área de la Inteligencia Artificial.

Este último, "redes neuronales", será el que nos ocupará en este TFM, en el que se desarrollará una red neuronal artificial para la detección y reconocimiento de gestos manuales. También se tratará el llamado "machine learning", o como se conoce en castellano, "aprendizaje automático", dado que lo interesante de la inteligencia artificial actualmente es su capacidad de aprender por sí sola de los datos.

Una red neuronal artificial es un modelo matemático y computacional inspirado en cierto modo a su homólogo biológico, el cerebro, que está constituido por cientos de miles de células especiales llamadas neuronas. En una red neuronal artificial existen un conjunto de unidades conectadas entre sí, (las neuronas), las cuales envían información unas a otras. De forma más concreta, cada neurona estaría conectada a otra mediante un enlace en el que el valor de salida de la anterior neurona estaría multiplicado por un valor de peso del enlace.

1.1 Historia

A finales de los años 40 el psicólogo Donald Hebb trabajaba en una hipótesis sobre el aprendizaje basado en el mecanismo de plasticidad neuronal del cerebro, que ahora se conoce como aprendizaje de Hebb. Estas ideas empezaron a ser aplicadas por investigadores a los modelos computacionales en 1948.

Farley y Wesley A. Clark utilizaron máquinas computacionales, llamadas entonces simplemente “calculadoras”, para simular una red de Hebb en el MIT. Les siguieron otros trabajos y simulaciones de más investigadores con mejor o peor resultado.

En 1958 Frank Rosenblatt creó el perceptrón, que era una unidad básica de inferencia en forma de discriminador lineal a partir de la cual se podía desarrollar un algoritmo capaz de generar un criterio para seleccionar una subclase a partir de otras, usando adición y sustracción simples. El perceptrón sería, resumiendo, una neurona artificial simple, como la de la ilustración 1.

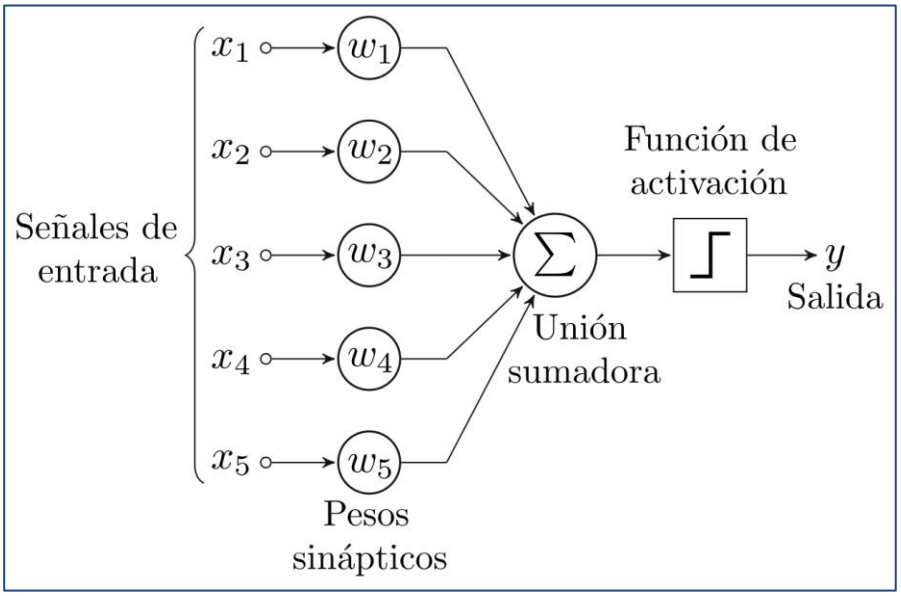

Ilustración 1. Neurona artificial. Autor: Alejandro Cartas

La investigación de redes neuronales prácticamente se estancó después de 1969, hasta que en 1986 David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Willians anuncian su paper "Learning representations by back-propagating errors", en el que trataban el tema de la propagación hacia atrás de errores y proponían soluciones computacionales para ello. Supuso un gran avance que trajo consigo más trabajos, pero nuevamente se produjo un estancamiento en las investigaciones y desarrollos sobre inteligencia artificial que no desapareció hasta los recientes avances en CPUs y GPUs, que permitirían ahora sí explotar todo el potencial de las redes neuronales.

2. Contexto

Las redes neuronales no son una idea nueva. Como se ha explicado, hacia los años 50 y 60 se empezaron a publicar los primeros conceptos. Sin embargo, nunca tuvieron un gran éxito, dado que se necesita una importante cantidad de recursos computacionales para entrenar y ejecutar una red neuronal con buenos resultados.

No obstante, en los últimos años se han conseguido grandes avances en cuanto a CPUs y al uso de GPUs para este tipo de computaciones. Reconocimiento facial, de voz, de objetos en imágenes, en vídeos, mejora de calidad en imágenes, mejora de framerate, detección de enfermedades, síntesis de voz, de imágenes, de vídeo... Las redes neuronales están empezando a resolver problemas que se le escapaban a los ordenadores.

Las redes neuronales parece que incluso pueden acabar dominando uno de los juegos que se les resiste a los ordenadores: el Go. Se trata de un juego de estrategia para dos personas que data de hace más de 4000 años. La peculiaridad de este juego es que el tablero puede ser de 19 x 19 casillas, y tiene 361 fichas. Por tanto, las posibles jugadas son de un orden tal que se escapa para cualquier paradigma de resolución por fuerza bruta. Por tanto, es necesario que la máquina que pretendemos que juegue a Go “aprenda” realmente a hacerlo. Es aquí donde entran las redes neuronales, siendo AlphaGo la más prometedora, habiendo conseguido ganar a uno de los mejores jugadores del mundo de la disciplina.

Ilustración 2. Alpha Go en una competición. Fuente: ABC Blogs

De todas formas, estas redes tampoco son la panacea. Es posible, como unos investigadores lograron, usar una red neuronal para generar imágenes que engañan a otra red neuronal diseñada para reconocer objetos. De esta forma, lo que a nosotros nos parece una imagen aleatoria, para la red neuronal es un coche o una bicicleta. O por ejemplo, hacer creer a una red neuronal que eres otra persona, confundiendo a sistemas de seguridad. Y todo esto sin demasiadas complicaciones.

Aún así, se trata de un campo muy interesante y que promete bastantes avances a corto y largo plazo, sobre todo en el área del reconocimiento de imagen y de sonido. Es por todo esto que este TFM se centra en el desarrollo de una red neuronal, concretamente para el reconocimiento gestual, dado que supone avanzar en esta área de gran interés y potencial y permite mantenerse en el mundo cambiante de la tecnología.

Ilustración 3. Interés relativo a lo largo del tiempo en redes neuronales artificiales. Fuente: Google Trends.

3. Objetivos y/o alcance

Para deducir el alcance de este proyecto podemos fijarnos en el título del mismo, que no es otro que el de “Reconocimiento de gestos manuales mediante red neuronal artificial”. Por tanto, el objetivo o alcance del mismo será conseguir una red neuronal artificial que no sólo detecte, sino que reconozca gestos de una persona realizados con las manos. Se hace especial hincapié en la diferencia entre “detección” y “reconocimiento”, puesto que la detección sólo implica el conocimiento de si el usuario está o no realizando un gesto manual. Sin embargo, el reconocimiento implica primero detección, y después clasificación de lo detectado.

Por ello, en este proyecto se tratará de desarrollar y conseguir una red neuronal artificial que, una vez entrenada, al mostrarle una persona realizando distintos gestos con sus manos, los detecte y clasifique, completando el reconocimiento.

Para el entrenamiento mencionado hará falta determinar qué gestos queremos que la red neuronal identifique. Por tanto, un objetivo secundario será el de analizar y encontrar los gestos más adecuados para el proyecto, teniendo en cuenta la naturaleza de los mismos, su utilidad para futuros casos de uso, la facilidad de entrenamiento con los mismos, etc. Dentro de este objetivo nos encontramos también con la necesidad de hallar un dataset que contenga estos gestos manuales elegidos.

4. Beneficios

4.1 Beneficios económicos

El interés por la Inteligencia Artificial en general, y las redes neuronales en particular, aumenta cada día a pasos agigantados. En todos aquellos sectores en los que se maneja un amplio volumen de información está presente la IA, gracias a su capacidad de recogerla y procesarla. Estos poseen un gran valor para las empresas ya que permite a las compañías ser más competitivas y crear mejores campañas de marketing, por poner un ejemplo. Desde el punto de vista de la industria estas tecnologías pueden ofrecer mayor asistencia y complementar más eficientemente los procesos de producción, tomando datos sobre estos procesos para encontrar formas más eficientes de trabajar.

Por tanto, el desarrollo de una herramienta en el sector de la Inteligencia Artificial trae indiscutiblemente beneficios económicos al contribuir de forma positiva en el mismo. Y por otro lado, como producto en sí mismo, al tener bastantes posibles usos y usuarios finales.

4.2 Beneficios sociales

En cuanto a beneficios sociales se refiere, debemos destacar los posibles usos que tendría este proyecto. Por ejemplo, es posible desarrollar interfaces basadas en gestos manuales para sistemas de ocio multimedia. Si suponemos una “Smart TV” que reconociera gestos manuales, podríamos navegar por menús y elegir nuestras películas favoritas, o nuestro contenido preferido, desde el sofá con un simple gesto de la mano. Lo mismo se aplicaría a cualquier ordenador o dispositivo.

También, el reconocimiento de gestos manuales (y de gestos en general) puede ser útil para personas con ciertas discapacidades. Si hablamos de personas con discapacidades intelectuales o personas mayores a las cuales les cuesta entender el manejo de un ordenador o cualquier otro aparato usando las formas habituales, es posible desarrollar una interfaz basada en gestos simples e intuitivos, de la cual esta red neuronal desarrollada formaría la pieza fundamental. Así, estas personas podrían usar dichos aparatos con facilidad. Algo parecido podría plantearse con personas con discapacidades motrices, dado que el uso de gestos manuales (si la discapacidad no es severa) supone una ventaja respecto al uso del teclado y el ratón o incluso pantallas táctiles, debido a que se puede hacer a distancia y no requiere tanta precisión.

4.3 Beneficios técnicos

Como se ha comentado anteriormente, al encontrarse el sector de la inteligencia artificial en auge, el desarrollo de este TFM no sólo aportaría beneficios económicos y sociales, sino también técnicos. Esto es debido a que realizar una aportación útil al sector supone avances concretos o poner sobre la mesa nuevas perspectivas, ideas y problemáticas.

Desarrollar una red neuronal capaz de reconocer gestos manuales significa aportar otro granito de arena al área del reconocimiento de imágenes en particular, como también al de la Inteligencia Artificial en general.

5. Estado del arte

En este apartado estudiaremos el estado actual de la tecnología y la investigación relacionadas con nuestro objetivo, el cual consiste en desarrollar una red neuronal artificial capaz de reconocer gestos manuales. También comentaremos diferentes algoritmos o conceptos relacionados que nos serán útiles en nuestro proyecto.

A la hora de comenzar esta búsqueda de información en torno a las redes neuronales artificiales, nos topamos primero con una serie de conceptos básicos sobre los que se asientan dichas redes, y que conviene explicar, concepto a concepto, hasta llegar a las redes neuronales en sí mismas, y las diferentes tecnologías asociadas.

5.1 Modelos matemáticos y regresión lineal

El primero de esos conceptos sería el de “modelo matemático”. Un modelo matemático sería un tipo de modelo científico que, empleando ciertas fórmulas o formulismos matemáticos, permite expresar relaciones, variables, parámetros y en general situaciones del mundo real. Uno de los modelos matemáticos más sencillos sería el de la regresión lineal, mediante el cual podemos aproximar la relación de dependencia entre una variable dependiente y otras independientes. [1]

Para explicarlo mejor, usaremos la gráfica de la ilustración 4.

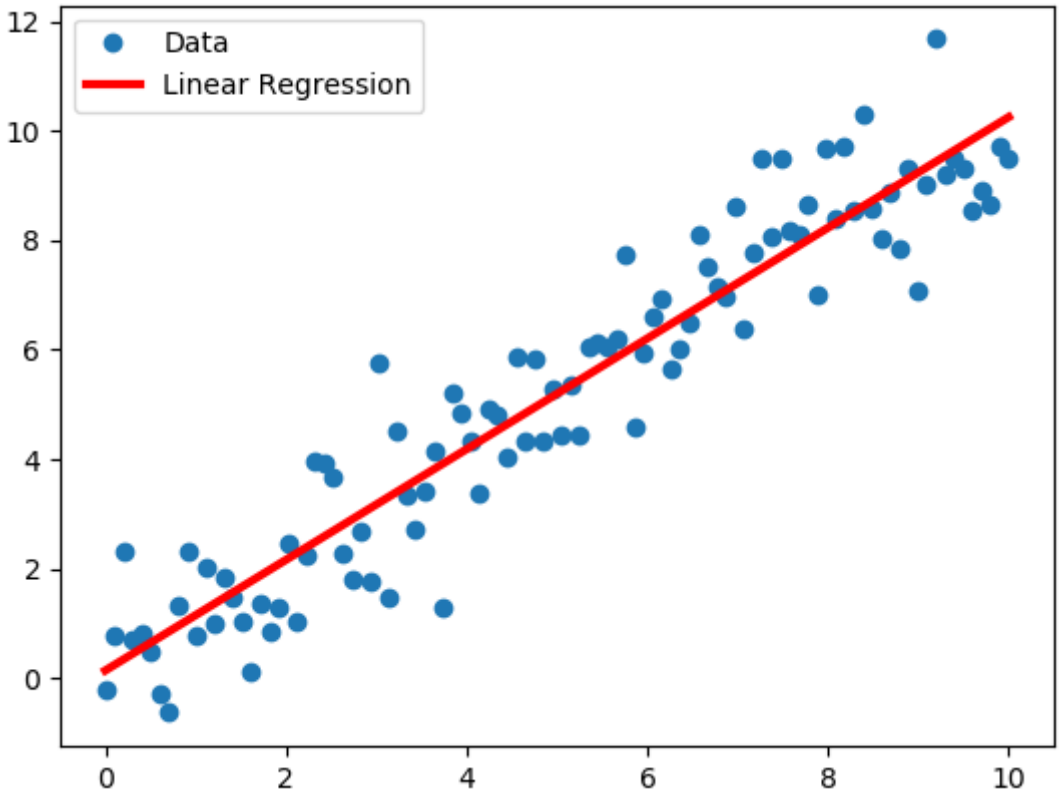

Ilustración 4. Regresión lineal. Autor: Hieu Tran. Fuente: ResearchGate.

El eje horizontal sería la variable independiente X y el eje vertical la variable dependiente Y. Lo que observamos en la imagen es una nube de puntos, nuestros datos de la realidad, que parecen conformar una recta imaginaria a lo largo de la gráfica. Gracias a esto podemos trazar una recta con una ordenada en el origen n y una pendiente m, conformando la famosa ecuación de la recta Y = mX + n. Usando esta ecuación que hemos obtenido de la nube de puntos tendremos un modelo aproximado de la realidad, mediante el cual podremos predecir y aproximar valores Y en base a nuevos datos X, de los cuales ya no conocemos en un principio su Y. Estos datos X podrían ser, por ejemplo, el número medio de habitaciones de una vivienda en un barrio, y el valor Y, su precio medio. De esta forma, obteniendo pares de valores X e Y de diferentes barrios podremos crear un modelo y desde ahí predecir valores de vivienda de otros barrios en base a su número medio de habitaciones.

Aquí hemos trazado la recta nosotros mismos, pero lo interesante sería encontrar una forma en la que de forma automática encontrásemos la recta que mejor se ajusta a los datos, es decir, el mejor modelo. Hay que señalar que podemos tener no sólo una variable independiente X, sino que podríamos tener, por ejemplo, dos variables independientes X1 y X2, siendo en este caso un plano y no una recta lo que tendríamos que buscar, y la nube de puntos se encontraría en 3 dimensiones. En la mayoría de los casos tendremos n variables independientes, siendo n un número posiblemente muy grande (n=1000, n=100.000, …), obteniendo entonces un hiperplano en un espacio multidimensional. [2]

Pero la forma en la que se obtienen los modelos de regresión lineal simple (una variable independiente) es la misma que para los modelos de regresión lineal múltiple (múltiples variables independientes), por lo que la explicación se simplifica. De todas formas, el objetivo de este apartado no es profundizar extremadamente en estos conceptos, sino dar unas pinceladas para entender la lógica subyacente de los mismos, y en esta lógica subyacente entran los conceptos de error, función de coste, y minimización del error.

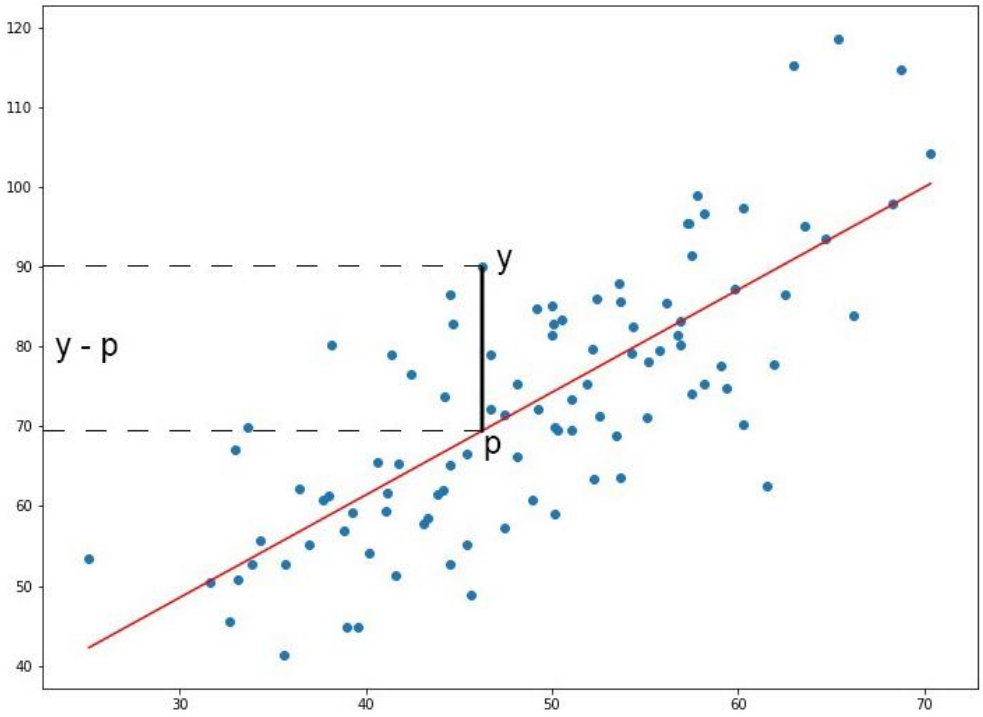

Ilustración 5. Regresión lineal y error.

En la gráfica de la ilustración 5, podemos observar: la nube de puntos, la recta de nuestro modelo, el valor real de un punto en concreto (y) y el valor predicho (p), siendo la diferencia de estos valores nuestro error. De todos modos, no estamos trabajando con un valor, sino con varios valores, por lo que en la práctica lo que se hace es calcular los errores individuales de nuestros datos, elevarlos al cuadrado y calcular la media, lo que se denomina “Error Cuadrático Medio” (ilustración 6).

$$L(x) = \sum_{i=1}^{n} (y_i - p_i)^2$$

Ilustración 6. Error cuadrático medio.

Esta sería nuestra “función de coste”, y es la función que buscaremos minimizar para así obtener la recta que mejor se ajuste a nuestro modelo. Y para minimizar la función, como en cualquier otro caso, el procedimiento consiste en calcular las derivadas parciales e igualar a cero. Estos cálculos se suelen expresar en forma matricial dado que es más simple, y además, de esta forma se pueden aprovechar las ventajas de los procesadores gráficos (GPUs), que están diseñados para este tipo de cálculos como explicaremos más adelante.

Resumiendo, la regresión lineal es un tipo de modelo matemático que, como tal, intenta modelar de forma aproximada una situación del mundo real, obteniendo la recta que minimiza el error entre sus propios valores y los valores reales. Esta es la lógica subyacente que es esencial entender, dado que se repite de forma constante en el área del “machine learning” o

“aprendizaje automático”; hallar una función de coste que representa el error, y minimizar esta función para minimizar el error, hallando así el mejor modelo.

5.2 Descenso del gradiente

El problema del cálculo de la regresión lineal mediante el método explicado es que en cuanto tenemos más variables, la complejidad computacional se eleva. Esto es debido a que entre los cálculos matriciales que se deben realizar está el de inversión de matrices, el cual es bastante caro computacionalmente hablando. Por esta razón, existen otros métodos iterativos que ahorran en tiempo y recursos computacionales como es el “Descenso del gradiente”.

En este algoritmo lo que pretendemos calcular es el gradiente de la función de coste, es decir, la dirección y sentido en el que dicha función aumenta. Al obtener este gradiente tomaremos el sentido contrario (sentido en el que la función se minimiza), y avanzaremos en ese sentido. El ratio en el que avanzaremos se llama “learning rate” o “ratio de aprendizaje”. Una vez en el nuevo punto, volveremos a realizar el mismo cálculo del gradiente y se avanzará en su sentido negativo usando el mismo ratio de aprendizaje. Así, de forma iterativa, llegaremos al punto mínimo de la función de coste; es decir, habremos minimizado el error. [3, p. 1]

Si usamos un learning rate muy grande convergeremos antes hacia la solución, pero corremos el peligro de saltarnos un mínimo al realizar saltos muy grandes, pudiendo no converger nunca. En cambio, con un learning rate muy pequeño, nos aseguraremos de converger virtualmente siempre, pero a un coste de tiempo mucho mayor [3, pp. 2,3]. Por tanto, un reto en el área de la inteligencia artificial que todavía trae quebraderos de cabeza a los ingenieros es encontrar un learning rate adecuado para sus proyectos concretos.

5.3 Redes Neuronales

Estamos ya en disposición de tratar las redes neuronales en sí mismas y su unidad fundamental, la neurona.

5.3.1 La Neurona

Tal y como hemos hablado antes de forma breve, una neurona sería una unidad computacional simple que poseería varias entradas y una salida. Estas entradas reciben la información del resto de neuronas y se les multiplica por el peso asociado al enlace entre pares de neuronas. Dentro de la neurona se realizaría la suma ponderada de estas entradas, además de aplicarle opcionalmente un sesgo. [4, p. 46]

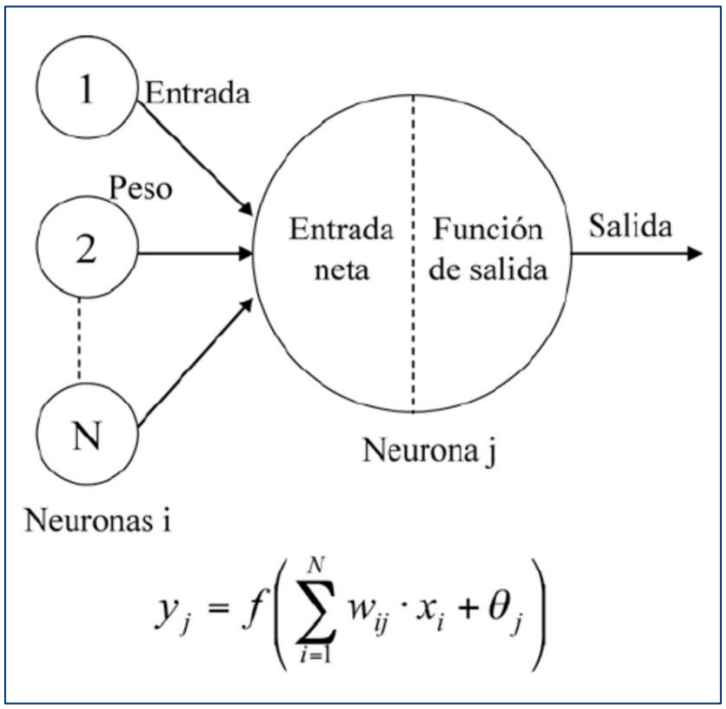

Ilustración 7. Neurona artificial y su expresión matemática. Autor: Albert Sesé. Fuente: ResearchGate.

En la expresión de la ilustración 7, si suponemos N=1 lo que obtenemos es una ecuación de una recta para cada neurona j, al tener sólo una entrada. Con N>1 obtenemos una ecuación de un hiperplano, al tener más de una entrada. Ignorando la función de activación f, de momento se podría decir que tenemos un problema clásico de regresión lineal en cada neurona. Por tanto, podemos aplicar los métodos resolutivos habituales de los casos de regresión lineal para obtener los parámetros óptimos de la neurona (pesos y sesgo).

5.3.2 La Red Neuronal Artificial

De todas formas, lo interesante de las neuronas artificiales es usarlas en conjunto, como una red. El motivo es que para la gran mayoría de los problemas no es posible crear un modelo mediante simplemente regresión lineal.

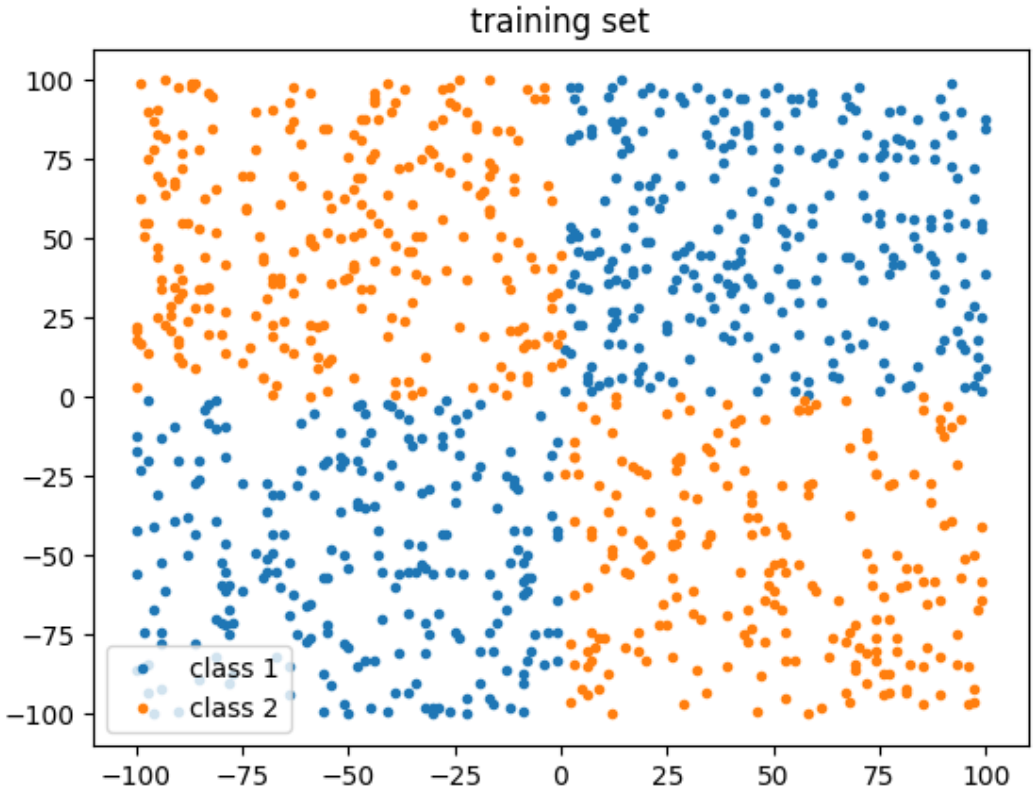

Ilustración 8. Training set.

La regresión lineal también se usa para encontrar la recta que separa dos o más clases de datos. Sin embargo, en el ejemplo de la ilustración 8, es imposible trazar una sola recta que separe ambas clases de datos. Son necesarias al menos dos rectas, de ahí la limitación de la regresión lineal y el potencial de las redes neuronales. Porque usando, por ejemplo, dos neuronas, podríamos encontrar las dos rectas óptimas que separarían las dos clases de datos (ilustración 9).

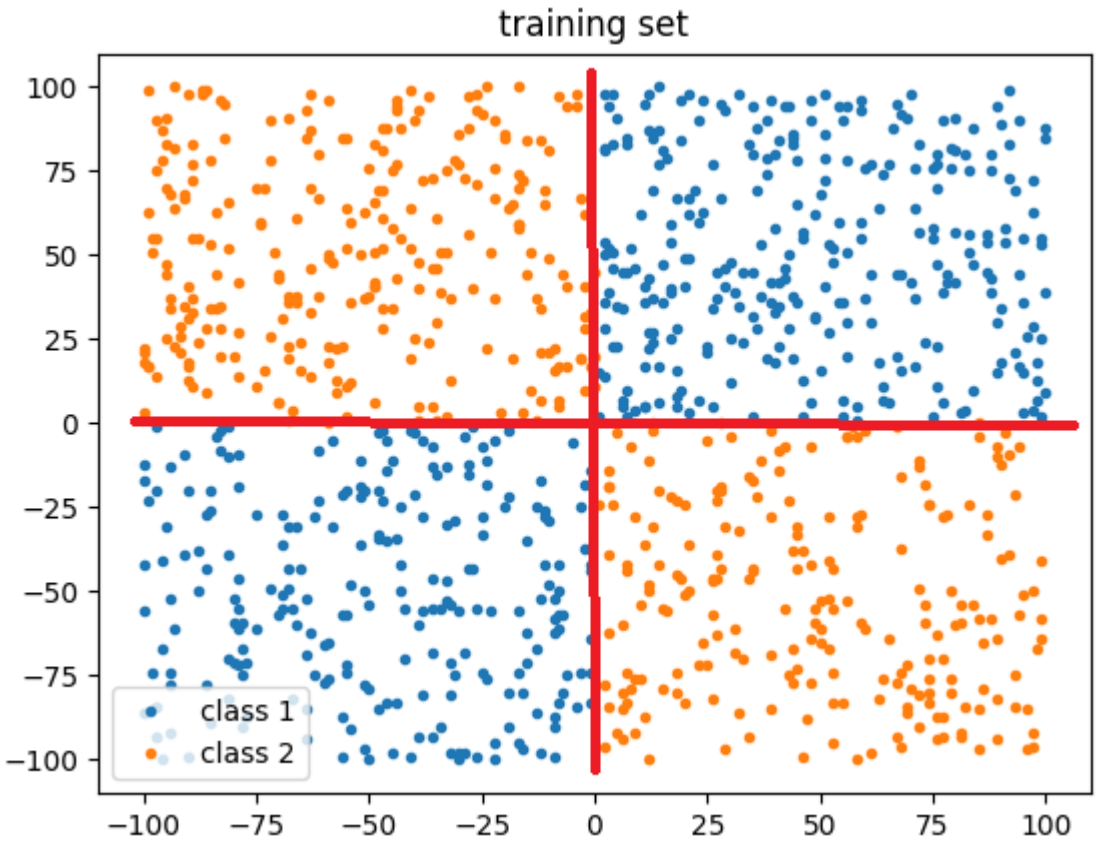

Ilustración 9. Forma óptima de separar los datos.

Por lo tanto, queda claro que para modelar problemas más complejos se hace necesario el uso de múltiples neuronas formando redes neuronales (ilustración 10). Estas neuronas pueden estar colocadas dentro de la red de diferentes formas.

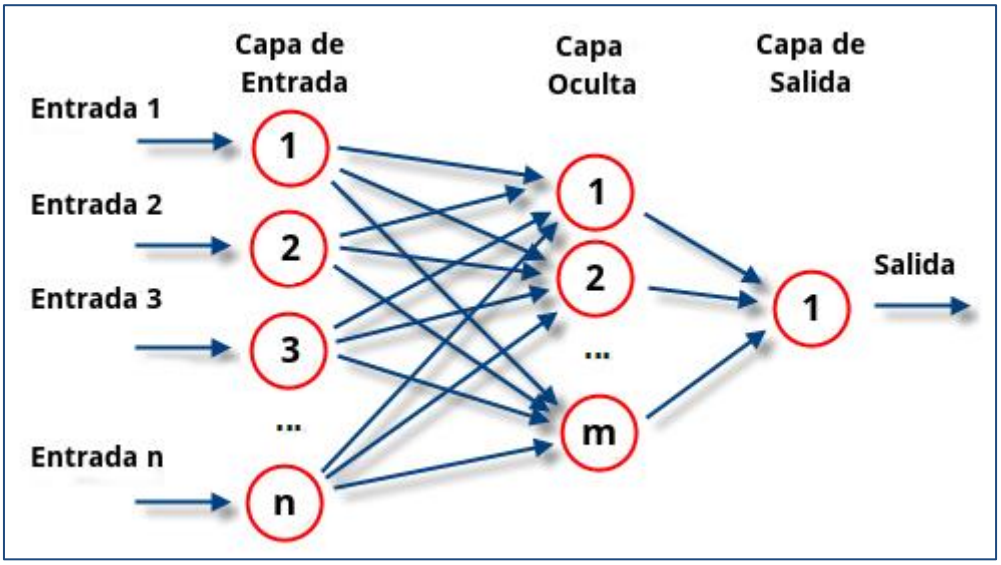

Ilustración 10. Capas de una red neuronal artificial.

Se organizan en capas, recibiendo en principio la misma información cada neurona de la misma capa y enviándola, procesada, a la capa siguiente. La primera capa sería la capa de entrada, la última, capa de salida, y las capas intermedias, capas ocultas. [4, p. 48]

Normalmente, todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente, aunque pueden darse casos en los que no. En otras ocasiones también existen “saltos” o “skip connections” en las que se conectan varias neuronas de capas no consecutivas. Hay que aclarar que no existe una arquitectura “óptima” o concreta para cada modelo o problema, siendo la “prueba y error” el método que un programador usará al programar una red neuronal [4, p. 49]. No obstante, para algunos problemas muy concretos y ya estudiados en profundidad, sí existen ciertas directrices para construir arquitecturas o incluso ciertas arquitecturas concretas. Pero en general será la práctica e intuición del programador la que le acercará a la arquitectura correcta.

5.3.3 Función de activación

Otro detalle que hay que señalar es que si como hemos dicho en cada neurona se estaría modelando un problema de regresión lineal, al conectar unas neuronas con otras estaríamos juntando funciones lineales con funciones lineales. Esto provoca que todas colapsen en una sola función lineal, perdiéndose las capacidades inherentes de juntar varias neuronas en vez de una sola. Es decir, sumar rectas da como resultado otra recta. Por esta razón se deben introducir no-linealidades en las salidas de las neuronas evitando el colapso en una sola recta. Esto se realiza mediante las “funciones de activación”.

Si lo que hacíamos en cada neurona era calcular como valor de salida una suma ponderada de nuestros valores de entrada, lo que haremos ahora será pasar dicho valor de salida por nuestra función de activación, que distorsionará nuestro valor de salida añadiéndole deformaciones no-lineales. Así se podrá concatenar de forma efectiva la computación de varias neuronas. [4, pp. 51, 52]

Una de las funciones de activación más sencillas es la función Relu o rectificadora.

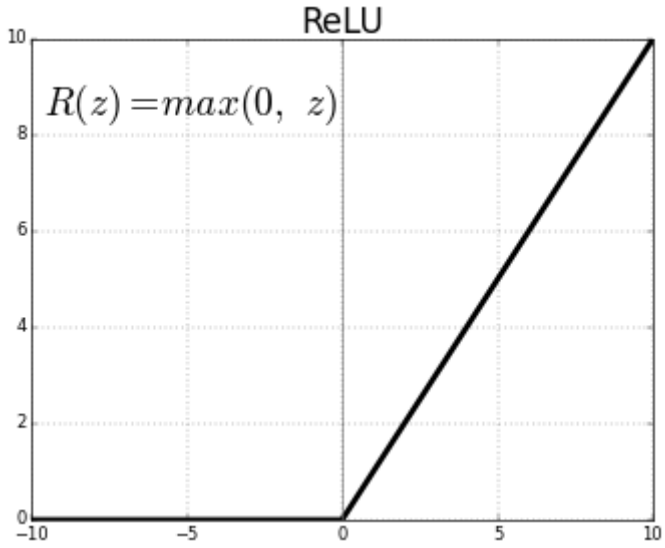

Ilustración 11. Función de activación ReLu.

Como se aprecia en la ilustración 11, lo que hace Relu es convertir a 0 los valores negativos de la entrada y dejar intactos los valores positivos. Algo tan sencillo ya implica una no-linealidad que habilita el uso de varias neuronas consecutivas. [5]

Otra función de activación es la función “Softmax”. Se emplea para "comprimir" o “convertir” un vector *K*-dimensional, z, de valores reales arbitrarios en un vector *K*-dimensional, σ(z), de valores reales en el rango [0, 1]. Es decir, nos permite acotar valores arbitrarios a un rango entre 0 y 1. [6, p. 115]

$$\sigma : \mathbb{R}^K \to [0,1]^K$$
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{para } j = 1, \ldots, K.$$

Ilustración 12. Función de activación SoftMax.

Esto es útil en teoría de la probabilidad, donde la salida de la función Softmax puede ser usada para representar una distribución categórica; la distribución de probabilidad sobre K diferentes posibles salidas. Por esta razón, utilizaremos la función de activación softmax en nuestra capa de salida de la red neuronal, para así obtener un rango de probabilidades de los posibles valores de salida. De esta forma, al usar la red neuronal para detectar gestos, ésta nos dirá el gesto con más probabilidad de estar siendo detectado.

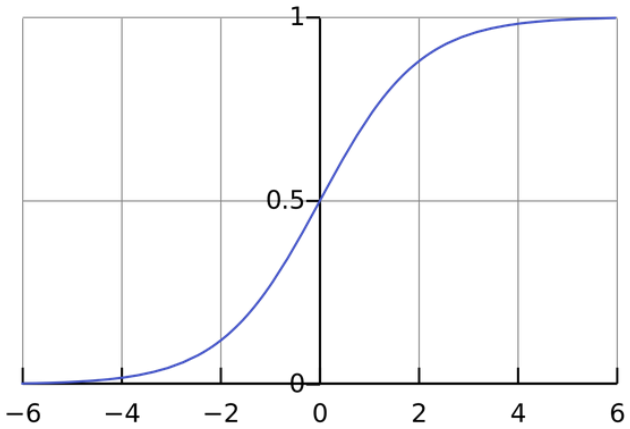

Ilustración 13. Gráfica de la función Softmax.

Con todo esto ya tendríamos las bases para construir una red neuronal, pero falta lo más importante; cómo se realiza el aprendizaje automático.

5.3.4 Backpropagation

El término “backpropagation” y su uso en redes neuronales fue anunciado por David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams en su paper "Learning representations by

back-propagating errors" de 1986. [7] En castellano significa propagación hacia atrás o retro propagación, y sería un método de cálculo del gradiente utilizado en algoritmos de aprendizaje para entrenar redes neuronales.

En apartados anteriores hablábamos del algoritmo del descenso del gradiente para ajustar los parámetros del modelo de la regresión lineal, y es lógico pensar que esto mismo bastaría para ajustar los parámetros de las redes neuronales, dado que se podría decir que cada neurona es un problema de regresión lineal en miniatura. El problema es que, al tener varias neuronas conectadas a otras, este cálculo se complica. En la regresión lineal, para calcular el gradiente, de forma intuitiva nos preguntamos "¿cuánto varía el coste (el error) ante un cambio del peso del enlace o del sesgo?". Matemáticamente esto se definía con las derivadas parciales de la función de coste con cada uno de los parámetros (pesos y sesgos). En las redes neuronales esto es lo mismo, pero variar un parámetro de un enlace o neurona influye en las siguientes neuronas y a su vez en las siguientes, etc.

Por ello, se usa backpropagation para calcular los diferentes gradientes de la función de coste global, y así poder usar el algoritmo del descenso del gradiente para ajustar los parámetros de la red. La forma en la que lo hace es la siguiente: esencialmente, backpropagation analiza toda la cadena de responsabilidades desde el error producido en la salida hacia atrás, mirando en cada capa qué neurona es la más responsable de producir dicho error y responsabilizándola por ello. En la siguiente capa hacia atrás, sólo tendremos que mirar cuán responsables son las neuronas directamente conectadas a la última que acabamos de analizar, y así sucesivamente. Esto además es eficiente, dado que ahorra analizar múltiples caminos que poco o nada afectan al error actual. Frente a los métodos anteriores que consistían en usar fuerza bruta para ver cuánto afectaba al error el variar levemente cada parámetro existente en una red neuronal, backpropagation supone un tremendo avance que ahorra computacionalmente y, por lo tanto, temporalmente. [7]

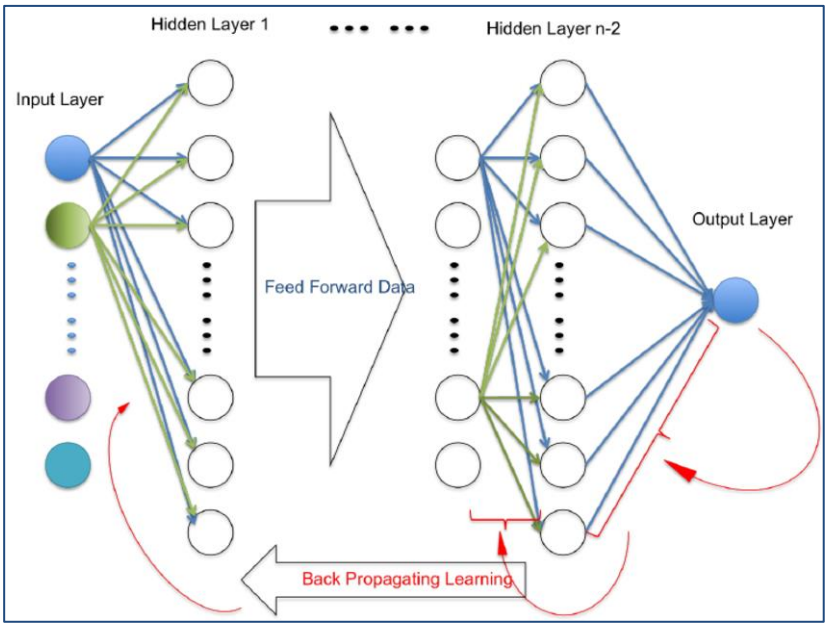

Ilustración 14. Backpropagation. Autor: Eun Young Kim. Fuente: ResearchGate.

Resumiendo, este algoritmo consiste en ir propagando el error hacia atrás, viendo qué neuronas son las responsables de dicho error y cuán erróneos son sus parámetros. Podemos ver una representación gráfica del algoritmo en la ilustración 14.

5.3.5 Entrenamiento

Antes de explicar en detalle las diferentes formas que puede adquirir una red neuronal, nos detendremos para tratar brevemente el proceso de entrenamiento de nuestra red. Básicamente, lo interesante de la inteligencia artificial actualmente es la capacidad de aprendizaje automático a partir de los datos. En vez de tener que ajustar manualmente los parámetros de nuestra red, usaremos datos para que, mediante backpropagation y el algoritmo del descenso del gradiente, sea esta misma la que autoajuste sus parámetros de forma iterativa.

Si por ejemplo nuestro problema es de clasificación, lo que haremos será preparar un set de datos (dataset) con sus respectivas etiquetas descriptivas asociadas. Después iremos pasando de forma iterativa, y en repetidas ocasiones, este dataset a la red, obteniendo a la salida un resultado que, en un primer momento, será aleatorio y mostrará un error. Este error, como hemos explicado en el apartado anterior, se propagará hacia atrás, y se irán modificando los parámetros hasta que la red se vaya ajustando más y más al modelo requerido, minimizando el error a la salida. [8]

Existen distintos términos importantes con respecto al entrenamiento, tales como:

Epoch: En castellano “época” o “ciclo”, consiste en el proceso de pasar enteramente el dataset. Normalmente, para entrenar la red usaremos varias epochs, es decir, pasaremos varias veces todo el dataset a nuestra red. Cuantas más epochs tengamos, mejor se entrenará la red, aunque costará más tiempo y también puede aparecer “overfitting”. [9]

Batch: Subdivisión del dataset. Si el tamaño del batch es 1, cada dato del dataset introducirá una modificación en los parámetros de la red. Si el tamaño es n, pasarán por la red n datos antes de introducir modificaciones. Esto no significa que se ignoren los n-1 datos hasta el dato n, sino que esos n datos contribuyen al error que después se propagará hacia atrás. Un tamaño grande de batch entrenará mejor la red y más rápido, pero ocupará más en memoria. [9]

Iteration: Cada paso de un batch por la red neuronal.

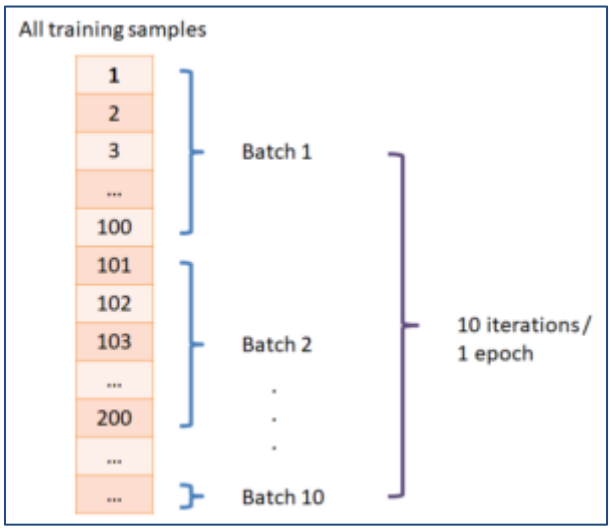

Ilustración 15. Diferencias entre batch y epoch.

Learning rate: Es el ratio de aprendizaje. Un learning rate grande hará más rápido el entrenamiento, pero puede hacer que nos saltemos el mínimo de la función de coste y nunca converjamos. Un learning rate pequeño nos asegurará encontrar el mínimo, pero costará más tiempo. [10]

Training Set: Set de entrenamiento. Fracción del dataset destinado al entrenamiento. Normalmente, entre un 80%-90% del dataset. [11]

Validation Set: Set de validación. Pequeña fracción del dataset original que no se usa para el entrenamiento y se usa para la validación. Normalmente, entre un 10%-20% del dataset. Es importante que, tanto el training set como el validation set, formen parte del mismo dataset y no sean de distintos datasets. [11]

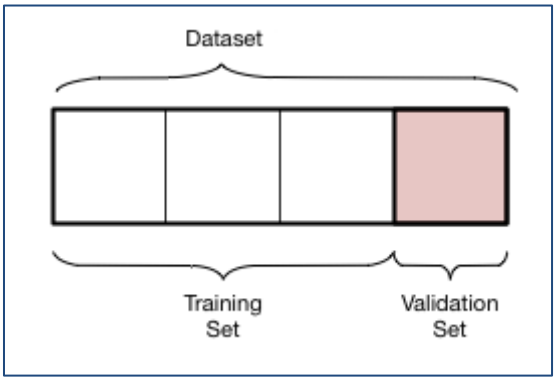

Ilustración 16. Dataset, Training Set y Validation Set.

Validación: Proceso mediante el cual se comprueba el correcto entrenamiento de la red. Normalmente, tras cada epoch se pasa el set de validación y se comprueba el error. Permite comprobar si nuestra red sufre “overffiting”. [8]

Overfitting: Sobreentrenamiento. Sucede cuando nuestra red se ajusta demasiado al training set y, en vez de aprender, “memoriza”. Se puede detectar en el proceso de validación, cuando

en un punto nuestro error de entrenamiento sigue bajando, pero el error de validación empieza a subir, porque nuestra red no generaliza bien y sólo reconoce los datos de entrenamiento. Hay diversas soluciones como aumentar el tamaño del dataset, añadir regularización, modificar los hiperparámetros (learning rate, batch, epochs, etc.), añadir capas dropout, etc. [8]

Underfitting: Cuando no hemos entrenado suficiente a la red y no aprende a generalizar. [8]

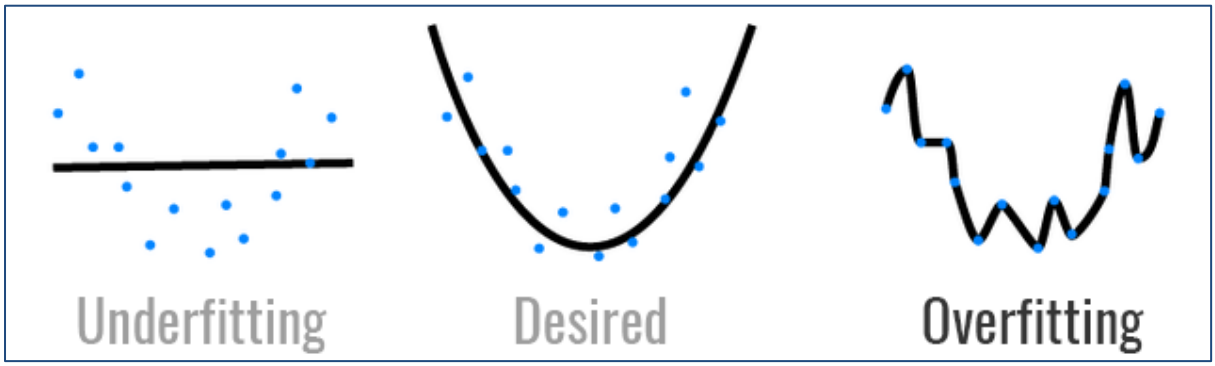

Ilustración 17. Underfitting, entrenamiento correcto, y overfitting. Fuente: Hackernoon.

Una vez entendido qué es una red neuronal y cómo se entrena, podemos pasar a explicar los diferentes tipos de redes neuronales interesantes para solucionar nuestro problema de interés, que es el de reconocer gestos manuales mediante una red neuronal artificial.

5.3.6 Redes neuronales convolucionales

Hasta ahora cuando hablábamos de datos nos referíamos simplemente a datos numéricos que individualmente poseen un significado. Es decir, podemos introducirlos en el entrenamiento individualmente uno a uno, dato a dato. Sin embargo, si lo que queremos es llegar a un modelo basado en una red neuronal que detecte gestos manuales, ¿cuáles son nuestros datos? ¿Cómo los introducimos? Si son imágenes, ¿podemos introducirlas de golpe? ¿Píxel a píxel? Estas y otras preguntas nos indican que no es una cuestión trivial.

Analizando las tecnologías existentes, encontramos una técnica de interés: las redes neuronales convolucionales. Estas tomarían como datos imágenes digitales, por ejemplo, en esquema RGB.

Un fotograma digital en color en el esquema RGB está formado por una matriz tridimensional (m*n*p), donde m y n se corresponden con la anchura y la altura de la imagen, mientras que p representa el plano de color. Este puede ser rojo, verde o azul, que son los colores primarios con los que al combinarlos conseguimos el resto de colores (ilustración 18). Cada plano de color contiene una matriz de valores que se corresponden con el grado de intensidad de color sobre un determinado píxel, siendo el primer valor de la matriz el correspondiente con el valor del primer píxel (ilustración 19). Este es la unidad mínima de visualización de una imagen digital y se ordenan de izquierda a derecha y de arriba a abajo. [12]

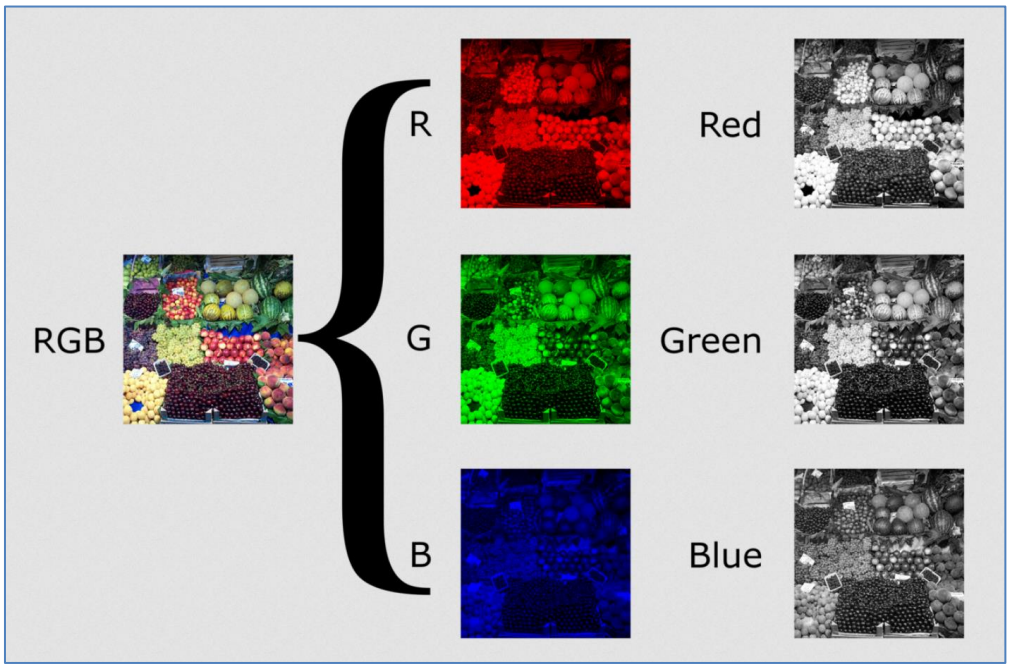

Ilustración 18. Descomposición imagen RGB. Autor: Nevit Dilmen

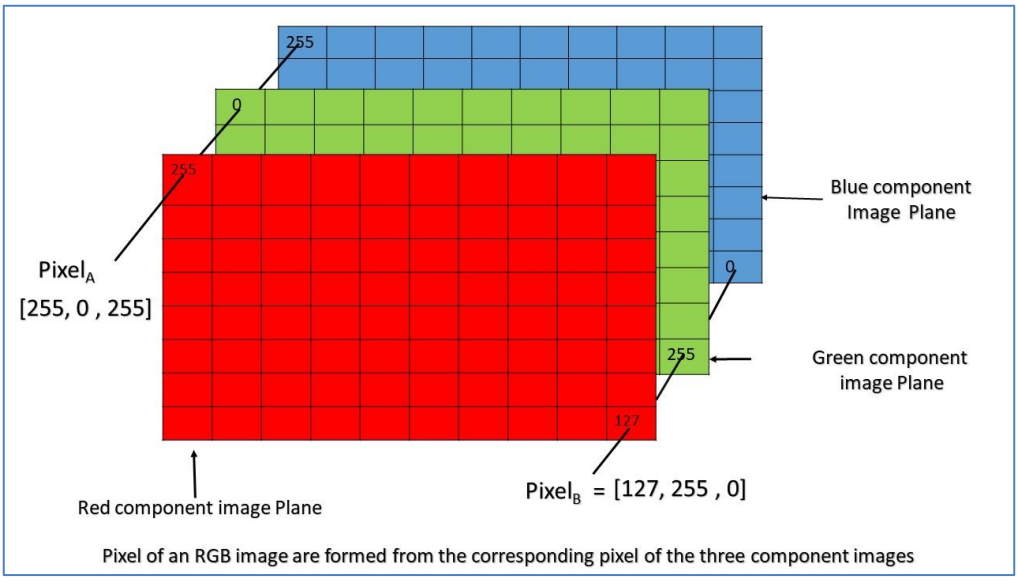

Ilustración 19. Planos matriciales RGB.

Lo primero que podríamos pensar sería en introducir a la red directamente los píxeles uno a uno, como valores numéricos según su intensidad. Pero esto supone un problema ya que tenemos 3 canales de colores. Por lo tanto, una cosa que se suele hacer es convertir la imagen a escala de grises y así tener sólo un plano. De todas formas, aunque hagamos esto, seguimos sin poder introducir en nuestra red los píxeles uno a uno, puesto que contienen información espacial que al introducir consecutivamente perderemos. Es decir, si introducimos los píxeles uno a uno por filas, perderemos la información y patrones que poseen entre sí los píxeles de una fila con los de la siguiente. Esto se puede hacer para

imágenes de muy poca resolución (decenas de píxeles) y redes neuronales simples pero no para reconocer patrones más complejos como gestos manuales.

Es aquí donde entran las redes neuronales convolucionales. Estas están diseñadas especialmente para tratar datos estructurados en 2D, como imágenes o señales de voz presentadas en espectrogramas.[13] La mayor ventaja es que poseen muchos menos parámetros a entrenar que una red neuronal con el mismo número de capas ocultas, por lo que su entrenamiento es más rápido. La función de las redes convolucionales es tratar de buscar características locales en pequeños grupos de entradas (en el caso de las imágenes, de píxeles), como pueden ser bordes, colores, líneas horizontales o verticales, etc. Si introducimos más capas, la red neuronal podrá descubrir más y más características complejas de la imagen. Al ser capas ocultas de la propia red no sabremos, en principio, qué filtros en concreto aparecerán.

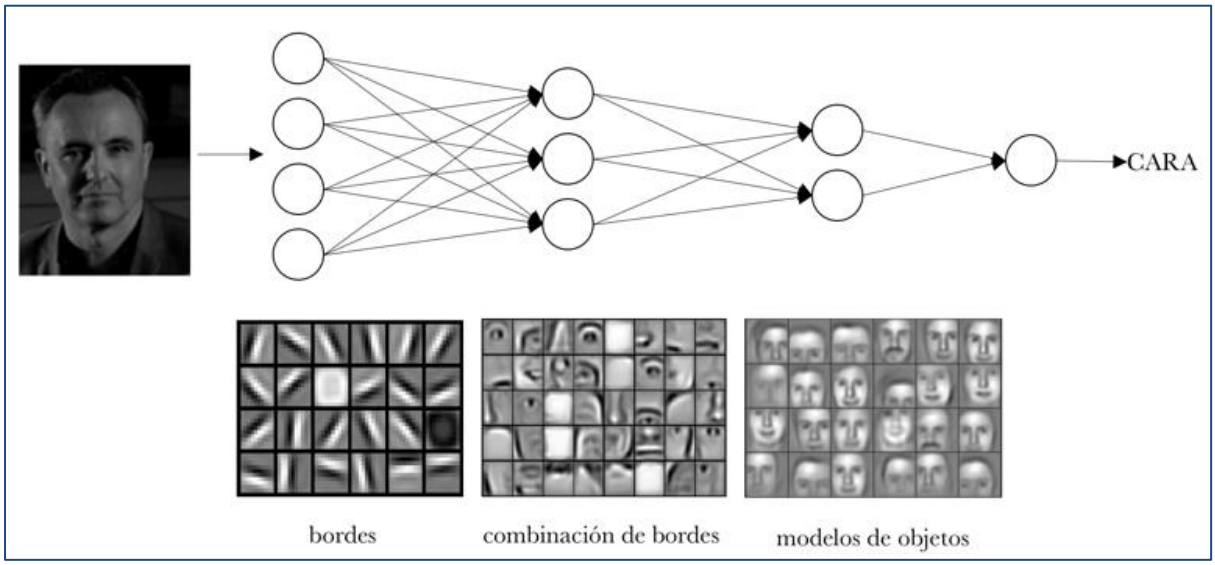

Ilustración 20. Mapas de características de una red neuronal convolucional. Fuente: Jordi Torres AI.

La red toma como entrada los pixeles de una imagen. Si tuviéramos una imagen con 30×30 píxeles equivaldría a 900 neuronas en la capa de entrada (suponiendo que la imagen está en escala de grises). Si tuviéramos una imagen en RGB necesitaríamos 3 canales usando 30x30x3 = 2700 neuronas de entrada. Después pasarían a realizarse las “convoluciones” que consisten en tomar grupos de píxeles (cercanos) de la imagen de entrada e ir operando matemáticamente (producto escalar) con una pequeña matriz llamada kernel. Ese kernel, de tamaño n×n píxeles, recorre las neuronas de entrada (de izquierda a derecha y de arriba a abajo) en pasos de cierto tamaño llamado stride, y genera una nueva matriz de salida que será nuestra nueva capa de neuronas ocultas.

En cuanto al kernel, tendremos un número grande de ellos, siendo distintos entre sí y llamándose su conjunto “filtros”. Por poner un ejemplo, en la primera convolución podríamos tener 32 filtros, lo que nos daría 32 matrices de salida. Si usáramos 64, nos quedarían 64 matrices de salida. A medida que el kernel se desplaza, vamos obteniendo una “nueva imagen” filtrada por el kernel. Siguiendo con el ejemplo, obtendríamos 32 imágenes filtradas nuevas, o dicho de otra forma, nuestra imagen de entrada se convierte en otra imagen de

mayor profundidad. Estas imágenes nuevas van “dibujando” ciertas características de la imagen original; están “aprendiendo”.

El problema principal de todo esto es que con cada “nueva imagen” también necesitamos nuevas neuronas que alimentar. Si suponemos el ejemplo de 30x30 neuronas que nos obligaban a tener 900 neuronas a la entrada, al pasarlas por 32 filtros necesitaríamos para la siguiente convolución 30*30*32=900*32=28800 neuronas. Este número aumentaría cada vez más con las siguientes convoluciones, por lo que se hace necesaria una solución a este problema.

Es aquí donde entra el subsampling, en el que se reduce el tamaño de las imágenes filtradas pero en donde prevalecen las características más importantes detectadas por cada filtro. Existen distintos métodos, pero el más común es el de Max-Pooling, que se muestra en la ilustración 21.

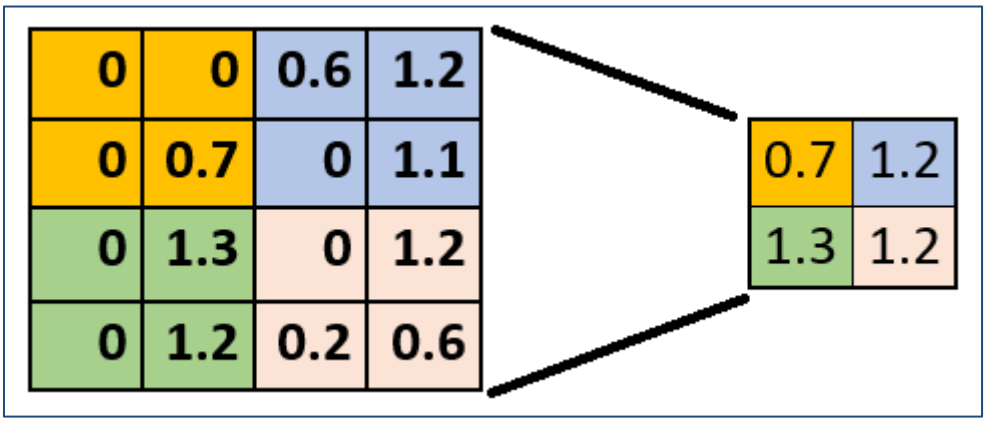

Ilustración 21. Max-pooling.

Suponiendo, por ejemplo, un max-pooling de 2*2, recorreremos cada una de las 32 imágenes de características obtenidas anteriormente de izquierda a derecha y de arriba a abajo tomando 2*2 = 4 píxeles, y preservando solamente el máximo valor. Así, 4 píxeles se ven reducidos a uno sólo, reduciéndose la imagen a la mitad. Por tanto, nos quedarán 32 imágenes de 15*15, necesitando ahora sólo 7200 neuronas y no 28800.

De esta forma podremos ir añadiendo capas convolucionales para ir captando más y más detalles sin introducir tanta complejidad y número de neuronas.

Al terminar de pasar nuestra imagen de entrada por nuestras capas convolucionales, tendremos una imagen con tanta profundidad como filtros de nuestra última capa. Por esto, lo siguiente que se hace es “aplanar” nuestra imagen y así se puede pasar ya a una capa de neuronas “tradicional” en un modo “fully connected”, es decir, el modo en el que todas las neuronas anteriores estarán conectadas con todas las de esta capa.

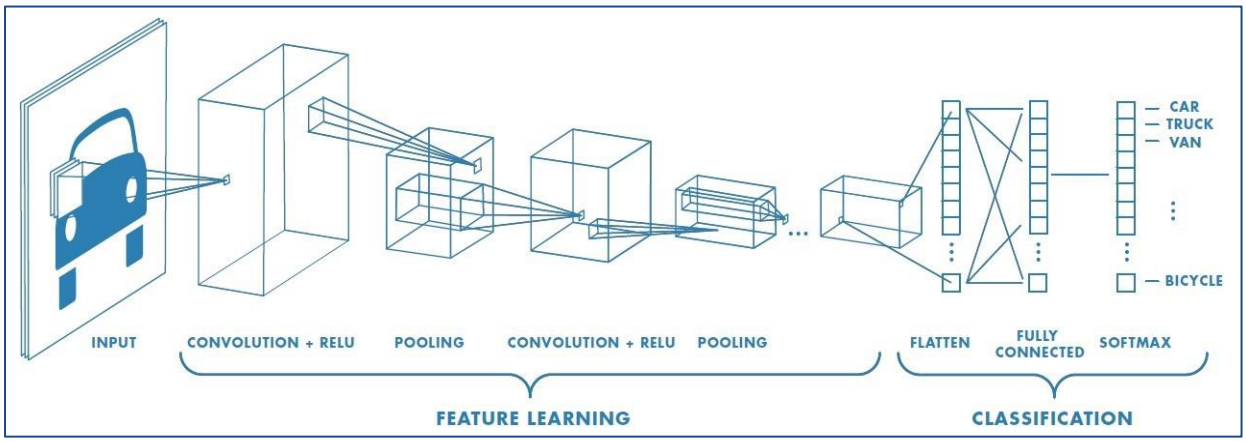

Ilustración 22. Esquema básico de una red neuronal convolucional.

Lo último será nuestra capa de salida con el mismo número de neuronas como casos a clasificar (en nuestro caso gestos a clasificar). Esta última capa usará una función de activación “Softmax”, que ya se ha explicado su utilidad para problemas de clasificación. En las capas convolucionales, en cambio, lo normal es usar “ReLu”.

Este tipo de redes ha revolucionado el mundo de la inteligencia artificial, y con este otros campos de la ciencia, debido a su capacidad para procesar imágenes de forma automática. Por ejemplo, unos investigadores de Google han conseguido predecir la tensión arterial, la edad y varios parámetros más, analizando una fotografía de la retina del sujeto mediante una red neuronal convolucional. [14]

Las redes convolucionales también son útiles para detectar y reconocer movimiento en imágenes y vídeo, como unos investigadores probaron al crear una red neuronal convolucional que detecta el movimiento de los peatones en la calle, cuestión de vital importancia para el funcionamiento seguro de sistemas de conducción autónoma. [15] Esto es útil para nuestro tema de interés, dado que los gestos manuales están conformados por diferentes movimientos de las mismas. De hecho, existe otra investigación en la que se propone una red neuronal convolucional para detectar de forma precoz ciertos “movimientos estereotípicos” propios del autismo u otras afecciones. [16] Por tanto, se trata de una tecnología a tener en cuenta para nuestro proyecto.

5.3.7 Redes Convolucionales 3D

Una variación de las redes convolucionales serían las redes convolucionales 3D. Su funcionamiento es muy similar al de una red convolucional original, simplemente añadiendo una dimensión más a nuestros filtros y operaciones. Esto nos permite usar datos con 3 dimensiones, como por ejemplo vídeos, para poder reconocer patrones en imágenes a lo largo del tiempo.

5.3.8 Redes LSTM

Una red LSTM (Long short-term memory) es un tipo de red neuronal recurrente y, como tal, contiene bucles de realimentación, permitiendo que a través de ellos persista información durante ciclos de entrenamiento. [17]

Al existir esta “percepción temporal” se pueden usar no sólo para procesar elementos individuales de información (como imágenes), sino secuencias enteras de datos (como vídeo o audio). Por ejemplo, las redes LSTM se usan para reconocimiento de escritura [18], reconocimiento del habla [19], detección de movimiento o incluso para detección de anomalías en tráfico de redes de ordenadores conformando sistemas de detección de intrusión. Por tanto, las redes LSTM son útiles para reconocimiento de patrones en vídeos, puesto que recuerdan el estado de las imágenes anteriores mientras se analiza la actual. Esto es relevante para la detección de gestos manuales puesto que dichos gestos ocuparán varios fotogramas en un vídeo.

El problema que tenían las redes recurrentes es su corta memoria (de ahí, short-term memory), por lo que se les escapaban secuencias de datos largas. Para solucionar este problema, se introdujo las redes LSTM. [20]

El concepto clave de las redes LSTM es la “cell” (celda, o célula) y el “cell state”, que sería su estado. El “cell state” sería como una “autopista” de transporte que transfiere información por toda la secuencia y sería como la memoria de la red neuronal. Esto le permite llevar información de pasos previos y de toda la secuencia hasta los pasos más tardíos, reduciendo los efectos de la memoria a corto plazo. Mientras este estado de la célula va viajando por otras células, se va añadiendo o quitando información según las diferentes “gates” o puertas. Estas gates serán las de “forget gate”, para decidir qué información se deberá mantener o eliminar, la input gate, para actualizar el estado de la célula, y por último la output gate, para actualizar el siguiente “estado oculto” de la célula al estilo de las redes recurrentes originales. [21]

Por último, una función de activación muy importante en este tipo de redes sería la función sigmoide, cuya generalización ya hemos explicado antes y sería la función softmax. Por tanto, permite comprimir valores entre 0 y 1. En este contexto, valores cercanos a 0 sería información a olvidar, y valores cercanos a 1 supondría información valiosa a mantener.

Por tanto, vemos que este tipo de redes son tremendamente útiles para detectar y reconocer patrones a lo largo del tiempo, junto con las redes convolucionales 3D. De hecho, es usado, por ejemplo, para reconocer la actividad humana a lo largo del tiempo, usándose en smartphones para aplicaciones de fitness mediante diferentes sensores. [22]

5.4 Aceleración por GPU

Las diferentes arquitecturas de redes neuronales requieren una gran cantidad de recursos computacionales debido a sus numerosas neuronas, que pueden llegar a ser incluso miles de millones. Por esta razón, tecnologías para acelerar el proceso de entrenamiento se hacen tremendamente útiles y necesarias, como por ejemplo, la aceleración por GPU. [23]

Por sus siglas, una GPU es una Graphics Processing Unit, o unidad de procesamiento de gráficos. Se trataría de un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para así aligerar la carga de trabajo del procesador central en aplicaciones como videojuegos, aplicaciones 3D, aplicaciones de diseño gráfico, etc. De esta forma,

mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse al resto de cálculos. [24]

La CPU es una unidad de procesamiento general que está preparada para operaciones matemáticas y lógicas para realizar prácticamente cualquier tarea a una velocidad suficiente. El problema viene cuando usamos programas con una carga gráfica muy grande, como videojuegos, programas de diseño 3D, etc. Estos programas, al estar continuamente procesando formas y texturas de forma matricial, hacerlo por medio de la CPU se vuelve costoso computacionalmente hablando. Para eso existe la GPU que aporta varias ventajas.

La primera de todas es que tenemos un procesador más. Este procesador está específicamente diseñado para trabajar con gráficos, funciones específicas y en general toda forma de operación matricial y cuenta con una arquitectura basada en el procesamiento en paralelo. Resumiendo, con la aceleración por GPU lo que hacemos es quitar cierto trabajo a la CPU y dárselo a la GPU, que lo hará más rápido y mejor.

En el caso del desarrollo de redes neuronales (concretamente en el entrenamiento y a veces en su uso) es posible usar aceleración por GPU pero, sin embargo, no se está trabajando con formas geométricas o gráficos. Entonces, ¿para qué usan la GPU? De forma simple, se puede explicar recordando que la mayoría de los cálculos necesarios en esta área se representan y procesan de forma matricial, igual que los gráficos. Es por esto que podemos usar la GPU para por un lado liberar a la CPU de esta carga de trabajo, y por otro hacerlo de forma más rápida y eficiente. [25]

Esta forma de trabajar se ha vuelto muy popular y fabricantes de tarjetas gráficas como Nvidia han sacado tarjetas pensadas para este fin, así como drivers y librerías específicas (CUDA y cuDNN).

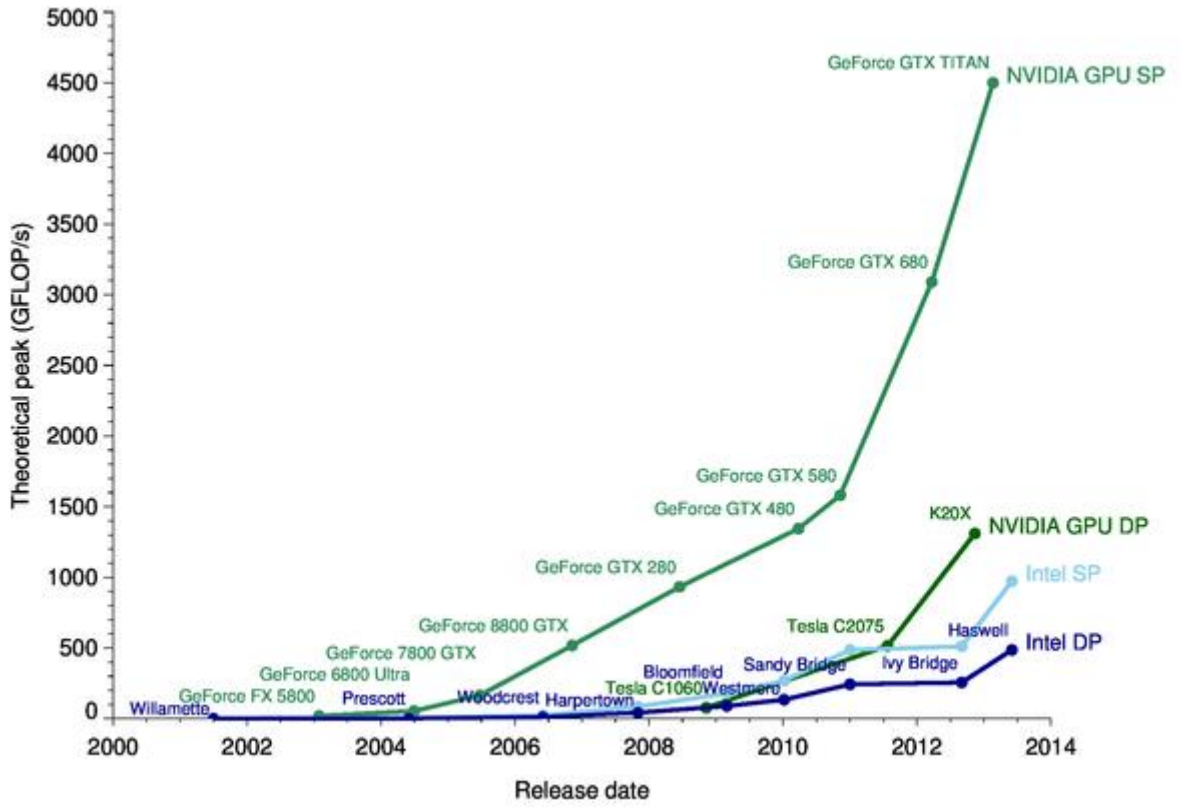

Ilustración 23. Capacidad de procesamiento de GPU (verde) vs CPU (azul) al entrenar redes neuronales. Muestra la mejora de dicha capacidad a lo largo de los años, además de comparar GPU vs CPU. Autor: Michael Galloy.

De hecho, con el ultimo framework de Nvidia dedicado a la aceleración de IA por GPU se pueden conseguir reducciones del tiempo de computación de hasta 50 veces. [26]

5.5 Lenguajes de programación en el desarrollo de redes neuronales

Las redes neuronales se pueden desarrollar en casi cualquier lenguaje de programación disponible. Sin embargo, existen varios lenguajes que ofrecen la flexibilidad y las herramientas necesarias para desarrollar redes neuronales y experimentar con ellas fácilmente.

5.5.1 Python

Python es un lenguaje de programación interpretado, orientado a objetos de alto nivel y con semántica dinámica. Posee una sintaxis relativamente fácil que hace énfasis en la legibilidad del código, facilitando su depuración y favoreciendo la productividad. Su potencia, carácter open source y su facilidad de aprendizaje le ha llevado a tomar la delantera en cuanto a redes neuronales e inteligencia artificial se refiere. [27] De hecho, si en el portal de repositorios GitHub hacemos una búsqueda con las palabras “neural network” (red neuronal en inglés), obtendremos unos 32.516 resultados para Python a fecha de junio de 2020, superando en

casi 8 veces más a su competidor más próximo, C++ (4.272 resultados). Esto nos hace una idea de su uso e influencia en el sector.

Python fue creado por Guido Van Rossum en 1991 y, como curiosidad, debe su nombre a la gran afición de su creador por las películas de Monty Python.

Además de librerías de herramientas científicas, numéricas, de análisis y estructuras de datos, o de algoritmos de Machine Learning como NumPy, Matplotlib, Pandas, Scikit-learn TensorFlow o Keras, Python ofrece entornos interactivos de programación orientados al Data Science y la creación de redes neuronales artificiales.

Por otro lado, Python es un lenguaje de programación multiparadigma. Es decir, permite varios estilos de programación: orientada a objetos, programación imperativa o programación funcional. También posee un modo interactivo en el cual se escriben las sentencias en algo parecido a un intérprete de comandos, pudiendo ser introducidas una a una para así ver el resultado inmediatamente. Esto permite probar porciones de código antes de integrarlo como parte de un programa, lo que puede ser de gran ayuda en el desarrollo de redes neuronales, al permitirnos cambiar pequeños parámetros de nuestra red y ver cómo estos afectan, sin tener que ejecutar todo el código cada vez.

Para hacernos una idea de la sintaxis de Python, en la ilustración 24 se muestra un fragmento de código.

```
# example of decorator
def sampleDecorator(func):
    def addingFunction():
        # some new statments or flow control
        print("This is the added text to the actual function.")
        # calling the function
        func()

    return addingFunction


@sampleDecorator
def actualFunction():
    print("This is the actual function.")
```

Ilustración 24. Ejemplo de código en Python.

5.5.2 C++

C++ es un lenguaje de programación presentado en 1979 por Bjarne Stroustrup. La intención de su desarrollo fue extender al lenguaje de programación C mecanismos para permitir la programación y manipulación de objetos. Esto hace que, desde el punto de vista de los lenguajes orientados a objetos, C++ sea un lenguaje híbrido.

Algunas de las características adicionales de C++ serían la capacidad de agrupación de instrucciones, su naturaleza como lenguaje muy didáctico mediante el cual se pueden sentar

las bases para aprender otros lenguajes con gran facilidad, gran número de compiladores en diferentes plataformas y sistemas operativos, la posibilidad de separación de un programa en módulos que admiten compilación independiente, etc.

Frente a la facilidad de Python, C++ es quizás algo tosco y complicado para la experimentación con redes neuronales. Sin embargo, librerías como TensorFlow también existen en C++, con una sintaxis parecida, lo que facilita las cosas. [28]

5.6 Librerías de apoyo para desarrollo de redes neuronales

A la hora de programar una red neuronal, uno puede codificar todo desde cero sin ayuda de ninguna librería, programando cada neurona, capa, función de activación, derivadas parciales, etc. Por supuesto, no es imposible, pero es obvio que resultaría tremendamente costoso. Además, si se quiere experimentar con la arquitectura, supondrá cambiar por completo el código, lo que complicaría aún más su desarrollo.

Por esta razón, existen librerías de apoyo para el desarrollo de redes neuronales, de las cuales TensorFlow es la primera que vamos a comentar. No obstante, existen más librerías, y a veces unas se usan junto con otras para complementar funcionalidades. También hay que destacar que lo interesante de estas librerías es que la inmensa mayoría de ellas son de código abierto, y/o software libre, lo cual facilita la colaboración y ha propiciado un aumento exponencial en el desarrollo de IA. [29]

5.6.1 TensorFlow

TensorFlow es una biblioteca de código abierto para machine learning desarrollado por Google, y podría decirse que es la plataforma más importante del mundo en este ámbito, siendo la herramienta líder en el sector, además de ser usada por empresas tan dispares como Airbnb, Airbus, Coca-Cola, Intel, PayPal, etc., para sus distintas herramientas. [30] Puede correr en múltiples CPUs y GPUs, y está disponible para Windows, Linux, macOS, e incluso plataformas móviles como Android e iOS.

El nombre TensorFlow deriva de las operaciones que las redes neuronales realizan sobre arrays multidimensionales de datos. Estos arrays multidimensionales se llaman "tensores". Como dato curioso, en mayo de 2016 Google presentó lo que se conoce como TPU (Tensor Processing Unit). Se trata de un tipo de procesador específico para el aprendizaje automático y adaptada para TensorFlow en el que según Google han tenido un rendimiento 10 veces mayor en tareas de aprendizaje automático que los sistemas tradicionales con GPU. [31]

El impacto que ha logrado Google liberando TensorFlow es espectacular, logrando que a fecha de marzo de 2020 existan más de 77.000 repositorios de código donde se referencia TensorFlow (sólo contando GitHub).

Google lleva usando esta tecnología varios años, aplicándola a muchos de sus servicios como por ejemplo Gmail, donde se usa en el componente Smart Reply para generación de

respuestas automáticas, o en el servicio de traducción Google Translation, donde es usado para realizar millones de traducciones todos los días entre multitud de idiomas.

TensorFlow también podría definirse como una librería de diferenciación automática, lo que la coloca en la categoría de alto nivel. Como su nombre indica, una librería de diferenciación automática se encarga de calcular automáticamente todas las derivadas parciales necesarias en la optimización de cualquier arquitectura que se diseñe. Esto es una enorme ventaja puesto que nos evita calcular todas las miles de derivadas parciales necesarias.

5.6.2 PyTorch

Dentro de las librerías de diferenciación automática nos encontramos también con PyTorch.

PyTorch es un paquete de Python que está diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores, permitiendo además su ejecución en GPU para acelerar los cálculos. [32] En este sentido, es similar a TensorFlow.

A pesar de ser una librería bastante reciente dispone de una gran cantidad de manuales y tutoriales donde encontrar ejemplos además de una comunidad que crece cada día, existiendo más de 37.000 referencias en GitHub. También dispone de una interfaz muy sencilla para la creación de redes neuronales pese a trabajar de forma directa con los tensores. Esto le permite funcionar sin la necesidad de una librería a un nivel superior como pueda ser Keras para TensorFlow.

Otra diferencia con TensorFlow es que PyTorch trabaja con grafos dinámicos en vez de estáticos. Esto permite que en tiempo de ejecución se puedan ir modificando las funciones, con lo que el cálculo del gradiente variará con ellas, facilitando la depuración de código. [33]

5.6.3 Keras

Keras es una librería para desarrollo de Redes Neuronales de código abierto escrita en Python. Es capaz de ejecutarse sobre TensorFlow, lo que hace que ambas incrementen su potencial en una especie de simbiosis. Está especialmente diseñada para posibilitar la experimentación en poco tiempo pudiendo manipular fácilmente las redes neuronales, desde sus parámetros hasta arquitecturas completas. Su código se centra en ser amigable para el usuario, modular y extensible. Ofrece un conjunto de abstracciones muy intuitivas y de alto nivel, haciendo más sencillo el desarrollo de modelos de deep learning independientemente del backend computacional utilizado. [34]

En Keras se encuentran implementados los principales bloques constructivos de las redes neuronales, tales como las propias capas, funciones de activación, optimizadores, etc. De esta forma cuando queramos añadir una capa específica sólo tenemos que indicar las características principales de esta capa, es decir, qué tipo de capa queremos, su número de neuronas, su función de activación y poco más. Por esto Keras es muy útil y sencilla a la hora de construir redes neuronales paso a paso y también a la hora de experimentar rápidamente.

Todo esto pondría a Keras en un nivel inmediatamente superior que podríamos definir como el nivel de librerías de composición de capas.

A fecha de abril de 2020, existen más de 32.000 referencias a Keras en el repositorio GitHub, y contaría con más de 250.000 usuarios.

5.6.4 Scikit-learn

Scikit-learn es una útil librería para Machine Learning en Python, de código abierto y es reutilizable en varios contextos fomentando el uso académico y comercial. Proporciona un conjunto de algoritmos de aprendizaje supervisados y no supervisados en Python. Además, está construida sobre SciPy (Scientific Python) e incluye librerías como NumPy, Pandas, SciPy, etc. [35]

A SciKit-learn podríamos situarla en un nivel ya no de capas, sino de modelos completos. En este nivel las cosas se vuelven todavía más sencillas, dado que se nos dan las redes como un todo, como un modelo predefinido en el que sólo tenemos que tocar ciertos hiperparámetros para encontrar nuestro modelo. Esto supone rapidez y sencillez a la hora de experimentar, pero quizás se nos queda muy limitado para realizar proyectos concretos puesto que como se ha explicado ganamos sencillez a costa de flexibilidad.

5.6.5 NumPy

Aunque no se trata de una librería específica y exclusiva para el desarrollo de redes neuronales, o de inteligencia artificial en general, NumPy es un proyecto de computación numérica a tener en cuenta. Consiste en una librería de código abierto fundamental para cualquier desarrollo científico en Python. Provee arrays multidimensionales, objetos derivados de estos, rutinas para operaciones rápidas en arrays incluyendo operaciones matemáticas, lógicas, manipulación de dimensiones, clasificación, transformadas discretas de Fourier, etc. [36]

5.6.6 OpenCV

OpenCV es una librería de código abierto para visión por ordenador y machine learning. Posee más de 2500 algoritmos optimizados, orientados para el reconocimiento facial, identificación de objetos, clasificación de acciones humanas en vídeos, seguimiento de movimiento, etc. En definitiva, es una librería imprescindible para el procesado y reconocimiento de imágenes. OpenCV tiene más de 47.000 usuarios y ha sido descargado más de 18 millones de veces, además de ser usado por compañías como Google, Yahoo!, Microsoft, Intel o IBM. [37]

5.7 Datasets

Una de las partes más importantes, si no la que más, a la hora de desarrollar y entrenar nuestra red neuronal, es tener un conjunto de datos adecuados. Existen multitud de “datasets” disponibles para el usuario en la red, pero no todos están orientados a la detección de gestos manuales. A continuación comentaremos algunos que sí lo están.

5.7.1 Hand Gesture Database – Universidad Politécnica de Madrid

Este dataset, del grupo de Tratamiento de Imágenes de la Universidad Politécnica de Madrid, está compuesto por un set de secuencias de imágenes a color y ha sido usado para validar un sistema de reconocimiento de gestos manuales para interacción humano-máquina. [38]

Por ello, los gestos de este set están diseñados teniendo en cuenta las funcionalidades del ratón de un ordenador. Además, están grabados en una escena realista, con un fondo no uniforme y objetos en movimiento. Está dividido en dos sets, conformados por 6 personas distintas realizando 5 gestos diferentes, los cuales se muestran en la ilustración 25.

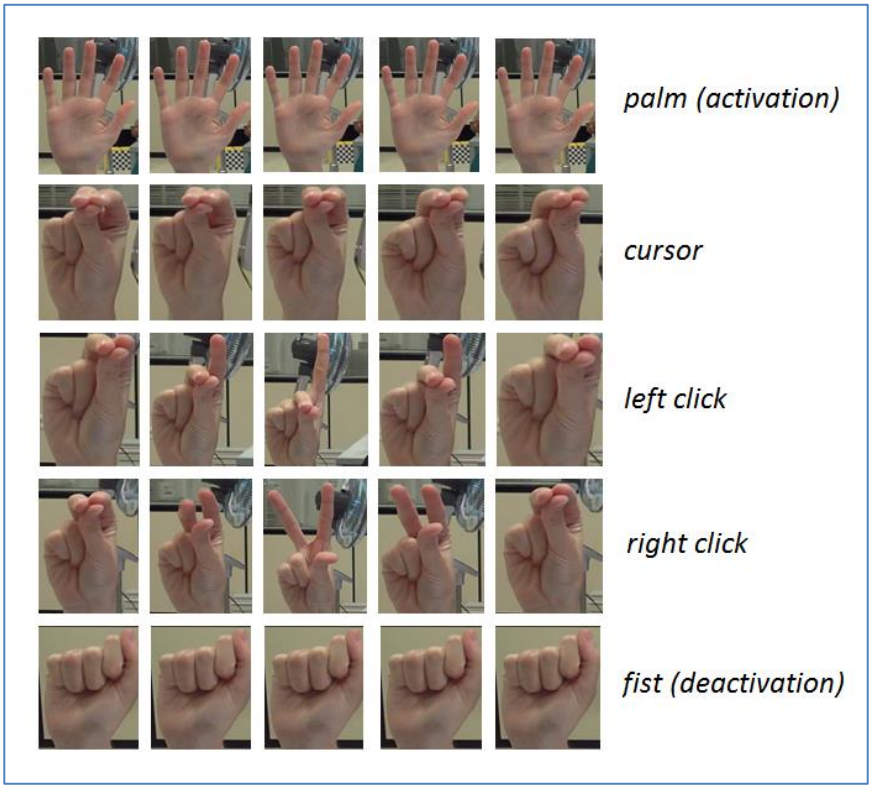

Ilustración 25. Hand Gesture Database – Universidad Politécnica de Madrid

5.7.2 Hand Gesture Recognition Database - Leapmotion

Este dataset está compuesto por 10 gestos manuales realizados por 10 personas distintas. Dichos gestos están capturados mediante el sensor “Leap Motion”. [39]

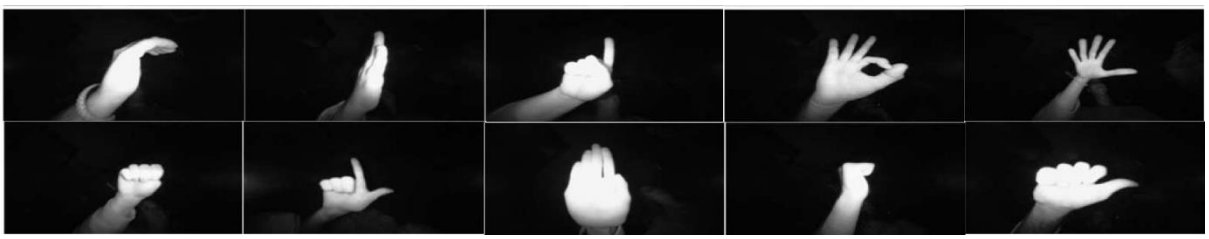

Ilustración 26. Hand Gesture Recognition Database - Leapmotion

5.7.3 “20BN-JESTER” Dataset V1

Este dataset cuenta con 148.092 vídeos de personas realizando un total de 27 gestos manuales distintos. Está dividido en un set de entrenamiento y otro de validación, conteniendo estos 118.562 vídeos y 14.787 vídeos respectivamente además de 14.743 vídeos para testear por nuestra cuenta la red que desarrollemos.

Cuenta también con varios archivos .csv en los que se asocia cada vídeo con su etiqueta, es decir, el gesto correspondiente que lo define (ilustración 27). Entre los gestos disponibles tenemos algunos como “no gesture” (sin gesto), “stop sign” (señal de stop), “swiping left” (desplazamiento a la izquierda), etc. Podemos concluir que es un dataset orientado a establecer interfaces humano-máquina en las que los gestos manuales sustituirían al conjunto ratón-teclado, por ejemplo.

	A	B
1	34870	Drumming Fingers
2	56557	Sliding Two Fingers Right
3	129112	Sliding Two Fingers Down
4	63861	Pulling Two Fingers In
5	131717	Sliding Two Fingers Up
6	23595	Zooming Out With Two Fingers
7	93002	Pulling Hand In
8	136859	Thumb Up
9	68574	Swiping Right
10	119263	Zooming In With Two Fingers
11	60572	Stop Sign
12	118480	Doing other things
13	6522	Swiping Down

Ilustración 27. Organización en archivo .csv

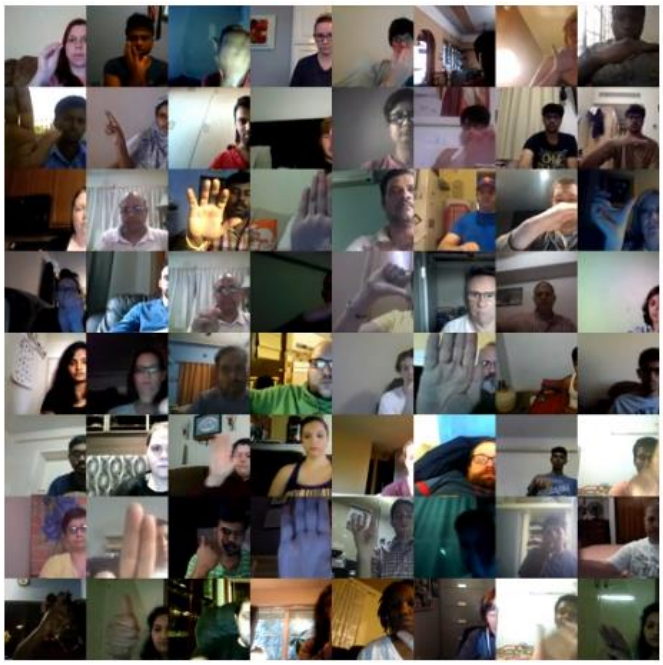

Ilustración 28. Múltiples ejemplos de 20BN-JESTER.

Este dataset se puede usar de forma gratuita para fines académicos, y sus 148.092 vídeos ocuparían unos 22.8 GB.

6. Componentes y estructura funcional del sistema

La estructura de nuestro proyecto constará de varias partes. Por un lado, la red neuronal en sí misma, que tendrá una estructura concreta, con capas de distintos tipos y otras características. Por otro, el software donde se creará, desarrollado en un lenguaje de programación en concreto con sus correspondientes librerías de apoyo. Esto significa que habrá que tener en cuenta los distintos lenguajes de programación existentes y sus librerías, y analizar su idoneidad. Este software también tendrá diferentes partes, o módulos, dependiendo de si estamos en la fase de desarrollo y experimentación de la red neuronal, en la fase de testeo, preparación del dataset, etc.

También deberá "correr" sobre cierto hardware, ya sea para la fase de entrenamiento, testeo o su uso una vez entrenada. Esta sección es de gran importancia, porque limitará el alcance del proyecto si no se elige el adecuado, sobre todo para la parte del entrenamiento que es donde más capacidad de procesamiento y de memoria se necesita. Es recomendable que este hardware cuente con una tarjeta gráfica relativamente potente para así poder aprovechar el potencial de la aceleración por GPU. Dicho hardware también deberá contar con alguna forma de captura de vídeo, ya sea por webcam u otros medios.

La última parte, y quizás más importante, sería el dataset, con el que entrenaremos nuestra red.

En resumen, tendremos:

- Hardware
    - Un PC con webcam y tarjeta gráfica
- Data Set
- Software
    - Módulos software para:
        - Preparar un set de entrenamiento
        - Preparar un set de validación
        - Entrenar la red neuronal
        - Testear/usar la red neuronal
    - Estos módulos son codificados mediante:
        - Lenguaje de programación concreto
        - Librerías de apoyo para desarrollo de redes neuronales
    - Con todo ello desarrollamos, entrenamos y testeamos la red neuronal

Para cada parte, habrá que analizar de entre las diferentes alternativas existentes y elegir la solución más adecuada.

7. Análisis de alternativas

A la hora de desarrollar la red neuronal convolucional para el reconocimiento de gestos manuales debemos encontrar cuáles son las herramientas más convenientes para este fin, además de comprobar otros criterios. Este análisis lo debemos realizar para cada una de las partes del sistema, es decir, para el hardware, los distintos módulos software, y el data set.

7.1 Lenguaje de programación

7.1.1 Alternativas

De entre los distintos lenguajes de programación existentes, se ha visto que los más adecuados, y los que más se usan para el desarrollo de redes neuronales son Python y C++. Por tanto, nuestras alternativas serán estos dos lenguajes.

7.1.2 Criterios de selección

Para elegir el lenguaje de programación adecuado para nuestro proyecto, debemos fijarnos en diversos criterios tales como facilidad de uso, librerías y bibliotecas disponibles para cada lenguaje, nivel de abstracción, etc. También son importantes otros aspectos como cuán usados son en la investigación, puesto que cuanto más usado sea un lenguaje mayores serán los recursos bibliográficos disponibles y la ayuda en internet.

7.1.3 Selección de la solución

Teniendo en cuenta los criterios mencionados, se selecciona como lenguaje para el proyecto el lenguaje de programación Python, por su nivel de abstracción y facilidad de uso, así como por ser el más usado en el desarrollo de redes neuronales, lo que nos será de gran aportación a la hora de buscar bibliografía o ayuda en internet.

7.2 Librerías de apoyo para redes neuronales

7.2.1 Alternativas

Las alternativas disponibles que se han analizado consisten en TensorFlow, Keras, PyTorch, Scikit-Learn, NumPy y OpenCV.

7.2.2 Criterios de selección

En cuanto a los criterios de selección de las librerías de apoyo, buscaremos unos criterios similares a los del lenguaje de programación: facilidad de uso, nivel de abstracción, cantidad de recursos bibliográficos disponibles, etc.

Sobre todo, es importante el del nivel de abstracción, ya que determinará la facilidad con la que se podrá experimentar en el desarrollo de nuestra red neuronal artificial. Un nivel muy

alto nos permitirá crear rápidamente y sin apenas conocimiento nuestra red, pero no nos dará tanta flexibilidad como la que nos daría un nivel más bajo de abstracción.

7.2.3 Selección de la solución

Al tener en cuenta los criterios de selección, se ha visto que el uso combinado de TensorFlow junto con Keras nos iba a ofrecer un buen nivel de abstracción junto con un buen nivel de flexibilidad. Al trabajar con Keras sobre TensorFlow, podremos trabajar a nivel de "capa", y así experimentar rápidamente con la arquitectura de nuestra red, para obtener los mejores resultados. También son las librerías más usadas, por lo que encontraremos gran cantidad de trabajos y bibliografía en caso de necesidad.

Hay que añadir que habrá que trabajar con multitud de librerías más, pero no son relevantes de analizar dado que no constituyen el "núcleo" de la red neuronal y no condicionan el desarrollo ni la arquitectura de la misma. Simplemente, nos serán útiles para preprocesar los datos u otras cuestiones adicionales. Algunas de estas librerías serán OpenCV, NumPy o Scikit-Learn.

7.3 Dataset

7.3.1 Alternativas

Las alternativas más relevantes en este aspecto son las de "Hand Gesture Database – Universidad Politécnica de Madrid", "Hand Gesture Recognition Database – Leapmotion" y "20BN-JESTER" Dataset V1.

7.3.2 Criterios de selección

La elección del dataset adecuado es probablemente uno de los aspectos más importantes del proyecto, dado que puede incluso condicionar el enfoque, arquitectura u otros parámetros del mismo. A la hora de su elección, debemos fijarnos en su tamaño y naturaleza. Un dataset pequeño no tendrá suficientes muestras para entrenar a nuestra red, por lo que podremos encontrarnos con underfitting o incluso overfitting si la entrenamos demasiado, puesto que la red memorizará esos pocos ejemplos sin poder generalizar. También es importante su naturaleza, es decir, cómo está conformado. Lo ideal es un dataset variado, con multitud de ejemplos distintos para las mismas categorías. En definitiva, si lo que buscamos es un dataset para el reconocimiento de gestos manuales, necesitaremos un gran número de ejemplos de personas distintas realizando distintos gestos.

7.3.3 Selección de la solución

Como lo que buscamos es un gran dataset variado, se ha llegado a la conclusión de que el dataset "20BN-JESTER Dataset V1" era el más adecuado para el proyecto. Sus 148.092 vídeos distintos de personas realizando 27 gestos diferentes, nos dotarán de la cantidad y variedad necesarias para un correcto entrenamiento de la red neuronal artificial.

7.4 Hardware

Para la parte hardware, contamos únicamente con un PC de la marca “Lenovo”, con procesador Intel Core i7-6700HQ CPU @ 2.60GHz, webcam integrada para la adquisición de vídeo del usuario, y una tarjeta gráfica Nvidia GeForce GTX 950 M, con 4 GB de memoria de vídeo y capacidad para aceleración por GPU para el entrenamiento de redes neuronales. Por tanto, usaremos este equipo con estas características para el desarrollo de nuestro proyecto, y aprovecharemos la aceleración por GPU.

7.5 Arquitectura de la red neuronal artificial

7.5.1 Alternativas

Las diferentes redes neuronales que se han encontrado útiles para resolver nuestro problema de reconocimiento de gestos manuales son dos: las redes convolucionales y las redes LSTM, concretamente en sus variantes 3D y 2D respectivamente.

7.5.2 Criterios de selección

A la hora de seleccionar el tipo de red neuronal y su arquitectura, el criterio de partida es si la red sirve o no para resolver el problema del reconocimiento de gestos manuales, en base a nuestra investigación preliminar.

Además, lo normal es que la arquitectura inicial nunca sea la final. Esto es debido a que cuando se desarrolla una arquitectura concreta de una red neuronal, al entrenarla, se encuentran fallos o resultados todavía no óptimos como el fenómeno del overfitting. Nunca sabremos que nuestra elección es la mejor hasta estar entrenada y testeada. Por ello, se van haciendo pequeños cambios en la red y se vuelve a entrenar y a comprobar, resultando en un proceso iterativo de diseño, entrenamiento y testeo.

En resumen, se elegirá un tipo de red que preveamos que sea útil en el alcance de nuestro objetivo, pero que seguramente se distancie mucho del resultado final.

7.5.3 Selección de la solución

Al ser ambas alternativas, redes convolucionales y redes LSTM, útiles en la resolución de nuestro problema, se ha propuesto el uso combinado de ambas como diferentes capas. Al hacer esto, podremos lograr que nuestra red aprenda pequeñas características espacio-temporales de los gestos manuales en cortos periodos de tiempo, gracias a las capas convolucionales 3D, para luego, a partir de esas características, aprenda otras características espacio-temporales mucho más largas en el tiempo, gracias a las capas LSTM.

7.6 Gestos manuales

7.6.1 Alternativas

Dentro del dataset elegido, 20BN-JESTER Dataset V1, tenemos 27 gestos manuales. Uno a uno, estos serían: Doing other things, Drumming Fingers, No gesture, Pulling Hand In, Pulling Two Fingers In, Pushing Hand Away, Pushing Two Fingers Away, Rolling Hand Backward, Rolling Hand Forward, Shaking Hand, Sliding Two Fingers Down, Sliding Two, Fingers Left, Sliding Two Fingers Right, Sliding Two Fingers Up, Stop Sign, Swiping Down, Swiping Left, Swiping Right, Swiping Up, Thumb Down, Thumb Up, Turning Hand Clockwise, Turning Hand Counterclockwise, Zooming In With Full Hand, Zooming In With Two Fingers, Zooming Out With Full Hand y Zooming Out With Two Fingers.

7.6.2 Criterios de selección

El criterio de selección a seguir consiste en elegir los gestos más adecuados que se necesitarían para desarrollar, con nuestra red neuronal ya entrenada, una interfaz humano-computadora, en el que el usuario usaría gestos manuales para navegar por menús, aceptar, cancelar, etc. Dicha interfaz no se implementará en este TFM, pero la red neuronal artificial del mismo sí estará orientada con ese fin. Por ello, es importante elegir unos gestos adecuados.

7.6.3 Selección de la solución

En base a los criterios de selección, se han elegido los siguientes gestos manuales: “Swiping Left”, “Swiping Right”, “Swiping Up”, “Swiping Down” , “Thumb Up”, “Thumb Down”, “Stop Sign”, “Pulling Hand In” y “Pushing Hand Away”.

Los agrupamos de la siguiente manera:

Desplazamientos de la mano:

- Swiping Right (a la derecha)
- Swiping Left (a la izquierda)
- Swiping Up (arriba)
- Swiping Down (abajo)

Zoom in/out

- Pulling Hand In (mover la mano hacia dentro, zoom in)
- Pushing Hand Away (mover la mano hacia fuera, zoom out)

Parada

- Stop Sign (señal de stop)

Aprobación

- Thumb Up (pulgar arriba, aceptar)
- Thumb Down (pulgar abajo, cancelar)

8. Diseño y desarrollo de la solución

Este apartado documenta todo el proceso de desarrollo de la solución, partiendo desde cero hasta la consecución del reconocimiento de gestos manuales mediante una red neuronal artificial.

Para conseguir el objetivo del proyecto, se ha desarrollado un módulo software en el que se modela una red neuronal artificial. Esta red ha sido entrenada con un conjunto de datos concreto y ha aprendido a reconocer diferentes gestos manuales a partir de esos datos. Después, se ha creado otro módulo software, el cual usaría esta red neuronal ya entrenada para detectar y reconocer gestos manuales usando la webcam de un ordenador. Otros módulos software también serían los dedicados a la preparación de un set de datos manejable a partir del set de datos original.

8.1 Preparación de los datos

8.1.1 Obtención del “training set” y del “validation set”

El set de datos de partida es relativamente grande y posee muchos gestos distintos. Al menos, al principio de nuestro desarrollo, no usaremos todos los gestos disponibles, ni tampoco todas las muestras de cada gesto. Simplemente iremos probando mediante ensayo y error la capacidad de nuestra red neuronal para reconocer unos pocos gestos manuales, y la capacidad de nuestro equipo de procesar y entrenar la arquitectura deseada. Cuando tengamos resultados óptimos, iremos aumentando sus capacidades a más y más gestos. Además, necesitamos separar una parte de los datos para entrenar la red y otra para validarla.

Por esto, se ha creado un módulo software que a partir del set de datos original crea un training set y un validation set con los gestos elegidos, dándonos ese dinamismo que necesitamos para trabajar o no con los datos que queremos en cada momento.

```
#la siguiente función copia las samples que queremos del dataset según número de samples y clases (gestos)

for c in gestos:                #Para cada gesto que necesitamos
    file = open(csv_file, 'r')
    reader = csv.reader(file, delimiter=';')

    for line in reader:         #leemos una línea del archivo .csv donde están los gestos con sus referencias
        target = line[1]        #el segundo elemento de la linea es la clase (el gesto)
        ref = line[0]           #el primero es su referencia  (un número que es el nombre del directorio donde está el gesto)

        if target == c:         #si la clase leída coincide con la que buscamos

            ondo=1
            if ondo == 1:    #si se copia bien
                contador += 1   #aumentamos el contador

                rows.append([ref, target])   # y ponemos la referencia y la clase en la lista "rows"
            if contador == samples:    #finalmente comprobamos si tenemos las samples que queremos
                contador=0             #si es así, reiniciamos el contador y
                break               # finalizamos la búsqueda para seguir con la siguiente clase
```

Ilustración 29. TrainSet_Creating.py

El código mostrado en la ilustración 29 sería el grueso de este módulo. Básicamente, lo que hacemos es leer un archivo .csv que viene con el dataset elegido, y sobre él vamos buscando los gestos que queremos para después copiar sus referencias en otro archivo .csv que será el que nosotros usaremos.

	A	B	C	D
1	34870	Drumming Fingers		
2	56557	Sliding Two Fingers Right		
3	129112	Sliding Two Fingers Down		
4	63861	Pulling Two Fingers In		
5	131717	Sliding Two Fingers Up		
6	23595	Zooming Out With Two Fingers		
7	93002	Pulling Hand In		
8	136859	Thumb Up		
9	68574	Swiping Right		
10	119263	Zooming In With Two Fingers		
11	60572	Stop Sign		
12	118480	Doing other things		
13	6522	Swiping Down		
14	20706	No gesture		
15	42237	Thumb Down		
16	142698	Pulling Hand In		
17	56025	Rolling Hand Forward		
18	118847	Pushing Hand Away		
19	63763	Zooming Out With Two Fingers		
20	133442	Zooming Out With Full Hand		
21	100344	Shaking Hand		

Ilustración 30. Ejemplo de jester-v1-train.csv

El archivo .csv del que se habla se llama “jester-v1-train.csv”, y una muestra de él aparece en la ilustración 30. Como se puede observar, en la primera columna se encuentran las referencias numéricas a cada gesto (más correctamente, a cada directorio que contiene un vídeo de un gesto distinto), y en la segunda columna se indica qué gesto es cada uno. En total, tenemos 118562 vídeos de 27 gestos distintos referenciados en este archivo, y se corresponde con el conjunto de entrenamiento original. Pero, como hemos dicho, este conjunto es muy grande, de ahí nuestra necesidad de elegir un training set más adecuado y manejable para nuestro equipo.

Para ello, nuestro programa hace lo siguiente: para cada clase (gesto) que queremos, lee el archivo y mira una línea. Si esa línea se corresponde a un vídeo del gesto que queremos, copiamos la referencia de ese vídeo en nuestro archivo con extensión .csv personalizado. Esto se va realizando iterativamente hasta que completamos el número de samples (número de vídeos) que queremos para cada gesto.

Este archivo personalizado con las referencias será leído de forma consecutiva, pudiendo aparecer homogeneidades. Para evitar esto y hacer la lista heterogénea, mezclaremos las referencias de forma aleatoria (ilustración 31).

```
import random

random.shuffle(rows)            #la búsqueda y copia anterior la hemos realizado consecutivamente así que aleatorizamos la lista

# you can uncomment this code if you want to create your own csv for the samples
with open(path_destino+'train.csv', 'wt',newline='') as f:          #guardamos la lista en un archivo
    csv_writer = csv.writer(f, delimiter=';')
    csv_writer.writerows(rows)
```

Ilustración 31. Aleatorización.

La otra sección del módulo software creada para obtener un validation set personalizado es exactamente igual que el anterior salvo una diferencia, la cual es que mira en el archivo "jester-v1-validation.csv" que contiene las referencias a los vídeos del set de validación original. Pero en esencia el algoritmo es exactamente el mismo, y por esa razón no se explicará.

8.1.2 Preprocesamiento de los datos

Antes de crear y entrenar la red neuronal, es necesario un preprocesamiento de los datos de entrada. Esto es debido a que los vídeos del dataset vienen con diferentes tamaños, número de frames, resolución, etc., por lo que se hace necesario procesar y normalizar estos datos antes de introducirlos en nuestra red. También se pasarán las imágenes a escala de grises, para así reducir la profundidad de las imágenes y simplificar los datos y el procesamiento posterior.

Este preprocesamiento se realizará en un módulo software, para el cual hacen falta importar ciertas librerías de apoyo. Esto lo hacemos de la siguiente manera (ilustración 32).

```
import numpy as np
import cv2
import tensorflow as tf
import os
import math
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

Ilustración 32. Imports.

En orden, importamos las librerías NumPy, OpenCV, TensorFlow, el módulo del sistema Os, el módulo de Python Math, Pandas y Scikit-learn. Nótese que no se importa Keras, porque viene incluida con TensorFlow. Con esto, tendríamos lo necesario para comenzar.

Como comentábamos, empezamos el preprocesamiento pasando las imágenes a escala de grises. Para ello:

```
# return gray image
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])  #Y' = 0.2989 R + 0.5870 G + 0.1140 B
```

Ilustración 33. Función escala de grises.

Esta función se usará más adelante y simplemente pasaría la imagen de entrada de la función de un esquema RGB a escala de grises. Esto lo hace multiplicando a cada pixel del plano de color rojo por 0.2989, en el caso del plano verde por 0.5870, y por último en el caso del azul por 0.1140. Después se suman los valores obteniendo una imagen en escala de grises. Dicho de otra forma, se trata de una suma ponderada de los diferentes planos de colores de un solo píxel. Nótese que se le da más valor al color verde, puesto que es el color que mejor detecta el ojo humano. [40]

Otra cuestión importante es normalizar las longitudes de los vídeos. Cada vídeo del data set viene almacenado en su respectivo directorio como una secuencia de imágenes. Es decir, no están almacenados como archivos de vídeo, si no como un montón de archivos de imágenes que corresponden con sus fotogramas. Algunos tienen más imágenes que otros, pero de media tienen unas 30 imágenes. Por eso, elegiremos esta cifra para normalizar.

Esto lo haremos eliminando los últimos fotogramas a partir de 30 para los vídeos más largos, y copiando el último fotograma hasta conformar 30 para los vídeos más cortos. De esta forma, todos tendrán 30 fotogramas.

La parte del código que realiza esto es la siguiente (ilustración 34):

```
max_frames = 30 # número máximo de frames (fotogramas)
# esta función unifica los frames para cada muestra de datos
def unificar_frames(path):
    # obtenemos las frames
    frames = os.listdir(path)
    frames_contador = len(frames)
    # unificamos
    if max_frames > frames_contador:
        # duplicamos el último frame si es más corto que lo máximo
        frames += [frames[-1]] * (max_frames - frames_contador)
    elif max_frames < frames_contador:
        #eliminamos los últimos frames si es más largo que lo máximo
        frames = frames[0:max_frames]
    return frames
```

Ilustración 34. Función de unificación.

La otra cuestión importante es que todos los fotogramas tengan las mismas dimensiones, que en nuestro caso serán 64x64. Para ello usamos las funciones de la ilustración 35, y utilizamos también la función cv2.resize de OpenCV.

```
# resize frames
def resize_frame(frame):
    frame = img.imread(frame)
    frame = cv2.resize(frame, (64, 64))
    return frame
```

Ilustración 35. Función de reescalado.

Los gestos manuales abarcan varios fotogramas en el tiempo, de ahí que usemos capas convolucionales y LSTM. Esto significa que tendremos que “alimentar” a nuestra red con varios fotogramas a la vez, que en nuestro caso, por elección propia, serán 15 fotogramas a la vez. Por ello, cada muestra, es decir, cada vídeo, se dividirá en dos partes iguales, obteniendo ahora 2 muestras. De esto se encarga la siguiente parte del código (ilustración 36), con el que dividimos cada muestra de 30 fotogramas en 2 muestras de 15 fotogramas.

```
training_targets = [] # lista de clases (gestos)
new_frames = [] # lista para almacenar los frames

for target_key in targets[0]:   #targets es un diccionario: en 0 tiene una lista de target_keys y en 1 tiene una lista de targets
   #los target_keys son las referencias de cada video
   #los targets son los videos
   #por ello, vamos recorriendo cada video en el for
    new_frame = [] # nueva muestra

    directory= str(targets[0][target_key])  # pasamos a string la referencia del directorio
    frames = unificar_frames(path+directory) #unificamos el video de ese directorio (hacemos que tenga 30 frames)
    if len(frames) == max_frames: # comprobamos por si acaso
        for frame in frames: #para cada frame del video
            frame = resize_frame(path+directory+'/'+frame)  #hacemos que todas tengan el mismo tamaño de frame (64*64)
            new_frame.append(rgb2gray(frame))  #pasamos a escala de grises
            if len(new_frame) == 15: # particionamos en dos muestras de 15 y 15
                new_frames.append(new_frame) # las metemos en la lista
                #accedo al target con esa target_key, miro qué clase es, y meto el nombre de la clase en la lista de clases
                training_targets.append(targets_name.index(targets[1][target_key]))
                #libero la lista new_frame
                new_frame = []
                counter_training+=1
```

Ilustración 36. División de vídeo.

Además, es aquí donde usamos los métodos anteriores de unificación, reescalado y transformación a escala de grises.

Aun así, todavía hay que pasar estas imágenes a un formato que la red neuronal pueda leer. Al usar Keras, nuestra red aceptará arrays de la librería NumPy, por lo que tenemos que transformar nuestros datos para que sean un array de ese tipo. También es interesante pasar los datos a float32, de 32 bits, en vez de trabajar con 64 bits. Con 64 bits usaríamos el doble de memoria y el doble de tiempo de procesamiento y entrenamiento, para no ganar apenas ventajas puesto que nos vale con 32 bits para definir con buenos detalles las imágenes.

```
# convertimos datos de entrenamiento a np float32
training_data = np.array(new_frames[0:counter_training], dtype=np.float32)
```

Ilustración 37. Conversión de tipo de datos.

De esta forma ya tendremos nuestro array NumPy de 32 bits.

También pasamos nuestras etiquetas (los nombres de los gestos de cada muestra) a un formato NumPy (ilustración 38).

```
y_train = np.array(training_targets)
```

Ilustración 38. Formato NumPy.

El último cambio a realizar a nuestros datos sería el de la estandarización, que consiste en reescalar los datos para que tengan media μ=0 y desviación típica σ=1. Esto es algo obligatorio para redes neuronales, puesto que de no hacerlo, algunos errores cobrarían más importancia que otros, entrenándose la red de forma no uniforme o no entrenándose en absoluto. Para ello, usamos las funciones de la ilustración 39, entre las que se encuentra la función “StandardScaler()” del paquete Scikit-learn.

```
scaler = StandardScaler()     #creamos un objeto de la clase sklearn.preprocessing.StandardScaler
#transformamos a un array de una dimension y estandarizamos
scaled_images = scaler.fit_transform(training_data.reshape(-1, 15*64*64))
scaled_images  = scaled_images.reshape(-1, 15, 64, 64, 1)  #devolvemos la forma

#lo devolvemos a un array Numpy óptimo para pasarlo a la red
x_train = np.array(scaled_images)
```

Ilustración 39. Estandarización.

Finalmente, tendríamos listo el array “x_train” para pasarlo ya a la red neuronal y comenzar el entrenamiento.

Para los datos de validación, haremos exactamente el mismo proceso, por lo que no se explicará.

8.2 Aceleración por GPU

La aceleración por hardware, en este caso por GPU, es una técnica muy usada para acelerar el entrenamiento de redes neuronales artificiales. En nuestro caso, la GPU utilizada es la de la tarjeta gráfica Nvidia GeForce GTX 950M, que cuyo nombre indica, es de la marca Nvidia. Esta marca pone a disposición de los usuarios diferentes bibliotecas y librerías para implementar la aceleración por GPU en sus dispositivos, como CUDA y cuDNN.

CUDA es un toolkit de Nvidia que provee un entorno para el desarrollo de aplicaciones, haciendo uso de la aceleración por GPU. [41] Por otra parte, cuDNN es un conjunto de librerías de primitivas aceleradas por GPU para el desarrollo de redes neuronales artificiales. [42] Para usar la aceleración por GPU sobre nuestra tarjeta gráfica Nvidia, necesitamos instalar tanto CUDA como cuDNN, además de realizar otras configuraciones en nuestro equipo. Esto no se explicará aquí. Sin embargo, existe una guía detallada en la página ofical de TensorFlow, en el apartado de “Asistencia para GPU” (https://www.TensorFlow.org/install/gpu).

Lo que sí hay que señalar, es que debemos asegurarnos de que la versión que usamos de TensorFlow es compatible con la aceleración por GPU. Para TensorFlow 1.15 y versiones anteriores, los paquetes de CPU y GPU son independientes, pero para versiones superiores no hay que preocuparse y nuestro TensorFlow permitirá dicha aceleración. Para comprobar nuestra versión de TensorFlow instalada hacemos lo mostrado en la ilustración 40, en el intérprete de comandos de Python.

```
>>> import tensorflow as tf
2020-07-12 15:44:44.207588: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cudart64_101.dll
>>> tf.__version__
'2.1.0'
```

Ilustración 40. Comprobación de versión TensorFlow.

Como vemos, nuestra versión de TensorFlow es la 2.1.0, superior a la 1.15, por lo que permite la aceleración por GPU.

El último “comando” que hemos escrito en la consola, lo escribiremos también como sentencia en nuestro software, para asegurarnos de qué versión tenemos cada vez que ejecutemos el programa. Después, debemos comprobar cuántos y qué dispositivos tiene nuestro equipo capaces de permitir la aceleración por GPU. Esto lo haremos de la siguiente manera (ilustración 41), en nuestro código Python.

```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

Ilustración 41. Comprobación aceleración GPU. Input.

El resultado de ejecutar esto sería lo siguiente (ilustración 42).

```
2020-07-12 16:18:37.547046: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
2020-07-12 16:18:37.561608: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library nvcuda.dll
2020-07-12 16:18:37.607847: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1555] Found device 0 with properties:
pciBusID: 0000:01:00.0 name: GeForce GTX 950M computeCapability: 5.0
coreClock: 1.124GHz coreCount: 5 deviceMemorySize: 4.00GiB deviceMemoryBandwidth: 29.83GiB/s
2020-07-12 16:18:37.621190: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cudart64_101.dll
2020-07-12 16:18:37.799876: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cublas64_10.dll
2020-07-12 16:18:37.939742: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cufft64_10.dll
2020-07-12 16:18:37.992328: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library curand64_10.dll
2020-07-12 16:18:38.118891: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cusolver64_10.dll
2020-07-12 16:18:38.194988: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cusparse64_10.dll
2020-07-12 16:18:38.433906: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cudnn64_7.dll
2020-07-12 16:18:38.446424: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1697] Adding visible gpu devices: 0
2020-07-12 16:18:45.227908: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1096] Device interconnect StreamExecutor with strength 1 edge matrix:
2020-07-12 16:18:45.236551: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1102]      0
2020-07-12 16:18:45.240826: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] 0:   N
2020-07-12 16:18:45.247316: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1241] Created TensorFlow device (/device:GPU:0 with 3033 MB memory) -> physical GPU (device: 0, name: GeForce GTX 950M, pci bus id
0000:01:00.0, compute capability: 5.0)
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 4966048915014491445
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 3181130547
locality {
  bus_id: 1
  links {
  }
}
incarnation: 17297127081769882237
physical_device_desc: "device: 0, name: GeForce GTX 950M, pci bus id: 0000:01:00.0, compute capability: 5.0"
]
```

```
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 3181130547
locality {
  bus_id: 1
  links {
  }
}
incarnation: 14359617424826487195
physical_device_desc: "device: 0, name: GeForce GTX 950M, pci bus id: 0000:01:00.0, compute capability: 5.0"
]
```

Ilustración 43. Comprobación aceleración GPU. Zoom output.

Podemos observar que se ha detectado nuestra tarjeta gráfica, por lo que podremos usarla para acelerar el entrenamiento de nuestra red neuronal. No habría que hacer nada más, por lo que ya tenemos todo listo para comenzar el desarrollo de la red neuronal.

8.3 Desarrollo iterativo de la red neuronal y entrenamiento

Cuando hablamos de cómo se desarrolla una red neuronal por un lado, y de cómo se entrena por otro, en verdad estamos hablando de dos caras de la misma moneda. Un aspecto crucial y que debe quedar claro, es que el desarrollo de una red neuronal es un proceso iterativo. Normalmente, se plantea una arquitectura de partida que creemos que va a dar buenos resultados y se entrena esa red neuronal con esa arquitectura. Después, analizando diferentes parámetros, observamos si se ha producido overfitting, underfitting, o un entrenamiento correcto. Si hay alguno de esos problemas y no ha habido un entrenamiento correcto, habrá que introducir cambios en la arquitectura, modificar parámetros del entrenamiento, mejorar nuestro dataset, etc. Una vez hechos estos cambios, se vuelve a entrenar y a analizar, resultando en un proceso iterativo de ensayo y error que en la mayoría de ocasiones depende de la intuición y conocimientos del programador.

8.3.1 Arquitectura de partida

La arquitectura de partida que se ha elegido es una combinación de las redes convolucionales 3D y las redes LSTM 2D. Por ello, tendremos una capa de entrada que será una capa convolucional 3D con 32 filtros, seguida de su correspondiente capa de “max pooling”. Después tendremos otra capa convolucional 3D con 64 filtros, seguida también de su correspondiente capa de “max pooling”. La última capa de esta primera parte será nuestra capa LSTM 2D.

Recordar que, al terminar de pasar nuestra imagen de entrada por nuestras capas convolucionales, y nuestra capa LSTM, tendremos una imagen con tanta profundidad como filtros de nuestra última capa. Por ello, lo siguiente que se hace es “aplanar” nuestra imagen para pasar ya a una capa de neuronas “tradicional” en un modo “fully connected”, es decir, el modo en el que todas las neuronas anteriores estarán conectadas con todas las de esta capa, como una malla.

Recordar también que antes de la existencia de las redes convolucionales, lo que se hacía era ajustar los filtros a mano para luego ya pasar las imágenes filtradas a una red neuronal convencional. Lo que hacemos aquí, en cambio, es hacer que esos filtros sean parte de la red neuronal artificial completa, dejando que esta aprenda los coeficientes correctos de los filtros, como también los de la red neuronal convencional que le sigue.

Por todo esto, lo que hacemos después es pasar todo a una capa “Flatten” o de aplanamiento, para poder pasar ya a una capa densa “fully connected” con 128 neuronas y función de activación ReLu, y por último una capa densa de salida con tantas neuronas como clases y con función de activación “SoftMax”, típica en problemas de clasificación. Nuestro desarrollo será iterativo, con unas pocas clases (gestos) de partida, para luego ir aumentando las

capacidades de reconocimiento, por lo que el número de neuronas de esta última clase irá variando.

En la ilustración 44 vemos un esquema de esta primera arquitectura.

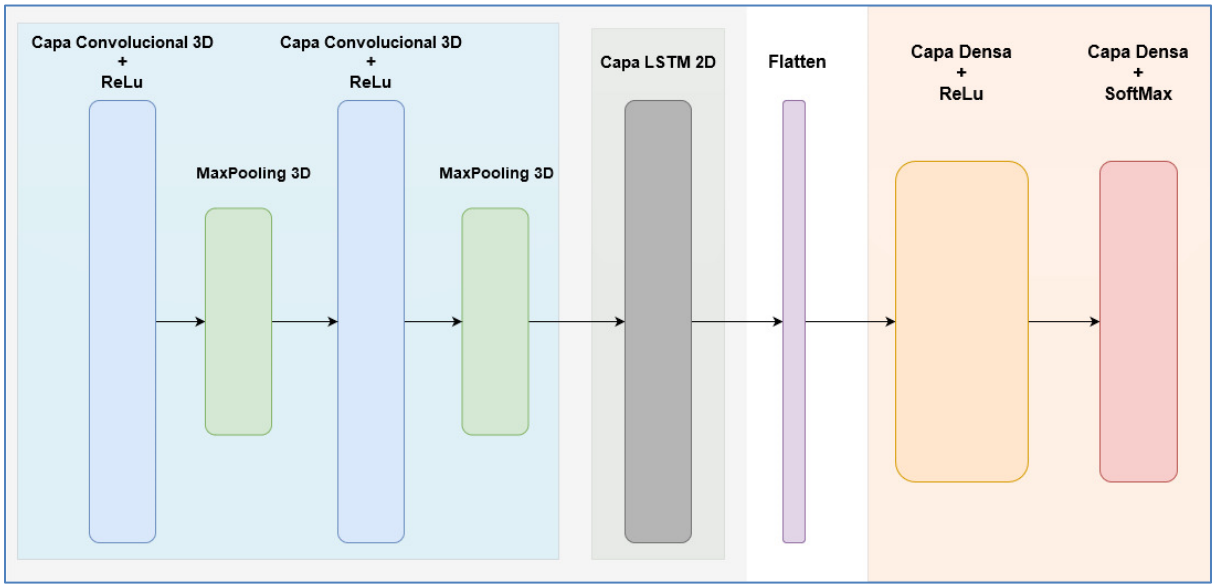

Ilustración 44. Primera arquitectura propuesta.

Antes de codificar nada, necesitamos importar las librerías de apoyo (ilustración 45).

```
import numpy as np
import cv2
import tensorflow as tf
import os
import math
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

Ilustración 45. Importación librerías de apoyo.

El código en el que implementamos la arquitectura es el siguiente (ilustración 46):

```python
# MI MODELO
class HandGestureReconModel(tf.keras.Model):
  def __init__(self):
    super(HandGestureReconModel, self).__init__()
    # Convoluciones
    self.conv1 = tf.compat.v2.keras.layers.Conv3D(32, (3, 3, 3), activation='relu', name="conv1", data_format='channels_last')
    self.pool1 = tf.keras.layers.MaxPool3D(pool_size=(2, 2, 2), data_format='channels_last')   #maxpooling
    self.conv2 = tf.compat.v2.keras.layers.Conv3D(64, (3, 3, 3), activation='relu', name="conv1", data_format='channels_last')
    self.pool2 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')

    # LSTM
    self.convLSTM =tf.keras.layers.ConvLSTM2D(40, (3, 3))

    #Flatten
    self.flatten =  tf.keras.layers.Flatten(name="flatten")

    # Capas densas
    self.d1 = tf.keras.layers.Dense(128, activation='relu', name="d1")
    self.out = tf.keras.layers.Dense(4, activation='softmax', name="output")

    #La última capa tiene que tener tantas neuronas como clases


  def call(self, x):
    x = self.conv1(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.pool2(x)
    x = self.convLSTM(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.out(x)
```

Ilustración 46. Clase "HandGestureReconModel".

Lo que hemos hecho en estas líneas es crear una clase de Python llamada “HandGestureReconModel” que será nuestro modelo, nuestra red neuronal artificial, y que heredará las funcionalidades de la clase “tf.Keras.Model”. Esto es necesario para trabajar con las capas del conjunto “Keras” más “TensorFow”.

La primera parte de la clase es la inicialización, el método “__init__(self)”. Este método se llama automáticamente cada vez que creemos un objeto de esta clase, y en él declaramos las capas de nuestra red neuronal.

```python
# MI MODELO
class HandGestureReconModel(tf.keras.Model):
  def __init__(self):
    super(HandGestureReconModel, self).__init__()
```

Ilustración 47. Inicialización.

```python
# Convoluciones
self.conv1 = tf.compat.v2.keras.layers.Conv3D(32, (3, 3, 3), activation='relu', name="conv1", data_format='channels_last')
self.pool1 = tf.keras.layers.MaxPool3D(pool_size=(2, 2, 2), data_format='channels_last')   #maxpooling
self.conv2 = tf.compat.v2.keras.layers.Conv3D(64, (3, 3, 3), activation='relu', name="conv1", data_format='channels_last')
self.pool2 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')
```

Ilustración 48. Convoluciones.

Usando Keras, para añadir una capa convolucional 3D llamamos a “tf.compat.v2.Keras.layers.Conv3D()”. El primer argumento que pasamos a esta función es el número de filtros que queremos que tenga la capa convolucional. Después pasaremos una lista de 3 enteros entre paréntesis, que serán la profundidad, altura y ancho de nuestra

ventana de convolución 3D. También pasaremos qué tipo de función de activación queremos (en nuestro caso, activation = 'relu') y el orden de las dimensiones de nuestro input a la red (data_format='channels_last'). Todo esto lo podemos ver en la ilustración 48.

Para añadir las capas de MaxPooling, usaremos "tf.Keras.layers.MaxPool3D()", cuyo primer argumento será el tamaño del pooling (pool_size) y después el orden de las dimensiones.

```
# LSTM
self.convLSTM =tf.keras.layers.ConvLSTM2D(40, (3, 3))
```

Ilustración 49. Capa LSTM.

A la hora de añadir la capa LSTM, usaremos "tf.Keras.layers.ConvLSTM2D()". Es similar a una capa LSTM, pero posee varias funcionalidades convolucionales. El primer argumento que le pasamos es un entero, que determina la profundidad a la salida, o el número de filtros en la convolución (ilustración 49).

Después añadiremos la capa de "aplanamiento", con "tf.Keras.layers.Flatten()".

```
#Flatten
self.flatten =  tf.keras.layers.Flatten(name="flatten")
```

Ilustración 50. Capa Flatten.

Y por último, las dos capas densas "fully connected" del final (ilustración 51).

```
# Capas densas
self.d1 = tf.keras.layers.Dense(128, activation='relu', name="d1")
self.out = tf.keras.layers.Dense(4, activation='softmax', name="output")
```

Ilustración 51. Capas densas.

Esto lo haremos con "tf.Keras.layers.Dense()", siendo el primer argumento el número de neuronas y el segundo la función de activación.

Ya sólo quedaría establecer el orden de las capas, que lo haremos creando otro método como se puede ver en la ilustración 52.

```
def call(self, x):
  x = self.conv1(x)
  x = self.pool1(x)
  x = self.conv2(x)
  x = self.pool2(x)
  x = self.convLSTM(x)
  x = self.flatten(x)
  x = self.d1(x)
  return self.out(x)
```

Ilustración 52. Orden de capas.

Este método se llamará automáticamente cada vez que instanciemos un objeto la clase de nuestro modelo.

8.3.2 Primer entrenamiento

Para nuestro primer entrenamiento, nuestros gestos o clases serán sólo 4: “Swiping Left” (deslizar a la izquierda), “Sliding Two Fingers Right” (deslizando dos dedos hacia la derecha), “Thumb Up” (pulgar arriba) y “No Gesture” (sin gesto). Son 3 gestos sencillos más uno adicional, “No gesture”, que es necesario enseñárselo a la red para que cuando el usuario esté quieto lo reconozca como tal y no intente reconocerlo como, por ejemplo, que está con su pulgar hacia arriba. Porque si la red no sabe qué es estar quieto, intentará decirnos que es cualquier otra cosa de las que sí sabe.

Hay que añadir que no son los gestos finales deseados, pero de momento nos servirán para ir probando y mejorando la red neuronal artificial.

```
targets_name = [
     'Swiping Left', 'No gesture', 'Sliding Two Fingers Right', 'Thumb Up'
    ]
```

Ilustración 53. Etiquetas de los gestos manuales.

En este primer entrenamiento, se han usado 1000 muestras (1000 vídeos) de cada clase, haciendo un total de 1000 muestras * 4 clases = 4000 muestras.

Lo primero que tenemos que hacer para entrenar la red, una vez creada la arquitectura, es crear un objeto de nuestra clase HandGestureReconModel().

```
modelo = HandGestureReconModel()     #creamos el modelo
```

Ilustración 54. Creación del modelo.

Después usamos el método “compile()” para configurar el modelo con las métricas que queremos medir, el optimizador a usar, etc. El algoritmo del “descenso del gradiente” sería el algoritmo de optimización más simple, pero existen otros mucho mejores y más avanzados. En cuanto a reconocimiento de imágenes, el algoritmo “Adam” es el más usado por sus buenos resultados. [43] [44] Básicamente, Adam es el algoritmo del descenso del gradiente pero con modificaciones, como por ejemplo, un learning rate variable. Por todo esto, usaremos Adam.

```
# compilamos
modelo.compile(loss='sparse_categorical_crossentropy',
               optimizer=tf.keras.optimizers.Adam(), metrics = ['accuracy'])
```

Ilustración 55. Compilación del modelo.

Este optimizador busca minimizar el error, que en nuestro caso será "sparse_categorical_crossentropy" por ser el más común en problemas de clasificación, y el que mejores resultados da. [45] Lo que hace sparse categorical crossentropy es comparar la etiqueta predicha con la etiqueta real y calcular el error o pérdidas.

El último argumento del método, "metrics", simplemente se usa para juzgar nosotros mismos cómo se está entrenando la red, pero la función que se use en este argumento no introducirá ningún cambio en la red. Es, por tanto, simplemente una métrica para la valoración del programador. Usamos "accuracy" (exactitud) para saber lo bien o mal que nuestra red es capad de reconocer los gestos mostrados.

Con los datos ya preprocesados anteriormente, y la arquitectura de la red neuronal creada y compilada, ya estamos listos para comenzar el entrenamiento (ilustración 56).

```
modelo.fit(x_train, y_train,
                    validation_data=(x_val, y_val),
                    batch_size=32,
                    epochs=7)
```

Ilustración 56. Entrenamiento.

Esto lo hacemos con el método "fit", cuyos dos primeros argumentos son los arrays NumPy con nuestros datos de entrenamiento y sus etiquetas (x_train, y_train). El siguiente argumento serían los datos de validación (validation_data=(x_val, y_val)), después vendría el tamaño del batch, y por último el número de ciclos.

Para guardar el modelo una vez entrenado, y así poder usarlo para nuestras tareas, usaríamos lo que aparece en la ilustración 57.

```
# guardamos los pesos de la red neuronal, es decir, guardamos el modelo
modelo.save_weights('weights/path_to_my_weights3', save_format='tf')
```

Ilustración 57. Guardado del modelo.

En este primer entrenamiento se han usado, por tanto, 4000 muestras (1000 por cada gesto), un tamaño de batch de 32 y 7 ciclos. Durante el entrenamiento, nos van apareciendo por consola ciertos mensajes para que evaluemos cómo se está entrenando la red.

Epoch 00005: saving model to training_today/cp-0005.ckpt

8000/8000 [==============================] - 119s 12ms/sample - loss: 0.4089 - sparse_categorical_accuracy: 0.8623 - val_loss: 0.5266

Por ejemplo, podemos ver en qué ciclo nos encontramos, cuántas muestras han pasado ya por la red con respecto al total de muestras, cuánto tarda cada muestra en pasar por la red, las pérdidas, etc. En lo que nos tenemos que fijar sobre todo es en las pérdidas (loss) y en las pérdidas de validación (val_loss). Ya se ha explicado que el overfitting es un fenómeno

que se produce cuando las pérdidas de validación son mayores que las pérdidas de entrenamiento, y empiezan a divergir entre sí. Por esto, podemos observar que en el ejemplo parece estar produciéndose overfitting, dado que:

loss: 0.4089 < val_loss: 0.5266

De todas formas, estos parámetros numéricos no son siempre los más adecuados para, en nuestro problema, comprobar la utilidad del sistema. Cuando tenemos varios gestos a clasificar, la red neuronal asigna un porcentaje de probabilidad a cada gesto, siendo el mayor de esos porcentajes el gesto predicho. Esto crea situaciones en las que, aun prediciendo correctamente el gesto con, por ejemplo, un 20% de probabilidad, existan otros con un 19%, 17%, etc., así hasta completar el 100% del total. Esto provoca que aun habiendo predicho correctamente el gesto, no lo ha distinguido con gran claridad del resto, haciendo que el error "loss" sea bastante grande. Llegamos a una situación en la que nuestra red funciona correctamente (predice el gesto correcto), pero el valor numérico "loss" parece decirnos lo contrario, al ser relativamente grande. Aun así, al principio observaremos estos parámetros, sobre todo para hacernos una idea de cómo va el entrenamiento, mediante gráficos como el de la ilustración 58. Cuando veamos que nuestro "loss" sigue siendo grande pero empieza a bajar, podremos testear la red neuronal y comprobar cómo de buena es realmente.

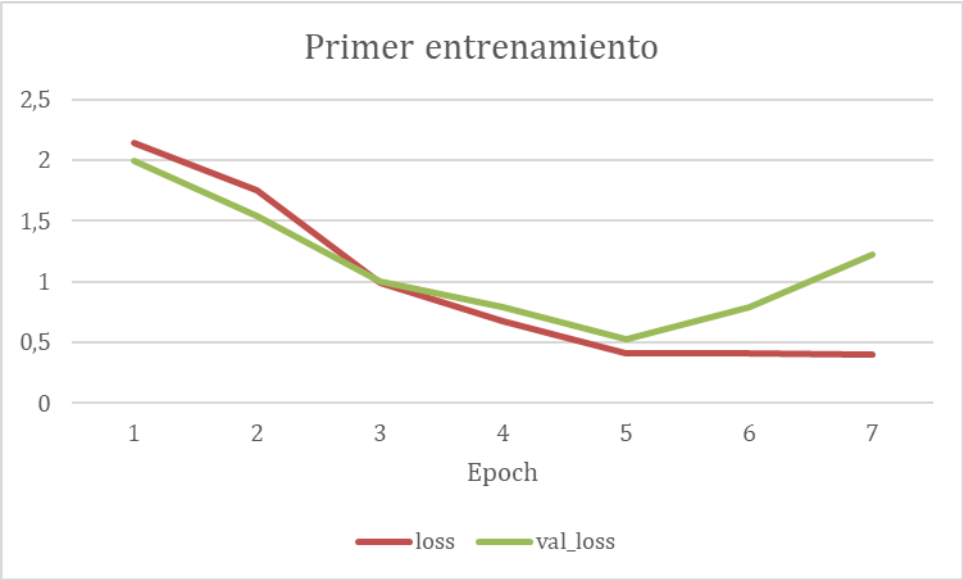

Ilustración 58. Primer entrenamiento.

Podemos ver cómo a partir del quinto ciclo "loss" y "val_loss" comienzan a divergir, lo que nos indica overfitting. Además, testeando la red neuronal de forma manual tampoco vemos resultados positivos.

8.3.3 Modificaciones de parámetros: tuning

A la vista de los resultados negativos que hemos observado, en este momento se propone la modificación de diferentes parámetros, lo que se conoce como "tuning". Lo que se suele

cambiar primeramente al ocurrir el fenómeno del overfitting, es el learning rate. Si tenemos un learning rate muy pequeño, es probable que la red neuronal esté avanzando muy poco a poco memorizando (y no aprendiendo) cada detalle, reduciendo al mínimo el error, sin aprender a generalizar. Por ello, un learning rate más grande, le permitiría más flexibilidad y mayor capacidad de generalización.

Teniendo esto en cuenta, se propone cambiar el learning rate por defecto (lr = 0.001) a otros valores, como por ejemplo lr=0.005 o lr=0.01.

En la ilustración 59 vemos los resultados.

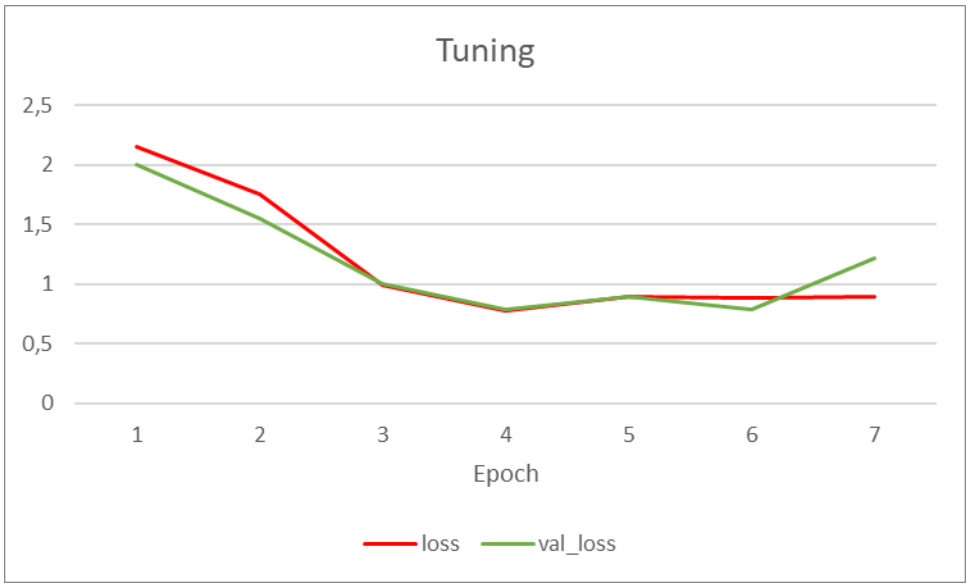

Ilustración 59. Resultados después de tuning.

Estos resultados aún peores tienen su razón en la naturaleza del algoritmo Adam. La mayoría de algoritmos de optimización más sencillos, como el descenso del gradiente, o el descenso del gradiente estocástico, usan un learning rate fijo e invariable durante todo el entrenamiento. En cambio, Adam va adaptando el learning rate durante el entrenamiento, usando una serie de parámetros dinámicos internos. Esto hace que parta de un learning rate inicial por defecto, lr=0.001, pero a medida que transcurre el entrenamiento, va deslizándose hacia otros valores más adecuados en cada momento. Por ello, si nosotros modificamos este parámetro inicial (el cual es el más óptimo para este algoritmo), y además lo hacemos fijo, perderemos la eficiencia y la total funcionalidad de este algoritmo, haciéndolo fracasar. [44]

Lo siguiente que se proponía modificar era el número de ciclos, el tamaño del batch y el número de muestras de entrenamiento. Al cabo de numerosas iteraciones y cambios a estos dos primeros parámetros, no se obtenían resultados muy diferentes. Por otro lado, modificar el número de muestras de entrenamiento suponía aumentar en gran medida las mismas, rozando los límites computacionales de nuestro equipo y sin mayores resultados nuevamente.

8.3.4 Capas Dropout

Al agotar los posibles cambios paramétricos posibles, la siguiente propuesta es la de modificar la arquitectura de la red añadiendo capas “Dropout”. Sería un cambio que sólo afectaría a la red a la hora de su entrenamiento, pero no a la hora de su uso. Es decir, las capas Dropout sólo afectarían al proceso de entrenamiento. Lo que hacen estas capas es apagar o encender ciertas neuronas de la red, de forma aleatoria, para de alguna forma “poner trabas” a la misma a la hora de su entrenamiento. Así, la red neuronal buscará en cada momento caminos alternativos distintos para un mismo problema, aprendiendo a generalizar y evitando que “memorice” un solo camino. [46]

Teniendo esto en cuenta, se propone añadir una capa dropout en la siguiente configuración que se muestra en la ilustración 60.

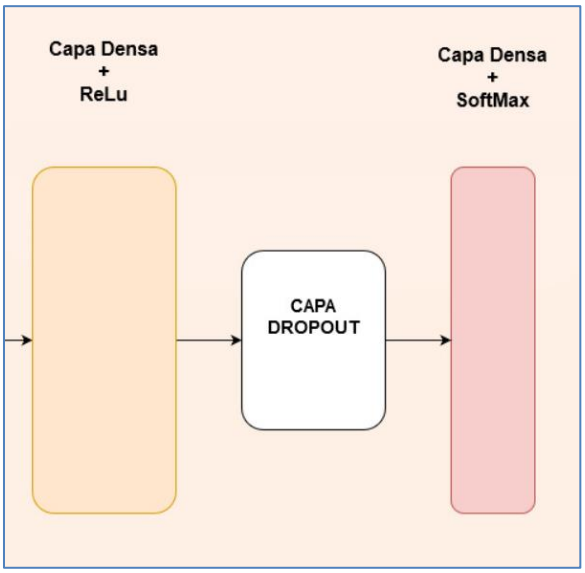

Ilustración 60. Capa Dropout.

Esta capa Dropout iría después de la primera capa densa de 128 neuronas, provocando el efecto deseado en esa capa.

```
#Añado dropout
self.drop_out = tf.keras.layers.Dropout(rate=0.5, name='drop_out')
```

Ilustración 61. Función capa Dropout.

El primer parámetro sería el ratio del dropout, que sería un número entre 0 y 1, e indicaría la fracción de las neuronas a apagar. Es decir, con un ratio igual a 0.5, haríamos que a cada momento la mitad de las neuronas de la capa anterior estén apagadas de forma aleatoria.

Una vez hecho este cambio, seguimos con nuestro entrenamiento, obteniendo los siguientes resultados (ilustración 62).

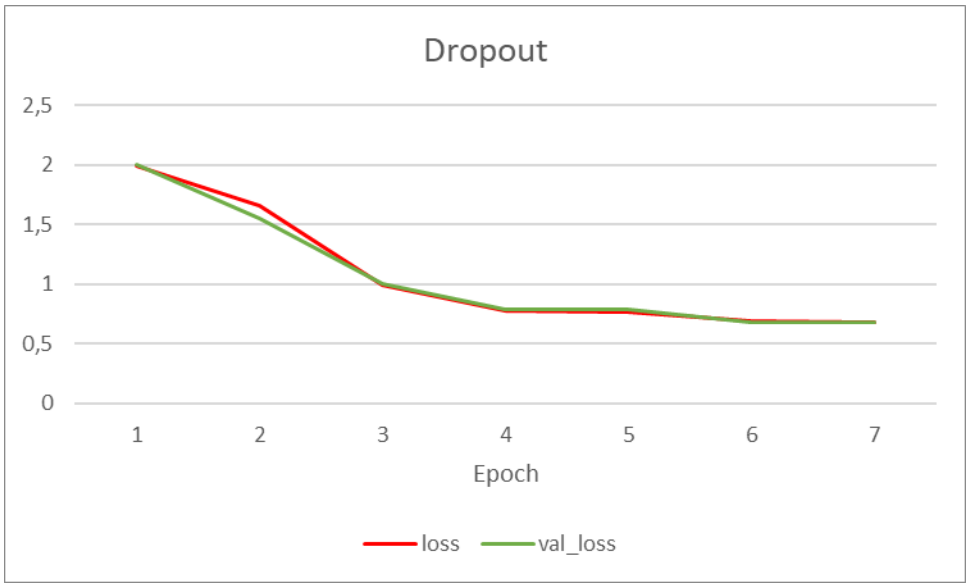

Ilustración 62. Resultados después de dropout.

Podemos comprobar que ya no se produce overfitting, aunque los resultados tampoco siguen siendo demasiado óptimos, dado que las pérdidas siguen estando lejanas a cero.

8.3.5 Capas Spatial Dropout y nuevos gestos manuales

Tras la naturaleza de los resultados obtenidos, se proponen dos cambios adicionales. El primero, sería añadir más capas dropout, pero en este caso “espaciales”. Estas serían como las capas dropout normales, pero aplicadas a las capas convolucionales. En vez de apagar y encender neuronas individuales de forma aleatoria, actúa sobre los mapas de características 3D enteros de las capas convolucionales. [47]
Esto reduciría aún más el overfitting, al repartir el dropout por toda la red, y no sólo al final de la misma. En código, se implementaría así (ilustración 63):

```
self.conv1 = tf.compat.v2.keras.layers.Conv3D(32, (3, 3, 3), activation='relu', name="conv1", data_format='channels_last')
self.sp_drop3D= tf.keras.layers.SpatialDropout3D(0.2,data_format='channels_last')
```

Ilustración 63. Capa Spatial Dropout.

Vemos cómo después de una capa convolucional implementamos la capa de Keras “tf.Keras.layers.SpatialDropout3D”. Los argumentos serían los mismos que en una capa dropout normal, siendo el primero el “rate” o ratio al que se produce el efecto “dropout”. En este caso, se ha añadido una primera capa con un rate de 0.2, para después añadir una segunda capa con rate 0.3, que se aplicaría a la segunda capa convolucional. Así, tendríamos el dropout repartido de forma progresiva.

Otro cambio que se propone en este momento es el de cambiar los gestos a reconocer por la red neuronal artificial. Obviamente, la arquitectura final reconocerá un mayor número de gestos que los que en este momento está aprendiendo. Pero también es cierto que para que llegue ese momento, es necesario comenzar con ciertos resultados positivos, aunque sea con un menor número de gestos, y a partir de ahí, ir aumentando las capacidades de la misma hasta llegar a los gestos deseados.

Por ello, se plantea cambiar los gestos a reconocer por la red neuronal en este momento, para pasar a unos mucho mejor diferenciados entre sí, y que intuitivamente serán mucho más sencillos de reconocer. Los gestos propuestos ahora serían: “swiping left”, “swiping right”, “swiping up”, “swiping down” (deslizar a la izquierda, deslizar a la derecha, deslizar arriba y deslizar abajo, respectivamente). Estos gestos, al ser movimientos claros en direcciones contrapuestas, serán mucho más fáciles de reconocer por la red neuronal, lo que nos permitirá hacer progresos.

```
targets_name = [
    'Swiping Up', 'Swiping Down', 'Swiping Right', 'Swiping Left', 'No gesture'
    ]
```

Ilustración 64. Etiquetas de los gestos manuales.

Como en el caso anterior, también debemos añadir el gesto “No gesture”, literalmente “ningún gesto”, para que la red sepa cuándo estamos realizando ningún gesto y no intente reconocer algo que no hay.

En resumen, tendríamos dos capas dropout espaciales, una capa dropout normal, y 5 gestos nuevos. La nueva arquitectura se muestra en la ilustración 65.

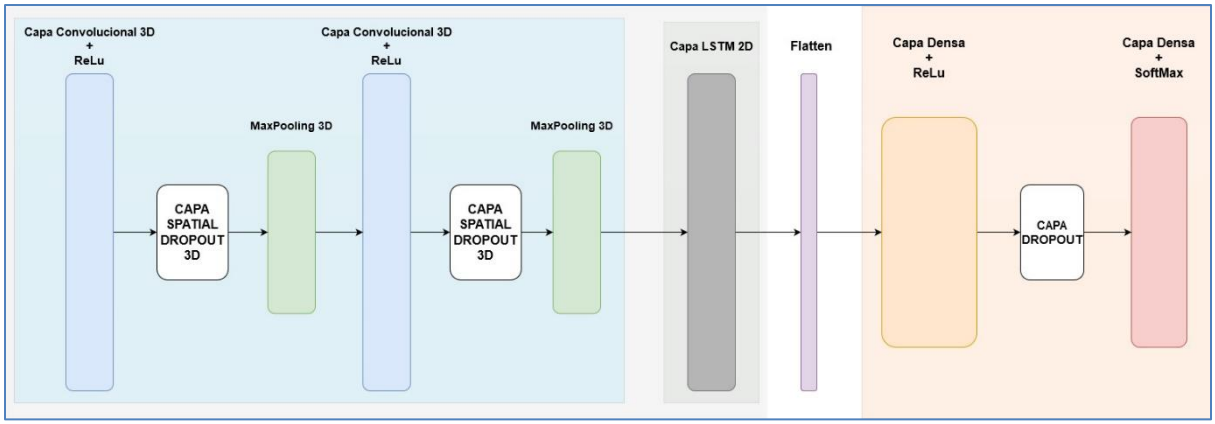

Ilustración 65. Arquitectura actual.

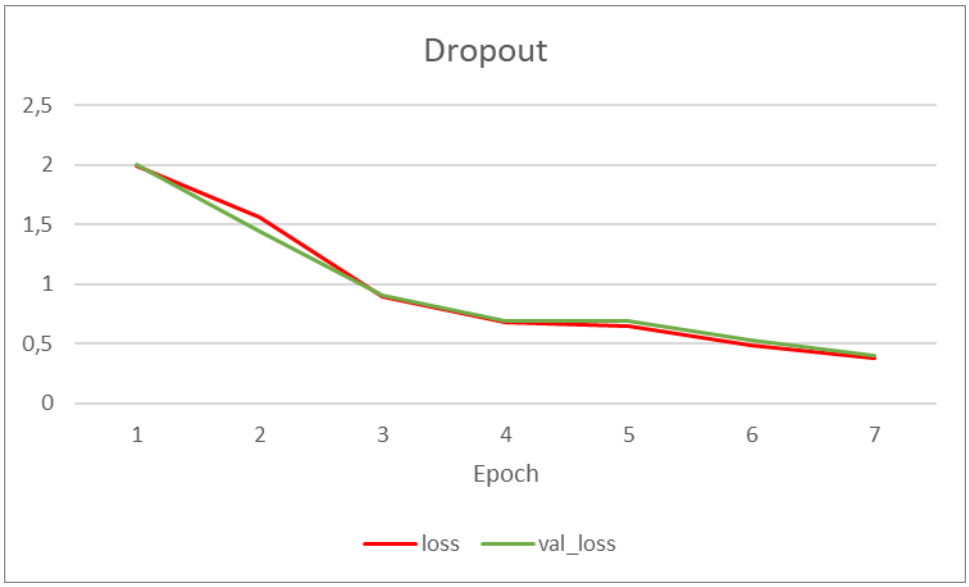

Ilustración 66. Resultados tras SpatialDropout.

Como vemos en la ilustración 66, obtenemos aún mejores resultados que en los anteriores casos (error y error de validación por debajo de 0.5), además de empezar a tener resultados positivos al testear de forma manual la red neuronal.

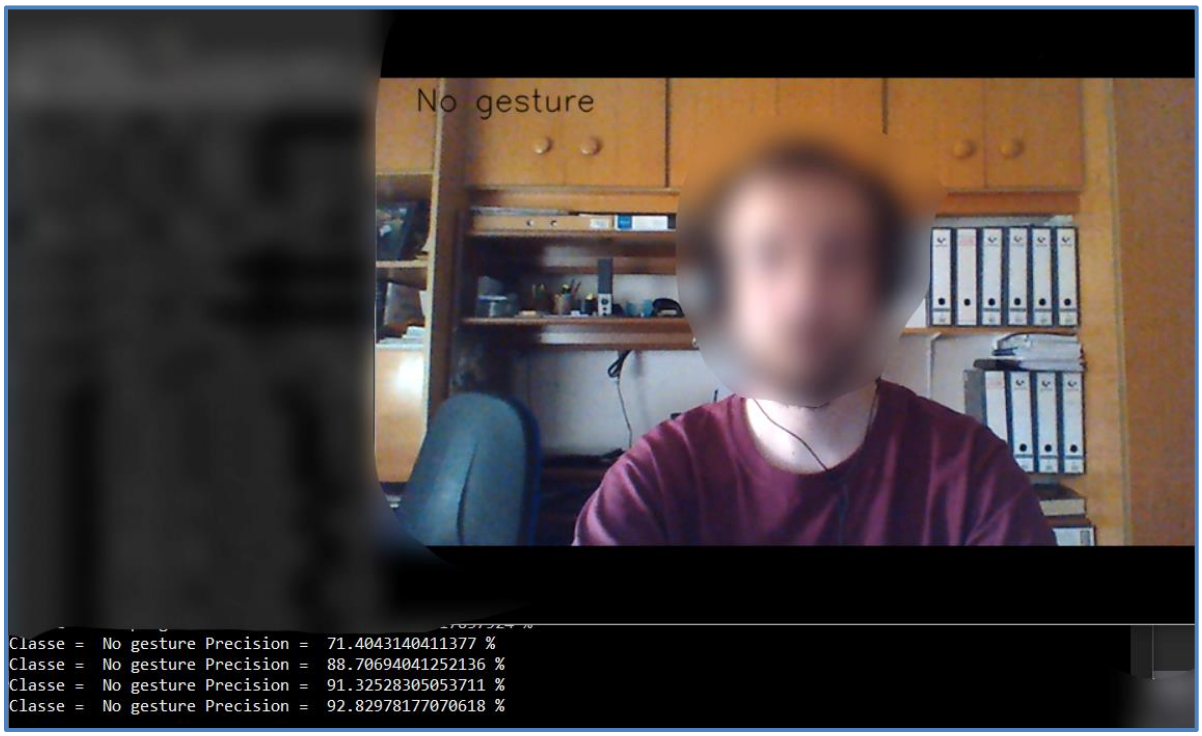

Ilustración 67. Gesto "No gesture". Precisión: 92.83%

En la ilustración 67, podemos ver a la red neuronal detectando correctamente que el usuario está quieto, con una precisión elevada (92,83%).

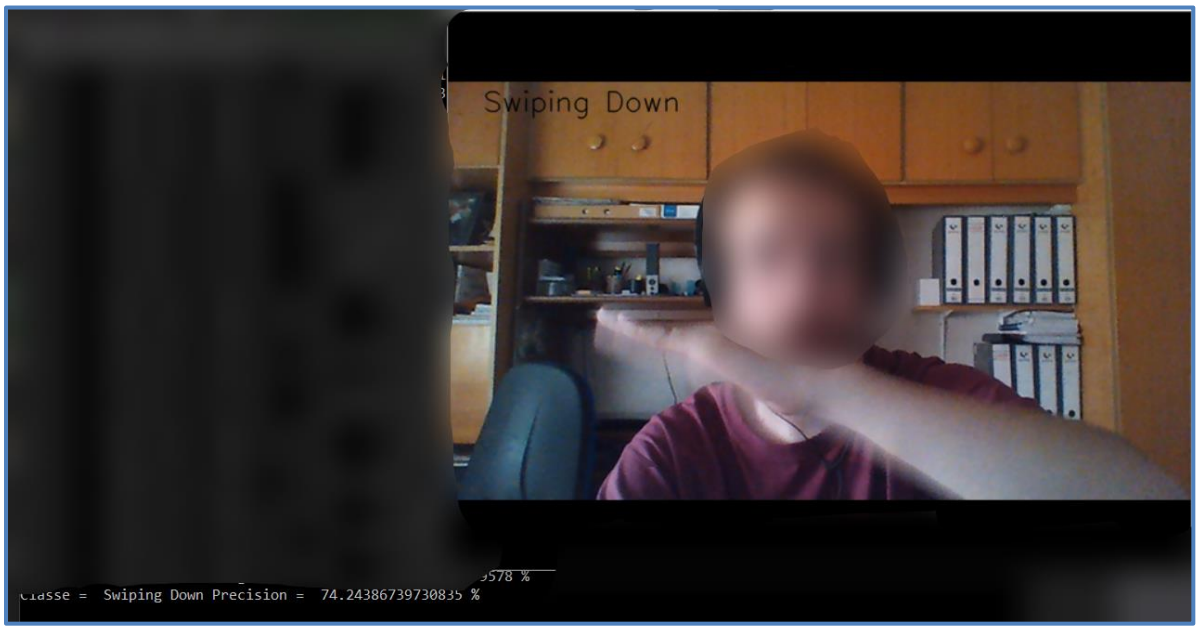

Ilustración 68. Gesto "Swiping down".

En la ilustración 68, comprobamos que la red también reconoce el gesto “Swiping down”, que consiste en desplazar la mano hacia abajo. En las ilustraciones 69, 70 y 71, vemos también cómo la red neuronal reconoce correctamente el resto de gestos propuestos hasta el momento: desplazamiento hacia la izquierda, derecha y arriba. También podemos ver la precisión del reconocimiento en la esquina inferior izquierda de cada imagen.

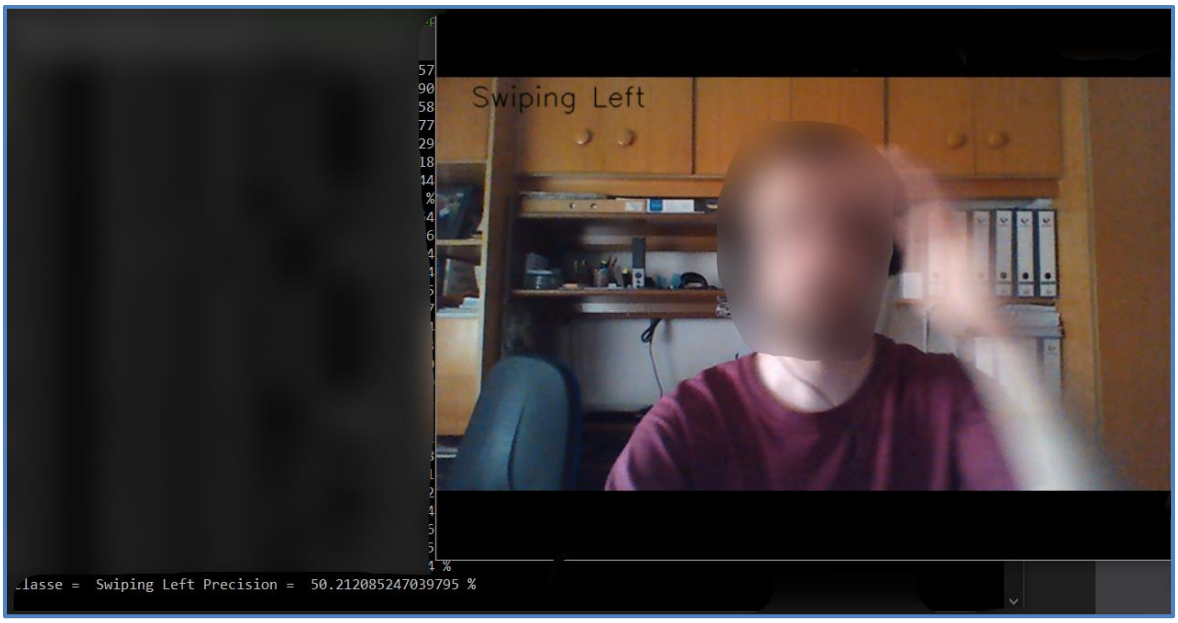

Ilustración 69. Gesto "Swiping left".

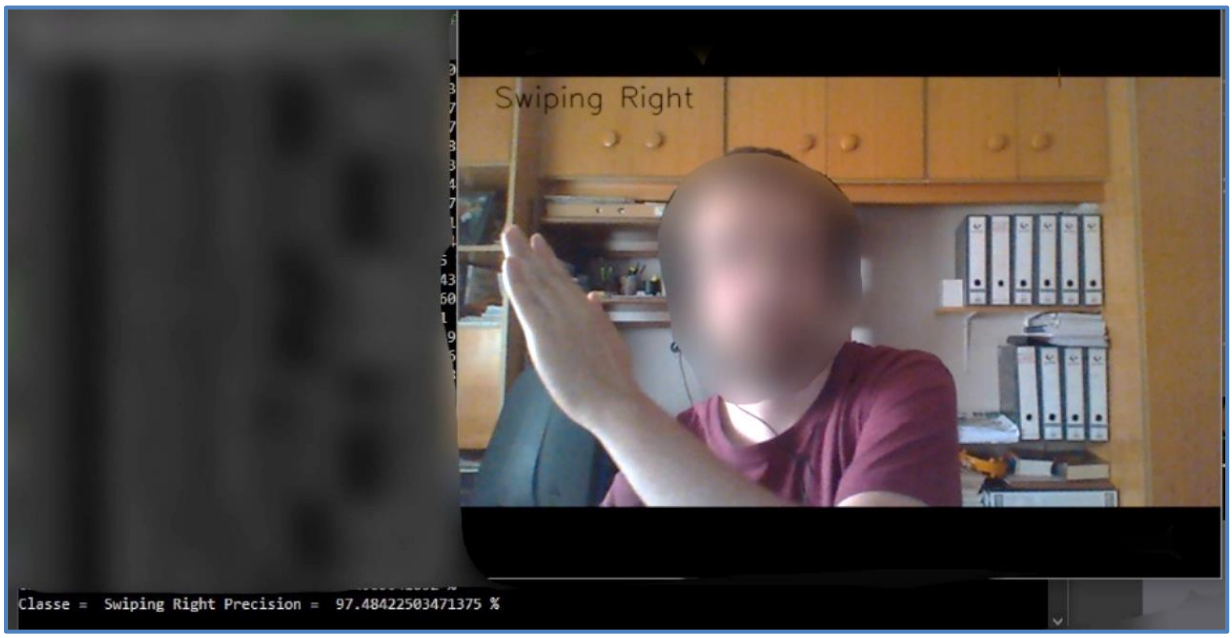

Ilustración 70. Gesto "Swiping right".

Ilustración 71. Gesto "Swiping up".

8.3.6 Batch Training

Durante todos estos pasos, nos hemos visto limitados por la cantidad de memoria de nuestro ordenador, o más correctamente, de nuestra GPU. Hemos usado como mucho 5 gestos, por un total de 1000-1500 muestras por gesto, lo que hace un máximo de 7500 muestras totales. Cada gesto, es decir, cada vídeo repartido en imágenes individuales, de media pesa un total de 150 KB. Si son 7500 vídeos, 7500 * 150 KB = 1.125 GB, a los que habría que sumar también los datos de validación, que normalmente hemos usado unas 200 muestras, haciendo un total de 1.125 GB + 200*5*150KB = 1.275 GB. Si quisiéramos, por poner un

ejemplo, pasar a 7 gestos distintos con 2500 muestras cada uno, y 400 muestras de validación, pasamos a tener 7*2500*150 KB + 7*400*150 KB = 3.045 GB.

```
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 3181130547
locality {
  bus_id: 1
  links {
  }
}
incarnation: 14359617424826487195
physical_device_desc: "device: 0, name: GeForce GTX 950M, pci bus id: 0000:01:00.0, compute capability: 5.0"
]
```

Ilustración 72. Capacidad GPU.

Resulta que nuestro límite de memoria son 3.181 GB, por lo que nos quedamos muy justos de memoria y no podremos avanzar mucho más allá. Por ello, nos vemos obligados a encontrar una solución que resuelva esta limitación.

Ésta la encontramos en lo que se conoce como “Batch Training”. Este método de entrenamiento consiste en pasar a la red neuronal el data set entero, pero cargando en memoria pequeñas porciones. Concretamente, el tamaño de estas porciones coincide con el tamaño del batch, así que pasamos a la memoria el data set batch a batch. [48]

A primera vista, esto parece no suponer un gran cambio. Sin embargo, al poder pasar poco a poco nuestro data set, ya no dependemos de la cantidad de memoria disponible, siempre que tengamos espacio para el batch. Así, podremos elegir un data set del tamaño que queramos, y nuestra red neuronal aprenderá de muchos más ejemplos que antes, mejorando su entrenamiento y, por tanto, sus capacidades. Podremos elegir datasets de 2000 imágenes, 20.000, 2.000.000, etc. Ahora, la única limitación será el tiempo que tardemos en procesar todo.

Todo esto se puede implementar en Python haciendo uso de lo que se conoce como “generadores”. Estos serían funciones que se comportan como iteradores, que sólo actúan cuando son llamados. Por ejemplo, en nuestro caso, podríamos tener un generador que va iterando sobre todo el data set, y que nos fuera dando cada vez un fragmento del mismo, es decir, un batch. De hecho, podríamos extender este concepto al preprocesamiento de las imágenes, para no tener que esperar a la total finalización del mismo cada vez que queramos entrenar la red neuronal, y simplemente preprocesar cada batch, pasarlo a la memoria, y después entrenar la red con ese batch.

Para esto último, modificamos nuestro código y creamos un generador que preprocesa un batch de 25 muestras, es decir, de tamaño de batch igual a 25.

```
def proc_train_generator():
    counter_training = 0 # contador de muestras totales
    training_targets = [] # lista de clases (gestos)
    new_frames = [] # lista para almacenar los frames
    pos=0
    contador=0 # contador de muestras para nuestro generador

    for target_key in targets[0]:   #targets es un diccionario: en 0 tiene una lista de target_keys y en 1 tiene una lista de target
        #los target_keys son las referencias de cada video
        #los targets son los videos
        #por ello, vamos recorriendo cada video en el for
        pos+=1

        print("Adjusting "+str(pos)+"/"+ str(len_dirs) +" target: "+str(targets[0][target_key]))
        new_frame = [] # nueva muestra
        # Frames in each folder
        directory= str(targets[0][target_key]) #pasamos a string la referencia del directorio
        frames = unificar_frames(path+directory) #unificamos el video de ese directorio (hacemos que tenga 30 frames)
        if len(frames) == max_frames: # comprobamos por si acaso

            for frame in frames: #para cada frame del video
                frame = resize_frame(path+directory+'/'+frame) #hacemos que todas tengan el mismo tamaño de frame (64*64)

                new_frame.append(rgb2gray(frame)) #pasamos a escala de grises

                if len(new_frame) == 15: # partition en dos muestras de 15 y 15

                    new_frames.append(new_frame) # las metemos en la lista
                    contador+=1 #contamos una muestra más (generador)
                    training_targets.append(targets_name.index(targets[1][target_key]))
                    #accedo al target con esa target_key, miro qué clase es, y meto el nombre de la clase en la lista de clases
                    counter_training +=1 #contamos una muestra TOTAL más
                    print("Entrenamiento "+str(contador)+". Entrenamiento actual "+str(counter_training))
                    new_frame = []  #libero la lista new_frame
                if contador == 25:   #miro si he llegado al tamaño del batch
                    contador=0    # reseteo el contador
                    # convert training data to np float32
                    training_data = np.array(new_frames[0:counter_training], dtype=np.float32)  #paso a array Numpy de 32 bits
                    release_list(new_frames)  #libero la lista new_frames

                    yield training_data, training_targets  #termina la iteración, devuelvo el batch

                    new_frames = []
                    training_targets=[]
```

Ilustración 73. Generador de preprocesamiento.

Básicamente, esta función va preprocesando muestras como antes, pero con un contador que comprueba si hemos preprocesado ya un número de muestras igual al tamaño del batch. En caso positivo, devolvería dicho batch, es decir, las muestras y sus etiquetas.

En esta función llamada proc_train_generator() estaría incluido todo el preprocesamiento, menos la normalización. También tendríamos una función llamada proc_val_generator(), que realizaría lo mismo pero para el set de validación, por lo que su explicación se obvia.

El batch preprocesado se entregaría a otro generador, que lo pasaría a la red neuronal. Este último generador se codifica de la siguiente manera (ilustración 74).

```
def train_generator(trn):
    trn_counter=0
    if trn==1 :  #entrenamiento activado
        trn_counter=56000  #muestras totales
    elif trn==0: #validación activada
        trn_counter=5600

    while True:

        if trn==1 and trn_counter==56000:
            gen=proc_train_generator() #si estamos entrenando, crear generador para procesar
                                       #datos de entrenamiento
            trn_counter=0
        elif trn==0 and trn_counter==5600:  #si estamos validando, crear generador para procesar
                                           #datos de validación
            gen=proc_val_generator()
            trn_counter=0
        training_data, training_targets = next(gen)  #siguiente batch

        # Normalización

        scaler = StandardScaler()     #creamos un objeto de la clase sklearn.preprocessing.StandardScaler
        scaled_images  = scaler.fit_transform(training_data.reshape(-1, 15*64*64))
        #transformamos a un array de una dimensión y estandarizamos
        scaled_images  = scaled_images.reshape(-1, 15, 64, 64, 1) #devolvemos la forma

        #pasamos a un array Numpy óptimo para pasarlo a la red
        x_train = np.array(scaled_images)
        y_train = np.array(training_targets)

        # damos el batch a la función de entrenamiento
        yield x_train, y_train
        trn_counter+=25 #incrementamos un tamaño de batch cada vez
```

Ilustración 74. Generador de entrenamiento.

Este generador valdría tanto para generar los batches de entrenamiento, como para los de validación. Para ello, tiene una variable de entrada “trn”, que cuando vale 1 significa que estamos entrenando, y cuando vale 0 significa que estamos validando.

Lo siguiente sería crear dos objetos usando este generador. Uno para entrenar, y otro para validar (ilustración 75).

```
train=train_generator(1)
val=train_generator(0)
```

Ilustración 75. Creación de generadores.

Finalmente, entrenamos. Lo hacemos usando fit_generator(), una modificación del método “fit” que usábamos anteriormente y que sirve para usar generadores (ilustración 76).

```
modelo.fit_generator(train, steps_per_epoch=2240, epochs = 8,
          callbacks = [cp_callback],
          validation_data=val,
         validation_steps=224)
```

Ilustración 76. Entrenamiento con generadores.

El primer argumento sería nuestro generador del batch de entrenamiento, después vendría una variable que indica cuántas veces llamaremos al generador cada ciclo (steps_per_epoch). El siguiente argumento sería el número de ciclos, terminando con otros argumentos como el generador de los datos de validación y una variable que indica cuántas veces llamamos a este generador.

A continuación, debemos explicar cómo se calcula el parámetro “steps_per_epoch”. Supongamos que tenemos un tamaño de batch igual a 25, 7 clases y 4000 muestras por clase. Al tener 4000 muestras por clase y 7 clases, tenemos 28.000 muestras. Recordemos también que dividimos cada muestra en dos, por lo que al final tendremos 56.000 muestras, el doble que antes. En un ciclo, todas esas 56.000 muestras pasarán por la red neuronal en varios pasos, que será nuestro parámetro “steps_per_epoch”. Como hemos supuesto un tamaño de batch igual a 25, todas esas 56.000 muestras pasarán por la red neuronal en pequeños conjuntos de 25 muestras en cada paso, lo que hace un total de 56.000/25 = 2240 pasos por ciclo.

Con todo esto ya tendríamos listo nuestro código para utilizar cualquier tamaño de data set, siendo ahora el tiempo de procesamiento nuestra única limitación.

8.3.7 Ampliación de gestos

Ahora que podemos usar cualquier tamaño de dataset sin relativas restricciones, se plantea usar 7 gestos manuales en vez de 5, y pasar de un tamaño de 1500 muestras por gesto a 4000 muestras por gesto. Así podremos tener una red neuronal con mayor capacidad por un lado, y que estará mejor entrenada, por otro.

Por tanto, los gestos propuestos son los 5 anteriores (“swiping left”, “swiping right”, “swiping up”, “swiping down” y “no gesture”), más el gesto “Thumb Up” (pulgar arriba) y “Doing Other Things” (haciendo otras cosas). El gesto del pulgar hacia arriba es interesante, ya que podría ser usado por sistemas de interacción humano-computadora para simbolizar un “okay” o un “acepto”. El gesto “haciendo otras cosas” se ha visto que es necesario, puesto que cuando el usuario se encuentra realizando otros gestos que la red no conoce, sin estar quieto (no gesture), ésta intenta predecir de entre lo que ya conoce. Esto resulta en que, por ejemplo, si el usuario está distraído mirando el móvil, pero no está quieto, la red intentará entender qué está haciendo el usuario, de entre lo que ha sido entrenada. Quizás estamos usando el móvil, o estirando los brazos, cosas que no conoce la red, y sin embargo la red neuronal nos dirá de forma aleatoria que estamos levantando el pulgar o deslizando la mano a la derecha. Por ello, es necesario que la red conozca los gestos que queramos y que el resto los sitúe entre “ningún gesto”, cuando el usuario esté quieto, y “haciendo otras cosas”, cuando sea cualquier otro gesto.

```
targets_name = [
    'Swiping Up', 'Swiping Down', 'Swiping Right', 'Swiping Left', 'No gesture', 'Doing other things', 'Thumb Up'
    ]
```

Ilustración 77. Etiquetas gestos manuales.

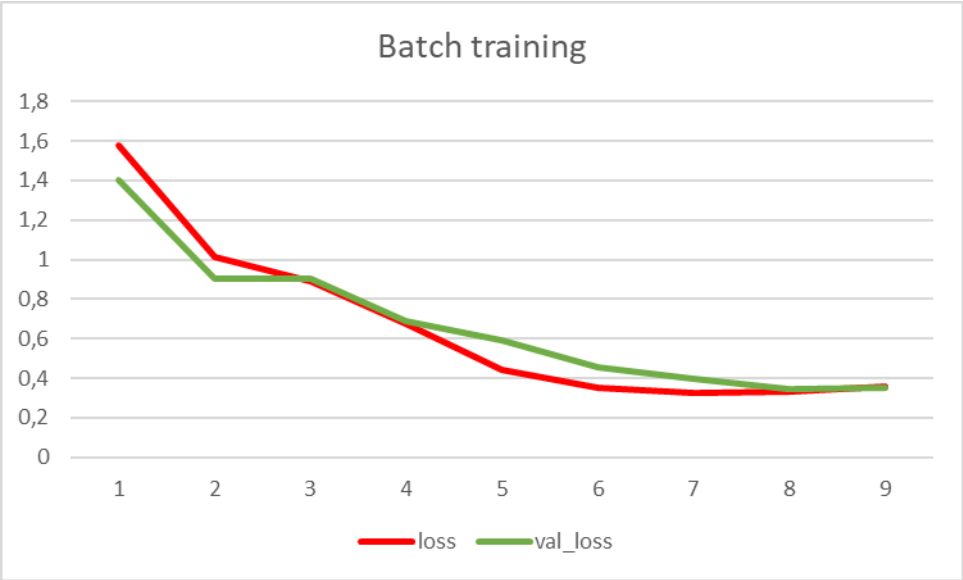

Ilustración 78. Resultados batch training.

Vemos entonces que, gracias al batch training, usar 4000 muestras en vez de 1500 permite que la red neuronal pueda generalizar mejor y el error se minimiza aún más, por debajo de 0.4. Además, hemos realizado un entrenamiento de 9 ciclos en vez de 7.

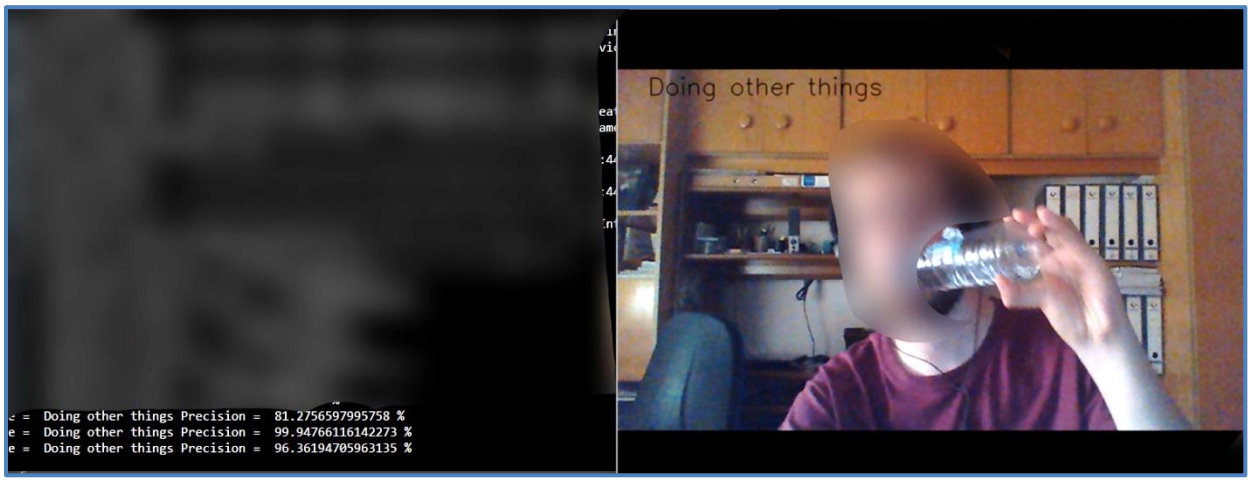

Ilustración 79. Gesto "Doing other things". El usuario está bebiendo agua.

También observamos cómo ahora la red neuronal reconoce cuándo el usuario está haciendo otras cosas (“Doing other things”). En la ilustración 79, el usuario está bebiendo agua y la red neuronal nos dice que está haciendo otras cosas, con gran precisión además (véase la esquina inferior izquierda de la imagen). En la ilustración 80, en cambio, el usuario está usando su teléfono móvil, y la red neuronal también nos dice que está haciendo otras cosas. Esto es precisamente lo que queremos.

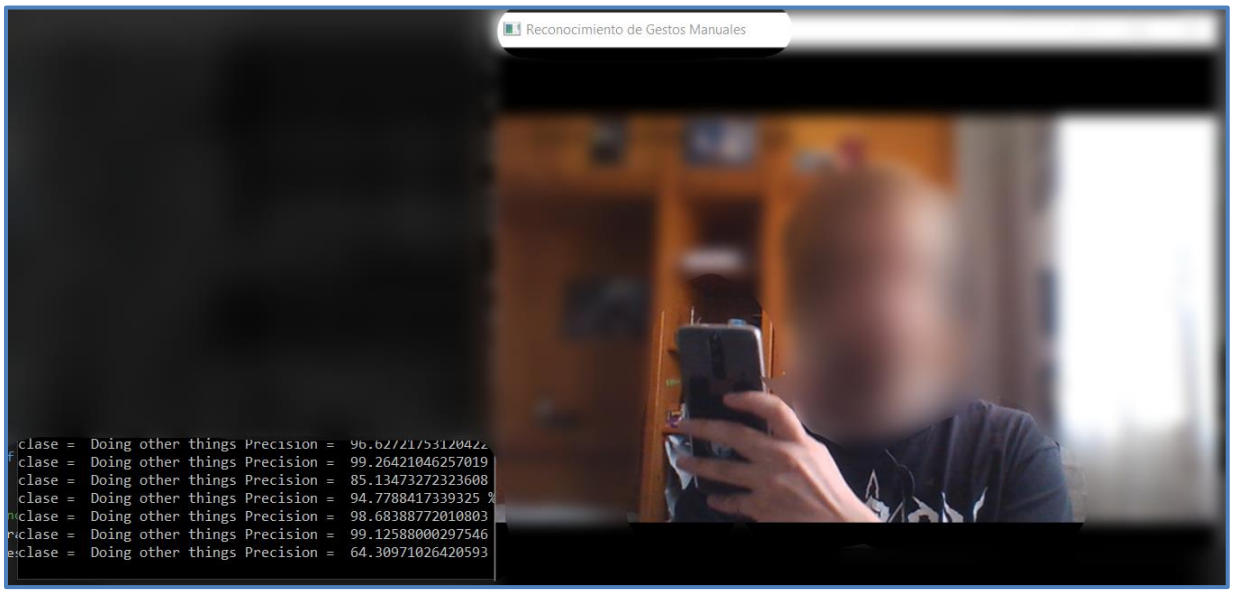

Ilustración 80. Gesto "Doing other things". El usuario está usando su teléfono.

Nótese también que, en la ilustración 80, el usuario se encuentra en otra habitación distinta, además de llevar ropa diferente, y aún así la red neuronal funciona con normalidad. Sumado esto a que la red neuronal no ha sido entrenada con ningún ejemplo en el que aparezca este usuario, o su entorno, significa que la red neuronal ha aprendido realmente qué es cada gesto; generaliza y aprende, no memoriza.

Por otro lado, en la ilustración 81 el usuario está levantando su pulgar, y la red neuronal lo reconoce como tal (“Thumb up”, con precisión del 95,05%).

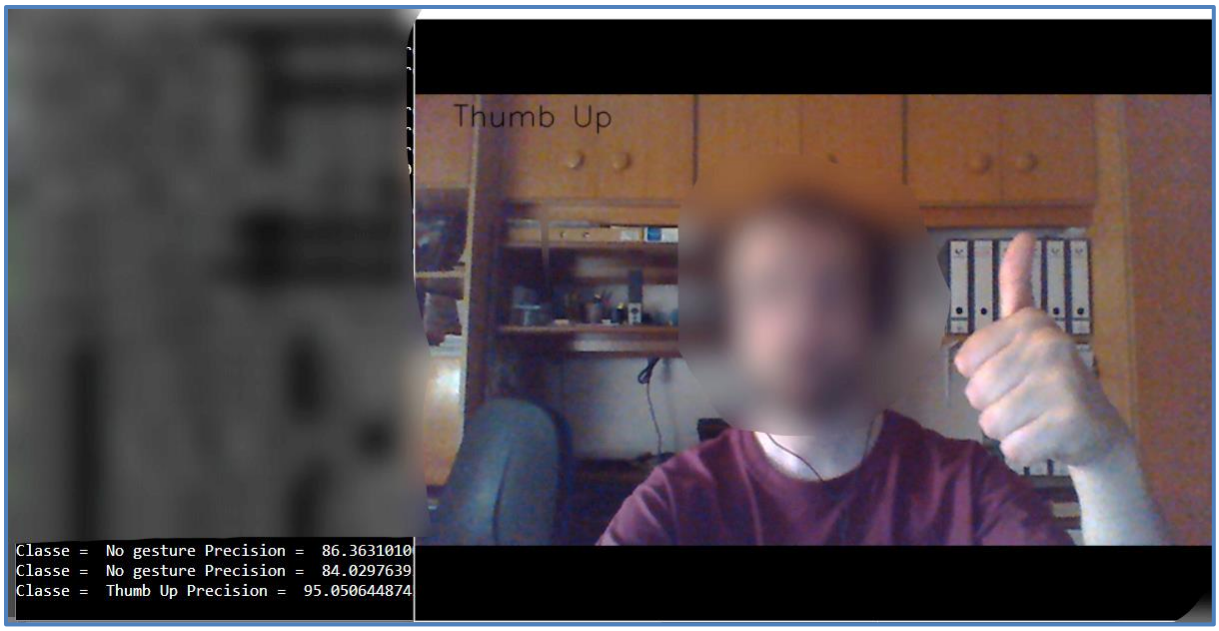

Ilustración 81. Gesto "Thumb up", pulgar arriba.

8.3.8 Arquitectura final

Dado que la arquitectura actual está dando buenos resultados, y el batch training nos da gran flexibilidad a la hora de poder usar data sets grandes, se plantea mantener la arquitectura actual ampliando aún más los gestos a reconocer por la red neuronal, hasta llegar a los deseados. Estos gestos serían los siguientes: “Swiping Left”, “Swiping Right”, “Swiping Up”, “Swiping Down” , “Thumb Up”, “Thumb Down”, “Stop Sign”, “Pulling Hand In”, “Pushing Hand Away”, “No Gesture” y “Doing Other Things”.

Los agrupábamos de la siguiente manera:

Desplazamientos
- Swiping Right (a la derecha)
- Swiping Left (a la izquierda)
- Swiping Up (arriba)
- Swiping Down (abajo)

Zoom in/out
- Pulling Hand In (mover la mano hacia dentro, zoom in)
- Pushing Hand Away (mover la mano hacia fuera, zoom out)

Parada
- Stop Sign (señal de stop)

Aprobación
- Thumb Up (pulgar arriba, aceptar)
- Thumb Down (pulgar abajo, cancelar)

Y por otro lado, los gestos adicionales necesarios para el correcto funcionamiento de la red neuronal
- No gesture (ningún gesto, usuario quieto)
- Doing Other Things (usuario haciendo otras cosas)

De esta forma, tendríamos los gestos necesarios para desplazarnos por menús, ampliar, aceptar o cancelar opciones, etc. Esto nos permitiría tener un sistema interfaz humano-computadora relativamente completo.

Tras entrenar durante 9 ciclos con 4000 muestras por gesto, obtenemos los siguientes resultados.

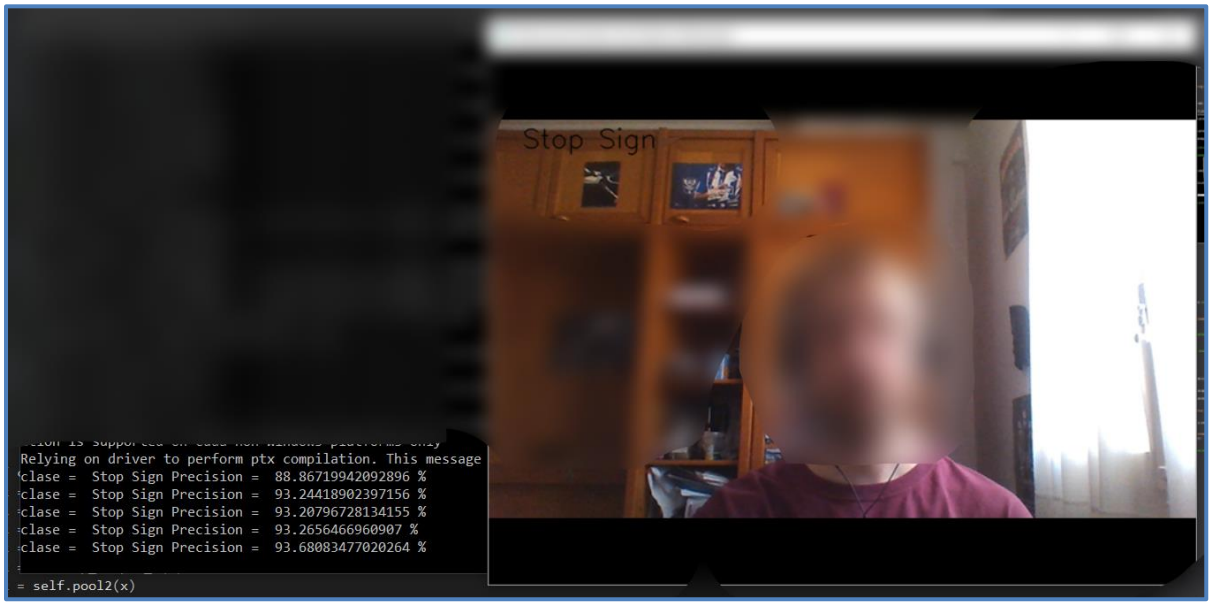

Ilustración 82. Gesto "Stop sign" reconocido, pero el sujeto está quieto.

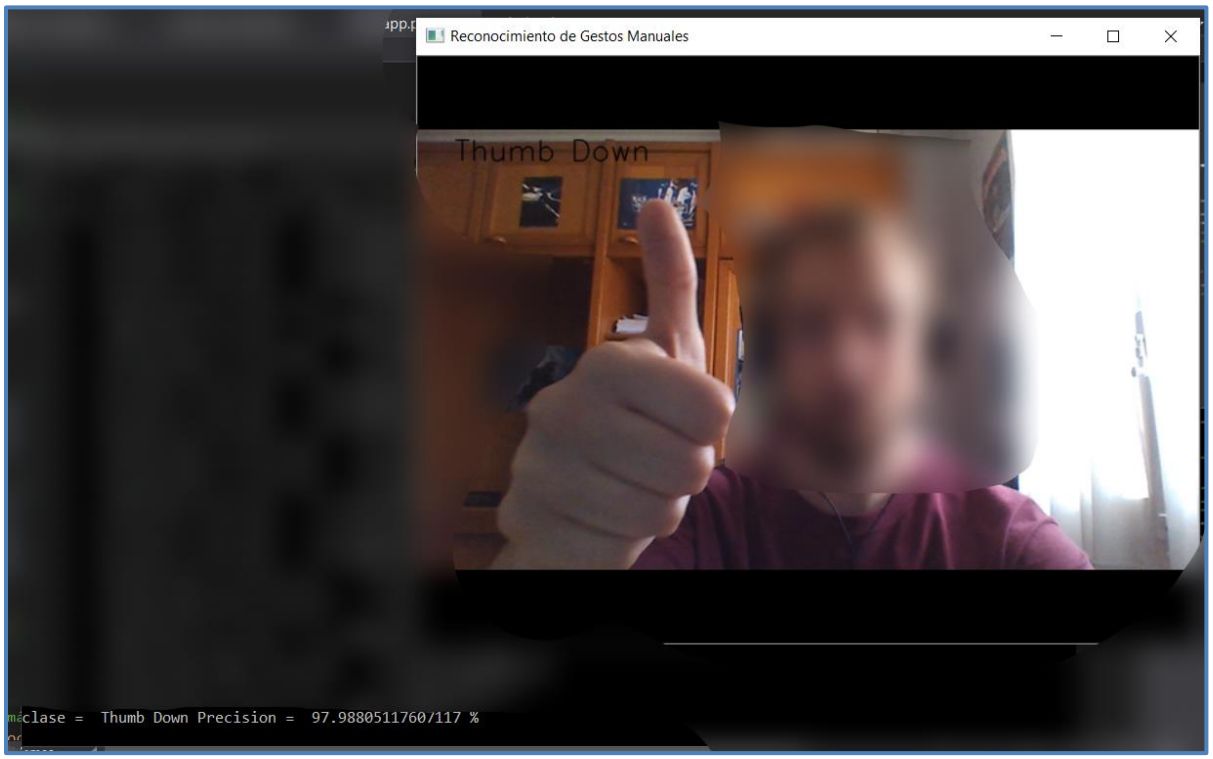

Ilustración 83. Gesto "Thumb Down" reconocido, pero el sujeto está haciendo "Thumb Up".

En las ilustraciones 82 y 83 podemos comprobar resultados erróneos, que nos sugieren que el modelo necesita más entrenamiento, además de que parece ser sensible a demasiada luminosidad. Si nos fijamos, cuando el usuario está quieto, parece que la luminosidad reflejada en la cara es reconocida falsamente como una palma abierta, confundiéndola con el gesto de la señal de stop (“Stop Sign”).

Por esto, se vuelve a entrenar al modelo durante 12 ciclos en vez de 9, y se realizan pruebas posteriores con una luminosidad más controlada.

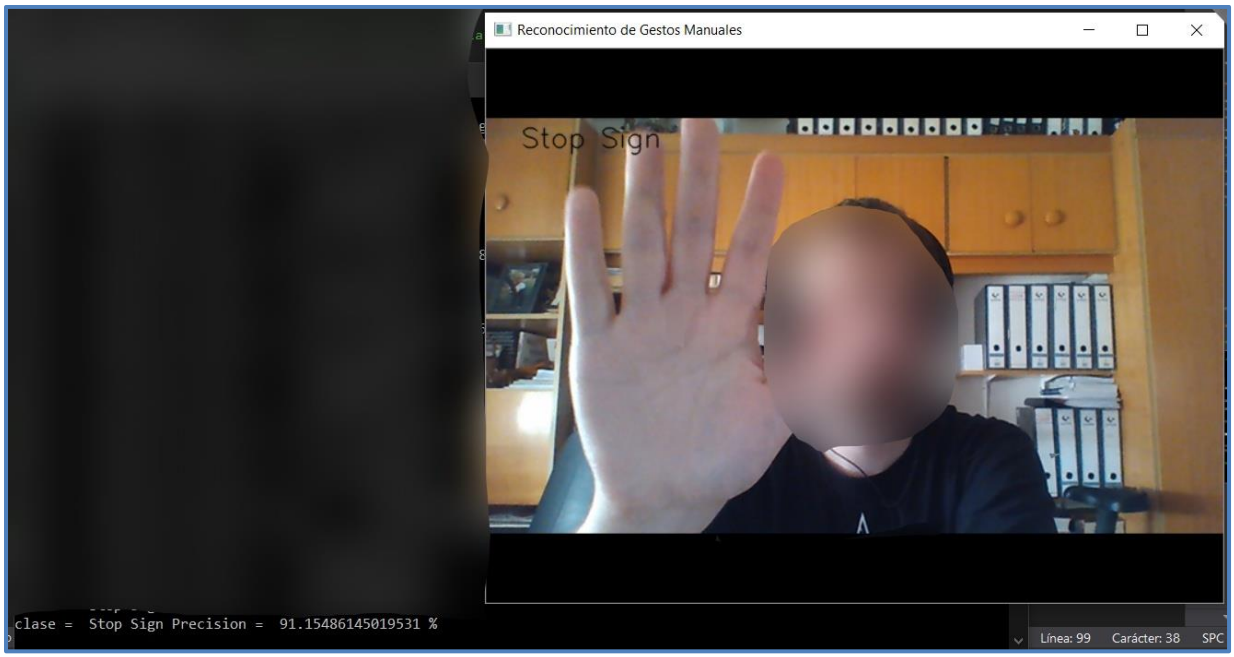

Ilustración 84. Gesto "Stop Sign" correctamente reconocido.

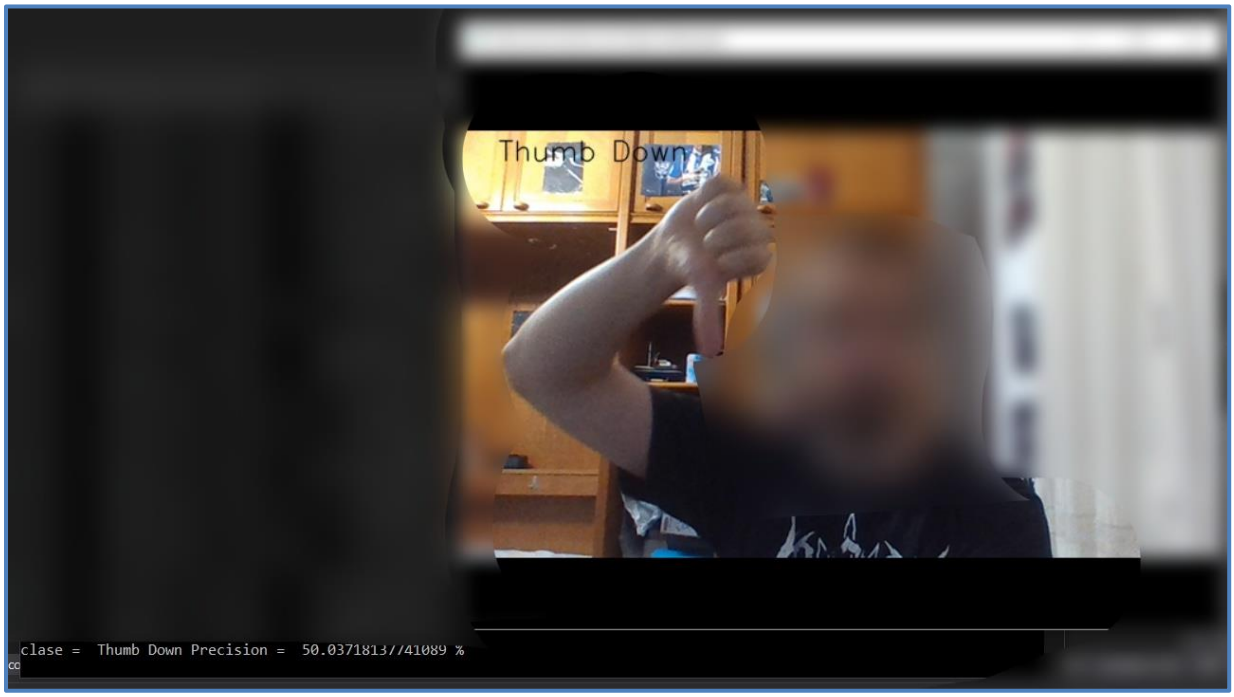

Ilustración 85. Gesto "Thumb down", pulgar hacia abajo.

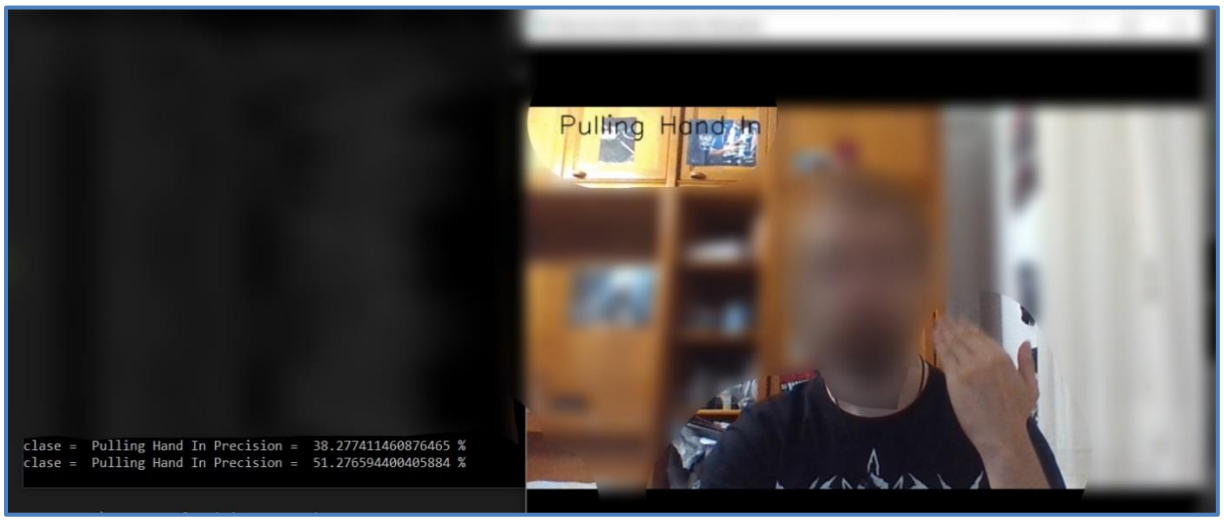

Ilustración 86. Gesto "Pulling Hand In". La mano se desplaza desde la cámara hasta el usuario.

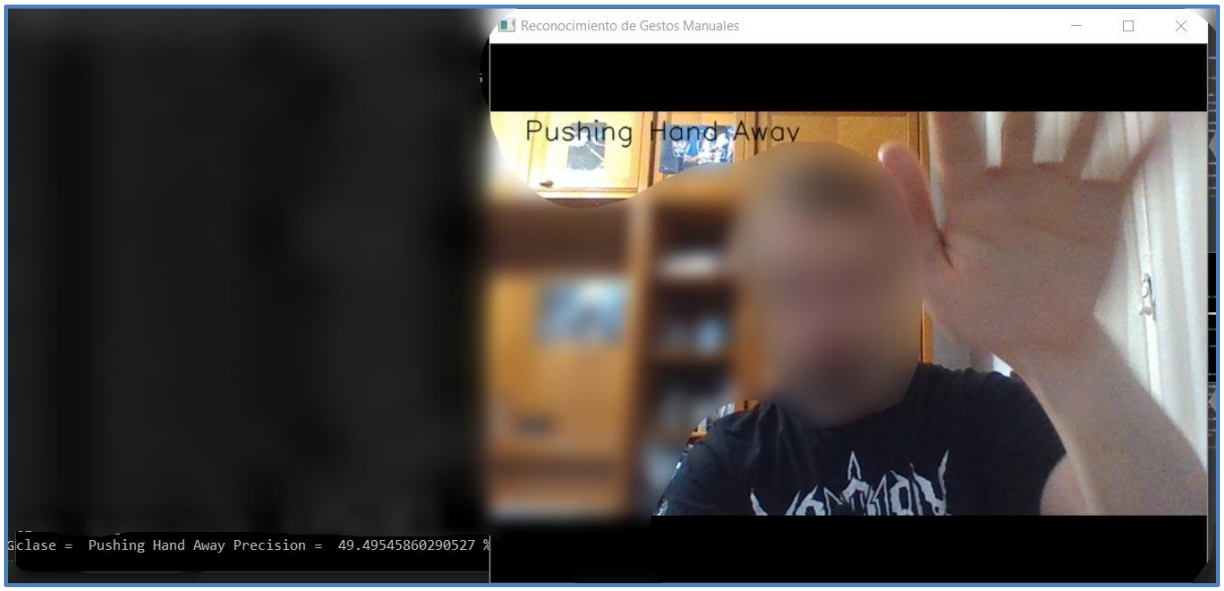

Ilustración 87. Gesto "Pushing Hand Away", la mano se desplaza desde el usuario hasta la cámara.

En las ilustraciones 84, 85, 86 y 87, podemos comprobar cómo la red neuronal ha aprendido correctamente a reconocer los gestos que hemos añadido al conjunto, por lo que nuestra red neuronal estaría ya completa.

8.4 Testeo y uso de la red neuronal

El último módulo software a explicar sería aquel en el que usamos nuestra red neuronal, ya sea para testearla en el proceso de desarrollo de la misma, o para su uso posterior. Lo que hace este módulo básicamente es crear el mismo modelo de nuestra red neuronal, cargar los pesos de la misma, capturar vídeo en directo por la webcam de nuestro equipo y pasar estas imágenes a la red neuronal artificial, que nos indicará por pantalla qué gesto estamos realizando.

Lo primero que tiene que hacer el programa es cargar las librerías de apoyo, como en los otros programas. Para ello (ilustración 88):

```
import tensorflow as tf
print(tf.__version__)
import numpy as np
import cv2
from sklearn.preprocessing import StandardScaler
```

Ilustración 88. Imports.

En orden, importamos TensorFlow, NumPy, OpenCV y Scikit-learn, que son las librerías que necesitaremos para testear y usar la red neuronal, así como para usar la webcam.

Una vez importadas, podemos usar bastante código que ya tenemos en el módulo de entrenamiento. Primero, al igual que en ese módulo, debemos establecer las clases que queremos reconocer, que serán las mismas con las que la red neuronal ha sido entrenada. Por ejemplo:

```
clases = ['Swiping Up', 'Swiping Down', 'Swiping Right', 'Swiping Left',
          'No gesture', 'Doing other things', 'Thumb Up']
```

Ilustración 89. Etiquetas gestos manuales.

Lo siguiente sería la clase de Python en la que creamos nuestro modelo, que será la misma que la que se encuentra en el módulo de entrenamiento. Al ser la misma clase, y además depender de en qué momento del desarrollo de la red neuronal nos encontramos, no se explicará de nuevo.

Una vez creada la clase del modelo, creamos un objeto de esa clase, pero no lo compilamos, ya que ahora no estamos entrenando. Lo que sí haremos será cargar los pesos que hemos obtenido durante el entrenamiento (ilustración 90).

```
nuevo_modelo = HandGestureReconModel()   #creamos modelo


#%%  #cargamos los pesos
nuevo_modelo.load_weights("D:\\pesos_red_neuronal")
```

Ilustración 90. Carga del modelo entrenado.

Cuando tenemos el modelo preparado, sólo falta capturar vídeo en directo por la webcam, procesarlo, y entregárselo a la red neuronal, para que ésta nos diga qué gesto manual estamos haciendo. Esto lo hacemos en el fragmento de código de la ilustración 91.

```
video = []
capturando = cv2.VideoCapture(0)    #Comenzamos a capturar por la webcam
clase =''

while(True):
    # Capturamos cada imagen
    ret, frame = capturando.read()
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) #pasamos las imágenes a gris
    video.append(cv2.resize(gray_frame, (64, 64))) #la reescalamos y metemos en el array

    if len(video) == 30:
        video_predecir = np.array(video, dtype=np.float32)    #lo pasamos a array adecuado Numpy de 32 bits
        video_predecir = video_predecir.reshape(-1, 30, 64, 64, 1)     #reescalamos otra vez
        prediccion = nuevo_modelo.predict(video_predecir)       #predecimos
        clase = clases[np.argmax(prediccion)]               #qué clase es

        print('clase = ',clase, 'Precision = ', np.amax(prediccion)*100,'%')  #imprimimos la predicción por consola
        video = []   #vaciamos

    cv2.putText(frame, clase, (30, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 0),1,cv2.LINE_AA)      #ponemos la predicción en el video

    # Mostramos las imágenes por pantalla
    cv2.imshow('Reconocimiento de Gestos Manuales',frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Cuando terminemos, destruimos ventanas y demás
capturando.release()
cv2.destroyAllWindows()
```

Ilustración 91. Captura de imagen y predicción de gesto manual.

En resumen, lo que hace el código de la ilustración, es simplemente capturar mediante la webcam un vídeo en directo. Las imágenes de este vídeo después, son pasadas a escala de grises y a un tamaño adecuado, para así tener el mismo formato con el que la red neuronal fue entrenada. Acto seguido usamos el método “predict()” para predecir el gesto manual, y después lo imprimimos por consola, además de aparecer en una ventana donde vemos también el vídeo.

9. Pruebas y resultados

Durante el desarrollo de la red neuronal, así como una vez terminada la misma, se han realizado una serie de pruebas y testeos para comprobar la eficacia de la red neuronal en torno al reconocimiento de gestos manuales. Por tanto, en este apartado se explicará qué tests se han llevado a cabo en cada fase y qué resultados han sido obtenidos.

9.1 Tipos de pruebas

Por un lado, durante el entrenamiento de la red neuronal, se producía un testeo automático al final de cada ciclo. Este test era el proceso llamado “validación”. Mediante éste, podíamos ir comprobando cómo de bien reconocía nuestra red neuronal los gestos manuales de una serie de vídeos, los cuáles formaban el “set de validación”. El resultado de la validación era devuelto con un valor numérico llamado “val_loss”, que hace referencia al concepto de “error de validación”. Este error de validación lo comparábamos con el error “loss” del propio entrenamiento: si era ligeramente menor o ligeramente mayor, el entrenamiento estaba siendo relativamente correcto, pero, si por el contrario, el error de validación se iba haciendo cada vez mayor mientras que el error disminuía, nos encontrábamos frente al fenómeno del “overfitting”.

Este “val_loss” se va calculando con cada muestra del set de validación, por lo que tomaremos el promedio como valor final. También hay que mencionar cuántas muestras de validación estamos usando en cada fase, porque influirá en ese promedio. Por ello, el número de muestras de validación en cada fase del proyecto ha sido siempre de 400 muestras.

Por otro lado, al finalizar cada entrenamiento, es decir, cada iteración en nuestro desarrollo de la red neuronal artificial, lo que hacíamos era testear manualmente usando la webcam del PC. Las razones de esto eran que, en primer lugar, los valores numéricos de los que hablábamos son útiles, pero hasta cierto punto sólo de forma orientativa. Lo que de verdad importa, es observar empíricamente que la red neuronal artificial reconoce realmente los gestos manuales deseados. Esto lo hacíamos mediante el módulo software de testeo y uso de la red neuronal, que capta imágenes en directo desde la webcam de nuestro PC, y las preprocesa y entrega a la red neuronal.

9.2 Resultados

9.2.1 Validación

A continuación, se muestra una tabla con cada error de validación correspondiente a cada ciclo y fase del desarrollo del proyecto. Las filas representan los ciclos, mientras que las columnas, la fase concreta del proyecto. Nótese que para las dos últimas fases tenemos dos filas más, puesto que el entrenamiento duró más tiempo.

Tabla 1. Resultados de la validación a lo largo del proyecto.

Ciclo	1º Entrenamiento	Tuning	Dropout	SpatialDropout	Batch Training	A. Final
1	2,001	2,002	2	1,9999	1,4021	1,2
2	1,5668	1,5565	1,4888	1,4955	0,85	0,799
3	1,0021	1,0106	1,002	0,9595	0,8505	0,699
4	0,8865	0,8645	0,6656	0,6891	0,6944	0,5899
5	0,5542	0,9055	0,7012	0,6902	0,5998	0,4888
6	0,7242	0,7885	0,6433	0,521	0,4207	0,4566
7	1,3353	1,3542	0,6898	0,4927	0,4002	0,3999
8					0,3788	0,3444
9					0,3802	0,2528

Podemos representar también, de forma gráfica, los resultados expuestos en la tabla 1, como se muestra en la ilustración 92.

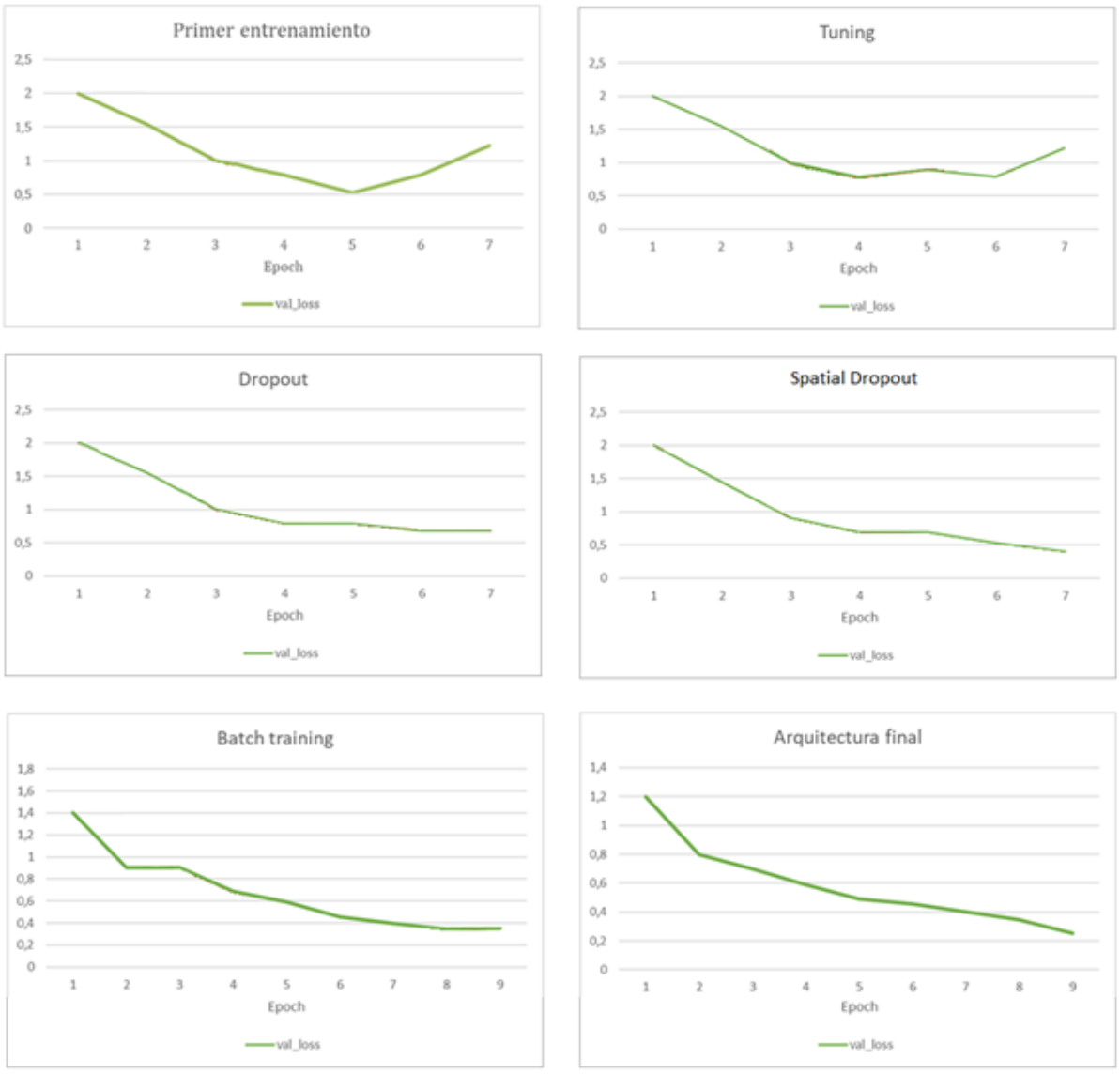

Ilustración 92. Resultados de la validación a lo largo del proyecto.

Por último, es interesante ver de forma gráfica, esta vez tridimensional, los resultados de la tabla 1 (ilustración 93).

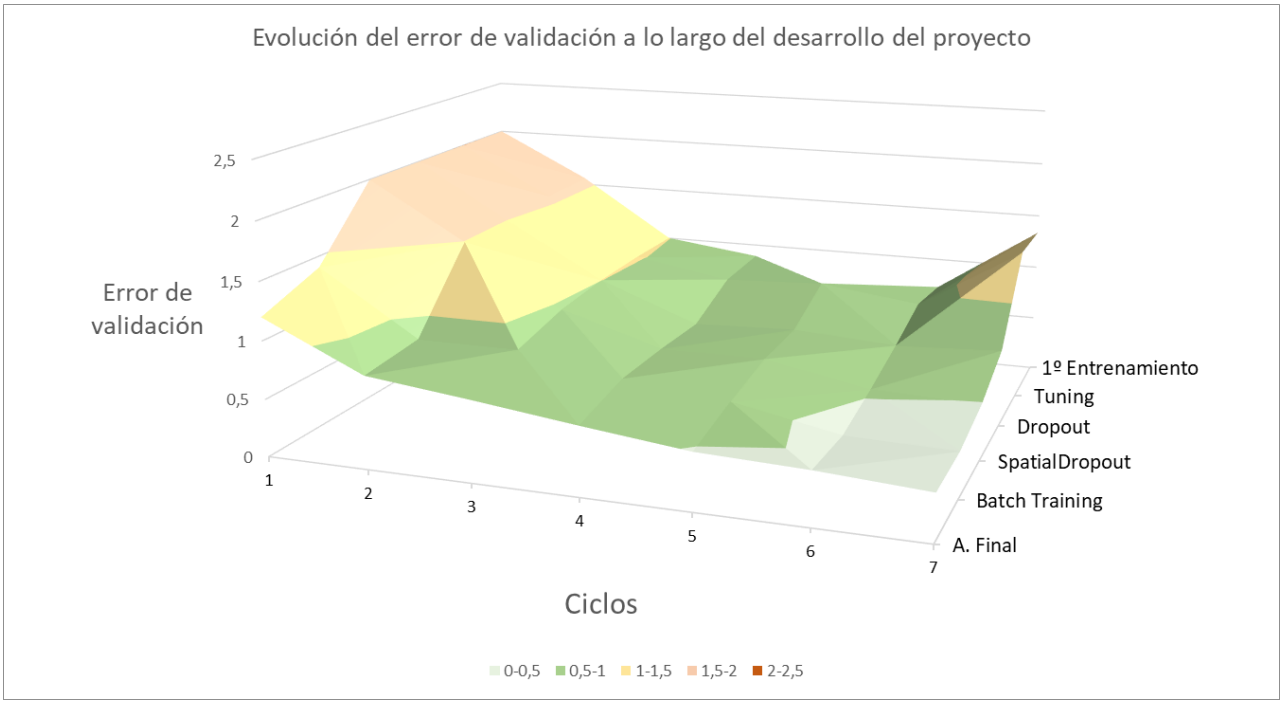

Ilustración 93. Gráfica tridimensional con los errores de validación.

En la ilustración 93, podemos ver en el eje 'x' el número de ciclos, en el eje 'y' cada fase del entrenamiento, y en el eje 'z' (altura), el error de validación. Además, tenemos un gradiente de colores que nos ayuda a apreciar mejor la altura de la gráfica, es decir, cuánto de grande es el error de validación para cada ciclo y fase del proyecto. Con todo esto en mente, al interpretar la gráfica de la ilustración 93, podemos ver cómo para las primeras fases del proyecto (1º Entrenamiento, Tuning), el error de validación comienza alto, después va bajando con los ciclos, para después volver a subir (overfitting). En cambio, para las últimas fases del proyecto (Batch Training, Arquitectura Final), además de empezar por niveles más bajos de error, conseguimos llegar a un valle más que evidente. Esto nos demuestra, de forma numérica, la efectividad de nuestra red neuronal y en concreto, la efectividad de nuestra arquitectura final.

9.2.2 Testeo

Finalmente, pasamos a probar nuestra red de forma manual, haciendo uso del módulo software dedicado al uso o testeo de la misma. Con este fin, un total de 3 usuarios han realizado los gestos manuales deseados, en 20 ocasiones cada uno, y usando la arquitectura final. A continuación, se muestran los resultados obtenidos.

En la tabla 2 se muestra la precisión media de cada gesto reconocido por usuario. Es decir, teniendo en cuenta los gestos de cada usuario reconocidos positivamente, se muestra con qué precisión media ha sucedido. Por tanto, no debemos confundir los porcentajes mostrados con la media de ocasiones en las que cada gesto ha sido reconocido. Esto último se mostrará en la tabla 3.

Tabla 2. Precisión media de cada gesto reconocido positivamente por usuario.

	Usuario 1	**Usuario 2**	**Usuario 3**	$\overline{X}$
Swiping Left	93.2%	90.12%	90%	*91.11%*
Swiping Right	94.2%	92%	93.33%	*93.18%*
Swiping Up	90.12%	89.39%	90.14%	*89.88%*
Swiping Down	95.36%	93.38%	94.93%	*94.55%*
Thumb Up	75.68%	80.33%	89.01%	*81.67%*
Thumb Down	50.63%	63.44%	49.98%	*54.68%*
Stop Sign	70.61%	69.08%	75.77%	*71.82%*
Pulling Hand In	35.98%	34.44%	41.98%	*37.47%*
Pushing Hand Away	29.37%	35.66%	37.81%	*34.28%*
No Gesture	97.09%	92.88%	96.56%	*95.51%*
Doing Other Things	97.98%	95.36%	93.51%	*95.62%*
			$\overline{X}$	*76.34%*

Según la tabla 2, en las pruebas realizadas obtenemos una precisión media del 76.34%. Esto quiere decir que, en las pruebas realizadas, de media, cuando la red neuronal reconoce positivamente un gesto, lo hace con una precisión del 76.34%. Es decir, la red neuronal está "bastante segura" de cada reconocimiento.

Por otra parte, en la tabla 3 vemos el porcentaje de gestos acertados por usuario.

Tabla 3. Porcentaje de gestos reconocidos positivamente por usuario.

	Usuario 1	**Usuario 2**	**Usuario 3**	$\overline{X}$
Swiping Left	100%	100%	100%	*100%*
Swiping Right	100%	100%	100%	*100%*
Swiping Up	100%	100%	100%	*100%*
Swiping Down	100%	100%	100%	*100%*
Thumb Up	95%	100%	90%	*95%*
Thumb Down	80%	80%	75%	*78.33%*
Stop Sign	90%	90%	85%	*88.33%*
Pulling Hand In	65%	70%	70%	*68.33%*
Pushing Hand Away	70%	70%	70%	*70%*
No Gesture	100%	100%	100%	*100%*
Doing Other Things	100%	100%	100%	*100%*
			$\overline{X}$	*90.91%*

Según la tabla 3, en un 90.91% de las pruebas realizadas, nuestra red ha reconocido de forma correcta el gesto mostrado, lo cual es un resultado muy aceptable.

10. Planificación del trabajo

En este apartado se muestran los distintos paquetes de trabajo en los que se ha dividido el proyecto. También se mostrarán los hitos del mismo y, finalmente, el diagrama de Gantt.

10.1 Descripción de tareas

10.1.1 Paquetes de trabajo

Tabla 4. Primer paquete de trabajo.

PT 1	Duración	Comienzo	Fin	CT (Ingeniero)	CT (Director)
Gestión del Proyecto *Seguimiento realizado para comprobar el correcto desarrollo del proyecto.*	206 días	vie 07/02/20	lun 07/09/20	*Total: 10h*	*Total: 20h*
T 1.1 Gestión y seguimiento del trabajo	203 días	lun 10/02/20	lun 07/09/20	10h	20h

Tabla 5. Segundo paquete de trabajo.

PT 2	Duración	Comienzo	Fin	CT (Ingeniero)
Preparación del proyecto *Adquisición de conocimientos y perspectiva necesarios para empezar a desarrollar el proyecto.*	28 días	lun 10/02/20	lun 09/03/20	*Total: 56h*
T 2.1 Análisis estado del arte	21 días	lun 10/02/20	lun 02/03/20	21h
T 2.2 Adquisición conocimientos	28 días	lun 10/02/20	lun 09/03/20	21h
T 2.3 Análisis alternativas	7 días	lun 02/03/20	lun 09/03/20	14h

Tabla 6. Tercer paquete de trabajo.

PT 3	Duración	Comienzo	Fin	CT (Ingeniero)
Desarrollo del proyecto *Fases llevadas a cabo en el desarrollo del proyecto en sí mismo.*	151 días	mar 10/03/20	mié 13/08/20	*Total: 604h*
T 3.1 Selección de alternativas	11 días	mar 10/03/20	sáb 21/03/20	44h
T 3.2 Diseño Arquitectura Proyecto	6 días	mié 25/03/20	mar 31/03/20	24h

T 3.3 Desarrollo Programa DataSet	5 días	jue 02/04/20	lun 07/04/20	20h
T 3.4 Preparación Aceleración GPU	4 días	mié 01/04/20	dom 05/04/20	16h
T 3.5 Desarrollo Programa Testeo	4 días	lun 06/04/20	jue 10/04/20	16h
T 3.6 Desarrollo Programa Entrenamiento	120 días	vie 10/04/20	mié 12/08/20	484h

Tabla 7. Cuarto paquete de trabajo.

PT4	Duración	Comienzo	Fin	CT (Ingeniero)	CT (Director)
Documentación y Presentación del Proyecto	149 días	lun 06/04/20	lun 07/09/20	*Total: 51h*	*Total: 10h*
T 4.1 Documentación del Proyecto	149 días	lun 06/04/20	lun 07/09/20	41h	9h
T 4.2 Presentación para el Proyecto	7 días	mié 31/08/20	lun 07/09/20	10h	1h

Teniendo en cuenta las horas totales de la Carga de Trabajo para cada Paquete de Trabajo, obtenemos las siguientes horas totales del proyecto, para cada miembro del Equipo de Trabajo:

Tabla 8. Carga de Trabajo para cada miembro del Equipo de Trabajo

Miembro del Equipo de Trabajo	Suma de las CT	Total
Ingeniero	10 + 56 + 604 + 51	*721*
Director	20 + 10	*30*

10.1.2 Hitos del proyecto

A continuación, se muestran los diferentes hitos alcanzados.

Tabla 9. Hitos del proyecto

	Hito	Fecha
Hito 1	Propuesta del Proyecto	07/02/20
Hito 2	Comienzo del Proyecto	10/02/20
Hito 3	Finalización de la documentación	07/09/20
Hito 4	Finalización de la presentación	07/09/20
Hito 5	Finalización del Proyecto	07/09/20

10.2 Diagrama de Gantt

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**PT 1 Gestión del Proyecto**	**206 días**	**vie 07/02/20**	**lun 07/09/20**
2	H1 Propuesta del Proyecto	0 días	vie 07/02/20	vie 07/02/20
3	T 1.1 Gestión y seguimiento del trabajo	203 días	lun 10/02/20	lun 07/09/20
4	H2 Comienzo del proyecto	0 días	lun 10/02/20	lun 10/02/20
5	**PT 2 Preparación del proyecto**	**28 días**	**lun 10/02/20**	**lun 09/03/20**
6	T 2.1 Análisis estado del arte	21 días	lun 10/02/20	lun 02/03/20
7	T 2.2 Adquisición conocimientos	28 días	lun 10/02/20	lun 09/03/20
8	T 2.3 Análisis alternativas	7 días	lun 02/03/20	lun 09/03/20
9	**PT 3 Desarrollo del proyecto**	**151 días**	**mar 10/03/20**	**jue 13/08/20**
10	T 3.1 Selección de alternativas	11 días	mar 10/03/20	sáb 21/03/20
11	T 3.2 Diseño Arquitectura Proyecto	6 días	mié 25/03/20	mar 31/03/20
12	T 3.3 Desarrollo Programa DataSet	5 días	jue 02/04/20	mar 07/04/20
13	T 3.4 Preparación Aceleración GPU	4 días	mié 01/04/20	dom 05/04/20
14	T 3.5 Desarrollo Programa Testeo	4 días	lun 06/04/20	vie 10/04/20
15	**T 3.6 Desarrollo Programa Entrenamiento**	**120 días**	**vie 10/04/20**	**mié 12/08/20**
16	T 3.6.1 Desarrollo Primera Arquitectura	16 días	vie 10/04/20	dom 26/04/20
17	T 3.6.2 Fase Tuning	15 días	lun 27/04/20	mar 12/05/20
18	T 3.6.3 Modificación DropOut	11 días	mié 13/05/20	dom 24/05/20
19	T 3.6.4 Modificación Spatial DropOut	12 días	jue 28/05/20	mar 09/06/20
20	T 3.6.5 Batch Training	20 días	mié 10/06/20	mar 30/06/20
21	T 3.6.6 Ampliación de Gestos	21 días	mié 01/07/20	mié 22/07/20
22	T 3.6.7 Arquitectura Final	18 días	vie 24/07/20	mar 11/08/20
23	**PT 4 Documentación y Presentación del Proyecto**	**149 días**	**lun 06/04/20**	**lun 07/09/20**
24	T 4.1 Documentación del Proyecto	149 días	lun 06/04/20	lun 07/09/20
25	H3 Finalización Documentación	0 días	lun 07/09/20	lun 07/09/20
26	T 4.2 Presentación para el Proyecto	7 días	lun 31/08/20	lun 07/09/20
27	H4 Finalización Presentación	0 días	lun 07/09/20	lun 07/09/20
28	H5 Finalización del Proyecto	0 días	lun 07/09/20	lun 07/09/20

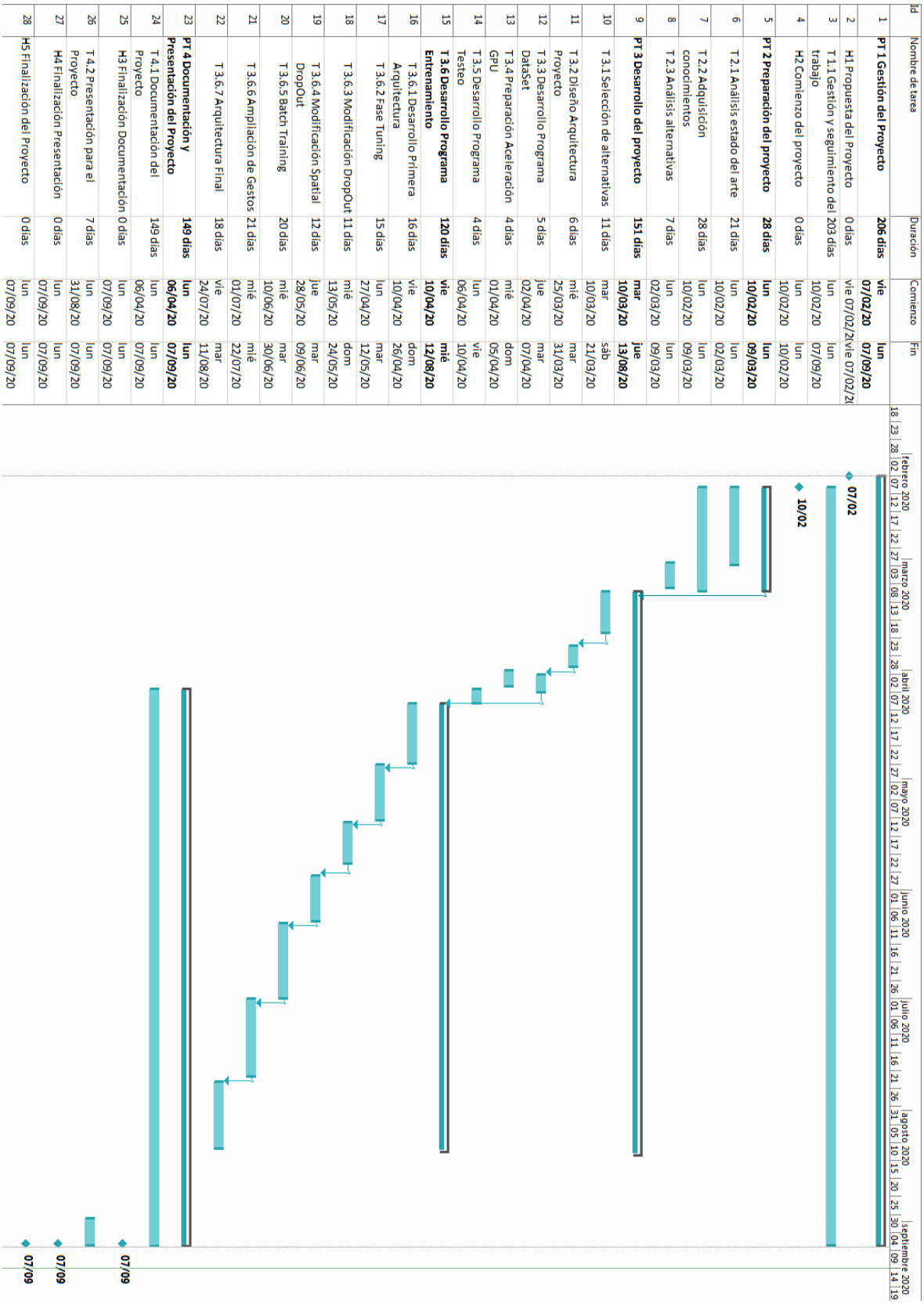

Ilustración 94. Diagrama de Gantt.

11. Desglose de costes

En este apartado se presenta el desglose de costes del proyecto, en el que se calculan, por un lado, los costes de los recursos humanos empleados y, por otro lado, los costes de los materiales utilizados.

En cuanto a costes materiales, no han habido de tipo fungible. Los únicos costes materiales han sido material amortizable.

11.1 Recursos humanos

En el apartado de recursos humanos, contamos con dos miembros en el equipo de trabajo. Por un lado, el director del proyecto, y por otro lado el que desarrolla el proyecto, el Ingeniero. Estos tienen un coste por hora trabajada diferente, y la cantidad de horas que le dedican al proyecto también varía.

En la tabla 10, se muestra un desglose de las horas internas de cada integrante del equipo.

Tabla 10. Recursos humanos.

	€/h	***Cantidad (h)***	***Total (€)***
Director del proyecto	40	30	1200
Ingeniero	25	721	18025
Total			**19225**

11.2 Recursos materiales

En cuanto a recursos materiales, hemos usado un ordenador personal tanto para desarrollar el proyecto como la documentación, contando este con todo el hardware necesario. Para escribir la documentación, hemos usado los diferentes programas de Office 2016.

Tabla 11. Recursos materiales.

	Coste inicial (€)	**Vida útil (meses)**	**Uso (meses)**	**Coste (€)**
Ordenador	1200	60	7.5	150
Office 2016	150	60	5	12.5
Total				**162.5**

Además de esto, se ha hecho uso de programas y librerías totalmente gratuitos, como el IDE Visual Studio, las librerías TensorFlow y Keras, y otras herramientas adicionales. Al no tener ningún coste, no se añaden a la tabla.

11.3 Resumen del desglose de costes

Por último, se presenta el resumen del descargo de gastos total, donde se tienen en cuenta todos los gastos mencionados anteriormente.

Tabla 12. Resumen desglose de costes.

	Coste
Horas internas	19225 €
Amortizaciones	162.5 €
Total	**19387.5 €**

12.Conclusiones

A lo largo de este TFM se ha logrado desarrollar una red neuronal artificial capaz de reconocer gestos manuales. Durante la elaboración del mismo, se han encontrado diversos problemas que se han conseguido solucionar, tales como el fenómeno del overfitting, las distintas limitaciones del hardware o problemas con el software en su codificación. Además, hay que tener en cuenta que el desarrollo de una red neuronal artificial es un proceso iterativo, de ensayo y error, lo cual trae muchos quebraderos de cabeza.

Sin embargo, gracias a los conocimientos adquiridos en el Máster Universitario en Ingeniería de Telecomunicación, continuación natural del Grado en Ingeniería en Tecnología de Telecomunicación, y a la gran cantidad de material científico de libre disposición sobre esta área, se han podido solventar dichos problemas. Prácticamente, todas las librerías de software disponibles para el desarrollo de inteligencia artificial son gratuitas, y/o de código abierto, por lo que tenemos garantizado el acceso a una inmensa cantidad de información y conocimiento científico.

Como ingeniero de telecomunicación, se ha podido buscar, encontrar y comprender, de entre toda esa cantidad de información, lo necesario para resolver los problemas hallados durante el desarrollo del proyecto. Por ejemplo, si aparecía un problema o error concreto en el código fuente, uno podía buscar dicho error en internet, para encontrarse con miles de páginas webs con soluciones o sugerencias, además de poder buscar en la propia documentación del lenguaje de programación o biblioteca correspondiente. Si por el contrario, el problema estaba relacionado con la teoría de las redes neuronales artificiales, uno podía buscar ese problema o esa duda en la web y encontrar, de nuevo, miles de artículos o documentos en los que se trataba la cuestión. Por supuesto, todo conocimiento ajeno ha sido referenciado debidamente.

De todas formas, el mayor problema encontrado ha sido el del fenómeno del “overfitting”. Este es un problema “maldito” para los programadores de redes neuronales, puesto que puede ser difícil de detectar y, aun detectándolo, se vuelve todavía más difícil encontrar su origen. Puede ser culpa de tener pocos datos de entrenamiento, de tener muchos pero muy similares entre sí, de que los datos de validación sean demasiado diferentes a los de entrenamiento o incluso provengan de otro dataset distinto, de que la arquitectura no sea adecuada, de que hayamos entrenado la red neuronal demasiado, etc. En este proyecto, la solución ha sido una mezcla de modificar la arquitectura, aumentar el tamaño del dataset y añadir capas “dropout”, como se vio sugerido en gran cantidad de artículos y documentos en la web.

Por tanto, el libre intercambio de ideas y creaciones está provocando un rápido crecimiento científico en la materia, además de aumentar el interés general en la misma. Tanto es así, que si uno escribe “red neuronal” en el buscador Google, obtendrá más de 37 millones de resultados, o incluso 137 millones de resultados si la búsqueda se realiza en inglés. Además, en el repositorio de código GitHub encontraremos más de 92.000 repositorios con redes neuronales, cifra que aumenta cada día.

En cuanto al reconocimiento de gestos manuales mediante red neuronal en particular, encontramos también bastantes investigaciones con diferentes aproximaciones. Una simple

búsqueda en Google Scholar sobre el tema nos dará unos 128.000 artículos científicos, cifra más que notable. Hay que destacar también que este TFM se ha enfocado hacia el área de las interfaces humano-computadora, para ser capaces de usar las manos para controlar un ordenador, pero también existen otros enfoques, como por ejemplo el reconocimiento de gestos manuales en el lenguaje de signos o señas, ayudando así a personas con discapacidades auditivas. Esto es algo en lo que ya se está investigando, y en el que se usan modelos basados en redes neuronales convolucionales 3D similares a la de este TFM. [50] Esto significa que sería posible, como mejora del proyecto, comprobar si el modelo usado en este trabajo es también capaz de reconocer los distintos gestos usados en el lenguaje de signos, completando aún más las funcionalidades de la red neuronal.

Algo a mencionar cuando hablamos de posibles mejoras, es el considerar el uso de hardware superior al utilizado. Como se ha visto, hemos sido capaces de sobrepasar las limitaciones físicas de nuestro equipo, en detrimento del tiempo. Esto provocaba que para completar una fase de entrenamiento, pasara un periodo de 24 horas o más, lo cual es una gran cantidad de tiempo que para nada facilita el trabajo. Si, por ejemplo, queríamos comprobar cómo cambiaba nuestro modelo al modificar un simple parámetro, había que esperar todo ese tiempo, resultando un proceso nada eficiente ni cómodo. Por ello, si se tuviera acceso a un equipo mucho más potente, podríamos ser capaces de realizar el mismo proyecto en mucho menos tiempo, además de poder seguir con su desarrollo ampliando capacidades y mejorando y haciendo más eficiente su arquitectura.

Otro aspecto a mencionar es la sensibilidad lumínica de nuestro modelo. Algunas de las pruebas se realizaron al lado de una ventana que producía un resplandor lateral bastante importante, y proyectaba gran cantidad de luz sobre el usuario. A simple vista, una persona podía seguir reconociendo los gestos manuales, es decir, un humano podía distinguirlos en las imágenes captadas, pero la red neuronal no era capaz. Esto en un principio se identificó como falsos errores de diseño, resultando en mayores quebraderos de cabeza hasta ser conscientes del fenómeno, y de que era una circunstancia del test en particular. Aun así, sería interesante incluir ejemplos de este tipo en el dataset, para entrenar a la red en condiciones lumínicas mucho más variables y desfavorables, haciéndola más robusta.

También hay que señalar que la red neuronal detecta ciertos gestos mejor que otros. Por ejemplo, los movimientos simples de desplazamiento son detectados más veces y con mayor precisión que otros más complejos, como pueden ser los del “pulgar hacia abajo” o “mover la mano hacia dentro”. De nuevo, si contáramos con un mejor hardware y más tiempo, podríamos ser capaces de mejorar aún más los resultados.

De todas formas, en resumen, se han cumplido los objetivos propuestos, obteniendo una red neuronal artificial muy completa, con una precisión de 76.34% y un porcentaje de acierto de 90.91%, según las pruebas realizadas.

Para terminar, cabe destacar el contexto en el que se ha realizado este TFM. Estamos viviendo un momento de auge increíble en torno a la inteligencia artificial en general, y a las redes neuronales artificiales en particular. Por todo esto, es casi una obligación mencionar al modelo GPT-3. Esta red neuronal es un modelo dedicado al procesamiento del lenguaje natural, y es la más potente actualmente. Es capaz de crear textos coherentes de cientos de

páginas, además de ser capaz de traducir textos e incluso pasar el Test de Turing, cosas para lo que no había sido entrenada. [49] Los creadores de esta IA, OpenAI, han decidido liberar este modelo para todo el público, con el fin de que otros investigadores lo analicen, se inspiren para otros modelos o incluso consigan mejorarlo. Esta es la mejor parte, puesto que permitirá aumentar aún más el desarrollo de las redes neuronales en particular, y de la ciencia en general, lo que es tremendamente positivo para el progreso humano.

Por tanto, vemos que el área de este TFM es un área que se supera a sí misma cada día, o incluso cada minuto, y que en los últimos años ha traído unos avances que no se habían visto en décadas. Estos avances, además, son exponenciales, puesto que permiten la expansión del conocimiento y la mejora de herramientas, permitiendo más y más avances, que son compartidos con toda la comunidad científica. Esto último, sumado a la versatilidad del Ingeniero de Telecomunicación, le permiten a éste adaptarse a las nuevas tecnologías independientemente de su especialidad.

13.Referencias

[1] D. A. Freedman, Statistical Models: Theory and Practice, Cambridge University Press, 2009.

[2] A. C. Rencher y W. F. Christensen, Methods of Multivariate Analysis, John Wiley & Sons, 2012.

[3] K. Hauser, «Lecture 4: Gradient Descent,» Indiana University, 2012.

[4] D. Rumelhart, G. Hinton y J. L. Mcclelland, A General Framework for Parallel Distributed Processing, 1986.

[5] J. Brownlee, «A Gentle Introduction to the Rectified Linear Unit (ReLu),» *Machine Learning Mistery,* 6 Agosto 2019.

[6] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[7] D. E. Rumelhart, G. E. Hinton y R. J. Willians, «Learning representations by bac-propagating errors,» *Nature,* 1986.

[8] V. Bushaev, «How do we 'train' neural networks?,» *Towards Data Science,* 27 Noviembre 2017.

[9] J. Brownlee, «What is the Difference Between a Batch and an Epoch in a Neural Network?,» *Deep Learning,* 20 Julio 2018.

[10] H. Zulkifli, «Understanding Learning Rates and How It Improves Performance in Deep Learning,» *Towards Data Science,* 2018.

[11] Google, «Conjuntos de entrenamiento y prueba: Separación de datos,» 2020. [En línea]. Available: https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data.

[12] R. Keim, «Understanding Color Models Used in Digital Image Processing,» *All About Circuits,* 2018.

[13] I. Goodfellow, Y. Bengio y A. Courville, Deep Learning, MIT Press, 2016.

[14] R. Poplin, A. V. Varadarajan, K. Blumer, Y. Liu, M. V. McConnell, G. S. Corrado, L. Peng y D. R. Webster, «Predicting Cardiovascular Risk Factors from Retinal Fundus Photographs using Deep Learning,» *Cornell University arXiv,* 2017.

[15] A. Dominguez-Sanchez, M. Cazorla y S. Orts-Escolano, «Pedestrian movement direction recognition using convolutional neural networks,» *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS,* pp. 3540-3548, 2017.

[16] N. Mohammadian Rad, A. Bizzego, K. Seyed Mostafa, G. Jurman, P. Venuti y C. Furlanello, «Convolutional Neural Network for StereotypicalMotor Movement Detection in Autism,» *Cornell University arXiv,* 2016.

[17] S. Hochreiter y J. Schmidhuber, «Long Short-term Memory,» *Neural Computation,* 1997.

[18] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke y J. Schmidhuber, «A Novel Connectionist System for Unconstrained Handwriting Recognition,» *IEEE Transactions on Pattern Analysis and Machine Intelligence,* pp. 855-868, 2009.

[19] H. Sak, A. Senior y F. Beaufays, «Long Short-Term Memory Recurrent Neural Network Architecturesfor Large Scale Acoustic Modeling,» *Google Research,* 2018.

[20] S. Boyko, «Applying Long Short-Term Memory for Video Classification Issues,» 2019. [En línea]. Available: https://www.apriorit.com/dev-blog/609-ai-long-short-term-memory-video-classification.

[21] M. Phi, «Illustrated Guide to LSTM's and GRU's: A step by step explanation,» *Towards Data Science,* 2018.

[22] S. Wilhelm Pienaar y R. Malekian, «Human Activity Recognition Using LSTM-RNN Deep Neural Network Architecture,» *Cornell University arXiv,* 2019.

[23] Nvidia, «Artificial Neural Network. Accelerating Artificial Neural Networks with GPUs,» 2020. [En línea]. Available: https://developer.nvidia.com/discover/artificial-neural-network.

[24] Nvidia, «What's the Difference Between a CPU and a GPU?,» 2009. [En línea]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/.

[25] O. Kyoung-Su y J. Keechul, «GPU implementation of neural networks,» *Pattern recognition,* pp. 1311-1314, 2004.

[26] Synced, «GTC 2019 | New NVIDIA One-Stop AI Framework Accelerates Workflows by 50x,» *Medium,* 19 Marzo 2019.

[27] Enzyme Advising Group, «Redes neuronales con Python: ¿por qué es el mejor lenguaje para IA?,» *Enzyme Advising group Blog,* 2019.

[28] J. V. Carratala, «¿Cual es el mejor lenguaje de programación para inteligencia artificial?,» *JOCARSA,* 2019.

[29] Vector ITC, «Inteligencia Artificial y Open Source: una combinación ganadora,» 2019. [En línea]. Available: https://openexpoeurope.com/es/inteligencia-artificial-y-open-source-una-combinacion-ganadora/.

[30] TensorFlow, «¿Por qué TensorFlow? Casos de Éxito,» 2020. [En línea]. Available: https://www.tensorflow.org/about/case-studies.

[31] L. Armasu, «Google's Big Chip Unveil For Machine Learning: Tensor Processing Unit With 10x Better Efficiency (Updated),» *Tom's Hardware,* 2016.

[32] N. Ketkar, «Introduction to PyTorch,» de *Deep Learning with Python*, 2017, pp. 195-208.

[33] PyTorch, « Tutorials > Learning PyTorch with Examples > TensorFlow: Static Graphs,» 2017. [En línea]. Available: https://pytorch.org/tutorials/beginner/examples_autograd/tf_two_layer_net.html.

[34] Keras, «About Keras,» 2020. [En línea]. Available: https://keras.io/about/.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion y o. Grisel, «Scikit-learn: Machine Learning in Python,» *JMLR,* pp. 2825-2830, 2011.

[36] The SciPy community, «What is NumPy?,» 2020. [En línea]. Available: https://numpy.org/doc/stable/user/whatisnumpy.html.

[37] OpenCV Team, «About,» 2020. [En línea]. Available: https://opencv.org/about/.

[38] A. Maqueda, C. del Blanco, F. Jaureguizar y N. García, «Human–computer interaction based on visual hand-gesture recognition using volumetric spatiograms of local binary patterns,» *Computer Vision and Image Understanding,* pp. 126-137, 2015.

[39] T. Mantecón, C. del Blanco, F. Jaureguizar y N. García, «Hand Gesture Recognition using Infrared Imagery Provided by Leap Motion Controller,» de *Int. Conf. on Advanced Concepts for Intelligent Vision Systems, ACIVS 2016*, Lecce, Italia, Springer, 2016, pp. 47-57.

[40] Gigahertz-Optik, «II.6. Spectral sensitivity of the human eye,» 2020. [En línea]. Available: https://light-measurement.com/spectral-sensitivity-of-eye/.

[41] Nvidia, «CUDA Toolkit,» 2020. [En línea]. Available: https://developer.nvidia.com/cuda-toolkit.

[42] Nvidia, «NVIDIA cuDNN,» 2020. [En línea]. Available: https://developer.nvidia.com/cudnn.

[43] A. Karpathy, «A Peek at Trends in Machine Learning,» *Medium,* 2017.

[44] S. Ruder, «An overview of gradient descent optimization algorithms,» *Cornel University arXiv,* 2017.

[45] Atlassian Confluence, «Multi-hot Sparse Categorical Cross-entropy,» 2018. [En línea]. Available: https://cwiki.apache.org/confluence/display/MXNET/Multi-hot+Sparse+Categorical+Cross-entropy.

[46] A. Budhiraja, «Dropout in (Deep) Machine learning,» *Medium,* 2016.

[47] TensorFlow, «tf.keras.layers.SpatialDropout3D,» 2020. [En línea]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/SpatialDropout3D.

[48] R. A. Abir, «How to train with TensorFlow without hurting the ram,» *Medium,* 2017.

[49] T. B. Brown, B. Mann, N. Ryder y M. Subbiah, «Language Models are Few-Shot Learners,» *arXiv,* 2020.

[50] J. Huang, W. Zhou, H. Li y W. Li, «Sign Language Recognition using 3D convolutional neural networks,» *IEEE,* 2015.

14. Anexo: Módulos software codificados

14.1 Módulo de creación del set de entrenamiento y del set de validación

```
#Trabajo de Fin de Máster
#Autor: Jon San Martín Garaluce
#UPV-EHU

#HandRecon_SetCreation.py
#Python 3.6

#---------------------------IMPORTS-------------------------------------------------------------
import csv
import shutil
import random

#---------------------------INICIALIZACIONES----------------------------------------------------
path_destino = 'D:\\jon\\Documents\\training_samples3\\' # Donde vamos a guardar las muestras
de entrenamiento
csv_file = 'data_csv/jester-v1-train.csv' # training csv file

contador = 0
samples= 4000 # número de muestras para el entrenamiento
gestos = [ 'No gesture', 'Thumb Down', 'Thumb Up', 'Pulling Hand In', 'Pushing Hand Away',
'Doing other things', 'Stop Sign', 'Swiping Up', 'Swiping Down', 'Swiping Left', 'Swiping
Right'] # las clases a usar
rows = []

path_destino_val = 'D:\\jon\\Documents\\validation_samples3\\' # donde guardamos las muestras
de validación
csv_fle_val = 'data_csv/jester-v1-validation.csv' # validation csv file

contador_val = 0
samples_val= samples/10 # número de muestras de validación (10 veces menos)

#---------------------------PROGRAMA------------------------------------------------------------

#la siguiente función copia las samples que queremos del dataset según número de samples y
clases (gestos)

for c in gestos:               #Para cada gesto que necesitamos
    file = open(csv_file, 'r')
    reader = csv.reader(file, delimiter=';')

    for line in reader: #leemos una línea del archivo .csv donde están los gestos con sus
referencias
        target = line[1]       #el segundo elemento de la linea es la clase (el gesto)
        ref = line[0]          #el primero es su referencia  (un número que es el nombre del
directorio donde está el gesto)

        if target == c:        #si la clase leída coincide con la que buscamos

            #no hace falta copiar, simplemente crear el .csv
            ondo=1
            if ondo == 1:    #si se copia bien
                contador += 1   #aumentamos el contador

                rows.append([ref, target]) #ponemos la referencia y clase en lista "rows"
            if contador == samples: # comprobamos tenemos las samples que queremos
                contador=0     #si es así, reiniciamos el contador y
                break        # finalizamos la búsqueda para seguir con la siguiente clase

random.shuffle(rows)  #la búsqueda y copia anterior la hemos realizado consecutivamente así
que aleatorizamos la lista
```

```
with open(path_destino+'train.csv', 'wt',newline='') as f:
#guardamos la lista en un archivo
    csv_writer = csv.writer(f, delimiter=';')
    csv_writer.writerows(rows)

rows = []


#la siguiente función copia las samples_val que queremos del dataset según número de
samples_val y clases (gestos)

for c in gestos:               #Para cada gesto que necesitamos
    file = open(csv_fle_val, 'r')
    reader = csv.reader(file, delimiter=';')

    for line in reader: #leemos línea del archivo .csv donde están los gestos con referencias
        target = line[1]       #el segundo elemento de la linea es la clase (el gesto)
        ref = line[0]          #el primero es su referencia  (un número que es el nombre del
directorio donde está el gesto)

        if target == c:        #si la clase leída coincide con la que buscamos


           #no hace falta copiar, simplemente crear el .csv
            ondo=1
            if ondo == 1:    #si se copia bien
                contador_val += 1   #aumentamos el contador_val

                rows.append([ref, target])   # ponemos referencia y clase en la lista "rows"
            if contador_val == samples_val:    #finalmente comprobamos si tenemos las
samples_val que queremos
                contador_val=0            #si es así, reiniciamos el contador_val y

                file.close
                break               # finalizamos búsqueda para seguir con la siguiente clase


random.shuffle(rows)

with open(path_destino_val+'validation.csv', 'wt', newline='') as f:
csv_writer = csv.writer(f, delimiter=';')
csv_writer.writerows(rows)
```

14.2 Módulo de entrenamiento

```
#Trabajo de Fin de Máster
#Autor: Jon San Martín Garaluce
#UPV-EHU

#HandRecon_Training.py
#Python 3.6

#--------------------------IMPORTS-------------------------------------------------------------

import numpy as np
import cv2
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

import os
import math
import pandas as pd
import matplotlib.image as img
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
tf.__version__

#----------------------------------INICIALIZACIONES-----------------------------------------

#training targets
targets = pd.read_csv('D:\\jon\\Documents\\training_samples3\\train.csv',header=None,sep =
";").to_dict()

# validation targets
targets_validation =
pd.read_csv('D:\\jon\\Documents\\validation_samples3\\validation.csv',header=None,sep =
";").to_dict()

# Las clases que queremos usar
targets_name = ['No gesture', 'Thumb Down', 'Thumb Up', 'Pulling Hand In', 'Pushing Hand
Away', 'Doing other things', 'Stop Sign', 'Swiping Up', 'Swiping Down', 'Swiping Left',
'Swiping Right']

nsalida = len(targets_name) #número de neuronas a la salida, coincide con el número de clases

nmuestras = 4000
tm_batch= 25
epocas = 12

nmuestras_totales= nmuestras*nsalida
nmuestras_totales=nmuestras_totales*2
pasos= nmuestras_totales/tm_batch
pasos_val=pasos/10
nmuestras_totales_val=nmuestras_totales/10

path = "D:\\jon\\Documents\\20bn-jester-v1\\"
path_cv = path
dirs = os.listdir(path)  #returns a list containing the names of the entries in the directory
given by path
dirs_cv = os.listdir(path_cv)

# number of samples for training and validation
len_dirs=len(dirs)-1 #longitud del directorio -1 = número carpetas, es decir, número samples
len_dirs_cv=len(dirs_cv)-1
#-------------------------------------------------------------------------------------------
```

```
#-------------------------------------FUNCIONES DIVERSAS---------------------------------

# return gray image
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140]) #Y' = 0.2989 R + 0.5870 G + 0.1140 B


max_frames = 30 # número máximo de frames (fotogramas)
# esta función unifica los frames para cada muestra de datos
def unificar_frames(path):
    # obtenemos las frames
    frames = os.listdir(path)
    frames_contador = len(frames)
    # unificamos
    if max_frames > frames_contador:
        # duplicamos el último frame si es más corto que lo máximo
        frames += [frames[-1]] * (max_frames - frames_contador)
    elif max_frames < frames_contador:
        #eliminamos los últimos frames si es más largo que lo máximo
        frames = frames[0:max_frames]
    return frames

# Resize frames
def resize_frame(frame):
    frame = img.imread(frame)
    frame = cv2.resize(frame, (64, 64))
    return frame

# Function to empty the RAM
def release_list(a):
   del a[:]
   del a

# Adjust training data
def proc_train_generator():
    counter_training = 0 # contador de muestras totales
    training_targets = [] # lista de clases (gestos)
    new_frames = [] # lista para almacenar los frames
    pos=0
    contador=0 # contador de muestras para nuestro generador

    for target_key in targets[0]:   #targets es un diccionario: en 0 tiene una lista de
target_keys y en 1 tiene una lista de targets
        #los target_keys son las referencias de cada video
        #los targets son los videos
        #por ello, vamos recorriendo cada video en el for
        pos+=1
        new_frame = [] # nueva muestra
        # Frames in each folder
        directory= str(targets[0][target_key]) #pasamos a string la referencia del directorio
        frames = unificar_frames(path+directory) #unificamos el video de ese directorio
(hacemos que tenga 30 frames)
        if len(frames) == max_frames: # comprobamos por si acaso

            for frame in frames: #para cada frame del video
                frame = resize_frame(path+directory+'/'+frame) #hacemos que todas tengan el
mismo tamaño de frame (64*64)

                new_frame.append(rgb2gray(frame)) #pasamos a escala de grises
                if len(new_frame) == 15: # partition en dos muestras de 15 y 15

                    new_frames.append(new_frame) # las metemos en la lista
                    contador+=1 #contamos una muestra más (generador)
                    training_targets.append(targets_name.index(targets[1][target_key]))
                    #accedo al target con esa target_key, miro qué clase es, y meto el nombre
de la clase en la lista de clases
                    counter_training +=1 #contamos una muestra TOTAL más
                    new_frame = []  #libero la lista new_frame
                if contador == 25:   #miro si he llegado al tamaño del batch
                    contador=0    # reseteo el contador
                    # convert training data to np float32
                    training_data = np.array(new_frames[0:counter_training], dtype=np.float32)
#paso a array Numpy de 32 bits
                    release_list(new_frames)  #libero la lista new_frames
```

```
                    yield training_data, training_targets  #termina iteración, devuelvo batch

                    new_frames = []
                    training_targets=[]


# LO MISMO PARA LA VALIDACIÓN
def proc_val_generator():
    counter_validation = 0
    pos=0
    cv_targets = []
    new_frames_cv = []
    contador_cv=0
    for val_key in targets_validation[0]:
        pos+=1
        #contador+=1
        new_frame = []
        # Frames in each folder
        directory= str(targets_validation[0][val_key])
        frames = unificar_frames(path_cv+directory)
        if len(frames)==max_frames:
            for frame in frames:
                frame = resize_frame(path_cv+directory+'/'+frame)
                new_frame.append(rgb2gray(frame))
                if len(new_frame) == 15:
                    new_frames_cv.append(new_frame)
                    cv_targets.append(targets_name.index(targets_validation[1][val_key]))
                    counter_validation +=1
                    contador_cv+=1
                    new_frame = []
                if contador_cv == 25:
                    contador_cv=0
                    # convert validation data to np float32
                    cv_data = np.array(new_frames_cv[0:counter_validation], dtype=np.float32)

                    release_list(new_frames_cv)

                    yield cv_data, cv_targets
                    new_frames_cv = []
                    cv_targets=[]

#-------------------------------------------------------------------------------------------------
#----------------------------------RED NEURONAL ARTIFICIAL--------------


# MI MODELO
class HandGestureReconModel(tf.keras.Model):
  def __init__(self):
    super(HandGestureReconModel, self).__init__()
    # Convoluciones
    self.conv1 = tf.compat.v2.keras.layers.Conv3D(32, (3, 3, 3), activation='relu',
name="conv1", data_format='channels_last')    #definimos cada capa
    self.sp_drop3D= tf.keras.layers.SpatialDropout3D(0.2,data_format='channels_last')
#después de la convolucional yo añado dropout para mejorar el overfiting
    self.pool1 = tf.keras.layers.MaxPool3D(pool_size=(2, 2, 2), data_format='channels_last')
#maxpooling
    self.conv2 = tf.compat.v2.keras.layers.Conv3D(64, (3, 3, 3), activation='relu',
name="conv2", data_format='channels_last')
    self.sp_drop3D_2= tf.keras.layers.SpatialDropout3D(0.3,data_format='channels_last')
    self.pool2 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')

    #self.conv3 = tf.compat.v2.keras.layers.Conv3D(128, (3, 3, 3), activation='relu',
name="conv3", data_format='channels_last')
    #self.sp_drop3D_3= tf.keras.layers.SpatialDropout3D(0.2,data_format='channels_last')
    #self.pool3 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')

    # LSTM y capa Flatten
    self.convLSTM =tf.keras.layers.ConvLSTM2D(40, (3, 3))
    self.flatten =  tf.keras.layers.Flatten(name="flatten")
```

```
    # Dense layers
    self.d1 = tf.keras.layers.Dense(128, activation='relu', name="d1")
    #Añado dropout
    self.drop_out = tf.keras.layers.Dropout(rate=0.4, name='drop_out')
    self.out = tf.keras.layers.Dense(nsalida, activation='softmax', name="output")

    #ATENTO la última capa tiene que tener tantas neuronas como clases


  def call(self, x):
    x = self.conv1(x)
    x = self.sp_drop3D(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.sp_drop3D_2(x)
    x = self.pool2(x)

    #x = self.conv3(x)
    #x = self.sp_drop3D_3(x)
    #x = self.pool3(x)


    x = self.convLSTM(x)
    #x = self.pool2(x)
    #x = self.conv3(x)
    #x = self.pool3(x)
    x = self.flatten(x)
    x = self.d1(x)
    x = self.drop_out(x)
    return self.out(x)


modelo = HandGestureReconModel()    #creamos el modelo


#loss and optimizer methods  (compilamos)
modelo.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(),
              metrics = ['accuracy'])
#-----------------------------------------------------------------------------------------



#----------------------------------ENTRENAMIENTO-----------------------------------------------
#----------------------------------------------------------------------------------------------
def train_generator(trn):
    trn_counter=0
    if trn==1 :  #entrenamiento activado
        trn_counter= nmuestras_totales  #muestras totales
    elif trn==0: #validación activada
        trn_counter= nmuestras_totales_val

    while True:

        if trn==1 and trn_counter== nmuestras_totales:
            gen=proc_train_generator() #si estamos entrenando, crear generador para procesar
                                       #datos de entrenamiento
            trn_counter=0
        elif trn==0 and trn_counter== nmuestras_totales_val:  #si estamos validando, crear
generador para procesar
                                           #datos de validación
            gen=proc_val_generator()
            trn_counter=0
        training_data, training_targets = next(gen)  #siguiente batch
```

```
        # Normalización

        scaler = StandardScaler() #creamos un objeto de la clase
sklearn.preprocessing.StandardScaler
        scaled_images  = scaler.fit_transform(training_data.reshape(-1, 15*64*64))
        #transformamos a un array de una dimensión y estandarizamos
        scaled_images  = scaled_images.reshape(-1, 15, 64, 64, 1) #devolvemos la forma

        #pasamos a un array Numpy óptimo para pasarlo a la red
        x_train = np.array(scaled_images)
        y_train = np.array(training_targets)

        # damos el batch a la función de entrenamiento
        yield x_train, y_train
        trn_counter+= tm_batch #incrementamos un tamaño de batch cada vez


train=train_generator(1)
val=train_generator(0)

# A continuación cargamos pesos si queremos continuar el entrenamiento.
filepath_model = "model_salvado"
respuesta= input("¿Continuar entrenamiento? s/n")

if respuesta=="s" :
    modelo.load_weights("D:\\jon\\Documents\\model_salvado")



checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath_model, monitor='loss', verbose=1,
save_best_only=True, mode='min')
#si una época mejora los resultados de la anterior, guardamos el modelo ---> save_best_only=
True
callbacks_list = [checkpoint]
history = modelo.fit_generator(train, steps_per_epoch=pasos, epochs = epocas,
                        callbacks = callbacks_list,
                        validation_data=val,
                        validation_steps=pasos_val)


history.history
```

14.3 Módulo de testeo y uso

```
#Trabajo de Fin de Máster
#Autor: Jon San Martín Garaluce
#UPV-EHU

#HandRecon_Test.py
#Python 3.6

#--------------------------IMPORTS-------------------------------------------------------------

import tensorflow as tf
print(tf.__version__)
import numpy as np
import cv2
from sklearn.preprocessing import StandardScaler

#--------------------------------INICIALIZACIONES---------------------

clases = ['No gesture', 'Thumb Down', 'Thumb Up', 'Pulling Hand In', 'Pushing Hand Away',
'Doing other things', 'Stop Sign', 'Swiping Up', 'Swiping Down', 'Swiping Left', 'Swiping
Right']

nsalida = len(clases)

#--------------------------------MODELO--------------------------------
class HandGestureReconModel(tf.keras.Model):
  def __init__(self):
    super(HandGestureReconModel, self).__init__()
    # Convoluciones
    self.conv1 = tf.compat.v2.keras.layers.Conv3D(32, (3, 3, 3), activation='relu',
name="conv1", data_format='channels_last')
    self.sp_drop3D= tf.keras.layers.SpatialDropout3D(0.2,data_format='channels_last')
    self.pool1 = tf.keras.layers.MaxPool3D(pool_size=(2, 2, 2), data_format='channels_last')
    self.conv2 = tf.compat.v2.keras.layers.Conv3D(64, (3, 3, 3), activation='relu',
name="conv1", data_format='channels_last')
    self.sp_drop3D_2= tf.keras.layers.SpatialDropout3D(0.3,data_format='channels_last')
    self.pool2 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')

    #self.conv3 = tf.compat.v2.keras.layers.Conv3D(128, (3, 3, 3), activation='relu',
name="conv3", data_format='channels_last')
    #self.sp_drop3D_3= tf.keras.layers.SpatialDropout3D(0.2,data_format='channels_last')
    #self.pool3 = tf.keras.layers.MaxPool3D(pool_size=(2, 2,2), data_format='channels_last')

    # LSTM y capa Flatten
    self.convLSTM =tf.keras.layers.ConvLSTM2D(40, (3, 3))
    self.flatten =  tf.keras.layers.Flatten(name="flatten")

    # Dense layers
    self.d1 = tf.keras.layers.Dense(128, activation='relu', name="d1")
    #Añado dropout
    self.drop_out = tf.keras.layers.Dropout(rate=0.5, name='drop_out')
    self.out = tf.keras.layers.Dense(nsalida, activation='softmax', name="output")

    #La última capa tiene que tener tantas neuronas como clases
def call(self, x):
    x = self.conv1(x)
    x = self.sp_drop3D(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.sp_drop3D_2(x)
    x = self.pool2(x)

    #x = self.conv3(x)
    #x = self.sp_drop3D_3(x)
    #x = self.pool3(x)

    x = self.convLSTM(x)
    #x = self.pool2(x)
    #x = self.conv3(x)
    #x = self.pool3(x)
    x = self.flatten(x)
    x = self.d1(x)
```

```
    x = self.drop_out(x)
    return self.out(x)

#%%-----------------------
nuevo_modelo = HandGestureReconModel()   #creamos modelo
#%%-----------------------

#%%  #cargamos los pesos
nuevo_modelo.load_weights("D:\\jon\\Documents\\model_salvado")



#-------------------------------------------------------------TEST-----------------------------

#%%
video = []
capturando = cv2.VideoCapture(0)    #Comenzamos a capturar por la webcam
clase =''

while(True):
    # Capturamos cada imagen
    ret, frame = capturando.read()
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) #pasamos las imágenes a gris
    video.append(cv2.resize(gray_frame, (64, 64))) #la reescalamos y metemos en el array

    if len(video) == 30:
        video_predecir = np.array(video, dtype=np.float32)    #lo pasamos a array adecuado
Numpy de 32 bits
        video_predecir = video_predecir.reshape(-1, 30, 64, 64, 1)     #reescalamos otra vez
        prediccion = nuevo_modelo.predict(video_predecir)       #predecimos
        clase = clases[np.argmax(prediccion)]                #qué clase es

        print('clase = ',clase, 'Precision = ', np.amax(prediccion)*100,'%')  #imprimimos la
predicción por consola
        video = []   #vaciamos

    cv2.putText(frame, clase, (30, 85), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0,
0),1,cv2.LINE_AA)      #ponemos la predicción en el video

    # Mostramos las imágenes por pantalla
    cv2.imshow('Reconocimiento de Gestos Manuales',frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Cuando terminemos, destruimos ventanas y demás
capturando.release()
cv2.destroyAllWindows()
```