UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

CLASIFICACIÓN DE CLIPS DE VÍDEO MEDIANTE SOLUCIONES BASADAS EN DEEP LEARNING: ESTUDIO PRÁCTICO SOBRE EL TRATAMIENTO DE LA DIMENSIÓN TEMPORAL POR REDES NEURONALES

Francisco Javier Sanguino Bautiste

Madrid, Junio 2020

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO DE FIN DE GRADO

Título: Clasificación de clips de vídeo mediante soluciones basadas en Deep Learning
Autor: D. Francisco Javier Sanguino
Tutor: D. Narciso García
Departamento: Señales, Sistemas y Radiocomunicaciones (SSR)

MIEMBROS DEL TRIBUNAL

Presidente:

Vocal:

Secretario:

Suplente:

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de .....

Madrid, a ..... de ..... de 2020

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

CLASIFICACIÓN DE CLIPS DE VÍDEO MEDIANTE SOLUCIONES BASADAS EN DEEP LEARNING: ESTUDIO PRÁCTICO SOBRE EL TRATAMIENTO DE LA DIMENSIÓN TEMPORAL POR REDES NEURONALES

Francisco Javier Sanguino Bautiste

Madrid, Junio 2020

Agradecimientos

Agradezco al profesor Frank Wang, de la National Taiwan University, su orientación en mis primeros pasos en el mundo del Aprendizaje Profundo.

También me gustaría agradecerle a Gloria Ayllón su ayuda en la confección de las imágenes de este trabajo.

Por último, agradezco a Narciso García, tutor del trabajo, su orientación y dedicación.

Resumen

El Aprendizaje Profundo ha revolucionado la computación en los últimos años. El avance de los procesadores ha posibilitado el entrenamiento y ejecución de estos algoritmos, basados en Redes Neuronales. Concretamente, el campo de la Visión Artificial se ha visto beneficiado de este tipo de algoritmos debido a la equivalencia entre las Redes Neuronales Convolucionales y los filtros. En estos algoritmos, la Red Neuronal aprende los parámetros que mejor se ajustan a los datos de entrada. Un problema que se puede resolver con estas técnicas es la clasificación de clips de un vídeo. Sin embargo, en los vídeos surge la cuestión del tratamiento de la dimensión temporal.

El objetivo de este trabajo es comparar el rendimiento de dos tipos de Redes Neuronales (Convolucionales y Recurrentes) en el tratamiento de la dimensión temporal. Para ello, se ha utilizado un conjunto de datos de naturaleza secuencial, el *Breakfast Dataset*, que muestra recetas de cocina con etiquetas de acciones como *coger* o *pelar*. Se ha seguido una metodología lo más exhaustiva posible, proponiendo estrategias para reducir el tiempo de entrenamiento de las redes. Ambas redes se componen de una red troncal que extrae características de los cuadros de los vídeos. Estas características se introducen en una Red Neuronal de cada tipo para fusionar los distintos cuadros y tratar la dimensión temporal.

Tras la utilización de diversas técnicas como el cálculo de matrices de confusión o la representación de los resultados gracias al algoritmo t-SNE, se concluyó que la Redes Neuronales Recurrentes tienen un $10\,\%$ más de precisión que las Convolucionales con un tiempo de procesamiento similar.

Summary

Deep Learning has transformed computer science in the last few years. The increase in computational power has enabled the application of this kind of algorithms, based on Neuronal Networks. Specifically, Computer Vision has used these techniques because Convolutional Neuronal Networks are equivalent to filters. A problem that can be solved using Deep Learning is video classification. Nevertheless, the issue of processing a temporal dimension appears in videos.

The goal of this piece of research is to compare the performance of two types of Neuronal Networks (Convolutional and Recurrent) to process the temporal dimension. To archive, this goal, a dataset with a sequential nature has been used (Breakfast Dataset), which shows videos of cooking recipes with labels such as *take* or *peel*. An exhaustive methodology has been followed, proposing strategies to decrease the training time of networks. Both networks are formed by a backbone that extracts features from video frames. These features are introduced in a Neuronal Network of each type to fusion the frames and process the temporal dimension.

After using some techniques like confusion matrixes or data visualization with the t-SNE algorithm, the conclusion was that Recurrent Neuronal Networks have $10\,\%$ more of accuracy with similar processing time.

Palabras Clave

Aprendizaje profundo, clasificación de vídeo, Redes Neuronales Convolucionales, Redes Neuronales Recurrentes, algoritmo supevisado.

Key Words

Deep Learning, video classification, Convolutional Neuronal Networks, Recurrent Neuronal Networks, supervised algorithm.

Índice

1. Introducción

El *Aprendizaje Profundo* (*Deep Learning* en inglés) ha supuesto una revolución en el tratamiento de datos. Las *Redes Neuronales* (NN por sus sigles en inglés, Neuronal Networks) proporcionan la arquitectura básica de este tipo de algoritmos. Estas redes, mediante técnicas de aprendizaje automático, son capaces de cambiar según los datos que se le proporcionan en un proceso que se denomina entrenamiento. Por ello, estos algoritmos se enmarcan dentro del campo de la *Inteligencia Artificial*, ya que el ordenador aprende a resolver un problema específico gracias a estos datos con una mínima intervención humana.

Con la aplicación de técnicas de *Deep Learning*, el paradigma de diseño de algoritmos ha cambiado completamente, pudiéndose establecer un símil con la manera en la que los humanos obtienen conocimiento. Antes, los ordenadores basaban su funcionamiento en aplicar unos pasos previamente definidos por un humano, algo parecido a cuando un experto explica conceptos. Ahora, basta con proporcionarle una gran cantidad de datos a estas redes para resolver problemas similares, asemejándose a cuando las personas aprendemos mediante la experiencia.

No obstante, no conviene pensar que estas técnicas son la panacea y basta con recoger unos datos para meterlos en una caja negra que resuelve el problema sola. Como en cualquier conocimiento obtenido de la experimentación, hay que realizar multitud de experimentos variando algunos parámetros para mejorar la actuación de las redes. Una metodología exhaustiva y buen conocimiento teórico sobre lo que está sucediendo en las redes neuronales debe ser imprescindible para diseñar este tipo de algoritmos.

Un área que se ha visto muy beneficiada con la aplicación de estas técnicas de *Deep Learning* es la *Visión Artificial* (*Computer Vision* en inglés) debido a la equivalencia de las *Redes Neuronales Convolucionales* con los filtros. Por ello, las redes neuronales se han demostrado muy efectivas a la hora de tratar las características espaciales de las imágenes. Los resultados obtenidos con estos algoritmos en los últimos años son excelentes e incluso hay investigadores que se dedican sólo a estudiar sobre este tipo de tecnologías. Además, los congresos de procesado de imágenes se han llenado de artículos que aplican estos algoritmos.

Sin embargo, a los sistemas de vídeo se le añade el problema de su tamaño, que complica su procesamiento en tiempos razonables. Las Redes Neuronales que trabajan con vídeos tardan un tiempo sustancialmente mayor en entrenarse. Además, en la utilización de algoritmos de *Deep Learning* para su procesado, ha surgido una cuestión: cómo tratar la nueva dimensión temporal que aparece, de una naturaleza completamente distinta a las espaciales que encontrábamos en las imágenes. En los últimos años se han propuesto varias técnicas entre las que destacan la aplicación de las Redes Neuronales Convolucionales, que tratan la dimensión temporal de manera parecida a la espacial y las Redes Neuronales Recurrentes, que son una novedad en el campo de la *Visión Artificial*.

El objetivo de este trabajo es explorar esta problemática, resolviendo un problema básico en los vídeos: su clasificación. Además, se tratará de utilizar una metodología similar a la que seguiría un ingeniero que debe resolver este problema. Por ello, se han escogido un conjunto de vídeos que muestran a personas preparando recetas que podrían haber sido recopilados en cualquier cocina para ilustrar la metodología en un problema del mundo real. También se propondrán estrategias que disminuyan el tiempo de entrenamiento de las redes.

Más concretamente, se diseñará un sistema que asigne etiquetas a un corte de vídeos de cocina según las tareas que se están realizando durante la receta: cortar, coger, poner, añadir, etc. Para ello, tras preprocesar los cuadros en una red neuronal preentrenada que se utiliza para clasificación de imágenes con el objetivo de extraer algunas características, se entrenarán de principio a fin dos redes neuronales para tratar la dimensión temporal, una basada en Redes Neuronales Convolucionales y la otra Redes Neuronales Recurrentes. De esta forma, se podrán comparar los resultados en ambas.

La metodología que he seguido en este trabajo consiste, por tanto, en el diseño de dos soluciones para el mismo problema. Para cada uno de los diseños, se ha seguido el mismo procedimiento. En

primer lugar, se han recogido los datos, que en nuestro caso es un conjunto de vídeos recogido por la Universidad de Brown llamado *Breakfast Dataset*. En él se muestran actores preparando distintas recetas de desayuno. Se ha escrito un pequeño programa para preprocesar los vídeos y adaptarlos al problema a resolver. En segundo lugar, se programaron con la librería *Pytorch* las arquitecturas de ambas redes y su entrenamiento. Precisamente, el resultado del entrenamiento son los distintos modelos aprendidos por el ordenador que mejor se ajustan a los datos proporcionados.

Los resultados que he obtenido muestran que conviene usar Redes Neuronales Recurrentes para el tratamiento de la dimensión temporal que aparece en los vídeos con respecto a las imágenes. La diferencia de precisión de un sistema basado en RNN aumentaba más de un $10\,\%$ con respecto a uno basado en CNN. Además, se han establecido y explotado algunas estrategias para que el entrenamiento de las redes basadas en vídeos bajasen de aproximadamente 100 horas a menos de 40 minutos.

Debido a que casi toda la bibliografía que tiene que ver con estos temas es en inglés e, incluso, muchas veces en artículos en castellano se utilizan esas mismas palabras en su forma inglesa, a lo largo de este trabajo se introducirá el término en castellano y se indicará su traducción en inglés. Su uso será en castellano o inglés independientemente aunque se mantendrán las siglas en inglés por su utilización en castellano es prácticamente nulo.

1.1. Objetivos

Los objetivos que he planteado en este trabajo han sido:

1. Explorar la metodología que se debe de seguir para diseñar un sistema basado en *Deep Learning*.
2. Proponer algunas estrategias para disminuir el tiempo de entrenamiento de las Redes Neuronales que procesan vídeos.
3. Diseñar dos sistemas, uno basado en Redes Neuronales Convolucionales y el otro basado en Redes Neuronales Recurrentes con el fin de comparar dos sistemas cuál es la manera más efectiva de tratar el problema de la dimensión temporal de los vídeos y establecer.
4. Profundizar en el entrenamiento de Redes Neuronales en *Python* usando la librería *Pytorch* y ser capaz de manejarse en su documentación.
5. Profundizar en mis conocimientos de Deep Learning y ser capaz de explicar el funcionamiento de las Redes Neuronales diseñando además imágenes propias que ilustren la explicación produciendo un material en castellano a una temática que tradicionalmente está en inglés.

2. Estado del Arte

En esta sección, tras una breve introducción, se explicará la base teórica de las redes neuronales y el funcionamiento de las Redes Neuronales Convolucionales y Recurrentes, elementos básicos de este trabajo. También se introducirán las herramientas de análisis que se han utilizado. Más tarde, se tratarán los últimos avances en la clasificación de vídeo.

2.1. Introducción

Las ramas de Inteligencia Artificial y Visión Artificial (*Computer Vision* por su nombre en inglés) han estado intrínsecamente relacionadas desde su nacimiento. Cuando se plantearon los primeros estudios sobre máquinas inteligentes en la década de 1970, muchos investigadores creyeron que la perspectiva más acertada era enseñar a esas máquinas por medio del sentido de la vista. Es decir, el objetivo era imitar el aprendizaje de un bebé en los ordenadores [22, Pág. 11].

Rápidamente, los investigadores se dieron cuenta de que surgían dos problemas. El primero de ellos, y más evidente, era que en esa época no había un conjuntos de datos (*Dataset* en inglés) de vídeo lo suficientemente grande para compararse con la cantidad de horas que un niño puede recibir información a través de la vista durante los primeros años de su vida.

El segundo lo experimentó el pobre estudiante de grado Gerald Jay Sussman cuando, en 1966, le fue asignada la tarea de "pasarse el verano intentando conectar una cámara con el ordenador y conseguir que la máquina describiese lo qué veía-por el profesor Marvin Misky del MIT [22, Pág. 11]. Ahora se sabe que el problema tenía alguna implicación más de la que previó su profesor y era algo más complicado que lo que hubiese podido hacer en el verano. En esa época, no había potencia computacional suficiente ni se disponía de las técnicas adecuadas. En este trabajo se tratará, en parte, esta tarea al clasificar vídeos en función de la actividad que ocurre en ellos.

Después de la comprobación de que la Visión por Computadora debía ir más allá que intentar imitar la inteligencia humana, los caminos de estos dos campos se separan irremediablemente. En la actualidad, y tras cambiarse totalmente la perspectiva desde la que nos acercamos a ambas áreas, han vuelto a unirse al comprobarse que los últimos avances en Inteligencia Artificial (las Redes Neuronales) funcionan muy bien en el campo de la Visión por Ordenador.

2.2. Aprendizaje Profundo

Normalmente, en los algoritmos que procesan cualquier tipo de datos, lo que se busca es una transformación de los datos para que su representación se adecue al problema que se intente resolver.

Se llama Aprendizaje Profundo (*Deep Learning*) a las técnicas propias del Aprendizaje Automático (*Machine Learning*) que sustituyen la obtención de características mediante métodos tradicionales por métodos en los que el ordenador aprende cuál es la característica que mejor se adapta a los datos usados. Por tanto, las capas de las que se compone el sistema no están diseñadas por humanos, si no que se construyen usando procedimientos de aprendizaje.

Es por ello por lo que los algoritmos de *Deep Learning* se han adaptado tan bien al campo de la Visión Artificial. La mayoría de algoritmos de Visión Artificial tratan de extraer una serie de características procesando los datos (píxeles normalmente) para resolver la tarea indicada. Por ejemplo, mediante filtros en una imagen se pueden detectar los bordes de los objectos. La aplicación en sistemas de *Deep Learning* es inmediata, porque ellos también extraen características intermedias.

Los algoritmos de *Deep Learning* se dividen en supervisados y no supervisados. En los supervisados, los datos utilizados están etiquetados, esto es, tienen asociado otro dato que es lo que el sistema debería de calcular si se utilizara como entrada. Se utilizan por ejemplo para tareas de clasificación. En cambio,

en los no supervisados no existen una etiqueta, por ello, se utilizan para problemas en los que se necesite agrupar los datos. En este trabajo se cubrirán los supervisados.

2.2.1. Redes Neuronales (NN)

Los algoritmos de *Deep Learning* basan su funcionamiento en estructuras denominadas Redes Neuronales (*Neuronal Networks*, NN, en inglés). La arquitectura estas redes se puede ver en la figura 1. En el sistema, los círculos representan datos. Cada fila de datos podría entenderse como un vector. Por simplificar la explicación, se ha optado por reducir la dimensión de los datos de una matriz en 2D (imágenes) por un vector en 1D, en la siguiente sección se abstraerá para sistemas en 2D.

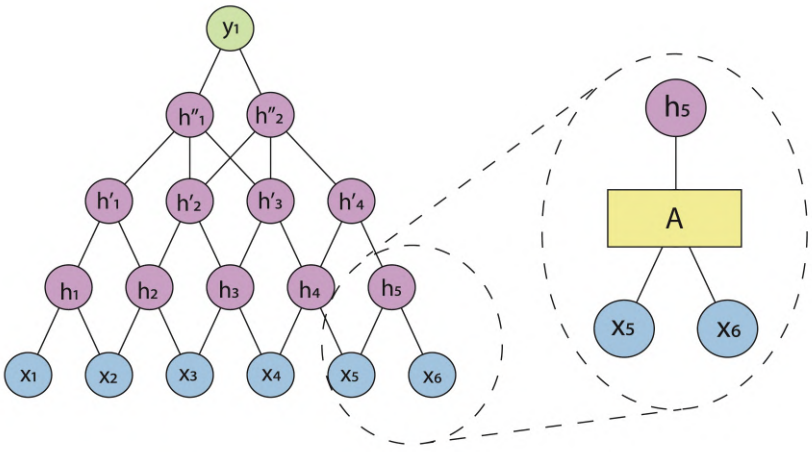


Figura 1 – Ejemplo de una arquitectura de una red neuronal

El sistema de la figura 1 se compone de varios vectores. Está el vector de entrada $\boldsymbol{x}$ y el vector de salida $\boldsymbol{y}$. También están los vectores $\boldsymbol{h}$, $\boldsymbol{h'}$, $vech''$, que son las características ocultas (*hidden features* en inglés), datos intermedios. Cada dato calculado depende de los valores que llegan a él usando la función A. Por ejemplo, h_5 depende de x_5 y x_6.

La función A es la clave de los algoritmos de *Deep Learning*. Se compone de un función lineal y una operación no lineal sobre cada componente del vector. En la figura 1 se podrían calcular los datos de la siguiente manera:

$$\begin{aligned} h_1 &= A(x_1, x_2) = \sigma(\omega_1 x_1 + \omega_2 x_2) \\ h_2 &= A(x_2, x_3) = \sigma(\omega_3 x_2 + \omega_4 x_3) \\ h_3 &= A(x_3, x_4) = \sigma(\omega_5 x_3 + \omega_6 x_4) \\ h_4 &= A(x_4, x_5) = \sigma(\omega_7 x_4 + \omega_8 x_5) \\ h_5 &= A(x_5, x_6) = \sigma(\omega_9 x_5 + \omega_{10} x_6) \end{aligned}$$

Que se puede representar como $\boldsymbol{h} = \sigma(W\boldsymbol{x})$, siendo $W\boldsymbol{x}$ la aplicación lineal y σ la función no lineal:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \end{bmatrix} = \sigma \left(\begin{bmatrix} \omega_1 & \omega_2 & 0 & 0 & 0 & 0 \\ 0 & \omega_3 & \omega_4 & 0 & 0 & 0 \\ 0 & 0 & \omega_5 & \omega_6 & 0 & 0 \\ 0 & 0 & 0 & \omega_7 & \omega_8 & 0 \\ 0 & 0 & 0 & 0 & \omega_9 & \omega_{10} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \right)$$

Como se puede ver, los pesos tienen valores distintos. Para el caso general, con $\boldsymbol{h}$ de tamaño m y $\boldsymbol{x}$ de tamaño n:

$$\begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_m \end{bmatrix} = \sigma \left(\begin{bmatrix} \omega_{1,1} & \omega_{1,2} & \dots & \omega_{1,n} \\ \omega_{2,1} & \omega_{2,2} & \dots & \omega_{2,n} \\ \vdots & & \ddots & \\ \omega_{m,1} & \omega_{m,2} & \dots & \omega_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \right)$$

A la función σ se le llama función de activación (*activation function* en inglés) y, como las neuronas biológicas, es una forma de saber si esa neurona (los datos representados por círculos en nuestros ejemplos) está activada o no. Matemáticamente es una función no lineal monótonamente no decreciente que se aplica en cada componente del vector. Para desarrollos teóricos se suele usar una sigmoide, sin embargo, computacionalmente para la Visión Artificial se utiliza función ReLu, ya que se ahorra bastante capacidad y los resultados no son significativamente peores como se demostró en [4].

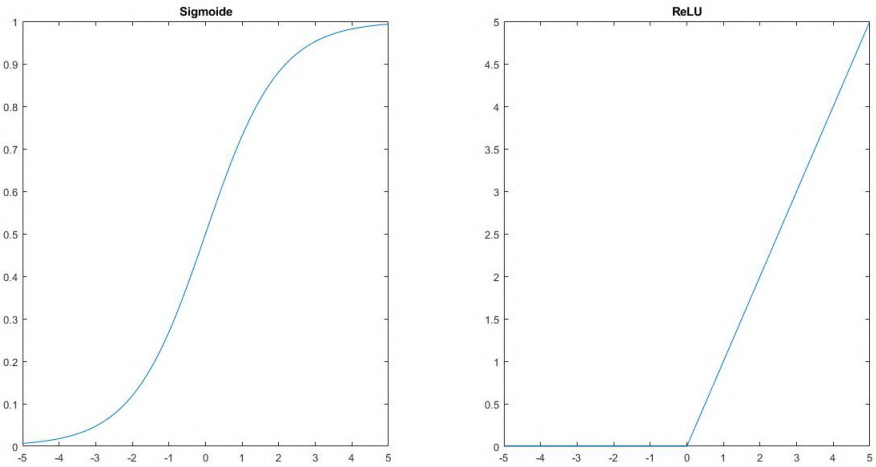


Figura 2 – Sigmoide vs. ReLU

Lo explicado hasta ahora se corresponde con las operaciones a las que se somete al vector de entrada para calcular la salida (*forward path* en inglés). Sin embargo, la potencia de este tipo de algoritmos radica en que son capaces de calcular los pesos (ω) que mejor se adaptan a los datos de entrada. Por tanto, el diseño del sistema no basta en escoger la función A, la construcción del sistema se divide en varias fases. A continuación se detalla el procedimiento para algoritmos de aprendizaje supervisado por ser más pertinentes en este trabajo.

1. **Recoger Datos**. Estos algoritmos deben estar basados en una cantidad enorme de datos. Antes de empezar a diseñar el propio sistema, se deben reunir muchos datos (imágenes, vídeos o datos estadísticos) etiquetados correspondientemente con el problema que intentamos resolver. El conjunto de estos datos se denomina *Dataset*.

2. **Entrenamiento**. En primer lugar, se diseña la arquitectura de la red (función A en el ejemplo anterior) para después introducir datos en ella. Se trata de escoger qué función no lineal se va a escoger y qué elementos serán distintos de 0 en la matriz W. Tras pasar los datos de entrada a través de la red inicializando los pesos ω (de manera aleatoria generalmente), se compara la salida de la red con la etiqueta correspondiente y mediante técnicas de retropropagación (*backpropagation* en inglés), el ordenador calcula nuevos valores de los pesos de las matrices W, en los distintos niveles. Estos nuevos valores de ω son los que se utilizarán para la siguiente iteración, cuando se procese

el siguiente dato. Tras repetir estas operaciones con todos los datos varias veces, obtendremos los valores definitivos de W.

3. **Uso**. Cuando el entrenamiento se ha completado, ya se puede usar el sistema para producir salidas a partir de entradas que no están en el propio *Dataset*. Cabe destacar que el algoritmo depende íntegramente de los datos del *Dataset* utilizado, si utilizásemos otro *Dataset* los resultados en esta parte podrían ser completamente distintos aunque mantuviésemos la arquitectura.

De esta forma, lo que se está haciendo matemáticamente es resolver las incógnitas ω. Para ello, se utilizan técnicas basadas en la optimización. Uno de los primeros artículos en usar estos métodos para redes neuronales fueron *LeCun et al.* en [13]. El algoritmo se basa procesar algunos datos de entrada en la red y computar el error entre la salida y la etiqueta. Como se muestra en la figura 3(b), se calcula el gradiente de la función error con respecto a la salida y. En el ejemplo de la imagen, simplemente se resta la predicción (y_i) menos el valor de la etiqueta (t_i), pero hay distintas técnicas para hacerlo, atendiendo a las diferentes tareas que pueden resolver las redes neuronales. Aplicando la regla de cadena, se pueden hallar los gradientes para cada capa anterior de la red.

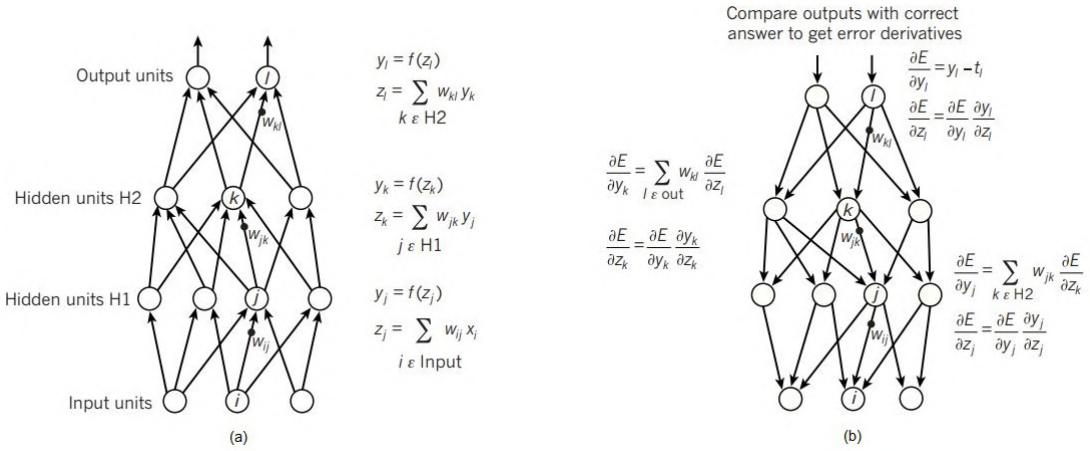


Figura 3 – *Forward path* en (a) y *Backpropagation* en (b). Obtenido de [12]

Una vez se han calculado los gradientes, se actualizan los pesos ω intentando encontrar el mínimo de la función de coste. El gradiente es muy útil, pues permite modificar el valor de ω en la dirección en la que se haya el mínimo. En la imagen 4, se muestra este procedimiento muy simplificado. En (a), hay una gráfica en la que la función error (eje y) depende de solo un parámetro y se ha calculado su gradiente en el valor de ω original (eje x). Esto da información sobre hacia donde hay que mover el ω para alcanzar el mínimo. Si $\frac{dE}{d\omega}(\omega^i) < 0$ hacia la derecha y hacia la izquierda en caso contrario. El salto entre el anterior valor (ω^1) y el nuevo (ω^2 en (b)), ilustrado con una flecha en las figuras (a) y (b) es proporcional a un parámetro llamado tasa de aprendizaje (*learning rate* en inglés) y al gradiente.

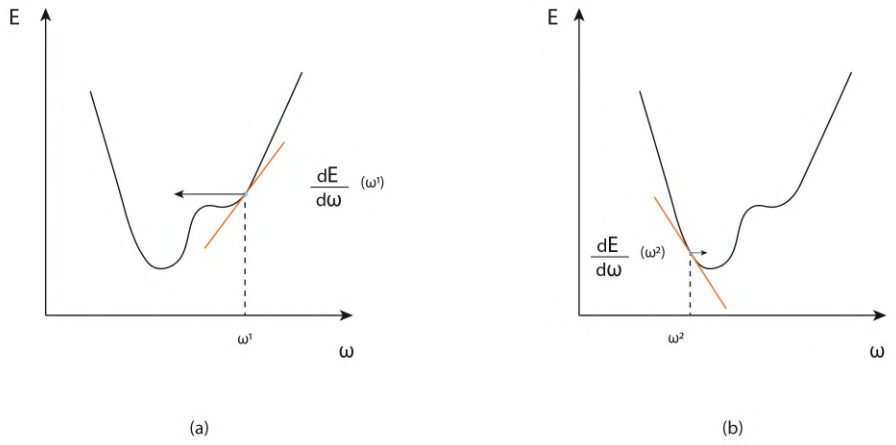


Figura 4 – Valor de un parámetro frente a la función error

Las Redes Neuronales normalmente tienen más de un parámetro, por tanto, la gráfica mostrada en la figura 4 tendría tantas dimensiones como parámetros tenga la red. No obstante, el algoritmo trabaja de la misma manera.

Esta técnica para encontrar el mínimo llamada descenso por gradiente (*gradient descent* en inglés) y da lugar a varios algoritmos que consiguen encontrar el mínimo sobre la función de coste llamados optimizadores. Algunos de los más importantes usados en *Deep Lerning* son el Adam y el SGD (*Stochastic Gradient Descent*). Su uso está recomendado según el tipo de problema que se intente resolver.

Además, cada uno de estas optimizadores definen una serie de parámetros asociados a sus algoritmos como la tasa de aprendizaje (*learning rate* en inglés) o la caída de peso (*weight decay* en inglés). La buena elección de estos parámetros es fundamental para un correcto entrenamiento de la red. Si, por ejemplo, la tasa de aprendizaje es muy pequeña, el valor de ω puede caer en un mínimo local. Sin embargo, con una tasa de aprendizaje grande, el valor de ω oscilaría sin llegar nunca a un mínimo.

Como forma de la gráfica de la figura 4 está oculta para el usuario, lo que se suele hacer para escoger estos parámetros es mostrar el valor de la función de coste en función del número de iteraciones del entrenamiento, gráfica que, por razones evidentes, debe ser siempre descendente. También se suele comenzar con un valor razonable (alrededor de $0,001$ para la tasa de aprendizaje) y se va modificando para estudiar el comportamiento de la red.

Otra elección en el diseño de la red y que depende del problema que se vaya a resolver es el cálculo de la función de coste, tal vez la más famosa sea la asociada al método de mínimos cuadrados para problemas de regresión. Para problemas de clasificación hay varias opciones pero la utilizada normalmente con *pytorch* es la de entropía cruzada.

En resumen, lo que hace la etapa de entrenamiento en una Red Neuronal es ajustar un modelo a unos datos, parecido a una regresión lineal pero usando técnicas más complejas debido a que el modelo es también bastante más complejo que una recta.

2.2.2. Redes Neuronales Convolucionales (CNN)

Una de las redes más utilizadas en el campo de la Visión Artificial (y en muchos otros) son las redes convolucionales (CNN por sus siglas en inglés *Convolutional Neuronal Networks*).

Realmente lo que ocurre en estas redes es que los pesos con los que se conecta una neurona con la siguiente se repiten periódicamente y su operación puede recordar a cómo funciona un filtro. Los valores de cada filtros son los que determina el algoritmo de aprendizaje. En el siguiente ejemplo se puede ver su funcionamiento:

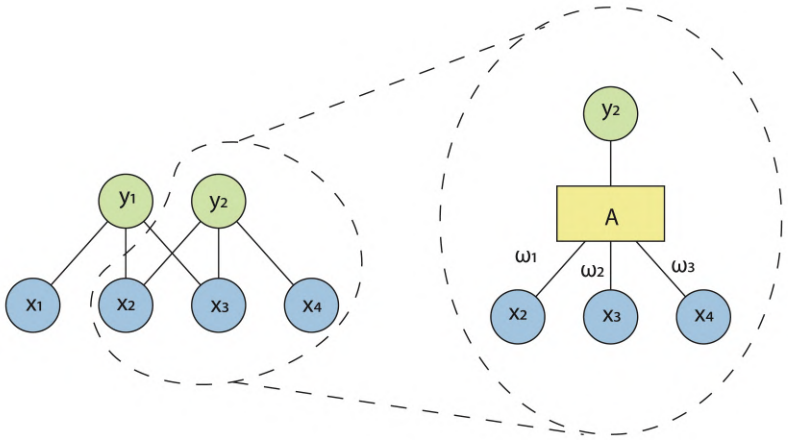


Figura 5 – Ejemplo de arquitectura de una CNN

En este tipo de red se define A como:

$$\begin{aligned} y_1 &= A(x_1, x_2, x_3) = \sigma(\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + b) \\ y_2 &= A(x_2, x_3, x_4) = \sigma(\omega_1 x_2 + \omega_2 x_3 + \omega_3 x_4 + b) \end{aligned}$$

Como se puede ver, es igual que una Red Neuronal normal pero con los pesos repetidos. Se añade el vector $\boldsymbol{b}$ (sesgo, *bias* en inglés) para tener en cuenta tendencias. Las componentes del vector $\boldsymbol{b}$ también se aprenden. Realmente, como en el caso anterior, podría ser definido como una relación matricial con la ecuación $\boldsymbol{y} = \sigma(W\boldsymbol{x})$:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \sigma\left(\begin{bmatrix} \omega_1 & \omega_2 & \omega_3 & 0 \\ 0 & \omega_1 & \omega_1 & \omega_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$

Generalizando para n entradas y m salidas, siendo i el número de valores de la entrada de los que depende cada dato de salida:

$$\begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_m \end{bmatrix} = \sigma\left(\begin{bmatrix} \omega_1 & \omega_2 & \cdots & \omega_i & 0 & \cdots & 0 \\ 0 & \omega_1 & \omega_2 & \cdots & \omega_i & \cdots & 0 \\ \vdots & & & & & & \\ 0 & \cdots & 0 & \omega_1 & \cdots & \omega_{i-1} & \omega_i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_4 \end{bmatrix} \right)$$

Siguiendo con el ejemplo, esta operación recuerda a una convolución. Efectivamente, como en las filas de la matriz W se repiten valores, la operación $W\boldsymbol{x}$ es equivalente a la convolución $\boldsymbol{\omega} * \boldsymbol{x}$, siendo un $\boldsymbol{\omega}$ un vector con los pesos $\omega_1, \omega_2, \omega_3$.

En la siguiente imagen se puede ver la relación entre la operación convolución ($\boldsymbol{\omega} * \boldsymbol{x}$) y la matricial ($W\boldsymbol{x}$).

$$\boldsymbol{\omega} * \boldsymbol{x} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \omega_1 x_1 \\ \omega_2 x_1 + \omega_1 x_2 \\ \omega_3 x_1 + \omega_2 x_2 + \omega_1 x_3 \\ \omega_3 x_2 + \omega_2 x_3 + \omega_1 x_4 \\ \omega_3 x_3 + \omega_2 x_4 \\ \omega_3 x_4 \end{bmatrix} = \begin{bmatrix} \omega_1 & 0 & 0 & 0 \\ \omega_2 & \omega_1 & 0 & 0 \\ \omega_3 & \omega_2 & \omega_1 & 0 \\ 0 & \omega_3 & \omega_2 & \omega_1 \\ 0 & 0 & \omega_3 & \omega_2 \\ 0 & 0 & 0 & \omega_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

En realidad, en las CNN, para que el vector de salida se corresponda exactamente con la multiplicación $W\boldsymbol{x}$ la operación correspondiente no es convolución completa, si no que solo se cogen la parte central, de forma que el resultado tiene necesariamente el dominio correspondiente a la siguiente capa de datos. En el ejemplo, los dos valores centrales.

Sin embargo, por convenio, en la mayoría de paquetes de programación, en vez de utilizar la operación convolución entre los vectores $\boldsymbol{\omega}$ y $\boldsymbol{x}$, calculan la correlación cruzada entre ambos vectores ($\boldsymbol{\omega} \star \boldsymbol{x}$). Sin embargo, en la literatura, por comodidad, siempre se habla de convolución. Como se sabe, el primer paso de una convolución es la inversión del operador $h(t)$ por $h(-t)$, mientras que en una correlación cruzada se utiliza sin invertir. La relación entre las operaciones en los vectores ejemplo se muestra a continuación.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} * \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \star \begin{bmatrix} \omega_3 \\ \omega_2 \\ \omega_1 \end{bmatrix}$$

De esta forma, se ha demostrado que la operación lineal en las CNN se puede ver como una convolución y no como una multiplicación de matriz por vector. Este es un avance de las CNN, en las operaciones de convolución se pueden usar algunos herramientas de cálculo óptimo para reducir su tiempo de ejecución con respecto a la multiplicación de matrices.

Además, como se ha mencionado anteriormente, esta operación se puede comparar con la de un filtro en una imagen en 2D si se considera cada dato (bola) como el valor de un píxel (1 datos por píxel en cada imagen en blanco y negro y 3 por cada una en RGB). Un filtro se puede usar para detectar los bordes de una imagen como muestra la figura 6.

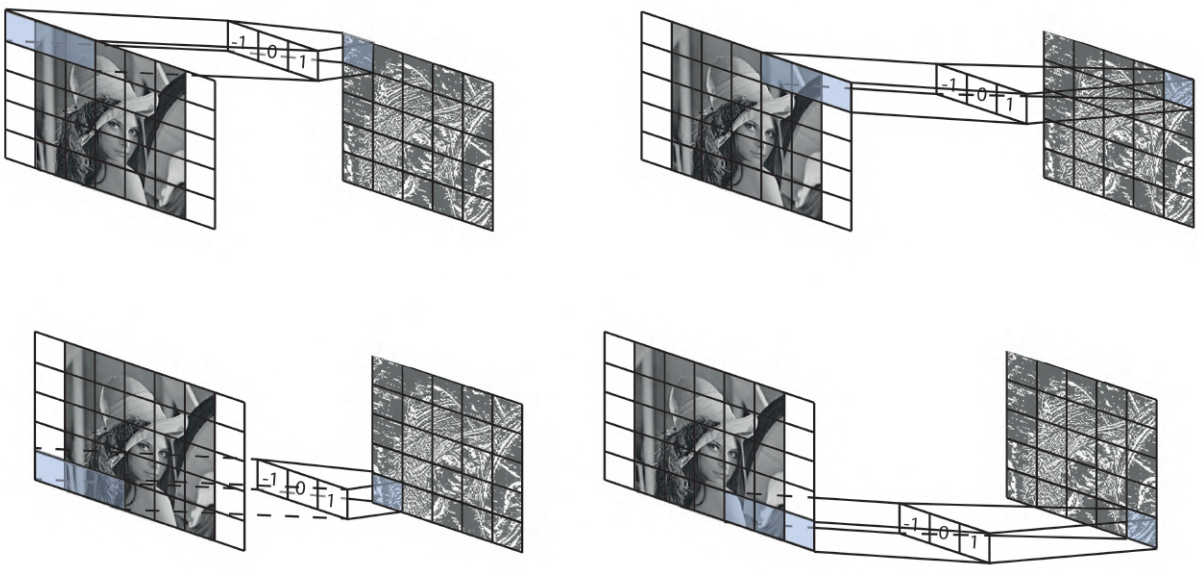


Figura 6 – Ejemplo de un filtro

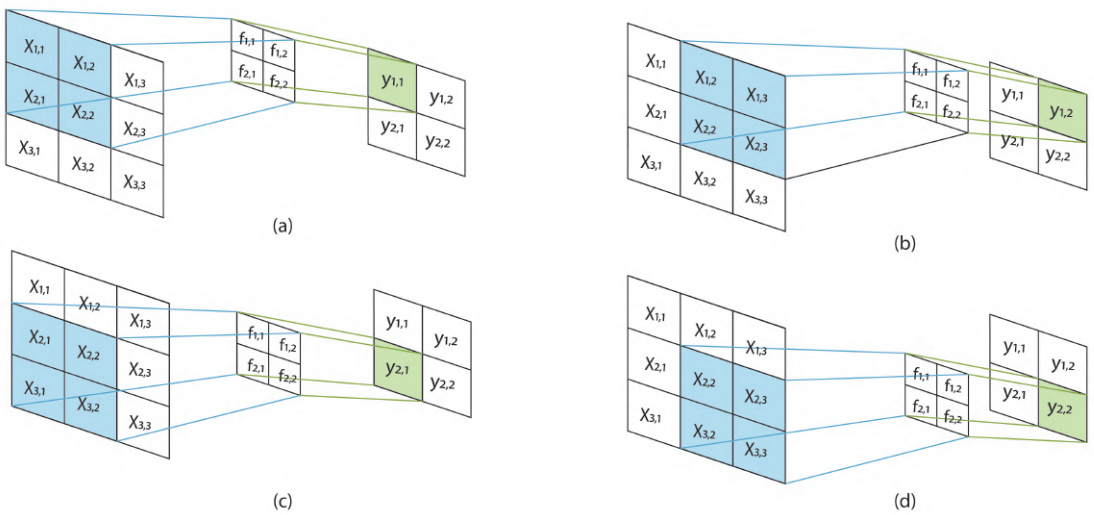


Figura 7 – Ejemplo de una operación de un filtro de 2x2 sobre una imagen de 3x3

Si tenemos una imagen de tamaño 3×3 y un filtro de tamaño 2×2 como los de la figura 7 las componentes de la imagen de salida se podrían calcular como:

$$\begin{aligned} y_{1,1} &= x_{1,1}f_{1,1} + x_{1,2}f_{1,2} + x_{2,1}f_{2,1} + x_{2,2}f_{2,2} \\ y_{1,2} &= x_{1,2}f_{1,1} + x_{1,3}f_{1,2} + x_{2,2}f_{2,1} + x_{2,3}f_{2,2} \\ y_{2,1} &= x_{2,1}f_{1,1} + x_{2,2}f_{1,2} + x_{3,1}f_{2,1} + x_{3,2}f_{2,2} \\ y_{2,2} &= x_{2,2}f_{1,1} + x_{2,3}f_{1,2} + x_{3,2}f_{2,1} + x_{3,3}f_{2,2} \end{aligned}$$

Que realmente es equivalente a una convolución por lo que se pueden aplicar todo lo anteriormente explicado para 1D.

A parte del tamaño del filtro, en las redes que se diseñen, hay que definir otros parámetros. Estos parámetros, a parte de deber ser utilizados dependiendo de la función que tiene el filtro, son importantes para definir el tamaño de la imagen de salida. En las figuras 8, 9, 10 (obtenidas de [3]) se muestra su efecto. Por simplicidad, se muestran los cuatro primeros pasos, se ha omitido el filtro y sólo se muestra la imagen de entrada (azul) y la de salida (verde). Cuando se dice que se aplica con un valor n realmente se considera que es $n \times n$, si no es simétrico, se especifican los dos valores. Estos efectos son:

- Paso (*stride*): es lo que se avanza en cada paso del filtro, el valor normal es 1. Se puede ver en la figura 8 con valor 2.
- Relleno (*padding*): es la cantidad de píxeles que se añaden alrededor de la imagen original, por defecto se considera una orla de 0 píxeles. Su efecto se muestra en la figura 9 con valor 2.
- Dilatación (*dilatation*): es cada cuantos píxeles de la imagen original se cogen para pasar por el filtro, en la figura 10 con valor 2. Si se quiere que no tenga efecto debe de tener un valor de 1

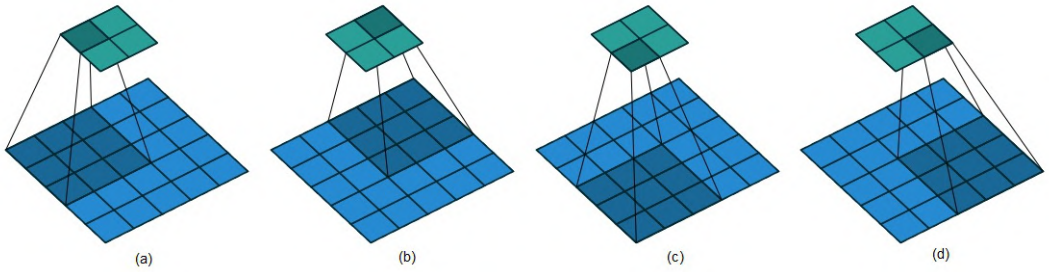


Figura 8 – Cuatro primeros pasos de la aplicación de un filtro con paso 2×2.

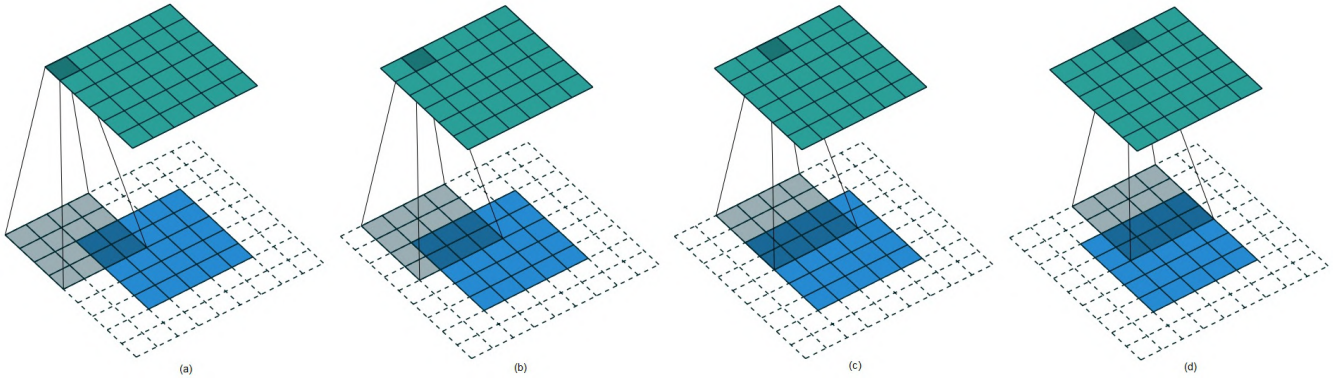


Figura 9 – Cuatro primeros pasos de la aplicación de un filtro con relleno 2×2.

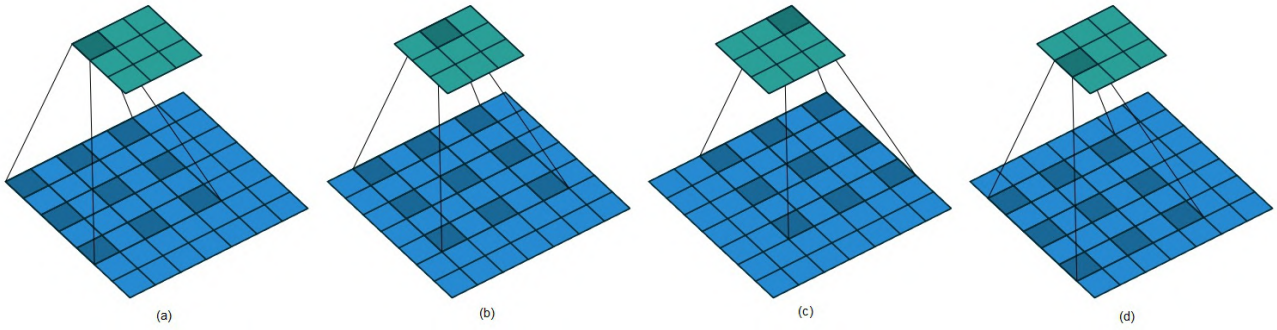


Figura 10 – Cuatro primeros pasos de la aplicación de un filtro con dilatación 2×2.

A parte de estas capas que se componen de una operación convolucional y una de activación, hay otro tipo de operaciones en las CNN que transforman la imagen y no se pueden comparar con filtros, son las llamadas capas de agrupación (*pooling* en inglés). Estas capas realizan una operación entre los píxeles. Se pueden utilizar por ejemplo para calcular la media de los valores de grupo de píxeles en una imagen como en la figura 11 (agrupación por media, *Average Pooling*) o el mayor valor (agrupación por máximos, *Max Pooling*). En estas capas también se utilizan los conceptos de paso, relleno y dilatación. Por tanto, se usan, para reducir la cantidad de información que se obtiene de la capa anterior.

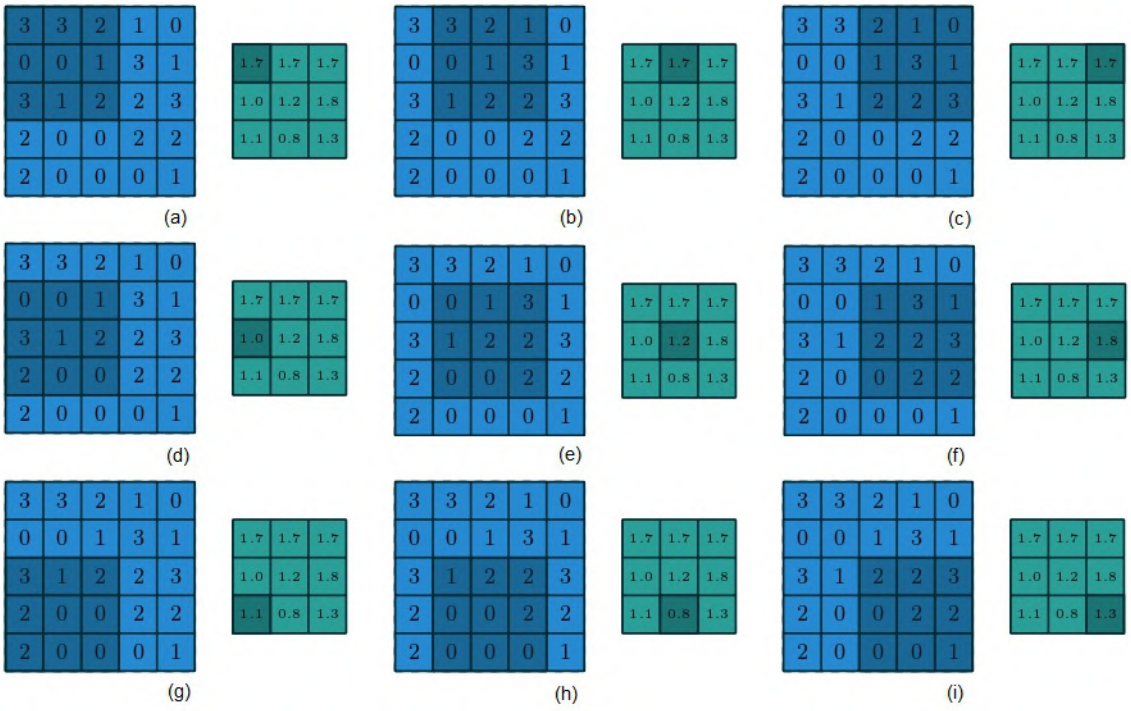


Figura 11 – Agrupación por media de 3x3 con paso y dilatación de 1 y relleno de 0. Obtenido de[3]

Finalmente, en última posición suele usarse una capa denominada de clasificación en las que se calculan las probabilidades de pertenecer a una clase u otra. Es simplemente una capa con cada una de las salidas conectadas con todas las entradas de la capa, por ello también suele llamarse capa totalmente conectada (*fully connected layer* en inglés). Al llegar a estas capas normalmente los datos de los que se dispone son características intermedias que han perdido las 2 dimensiones de las imágenes, por lo que la entrada de esta capa es un vector. Además, estas capas no son convolucionales, funcionan como las explicadas en la sección **Redes Neuronales (NN)**, ya que todos sus pesos son distintos. Un ejemplo se puede ver en la figura 12.

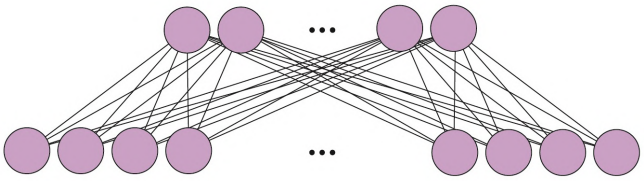


Figura 12 – Ejemplo de una capa totalmente conectada

2.2.3. Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (*RNN* por su acrónimo en inglés) son un tipo de redes neuronales en las que su salida no sólo está influenciada por la entrada en ese momento, si no por todo el historial de entradas.

Al igual que en las CNN, en la figura 13, los círculos representan datos. En este caso, cada x_i puede ser una imagen o un vector pero es parte de una secuencia. Por ejemplo, x puede ser un vídeo con x_i siendo sus cuadros o x puede ser un texto con x_i siendo las palabras que lo componen.

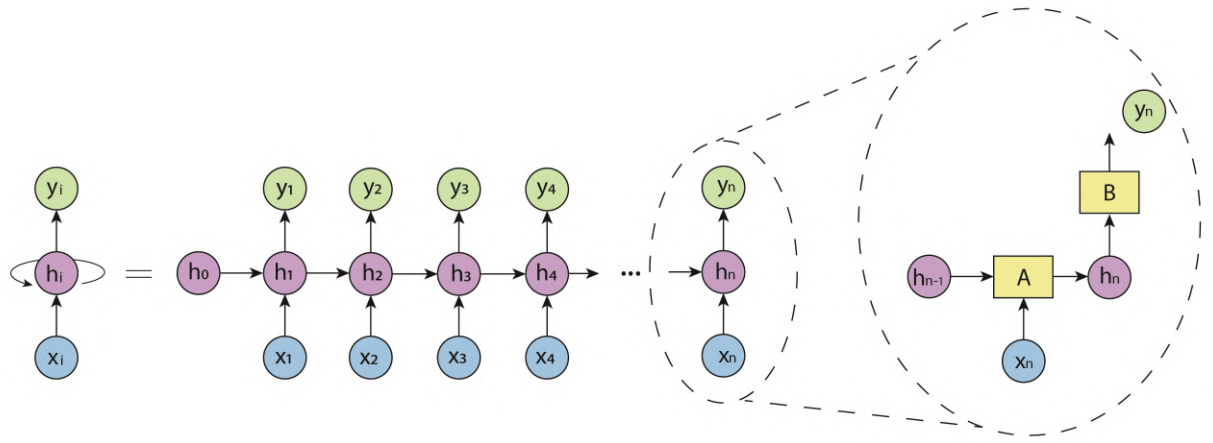


Figura 13 – Representación de una RNN

Como se puede ver en la figura 13, los datos de salida se calculan mediante:

$$\begin{aligned} y_i &= B(h_i) \\ h_i &= A(x_i, h_{i-1}) \end{aligned}$$

Como se puede apreciar una arquitectura RNN está compuesto por un módulo (A) que se va repitiendo. El módulo A es el mismo en todas sus repeticiones (también se comparten los pesos). Por tanto, su funcionamiento es independiente si para el mismo conjunto de datos las secuencias de entradas son de distinta longitud, es decir, la red funciona si el vídeo tiene 200 o 300 cuadros.

Dependiendo del número de entradas y salidas hay varios tipos de RNN. En la siguiente imagen pueden verse sus esquemas:

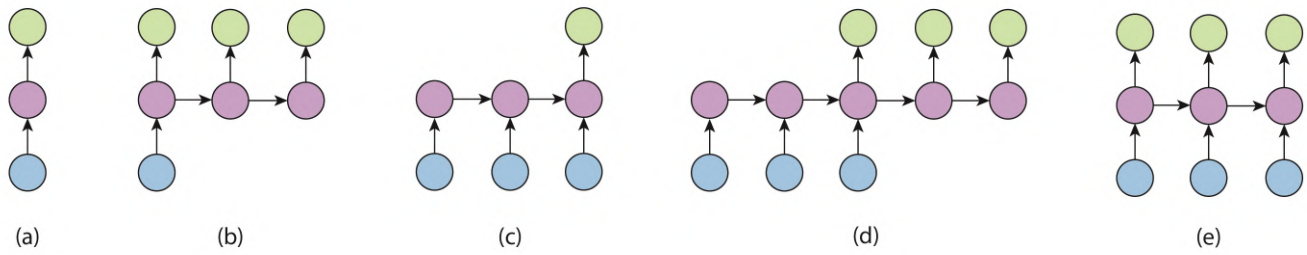


Figura 14 – Tipos de una RNN

Para la clasificación de vídeo, que tiene varias entradas (los cuadros del vídeo) y una salida (la etiqueta del vídeo), se utiliza el tipo que se muestra en la figura 14 (c).

En las anteriores imágenes se puede ver que el funcionamiento de una RNN consiste en actualizar el vector temporal cada vez que calculan las operaciones pertinentes para el sistema A. Computacionalmente se puede entender como una función "step"que en el paso t actualiza el vector h_t en función de la entrada x_t y del valor h_{t-1} anterior. El siguiente código lo muestra

```
rnn = RNN()
y = rnn.step(x)
```

Siendo la clase RNN:

```
class RNN:
        ...
        def step(self, x):
                self.h = updateH(self.h, x)
                y = updateY(self.h)
                return y
```

Por tanto, la clave del funcionamiento de las RNN está en entender cómo se actualiza el vector temporal y cómo se calcula la salida a partir de él, esto es, las funciones *updateH* (A en la imagen) y *updateY* (B en la imagen).

Cabe destacar que la función $updateY$ depende sólo del vector temporal actualizado y cambiará dependiendo del tipo de tarea. Normalmente será una CNN o algunas capas totalmente conectadas (*fully connected*). Por ello, se centrará el estudio en A.

En los últimos años, se ha experimentado con distintos tipos de RNN que actualizan sus valores de formas diferentes. Se descubrió que las RNN originales eran incapaces de aprender la dependencia entre entradas a largo plazo: el vector temporal en el paso t_n era muy similar si se empezaba en el paso t_k o t_0 cuando la diferencia entre k y n era grande. Este problema lo resuelven las LSTM (*Long-Short Term Memory* por sus siglas en inglés) propuesto en [7], una variación de las RNN tradicionales que se usa en la actualidad.

El esquema de un módulo LSTM se puede ver en la siguiente figura:

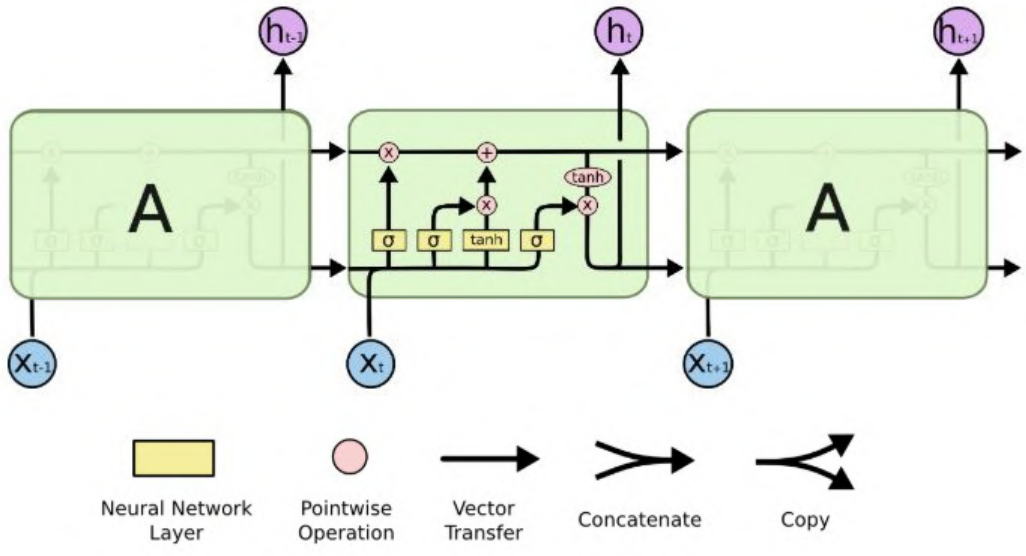
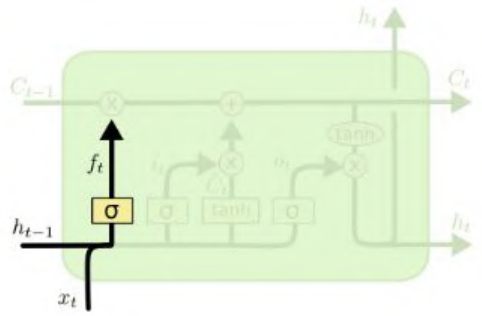


Figura 15 – Esquema de una LSTM. Obtenida de [16]

A parte del vector temporal $\boldsymbol{h_n}$, las LSTM incorporan otro vector $\boldsymbol{c_n}$, que es el vector que guarda el estado de la célula. Además, el módulo se divide la zona en la que elimina cierta información, en la que aprende información de la entrada y en la calcula la salida.

La zona de eliminación es está destacada en la siguiente figura y se refiere a la zona en la que se olvidan determinados valores de c_t.

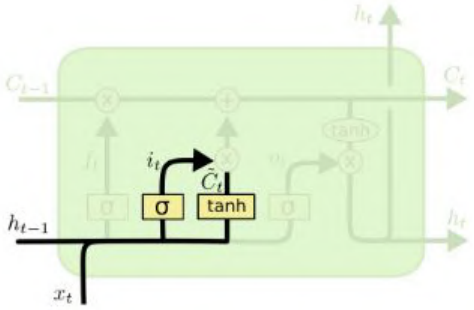


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figura 16 – Zona de olvidar en una LSTM. Obtenida de [16]

Se compone de una capa convolucional que calcula el vector $\boldsymbol{f_t}$ a partir de la concatenación del vector de entrada x y el vector temporal anterior $\boldsymbol{h_{t-1}}$. f_t tiene el mismo tamaño que $\boldsymbol{c_{t-1}}$, cada valor de $\boldsymbol{f_t}$ está entre 0 y 1 y se corresponde con cuánto de su valor tiene que porcentaje de su valor tiene que eliminar $\boldsymbol{c_{t-1}}$.

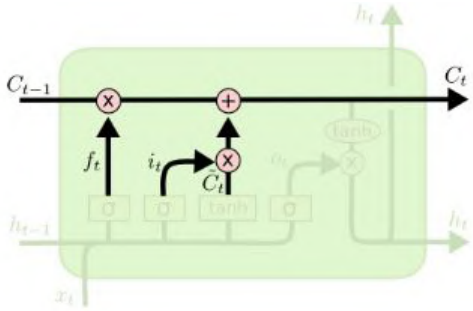
La zona de aprendizaje se muestra en la figura 17. En ella se calcula las señales $\boldsymbol{i_t}$ y $\boldsymbol{\tilde{c}_t}$ con un par de capas convolucionales. El vector i_t es equivalente a f_t y decide que valores se actualizarán, mientras que $\boldsymbol{\tilde{c}_t}$ es la señal de nuevos valores para el estado de la célula. Cabe destacar el uso de la función $\tanh$ para calcular $\boldsymbol{\tilde{c}_t}$ y no la función sigmoide σ. La diferencia entre ambas es su que su recorrido varía entre entre -1 y 1 en vez de entre 0 y 1.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figura 17 – Zona de aprendizaje en una LSTM. Obtenida de [16]

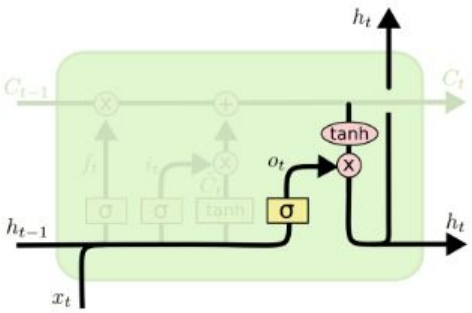
Estas dos zonas se combinan para calcular el nuevo estado de la célula $\boldsymbol{c_t}$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figura 18 – Zona de cálculo del estado de la celda. Obtenida de [16]

Finalmente, se calcula el vector temporal $\boldsymbol{h_t}$ con otra capa convolucional para producir el vector $\boldsymbol{o_t}$, que decide qué valores se van a utilizar en la salida y lo multiplica por $\tanh(\boldsymbol{c_t})$ para que tenga valores entre -1 y 1. Estas operaciones se pueden ver en la figura 19



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figura 19 – Cálculo de la salida en una LSTM. Obtenida de [16]

2.2.4. Herramientas de evaluación para Redes Neuronales

Al tener tantos parámetros de red y entrenamiento es imprescindible tener herramientas de análisis que evalúen el comportamiento de las redes neuronales.

En primer lugar, se suele asignar una puntuación a las redes entrenadas en función de lo bien que se predigan las etiquetas. Para ello, se comparan las predicciones con sus etiquetas reales. Estas métricas suelen seguir la fórmula:

$$p = \frac{a}{n} \times 100$$

Donde la puntuación (p) es el porcentaje de dividir los aciertos (a) entre el número de predicciones (n). La métrica más sencilla, denominada precisión (*acuraccy* en inglés), consiste en contar como aciertos cada vez que la predicción coincide con la etiqueta real. También existen otras métricas como la *top-k*, que considera acierto que la etiqueta real esté entre las k etiquetas más probables.

El siguiente parámetro que se suele medir es el tiempo. De esta forma, el tiempo de entrenamiento resulta un parámetro interesante, sobre todo si se plantea reentrenar la red a medida que se vayan etiquetando los datos. Sin embargo, al parámetro que más importancia se le suele dar es al tiempo de uso, que es el tiempo en el que la red tarda en procesar un dato de entrada hasta convertirlo en una predicción. Además, a parte de dar un valor numérico, este tiempo lo que hace es dividir los algoritmos entre *en linea*, si se procesan los datos más rápido que se muestrean, o *fuera de línea*, si no da tiempo a procesar los datos en tiempo real.

Las métricas anteriores miden el rendimiento de la red, sin embargo, para conocer su comportamiento y analizar qué aspectos de la red conviene mejorar, se suelen utilizar herramientas de visualización.

Entre estas herramientas destacan métodos estadísticos que reducen el número de dimensiones de las características intermedias para poder representarlas en 2D o 3D. En estos algoritmos, cada conjunto de (n) características intermedias asociadas a un dato representan vector de dimensión n se convierte en un vector de dimensión 2 o 3. Uno de los más utilizados es el Análisis de Componentes Principales (*PCA* por *Principal Component Analisis* en inglés), en el cual se escogen 2 o 3 dimensiones que tengan mayor varianza como dimensiones de salida. Con el avance del Machine Learning en los últimos años, han aparecido nuevas técnicas específicas para este área. Uno de los nuevos algoritmos se llama t-SNE (*T-Distributed Stochastic Neighbor Embedding*) y fue propuesto en [15] en 2008. Este método consigue representar con puntos muy cercanos las características parecidas mediante técnicas no lineales. Lo que representan matemáticamente estas técnicas es una aplicación de un espacio vectorial de n dimensiones a uno de 2 o 3. Si se visualizan, gracias a estas técnicas, las características intermedias de salida de cada capa de la red neuronal, se apreciarán agrupaciones de puntos a medida que avanzamos en ella, ya que bajo un funcionamiento ideal, las características de una clase, se irán diferenciando progresivamente de las de una clase distinta. La figura 20 muestra la representación ideal de las características de la última capa después de aplicar el algoritmo t-SNE.

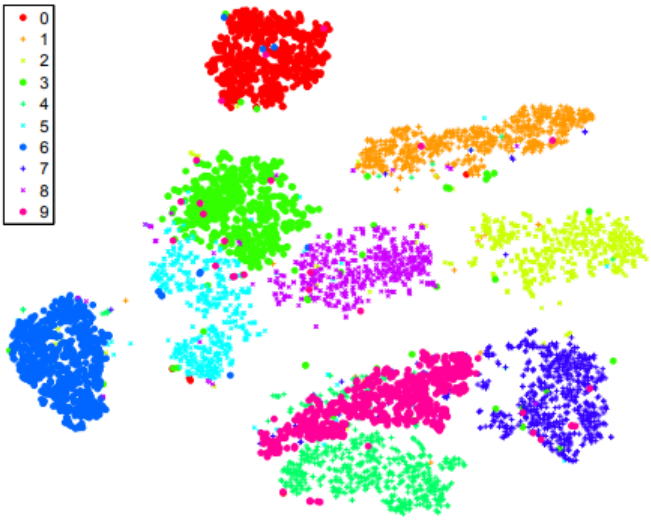


Figura 20 – Representación ideal de las características de la última capa. Obtenido de [15]

Otra herramienta que se utiliza para entender el comportamiento de la red es la matriz de confusión. Esta matriz sólo permite conocer el resultado final ya que se construye basándose en las predicciones. En el cuadro 1 se muestra el formato habitual, en las columnas se encuentran las predicciones de la red y en las filas las etiquetas reales de manera que en la casilla de la fila i y la columna j se muestra el número de entradas para los que la red ha predicho la clase j pero realmente son de la clase i.

		Predicciones			
		Clase 0	Clase 1	...	Clase n
Etiqueta	Clase 0				
	Clase 1				
	⋮				
	Clase n				

Cuadro 1 – Formato de la matriz de confusión

2.3. Visión Artificial

Lo que inicia el campo de la Visión Artificial en 1970 es *el deseo de reconstruir la estructura en tres dimensiones del mundo a través de imágenes y usar esto como un paso más a la hora de entender la escena* [22, Pág. 11] para avanzar en la Inteligencia Articial. Además, esto mismo lo diferencia de las investigaciones anteriores de procesado digital de imágenes.

Además, durante esta época se establecieron las bases del área dividiendo el campo en tres niveles distintos de abstracción: 1. la teoría computacional que tiene como objetivo estudiar las limitaciones que se conocen, 2. el diseño de algoritmos que trata de ver cuales son la entrada, la salida del sistema y que representaciones intermedias tiene sentido calcular y, por último, 3. la implementación hardware se dedica a desarrollar los microchips óptimos para correr los algoritmos [22, Pág. 13].

Aunque sin perder de vista los otros niveles, pues tienen implicaciones evidentes, en el presente trabajo me centraré en el segundo nivel.

El vídeo es un elemento que se ha desarrollado mucho en los últimos años debido a los avances en el hardware de las máquinas, ahora es mucho más sencillo su procesado. En ese sentido, el vídeo se ha revalorizado y el consumo de servicios basados en vídeo ha crecido. YouTube se ha impuesto sobre servicios de fotografía como Picassa, que también es de Google. En los últimos años, servicios como Netflix, Movistar+ o HBO han conseguido que el vídeo esté en todos los hogares y han surgido nuevas redes sociales como TikTok que se basan exclusivamente en los vídeos. Las retransmisiones en directo también se han visto beneficiadas gracias a técnicas de procesado de vídeo.

Un problema básico que se plantea en el procesado de vídeos es su clasificación. En esa línea, siempre se ha tratado de avanzar en el reconocimiento de actividades humanas debido a sus evidentes aplicaciones. Los algoritmos que se han utilizado en los últimos años se tratan en la siguiente sección.

2.3.1. Clasificación de Vídeo - Actividades Humanas

Se denomina reconocimiento de actividades humanas (HAR, *Human Activity Recognition*, por sus siglas en inglés) a la clasificación de clip de videos con el objetivo de identificar qué están haciendo sus actores. Este área de la Visión Artificial ha crecido en los últimos años por el gran número de aplicaciones y los buenos resultados obtenidos.

Entre sus aplicaciones destacan interacción persona-máquina, donde el ordenador puede detectar un determinado gesto y realizar una operación en consecuencia; la videovigilancia inteligente para detectar comportamientos extraños en tiendas o aeropuertos; la búsqueda de vídeos basada en el contenido de los mismo, en la que ahorraría mucho trabajo de etiquetado manual de vídeos o en aplicaciones de realidad virtual.

Los principales problemas a los que se enfrenta un algoritmo de Reconocimiento de Actividades Humanas son de diversa tipología. En primer lugar, cabe destacar las complejidades en el entorno, no todos los vídeos que analizamos comparten las mismas características. La variación entre las clases se debe tener también en cuenta, los algoritmos deben tratar de encontrar la manera de identificar lo mejor posible estas variaciones para obtener mejores resultados. La mayoría de las veces, los sistemas están formados por varias cámaras, hasta el gesto más sencillo cambia dependiendo del punto de vista, lo que produce problemas en su identificación. Sin embargo, el problema más difícil de solucionar tal vez sean las oclusiones, muy frecuentemente la cámara pierde la visión del sujeto y debe imaginar. lo que está sucediendo mientras no ve sus movimientos. En cualquier caso, dependiendo de la naturaleza de los vídeos, habrá un tipo de problema que sea más dominante que el otro y, por lo tanto, identificar la correcta problemática será determinante a la hora de escoger una perspectiva determinado y diseñar un algoritmo eficiente[20, Pág. 2].

En general, todos los algoritmos tratan de imitar el sistema de visión humana. En primer lugar, identifica la forma humana gracias al movimiento de píxeles entre cuadros y las formas que definen para después identificar la propia actividad que se está realizando. Es por ello por lo que los algoritmos de está tarea son muy representativos del campo de Visión por Ordenador: cada solución se puede dividir en dos fases, una fase de extracción de características (o *features*, como lo identifica la literatura en inglés) que sean pertinentes para el problema y otra de clasificación basándose en las características anteriormente obtenidas con un clasificador genérico entrenable[20, Pág. 2].

La manera de aproximarse al extractor de características es lo que divide los algoritmos en dos grandes tipos: los que utilizan características extraídas tras aplicar las transformaciones definidas por el ingeniero y los que aprenden las características autónomamente en función de los datos de entrada. Los primeros han sido los más usados tradicionalmente mejorando sus resultados a lo largo del tiempo. Sin embargo, con la aparición del Aprendizaje Profundo, se han empezado a preferir las basadas en aprendizaje, ya que se adaptan mejor a la naturaleza cambiante de los vídeos que se analizan.

A continuación se hará un repaso a los principales algoritmos basados en aprendizaje, ya que es el

tema que se tratará en este trabajo[1].

2.3.2. Basados en aprendizaje

Los hay basados en *Deep Learning* y no basados en *Deep Learning*

Entre los que no utilizan *Deep Learning* se encuentran los basados en programación genética [14], poco desarrollados hasta el momento pero que ha llegado a alcanzar una precisión de $95\,\%$ en el conjunto de datos KTH, o los basados en aprendizaje de diccionario disperso donde se encuentra [5].

Sin embargo, por el tema de este trabajo es más pertinente comentar los últimos avances en los algoritmos de *Deep Learning*, concretamente, los basados en aprendizaje supervisado.

Desde 2014 han aparecido dos enfoques principales para solucionar este tipo de problemas: por una parte surgen las Redes Neuronales Convolucionales en 3D, que representan la evolución natural de las redes en 2D que se aplicaban en imágenes pero en vídeos, y por otra parte aparecen los enfoques que tratan el vídeo cuadro a cuadro para después unir los resultados de cortos periodos de tiempo. Ambas propuestas tratan de resolver la aparición de una dimensión adicional utilizando las herramientas de las que disponen las redes neuronales, sin embargo, sus aproximaciones son completamente distintas. Mientras que la primera utiliza aprendizaje con pesos temporales (además de los espaciales de las 2D-CNN), la otra utiliza una capa de completamente conectada para unir resultados. También se han intentado aplicar técnicas de basadas en Redes Neuronales Recurrentes, que se utilizaban para procesamiento de textos.

1. Los primeros artículos que obtuvieron atención de la comunidad científica que reconocían actividades en vídeos usando Deep Learning procesando los vídeos cuadro a cuadro para después unir algunos de ellos usando capas completamente conectadas fueron [10] y [21] en 2014. Concretamente, [10] propone maneras de fusionar los cuadros procesados, se pueden ver en la imagen 21. Estas arquitecturas se entrenaron sobre el *Sports 1M Dataset* y obtuvieron una puntuación de $60,9\,\%$ para una métrica de Top1 y de $80,2\,\%$ para Top5. Cabe destacar que este *Dataset* está etiquetado al detalle y diferencia por ejemplo las actividades de ciclismo, ciclismo en ruta y ciclismo en pista, por lo que la última métrica puede parecerse más a los resultados que publican otros estudios sobre *Datasets* que no estén etiquetados tan al detalle. En cualquier caso, parece que en los últimos años se ha abandonado en favor de las Redes Convolucionales en 3D.

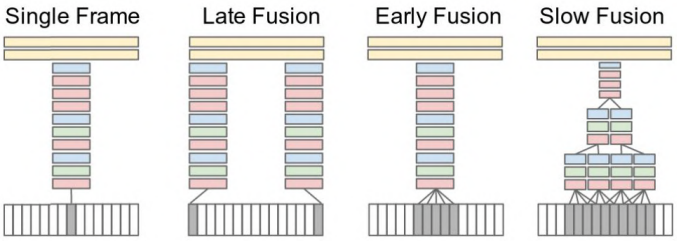


Figura 21 – Fusiones propuestas en [10]

2. Entre los artículos que proponen algoritmos basados en la utilización deRedes Convolucionales en 3D, destaca [8] publicado por primera vez en 2010, pero cuyos resultados empezaron a ser relevantes en 2012 después de aparecer en el *IEEE Transactions Pattern Analysus and Machine Intelligence*. Esta investigación propone usar dos capas convolucionales en 3D alternadas con varias capas de *pooling* para acabar con una de clasificación. Este artículo es el que inicia la linea de investigación utilizando este enfoque con resultados bastante buenos para la época en un *Dataset*

[1] Si se tiene interés en conocer el resto conviene leer [20], review que recoge todos los avances en este tema durante la última década

de vídeos de seguridad en el aeropuerto de Gatwick (Londres) y en el *KTH Dataset* con una puntuación de $90,2\,\%$ sin ninguna técnica de preprocesado de imagen. Esta línea de investigación se ha ido desarrollando a lo largo de los años y se han ido obteniendo resultados bastante satisfactorios en distintos *Datasets* como por ejemplo el artículo [23].

3. No obstante, no son los únicos enfoques. En los últimos años se han empezado a usar las Redes Neuronales Recurrentes (*Recurrent Neuronal Networks*, RNN, en inglés), que ya se utilizaban en áreas como el procesamiento de lenguaje natural gracias a sus propiedades secuenciales. De hecho, una de las primeras investigaciones en este tema aplicadas a vídeos fue propuesta por [1] y se originó por la inquietud de ponerle un título a unos clips.[2] Naturalmente, propone las herramientas ya utilizadas en el área de procesamiento de lenguaje natural, las RNN. Concretamente, propone usar LSTM, un tipo de RNN, una herramienta que se vuelve incluso útil sin la parte de procesamiento de lenguaje natural. [1] propone la arquitectura mostrada en la figura 22, que consiste en procesar cada cuadro de los vídeos con una CNN para después utilizar esas características obtenidas como entrada de una LSTM. La salida de la LSTM será la etiqueta del vídeo.

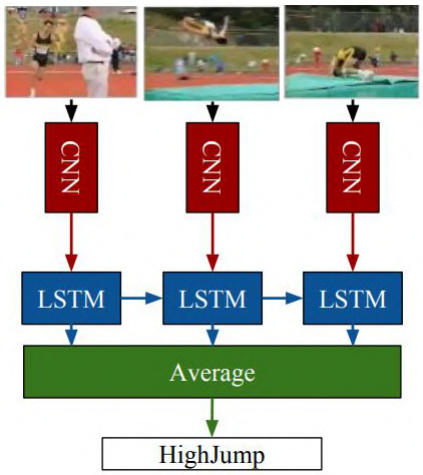


Figura 22 – Arquitectura para clasificación de vídeo propuesta por [1]

Parece que estos tres tipos de enfoques siguen una evolución natural. Ante el nuevo problema de la dimensión adicional, primero se intenta hacer lo que se sabe en 2Dy se intenta unir la tercera dimensión de una manera más tradicional. Después, se abstraen los conocimientos en 2Dpara proponer una solución similar en 3D. Por último, se aplica una nueva tecnología que se adapta mejor a la naturaleza secuencial de la tercera dimensión, que no se basa en la localidad. Es por ello por lo que en este trabajo, se utilizarán el primer enfoque y el último. De esta forma, se muestra cómo la tecnología avanza gracias a la naturaleza colaborativa de los investigadores, que se nutren de lo que ya se ha hecho para proponer soluciones distintas.

2.3.3. Conjuntos de datos

En este tipo de algoritmos recoger el conjunto de datos (*Dataset* en inglés) y etiquetarlo suele ser la labor que más tiempo ocupa, ya que se necesitan una cantidad enorme de datos. Además, como un vídeo ocupa más que una imagen, su procesado lleva más tiempo. Consecuentemente, el tiempo de entrenamiento crece significativamente. Una solución que se suele usar es disminuir la calidad de los vídeos para que ocupen menos datos, sin embargo la calidad del resultado, dependerá de la calidad

[2] Si hubiese llegado 45 años antes, le hubiese ahorrado mucho trabajo a nuestro estudiante del MIT

del vídeo y esto no podrá hacerse indefinidamente. Por lo que, en este tipo de investigaciones, incluso disponiendo de máquinas con mucha capacidad para entrenar la red como las gamas más altas de GPUs de *Nvidia* o las TPUs de *Google* hay que encontrar un compromiso entre la cantidad de datos utilizados y la cantidad de datos que podrían ser utilizados para obtener los mejores resultados. Algunas técnicas incluyen el submuestreado de los vídeos o la reducción del tamaño de los cuadros.

Este hecho no ha evitado que surjan multitud de conjuntos de datos *open source* con un tamaño enorme y una calidad de vídeo significativa. Esto permite al investigador decidir que técnicas de disminución de datos se ajustan mejor a su estudio. Estos *Datasets* de vídeo se suelen clasificar en función de su temática.

Por una parte, destacan los que graban a actores realizando acciones y gestos. Entre ellos, el *KTH* es el más utilizado por su gran cantidad de vídeos. Este tipo de *Datasets* pueden ser útiles en la realidad para detectar lenguaje de signos, por ejemplo.

Otro grupo destacado son las que se basan en películas o vídeos ya grabados. Estos datos tienen la ventaja de que no hace falta grabar los vídeos, simplemente etiquetarlos. Un conjunto importante es el *UCF50*, otro el es *Sports-1M Dataset*.

También son importantes los vídeos de cámaras de seguridad, aunque este tipo de *Datasets* se suelen utilizar para problemas de seguimiento (*tracking* en inglés) y no para clasificación de vídeos.

Finalmente, están los conjuntos de datos que se ocupan de actividades normales en las rutinas de las personas. Estos conjuntos suelen ser los utilizados cuando se quieren resolver problemas más reales y que sean menos orientados a la investigación básica. Un conjunto de datos de este tipo puede ser perfectamente el que pida hacer un cliente que tiene una necesidad concreta. Destaca por la calidad de sus cuadros (1624×1224) y su tamaño (5609 vídeos con más de $800{,}000$ cuadros en total) el *Cooking Dataset* [19]. No obstante, su calidad puede dificultar su procesamiento, el conjunto de vídeos propuesto en [11] llamado *Breakfast Dataset* soluciona este problema proporcionando cuadros de un tamaño inferior (320×420).

3. Método experimental

En esta sección se explicará la metodología seguida para diseñar los sistemas. Como se ha comentado en la **Introducción**, se han realizado dos experimentos para comparar dos maneras de clasificar vídeos basadas en *Deep Learning*. Por una parte, se ha diseñado un sistema que usa las Redes Neuronales Convolucionales para procesar la dimensión temporal. Por otra parte, se ha propuesto un sistema basado en Redes Neuronales Recurrentes.

Si se recuerda en la sección de **Aprendizaje Profundo**, se clasificó las fases del diseño de un algoritmo supervisado de Aprendizaje Profundo en recogida de datos, entrenamiento y uso. Por ello, cada experimento se ha dividido en estas fases. A continuación se tratan las características específicas de cada una de ellos.

Ante la imposibilidad de acceder a recursos de la universidad por problemas en la conexión remota, se optó por usar la herramienta *Google Colab* para ejecutar los programas. Este servicio proporciona un entorno de *jupyter notebooks* con GPUs bastante potentes y es gratuito. Sin embargo, el usuario no controla la capacidad a la que accede y según se va utilizando, reduce la potencia disponible con el fin de repartir los recursos de manera equitativa entre los usuarios. Por ello, uno de los objetivos prioritarios durante el entrenamiento ha sido tratar de diferenciar las fases del diseño del algoritmo lo más posible. De está forma se ahorra tiempo. Más concretamente, en la fase de recogida de datos se trató de dejar los datos preparados para cargar esos datos y entrenar la red directamente sin necesidad de ningún tipo de transformación adicional cada vez que se mete un dato en la red.

3.1. Recogida de datos

Esta fase la comparten los dos experimentos. En primer lugar se hará un repaso a las características de los vídeos utilizados. Posteriormente, se describirá el preprocesado al que se han visto sometidos para adaptar los datos al problema que se intenta resolver y prepararlos para que simplemente haya que cargarlos al principio.

3.1.1. Características del Dataset

Se han seleccionado los vídeos del *Breakfast Dataset* [11], preparados por el grupo *Serre Lab* de la Universidad de Brown. En el conjunto de vídeos participaron 52 personas, cada una prepara 10 recetas de diversa índole. Se utilizaron 18 cocinas diferentes. Cada receta, se ha grabado con un número de cámaras variable entre 3 y 5 según la disposición de la cocina. La posición de ellas entre una cocina y otra también variaba.

El total grabado superaba las 77 horas de vídeo (más de 4 millones de cuadros). Los vídeos tienen una resolución de 320x240 píxeles con una velocidad de 25 fps.

La grabación era totalmente improvisada, esto es, no tenía guión ni se había ensayado antes. Simplemente, antes de la grabación, se le daba a los participantes la receta y se les pedía que la preparasen. Por tanto, la forma de realizar cada acción es muy diferente incluso en la misma receta. Las 10 recetas eran:

- Café
- Zumo de Naranja
- Chocolate caliente
- Té
- Un bol de cereales

- Huevos fritos
- Tortitas
- Ensalada de frutas
- Un sandwich
- Huevos revueltos

En cuanto a la anotación, se etiquetó cada cada cuadro en 48 categorías diferentes entre las que se incluían, entre otras, *coger un cartón de leche*, *coger un plato* o *batir huevos*. De esta manera manera, el resultado del etiquetado es un archivo por cada vídeo de la receta que especificaba qué cuadros comprendía cada actividad que se realizaba en el vídeo correspondiente.

Se ha escogido este *Dataset* debido a su fuerte componente secuencial. Se clasifican actividades que, congeladas en un momento pueden ser parecidas, como por ejemplo cortar y pelar. Si se hubiese escogido un problema que clasificase vídeos de deporte, por ejemplo, el problema sería más trivial: bastaría con clasificar los cuadros de un corte de vídeo de manera independiente y luego hacer la media de sus resultados.

3.1.2. Preprocesado

Como se ha comentado, la potencia computacional necesaria para entrenar las redes neuronales es grande. Además, utilizar vídeos en vez de imágenes aumenta el tiempo de entrenamiento y la capacidad que debe tener la máquina. Con el fin de simplificar el entrenamiento y que fuese abordable en un tiempo razonable se tomaron dos decisiones que simplificaban el problema:

- Por una parte, en vez de utilizar las 48 categorías, se han agrupado varias de ellas bajo la misma etiqueta. Por ejemplo, en vez de tener dos etiquetas como *poner fruta en un bol* y *poner huevo en un plato*, se ha utilizado *poner* para ambas. La tabla 2 muestra qué etiquetas del dataset original corresponden a qué categorías en nuestros experimentos. De esta manera, se consigue igualar más o menos el número de vídeos para cada clase[3].
- También se realizó un submuestreo en los vídeos y en vez de utilizar la velocidad de cuadros proporcionada en el Dataset (25 fps), se utilizó 5 fps, una velocidad que consigue captar las actividades humanas. Así, se reducía el número de cuadros y se aceleraba su procesamiento. La tabla también muestra el número de vídeos que había de cada tipo y la suma de frames con el submuestreo realizado.

El presente estudio no intenta clasificar cada cuadro de un vídeo para establecer su secuencia, si no que solo trata de dar una etiqueta global por cada vídeo que procesa. Por tanto, es necesaria una última adaptación de los datos obtenidos de la universidad de Brown. Para ello, se escribió un pequeño programa en *Python* que cogiera cada vídeo de una receta y su correspondiente etiquetado cuadro a cuadro y lo dividiese en clips de vídeo en el que cada cuadro de ese corte tuviera la misma etiqueta. El programa utilizado se puede consultar en **Anexos**. El número de vídeos y de cuadros por cada etiqueta se puede consultar en el cuadro 2.

[3] El problema de tener un número descompensado de vídeos de una clase es que después del entrenamiento la red tenderá a predecir la clase más común y a dar porcentajes de precisión muy altos cuando realmente la red no está funcionando correctamente. Por ejemplo, en un conjunto de datos con las etiquetas 0 y 1, si se tiene un $90\,\%$ de datos que son 0, probablemente la precisión del modelo sea aproximadamente $90\,\%$ porque detecte muy bien los datos que son 0. Sin embargo, no se considera que funcione bien porque los falsos negativos (detectar 0 cuando es 1) abundarán. Existen técnicas para evitarlo pero quedan fuera del ámbito de este trabajo y consumen más recursos computacionales

Clases originales	Traducción clases	Clases utilizadas	Vídeos	Cuadros	Etiqueta
SIL	Silence	**Nada**	625	13160	0
Pour milk	Verter leche	**Verter** (Pour)	477	23873	1
Pour juice	Verter zumo				
Pour cereals	Verter cereales				
Pour water	Verter agua				
Pour coffee	Verter café				
Pour oil	Verter aceite				
Pour dough2pan	Verter masa en sartén				
Pour egg2pan	Verter huevo en sartén				
Pour sugar	Verter azucar				
Pour flour	Verter harina				
Cut fruit	Cortar fruta	**Cortar** (Cut)	679	61520	2
Cut bun	Cortar panecillo				
Cut orange	Cortar naranja				
Crack egg	Romper huevo	**Crack** (Romper)	416	28843	3
Take plate	Coger plato	**Take** (Coger)	918	24976	4
Take cup	Coger taza				
Take bowl	Coger bol				
Take glass	Coger vaso				
Take topping	Coger topping				
Take squeezer	Coger exprimidor				
Take eggs	Coger huevos				
Take knife	Coger cuchillo				
Take butter	Coger mantequilla				
Put fruit2bowl	Poner fruta en bol	**Poner** (Poner)	908	35210	6
Put egg2plate	Poner huevo en plato				
Put topping	Poner topping				
Put pancake	Poner tortita				
Put bunTogether	Juntar panecillos				
Add SaltPepper	Añadir Sal y Pimienta	**Añadir-Echar** (Add)	375	17974	7
Add TeaBag	Añadir bolsa de té				
Stir dough	Remover masa	**Remover** (Stir)	534	42529	8
Stir milk	Remover leche				
Stir egg	Remover huevo				
Stir cereals	Remover cereales				
Stir coffee	Remover café				
Stir fruit	Remover fruta				
Stir tea	Remover té				
Fry egg	Freír huevo	**Freír** (Fry)	336	123913	9
Fry pancake	Freír tortita				
Butter pan	Poner manteca a sartén	**Poner manteca** (Butter)	191	19016	10
Stirfry egg	Saltear huevo	**Saltear** (Stirfry)	198	34909	11
Peel fruit	Pelar futa	**Pelar** (Peel)	185	26563	12
Squeeze orange	Exprimir naranja	**Exprimir** (Squeeze)	156	21244	13
Spoon powder	Echar levadura	**Echar con cuchara** (Spoon)	301	19678	14
Spoon flour	Echar harina				
Spoon sugar	Echar azúcar				
Smear butter	Untar mantequilla	**Untar** (Smear)	121	15072	15

Cuadro 2 – Correspondencia de clases originales con las utilizadas y número de elementos de cada una en vídeos y cuadros totales

Este preprocesado es necesario para facilitar el entrenamiento teniendo en cuenta de los recursos computacionales de los que se disponía pero afectan negativamente en el buen funcionamiento de la red. Por una parte, el submuestreado elimina datos y por otra, el agrupamiento de distintas clases bajo la misma hace que dos actividades muy distintas puedan tener la misma etiqueta. Sus efectos en el entrenamiento de las redes propuestas en este trabajo se tratarán en la sección de **Resultados**

Tras los primeros experimentos quedó claro que el tiempo de entrenamiento seguía siendo demasiado alto. Una iteración de todos los datos tardaba 1 hora aproximadamente y se estimaba que se debían hacer unas 100 para entrenar la red. Por ello se decidió, probar con distintas estrategias para reducir el tiempo de entrenamiento. Entre ellas destacan guardar los vídeos como tensores (objeto de datos básico de *Pytorch*) con una extensión $.pt$ en vez de tenerlos en $.avi$ para evitar tener que hacer ese procesado en cada iteración. Otra estrategia fue probar con una red preentrenada como manera de preprocesar los datos ya que se conseguía reducir drásticamente la cantidad de datos por vídeo. Este preprocesado es distinto en cada experimento y se puede considerar parte del entrenamiento aunque realmente no se entrene nada porque es parte de la red.

Como cada estrategia de preprocesado cambiada dependiendo de la red que se intentaba entrenar, por lo que se explicará cada transformación en la parte correspondiente de la subsección de **Entrenamiento** y se explorará su impacto en los resultados en la sección de **Resultados y Análisis**. Sin embargo, en la mayoría de los casos, se consiguió reducir drásticamente el tiempo de entrenamiento a apenas media hora en total.

3.2. Entrenamiento

Para el entrenamiento se dividieron los datos preprocesados en dos conjuntos, el conjunto de entrenamiento ($75\,\%$ de los vídeos) y el conjunto de prueba ($25\,\%$). El conjunto de prueba se utilizará en la siguiente subsección (**Uso**), que consiste en la utilización de la Red Neuronal.

Se decidió utilizar la técnica de validación cruzada. Para ello, se dividieron los datos del conjunto de entrenamiento obtenidos en el preprocesado en tres partes con aproximadamente el mismo número de vídeos en cada una ($25\,\%$). En esta técnica, para cada experimento se realizan 3 entrenamientos de la red. En cada uno de esos entrenamientos, se utilizan dos de los tres subconjuntos para entrenar la red (datos de entrenamiento, $50\,\%$ de los datos originales). El subconjunto restante (datos de validación, $25\,\%$) se usa para encontrar la precisión del modelo cada vez que se realiza una iteración del entrenamiento. El objetivo de entrenar una misma red tres veces es garantizar que la precisión del modelo es independiente de las particiones realizadas. Normalmente en las investigaciones se suele dar la media de estas tres repeticiones. Sin embargo, debido a las limitaciones de procesamiento de las que se disponía para este trabajo y el tiempo enorme que tardaba cada red en entrenarse, solo se realizó una repetición por cada experimento. Aún así, la conjunto de validación se siguió manteniendo para dar una puntuación del modelo durante el entrenamiento sin utilizar el conjunto de prueba ni el de entrenamiento.

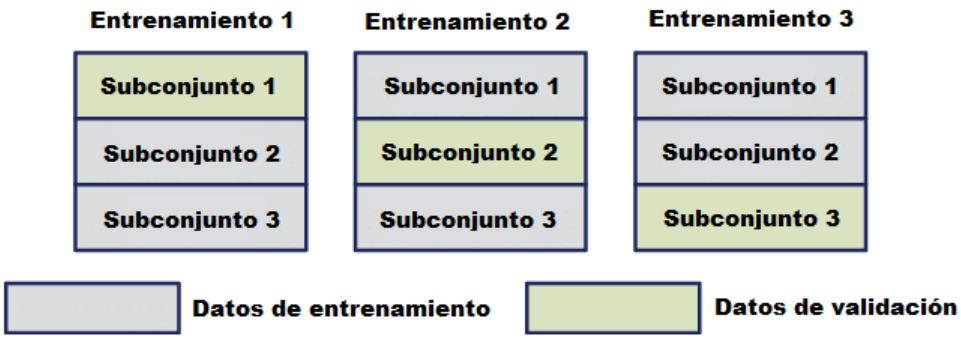


Figura 23 – Validación Cruzada

El programa para hacer el preprocesado de los vídeos y dividirlos en 4 subdivisiones iguales se puede consultar en los **Anexos**.

Tras este preprocesado de datos, se diseñaron las redes neuronales. La entrada de ambas arquitecturas es un vídeo de tamaño variable y la salida un número correspondiente a la categoría en la que se clasifica ese vídeo. Como se ha mencionado anteriormente el proceso de entrenamiento es por el cual la red aprende los pesos de los filtros que se utilizarán.

Los dos programas escritos, uno para cada tipo de red, comparten sus módulos. Siguiendo con los principios de programación modular, cada uno de estos módulos se han escrito en archivos distintos. Todos se pueden consultar en la sección de **Anexos**. Como se menciona en el objetivo 5 en la **Introducción**, se ha utilizado la librería *Pytorch*. Estos son:

- Módulo de datos (*data.py*): actúa como interfaz entre los datos y el programa. Se encarga de abrir los archivos en un formato que Python pueda utilizar. Se ha utilizado la interfaz de clase *Dataset* que proporciona *Pytorch* y se han modificado algunas funciones de ella. Concretamente, se redefine la función *__init__*, que se ejecuta al principio entrenamiento. Aunque el preprocesado ahorraba mucho tiempo de entrenamiento, cargar los datos desde la memoria ROM (*torch.load(file_name)*) no era inmediato (aproximadamente veinte minutos), se optó por cargarlos en la memoria RAM sólo en la inicialización y, así, evitar este tiempo en cada iteración. Es cierto que al comienzo del entrenamiento el tiempo crecía considerablemente pero también conseguía que el tiempo de entrenamiento se redujese drásticamente (aproximadamente solo diez minutos después de cargar todos los datos). La función *__len__* define la longitud del objeto de la clase y la función *__getitem__* coge el dato en función del índice *idx*.

- Módulo del modelo (*model.py*): define el modelo (red neuronal) que se va a utilizar. *Pytorch* define la clase *Module* para estos objetos. Se han redifinido las funciones *__init__* que inicializan las capas del modelo y *forward* que establece el camino que realiza la entrada *img* a través de las capas definidas anteriormente. Se explicará cada capa en las subsecciones posteriores según el tipo de red utilizada.

- Módulo de prueba (*test.py*): define la función *evaluate*, que coge los datos de entrada, los pasa por la red y los compara con sus etiquetas. Calcula la matriz de confusión y da una puntuación de precisión.

- Módulo de entrenamiento (*train.py*): realiza el entrenamiento ayudándose de las clases y funciones definidas en los anteriores módulos. Su diagrama de flujo se puede consultar en 24, se ha tratado de comentar el programa en **Anexos** con la misma terminología para facilitar su lectura. En primer lugar, tras cargar los datos de la partición correspondiente al entrenamiento y la validación

y realizar una serie de configuraciones necesarias, con un bucle for va recorriendo varias veces todos los vídeos de entrenamiento clasificados en distintos lotes. Un lote es un conjunto de vídeos. El número de vídeos incluido en cada lote se llama tamaño del lote (*batch size* en inglés) y es un parámetro del entrenamiento que depende, entre otras cosas, del tamaño de la memoria de la *GPU*. Como se puede ver en la figura 24, por cada lote que se pasa por el modelo se realiza una *backpropagation* para actualizar los parámetros de las redes neuronales. Otro parámetro importante en el entrenamiento es el número de épocas (*epoch* en inglés). Una época consiste en recorrer todos los datos de todos los lotes una vez. Por cada época que pasa se evalúa el modelo para dar su precisión. Se espera que la precisión del modelo aumente por cada época que pasa a medida que se van alcanzando los mínimos con el algoritmo de optimización hasta que converja a un valor, momento en el cual se sabrá que el modelo está entrenado.

Cabe destacar que aparte de estos dos parámetros de entrenamiento (tamaño del lote y número de épocas) existen otros que determinan el éxito del entrenamiento de la red, es decir, la capacidad del ingeniero para seleccionarlos puede hacer que cambie la puntuación del modelo. Estos son, entre otros el ritmo de aprendizaje (*Learning rate*). Como las redes neuronales se utilizan para tantas aplicaciones y hay tantos tipos distintos no hay un manual para escoger los mejores antes de realizar los experimentos. Para su selección, se suelen hacer muchos experimentos y se observan las tendencias al modificar un determinado parámetro. En los artículos de investigación se llegan a hacer cientos e incluso miles de pruebas para encontrar los parámetros que entrenan mejor el modelo. Sin embargo, en este trabajo por no poder acceder a máquinas tan potentes, ha resultado imposible realizar tantas pruebas como se hubiese querido para seleccionar los parámetros de entrenamiento óptimos.

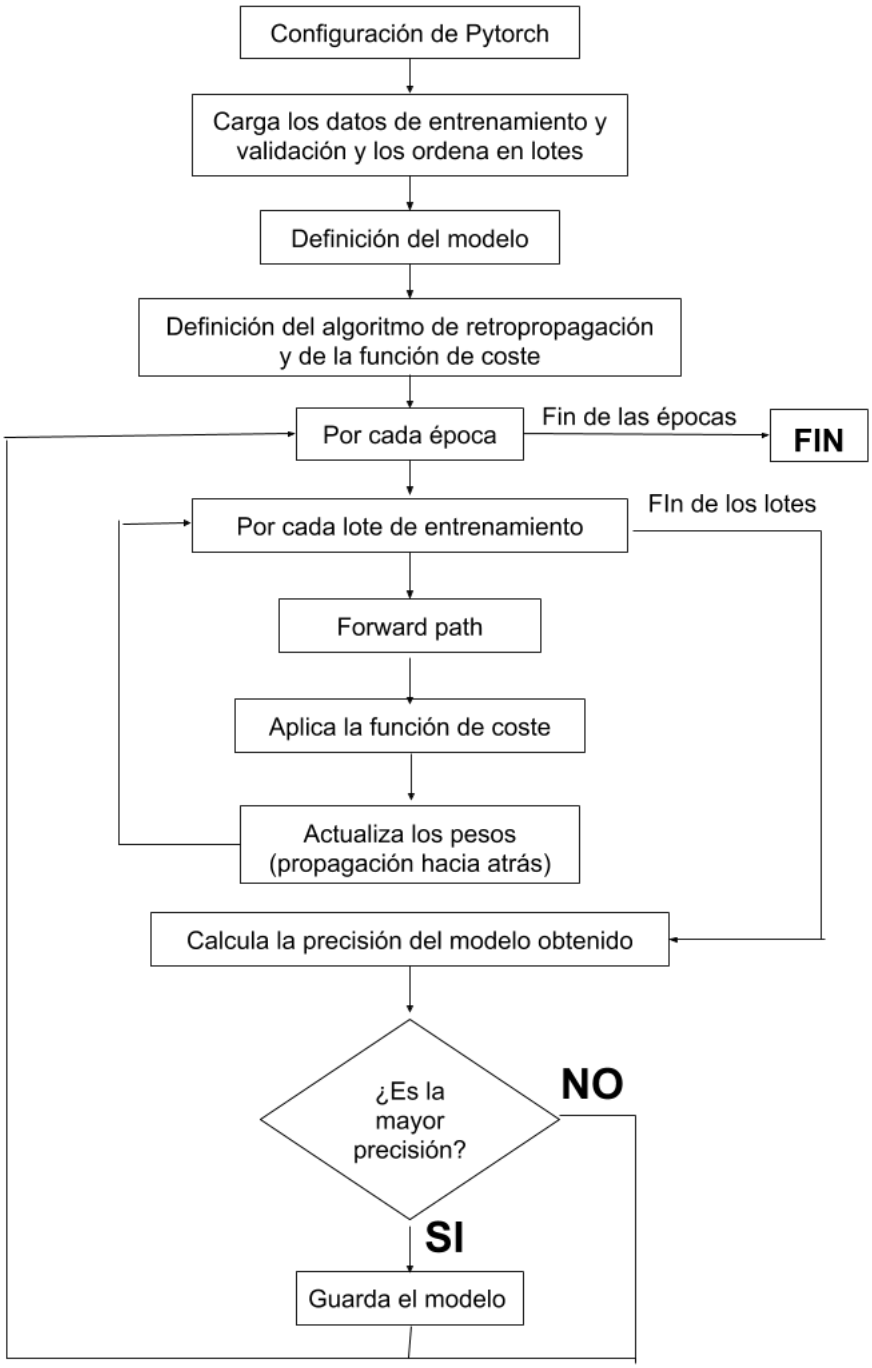


Figura 24 – Diagrama de flujo del módulo *train.py*

Como el objetivo 2 de la subsección de **Objetivos** es explorar como tratar la dimensión temporal de los vídeos con redes neuronales comparando el rendimiento de procesarlo mediante CNN y RNN, se han diseñado dos redes cuyo funcionamiento se basa en el preprocesado de cada cuadro de los vídeos para sacar unas características intermedias asociadas a esa imagen. Después, estas características se tratan de manera distinta dependiendo del tipo de red como se puede ver en las figuras 25 y 28.

3.2.1. CNN

En este tipo de red, las características intermedias obtenidas de cada cuadro, se procesan mediante el uso de Redes Neuronales Convolucionales. El diseño se ha inspirado del modelo de *Late fusion* del

artículo [10] que se muestra en la figura 21. La red se puede ver en la figura 25, se compone de una red troncal (una CNN) y el clasificador, que agrupa las características de los cuadros escogidos.

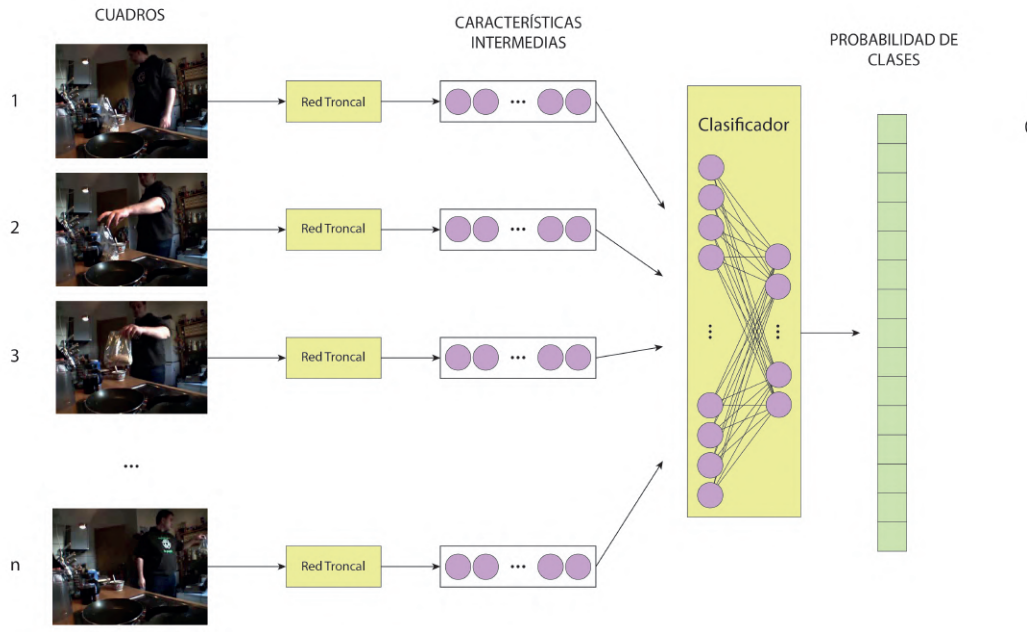


Figura 25 – Esquema de la red propuesta que trata la dimensión temporal con CNN

Este experimento, al ser menos complejo que el que trata la dimensión temporal con RNN, se ha utilizado para modificar varios factores de la red y así ilustrar cómo sería la metodología para diseñar un sistema basado en *Deep Learning* y de esta forma satisfacer el objetivo 1 de la sección *Introducción*. Esto consiste en variar algunos parámetros de la red para ver su efecto en el rendimiento. Sin embargo, debido a las limitaciones de Google Colab, en vez de basar el juicio solo en la precisión, se han tenido en cuenta otros elementos como el tiempo de entrenamiento o el tamaño de imágenes preprocesadas.

Además, como otro de los objetivos principales (3) es comparar la dimensión temporal, esta experimentación no se ha centrado en utilizar una red troncal que consiguiese extraer las características principales de manera eficaz para los datos utilizados, simplemente se han utilizado dos redes troncales que se han probado efectivas para la extracción de características en imágenes de otros conjuntos de datos.

Por tanto, el objetivo de esta experimentación es mostrar la metodología y definir unos parámetros de red que se puedan usar para el siguiente experimento, que basa la red en *RNN*

Los parámetros que se han modificado son:

- El número de imágenes que se procesan. Se han hecho experimentos con:
  - Dos cuadros
  - Cinco cuadros
  - Todos los cuadros
- La red troncal. Se han probado dos, se pueden ver en las figuras 26 y 27. En estos diagramas, las cajas representan las características intermedias de salida que se obtienen de las diferentes capas. Es decir, la primera caja son las características de salida de la primera capa. El color representa el tipo de capa: el rojo para las convolucionales, el verde para las normalizadoras, el azul para las de

agrupación, el morado para las ReLU y el amarillo para las completamente conectadas. Como se puede ver en ambas figuras, la salida es un vector de 1000 componentes (amarillo). Se ha rotado alguna caja para que la red cupiese en la imagen pero todas las dimensiones importantes están especificadas. Estas dos redes troncales son:

- La propia red propuesta en [10], como se puede ver en la imagen 26 consiste en: C(96,11,3)-N(96)-P(2)-C(256,5,1)-N(256)-P(2)-C(256,3,1)-C(384,3,1)-C(256,3,1)-MP(2). Donde C(d, f, s) es una capa convolucional en 2Dque usa d filtros (dimensión de la imagen de salida) de f×f con un paso(stride) de s, N(d) es una capa normalizadora que normaliza en cada una de las dimensiones d, MP(f) es una capa de agrupación por máximos (*max pooling*) entre grupos de f×f píxeles.
- Una red Resnet18 [6] preentrenada sobre el conjunto de datos *Imagenet*. Esta red ha demostrado tener muy bueno resultados para la clasificación de imágenes. Se le ha añadido una capa fully conected para afinar su rendimiento a nuestros datos. Como se puede ver en la figura 26, se compone de C(64,7,2,3)-N(64)-Relu-MP(3,2,1,1)-C(64,3,1,1)-N(64)-Relu-C(64,3,1,1)-N(64)-C(64,3,1,1)-N(64)-Relu-C(64,3,1,1)-N(64)-C(128,3,2,1)-N(128)-Relu-C(128,3,1,1)-N(64)-C(128,3,1,1)-N(128)-Relu-C(128,3,1,1)-N(128)-C(256,3,2,1)-N(256)-Relu-C(256,3,1,1)-N(256)-C(256,3,1,1)-N(256)-Relu-C(256,3,1,1)-N(256)-C(512,3,2,1)-N(512)-Relu-C(512,3,1,1)-N(512)-C(512,3,1,1)-N(512)-Relu-C(512,3,1,1)-N(512)-AAP(512)-FC(512,1000)-FC(1000,11000). Donde C(d,f,s,p) es una capa convolucional en 2D que usa d filtros (dimensión de la imagen de salida) de $f \times f$ con un paso(stride) de s, padding p y sin vector b de *bias*. N(d) es una capa normalizadora que normaliza en cada una de las dimensiones d con una eps de 10^{-5} y un momento de $0{,}1$. ReLU es la aplicación de la función de activación. MP(f, s, p, d) es una capa de agrupación por máximos entre grupos de f×f píxeles con un paso de s, un relleno de p y una dilatación de d. AAP(o) es una capa de agrupación por media adaptable *Adaptative Average Pool*(que coge la entrada y hace la media para producir un tensor de ox1x1. FC(i, o) es una capa completamente conectada (*fully connected* con i características de entrada y o características de salida.

- El clasificador, basado en NN con dos perspectivas distintas:
  - Hacer la media de las características obtenidas para cada frame de un vídeo y añadir algunas capas *fully conected* para clasificar.
  - Concatenar las características y añadir algunas capas *fully conected* al final.

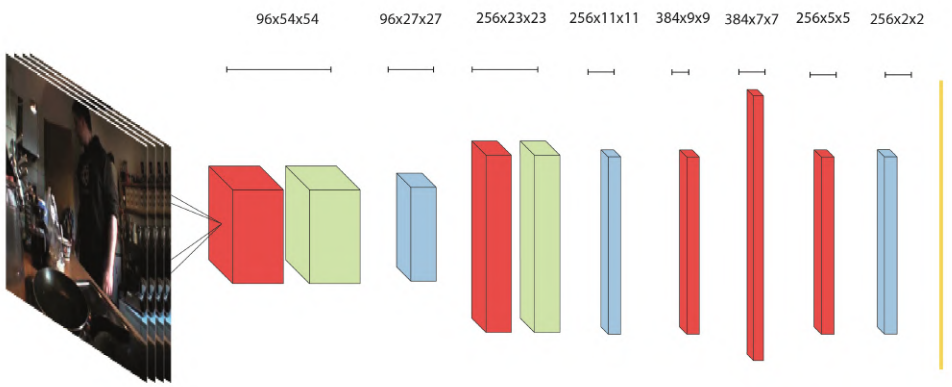


Figura 26 – Esquema de la red troncal basada en [10]

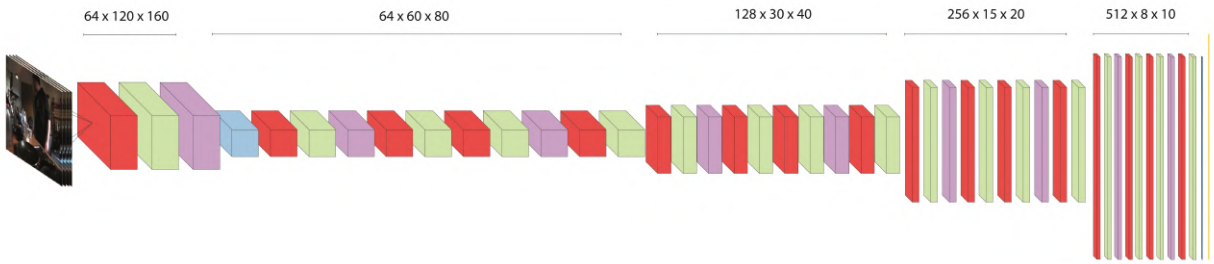


Figura 27 – Esquema de la red troncal implementada con una Resnet

Se podría, por fuerza bruta, probar cada combinación para ver cual da la mejor precisión. Sin embargo, como el acceso a GPUs no ha sido ilimitado y pese a que Google Colab es suficiente, se ha optado por probar sólo algunas y razonar cual sería la mejor combinación para probar a continuación. El objetivo, a parte de conseguir el sistema con la mejor precisión posible, era encontrar la red que tuviese la mejor relación entre tiempo de entrenamiento y el tamaño de los datos preprocesados para poder aplicar la misma red troncal a la RNN y que estos factores, escalados a sus necesidades no tuvieran valores inabordables.

Para cada modelo, se han realizado varios entrenamientos con el fin de ajustar lo mejor posible los parámetros de entrenamiento entre los que se encuentra el *learning rate* y el tamaño del lote. También es posible variar el tipo de función de coste utilizado y el algoritmo que realiza la backpropagation pero estos se han mantenido constantes porque se han seleccionado la entropía cruzada entre dos vectores como función de coste y el optimizador *Adam*, ya que dan buenos resultados para problemas de clasificación.

A continuación se detalla este proceso para ilustrar la metodología que se podría seguir al diseñar un sistema basado en Deep Learning (objetivo 1), que consiste en variar los parámetros de la red y realizar varios entrenamientos con distintos parámetros de entrenamiento para encontrar la mejor red.

En primer lugar, se empezó utilizando el modelo utilizado en [10], procesando dos cuadros, con la red troncal propuesta en el mismo artículo y utilizando una capa fully conected al final. Para ahorrar tiempo de entrenamiento, el preprocesado consistió en pasar los vídeos a tensores (objecto por excelencia de *Pytorch* que consiste en una matriz, en nuestro caso de tamaño nº cuadros (2) $\times$ nº canales (3) $\times$ altura $\times$ anchura). De está forma ahorrabamos el tiempo que el programa tardaba en abrir el vídeo en *Python*, transformarlo en un array de la librería *numpy* para después convertirlo en un tensor de aproximadamente 40 minutos para cada época. Los vídeos de las cuatro particiones ocupaban más de 25 GB. Aunque se realizó el experimento, rápidamente se descartó está opción, ya que la RNN necesita todos los cuadros del vídeo y el tamaño de datos que se manejaría sería de más de 100 GB. Los resultados tampoco fueron excesivamente buenos con una precisión que nunca alcanzaba el $8\,\%$.

Tras este experimento, se decidió utilizar una Resnet18 preentrenada como red troncal. Por tanto, al no haber *backpropagation* en la red troncal, no era necesario que los vídeos se pasaran por la red troncal en el entrenamiento. Además, a la vista de los malos resultados anteriores, en vez de coger sólo dos cuadros, se utilizaron cinco. En este caso, el preprocesado consistía en pasar 5 cuadros de $3 \times 320 \times 240$ por la Resnet18 para obtener 1000 características para cada uno de los 5 cuadros. Se ahorraba un espacio considerable, pues ahora las características de cada vídeo estaban almacenadas en un archivo de 20 KB, que contenía la matriz de nº cuadros (5) $\times$ nº de características (1000). En total, unos 125 MB para las cuatro particiones. En definitiva, si 0 es el primer cuadro y n el último, el modelo consistía en coger los cuadros 0, $\frac{n}{4}$, $\frac{n}{2}$, $\frac{3n}{4}$ y n, extraía sus características a través de la Resnet18, que se afinaba con una capa fully conected al final de también 1000 salidas. Si el vídeo tenía menos de 5 cuadros había alguno que se repetía. Después se concatenaban las características obteniendo un vector de 5000 elementos para meterlos en otra capa fully conected con 15 datos de salida, probabilidad de las 15 clases. Tras varios entrenamientos en los que se ajustó el ritmo de aprendizaje (*learning rate*) para que el entrenamiento

fuese óptimo, se llegaron a precisiones de hasta el $12\,\%$.

A continuación, se pensó que la puntuación podía ser tan mala debido a que la manera de fusionar las características de los cuadros no era buena para la nueva red troncal y se probó con fusionar las características haciendo la media de ellas en cada vídeo, de manera que la última era una capa *fully conected* de 1000 a 15. Los resultados empeoraron y la precisión alcanzó solo un $8\,\%$.

Una posible explicación es que no se habían cogido los suficientes cuadros para realizar este tipo de fusión, se probó eliminar la capa que afinaba la red troncal y hacer la media de las características de todos los cuadros obtenidas de procesar el vídeo a través de la red Resnet18 para después utilizar una capa *fully conected* de 1000 a 15 como en el caso anterior. Este experimento mejoró los resultados del caso anterior pero llegaron a apenas un $11\,\%$.

Sin embargo, la conclusión de estos dos últimos experimentos fue que concatenar las características daba mejor resultado, por lo que se volvió a coger sólo 5 cuadros utilizando una red troncal que consistía en una Resnet18 preentrenada con una capa fully convolutional para afinarla. Lo que se cambió fue el clasificador. Se hicieron experimentos sucesivos aumentando el número de capas completamente conectadas del clasificador. Con un clasificador consistente en 3 capas fully conected que bajan las características de 5000 a 2500, de 2500 a 1000 y de 1000 a 15 se consiguió una precisión de casi un $15\,\%$.

Como se ha podido comprobar, con pequeñas modificaciones del modelo, se ha podido aumentar su precisión a casi el doble de la precisión original. Evidentemente, los resultados no son los mejores si se busca utilizar este modelo en un entorno real. Esto puede deberse a que la red troncal no obtiene características suficientemente buenas para distinguir entre una clase y otra. En la sección de **Resultados y Análisis** se ahondará más en este análisis con ayuda de otras métricas. Se ha optado por mostrar sólo los resultados de la última red, debido a que se ha considerado que lo interesante de los modelos anteriores era únicamente su precisión y el tamaño de los datos preprocesados para ilustrar la metodología.

3.2.2. RNN

El objetivo de este experimento es comparar el tratamiento de la dimensión temporal entre una red basada en CNN y RNN (objetivo 3). Por ello, se utiliza la red troncal con mejor rendimiento de la sección anterior. De esta forma, fijando la red troncal, se puede comparar de manera efectiva y se evita tener que entrar en consideraciones del funcionamiento de la red troncal.

La red utilizada se puede ver en la figura 28. Como la red troncal estaba preentrenada, el módulo de *train.py* solo debía cargar las características que se habían preprocesado anteriormente. El funcionamiento por tanto, es igual que para la red basada en CNN. En este caso, lo único que se entrena es el módulo A, que se repite para todos los cuadros que tenga el vídeo, por lo que no hay necesidad de fijar un número de cuadros para todos los vídeos (n es el número de cuadros de cada vídeo y, de esta forma, es variable). Para facilitar la interpretación se ha omitido una capa completamente conectada a la salida del último módulo A en la figura 28, que es la que se encarga de la clasificación.

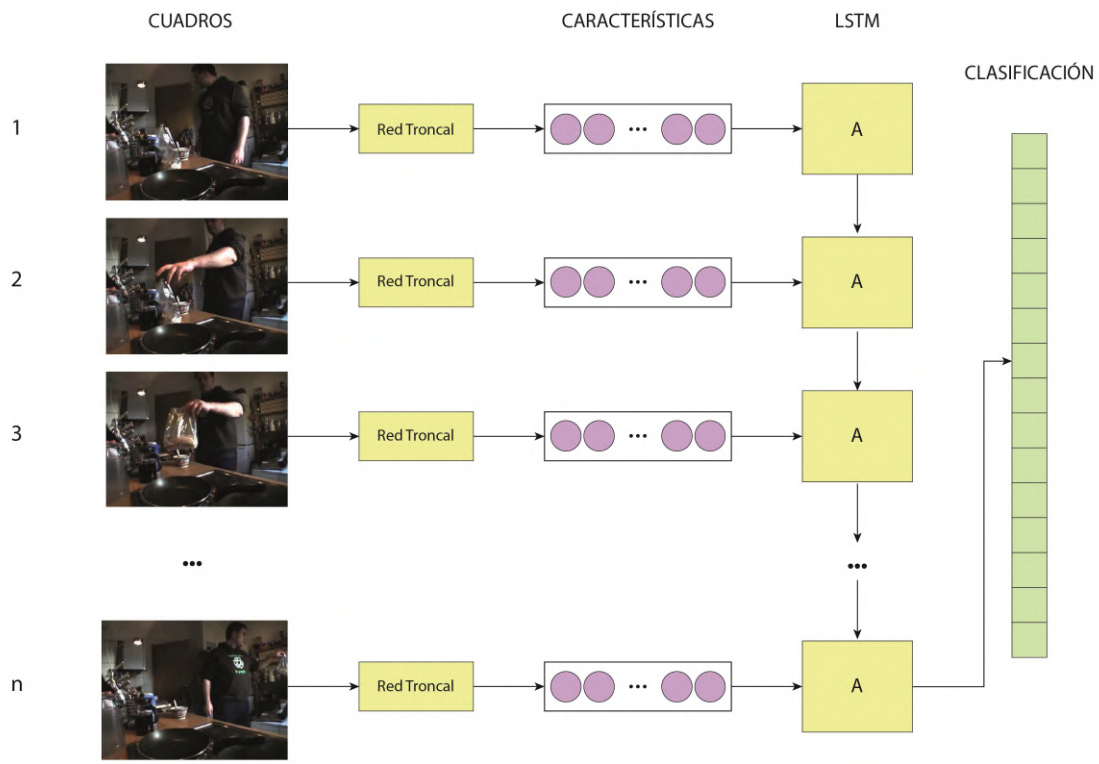


Figura 28 – Esquema de la red propuesta que trata la dimensión temporal con RNN

Como en los casos anteriores, se ha probado con varios parámetros de entrenamiento para asegurar que la red se entrena bien, aunque, aprovechando que los parámetros de red estaban fijos en este caso, se ha probado con muchos valores distintos de *learning rate* y *weight decay* en el optimizador. En la sección de **Resultados y Análisis** se muestra la mejor obtenida. Su tasa de aprendizaje es de $0{,}00002$ y un *weight decay* de $0{,}001$.

3.3. Uso

Esta fase consiste en pasar a través de la red seleccionada en la fase de entrenamiento el conjunto de vídeos de prueba. Se ha utilizado la métrica de top-3 para calcular su precisión. Esto es, considerar acierto si la clase a la que pertenece el vídeo está entre las tres mejores predicciones. Además, como se ha intentado ejecutar el programa en la misma sesión de Google Colab, se ha medido también el tiempo de ejecución para comparar qué red es más rápida, si la basada en CNN o la basada en RNN.[4].

Para ello, no se ha hecho ningún preprocesado del vídeo. En esta parte, se contaba el tiempo desde cargaba el vídeo, pasaba por la red troncal hasta hacía una predicción. De esta forma se comprueba si el sistema puede funcionar en línea, es decir, si puede funcionar a la vez que se está grabando el vídeo.

[4]Hubiese sido interesante coger está métrica de tiempo para el entrenamiento también pero Google Colab te asigna aleatoriamente a máquinas virtuales distintas cada cierto tiempo. Coger el tiempo de entrenamiento sobre ordenadores distintas no es útil, pues la capacidad de computación cambia

4. Resultados y Análisis

4.1. CNN

La figura 29 muestra el valor de la función error con respecto al número de iteraciones. Una iteración corresponde a un paso de un lote por la red para predecir sus clases y su propagación hacia atrás. La única diferencia entre este valor y el valor de la función de coste es que la función de coste se refiere normalmente a la media de la función error de todas las iteraciones de una misma época. Por tanto, como en la función de coste, lo que se busca es que la función error vaya disminuyendo en cada iteración.

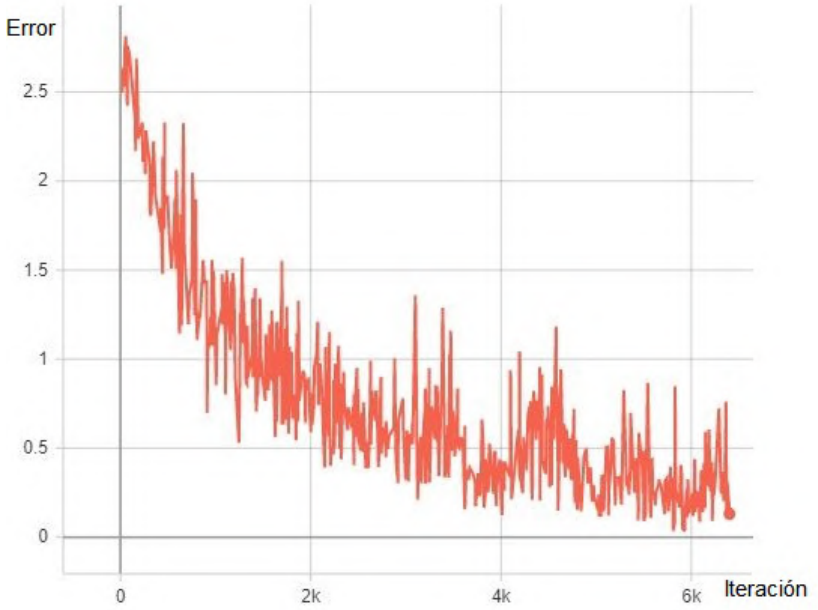


Figura 29 – Valor de la función error con respecto a las iteraciones

En la figura 29 se cumple esta tendencia a la baja.

En la figura 30 se muestra la variación de precisión con respecto a cada una de las 64 épocas. Como se ha comentado anteriormente, una época es el paso de todos los datos por la red. Esta métrica es varía entre $16\,\%$ y $11,5\,\%$. Idealmente, este valor, debería subir en cada época hasta converger a un valor, donde se considera que la red se ha terminado de entrar. Si, tras estabilizarse en un valor, empieza la precisión vuelve a bajar y la función error continúa bajando, se está produciendo un efecto llamado sobreajuste *overfitting*. En el cual el modelo se ajusta demasiado a los datos y los umbrales de decisión no son capaces de distinguir bien entre nuevos datos de una clase que tienen alguna característica similar a las de otra clase.

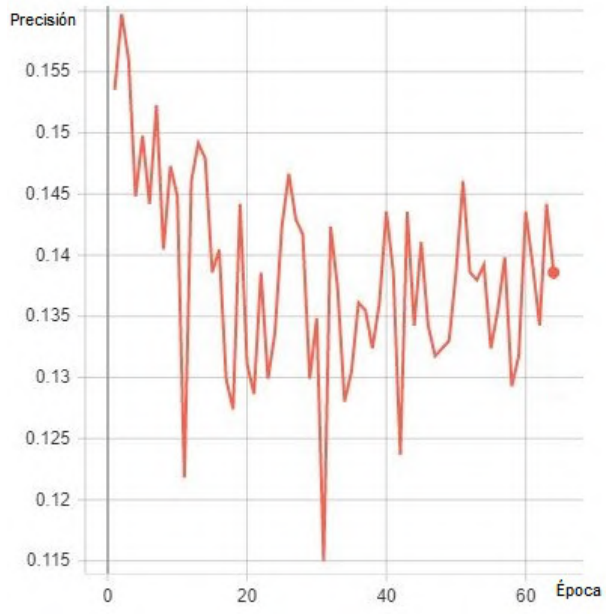


Figura 30 – Valor de la precisión para cada época

En la imagen 30 no se aprecia ninguna de estas dos tendencias. Lo que si ocurre es que dar valores altos en las épocas iniciales, sufre una bajada y empieza a variar entre $12,5\,\%$ y $15\,\%$. Esto ocurre porque en las primeras épocas, la red sólo detecta las clases de las que hay más datos de entrenamiento. Aunque estos datos, son mejores, se ha optado por descartarlos porque realmente la red no estaba funcionando como debería funcionar. Por tanto, se ha seleccionado el modelo de la época 26, que tiene una precisión de $14,67\,\%$ sobre los datos de validación. Esta precisión no es lo suficientemente buena para un modelo de *Deep Learning*, por lo que a continuación se analizan las razones. Todos los resultados y análisis que se muestran a continuación son de este modelo por ser de mejor precisión.

Este efecto, que al principio solo se predigan las clases más comunes del entrenamiento suele ocurrir en todas las redes, lo que normalmente no ocurre es que la precisión después de ellos baje. Aunque esta tendencia no se pueda atribuir a un sobreajuste porque no es lo suficientemente clara, se puede apreciar un efecto similar (de decrecimiento tras las primeras épocas) pero debido a otras razones. Para explicarlo conviene calcular la matriz de confusión, que se muestra en el cuadro 3. La fila i de la columna *Total et* es el número total de vídeos etiquetados con i y la columna j de la fila *Total pred* es el número de imágenes cuya predicción es j.

0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Total et
0	41	13	18	6	23	20	8	12	4	0	0	1	1	7	1	155
1	8	15	22	5	16	15	8	12	5	1	1	1	3	5	2	119
2	22	14	27	7	13	25	7	29	11	1	2	4	6	8	7	183
3	10	5	11	12	12	17	6	19	2	0	5	0	3	3	0	105
4	30	8	22	8	37	27	12	20	2	0	0	4	3	6	0	179
5	34	9	27	13	29	62	11	33	7	0	3	1	1	10	3	243
6	13	11	9	11	9	10	5	9	4	0	4	0	2	2	1	90
7	14	15	17	7	11	20	6	25	4	0	0	1	1	4	3	128
8	13	8	7	7	6	8	7	9	4	2	2	2	1	1	2	79
9	6	4	9	2	5	10	5	7	3	0	1	3	1	5	1	62
10	5	1	7	2	3	14	2	6	7	0	2	0	0	1	0	50
11	6	4	9	6	6	13	6	7	3	0	1	1	0	0	0	62
12	4	2	3	4	2	8	3	4	3	0	2	0	3	0	0	38
13	4	7	10	3	12	14	6	12	5	0	0	2	1	5	0	81
14	5	3	5	4	2	5	3	4	0	0	0	0	1	1	2	35
Total pred	215	119	203	97	186	268	95	208	64	4	23	20	27	58	22	

Cuadro 3

Al igual que pasa con el sobreajuste, de la matriz de confusión se puede concluir que la red confunde las clases. Este hecho se puede atribuir a la naturaleza de los datos y de la red troncal utilizada. Como se comentaba, la naturaleza de los datos utilizados es completamente distinta a la de los datos sobre los que se ha entrado la red *Resnet* (Conjunto de datos *Imagenet*). En *Imagenet* un humano podría diferenciar entre clases perfectamente con sólo una imagen. Por ejemplo, en la figura 31 se puede apreciar a un perro. Sin embargo, para el conjunto de datos de este trabajo, las imágenes estáticas sobre las que se aplica la red troncal pueden ser muy parecidas. Por ejemplo, en la imagen 32 puede parecer que está cortando pero tal vez esté poniendo, pelando o cogiendo. Este problema se debe a la agrupación de clases que hicimos en la etapa de preprocesado para adaptar los datos a las capacidades de computación.

Figura 31 – Imagen de un perro del conjunto *Imagenet*

Figura 32 – Imagen de una receta del conjunto *Breackfast*

Como se puede apreciar en las casillas $(4, 5)$ y $(5, 4)$ de la matriz de confusión, en dos actividades que en estático pueden resultar similares como son coger (etiqueta 4) y poner (etiqueta 5) la red no las diferencia bien y la clase que más detecta para la clase coger después de la propia clase es poner y viceversa. No es en la única pareja de clases que ocurre y también se puede apreciar un efecto parecido entre la clase 6 (añadir) y la 5 (poner).

Por último, se han visualizado los datos de entrada a la última capa lineal. Para cada vídeo, son 1000 características intermedias. Para reducir su dimensión de 1000 a 2 se ha utilizado el método t-SNE. El resultado se muestra en 33. En el gráfico, no se aprecia ningún cúmulo de puntos del mismo color, lo que confirma que la red no tiene mucha capacidad para diferenciar clases.

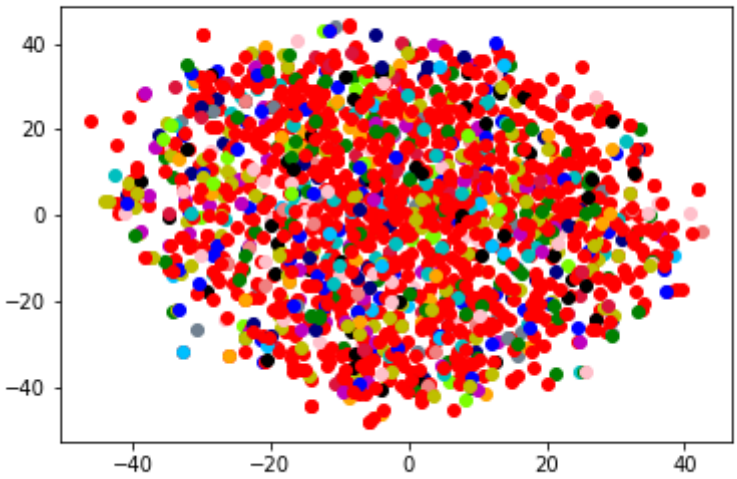


Figura 33 – Visualización de las características de entrada de la última capa

Finalmente, el tiempo en el que este tipo de red tardaba en procesar los $136{,}746$ cuadros de los 1636 vídeos del conjunto de prueba era 3651 s (60 minutos y 51 segundos). Merece la pena recordar que en este caso sólo se procesaban 5 cuadros por vídeo pero se considera que se han sido $136{,}746$ cuadros porque la selección de esos 5 cuadros es parte de la red. Por tanto, el tiempo necesario esta red puede procesar $37, 45$ cuadros por segundo, un valor que permite procesar vídeos de 25 fps a medida que se van grabando. La métrica *top-3* para este conjunto de prueba es de $36, 19\,\%$ (prec $16, 50\,\%$)

4.2. RNN

En este caso se ha escogido el modelo de la época 84 de 100, que tiene una puntuación de $25,79\,\%$ sobre el conjunto de datos de validación. Para este experimento se puede consultar su función error en función de la iteración en la figura 34. En esta figura se comprueba que tiene una tendencia descente, lo que garantiza el entrenamiento de la red.

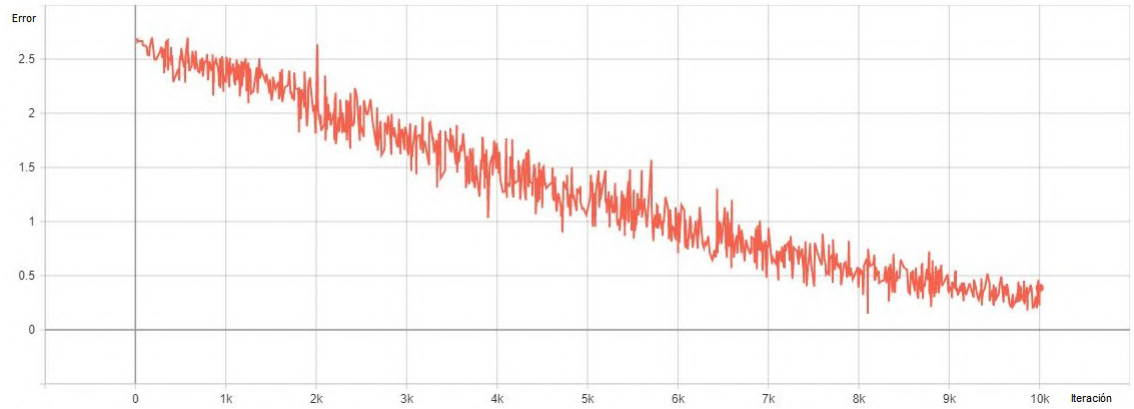


Figura 34 – Valor de la precisión para cada época

En la figura 35 se muestra la evolución de la métrica de precisión para las distintas épocas. El resultado es el esperado, aumentando en cada iteración hasta que converge a un valor alrededor de $25\,\%$. La precisión nunca disminuye por lo que se puede concluir que no hay sobreajuste. Aunque, como se puede comprobar en la figura 35 había valores superiores a los de la época 84 (en la época 44 se llegaba a superar una precisión del $27\,\%$), se ha optado por desecharlos debido a que su matriz de confusión mostraba que no se estaban prediciendo datos de algunas clases, por lo que su rendimiento no era tan bueno.

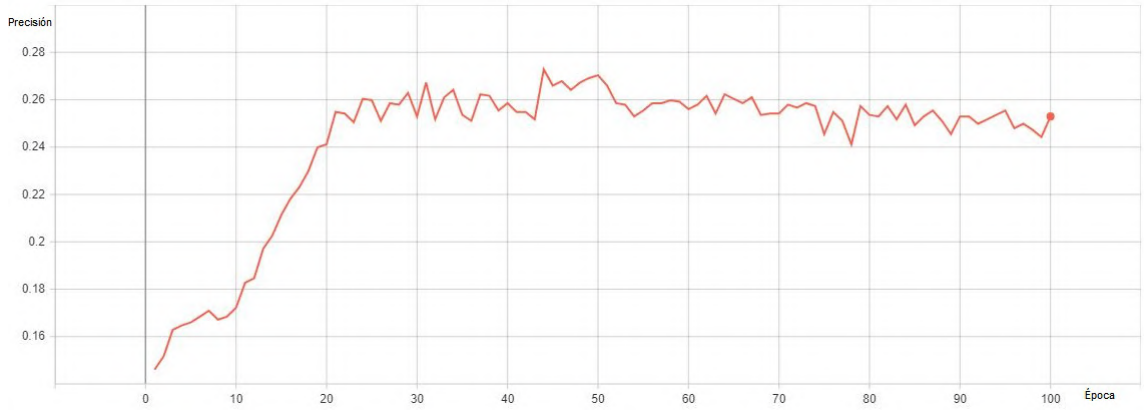


Figura 35 – Esquema de la red propuesta que trata la dimensión temporal con RNN

Al igual que en el caso anterior, un $25\,\%$ de precisión tampoco es ideal. Sin embargo aumenta considerablemente respecto al anterior. A continuación se analizarán las razones.

La matriz de confusión para este modelo de la época 84 se muestra en el cuadro 4. En esta tabla también se aprecia un comportamiento del modelo parecido al de la red basada en CNN aunque ha crecido el número de aciertos.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Total gt
0	84	3	10	2	18	23	8	3	2	0	0	0	0	2	0	155
1	2	22	17	7	14	28	8	9	3	1	1	0	0	7	0	119
2	2	16	46	12	11	25	12	35	10	2	3	2	1	5	1	183
3	2	16	14	6	13	13	10	20	3	1	0	0	1	5	1	105
4	29	5	6	2	90	37	6	4	0	0	0	0	0	0	0	179
5	36	21	30	5	35	82	10	18	1	0	0	2	0	3	0	243
6	1	12	11	8	8	15	9	12	5	0	1	0	0	8	0	90
7	7	7	26	5	11	23	8	21	8	0	4	3	1	4	0	128
8	0	2	8	0	1	8	2	3	39	0	5	3	3	5	0	79
9	1	2	18	3	3	9	3	6	10	0	2	1	1	3	0	62
10	0	1	8	1	1	4	0	12	13	0	2	3	1	3	1	50
11	2	3	14	6	0	4	3	5	9	0	5	5	2	3	1	62
12	0	1	8	1	0	4	1	7	7	0	4	2	2	1	0	38
13	1	12	8	10	7	12	5	13	5	1	2	0	1	4	0	81
14	0	2	4	3	1	4	0	6	6	0	0	2	1	3	3	35
Total pred	167	125	228	71	213	291	85	174	121	5	29	23	14	56	7	

Cuadro 4

En la representación de la figura 36 tras la aplicación del algoritmo t-SNE tampoco se aprecian grupos de colores diferenciados pero empieza haber distancia entre algunos puntos apreciándose formas. Esto puede deberse a que la red troncal no consigue producir características que diferencien bien las clases.

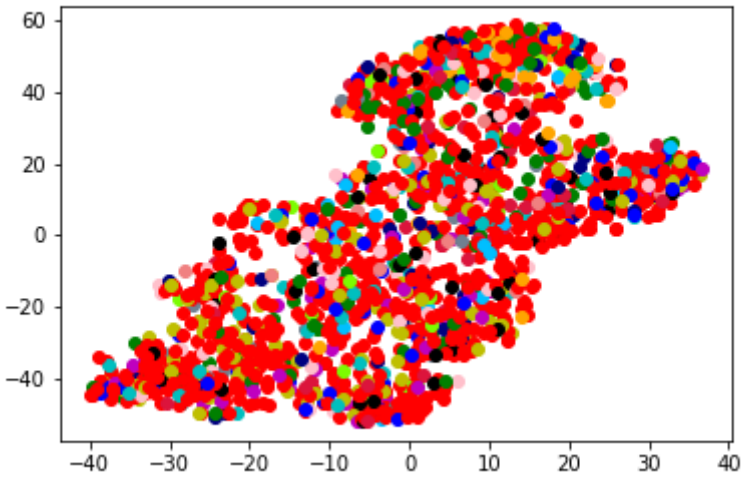


Figura 36 – Visualización de las características de entrada de la última capa

El tiempo que tardaba en procesar de principio a fin los 136746 cuadros del conjunto de prueba era de 3,800 s (63 minutos y 21 s) por lo que esta red procesa casi 36 cuadros por segundo, permitiendo su

aplicación en línea en vídeos de 25 fps. La métrica *top-3* para este conjunto es de $55,13\,\%$

4.3. Comparación entre CNN y RNN

Al compartir la red troncal y haber una diferencia de puntuación considerable entre ambos (de más de $10\,\%$ en precisión sobre el conjunto de validación) se puede concluir que las Redes Neuronales Recurrentes son una mejor opción a la hora de tratar la dimensión temporal. Es un resultado esperado, ya que las RNN se originaron para aprender secuencias temporales y tratar los vídeos con CNN que explotan las dimensiones espaciales (si son en 2D) o simplemente datos (si son en 1D).

En cuanto a la comparación de las matrices de confusión, se aprecia un funcionamiento similar. Sin embargo, si se compara con el número de predicciones de cada clase en la fila *Total pred* de la matriz de confusión del sistema RNN (tabla 4) con el número de cuadros que hay en cada clase en la tabla 2, se puede comprobar que las las clases 3, 9, 10, 11, 12, 13, 14 y 15 tienen menos predicciones son aquellas que cuyo número de cuadros es inferior. En cambio, en la matriz de confusión 3 no ocurre esto.

En las representaciones en 2D, se empiezan a apreciar cúmulos en la que muestra las características del sistema basado en RNN, aunque no de la misma clase. Este efecto está asociado a la mayor puntuación de ese sistema.

Finalmente, en el tiempo de uso no se aprecia una mejora apreciable por usar una red u otra. Mientras que el sistema diseñado basado en CNN puede procesar $37,45$ cuadros por segundo, el basado en RNN 36 cuadros por segundo. La diferencia de velocidades no es significativa comparada con la mejora en la métrica de precisión (un $10\,\%$ a favor de la RNN) o la métrica *top-3* (casi un $20\,\%$ a favor de RNN también).

5. Conclusión

En cuanto a la exploración sobre la metodología a seguir para diseñar un sistema basado en *Deep Learning*, en los experimentos de la red, se ha visto que hay dos grupos de parámetros que se pueden modificar. Por una parte, los parámetros de la red son elementos propios del sistema como pueden ser las capas de la red o la manera de escoger los datos de entrada. Estos parámetros modifican el diseño de la red y, por ello, para cada variación se ha de entrenar la red varias veces para encontrar los parámetros del entrenamiento (*learning rate*, *batch size*...) que dan un mejor resultado. De esta forma, la metodología consiste en realizar varios entrenamientos con parámetros de entrenamiento distinto para cada combinación de parámetros de red con el objetivo de encontrar la que mejor se adapte al problema concreto que se quiere resolver.

También se han propuesto distintas estrategias para disminuir el tiempo de entrenamiento de una red entre las que destacan el preprocesar los datos para guardarlos directamente como tensores y cargar estos datos en la inicialización del *Dataloader* para evitar tener que hacerlo en cada iteración. Si hay problemas de almacenamiento o capacidad de procesamiento, una buena estrategia puede ser también utilizar una red preentrenada que se haya demostrado efectiva para resolver el mismo problema y simplemente entrenar la última capa completamente conectada para afinarla.

Así mismo, se ha propuesto un sistema para clasificar vídeos. Este sistema consiste en el procesado de los cuadros con el objetivo de obtener unas características intermedias. Después, se usan esas características en un clasificador. Este clasificador es el que se encarga del unir los cuadros y por tanto de procesar la dimensión temporal del vídeo. Para esta tarea conviene usar una Red Neuronal Recurrente antes que una Red Neuronal Convolucional. En el sistema propuesto, había una diferencia de más de un $10\,\%$ de precisión sobre el conjunto de datos de validación y un $20\,\%$ con la métrica top-3 sobre el conjunto de prueba. Además, el tiempo de uso de más de 35 cuadros por segundo garantiza que ambas redes pueda aplicarse en línea con vídeos de 25 fps como los del conjunto de datos utilizado.

Por último, otro de los objetivos del trabajo que se ha cumplido ha sido profundizar en el uso de Pytorch. Se ha implementado el entrenamiento de las redes con una utilización básica de la librería redefiniendo las clases *Dataloader* y *Module*, implementando un entrenamiento de varias épocas y utilizando la potencia que proporcionan las GPUs con el método *.cuda()*. Se han usado además, funciones para guardar los modelos entrenados y los datos preprocesados. Debido a las características del entrenamiento, también se han utilizado herramientas más avanzadas que proporciona *Pytorch* como el uso de la funcion *my_collate* para cargar datos de distintas dimensiones con el *Dataloader* o la concatenación de *Dataloaders* para cargar dos subconjuntos de datos bajo entrenamiento. Además, se ha implementado en Pytorch una herramienta propia de otra librería de *Deep Learning*, como es Tensorboard de Tensorflow. Esta herramienta permite realizar gráficas con datos del entrenamiento que se actualizan en tiempo real para comprobar su rendimiento.

Por último, la realización de este trabajo y la búsqueda de material para él, me ha permitido consultar multitud de artículos científicos sobre Redes Neuronales, revisiones y blogs para comprender su funcionamiento. Su lectura me ha permitido acostumbrarme a aprender los conocimientos directamente de las fuentes primarias y conocer de manera más rigurosa los conceptos. De esta forma, he evitado consultar otro tipo de fuentes, de carácter más divulgativo.

5.1. Futuras líneas de trabajo

En primer lugar y tras comprobar que para tratar la dimensión temporal conviene utilizar RNN, con el fin mejorar la puntuación del sistema previsto, una futura investigación podría ir orientada a buscar una nueva red troncal que sí produjese características muy diferentes entre las clases. Si se quiere continuar con redes que clasifiquen las imágenes, un buen método sería procesar solo la zona del cuadro donde se produce la acción. Otro planteamiento que merece la pena explorar es que la red troncal detectase

el esqueleto de la persona realizando la acción (como propone [2]), de tal forma que las características intermedias fuesen los lugares de la imagen donde se encuentran las distintas partes del cuerpo del actor. En este caso, la clasificación se podría realizar atendiendo a la secuencia de movimientos, algo de lo que indudablemente se beneficiaría nuestra red, ya que diferentes clases tienen una fuerte componente secuencial.

Si se quiere seguir investigando sobre el tratamiento de la dimensión temporal, se podría probar una red basada en las Redes Neuronales Convolucionales en 3D propuestas en [8]. Aunque, estas redes tratan esa tercera dimensión como una dimensión temporal más, por lo que se utilizan en imágenes médicas en 3D, convendría compararlas con las RNN.

Finalmente, otra manera de continuar con la investigación, se podría utilizar un conjunto de datos en las que las clases etiquetadas no tuviesen una componente secuencial tan fuerte para comprobar si merece la pena usar Redes Neuronales Recurrentes o usando un algoritmo basado en CNN es suficiente.

6. Bibliografía

Referencias

[1] Donahue, J.; Anne Hendricks, L.; Guadarrama, S.; Rohrbach, M.; Venugopalan, S.; Saenko, K. y Darrell, T. *Long-term recurrent convolutional networks for visual recognition and description*. En *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 2625–2634 (2015).

[2] Du, Y.; Wang, W. y Wang, L. *Hierarchical recurrent neural network for skeleton based action recognition*. En *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 1110–1118 (2015).

[3] Dumoulin, V. y Visin, F. *A guide to convolution arithmetic for deep learning*. ArXiv e-prints (mar 2016).

[4] Glorot, X.; Bordes, A. y Bengio, Y. *Deep sparse rectifier neural networks*. En *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, páginas 315–323 (2011).

[5] Guha, T. y Ward, R. K. *Learning sparse representations for human action recognition*. IEEE transactions on pattern analysis and machine intelligence, 34(8):1576–1588 (2011).

[6] He, K.; Zhang, X.; Ren, S. y Sun, J. *Deep residual learning for image recognition*. En *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 770–778 (2016).

[7] Hochreiter, S. y Schmidhuber, J. *Long short-term memory*. Neural computation, 9(8):1735–1780 (1997).

[8] Ji, S.; Xu, W.; Yang, M. y Yu, K. *3d convolutional neural networks for human action recognition*. IEEE transactions on pattern analysis and machine intelligence, 35(1):221–231 (2012).

[9] Karpathy, A. *The unreasonable effectiveness of recurrent neural networks*. `https://karpathy.github.io/2015/05/21/rnn-effectiveness/` (May 2015).

[10] Karpathy, A.; Toderici, G.; Shetty, S.; Leung, T.; Sukthankar, R. y Fei-Fei, L. *Large-scale video classification with convolutional neural networks*. En *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, páginas 1725–1732 (2014).

[11] Kuehne, H.; Arslan, A. y Serre, T. *The language of actions: Recovering the syntax and semantics of goal-directed human activities*. En *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 780–787 (2014).

[12] LeCun, Y.; Bengio, Y. y Hinton, G. *Deep learning*. nature, 521(7553):436–444 (2015).

[13] LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W. y Jackel, L. D. *Backpropagation applied to handwritten zip code recognition*. Neural computation, 1(4):541–551 (1989).

[14] Liu, L.; Shao, L.; Li, X. y Lu, K. *Learning spatio-temporal representations for action recognition: A genetic programming approach*. IEEE transactions on cybernetics, 46(1):158–170 (2015).

[15] Maaten, L. v. d. y Hinton, G. *Visualizing data using t-sne*. Journal of machine learning research, 9(Nov):2579–2605 (2008).

[16] Olah, C. *Understanding lstm networks*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (August 2015).

[17] Olah, C. *Understanding convolutions*. http://colah.github.io/posts/2014-07-Understanding-Convolutions/ (July 204).

[18] Ramanathan, V.; Huang, J.; Abu-El-Haija, S.; Gorban, A.; Murphy, K. y Fei-Fei, L. *Detecting events and key actors in multi-person videos*. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 3043–3053 (2016).

[19] Rohrbach, M.; Amin, S.; Andriluka, M. y Schiele, B. *A database for fine grained activity detection of cooking activities*. En *2012 IEEE Conference on Computer Vision and Pattern Recognition*, páginas 1194–1201. IEEE (2012).

[20] Sargano, A. B.; Angelov, P. y Habib, Z. *A comprehensive review on handcrafted and learning-based action representation approaches for human activity recognition*. applied sciences, 7(1):110 (2017).

[21] Simonyan, K. y Zisserman, A. *Two-stream convolutional networks for action recognition in videos*. En *Advances in neural information processing systems*, páginas 568–576 (2014).

[22] Szeliski, R. *Computer vision: algorithms and applications*. Springer Science & Business Media (2010).

[23] Tran, D.; Bourdev, L.; Fergus, R.; Torresani, L. y Paluri, M. *Learning spatiotemporal features with 3d convolutional networks*. En *Proceedings of the IEEE international conference on computer vision*, páginas 4489–4497 (2015).

7. Anexos

7.1. Impacto del trabajo

Como el trabajo se puede considerar el comienzo del diseño de un sistema que clasifique vídeos, si en esta sección se considerara un sistema para cualquier tipo de vídeo, el análisis del impacto se complicaría pues el entorno de aplicación podría ser muy distinto. Evidentemente, no es igual la clasificación de vídeos deportivos con el fin de almacenarlos que la clasificación de las acciones que realizan las personas para vigilarlas. Por ello, se va a considerar que el sistema tratará de clasificar vídeos de cocina (por ser el *dataset* que se ha utilizado en este trabajo) para una escuela de cocina online.

7.1.1. Impacto Social

Tal vez este aspecto sea el más variable dependiendo de los vídeos para los que se quiera diseñar el sistema. En el caso de la escuela de cocina, puede favorecer hábitos de vida saludables en hogares que han optado por platos preparados en vez de alimentos frescos en los últimos años por comodidad. Podría reducir la obesidad y prevenir enfermedades.

7.1.2. Impacto Económico

El impacto económico se puede ver reflejado sobre todo en la mejora de productividad de los empleados de la escuela de cocina. Si un sistema les etiqueta los vídeos, pueden ahorrar una gran cantidad de recursos en un trabajo tan tedioso y dedicarlo a mejoras en la docencia de la escuela.

7.1.3. Impacto Medioambiental

El impacto medioambiental negativo se reduce a los materiales utilizados para la construcción del ordenador y la energía que se consume en el entrenamiento y uso de la red.

En ese sentido, un impacto positivo del proyecto pueden ser las estrategias que se han propuesto para reducir el tiempo de entrenamiento, pues también reduciría la cantidad de energía necesaria para cada entrenamiento. Además, que un sistema etiquete automáticamente los vídeos puede contribuir a que se vea también reducido el tiempo respecto al que se usaría si este trabajo lo realiza una persona.

7.1.4. Responsabilidad ética y profesional

En el diseño de estos sistemas, la mayor cuestión ética es si el conjunto de los datos utilizados son obtenidos legítimamente con la autorización de la persona que los genera, en este caso, la persona que se ve en el vídeo preparando la receta. Aunque es verdad que el tratamiento de los datos durante el entrenamiento es anónimo y es posible que el ingeniero que diseñe el sistema no vea ningún vídeo en todo el proceso de entrenamiento conviene no olvidar que en los últimos tiempos una gran cantidad de empresas se han beneficiado económicamente con la utilización de sistemas similares. Por ello, es importante que el usuario que aparezca en el vídeo sea consciente de la forma en la que se va a tratar el vídeo y obtener su autorización explícita para esta tarea.

En el caso del trabajo, los vídeos se han obtenido de una base de datos Open Source, en la que los participantes de los vídeos eran conocían para qué eran los vídeos.

7.2. Presupuesto económico

Como el trabajo se puede considerar el comienzo del diseño de un sistema que clasifique vídeos, en esta sección se considerará un sistema general de este tipo. De esta forma, se desarrollará un presupuesto considerando que las partes de diseño de un sistema son recogida de datos, entrenamiento de la red y uso.

El único material físico necesario para el desarrollo de un proyecto de este tipo es un ordenador con una GPU con suficiente potencia. Con una GPU Nvidia 2080 Ti de unos 1500 € es suficiente. El resto de componentes del ordenador puede subir el precio total a 3000 €

La recogida de datos es la tarea más laboriosa y la que lleva más tiempo. Sin embargo, es un trabajo que puede realizar gente no especialista al ser un trabajo repetitivo. Se considerará que se subcontrata con Google Services[5]. Según su página web, etiquetar 5 segundos de un vídeo con una etiqueta costaría $0,0172$ \$ para los primeros 250000 segundos y $0,012$ \$ para los siguientes. Para similar al utilizado, de 77 horas (277200 segundos) el precio sería $250000 \times 0,0172 + 27200 \times 0,012 = 4626,4$ \$, unos 4200 €. El trabajo previo de recolección de datos o de contratación de actores se estimará en unos 800 € para un total de 5000 €.

El trabajo de entrenamiento lo puede desarrollar un ingeniero de datos especializado en Aprendizaje Profundo para Visión Artificial. Como los 12 ECTS se corresponden con aproximadamente dos meses de trabajo a tiempo completo. Aunque el trabajo asociado a la escritura de la memoria se puede ahorrar, el diseño del sistema completo, con una red troncal que se adapte a los datos puede ser realizado en 3 meses. Según la página web *dataconomy.com* [6], un ingeniero investigador, perfil necesario, cobra de media 35900 € brutos al año en España, repartidos en 14 pagas son unos 2600 € al mes. Por tanto, el precio total de mano de obra sería 7800 €.

Para su uso, al suponer que el cliente no está familiarizado con la programación, se creará una plataforma web que permita subir los vídeos que se quieren clasificar y se ejecute la red para realizar una predicción. Se estima un coste de unos 500 €

En la tabla 5 se muestran estos costes.

Elemento	Coste
Ordenador	3000 €
Recolección de datos	5000 €
Entrenamiento de la red	7800 €
Aplicación web	500 €
Total	16300 €

Cuadro 5 – Presupuesto para un sistema completo

[5] https://cloud.google.com/ai-platform/data-labeling/pricing
[6] https://dataconomy.com/2019/01/snapshot-data-scientist-salaries-and-jobs-in-europe/

7.3. Código del Preprocesado

7.3.1. Agrupación de clases y separación de clips

```
import skvideo.io
import skimage.transform
import numpy as np
import glob
import os
import csv

def label_num(clip_label):
    if clip_label == 'SIL':
        return 0
    clip_label = clip_label.split('_')[0]
    if clip_label == 'pour':
        return 1
    if clip_label == 'cut':
        return 2
    if clip_label == 'crack':
        return 3
    if clip_label == 'take':
        return 4
    if clip_label == 'put':
        return 5
    if clip_label == 'add':
        return 6
    if clip_label == 'stir':
        return 7
    if clip_label == 'fry':
        return 8
    if clip_label == 'butter':
        return 9
    if clip_label == 'stirfry':
        return 10
    if clip_label == 'peel':
        return 11
    if clip_label == 'squeeze':
        return 12
    if clip_label == 'spoon':
        return 13
    if clip_label == 'smear':
        return 14

def is_stereo(person, list):
    for cam in list:
        if cam == person + '/stereo':
            return True
    return False


dire = '/media/fj-sanguino/Elements/Breakfast/BreakfastII_15fps_qvga_sync/'
people = sorted(glob.glob(dire + '*'))

split = [people[0:12], people[13:25], people[26:38], people[39:52]]

for sp in range(len(split)):
    spli = split[sp]
```

```
    print('----------------------SPLIT ' + str(sp) + '----------------------')
    out_dir = '/media/fj-sanguino/Elements/Breakfast/Splits/split' + str(sp)
    if not os.path.exists(out_dir):
        os.makedirs(out_dir)
    csv_file = os.path.join(out_dir, 'labels_split' + str(sp) + '.csv')
    print(csv_file)
    with open(csv_file, mode='w') as f:
        employee_writer = csv.writer(f)
        employee_writer.writerow(['video_index', 'video_name', 'clip_start', '
clip_end','clip_name', 'label_name', 'label_number'])
    video_index = 0
    for person in spli:
        print('------------' + person[person.rfind('/')+1:] + '------------' )
        cams = sorted(glob.glob(person + '/*'))
        #print(cams)
        if is_stereo(person, cams):
            cams.remove(cams[cams.index(person + '/stereo')])#removes stereo
        #print(cams)
        for cam in cams:
            print('------' + cam[cam.rfind('/', 0, cam.rfind('/')) + 1: ] + '
------')

            labels = sorted(glob.glob(cam + '/*.labels'))
            #print(len(labels))
            videos = [label[:label.rfind('.')]for label in labels]
            #print(len(videos))
            for vid_Idx in range(len(labels)):
                label = labels[vid_Idx]
                video = videos[vid_Idx]
                print('---' + video[video.rfind('/')+1:video.rfind('.')] + '---' )
                f = open(label, "r")
                clips=[]
                for x in f:
                    clips.append(x[:-2].split(' '))
                #print(clips)


                videogen =skvideo.io.vreader(video)
                frames = []
                i = 0
                for frameIdx, frame in enumerate(videogen):
                    init_frame, end_frame = [int(x) for x in clips[i][0].split('-'
)]
                    clip_label = clips[i][1]
                    #print(init_frame, end_frame, frameIdx)
                    frames.append(frame)
                    if frameIdx + 1 >= end_frame:
                        #print(frameIdx)
                        i = i + 1
                        frames = np.asarray(frames)
                        frames = frames.astype(np.uint8)
                        #print(frames.shape)
                        cam_only = video[video.rfind('/', 0, video.rfind('/'))+1:
video.rfind('/')]
                        video_only = video[video.rfind('/')+1:video.rfind('.')]
                        output_file = os.path.join(out_dir, video_only + '_' +
cam_only + '-' + str(i))
                        #print(output_file, label_num(clip_label))
                        skvideo.io.vwrite(output_file + '.avi', frames)
```

```
                        #2print(frames.shape)
                        fields = [video_index, video[video.rfind('/', 0, video.
    rfind('/', 0, video.rfind('/')))+1:],
                                  init_frame, end_frame, output_file[output_file.
    rfind('/', 0, output_file.rfind('/'))+1:] + '.avi', clip_label, label_num(
    clip_label)]
                        video_index = video_index + 1
                        with open(csv_file, 'a') as f:
                            writer = csv.writer(f)
                            writer.writerow(fields)
                        frames = []
                #init_frame, end_frame = [int(x) for x in clips[i][0].split('-')]
                #clip_label = clips[i][1]
                if end_frame - 3 == frameIdx:
                    i = i + 1
                    frames = np.asarray(frames)
                    frames = frames.astype(np.uint8)
                    #print(frames.shape)
                    cam_only = video[video.rfind('/', 0, video.rfind('/')) + 1:
    video.rfind('/')]
                    video_only = video[video.rfind('/') + 1:video.rfind('.')]
                    output_file = os.path.join(out_dir, video_only + '_' +
    cam_only + '-' + str(i))
                    #print(output_file, label_num(clip_label))
                    skvideo.io.vwrite(output_file + '.avi', frames)
                    fields = [video_index, video[video.rfind('/', 0, video.rfind('
    /', 0, video.rfind('/')))+1:], init_frame, frameIdx,
                              output_file[output_file.rfind('/', 0, output_file.
    rfind('/'))+1:] + '.avi', clip_label, label_num(clip_label)]
                    video_index = video_index + 1
                    with open(csv_file, 'a') as f:
                        writer = csv.writer(f)
                        writer.writerow(fields)
                    frames = []
```

7.3.2. RNN

Después de realizar un submuestrado en el que se queda con un cuadro de cada cinco, procesa todos los cuadros un vídeo a través de una red Resnet preentrenada sobre *Imagenet* para obtener el 1000 características intermedias. Lo guarda como archivo *.pt*

```
import numpy as np
import skvideo.io
import skimage.transform
import csv
import collections
import os
import glob
import time

import torchvision.transforms as transforms
import torch
import torch.nn as nn

'''Función para transformar el vídeo a un array de numpy submuestreando con un
    factor de 5'''
def readShortVideo(video, downsample_factor=5, rescale_factor=1):
```

```
16
17      videogen = skvideo.io.vreader(video)
18      frames = []
19      for frameIdx, frame in enumerate(videogen):
20          if frameIdx % downsample_factor == 0:
21              frame = skimage.transform.rescale(frame, rescale_factor, mode='
        constant', preserve_range=True, multichannel=True, anti_aliasing=True).astype(
        np.uint8)
22              frames.append(frame)
23          else:
24              continue
25
26      return np.array(frames).astype(np.uint8)
27
28 '''Definición de la red'''
29 class Stractor_resnet(nn.Module):
30
31      def __init__(self):
32          super(Stractor_resnet, self).__init__()
33          self.resnet18 = torch.hub.load('pytorch/vision:v0.6.0', 'resnet18',
        pretrained=True)
34
35      def forward(self, img):
36
37        x = self.resnet18(img)
38
39        return x
40
41 '''Define una función que transforma los cuadros del vídeo en tensores,
        normalizando sus valores para disminuir el tiempo del entrenamiento'''
42 def transform_vid(vid):
43   MEAN = [0.5, 0.5, 0.5]
44   STD = [0.5, 0.5, 0.5]
45
46   transform = transforms.Compose([
47               transforms.ToPILImage(mode='RGB'),
48               transforms.ToTensor(),  # (H,W,C)->(C,H,W), [0,255]->[0, 1.0] RGB->
        RGB
49               transforms.Normalize(MEAN, STD)
50           ])
51   frames = []
52   for frame in vid:
53     frames.append(transform(frame))
54
55   return torch.stack(frames)
56
57 data_dir = '/content/drive/My Drive/Splits/'
58 out_dir  = '/content/drive/My Drive/Splits_resnet/'
59
60 splits = ['split3']
61
62 model = Stractor_resnet()
63 begin_time = time.time()
64
65 for split in splits:
66   c = 0
67   video_inDir = os.path.join(data_dir, split)
68   video_outDir = os.path.join(out_dir, split)
69   if not(os.path.exists(out_dir)):
```

```
    os.mkdir(out_dir)
  if not(os.path.exists(video_outDir)):
    os.mkdir(video_outDir)
  done_vid = sorted(glob.glob(video_outDir+'/*.pt'))
  model = model.cuda()
  for video_inPath in sorted(glob.glob(video_inDir+'/*.avi')):

    video_outPath = video_inPath[video_inPath.rfind('/')+1:video_inPath.rfind('.')
    ] + '.pt'
    video_outPath = os.path.join(video_outDir, video_outPath)

    if (video_outPath in done_vid) or (video_outPath in too_big):
      c += 1

      print(c, video_outPath, 'continue', time.time()-begin_time)
      continue

    video_np = readShortVideo(video_inPath) #De archivo de vídeo a vector numpy
    video = transform_vid(video_np) #De vector numpy a tensor
    video = video.cuda()
    with torch.no_grad():
      out = model(video) #Procesa el vídeo a través de la Resnet
    out = out.cpu()
    torch.save(out, video_outPath) #Guarda el vídeo
    c += 1
    print(c, video_outPath, out.shape, time.time()-begin_time)
print('finished')
```

7.3.3. CNN

De las características anteriores, coge 5 por vídeo.

```
import glob
import os
import numpy as np
import torch
import time

data_dir = '/content/drive/My Drive/Splits_resnet/'
out_dir  = '/content/drive/My Drive/Splits_resnet_CNN/'

splits = ['split3']

for split in splits:
    c = 0
    video_inDir = os.path.join(data_dir, split)
    video_outDir = os.path.join(out_dir, split)
    if not(os.path.exists(out_dir)):
      os.mkdir(out_dir)
    if not(os.path.exists(video_outDir)):
      os.mkdir(video_outDir)
    done_vid = sorted(glob.glob(video_outDir+'/*.pt'))
    begin_time = time.time()
    for video_inPath in sorted(glob.glob(video_inDir+'/*.pt')):
        video_outPath = video_inPath[video_inPath.rfind('/')+1:]
        video_outPath = os.path.join(video_outDir, video_outPath)
        if (video_outPath in done_vid):
```

```
            c += 1
            print(c, video_outPath, 'continue', time.time()-begin_time)
            continue

        features = torch.load(video_inPath) #Carga el tensor asociado al vídeo
        f1 = 0
        f2 = features.shape[0]-1
        f3 = int((f1 + f2) / 2)
        f4 = int((f1 + f3) / 2)
        f5 = int((f2 + f3) / 2)
        frames = [features[f1], features[f2], features[f3], features[f4], features
    [f5]] #Coge 5 cuadros
        frames = torch.stack(frames)
        torch.save(frames, video_outPath) #guarda el nuevo tensor
        c += 1
        print(c, video_outPath, frames.shape, time.time()-begin_time)
```

7.4. Código del Entrenamiento

7.4.1. Utilidades (utils.py)

Lo comparten tanto el entrenamiento CNN como el RNN, se encargan de leer el archivo csv donde están el nombre de los vídeos y sus etiquetas y de cargar los vídeos a un array numpy.

```
import numpy as np
import skvideo.io
import skimage.transform
import csv
import collections
import os

def readShortVideo(video_path, downsample_factor=5, rescale_factor=1):

    video = os.path.join(video_path)
    videogen = skvideo.io.vreader(video)
    frames = []
    for frameIdx, frame in enumerate(videogen):
        if frameIdx % downsample_factor == 0:
            frame = skimage.transform.rescale(frame, rescale_factor, mode='
    constant', preserve_range=True, multichannel=True, anti_aliasing=True).astype(
    np.uint8)
            frames.append(frame)
        else:
            continue

    return np.array(frames).astype(np.uint8)


def getVideoList(data_path):
    result = {}

    with open (data_path) as f:
        reader = csv.DictReader(f)
        for row in reader:
            for column, value in row.items():
                result.setdefault(column,[]).append(value)

```

```
    od = collections.OrderedDict(sorted(result.items()))
    return od
```

7.4.2. CNN

Arguments

```
class Args:
  random_seed = 999
  gpu = 0

  # Datasets parameters
  data_dir = '/content/drive/My Drive/Splits_resnet_CNN/'
  val_split = 'split2'

  #Training parameter
  epoch = 200
  val_epoch = 1
  train_batch = 32
  lr = 0.0002
  weight_decay = 0.0005
  momentum = 0.9

  #Output
  save_dir = '/content/drive/My Drive/log_linear3'
```

Dataloader

```
import os
import torch
import numpy as np

import torchvision.transforms as transforms
import time

from torch.utils.data import Dataset

MEAN = [0.5, 0.5, 0.5]
STD = [0.5, 0.5, 0.5]


class DATA(Dataset):
  '''Se ejecuta cada vez que se llama a DATA()'''
    def __init__(self, args, split='split0'):

        ''' Carga parámetros basicos del Dataloader'''
        self.split = split
        self.data_dir = args.data_dir
        self.video_dir = os.path.join(self.data_dir, self.split)

    ''' Crea el diccionario self.dic con los datos del .csv '''
        self.label_path = os.path.join(self.video_dir, 'labels_' + split + '_final
    .csv')
        self.dic = getVideoList(self.label_path)

        videos = []
        c = 0
```

```
        begin_time = time.time()
        '''Carga todos los vídeos en la variable vídeos para que no se carguen en cada época'''
        for video_path in self.dic.get('clip_name'):
            c += 1
            features = torch.load(os.path.join(self.data_dir, video_path)).cpu()
            #features = torch.mean(features, 0)
            print('working in ', os.path.join(self.data_dir, video_path), features.shape,  '{}/{}'.format(c, len(self.dic.get('clip_name'))))
            videos.append(features)
        self.dic['video'] = videos

    '''longitud del dataloader'''
    def __len__(self):
        return len(self.dic.get('video_name'))

    '''se ejecuta cuando se quiere coger un elemento del Dataloader'''
    def __getitem__(self, idx):
        video = {}
        for x, y in self.dic.items():
            video[x] = y[idx]

        label = int(video.get('label_number'))

        frames = video.get('video')

        return frames, label
```

Model

```
import torch.nn as nn
import torch.nn.functional as F

'''Modelo con Resnet18'''
class Late_fusion(nn.Module):
    '''Se ejecuta siempre que se dé la instrucción Late_fusion()'''
    def __init__(self):
        super(Late_fusion, self).__init__()

        self.fc_pre = nn.Linear(1000, 1000)


        self.fc1_concat = nn.Linear(5000, 5000)
        self.fc4_concat = nn.Linear(5000, 2500)
        self.fc3_concat = nn.Linear(2500, 1000)
        self.fc2_concat = nn.Linear(1000, 15)

        #self.fc1_concat = nn.Linear(1000, 1000)
        #self.fc2_concat = nn.Linear(1000, 15)

    '''se llama con objeto_de_la_clase_DecoderRNN(sequence, n_frames)'''
    def forward(self, img):
        x = self.fc_pre(img)

        x = x.view(x.shape[0], x.shape[1]*x.shape[2])

        #x = torch.mean(x, 1)

```

```
        x = self.fc1_concat(x)
        x = self.fc4_concat(x)
        x = self.fc3_concat(x)

        out = self.fc2_concat(x)

        return x, out

'''Inspirado en Kaparthy et el.'''
class Late_fusion(nn.Module):

    def __init__(self):
        super(Late_fusion, self).__init__()

        ''' declare layers used in this network'''''

        self.conv1 = nn.Conv2d(3, 96, kernel_size=11, stride=3)#, padding=1)  # 64
    x64 -> 64x64
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(96)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)#, stride=2)  # 64x64 -> 32x32

        self.conv2 = nn.Conv2d(96, 256, kernel_size=5, stride=1)#, padding=1) # 32
    x32 -> 32x32
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(256)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)#, stride=2) # 32x32 -> 16x16

        self.conv3 = nn.Conv2d(256, 384, kernel_size=3, stride=1)#, padding=1) #
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(384, 384, kernel_size=3, stride=1)#, padding=1)
        self.relu4 = nn.ReLU()
        self.conv5 = nn.Conv2d(384, 256, kernel_size=3, stride=1)#, padding=1) #
        self.relu5 = nn.ReLU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)#, stride=2) #


        self.fc1 = nn.Linear(2048, 2048)
        self.fc2 = nn.Linear(2048, 15)
        #self.softmax = nn.Softmax(1)

        '''
        self.resnet18 = torch.hub.load('pytorch/vision:v0.6.0', 'resnet18',
    pretrained=True)
        self.fc1 = nn.Linear(1000, 1000)
        self.fc2 = nn.Linear(1000, 15)
        '''


    def forward(self, img):
        #print('-------------------------------------')
        #img = img.view(1, img.size(0), img.size(1), img.size(1))
        print(img.shape)
        frames_batch=[0, 0]
        b = torch.split(img, 1, dim=1)
        print(b[0].shape)
        frames_batch[0] = b[0].view(b[0].size(0),3,170,170)
        frames_batch[1] = b[1].view(b[1].size(0),3,170,170)

```

```
        features = []
        for frame_batch in frames_batch:
          x = self.conv1(frame_batch)
          print('After nn.Conv2d(in_channel=3, out_channel=96, kernel_size=11,
    stride=3)', x.shape)
          x = self.bn1(x)
          print('After nn.BatchNorm2d(96)', x.shape)
          x = self.maxpool1(x)
          print('After nn.MaxPool2d(kernel_size=2)', x.shape)

          x = self.conv2(x)
          print('After nn.Conv2d(in_channel=48, out_channel=256, kernel_size=3,
    stride=1, padding=1)', x.shape)

          x = self.bn2(x)
          print('After nn.BatchNorm2d(256)', x.shape)
          x = self.maxpool2(x)
          print('After nn.MaxPool2d(kernel_size=2)', x.shape)

          x = self.conv3(x)
          print('After nn.Conv2d(128, 384, kernel_size=3, stride=1)', x.shape)

          x = self.conv4(x)
          print('After nn.Conv2d(384, 384, kernel_size=3, stride=1)', x.shape)

          x = self.conv5(x)
          print('After nn.Conv2d(384, 256, kernel_size=3, stride=1)', x.shape)

          x = self.maxpool3(x)

          print('After nn.MaxPool2d(kernel_size=2)',x.shape)
          x = x.view(x.size(0), -1)
          print('After view', x.shape)
          features.append(x)
          #print('After nn.Linear(1024, 1024)', x.shape)
        concat = torch.cat(features, 1)
        #print('After concat', concat.shape)
        x = self.fc1(concat)
        #print('After fc1: nn.Linear(2048, 2048)', x.shape)
        out = self.fc2(x)
        #print('After fc3: nn.Linear(2048, 15)', x.shape)
        #out = self.softmax(x)
        #print('After softmax', out.shape)




        return x, out
```

Test

```
import torch
from sklearn.metrics import accuracy_score
import numpy as np

'''Calcula la matriz de confusión'''
def matrix(gts, preds):
    mat = [ [ 0 for i in range(16) ] for j in range(16) ]
```

```
    for i in range(16):
      mat[0][i] = i-1
      mat[i][0] = i-1
      mat[0][0] = 0

    for i in range(len(gts)):
        mat[gts[i]+1][preds[i]+1]+=1
    print('\n'.join(['' .join(['{:4}'.format(item) for item in row]) for row in mat
    ]))


'''Función que evalúa la red rnn y printea la matriz de confusión y su precisión
    '''
def evaluate(model, data_loader):
    model.eval()
    preds = []
    gts = []
    with torch.no_grad():  # No hay necesidad de calcular información del
    gradiente durante la validación
        for idx, (video, gt) in enumerate(data_loader):

                video = video.cuda()
                _, pred = model(video)

                _, pred = torch.max(pred, dim=1)

                pred = pred.cpu().numpy().squeeze()
                gt = gt.numpy().squeeze()

                preds.append(pred)
                gts.append(gt)

    gts = np.concatenate(gts)
    preds = np.concatenate(preds)
    print('gts', [list(gts).count(i) for i in range(0,15)], gts)
    print('pred', [list(preds).count(i) for i in range(0,15)], preds)
    matrix(gts, preds)
    return accuracy_score(gts, preds)
```

Train

```
import torch
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import numpy as np
import os
import time

from torch.utils.tensorboard import SummaryWriter

'''guarda el modelo'''
def save_model(model, save_path):
    torch.save(model.state_dict(), save_path)

'''Configuración de Pytorch'''
args = Args() #carga los parámetros

```

```
if not os.path.exists(args.save_dir): #crea el directorio donde guardar los datos
    os.makedirs(args.save_dir)

torch.cuda.set_device(args.gpu) #configura la GPU

np.random.seed(args.random_seed) #configura una semilla aleatoria
torch.manual_seed(args.random_seed)
torch.cuda.manual_seed(args.random_seed)

writer = SummaryWriter(os.path.join(args.save_dir, ?train_info?)) #configura el
    writer de Tensorboard

'''Carga los datos de entrenamiento y validación y los ordena en lotes'''
print('===> prepara los datos ...')

splits = ['split0', 'split1', 'split2']
splits.remove(splits[splits.index(args.val_split)])

train_data = torch.utils.data.ConcatDataset([DATA(args, split=splits[0]),
                                             DATA(args, split=splits[1])])
val_data =  DATA(args, split=args.val_split)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=args.train_batch,
                                           num_workers=4,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(val_data,
                                           batch_size=args.train_batch,
                                           num_workers=4,
                                           shuffle=False)

'''Define el modelo'''
print('===> prepara el modelo ...')
model = Late_fusion() #carga la clase
model = model.cuda() #carga a la GPU el modelo

'''Definición del algoritmo de backpropagation y de la función de coste'''
criterion = nn.CrossEntropyLoss() #define la función error

params = model.parameters()
optimizer = optim.Adam(params, lr=args.lr, weight_decay=args.weight_decay) #define
    el optimizador

iters = 0
accs = []
best_acc = 0
print('===> comienza el entrenamiento...')
begin_time = time.time()
'''Por cada época'''
for epoch in range(1, args.epoch + 1):
    model.train()
    '''Por cada lote de entrenamiento'''
    for idx, (video, label) in enumerate(train_loader): #loads several videos

            train_info = 'Epoch: [{0}][{1}/{2}]'.format(epoch, idx + 1, len(
    train_loader))

            iters += 1
```

```
74
75      '''Forward path'''
76            video = video.cuda()
77            _, output = model(video)
78
79            label = label.cuda()
80
81      '''Aplica la función de coste'''
82            loss = criterion(output, label)
83
84      '''Actualiza los pesos (propagación hacia atrás)'''
85            optimizer.zero_grad()
86            loss.backward()
87            optimizer.step()
88
89      '''Añade la información a tensorboard, a un archivo.txt y la imprime para
    hacer un seguimiento'''
90            train_info += ' loss: {:.4f}'.format(loss.data.cpu().numpy())
91            writer.add_scalar('loss', loss.data.cpu().numpy(), iters)
92
93            train_info += ' time: {}'.format(time.time()-begin_time)
94
95            #print(train_info)
96            f = open(os.path.join(args.save_dir, 'log.txt'), "a+")
97            f.write(train_info + '\n')
98
99  '''Fin de los lotes'''
100   if epoch % args.val_epoch == 0:
101     '''Calcula la precisión del modelo obtenido'''
102       acc = evaluate(model, val_loader)
103
104       '''Añade información'''
105       writer.add_scalar('val_acc', acc, epoch)
106       val_info = 'Epoch: [{}] ACC:{}'.format(epoch, acc)
107       print(val_info)
108       f = open(os.path.join(args.save_dir, 'log.txt'), "a+")
109       f.write(val_info + '\n')
110
111   '''¿Es la mayor precisión?'''
112       if acc > best_acc:
113           save_model(model, os.path.join(args.save_dir, 'model_best.pth.tar'))
114           best_acc = acc
115
116 '''Guarda el modelo de la época'''
117   save_model(model, os.path.join(args.save_dir, 'model_{}.pth.tar'.format(epoch)
    ))
```

7.4.3. RNN

Arguments

Clase que define algunos parámetros del programa.

```
1 class Args:
2   random_seed = 999
3   gpu = 0
4
5   # Datasets parameters
```

```
  data_dir = '/content/drive/My Drive/Splits_resnet/'
  val_split = 'split2'

  #Training parameter
  epoch = 100
  val_epoch = 1
  train_batch = 32
  lr = 0.0001
  weight_decay = 0.0005
  momentum = 0.9

  #Output
  save_dir = '/content/drive/My Drive/log_RNN2'
```

Dataloader

```
import os
import torch
import numpy as np

import torchvision.transforms as transforms
import time

from torch.utils.data import Dataset

MEAN = [0.5, 0.5, 0.5]
STD = [0.5, 0.5, 0.5]


class DATA(Dataset):
  '''Se ejecuta cada vez que se llama a DATA()'''
    def __init__(self, args, split='split0'):

        ''' Carga parámetros basicos del Dataloader'''
        self.split = split
        self.data_dir = args.data_dir
        self.video_dir = os.path.join(self.data_dir, self.split)

        ''' Crea el diccionario self.dic con los datos del .csv '''
        self.label_path = os.path.join(self.video_dir, 'labels_' + split + '_final.csv')
        self.dic = getVideoList(self.label_path)

        videos = []
        c = 0
        begin_time = time.time()
    '''Carga todos los vídeos en la variable vídeos para que no se carguen en cada época'''
        for video_path in self.dic.get('clip_name'):
            c += 1
            features = torch.load(os.path.join(self.data_dir, video_path)).cpu()
            #features = torch.mean(features, 0)
            #if (time.time()-begin_time % 60 < 5):
            print('working in ', os.path.join(self.data_dir, video_path), features.shape,  '{}/{}'.format(c, len(self.dic.get('clip_name'))))
            videos.append(features)
        self.dic['video'] = videos

```

```
  '''longitud del dataloader'''
    def __len__(self):
        return len(self.dic.get('video_name'))

  '''se ejecuta cuando se quiere coger un elemento del Dataloader'''
    def __getitem__(self, idx):
        video = {}
        for x, y in self.dic.items():
            video[x] = y[idx]

        label = int(video.get('label_number'))

        #frames = torch.load(os.path.join(self.data_dir, video.get('clip_name'))).cpu() #descomentar para cargar los datos en cada época
        #print(os.path.join(self.data_dir, video.get('clip_name')))
        frames = video.get('video')

        return frames, label
```

Model

```
import torch.nn as nn
import torch.nn.functional as F

'''Inspirado enm https://github.com/HHTseng/video-classification/blob/master/CRNN/functions.py'''
class DecoderRNN(nn.Module):
  '''Se ejecuta siempre que se dé la instrucción DecoderRNN()'''
    def __init__(self, CNN_embed_dim=1000, h_RNN_layers=1, h_RNN=256, h_FC_dim=128, drop_p=0.3, num_classes=15):
        super(DecoderRNN, self).__init__()

        self.RNN_input_size = CNN_embed_dim
        self.h_RNN_layers = h_RNN_layers
        self.h_RNN = h_RNN
        self.h_FC_dim = h_FC_dim
        self.drop_p = drop_p
        self.num_classes = num_classes

        self.LSTM = nn.LSTM(
            input_size=self.RNN_input_size,
            hidden_size=self.h_RNN,
            num_layers=h_RNN_layers,
            batch_first=True,
        )

        self.fc1 = nn.Linear(self.h_RNN, self.h_FC_dim)
        self.fc2 = nn.Linear(self.h_FC_dim, self.num_classes)

  '''se llama con objeto_de_la_clase_DecoderRNN(sequence, n_frames)'''
    def forward(self, sequence, n_frames):
        packed = torch.nn.utils.rnn.pack_padded_sequence(sequence, n_frames, batch_first=True)
        RNN_out, (h_n, h_c) = self.LSTM(packed, None)

        # FC layers
        #print('after RNN', h_n.shape)
        #print('after RNN h_n[-1]', h_n[-1].shape)
```

```
        x = self.fc1(h_n[-1])
        #print('after fc1', x.shape)
        x = F.relu(x)
        #print('after reLu', x.shape)
        x = F.dropout(x, p=self.drop_p, training=self.training)
        #print('after reLu', x.shape)
        output = self.fc2(x)
        #print('after fc2', output.shape)

        return x, output
```

Test

```
from sklearn.metrics import accuracy_score

'''Necesario para la RNN'''
def batch_padding(batch_fea, batch_cls):
    n_frames = [fea.shape[0] for fea in batch_fea]
    perm_index = np.argsort(n_frames)[::-1]

    # sort by sequence length
    batch_fea_sort = [batch_fea[i] for i in perm_index]
    n_frames = [fea.shape[0] for fea in batch_fea_sort]
    padded_sequence = nn.utils.rnn.pad_sequence(batch_fea_sort, batch_first=True)
    label = torch.tensor(np.array(batch_cls)[perm_index])
    return padded_sequence, label, n_frames

'''Calcula la matriz de confusión'''
def matrix(gts, preds):
    mat = [ [ 0 for i in range(16) ] for j in range(16) ]
    for i in range(16):
      mat[0][i] = i-1
      mat[i][0] = i-1
      mat[0][0] = 0

    for i in range(len(gts)):
        mat[gts[i]+1][preds[i]+1]+=1
    print('\n'.join([''.join(['{:4}'.format(item) for item in row]) for row in mat
    ]))

'''Función que evalúa la red rnn y printea la matriz de confusión y su precisión
    '''
def evaluate(rnn, data_loader, data_dir):
    rnn.eval()
    preds = []
    gts = []
    with torch.no_grad():
        for idx, (features, label) in enumerate(data_loader):

            sequence, label, n_frames = batch_padding(features, label)

            sequence = sequence.cuda()

            _, pred = rnn(sequence, n_frames)

            _, pred = torch.max(pred, dim=1)
            #print(pred.shape)
            pred = pred.cpu().numpy().squeeze()
```

```
            preds.append(pred)
            gts.append(label)

    gts = np.concatenate(gts)
    preds = np.concatenate(preds)

    print('gts', [list(gts).count(i) for i in range(0,15)], gts)
    print('pred', [list(preds).count(i) for i in range(0,15)], preds)
    matrix(gts, preds)
    return accuracy_score(gts, preds)
```

Train

```
import torch
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim

import numpy as np
import os
import time

from torch.utils.tensorboard import SummaryWriter

'''guarda el modelo'''
def save_model(model, save_path):
    torch.save(model.state_dict(), save_path)

'''Necesario para que todos los datos entren en la RNN teniendo distinto tamaño'''
def batch_padding(batch_fea, batch_cls):
    n_frames = [fea.shape[0] for fea in batch_fea]
    perm_index = np.argsort(n_frames)[::-1]

    batch_fea_sort = [batch_fea[i] for i in perm_index]
    n_frames = [fea.shape[0] for fea in batch_fea_sort]
    padded_sequence = nn.utils.rnn.pad_sequence(batch_fea_sort, batch_first=True)
    label = torch.LongTensor(np.array(batch_cls)[perm_index])
    return padded_sequence, label, n_frames


'''Obtenido de https://discuss.pytorch.org/t/how-to-create-a-dataloader-with-
    variable-size-input/8278'''
def my_collate(batch):
    imgs = [item[0] for item in batch]
    targets = [item[1] for item in batch]
    return imgs, targets


'''Configuración de Pytorch'''

args = Args() #carga los parámetros

torch.cuda.set_device(args.gpu) #configura la GPU

np.random.seed(args.random_seed) #configura una semilla aleatoria
torch.manual_seed(args.random_seed)
torch.cuda.manual_seed(args.random_seed)

```

```
writer = SummaryWriter(os.path.join(args.save_dir, 'train_info')) #configura el writer de tensorboard


args.save_dir = '/content/drive/My Drive/log_RNN_new_op{}_lr{}_wei{}'.format('Adam', args.lr, args.weight_decay) #donde guarda los datos dependiendo de los parámetros
if not os.path.exists(args.save_dir):
  os.makedirs(args.save_dir)


'''Carga los datos de entrenamiento y validación y los ordena en lotes'''
print('===> prepara los datos ...')
splits = ['split0', 'split1', 'split2']
splits.remove(splits[splits.index(args.val_split)])

train_data = torch.utils.data.ConcatDataset([DATA(args, split=splits[0]),
                                             DATA(args, split=splits[1])])
val_data =  DATA(args, split=args.val_split)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=args.train_batch,
                                           num_workers=4,
                                           collate_fn=my_collate,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(val_data,
                                           batch_size=args.train_batch,
                                           num_workers=4,
                                           collate_fn=my_collate,
                                           shuffle=False)


'''Define el modelo'''
print('===> prepara el modelo ...')

rnn = DecoderRNN() #carga la clase
rnn = rnn.cuda() #carga a la GPU el modelo


'''Definición del algoritmo de backpropagation y de la función de coste'''
criterion = nn.CrossEntropyLoss() #define la función error

params = rnn.parameters()
optimizer = optim.Adam(params, lr=args.lr, weight_decay=args.weight_decay) #define el optimizador



iters = 0
accs = []
best_acc = 0
print('===> Comienza el entrenamiento ...')
begin_time = time.time()
'''Por cada época'''
for epoch in range(1, args.epoch + 1):
  rnn.train()
  '''Por cada lote de entrenamiento'''
    for idx, (features, label) in enumerate(train_loader): #loads several videos
```

```
100
101
102       train_info = 'Epoch: [{0}][{1}/{2}]'.format(epoch, idx + 1, len(train_loader
    )) #para mostrar información
103
104         iters += 1
105
106     '''Forward path'''
107         sequence, label, n_frames = batch_padding(features, label)
108
109         sequence = sequence.cuda()
110         #print(sequence.shape)
111         _, pred = rnn(sequence, n_frames)
112
113         batch_lb = torch.from_numpy(np.asarray(label)).cuda()
114         #print(i, batch_lb)
115
116     '''Aplica la función de coste'''
117         loss = criterion(pred, batch_lb)
118
119     '''Actualiza los pesos (propagación hacia atrás'''
120         optimizer.zero_grad()
121         loss.backward()
122         optimizer.step()
123
124     '''Añade la información a un tensorboard, a un archivo.txt y la imprime para
    hacer un seguimiento'''
125         train_info += ' loss: {:.4f}'.format(loss.data.cpu().numpy())
126         writer.add_scalar('loss', loss.data.cpu().numpy(), iters)
127
128         train_info += ' time: {}'.format(time.time()-begin_time)
129
130         writer.add_scalar('loss', loss.data.cpu().numpy(), iters)
131
132         train_info += ' lr: {}, weight: {}, type: {}'.format(learning, weight, typ
    )
133
134         print(train_info)
135         f = open(os.path.join(args.save_dir, 'log.txt'), "a+")
136         f.write(train_info + '\n')
137
138     '''Fin de los lotes'''
139         if epoch % args.val_epoch == 0:
140           '''Calcula la precisión del modelo obtenido'''
141             acc = evaluate(rnn, val_loader, args.data_dir)
142
143       '''Añade información'''
144             val_info = 'Epoch: [{}] ACC:{}'.format(epoch, acc)
145             print(val_info)
146             f = open(os.path.join(args.save_dir, 'log.txt'), "a+")
147             f.write(val_info + '\n')
148
149
150
151             '''¿Es la mayor precisión?'''
152             if acc > best_acc:
153                 save_model(rnn, os.path.join(args.save_dir, 'model_best_rnn.pth.
    tar'))
154                 best_acc = acc
```

```
155
156         '''Guarda el modelo de la época'''
157     save_model(rnn, os.path.join(args.save_dir, 'model_{}_rnn.pth.tar'.format(
    epoch)))
```

7.5. Código del Uso

7.5.1. CNN

args

```
class Args:
  random_seed = 999
  gpu = 0

  # Datasets parameters
  data_dir = '/content/drive/My Drive/Splits/'
  val_split = 'split3'

  #Training parameter
  epoch = 200
  val_epoch = 1
  train_batch = 8
  lr = 0.0002
  weight_decay = 0.0005
  momentum = 0.9

  #Output
  save_dir = '/content/drive/My Drive/log_linear3'
```

Dataloader

```
import os
import torch
import numpy as np

import torchvision.transforms as transforms
import time

from torch.utils.data import Dataset

MEAN = [0.5, 0.5, 0.5]
STD = [0.5, 0.5, 0.5]


class DATA(Dataset):
    def __init__(self, args, split='split0'):

        ''' set up basic parameters for dataset '''
        self.split = split
        self.data_dir = args.data_dir
        self.video_dir = os.path.join(self.data_dir, self.split)
        #print(self.video_dir)

        ''' read the data list '''
        self.label_path = os.path.join(self.video_dir, 'labels_' + split + '_final
    .csv')
```

```
25
26          self.dic = getVideoList(self.label_path)
27
28          self.transform = transforms.Compose([
29                transforms.ToPILImage(mode='RGB'),
30                #transforms.Resize(170),
31                #transforms.CenterCrop(170),
32                transforms.ToTensor(),  # (H,W,C)->(C,H,W), [0,255]->[0, 1.0] RGB->
    RGB
33                transforms.Normalize(MEAN, STD)
34          ])
35
36          '''
37          videos = []
38          c = 0
39          begin_time = time.time()
40          for video_path in self.dic.get('clip_name'):
41              c += 1
42              features = torch.load(os.path.join(self.data_dir, video_path)).cpu()
43              #features = torch.mean(features, 0)
44              #if (time.time()-begin_time % 60 < 5):
45              print('working in ', os.path.join(self.data_dir, video_path), features
    .shape,  '{}/{}'.format(c, len(self.dic.get('clip_name'))))
46              videos.append(features)
47          self.dic['video'] = videos
48          '''
49
50      def __len__(self):
51          return len(self.dic.get('video_name'))
52
53      def __getitem__(self, idx):
54          video = {}
55          for x, y in self.dic.items():
56              video[x] = y[idx]
57
58          label = int(video.get('label_number'))
59
60          video_path = os.path.join(self.data_dir, video.get('clip_name'))
61          #print(os.path.join(self.data_dir, video.get('clip_name')))
62          frames = readShortVideo(video_path)
63
64          frames_torch = []
65          for f in frames:
66            frames_torch.append(self.transform(f))
67          frames_torch = torch.stack(frames_torch)
68
69          f1 = 0
70          f2 = frames_torch.shape[0]-1
71          f3 = int((f1 + f2) / 2)
72          f4 = int((f1 + f3) / 2)
73          f5 = int((f2 + f3) / 2)
74          frames_sub = [frames_torch[f1], frames_torch[f2], frames_torch[f3],
    frames_torch[f4], frames_torch[f5]]
75          frames_sub = torch.stack(frames_sub)
76
77          print('working in ', os.path.join(self.data_dir, video.get('clip_name')),
    frames_sub.shape,  '{}/{}'.format(idx, len(self.dic.get('clip_name'))))
78
79
```

```
        return frames_sub, label
```

Model

```
import torch.nn as nn
import torch.nn.functional as F

class Stractor_resnet(nn.Module):

    def __init__(self):
        super(Stractor_resnet, self).__init__()
        self.resnet18 = torch.hub.load('pytorch/vision:v0.6.0', 'resnet18',
    pretrained=True)

    def forward(self, img):

      x = self.resnet18(img)

      return x

class Late_fusion(nn.Module):

    def __init__(self):
        super(Late_fusion, self).__init__()

        self.fc_pre = nn.Linear(1000, 1000)


        self.fc1_concat = nn.Linear(5000, 5000)
        self.fc4_concat = nn.Linear(5000, 2500)
        self.fc3_concat = nn.Linear(2500, 1000)
        self.fc2_concat = nn.Linear(1000, 15)

        #self.fc1_concat = nn.Linear(1000, 1000)
        #self.fc2_concat = nn.Linear(1000, 15)

    def forward(self, img):
        x = self.fc_pre(img)

        x = x.view(x.shape[0], x.shape[1]*x.shape[2])

        #x = torch.mean(x, 1)

        x = self.fc1_concat(x)
        x = self.fc4_concat(x)
        x = self.fc3_concat(x)

        out = self.fc2_concat(x)

        return x, out
```

Test

```
import torch
from sklearn.metrics import accuracy_score
import numpy as np
import time

```

```
def batch_padding(batch_fea, batch_cls):
    n_frames = [fea.shape[0] for fea in batch_fea]
    perm_index = np.argsort(n_frames)[::-1]

    # sort by sequence length
    batch_fea_sort = [batch_fea[i] for i in perm_index]
    #print(len(batch_fea_sort))
    n_frames = [fea.shape[0] for fea in batch_fea_sort]
    padded_sequence = nn.utils.rnn.pad_sequence(batch_fea_sort, batch_first=True)
    label = torch.tensor(np.array(batch_cls)[perm_index])
    return padded_sequence, label, n_frames

def matrix(gts, preds):
    mat = [ [ 0 for i in range(16) ] for j in range(16) ]
    for i in range(16):
      mat[0][i] = i-1
      mat[i][0] = i-1
      mat[0][0] = 0

    for i in range(len(gts)):
        mat[gts[i]+1][preds[i]+1]+=1
    print('\n'.join([''.join(['& {:4}'.format(item) for item in row]) for row in
    mat]))

def evaluate(classi, resnet, data_loader):
    classi.eval()
    resnet.eval()
    preds = []
    gts = []
    probs = []
    feat = []
    begin_time = time.time()
    with torch.no_grad():  # do not need to caculate information for gradient
    during eval
        for idx, (frames, label) in enumerate(data_loader):  # loads several
    videos

            features = []
            for f in frames:
              f = resnet(f.cuda())
              features.append(f)
            features = torch.stack(features)


            fea, prob = classi(features)

            _, pred = torch.max(prob, dim=1)
            #print(pred.shape)
            pred = pred.cpu().numpy().squeeze()
            preds.append(pred)
            gts.append(label)

            prob = prob.cpu().numpy().squeeze()
            for p in prob:
              probs.append(p)

            fea = fea.cpu().numpy().squeeze()
            for f in fea:
                #print(f.shape)
```

```
                feat.append(f)
            #print(len(feat))

    print('time', time.time()-begin_time)
    gts = np.concatenate(gts)
    preds = np.concatenate(preds)
    feat = np.asarray(feat)
    probs = np.asarray(probs)
    #print(fea.shape)
    print('gts', [list(gts).count(i) for i in range(0,15)], gts)
    print('pred', [list(preds).count(i) for i in range(0,15)], preds)
    matrix(gts, preds)
    gts = np.asarray(gts)
    return gts, probs, feat, accuracy_score(gts, preds)
```

Main

```
import glob
import matplotlib.pyplot as plt

def load_model(args, best, epoch):
    model = Late_fusion()
    model = model.cuda()

    model_dir_class = os.path.join(args.save_dir)

    if best:
      model_std = torch.load(os.path.join(model_dir_class, 'model_best.pth.tar'),
    map_location="cuda:"+str(args.gpu))
      model.load_state_dict(model_std)
      model = model.cuda()

    else:

      model_std = torch.load(os.path.join(model_dir_class, 'model_{}.pth.tar'.
    format(epoch)), map_location="cuda:"+str(args.gpu))
      model.load_state_dict(model_std)
      model = model.cuda()

    return model

def my_collate(batch):
    #print(batch[0][0].shape, batch[0][1])
    imgs = [item[0] for item in batch]
    targets = [item[1] for item in batch]
    #print(imgs[0].shape, targers[0])
    return imgs, targets


"Obtenido de https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation
    /7840"
def accuracy(output, target, topk=(1,3)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)

    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
```

```
    correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res


args = Args()
print('=======> Loading Data')
val_data =  DATA(args, split=args.val_split)

val_loader = torch.utils.data.DataLoader(val_data,
                                         batch_size=args.train_batch,
                                         collate_fn=my_collate,
                                         num_workers=1,
                                         shuffle=False)

model = load_model(args, 0, 84)
resnet = Stractor_resnet()
resnet = resnet.cuda()
#print('==========>')
gts, pr, fea, acc = evaluate(model, resnet, val_loader)
print(fea.shape)
print(gts.shape)
np.save(os.path.join(args.save_dir, 'fea.npy'), fea)
np.save(os.path.join(args.save_dir, 'lab.npy'), gts)
print('top3', accuracy(torch.tensor(pr), torch.tensor(gts)))
print(acc)
```

7.5.2. RNN

Args

```
class Args:
  random_seed = 999
  gpu = 0

  # Datasets parameters
  data_dir = '/content/drive/My Drive/Splits/'
  val_split = 'split3'

  #Training parameter
  epoch = 200
  val_epoch = 1
  train_batch = 8
  lr = 0.0001
  weight_decay = 0.0005
  momentum = 0.9

  #Output
  save_dir = '/content/drive/My Drive/log_RNN_new_opAdam_lr2e-05_wei0.001'
```

Dataloader

```
import os
```

```
import torch
import numpy as np

import torchvision.transforms as transforms
import time

from torch.utils.data import Dataset

MEAN = [0.5, 0.5, 0.5]
STD = [0.5, 0.5, 0.5]


class DATA(Dataset):
    def __init__(self, args, split='split0'):

        ''' set up basic parameters for dataset '''
        self.split = split
        self.data_dir = args.data_dir
        self.video_dir = os.path.join(self.data_dir, self.split)
        #print(self.video_dir)

        ''' read the data list '''
        self.label_path = os.path.join(self.video_dir, 'labels_' + split + '_final.csv')

        self.dic = getVideoList(self.label_path)

        self.transform = transforms.Compose([
              transforms.ToPILImage(mode='RGB'),
              #transforms.Resize(170),
              #transforms.CenterCrop(170),
              transforms.ToTensor(),  # (H,W,C)->(C,H,W), [0,255]->[0, 1.0] RGB->RGB
              transforms.Normalize(MEAN, STD)
        ])

        '''
        videos = []
        c = 0
        begin_time = time.time()
        for video_path in self.dic.get('clip_name'):
            c += 1
            features = torch.load(os.path.join(self.data_dir, video_path)).cpu()
            #features = torch.mean(features, 0)
            #if (time.time()-begin_time  % 60 < 5):
            print('working in ', os.path.join(self.data_dir, video_path), features.shape,  '{}/{}'.format(c, len(self.dic.get('clip_name'))))
            videos.append(features)
        self.dic['video'] = videos
        '''

    def __len__(self):
        return len(self.dic.get('video_name'))

    def __getitem__(self, idx):
        video = {}
        for x, y in self.dic.items():
            video[x] = y[idx]

```

```
        label = int(video.get('label_number'))

        video_path = os.path.join(self.data_dir, video.get('clip_name'))
        #print(os.path.join(self.data_dir, video.get('clip_name')))
        frames = readShortVideo(video_path)

        frames_torch = []
        for f in frames:
          frames_torch.append(self.transform(f))
        frames_torch = torch.stack(frames_torch)

        print('working in ', os.path.join(self.data_dir, video.get('clip_name')),
    frames_torch.shape,  '{}/{}'.format(idx, len(self.dic.get('clip_name'))))


        return frames_torch, label
```

Model

```
import torch.nn as nn
import torch.nn.functional as F

class Stractor_resnet(nn.Module):

    def __init__(self):
        super(Stractor_resnet, self).__init__()
        self.resnet18 = torch.hub.load('pytorch/vision:v0.6.0', 'resnet18',
    pretrained=True)

    def forward(self, img):

      x = self.resnet18(img)

      return x

class DecoderRNN(nn.Module):
    def __init__(self, CNN_embed_dim=1000, h_RNN_layers=1, h_RNN=256, h_FC_dim
    =128, drop_p=0.3, num_classes=15):
        super(DecoderRNN, self).__init__()

        self.RNN_input_size = CNN_embed_dim
        self.h_RNN_layers = h_RNN_layers  # RNN hidden layers
        self.h_RNN = h_RNN  # RNN hidden nodes
        self.h_FC_dim = h_FC_dim
        self.drop_p = drop_p
        self.num_classes = num_classes

        self.LSTM = nn.LSTM(
            input_size=self.RNN_input_size,
            hidden_size=self.h_RNN,
            num_layers=h_RNN_layers,
            batch_first=True,  # input & output will has batch size as 1s
    dimension. e.g. (batch, time_step, input_size)
        )

        self.fc1 = nn.Linear(self.h_RNN, self.h_FC_dim)
        self.fc2 = nn.Linear(self.h_FC_dim, self.num_classes)

```

```
    def forward(self, sequence, n_frames):
        packed = torch.nn.utils.rnn.pack_padded_sequence(sequence, n_frames,
    batch_first=True)
        RNN_out, (h_n, h_c) = self.LSTM(packed, None)
        """ h_n shape (n_layers, batch, hidden_size), h_c shape (n_layers, batch,
    hidden_size) """
        """ None represents zero initial hidden state. RNN_out has shape=(batch,
    time_step, output_size) """
        #print(RNN_out[-1].shape)
        # FC layers
        #print('after RNN', h_n.shape)
        #print('after RNN h_n[-1]', h_n[-1].shape)
        x = self.fc1(h_n[-1])  # choose RNN_out at the last time step
        #print('after fc1', x.shape)
        x = F.relu(x)
        #print('after reLu', x.shape)
        x = F.dropout(x, p=self.drop_p, training=self.training)
        #print('after reLu', x.shape)
        output = self.fc2(x)
        #print('after fc2', output.shape)

        return x, output
```

Test

```
import torch
from sklearn.metrics import accuracy_score
import numpy as np
import time

def batch_padding(batch_fea, batch_cls):
    n_frames = [fea.shape[0] for fea in batch_fea]
    perm_index = np.argsort(n_frames)[::-1]

    # sort by sequence length
    batch_fea_sort = [batch_fea[i] for i in perm_index]
    #print(len(batch_fea_sort))
    n_frames = [fea.shape[0] for fea in batch_fea_sort]
    padded_sequence = nn.utils.rnn.pad_sequence(batch_fea_sort, batch_first=True)
    label = torch.tensor(np.array(batch_cls)[perm_index])
    return padded_sequence, label, n_frames

def matrix(gts, preds):
    mat = [ [ 0 for i in range(16) ] for j in range(16) ]
    for i in range(16):
      mat[0][i] = i-1
      mat[i][0] = i-1
      mat[0][0] = 0

    for i in range(len(gts)):
        mat[gts[i]+1][preds[i]+1]+=1
    print('\n'.join(['& {:4}'.format(item) for item in row]) for row in
    mat]))

def evaluate(rnn, resnet, data_loader):
    rnn.eval()
    resnet.eval()
    preds = []
```

```
    gts = []
    probs = []
    feat = []
    begin_time = time.time()
    with torch.no_grad():  # do not need to caculate information for gradient during eval
        for idx, (frames, label) in enumerate(data_loader):  # loads several videos

            features = []
            for f in frames:
              f = f.cuda()
              f = resnet(f)
              features.append(f)
            #features = torch.stack(features)

            sequence, label, n_frames = batch_padding(features, label)

            sequence = sequence.cuda()

            fea, prob = rnn(sequence, n_frames)

            _, pred = torch.max(prob, dim=1)
            #print(pred.shape)
            pred = pred.cpu().numpy().squeeze()
            preds.append(pred)
            gts.append(label)

            prob = prob.cpu().numpy().squeeze()
            for p in prob:
              probs.append(p)

            fea = fea.cpu().numpy().squeeze()
            for f in fea:
                #print(f.shape)
                feat.append(f)
            #print(len(feat))

    print('time', time.time()-begin_time)
    gts = np.concatenate(gts)
    preds = np.concatenate(preds)
    feat = np.asarray(feat)
    probs = np.asarray(probs)
    #print(fea.shape)
    print('gts', [list(gts).count(i) for i in range(0,15)], gts)
    print('pred', [list(preds).count(i) for i in range(0,15)], preds)
    matrix(gts, preds)
    gts = np.asarray(gts)
    return gts, probs, feat, accuracy_score(gts, preds)
```

Main

```
import glob
import matplotlib.pyplot as plt

def load_model(args, best, epoch):
    model = DecoderRNN()
    model = model.cuda()
```

```
7
8       model_dir_class = os.path.join(args.save_dir)
9
10      if best:
11        model_std = torch.load(os.path.join(model_dir_class, 'model_best_rnn.pth.tar
        '), map_location="cuda:"+str(args.gpu))
12        model.load_state_dict(model_std)
13        model = model.cuda()
14
15      else:
16
17        model_std = torch.load(os.path.join(model_dir_class, 'model_{}_rnn.pth.tar'.
        format(epoch)), map_location="cuda:"+str(args.gpu))
18        model.load_state_dict(model_std)
19        model = model.cuda()
20
21      return model
22
23  def my_collate(batch):
24      #print(batch[0][0].shape, batch[0][1])
25      imgs = [item[0] for item in batch]
26      targets = [item[1] for item in batch]
27      #print(imgs[0].shape, targers[0])
28      return imgs, targets
29
30
31  "Obtenido de https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation
        /7840"
32  def accuracy(output, target, topk=(1,3)):
33      """Computes the precision@k for the specified values of k"""
34      maxk = max(topk)
35      batch_size = target.size(0)
36
37      _, pred = output.topk(maxk, 1, True, True)
38      pred = pred.t()
39      correct = pred.eq(target.view(1, -1).expand_as(pred))
40
41      res = []
42      for k in topk:
43          correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
44          res.append(correct_k.mul_(100.0 / batch_size))
45      return res
46
47
48  args = Args()
49  print('=======> Loading Data')
50  val_data =  DATA(args, split=args.val_split)
51
52  val_loader = torch.utils.data.DataLoader(val_data,
53                                           batch_size=args.train_batch,
54                                           collate_fn=my_collate,
55                                           num_workers=1,
56                                           shuffle=False)
57
58  model = load_model(args, 0, 84)
59  resnet = Stractor_resnet()
60  resnet = resnet.cuda()
61  #print('==========>')
62  gts, pr, fea, acc = evaluate(model, resnet, val_loader)
```

```
print(fea.shape)
print(gts.shape)
np.save(os.path.join(args.save_dir, 'fea.npy'), fea)
np.save(os.path.join(args.save_dir, 'lab.npy'), gts)
print(accuracy(torch.tensor(pr), torch.tensor(gts)))
print(acc)
```

7.6. Código para la Representation

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

fea = np.load('/content/drive/My Drive/log_RNN_new_opAdam_lr2e-05_wei0.001/fea.npy', allow_pickle=True)
cls = np.load('/content/drive/My Drive/log_RNN_new_opAdam_lr2e-05_wei0.001/lab.npy', allow_pickle=True)
fea_test = []
for i in range(0, 10):
    fea_test.append(fea[i])

print("====================> Calculating TSNE")
print(fea.shape)
fea_embedded = TSNE(n_components=2).fit_transform(fea)
#print(fea)
#print(cls)
print("====================> Calculating the colors")
clr = []
for i in range(0, len(cls)):
    if (cls[i] == 0): clr.append('r')
    if (cls[i] == 1): clr.append('b')
    if (cls[i] == 2): clr.append('g')
    if (cls[i] == 3): clr.append('c')
    if (cls[i] == 4): clr.append('m')
    if (cls[i] == 5): clr.append('y')
    if (cls[i] == 6): clr.append('k')
    if (cls[i] == 7): clr.append('pink')
    if (cls[i] == 8): clr.append('orange')
    if (cls[i] == 9): clr.append('slategray')
    if (cls[i] == 10): clr.append('lawngreen')
    if (cls[i] == 11): clr.append('navy')
    if (cls[i] == 12): clr.append('deepskyblue')
    if (cls[i] == 13): clr.append('crimson')
    if (cls[i] == 14): clr.append('lightcoral')
    if (cls[i] == 15): clr.append('olive')
    else: clr.append('w')


print("====================> Painting data")

for i in range(0, len(fea_embedded)):
    plt.plot(fea_embedded[i][0], fea_embedded[i][1], clr[i], marker='o')

plt.savefig('/content/drive/My Drive/log_RNN_new_opAdam_lr2e-05_wei0.001/class.png')
```