



Universidad de Castilla-La Mancha
Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado
Grado en Ingeniería Informática
Computación

Reconocimiento de emociones mediante técnicas de Deep Learning

Guillermo López Bermejo

Diciembre, 2022



TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Computación

Reconocimiento de emociones mediante técnicas de Deep Learning

Autor: Guillermo López Bermejo

Tutor: Roberto Sánchez Reolid

Co-Tutor: María Teresa López Bonal

Diciembre, 2022

*A mi familia,
por darme todas las facilidades
para afrontar esta etapa académica.*

Declaración de autoría

Yo, Guillermo López Bermejo, con DNI 49217699E, declaro que soy el único autor del trabajo fin de grado titulado “Reconocimiento de emociones mediante técnicas de Deep Learning”, que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual, y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 2 de diciembre de 2022

Fdo.: Guillermo López Bermejo

Resumen

El objetivo de este TFG es realizar un estudio de cómo se podría aplicar Deep Learning para la detección de emociones en rostros humanos.

Para ello, en primer lugar se evaluará el desempeño de distintas arquitecturas de redes neuronales convolucionales ya existentes para nuestro problema en concreto para posteriormente optimizarlas, y a partir de las conclusiones obtenidas diseñar una arquitectura propia más adaptada al problema.

Para el desarrollo del proyecto, como ampliaremos más adelante, utilizaremos la base de datos para reconocimiento de expresiones faciales FER2013, cuyo estado del arte se sitúa entorno al 76.82% de accuracy utilizando un ensemble de varias redes neuronales.

En este proyecto se propone como modelo final una optimización de la arquitectura DenseNet201 que obtiene una accuracy de 69.35%. Además, como veremos durante el desarrollo del proyecto, exploraremos la optimización de nuestro modelo para la mejora de otras métricas más adecuadas para el problema que la propia accuracy.

Agradecimientos

Quería agradecer este trabajo a todos los profesores que han formado parte de mi trayectoria académica durante la carrera, pues sin sus conocimientos el desarrollo de este proyecto no habría sido posible, y en especial a mis tutores Maite y Roberto por haber sabido guiarme durante el desarrollo de trabajo. También quería agradecer a mi familia y amigos, pues su apoyo moral ha sido clave a la hora de mantenerme centrado y motivado hasta finalizar el trabajo.

Índice general

1	Introducción	1
1.1	Motivación y objetivos del proyecto	1
1.2	Organización del proyecto	1
1.3	Herramientas	2
2	Conceptos	5
2.1	Red neuronal	5
2.1.1	<i>Neurona</i>	5
2.1.2	<i>Topología de red neuronal</i>	8
2.1.3	<i>Aprendizaje de la red</i>	9
2.2	Red neuronal convolucional	13
2.3	FER2013	18
2.4	Métricas utilizadas	19
2.4.1	<i>Accuracy</i>	20
2.4.2	<i>Métricas para datos desbalanceados</i>	20
3	Reconocimiento de emociones	25
3.1	Metodología	26
3.2	Arquitecturas existentes	29
3.2.1	<i>Definiciones</i>	31
3.2.2	<i>Búsqueda</i>	42
3.2.3	<i>Comparativa</i>	70
3.3	Arquitectura Propia	74
3.3.1	<i>Modificación de VGG16</i>	74
3.3.2	<i>Optimización de DenseNet201</i>	77

4 Conclusiones y trabajos futuros.....	85
A Anexo 1: Algoritmos de búsqueda	87
A.1 Optimización bayesiana.....	87
B Anexo 2: Diagramas adicionales de bloques	89
B.1 Diagramas bloques ResNet	89
B.2 Diagramas bloques Inception	91
C Anexo 3: Capas adicionales	97
C.1 Batch normalization.....	97
C.2 Dropout	98
D Anexo 4: Optimizadores	101
D.1 SGD.....	101
D.2 Adam	101
E Anexo 5: Conceptos adicionales.....	103
E.1 Desvanecimiento del gradiente	103
Referencia bibliográfica.....	107

Índice de figuras

2.1	Neurona artificial	6
2.2	Distintas distribuciones en problemas de clasificación	6
2.3	Funciones de activación	7
2.4	Perceptrón multicapa	8
2.5	Función de pérdida	9
2.6	Descenso del gradiente	11
2.7	Proceso de backtracking	12
2.8	Partes red neuronal convolucional	14
2.9	Proceso de filtrado	15
2.10	Proceso de detección	15
2.11	Proceso de condensación	16
2.12	Niveles de mapas de características	16
2.13	Distribución de clases fer2013	18
3.1	Validación holdout	27
3.2	Validación cruzada	28
3.3	Aumento de datos	30
3.4	Arquitecturas VGG	32
3.5	Bloque residual	33
3.6	Error según capas (extraído de [Kaiming He, 2015])	34
3.7	Arquitecturas ResNet	35
3.8	Bloque DenseNet	36
3.9	Bloque convolucional	36
3.10	Bloque de transición	37
3.11	Arquitecturas DenseNet	38
3.12	InceptionV1	39
3.13	Arquitecturas InceptionV3 e Inception-ResNet	41
3.14	VGG matriz de confusión	47
3.15	VGG curvas ROC	48
3.16	ResNet matriz de confusión	54

3.17	ResNet curvas ROC	55
3.18	DenseNet matriz de confusión	61
3.19	DenseNet curvas ROC	62
3.20	Inception matriz de confusión	68
3.21	Inception curvas ROC	69
3.22	Curvas F1-score VGG	71
3.23	Curvas F1-score ResNet	71
3.24	Curvas F1-score DenseNet	72
3.25	Curvas F1-score Inception	72
3.26	Arquitectura VGG16 con capas residuales	75
3.27	Tipos de redimensionamiento	79
3.28	DenseNet optimizado matriz de confusión	82
3.29	DenseNet optimizado curvas ROC	83
B.1	Bloques ResNetV1 y ResNetV2	89
B.2	Bloques ResNet	90
B.3	Bloques InceptionV2	91
B.4	InceptionV4 bloque inicial	92
B.5	Bloques InceptionV4	93
B.6	Bloques de reducción InceptionV4	94
B.7	Bloques Inception-ResNet	95
B.8	Bloques de reducción Inception-ResNet	96
C.1	Dropout	98
C.2	Fases de dropout	99

Índice de tablas

2.1 Rangos AUC	22
3.1 Mejores modelos VGG	42
3.2 Primer modelo VGG	43
3.3 Validación cruzada primer modelo VGG	43
3.4 Segundo modelo VGG	44
3.5 Validación cruzada segundo modelo VGG	44
3.6 Tercer modelo VGG	45
3.7 Validación cruzada tercer modelo VGG	45
3.8 Mejor modelo VGG	46
3.9 Iteraciones modelo final VGG	46
3.10 Mejores modelos ResNet	49
3.11 Primer modelo ResNet	50
3.12 Validación cruzada primer modelo ResNet	50
3.13 Tercer modelo VGG	51
3.14 Validación cruzada segundo modelo ResNet	51
3.15 Tercer modelo VGG	52
3.16 Validación cruzada tercer modelo ResNet	52
3.17 Mejor modelo ResNet	53
3.18 Iteraciones modelo final Resnet	53
3.19 Mejores modelos DenseNet	56
3.20 Primer modelo DenseNet	57
3.21 Validación cruzada primer modelo DenseNet	57
3.22 Segundo modelo DenseNet	58
3.23 Validación cruzada segundo modelo DenseNet	58
3.24 Tercer modelo DenseNet	59
3.25 Validación cruzada tercer modelo DenseNet	59
3.26 Mejor modelo DenseNet	60
3.27 Iteraciones modelo final DenseNet	60
3.28 Mejores modelos Inception	63

3.29 Primer modelo Inception	64
3.30 Validación cruzada primer modelo Inception	64
3.31 Segundo modelo Inception	65
3.32 Validación cruzada segundo modelo Inception	65
3.33 Tercer modelo Inception	66
3.34 Validación cruzada tercer modelo InceptionV3	66
3.35 Mejor modelo Inception	67
3.36 Iteraciones modelo final Inception	67
3.37 Mejores modelos finales	70
3.38 Mejores modelos VGG con bloques residuales	76
3.39 Mejores modelos finales	79
3.40 Validación cruzada DenseNet optimizado	80
3.41 Iteraciones modelo final DenseNet optimizado	81

1. Introducción

En esta primera sección introductoria, se expondrán tanto la motivación y los objetivos de cara al desarrollo al proyecto como la organización y estructura del mismo, así como una breve explicación de las herramientas utilizadas.

1.1. Motivación y objetivos del proyecto

En los últimos años el deep learning ha irrumpido con fuerza en el campo del machine learning, esto es debido a su uso en tareas tan complicadas como la visión artificial.

En este TFG se abordará el uso de estos algoritmos para abordar el problema concreto del reconocimiento de emociones en personas utilizando imágenes, el objetivo es crear y entrenar una red que recibirá imágenes de personas expresando distintas emociones, y que debe ser capaz de clasificar esas imágenes atendiendo a la emoción expresada.

Dado que dichas emociones resultan de una respuesta biológica a un sentimiento, este tipo de sistemas podrían aplicarse para conocer, entre otras cosas, el grado de satisfacción de una determinada audiencia objetivo para un determinado producto, o bien podría resultar útil para ciertos estudios psicológicos durante una entrevista de trabajo.

El objetivo de este proyecto será, en primer lugar, realizar un estudio del desempeño de ciertas arquitecturas de redes neuronales existentes para este problema concreto. Tras este estudio, se utilizarán las conclusiones obtenidas para el diseño de un modelo propio u optimización de arquitectura existente que obtenga los mejores resultados posibles.

1.2. Organización del proyecto

Podemos dividir el proyecto en las siguientes partes:

1. **Introducción:** En esta sección se introducirán tanto la motivación y objetivos del presente proyecto como las herramientas utilizadas para la realización del mismo.

-
2. **Conceptos:** En esta sección se introducirán tanto los principios en los que se basa el funcionamiento de las redes neuronales artificiales (desde la simple neurona artificial hasta topologías más complejas) como otros conceptos clave en el desarrollo del proyecto, como la base de datos y las métricas utilizadas.
 3. **Reconocimiento de emociones:** En esta sección, tras una breve introducción sobre el concepto de reconocimiento de emociones y a la metodología utilizada, se procederá a evaluar el desempeño de 4 de las arquitecturas de redes neuronales convolucionales más conocidas para el reconocimiento de emociones. Finalmente, utilizando las conclusiones extraídas del estudio, se tratará de obtener la mejor arquitectura posible para esta tarea.
 4. **Conclusiones:** En este último apartado se realizará un breve resumen de lo aprendido durante el desarrollo y las conclusiones finales obtenidas.

1.3. Herramientas

Para el desarrollo del proyecto utilizaremos las siguientes herramientas:

- Para el desarrollo de las redes se utilizará el lenguaje de programación *Python*, éste sin duda se trata de la mejor opción para el desarrollo de redes neuronales, por la gran cantidad de herramientas y librerías con las que cuenta en el ámbito de la ciencia de datos.
- Todo el código se organizará en archivos de extensión tipo *.ipynb*, esto corresponde a un tipo especial de archivos python para el desarrollo de *jupyter notebooks*. Estas notebooks son ampliamente utilizadas en el ámbito de la ciencia de datos, ya que permiten organizar el código en dos tipos de celdas: *celdas ejecutables* y *celdas de texto*. Las celdas ejecutables nos permiten seccionar el código de manera que podemos ejecutar únicamente la parte del código que interesa ejecutar, ya sea para entrenar un modelo, mostrar gráficas, etc. Por otro lado las celdas de texto se trata de celdas no ejecutables, escritas en una variación del formato de marcado *Markdown* que podemos intercalar entre las celdas ejecutables para explicar y comentar los resultados, mostrar las conclusiones obtenidas, etc.
- Las redes neuronales serán desarrolladas utilizando la librería de *Keras*, que se trata de una librería para el desarrollo de modelos de *deep learning* construida sobre la base de la librería *Tensorflow*, utilizada para el desarrollo de todo tipo de modelos de *machine learning*. Estas dos librerías son, junto con *pytorch* en menor medida, las más utilizadas.

La librería *Keras* cuenta con variedad de métodos para el desarrollo de todo tipo de redes neuronales, desde redes neuronales sencillas cuyas capas se definen de forma secuencial, hasta redes más complejas con conexiones no secuenciales entre capas. La librería también cuenta con métodos para ajustar tanto el número de neuronas como el resto de parámetros que definen cada capa.

Como se explicará con más detalle en el apartado 3.2, en nuestro caso nos centraremos en evaluar arquitecturas ya existentes. Lo que nos permitirá, por un lado, utilizar arquitecturas con un buen rendimiento base y, por otro, aprovechar el preentrenamiento con el que ya cuentan dichas redes con otras bases de datos.

En concreto evaluaremos cuatro arquitecturas: VGG [Karen Simonyan, 2015], ResNet [Kaiming He, 2015], DenseNet [Gao Huang, 2016] e Inception [Christian Szegedy, 2015]. Dichas arquitecturas ya preentrenadas pueden invocarse mediante los métodos definidos en Keras, cuya documentación se encuentra en [Chollet, 2014].

2. Conceptos

En esta sección se expondrán varios conceptos comunes a todo el proyecto, desde todos aquellos relacionados con el funcionamiento de las redes neuronales artificiales hasta la base de datos y métricas utilizadas para tanto el entrenamiento como la evaluación de los modelos desarrollados.

2.1. Red neuronal

2.1.1. Neurona

Las redes neuronales son sistemas cuyo funcionamiento se basa en una abstracción de las redes neuronales cerebrales [Education, 2020].

La unidad más fundamental de una red neuronal artificial es la neurona, esta, al igual que una neurona biológica, cuenta con varios canales de entrada y salida, la función de la neurona es recibir las entradas provenientes de las neuronas de la capa anterior o de la propia entrada además de la neurona de sesgo, cada una de esas entradas tendrá asignado un peso y la función de la neurona será realizar la suma del resultado de multiplicar cada una de las entradas por el peso de cada entrada.

En la figura 2.1 podemos ver la representación de una neurona artificial, X_0 y X_1 representan los valores que llegan a la neurona, y W_0 , W_1 y W_2 los pesos asignados a esas entradas, la última neurona (1) representa la neurona de sesgo, que será la encargada de desplazar la función $f(x)$ en el eje horizontal[?].

Este tipo de transformaciones de por si podrían ser útiles para problemas de clasificación que puedan resolverse utilizando una o varias funciones lineales, pero existen muchos tipos de distribuciones distintas para problemas de clasificación, y muchos de ellos requieren de cierta **no linearidad** en las funciones para clasificar correctamente los datos. En la figura 2.2 podemos observar dos distribuciones distintas para un problema de clasificación, la distribución de la izquierda (2.2a) podría resolverse utilizando una red neuronal con función de activación lineal, mientras que para la distribución de la derecha (2.2b) es obligatorio utilizar

algún mecanismo para añadir no linearidad al modelo.

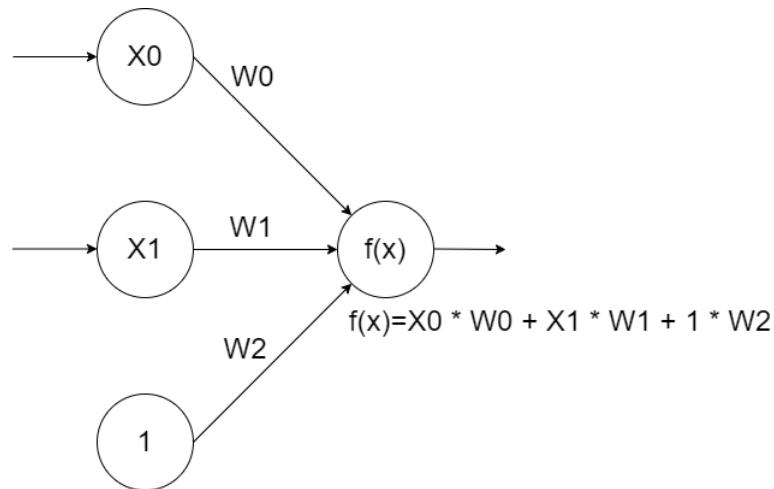


Figura 2.1: Neurona artificial

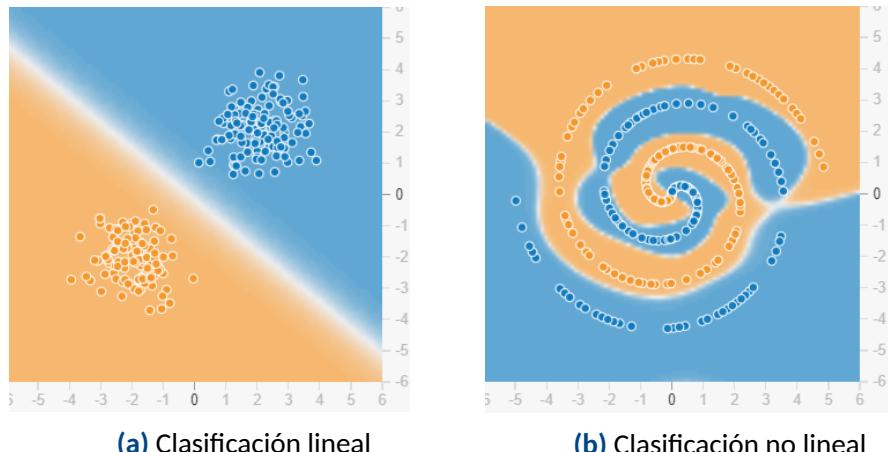


Figura 2.2: Distintas distribuciones en problemas de clasificación

Para añadir no linearidad al modelo se utiliza un tipo especial de funciones llamadas funciones de activación, estas se aplicarán sobre la función f(x) como paso adicional 2.3.

A continuación se definen brevemente algunas de las funciones de activación más importantes.

- Sigmoide:

$$f(x) = \frac{1}{1+e^{-x}} \quad (2.1)$$

Esta función (2.1) devuelve valores en el rango entre 0 y 1, lo que la hace muy adecuada para utilizarla en la última capa, ya que puede representar la probabilidad de pertenencia a cada una de las clases.

Su principal inconveniente son los bajos valores de la derivada en la función en la parte central de la misma, lo que acentúa el problema del desvanecimiento del gradiente (anexo E.1).

- Tanh:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.2)$$

Esta función (2.2) es muy similar a la sigmoide, con la diferencia de que no está acotada entre 0 y 1.

Se suele utilizar en las capas ocultas. Puesto que está centrada en 0, los valores de salida están diferenciados entre negativos, neutros y positivos.

Al igual que la función sigmoide, el valor de la derivada en los valores centrales es muy bajo y acentúa el problema del desvanecimiento del gradiente.

- ReLU:

$$f(x) = \max(0, x) \quad (2.3)$$

En este caso (función 2.3) sólo las neuronas con salida mayor que 0 serán activadas, lo que mejora la eficiencia computacional.

Se utilizan en las capas ocultas ya que su linearidad acelera la convergencia del gradiente descendiente.

El inconveniente de este tipo de función es la posible aparición de neuronas muertas que nunca serán activadas, ya que todos los valores de entrada negativos se transformarán en 0, lo que puede perjudicar la capacidad de aprendizaje del modelo [Baheti, 2022].

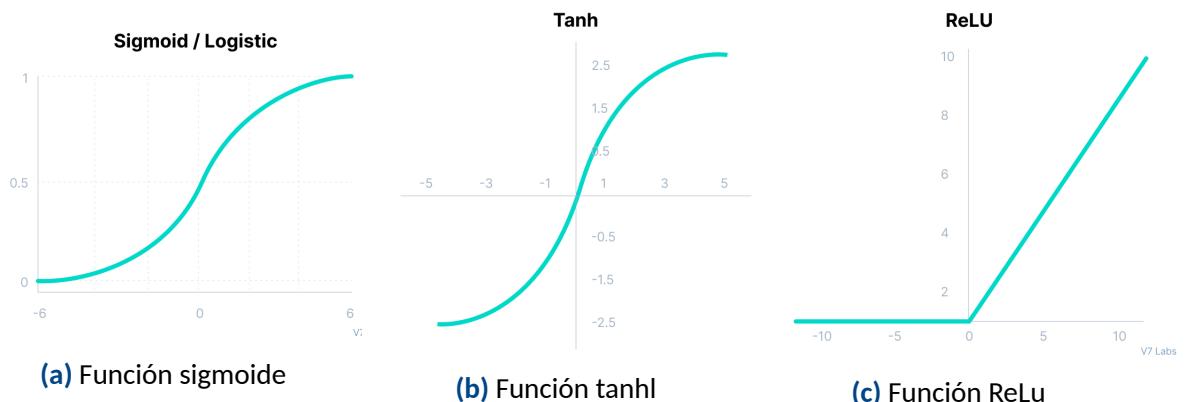


Figura 2.3: Funciones de activación

2.1.2. Topología de red neuronal

Una red neuronal está compuesta por una serie de capas, cada capa está formada por un número de neuronas, existen 3 tipos principales de capas:

- **Capa de entrada:** es la capa de entrada de los datos, y está conectada únicamente con la siguiente capa de la red, cada una de las neuronas representará una variable de entrada en la red
- **Capa oculta:** son las capas que se encuentran entre la capa de entrada y la de salida, existen diversos tipos de capas ocultas, y tanto el número como el número de neuronas de cada una de las capas dependerá de la complejidad y tipo de problema a resolver.
- **Capa de salida:** se trata de la última capa de la red, puede estar formada por una o varias neuronas, y el valor devuelto será considerado el resultado de la red. En problemas de clasificación, cada una de las neuronas de la capa de salida representa a cada una de las clases del problema, y el valor devuelto por cada una de ellas representará la probabilidad de pertenencia del registro caracterizado por las variables de entrada a la clase que representa la neurona, esta probabilidad se puede simular utilizando funciones de activación en la capa de salida que acotan el resultado entre 0 y 1 como es el caso de la función sigmoide.

Tanto el número de capas como de neuronas, y la manera en la que estas se conectan con las neuronas de otras capas definirán la topología de la red. Existe un gran abanico de topologías de red neuronal. De entre ellas, la más utilizada es el perceptrón multicapa (2.4), en la que tendremos una capa de entrada y otra de salida, variando el número de capas ocultas y de neuronas en ellas.

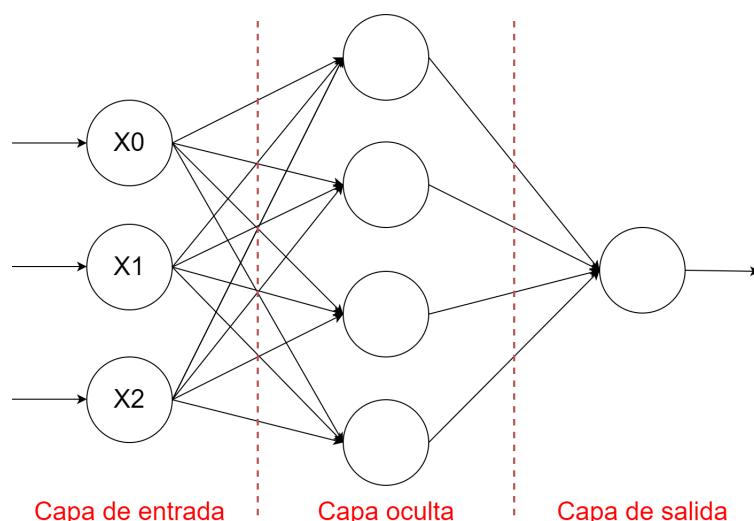


Figura 2.4: Perceptrón multicapa

2.1.3. Aprendizaje de la red

El aprendizaje de la red se produce de la siguiente manera: en primer lugar la información se propaga desde la capa de entrada hasta la de salida, y devuelve un resultado. Este resultado se compara con el resultado real mediante una función de pérdida cuyo resultado representará la disparidad entre el valor obtenido y el esperado.

Una vez calculada la pérdida, se procederá a recorrer la red hacia atrás en un proceso conocido como *backpropagation*, con el objetivo de actualizar los pesos de la red de manera que minimice la pérdida mediante el algoritmo de descenso del gradiente.

Función de pérdida

La función de pérdida es aquella que nos devuelve la disparidad entre el valor obtenido y el valor real, este valor será función de los pesos de la red, por lo que será clave a la hora de calcular el valor de actualización de los pesos de la red. Entre las propiedades que debe cumplir esta función, se encuentra la de ser continuamente derivable, lo que permitirá calcular la derivada de la función con respecto a cada peso a la hora de aplicar el gradiente descendiente 2.5.

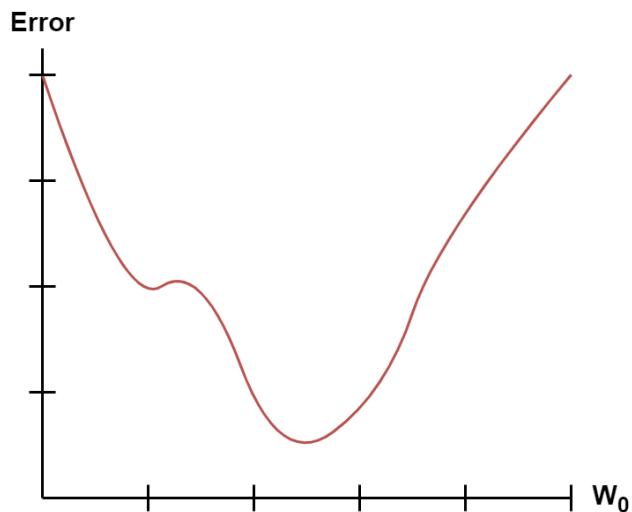


Figura 2.5: Función de pérdida

Existen varios tipos de funciones de pérdida. entre ellas podemos destacar el error cuadrático medio y la entropía cruzada.

- **Error cuadrático medio** [Sammut and Webb, 2010]: es la función de pérdida utilizada en problemas de regresión, y mide el promedio de los errores al cuadrado, es decir, la diferencia entre el valor estimado y el valor real al cuadrado (ecuación 2.4).

$$ECM = \frac{1}{n} \sum_{i=1}^n (\hat{Y}_i - Y_i)^2 \quad (2.4)$$

Siendo $\hat{\mathbf{Y}}$ el vector de n predicciones, e \mathbf{Y} el de verdaderos valores.

- **Entropía cruzada** [ichi.pro, 2021]: la entropía mide el nivel de incertidumbre de todas las posibles variables de resultado. La ecuación utilizada dependerá del valor de \mathbf{X} , de manera que si \mathbf{X} es continua, se utilizará la ecuación 2.5, mientras que si es discreta, se utilizará la definida en 2.6.

$$H(X) = - \int_x p(x) \log(p(x)) \quad (2.5)$$

$$H(X) = - \sum_x p(x) \log(p(x)) \quad (2.6)$$

Siendo las probabilidades predichas un vector de probabilidades de tipo $[0.775, 0.126, 0.039, 0.070]$ y las probabilidades reales un vector de tipo $[1, 0, 0, 0]$, la función de pérdida de entropía cruzada 2.7 compara la probabilidad de pertenencia de cada clase predicha con las probabilidades reales, penalizando las probabilidades en función de qué tan lejos se encuentra el valor esperado del predicho.

$$L = - \sum_{i=1}^n t_i \log(p_i) \quad (2.7)$$

Siendo n el número de clases, y \mathbf{t} y \mathbf{p} los vectores de probabilidad real y predicha, respectivamente.

Descenso del gradiente

Para recorrer la función de pérdida del modelo de manera que se pueda encontrar el mínimo global que minimiza el error, se suele recurrir al algoritmo del descenso del gradiente [Abril, 2021].

La idea tras este algoritmo es hallar la derivada de la función del error en función de cada uno de los pesos sinápticos de la red. Este valor de la derivada es conocido como el gradiente, y su inversa, como se muestra en la figura 2.6, indicará la dirección hacia la que debe desplazarse ese peso en particular para minimizar la función de error. Un gradiente de 0 nos indica que la función se encuentra en un mínimo local.

Por tanto el gradiente representará la dirección que produce mayor incremento en el error desde un determinado punto del espacio, y su negación (ecuación 2.8), la dirección que minimiza el error.

$$\delta = - \frac{\partial E}{\partial W_{i,j}} \quad (2.8)$$

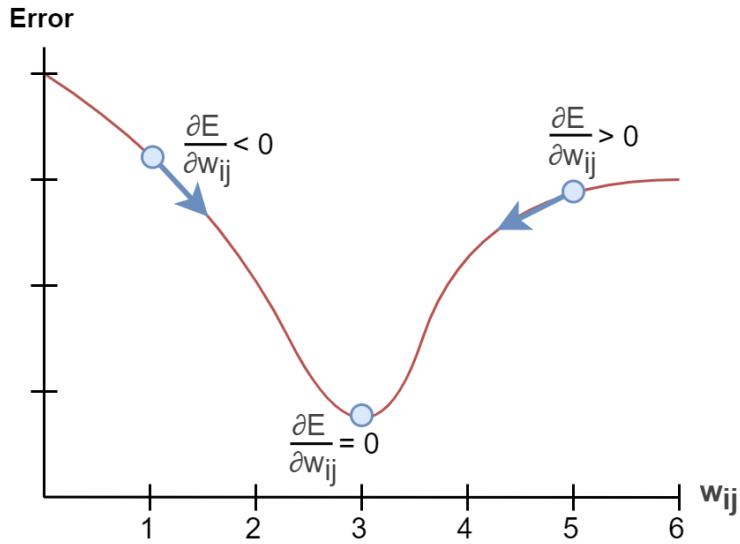


Figura 2.6: Descenso del gradiente

Backpropagation

Una vez definidos los conceptos necesarios, podemos introducir el algoritmo que se encarga de forma directa de la actualización de los pesos de la red neuronal, el algoritmo de *backpropagation*.

El algoritmo funciona de la siguiente manera: en primer lugar se introduce una entrada de datos a la red, esta entrada recorre toda la red hasta llegar a la última capa (capa de salida), la cual devuelve un resultado. Puesto que nos encontramos en fase de entrenamiento, podemos comparar la salida de la red con el valor real, y obtener un error que, como ya comentamos, podría ser el error cuadrático medio en caso de problemas de regresión o la entropía cruzada en el caso de problemas de clasificación.

El proceso de aprendizaje ocurre mediante la actualización de los pesos sinápticos de la red en función del error obtenido tras la comparación, y dicha actualización de los pesos dependerá de la capa en la que nos encontremos.

El proceso se muestra en la figura 2.7 y, como podemos observar, comienza en la neurona de la capa de salida hacia las capas anteriores de la red. Esta primera fase de aprendizaje se muestra en la figura 2.7a, y provocará la actualización de los pesos sinápticos que conectan la capa de salida con la capa inmediatamente anterior. Si tomamos como ejemplo la actualización del peso w_{bc} , el nuevo valor resultará de aplicar la ecuación 2.9.

$$w_{bc} = w_{bc} + \alpha \cdot \delta_c \cdot h_b \quad (2.9)$$

Siendo α el learning rate o tasa de aprendizaje, δ_c el incremento del gradiente en c , y h_b el valor en b .

Para la actualización de los pesos de capas anteriores a la salida, puesto que no podemos calcular el error para obtener el gradiente, calcularemos la influencia que ha tenido cada una de las neuronas de la capa en el error final. Atendiendo a la figura 2.7b, el nuevo peso para w_{ab} se obtendría tras aplicar la ecuación 2.10.

$$\delta_b = \left(\sum_{x \in s(b)} w_{bc} \cdot \delta_c \right) \cdot h_b \cdot (1 - h_b) \quad (2.10)$$

Siendo $s(b)$ los sucesores de b en la red.

De esa forma habríamos recorrido todas la capas de la red como se muestra en la figura 2.7b

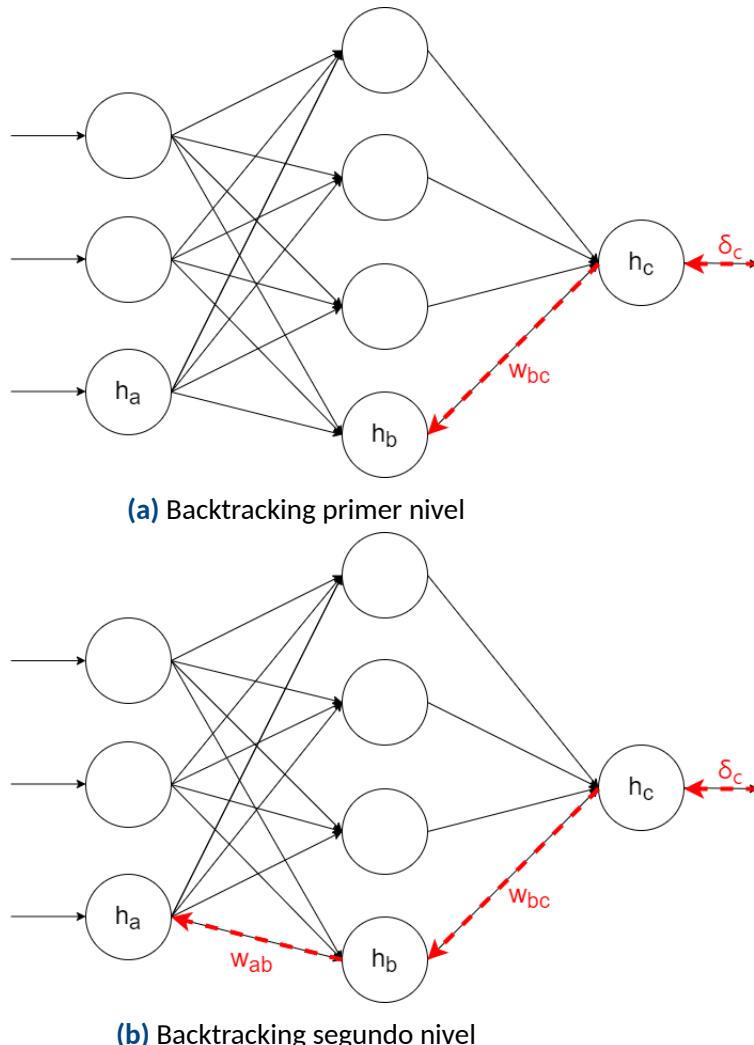


Figura 2.7: Proceso de backtracking

Sesgo y varianza

A la hora de desarrollar cualquier tipo de modelo basado en el **aprendizaje automático**, existen dos conceptos fundamentales a la hora de realizar la selección de modelos: el sesgo y la varianza [Baro, 2009].

El proceso de aprendizaje de un modelo de machine learning consiste en su entrenamiento con parte del conjunto de datos, este conjunto se conoce como conjunto de entrenamiento, y es a partir de este del que el modelo extraerá los patrones necesarios durante el aprendizaje.

Puesto que cada uno de los datos es diferente, un modelo excesivamente complejo que se ajuste al 100% a los datos de entrenamiento podría no ser conveniente si lo que buscamos es un modelo capaz de aprender patrones que permitan clasificar nuevos datos independientes al conjunto de entrenamiento, en este caso diríamos que se trata de un modelo que está **sobreajustando** a los datos de entrenamiento y por lo tanto cuenta con alto grado de **varianza**. Este grado de varianza suele ser proporcional a la complejidad del modelo.

Por otra parte, si lo que tenemos es un modelo que no se ajusta lo suficiente al conjunto de datos de entrenamiento, decimos que este modelo está **generalizando** y por tanto se trata de un modelo con alto grado de **sesgo**. En este caso el grado de sesgo se considera inversamente proporcional a la complejidad del modelo.

Lo ideal por tanto es encontrar un grado de compromiso entre el sesgo y la varianza del modelo, todo en base tanto a la naturaleza del problema a resolver como a la base de datos.

2.2. Red neuronal convolucional

Las redes neuronales artificiales son una adaptación de las redes neuronales para ser usadas en problemas de visión artificial, que es la disciplina que estudia la capacidad de una computadora para analizar y comprender imágenes del mundo real.

Una red neuronal convolucional está compuesta por dos partes principales: el cuerpo y la cabeza.

- **Cuerpo:** Es la parte de la red neuronal que se encarga de extraer los llamados "Mapas de características" de la imagen. El conjunto de todos los mapas de características permiten a la red "Entender" y diferenciar esa imagen de otras pertenecientes a otras clases. Existen muchos tipos de mapas de características que pueden extraerse dependiendo del problema concreto, y pueden ir desde atributos sencillos como líneas hasta colores, texturas y formas más complejas.
- **Cabeza:** Una vez extraída la información necesaria de las imágenes, la cabeza de la red clasifica las imágenes en las diferentes clases.

En la figura 2.8 podemos ver una representación de lo explicado anteriormente, el cuerpo de la red neuronal se encarga de extraer las características de la imagen de entrada, en este caso se trata de un coche del cual se extraen 4 características: ventanillas, chasis, faros y ruedas, para que finalmente la cabeza de la red lo clasifique como un "Volkswagen escarabajo".

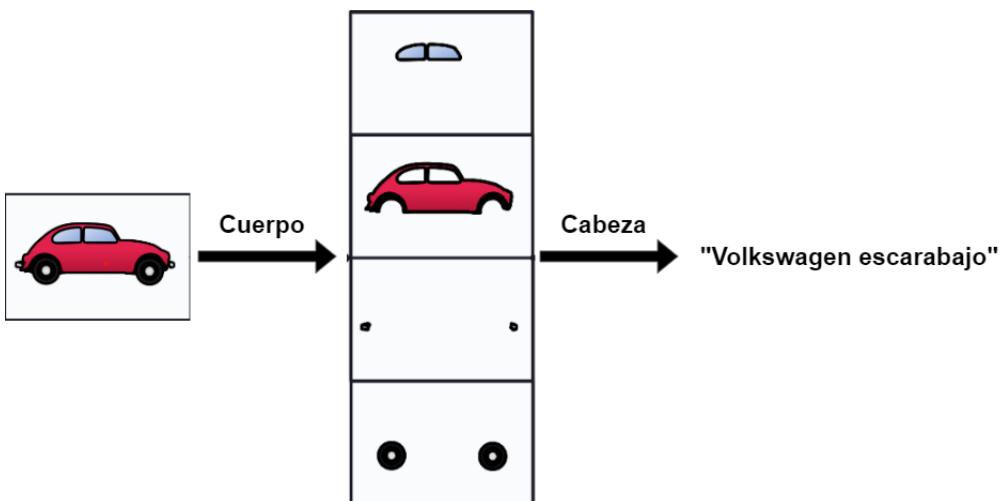


Figura 2.8: Partes red neuronal convolucional

Este tipo de redes funcionan de la siguiente manera:

En primer lugar, representamos la imagen como un conjunto de píxeles, cada uno de los cuales tendrá un valor dependiendo de la intensidad de ese píxel en la imagen. En imágenes con más de un color se emplearán varios canales, cada uno de ellos correspondiente a un color. Por ejemplo, en imágenes en blanco y negro solo será necesario un canal, mientras que en imágenes RGB se utilizarán 3 canales. Cada uno de estos píxeles puede ser una neurona de entrada en nuestra red, puesto que las imágenes son matrices bidimensionales, una red neuronal para imágenes de 48x48 en blanco y negro tendrá 2.304 neuronas de entrada.

La bidimensionalidad de las imágenes de entrada implica que esta información de entrada de los píxeles no aporte la suficiente información sobre la imagen, puesto que la posición de los mismos en la matriz también es relevante. Es por esto que debemos aplicar un tipo especial de transformación denominada **convolución**.

Este proceso de convolución se divide en tres fases:

- **Filtrado:** consiste en aplicar un "filtro" sobre la imagen en cuestión. Un filtro no es mas que una matriz de tamaño menor al de la imagen. Conforme el filtro se desplace sobre la imagen, los valores de entrada se multiplicarán por los del filtro. Existen diversos tipos de filtros encargados de detectar líneas, colores, texturas u otros atributos de la

Imagen, en la figura 2.9 podemos ver cómo aplicamos un filtro (matriz naranja) 2×2 capaz de detectar líneas horizontales sobre una imagen 6×6 , este filtro se desplazaría hasta cubrir toda la imagen. Los valores de estos filtros se irán actualizando conforme avanza el entrenamiento de la red hasta obtener aquellos que generen los mapas de características más adecuados.

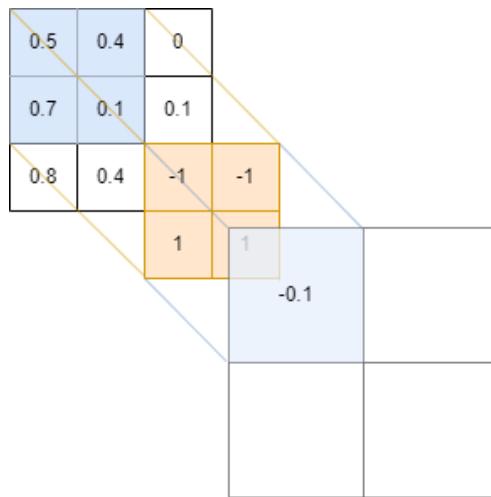


Figura 2.9: Proceso de filtrado

- **Detección:** una vez filtrada la imagen, se le aplica una función de activación tal como la función ReLU (introducida en apartado 2.1.1), de manera que todos los valores poco relevantes sean igualmente insignificantes, intensificando los valores relevantes y obteniendo el mapa de características. Como podemos ver en la figura 2.10, los valores negativos pasan a ser 0 mientras que los positivos se mantienen.

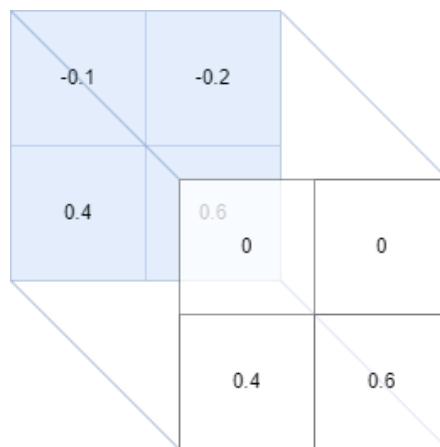


Figura 2.10: Proceso de detección

-
- **Condensación:** existen varios métodos de condensación, el más común es el conocido como "max pooling", que consiste en recorrer la imagen de manera similar a como se hizo durante el filtrado pero en lugar de aplicar un filtro, utilizar el mayor valor de entre los píxeles recorridos, como se muestra en la figura 2.11.

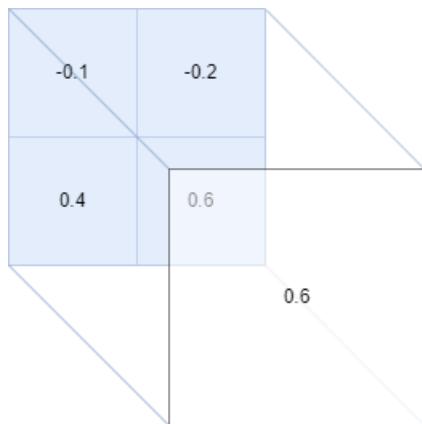


Figura 2.11: Proceso de condensación

Como hemos podido observar, la convolución tiene como efecto secundario la reducción de la resolución de la imagen. Existen distintas técnicas para evitar este fenómeno, tales como la agregación de bordes a la imagen original, todo depende del diseño de la arquitectura.

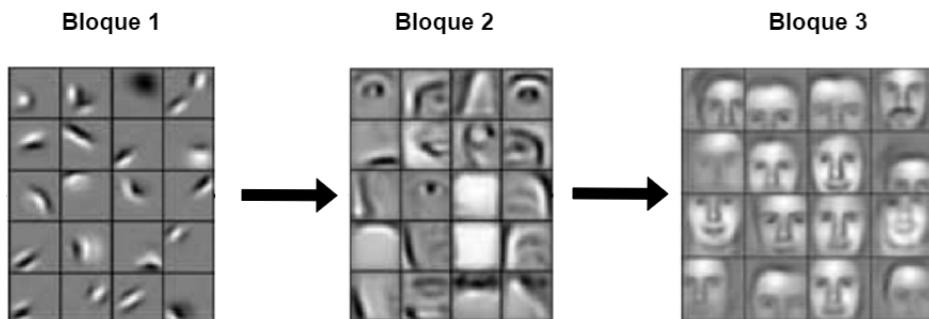


Figura 2.12: Niveles de mapas de características

Usualmente las capas convolucionales se agrupan en bloques. Por cada bloque, se agrupan algunas capas convolucionales que desembocan en una capa de condensación **max pooling** que aplica una reducción del tamaño de la imagen, de manera que en el siguiente bloque se puedan generar mas bloques de características. Además, esto permite que la información se vaya condensando de bloque en bloque, ya que en el caso de aplicar, por

ejemplo, filtros de 2x2, cada pixel que pasa de un bloque a otro contiene la información de los 4 píxeles anteriores, esto permite generar mapas de características con formas cada vez más complejas. Esto se puede observar en la figura 2.12, en la que el primer bloque de una red convolucional entrenada con rostros humanos se especializa en la detección de atributos sencillos como líneas y curvas, los cuales serán utilizados por el segundo y tercer bloque para la detección de atributos más complejos como rasgos faciales y rostros completos. Éste enfoque de "aprendizaje jerárquico" en el la profundidad de la red determina la obtención de representaciones cada vez más significativas es lo que se conoce como "**Deep Learning**" [Recuero, 2018].

2.3. FER2013

Para el desarrollo de este proyecto se utilizará la base de datos para reconocimiento de expresiones faciales **FER2013** [Abril, 2013], la cual cuenta con 35.887 imágenes con una resolución de 48x48 píxeles clasificadas en 7 emociones diferentes: "**Enfado**", "**Disgusto**", "**Miedo**", "**Felicidad**", "**Neutralidad**", "**Tristeza**" y "**Sorpresa**".

La distribución de las clases en la base de datos se muestra en la figura 2.13.

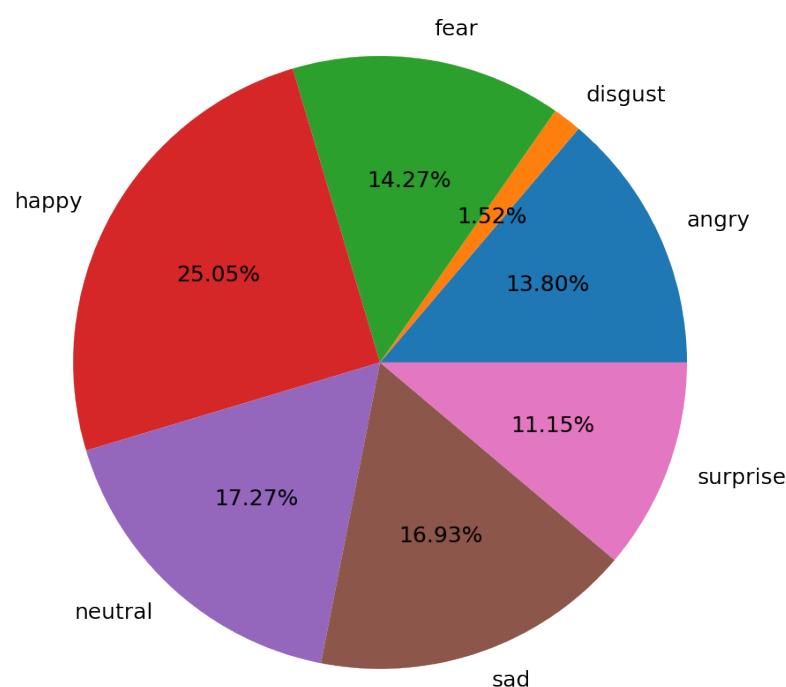


Figura 2.13: Distribución de clases fer2013

Podemos observar que la distribución de las clases no está balanceada, en particular contamos con una gran cantidad de registros para la clase "happy" (25% del total), mientras que la clase "disgust" solo representa un 1.52%, esto implica que la base de datos está desbalanceada.

En cuanto al **estado del arte** de modelos entrenados con este dataset, encontramos una tasa de acierto o accuracy (2.4.1) máxima de 76.82% [Paperswithcode, 2022].

Como veremos, este desbalanceo en la distribución de las clases puede llevar a que ciertas métricas no reflejen de manera adecuada el rendimiento de los modelos, por lo que uno de los objetivos del estudio será la optimización de un modelo que maximice otras métricas más adecuadas para la medida del desempeño en bases de datos desbalanceadas.

2.4. Métricas utilizadas

Nuestro modelo devolverá un array cuya longitud será igual al número de clases (7 en este caso), cada uno de los valores de esa matriz corresponde a la probabilidad de pertenencia de la imagen de entrada a cada una de las 7 clases. Para realizar la predicción final deberemos considerar únicamente el mayor valor del array.

Para la elección de las métricas debemos considerar que se trata de un problema de clasificación multiclase en el que todas las clases tienen la misma importancia.

Para definir las fórmulas utilizaremos las siguientes nomenclaturas [Fawcett, 2005]:

- Verdaderos positivos (**VP**): para una determinada clase, número de registros de la base de datos correspondientes a esa clase clasificados correctamente como esa determinada clase por el modelo.
- Verdaderos negativos (**VN**): para una determinada clase, número de registros de la base de datos no correspondientes a esa clase clasificados correctamente como otra clase (no necesariamente la correcta) por el modelo.
- Falsos positivos (**FP**): para una determinada clase, número de registros de la base de datos no correspondientes a esa clase clasificados incorrectamente como esa determinada clase por el modelo.
- Falsos negativos (**FN**): para una determinada clase, número de registros de la base de datos correspondientes a esa clase clasificados incorrectamente como otra clase por el modelo.
- Positivos totales (**P**): número total de positivos reales para una determinada clase ($VP + FN$).
- Negativos totales (**N**): número total de negativos reales para una determinada clase ($VN + FP$).

A su vez, como se vio en el apartado 2.3, debemos considerar que la base de datos no está totalmente balanceada, por lo que las clases que más aparecen podrían opacar a las que menos aparecen para ciertas métricas.

2.4.1. Accuracy

La accuracy (**ACC**) o tasa de acierto es la métrica que mide el número de veces que el modelo acierta en sus predicciones.

$$ACC = \frac{VP+VN}{P+N} \quad (2.11)$$

Como podemos ver en la fórmula 2.11, el peso que aportan los verdaderos negativos (VN) a la métrica implica que no resulte adecuada para conjuntos de datos desbalanceados, ya que los aciertos en la clase mayoritaria opacarán los fallos de las minoritarias, por lo que a pesar de que puede ser un buen indicador de las virtudes del modelo, deberemos considerar también otras métricas.

2.4.2. Métricas para datos desbalanceados

Como hemos introducido anteriormente, métricas como el accuracy no son buenos indicadores para medir la eficiencia de modelos sobre bases de datos desbalanceadas, ya que por ejemplo, un hipotético clasificador que trabajase sobre una base de datos con 990 muestras de la clase 1 y tan solo 10 de la clase 2, obtendría una tasa de acierto del 99% siendo un mal modelo cuyo único labor consistiría en clasificar todos los registros como pertenecientes a la clase 1 independientemente de los datos de entrada.

Métricas para clasificación multiclasé

Puesto que nuestro clasificador de emociones deberá clasificar los registros de entrada en 1 de entre 7 clases, nos encontramos ante un problema de clasificación multiclasé. Dado que este tipo de métricas se calculan en base a cada una de las clases, es preciso utilizar un método adecuado para promediar los resultados obtenidos y obtener un único valor que represente el valor de la métrica para el modelo. Existen diversos métodos para promediar la métricas de un clasificador multiclasé, siendo los dos más representativos la macromedia y la micromedia.

- **Macromedia:** Consiste en sumar las matrices de confusión obtenidas para cada una de las clases para posteriormente calcular las métricas sobre ésta.
- **Micromedia:** Consiste en obtener las matrices de confusión para cada una de las clases y calcular la media de todas, esto lo hace más adecuado para un conjunto de datos desbalanceado puesto que la puntuación obtenida para la clase con menos registros tendrá la misma relevancia que la obtenida para las clases con más registros.

En nuestro caso, puesto que nos encontramos ante un problema de clasificación multiclasé de 7 clases todas ellas con la misma importancia, optaremos por utilizar la **micromedia**.

Matriz de confusión

La matriz de confusión [Fawcett, 2005] es la base del resto de métricas. Se trata de una matriz bidimensional en la que cada columna representa el número de predicciones correspondiente a cada clase mientras que cada fila representa el valor real .

Siendo i las filas y j las columnas, cada celda C_{ij} representa el número de registros de la base de datos predicha como la clase j que realmente pertenece a la clase i . En nuestro caso, como se muestra en la ecuación 2.12, para que la matriz sea más interpretable, este valor se dividirá entre el total de registros reales que pertenecen a esa clase, de manera que obtendremos un valor entre 0 y 1.

$$C_{ij} = \frac{C_{ij}}{\text{Total}_j} \quad (2.12)$$

Recall

El Recall (R), también conocido como sensitividad o tasa de verdaderos positivos [Fawcett, 2005] , representa la fracción de casos positivos que se ha detectado, es decir de todos los casos que existen para una determinada emoción, cuantos de ellos se han clasificado como tales, tal y como puede verse en la ecuación 2.13.

$$R = \frac{VP}{VP+FN} \quad (2.13)$$

Precisión

La precisión o valor positivo predictivo [Fawcett, 2005] mide la fracción de los casos clasificados como positivos que lo eran realmente, es decir, de todos los casos clasificados como una determinada emoción, que fracción de ellos lo son realmente.

$$R = \frac{VP}{VP+FP} \quad (2.14)$$

F1-score

La F1-score [Taha AA, 2015] se utiliza para medir el balance o media armónica entre el recall y la precisión, tal y como se muestra en la ecuación 2.15.

$$F1 = \frac{2 \cdot R \cdot P}{R+P} \quad (2.15)$$

Curva ROC/AUC

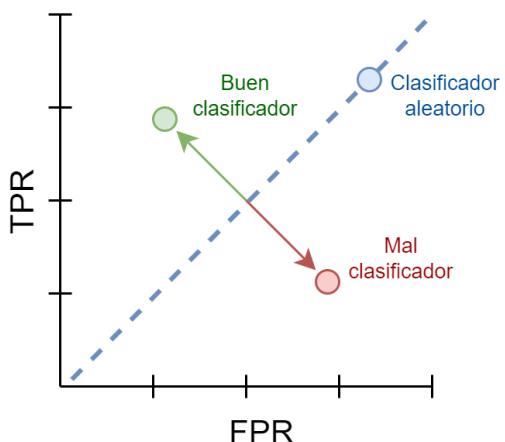
La curva ROC [Fawcett, 2005] se define como la curva que obtenemos si representamos en un eje de coordenadas cartesianas tanto la tasa de verdaderos positivos o sensibilidad en el eje Y como la tasa de falsos positivos (1 - especificidad) en el eje X.

La sensibilidad indica la capacidad de nuestro sistema de clasificar como positivos los casos realmente positivos, mientras que la especificidad indica la capacidad de clasificar

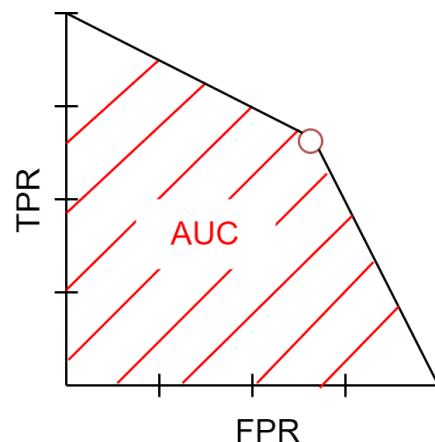
como negativos los casos realmente negativos. Por tanto la curva ROC obtenida (también conocida como espacio ROC), nos indicará el balance entre la tasa de verdaderos positivos (presuntos beneficios), y la tasa de falsos positivos (presuntos costes).

En la figura 2.14a se muestra la gráfica con los ejes cartesianos correspondientes a la curva ROC. Un clasificador perfecto situaría su puntuación ROC en la esquina superior izquierda (coordenadas (0,1)), y se caracterizaría por un 100% de sensibilidad y especificidad, mientras que cualquier clasificador aleatorio se situaría en cualquier punto a lo largo de la diagonal que une el punto (0,0) con el (1,1).

Por tanto los clasificadores que sitúen su puntuación por encima de la diagonal que representa la aleatoriedad se considerarán buenos clasificadores, mientras que los que se sitúen por debajo se considerarán clasificadores pobres (2.14a).



(a) Clasificadores según ROC



(b) AUC

AUC obtenida	Calidad del clasificador
0.5	Sin discriminación
0.5 - 0.7	Discriminación pobre
0.7 - 0.8	Discriminación aceptable
0.8 - 0.9	Discriminación excelente
0.9 - 1	Discriminación sobresaliente

Tabla 2.1: Rangos AUC

A la hora de utilizar esta información para la comparación de modelos, es frecuente utilizar el área bajo la curva (AUC), que como su propio nombre indica, representa la medida

del área que queda bajo la curva generada en el espacio ROC (2.14b).

La interpretación de esta métrica a la hora de evaluar modelos depende del campo de aplicación del modelo, pues no es lo mismo el coste de un error por parte del clasificador en un contexto médico con un modelo utilizado para la predicción de enfermedades que en un contexto de negocios, debiendo obtener el primero una mejor puntuación para considerarse un modelo adecuado. Para nuestro modelo, consideraremos una serie de rangos, mostrados en la tabla 2.1, extraídos de [David W. Hosmer Jr., 2013], estos rangos cubren desde una puntuación mínima de 0.5, siendo esta la que obtendría cualquier modelo que clasificase las entradas de forma aleatoria (línea discontinua azul en figura 2.14a), hasta puntuaciones entre el 0.97 y 1, considerándose estos como clasificadores sobresalientes.

3. Reconocimiento de emociones

Antes de comenzar el estudio, es preciso introducir tanto qué podemos considerar una emoción como los problemas y limitaciones asociados al reconocimiento de las mismas mediante sistemas de inteligencia artificial.

Se considera como una emoción a un proceso mental que refleja la actitud de evaluación de un individuo ante diferentes situaciones [?]. La principal diferencia entre una emoción y un sentimiento, es que mientras que las emociones están más relacionadas con reacciones biológicas ante los estímulos, los sentimientos son percepciones mentales ante esos estímulos, de manera que podemos considerar la alegría, la tristeza, el miedo o la ira como emociones, mientras que el amor, la felicidad o el odio están más relacionados con los sentimientos [Chen, 2019].

El reconocimiento de tanto las emociones como los sentimientos de las personas por parte de sistemas inteligentes ha sido objeto de estudio debido a la gran cantidad de aplicaciones en ámbitos como el marketing puesto que dichas emociones y sentimientos podrían indicar el grado de satisfacción de los clientes ante un determinado producto o servicio, de la misma manera que podrían utilizarse para analizar las aptitudes de una determinada persona en una entrevista de trabajo. Existen diversos patrones que permiten reconocer dichos sentimientos y emociones, ya sea analizando el propio lenguaje de la persona, su voz, sus expresiones faciales...

Para ello, deben realizarse ciertas asunciones y conocer las limitaciones. Entre dichas asunciones se encuentra la de asumir, por un lado, que las emociones sean clasificables y, por otro, que esta clasificación sea universalizable. En relación a la clasificación de emociones, existen distintos modelos para ello, siendo el más extendido el concluido por el estudio liderado por el psicólogo Paul Ekman en el año 1972, que concluye que existen seis emociones básicas que las personas expresan de manera universal: sorpresa, tristeza, miedo, felicidad, enfado y asco [Ekman, 1992]. Este tipo de clasificaciones, a pesar de no resultar una solución definitiva al problema, puesto que no todas las culturas expresan las emociones de la misma manera, sí que pueden dar lugar a aproximaciones que pueden resultar útiles en ciertos ámbitos que no requieren un reconocimiento y clasificación perfectos.

En este proyecto nos centraremos en el reconocimiento de emociones mediante el análisis de los rasgos faciales. Esto resulta viable puesto que, como hemos mencionado, las

emociones se caracterizan por producir respuestas biológicas en las personas, entre las que se encuentran las distintas expresiones faciales que pueden ser recopiladas en una base de datos de imágenes tal como FER2013, que a su vez se encuentran clasificadas en 7 clases muy similares a las propuestas por Ekman, por lo que una red neuronal convolucional entrenada con estos datos podría ser capaz de realizar una aproximación al reconocimiento de emociones.

3.1. Metodología

Para encontrar el mejor modelo para el reconocimiento de emociones, seguiremos la siguiente metodología:

En primer lugar evaluaremos el desempeño de varias arquitecturas de redes neuronales convolucionales ya existentes para la clasificación de la base de datos FER2013, para ello seguiremos la siguiente metodología para cada una de las arquitecturas:

- En la primera fase, evaluaremos el rendimiento de cada una de las arquitecturas por separado para determinar cuál es la mejor configuración de hiperparámetros posible.

Para ello se utilizará un algoritmo de búsqueda basado en **optimización bayesiana A.1** para probar las diferentes combinaciones de hiperparámetros y comprobar cuál es la más óptima para cada una de las arquitecturas.

Puesto que el algoritmo de búsqueda debe entrenar el modelo para cada posible combinación de hiperparámetros, utilizaremos el método de **validación holdout**, que como podemos observar en la figura 3.1, tras extraer de los datos el conjunto +de test, utiliza una única división fija entre el **conjunto de entrenamiento**, que como su propio nombre indica son los datos sobre los que la red realizará su aprendizaje mediante backtracking, y el **conjunto de validación** sobre el que se evaluarán, formado por el conjunto de datos que no ha sido utilizado para el aprendizaje. De esta manera, a diferencia de métodos que veremos más adelante como la validación cruzada, sólo será necesario entrenar el modelo una vez por cada posible configuración.

Para realizar la validación holdout, y puesto que nuestra base de datos es considerablemente grande, podemos permitirnos reservar un 20% del total de los datos como conjunto de test. Que únicamente se utilizará en la última validación para asegurar la consistencia del modelo.

Del 80% restante de la base de datos, reservaremos un 10% (del total) como conjunto de validación, y el resto (70% del total), será el conjunto utilizado para entrenar los modelos (3.1).

Puesto que en la validación holdout el conjunto de validación permanece fijo, cabe la posibilidad de que la división de la base de datos en entrenamiento/validación resulte, de forma fortuita, excesivamente bondadosa o perjudicial para el modelo en cuestión, por lo que tras obtener el mejor modelo, realizaremos una segunda validación sobre

los tres mejores modelos, esta vez utilizando el método de **validación cruzada**.

En este caso, como podemos observar en la figura 3.2, tras extraer el 20% de la base de datos para el test, dividiremos el resto de la base de datos en 8 conjuntos que utilizaremos para entrenar el modelo 8 veces. Para cada entrenamiento, utilizaremos uno de los 8 conjuntos como validación, y el resto para el entrenamiento, y las métricas finales las obtendremos a partir de la media de los 8 entrenamientos. De esta manera nos aseguraremos de que los resultados obtenidos en validación holdout no fueron fortuitos.

- Una vez realizada la validación cruzada e identificado el mejor modelo de entre los tres de manera consistente, realizaremos una última evaluación del modelo, esta vez introduciendo el 20% reservado que no se utilizó en las anteriores fases.

Para obtener resultados consistentes, repetiremos este entrenamiento 10 veces aleatorizando el reparto entre los conjuntos de entrenamiento y validación y para finalmente obtener la media.

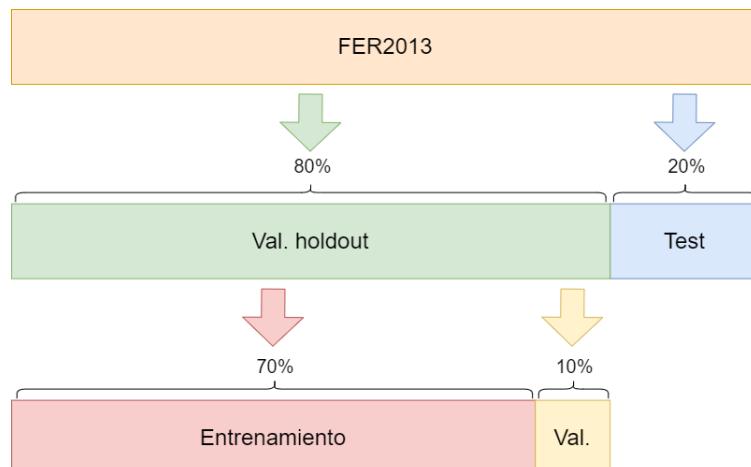


Figura 3.1: Validación holdout

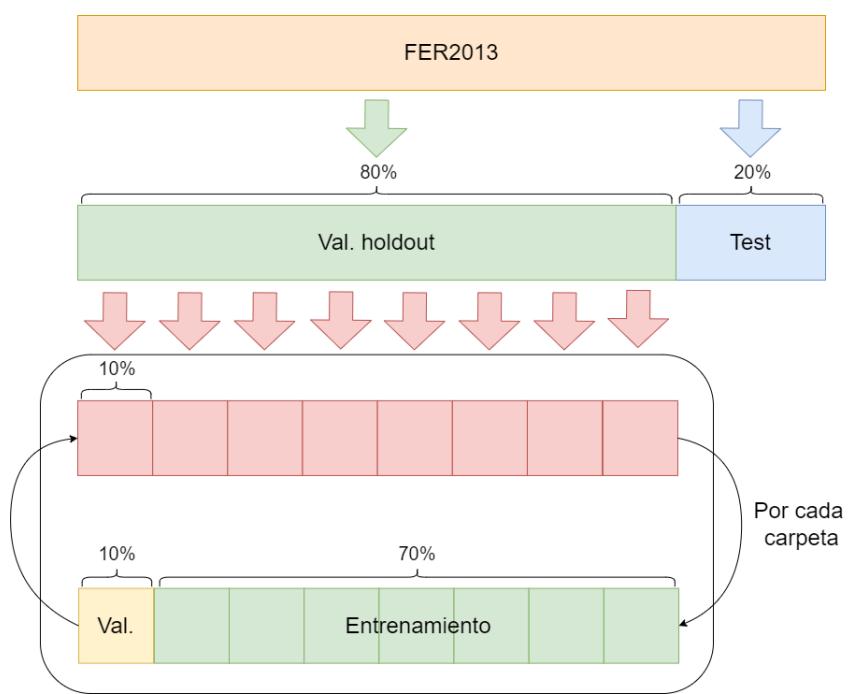


Figura 3.2: Validación cruzada

3.2. Arquitecturas existentes

En esta parte del proyecto nos dedicaremos a realizar un estudio del desempeño de arquitecturas de redes neuronales convolucionales ampliamente conocidas, todas ellas fueron desarrolladas para clasificar el conjunto de datos **ImageNet**, que contiene más de 14 millones con resolución 224x224 de imágenes correspondientes a 1000 clases de todo tipo, desde imágenes de personas hasta animales y todo tipo de objetos.

Para el entrenamiento de estos modelos utilizaremos una técnica llamada transfer Learning. Esta técnica consiste en utilizar una red neuronal preentrenada en lugar de entrenarla de cero con pesos aleatorios. En nuestro caso utilizaremos modelos cuya base ha sido pre-entrenada con ImageNet, esto, a pesar de que las clases de ImageNet no son las mismas que las de FER2013, permitirá al modelo aprovechar gran parte de las características más simples (las de las primeras capas y bloques convolucionales) para la clasificación de FER2013.

Como nombramos anteriormente, en primer lugar realizaremos una búsqueda bayesiana para optimizar los hiperparámetros para cada una de las arquitecturas, los hiperparámetros a optimizar serán los siguientes:

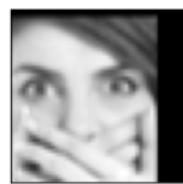
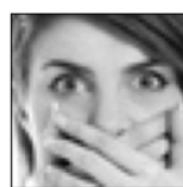
- **Learning rate:** Es la tasa de aprendizaje del modelo, consideraremos las tasas 0.01, 0.001 y 0.0001 dentro del espacio de búsqueda.
- **Aumento de datos:** el aumento de datos son transformaciones que se realizan sobre las imágenes con el objetivo de modificarlas levemente y así prevenir el sobreajuste durante el entrenamiento, valoraremos el añadir o no esta técnica durante el preprocesamiento de los datos.

Existen muchos tipos de transformaciones para aumentar los datos, para nuestros modelos utilizaremos cuatro: efecto espejo (figura 3.3a), traslaciones (figura 3.3b), zoom (figura 3.3c) y rotaciones (figura 3.3d). Estas cuatro transformaciones nos permitirán obtener variaciones de las imágenes sin llegar a modificar en extremo las originales.

- **Versión:** algunas de estas arquitecturas se presentan en diferentes versiones que introducen variaciones en tanto la profundidad como la estructura de la red, por lo que para cada una de las arquitecturas consideraremos distintas de sus versiones dentro del espacio de búsqueda.
- **Batch size:** representa el número de imágenes que recorrerán la red antes de realizar cada una de las actualizaciones de los pesos utilizando la media del gradiente obtenido, consideraremos los tamaños de batch size de 32, 64, 128 y 256 imágenes dentro del espacio de búsqueda.

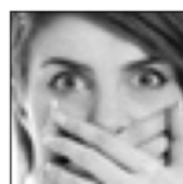
Existe un último hiperparámetro a optimizar, se trata de los epochs: un epoch representa que todas las entradas de la base de datos de entrenamiento han recorrido la red neuronal y se han utilizado para la actualización de sus pesos, tras cada epoch se evalúan las métricas obtenidas sobre el conjunto de validación.

El entrenamiento de las redes neuronales se divide en varios epochs, en nuestro caso utilizaremos una técnica llamada **Early Stopping** para que la red detenga el entrenamiento cuando esta deje de reflejar un aprendizaje, en nuestro caso se considerará que la red ha dejado de



(a) Efecto espejo

(b) Traslación



(c) Zoom

(d) Rotación

Figura 3.3: Aumento de datos

aprender cuando transcurran 6 epochs sin que la evaluación de la puntuación *F1-score* obtenida sobre el conjunto de validación aumente 0.001 unidades, por lo que no será necesario fijar el número de epochs ni incluirlo en el algoritmo de búsqueda.

Puesto que el espacio de búsqueda es relativamente grande, utilizaremos el sistema de validación conocido como validación holdout.

3.2.1. Definiciones

VGG

Visual Geometry Group (VGG) [Karen Simonyan, 2015] es una arquitectura de Red Neuronal Convolucional clásica. Se trata de una red neuronal con una profundidad considerable que varía en sus dos versiones, la versión **VGG16** cuenta con 16 capas mientras que **VGG19** cuenta con 19.

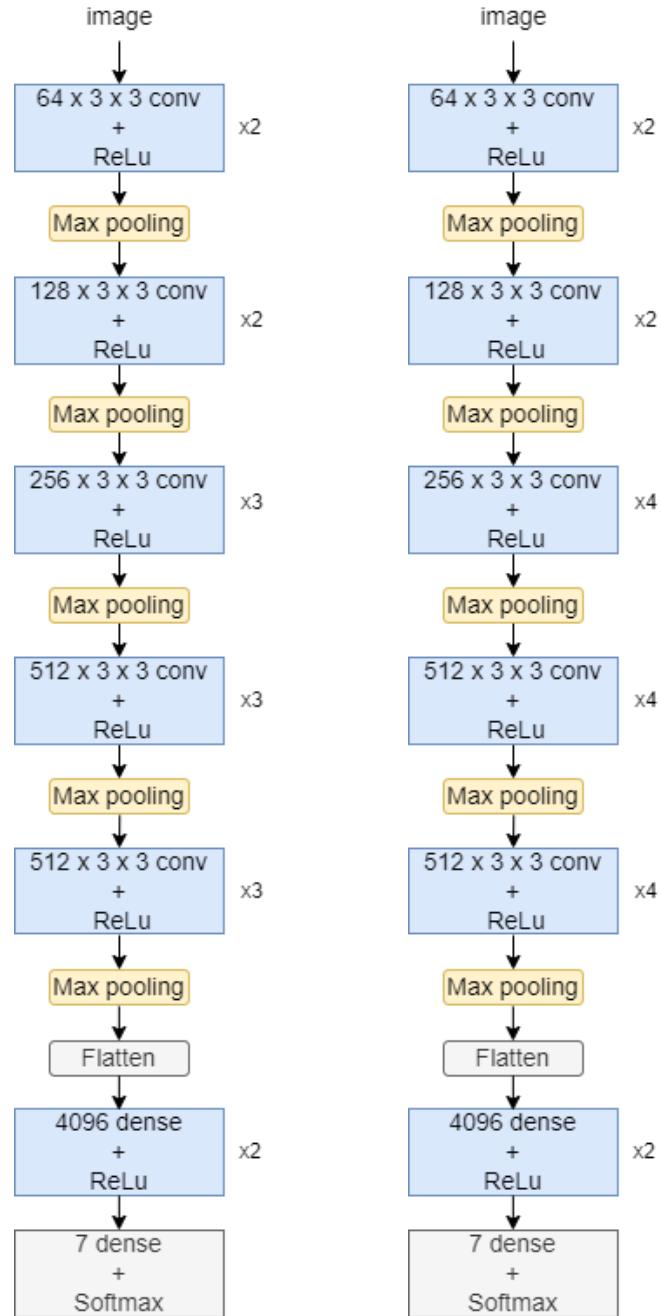
La arquitectura de dichas dos versiones se muestra en la figura 3.4.

Ambas redes están formadas por 5 bloques, formados por capas convolucionales con filtros de tamaño 3x3 con función de activación Relu a las que sigue una capa de **max pooling**. Dicha capa de max pooling cumplirá la función de condensación por máximos introducida en el apartado 2.2 reduciendo, además, las dimensiones de las imágenes a la mitad.

Las capas convolucionales del primer bloque generan 64 mapas de características, este número se multiplica por dos en cada uno de los tres siguientes bloques. Esto es posible sin aumentar en exceso las necesidades puesto que, como ya se nombró, cada capa de max pooling reducen las dimensiones de las imágenes a la mitad.

La diferencia entre ambas versiones reside en los últimos 3 bloques del cuerpo de la red, puesto que en la versión VGG16 están formados por 3 conjuntos **capas convolucional + ReLU** mientras que en VGG19 están formados por 4.

El cuerpo de la red está formado por dos capas densas totalmente conectadas de 4096 neuronas cada una seguida de la capa de clasificación con 1000 neuronas, pero puesto que en nuestro caso en concreto solo deberemos clasificar para 7 clases diferentes, debemos especializar la red reduciendo la última clase a 7 neuronas.



(a) Arquitectura VGG16

(b) Arquitectura VGG19

Figura 3.4: Arquitecturas VGG

ResNet

A la hora de diseñar la arquitectura de redes neuronales, se observa el siguiente fenómeno: la tasa de acierto de la red tiende a aumentar conforme se apilan capas y bloques en la red que aumentan la profundidad de la red, estos bloques generan mapas de características cada vez más complejos hasta que a partir de cierto umbral, el error empieza a aumentar debido al **problema del desvanecimiento del gradiente** (anexo E.1).

El nombre ResNet proviene de residual network [Kaiming He, 2015], y son redes cuya arquitectura cuenta con lo que denominamos como **bloques residuales**, que son bloques que incorporan "saltos" entre capas, por lo que, si consideramos $f(x)$ como la salida de la última capa del bloque y x la entrada del bloque, la función de salida del bloque corresponderá no a $f(x)$ como en un bloque tradicional, si no a $f(x) + x$, como se muestra en la figura 3.5. Esto mejora el flujo del gradiente permitiendo apilar más capas sin aumentar el error.

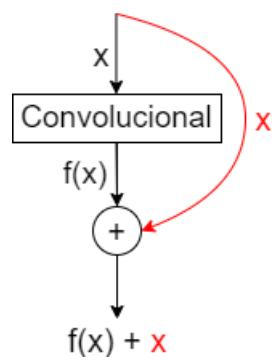


Figura 3.5: Bloque residual

En la figura 3.6 podemos ver una comparativa del error obtenido según el número de capas para dos arquitecturas distintas, la arquitectura 3.6a se trata de una red neuronal plana sin capas residuales mientras que la arquitectura 3.6b utiliza capas residuales, en el caso de la red neuronal plana (3.6a) observamos que la versión de 34 capas ha obtenido mayor error que la de 18, por lo que deja de ser conveniente apilar más capas, mientras que en el caso de la red con capas residuales (3.6b) podemos observar el fenómeno contrario: la versión con 34 capas obtiene menor error que la de 18, puesto que las capas residuales retrasan la aparición del problema del descenso del gradiente conforme se apilan capas.

Existen varias versiones de redes neuronales residuales, que por lo general varían en función del número de capas y del tipo de bloques residuales que utiliza:

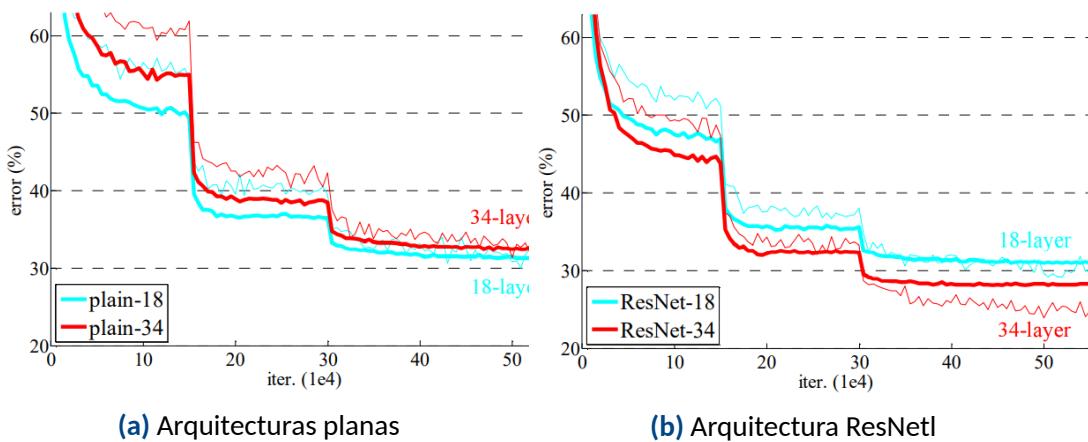
Según el tipo de bloques, podemos diferenciar entre las versiones **ResNetV1** y la más actual **ResNetV2**. En ambas versiones se utilizan las mismas capas ordenadas de distinta manera, la mayor diferencia se encuentra en que mientras que en ResNetV1 la última función de no-linealidad ReLU se aplica tras la suma de pesos, en ResNetV1 se realiza antes

(diagramas en anexo B.1).

Esto debería traducirse en que en la versión ResNetV2 el flujo del gradiente debería mejorar ya que no existe una función ReLU que desprecie los valores negativos.

Para la búsqueda, valoraremos la versión V2 de las arquitecturas **ResNet50**, **ResNet101** y **ResNet152**, dichas arquitecturas están formadas por la combinación de los cuatro bloques mostrados en el anexo B.2, y se diferenciarán únicamente en el número de dichos bloques que componen la arquitectura, siendo ResNet50 la versión menos profunda y ResNet152 la más profunda.

El diagrama de estas tres arquitecturas se muestra en la figura 3.7.



(a) Arquitecturas planas

(b) Arquitectura ResNet

Figura 3.6: Error según capas (extraído de [Kaiming He, 2015])

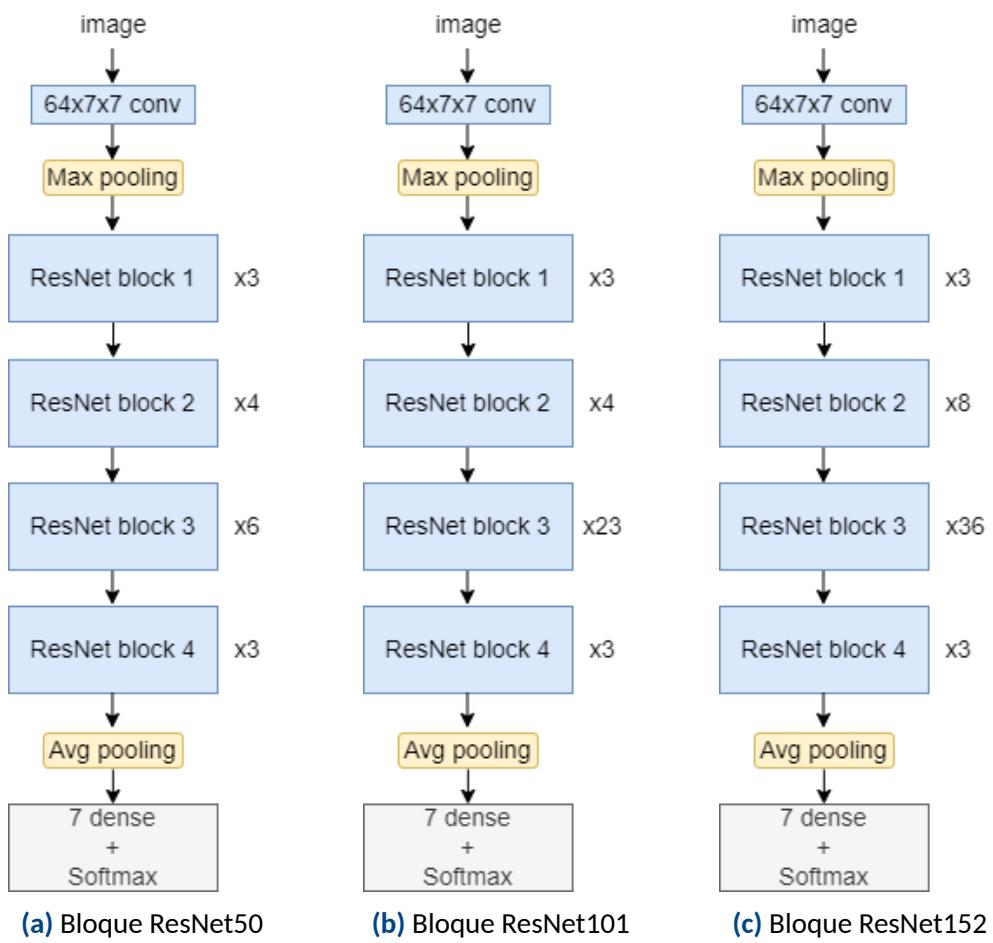


Figura 3.7: Arquitecturas ResNet

DenseNet

La arquitectura DenseNet [Gao Huang, 2016] se basa en maximizar el flujo de información por la red reutilizando el concepto de redes residuales para conectar cada una de las salidas convolucionales con el resto de salidas de su mismo bloque (véase el bloque 3.8). Para que esto sea posible, los mapas de características que fluyan dentro de un mismo bloque deben tener las mismas dimensiones.

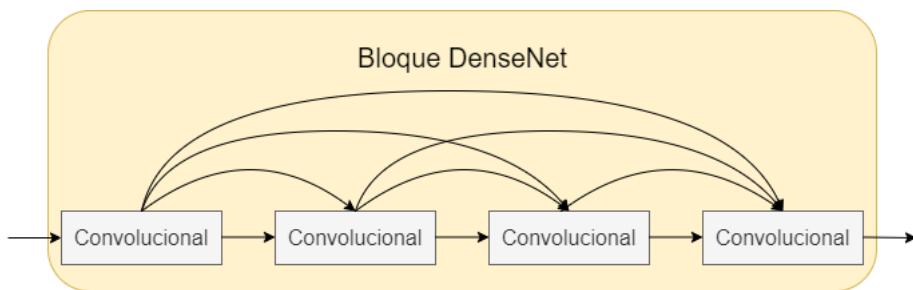


Figura 3.8: Bloque DenseNet

Cada bloque está formado a su vez por sub-bloques de capas convolucionales, estos, como podemos observar en la figura 3.9 están formados por dos secuencias batch normalization C.1 - ReLU - convolucional, la primera capa convolucional extrae 128 mapas de características con filtros 1x1, por lo que la función de estos filtros es únicamente regular el número de mapas de características que desembocan en la segunda secuencia, que obtiene 32 mapas de características con filtros 3x3.

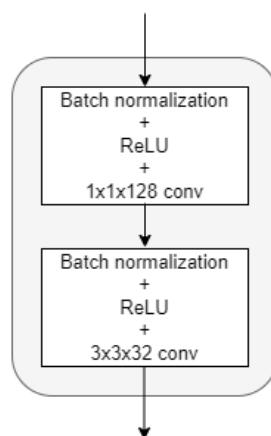


Figura 3.9: Bloque convolucional

En este caso y a diferencia de la arquitectura ResNet, al final de cada bloque los mapas de características en lugar de sumarse, se concatenan, por lo que la capa l recibirá l inputs provenientes de las l anteriores capas, y sus propios mapas de características pasarán a las $L - l$ capas siguientes, por lo que una red con L capas tendrá $\frac{L(L+1)}{2}$ conexiones.

Tras cada bloque DenseNet se aplica un **bloque de transición** al siguiente bloque cuyo objetivo es reducir las dimensiones de los mapas de características a la mitad, como podemos observar en la figura 3.10, este bloque está formado por la misma secuencia *normalización - ReLU - convolucional* que desemboca en una capa de average pooling 2x2 que reduce las dimensiones a la mitad.

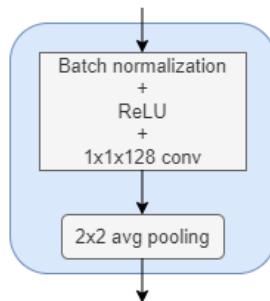


Figura 3.10: Bloque de transición

La reutilización de filtros en este tipo de redes reduce la tendencia de las mismas a aprender filtros redundantes, esto implica que se pueda obtener un gran rendimiento de utilizar capas con pocos filtros que agregarán los filtros generados al que en este caso, puesto que el clasificador realizará una predicción basándose en todos los mapas generados en la red (y no solo los últimos), se considera el conocimiento colectivo de la red.

Otra ventaja de este tipo de redes es la mejora del flujo del gradiente con respecto a las redes ResNet, esto es debido a que como vimos, las redes ResNet suman los mapas de características al final de cada bloque residual, lo que implica que la salida del bloque corresponda a $f(x) + x$, siendo f la transformación llevada a cabo por el bloque, lo que empeora el flujo del gradiente, mientras que las salidas de los bloques DenseNet corresponden a $f([x_0, x_1 \dots x_{l-1}])$ siendo $[x_0, x_1 \dots x_{l-1}]$ la concatenación de las capas anteriores de la red.

Además, las conexiones densas tienen un efecto regularizador que reduce el sobreajuste cuando se utilizan bases de datos pequeñas.

Existen varias versiones de la arquitectura DenseNet, en nuestro caso consideraremos tres de ellas: **DenseNet121**, **DenseNet169**, y **DenseNet201**, que variarán únicamente en el número de bloques convolucionales por bloque DenseNet. Como podemos ver en la figura B.1 solo se diferenciarán en el número de bloques convolucionales en los dos últimos bloques DenseNet, siendo DenseNet121 la arquitectura con menos capas y DenseNet201 la arquitectura con más capas.

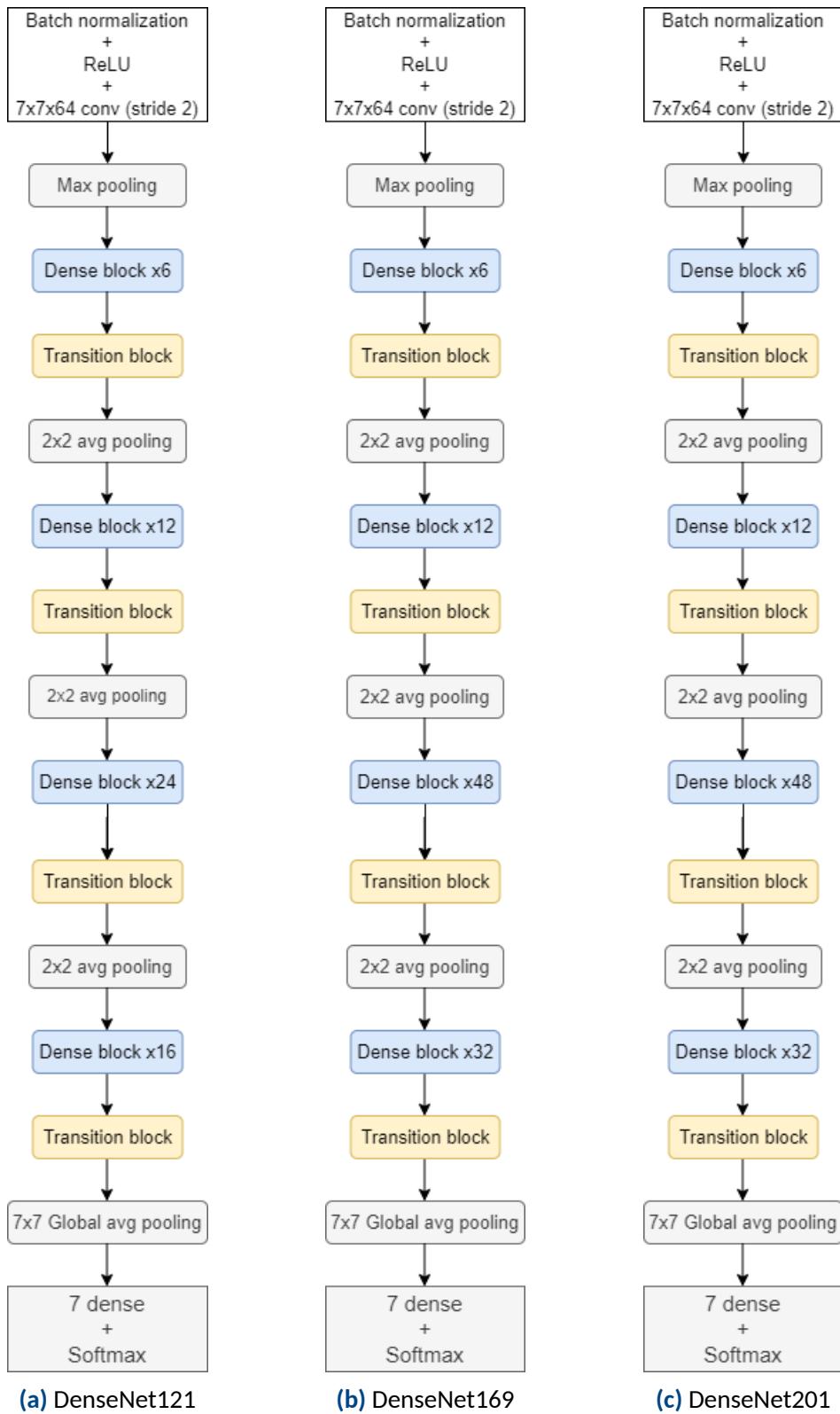


Figura 3.11: Arquitecturas DenseNet

Inception

El concepto tras la arquitectura Inception [Christian Szegedy, 2015] se basa en dejar de lado el aumento en la profundidad de las redes y optar por aumentar la "anchura" de las mismas, aplicando filtros de distintos tamaños sobre una misma entrada para aprender los distintos patrones.

Como podemos observar en la figura 3.12, la primera versión de Inception cuenta con módulos que aplican filtros 1x1, 3x3 y 5x5 sobre una misma entrada, además de una capa de max pooling de 3x3 que en este caso no aplicará reducción de las dimensiones de manera que las salidas de estas 4 capas se puedan concatenar. También se aplicarán convoluciones 1x1 a las capas convolucionales de 3x3 y 5x5 y a la capa de max pooling con el objetivo de reducir los mapas de características generados.

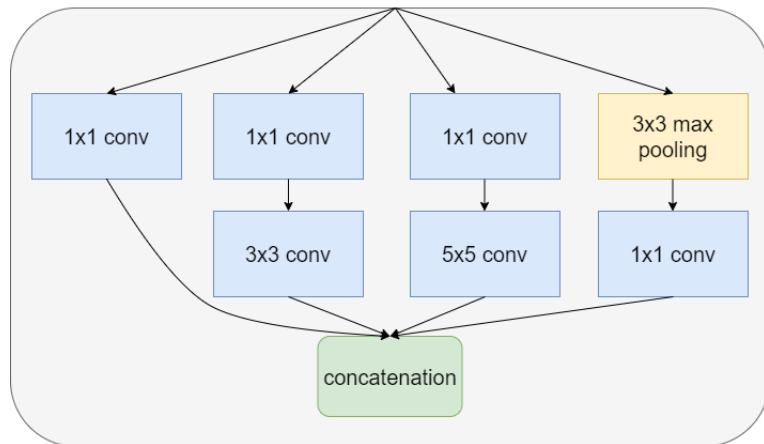


Figura 3.12: InceptionV1

Estos módulos se apilan para formar lo que se conoce como arquitectura **GoogleLeNet**, que está formada por 9 de estos módulos que desembocan en una capa de global average pooling. Además, la red cuenta con clasificadores auxiliares a lo largo de la estructura de la red adicionales al clasificador final cuyo objetivo será optimizar la fase de entrenamiento de la red. Dichos clasificadores auxiliares serán utilizados sólo durante el entrenamiento de la red, siendo eliminados para la fase de inferencia.

La segunda versión de Inception, conocida como **InceptionV2** [Christian Szegedy, 2015], combina tres tipos diferentes de bloques. Esta arquitectura propone mejorar el rendimiento computacional dividiendo los filtros de mayor tamaño en filtros más pequeños.

En este caso, se dividen los filtros que aplican las capas convolucionales de 3x3 y 5x5 en filtros más pequeños pero equivalentes, por lo que se sustituyen los filtros 5x5 por filtros 3x3, y estos a su vez por filtros 3x1 y 1x3, además, siguiendo la filosofía de Inception de no aumentar en exceso la profundidad de la red, las nuevas capas añadidas se expanden en

anchura y no en profundidad (véase anexo B.3).

InceptionV3 [Christian Szegedy, 2015] se trata de una versión mejorada de la arquitectura InceptionV2 sin aportar cambios drásticos a la misma. Entre las mejoras destaca la utilización del optimizador RMSProp, la incorporación de filtros convolucionales de mayor tamaño (7×7) y la regularización del modelo mediante la utilización de batch normalization en los clasificadores auxiliares y el suavizado de la predicción final.

La cuarta versión de la arquitectura Inception (**InceptionV4**) [Christian Szegedy, 2016] trata de modificar las versiones anteriores aportando módulos más uniformes y simples. En primer lugar se modifica secuencia inicial de capas de la red previa al primer módulo, podemos observar el diagrama de dicha secuencia en el anexo B.4.

Esta arquitectura utiliza tres módulos para construir la red (véase anexo B.5). Este tipo de arquitectura también se caracteriza por introducir bloques de reducción implícitos, a diferencia de las anteriores versiones en las que esta reducción se aplicaba dentro de los propios bloques Inception. Podemos ver los dos tipos de bloques de reducción utilizados en el anexo B.8.

La arquitectura **Inception-Resnet** [Christian Szegedy, 2016] trata de reutilizar los conceptos introducidos en InceptionV4 pero añadiendo también las virtudes de las capas residuales para mejorar el flujo del gradiente de igual manera que en ResNet.

Esta arquitectura, al igual que InceptionV4, utiliza tres tipos de bloques distintos (véase anexo B.7). En estos se aplica una última capa convolucional de 1×1 entre la capa de concatenación del bloque Inception y la capa que suma el resultado del bloque con la salida residual, esto es debido a que la operación de suma de mapas de características requiere que ambas salidas tengan la misma profundidad. Esta necesidad de unificar las dimensiones de las salidas también ocasiona la eliminación total de capas de max pooling para reducir la dimensionalidad dentro de los bloques Inception siendo los bloques de reducción (muestraos en el anexo B.8) en este caso los únicos en aplicar dicha reducción.

Para la búsqueda consideraremos en el espacio de búsqueda dos de las versiones más modernas de la arquitectura Inception: **InceptionV3** e **Inception-ResNet** (3.13).

Los bloques utilizados por estas arquitecturas serán los definidos previamente.

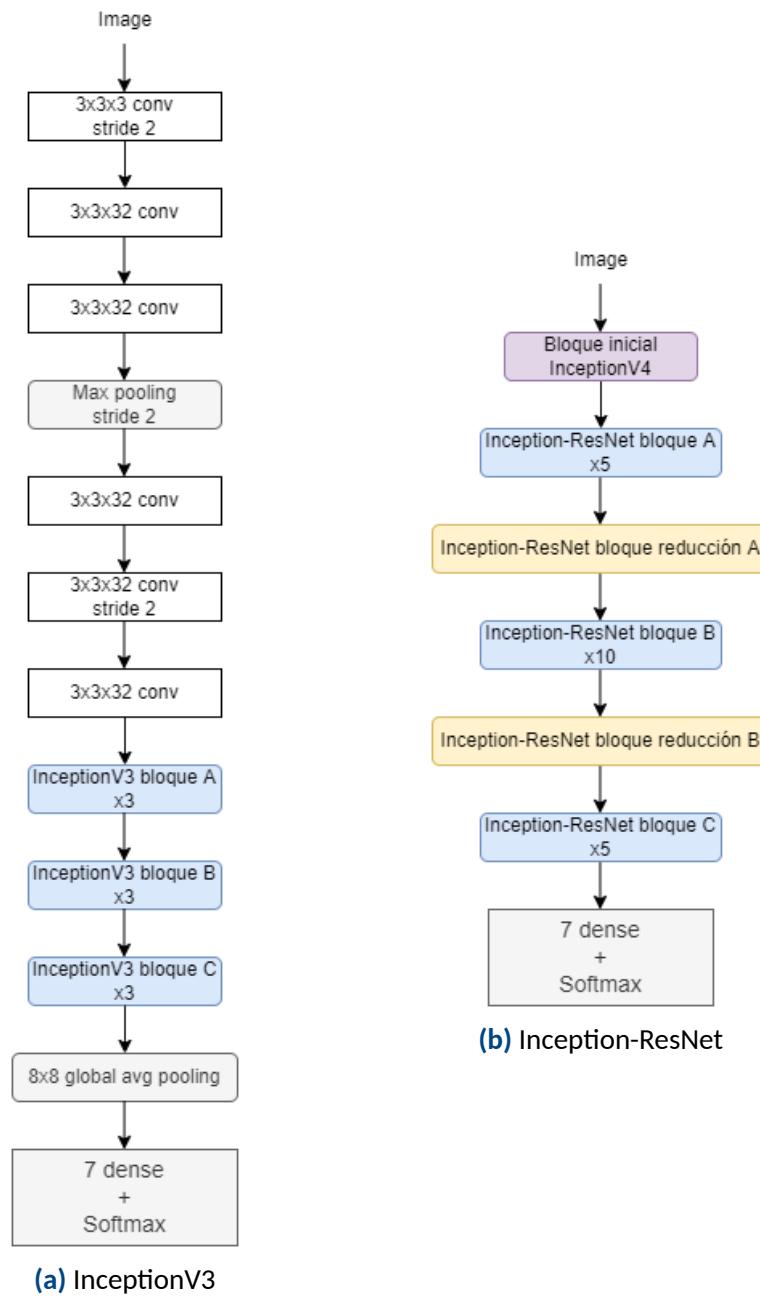


Figura 3.13: Arquitecturas InceptionV3 e Inception-ResNet

3.2.2. Búsqueda

En esta sección vamos a ejecutar el algoritmo de búsqueda bayesiana para la optimización de las arquitecturas. Como ya se comentó anteriormente, se optimizarán los hiperparámetros correspondientes al tipo de optimizador, el learning rate, el uso o no de aumento de datos y el batch-size, además de la versión de la arquitectura, siendo este último el único hiperparámetro que variará entre arquitecturas.

VGG

Para la búsqueda, además de los hiperparámetros comunes a todas las búsquedas consideraremos las versiones **VGG16** y **VGG19**.

Versión	Optimizador	LR	Aumento datos	Batch size	Accuracy	Precisión	Recall	F1
VGG16	Adam	0.0001	False	256	0.6447	0.6338	0.5999	0.6086
VGG16	Adam	0.0001	True	256	0.6524	0.6399	0.5940	0.6023
VGG19	Adam	0.0001	True	256	0.6319	0.6272	0.5808	0.5899
VGG19	Adam	0.0001	False	256	0.6285	0.6063	0.5886	0.5892
VGG16	Adam	0.0001	True	32	0.6701	0.5929	0.5850	0.5652

Tabla 3.1: Mejores modelos VGG

Como podemos observar en la tabla 3.1, ciertos hiperparámetros como el optimizador *Adam* con un *learning rate* de 0.0001 y un *batch size* relativamente alto de 256 parecen afectar de manera muy positiva al modelo, otros, como la versión y el aumento de datos parecen afectar en menor medida.

Podemos observar que el modelo que mejor *F1-score* ha obtenido en esta primera búsqueda se trata de la versión VGG16 con optimizador Adam, learning rate de 0.0001, sin aumento de datos y con batch size de 256.

En cuanto a la métrica de accuracy, observamos que no es estrictamente proporcional a la *F1-score* obtenida, pues como podemos observar en la tabla, el quinto modelo en cuanto a mejor *F1-score* obtenida es el primero en cuanto a accuracy. Esta singularidad surge a raíz del desbalanceo de los datos, que como ya mencionamos, hace que debamos considerar *F1-score* como la métrica más adecuada para la comparación.

Tras esta primera búsqueda, procedemos a ejecutar validación cruzada sobre los 3 mejores modelos, junto con la media y desviación típica de cada una de las métricas.

Validación del primer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
VGG16	Adam	0.0001	False	256

Tabla 3.2: Primer modelo VGG

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6478	0.6151	0.6016	0.5982
1	0.6512	0.6335	0.5916	0.5977
2	0.6088	0.6004	0.562	0.5644
3	0.6252	0.6331	0.6196	0.6173
4	0.6266	0.595	0.5903	0.5863
5	0.6237	0.6421	0.578	0.5894
6	0.6452	0.6448	0.5973	0.6075
7	0.648	0.6405	0.5969	0.6062
Media	0.6345	0.6255	0.5921	0.5958
Media (%)	63.45%	62.55%	59.21%	59.58%
Desviación típica	0.0155	0.0195	0.0169	0.0162

Tabla 3.3: Validación cruzada primer modelo VGG

Como podemos observar, obtenemos una F1-score final media de 0.5958 y una accuracy final media de 0.6345, resultados muy consistentes considerando los obtenidos durante la búsqueda.

Validación del segundo modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
VGG16	Adam	0.0001	True	256

Tabla 3.4: Segundo modelo VGG

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.5865	0.6015	0.5701	0.5718
1	0.6041	0.6024	0.5795	0.5776
2	0.5977	0.5921	0.5579	0.5632
3	0.5762	0.5687	0.5462	0.548
4	0.603	0.548	0.5683	0.5481
5	0.5914	0.5604	0.5475	0.5456
6	0.6009	0.5894	0.5711	0.5724
7	0.5889	0.5948	0.5564	0.5622
Media	0.5936	0.5822	0.5621	0.5611
Media (%)	59.36%	58.22%	56.21%	56.11%
Desviación típica	0.0096	0.0204	0.0120	0.0125

Tabla 3.5: Validación cruzada segundo modelo VGG

Como podemos observar, obtenemos una *F1-score* media de 0.5611, cifra considerablemente menor a la obtenida en el primer mejor modelo, y una accuracy final media de 0.5936, también menor a la obtenida en el primer modelo. Esto resulta interesante teniendo en cuenta que como observamos durante la búsqueda (resultados en tabla 3.1), el segundo mejor modelo en cuanto a *F1-score* obtenía mejor accuracy, por lo que deducimos que esta buena puntuación se debió más a la bondad aleatoria de los conjuntos utilizados para la búsqueda que a una superioridad real.

Validación del tercer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
VGG16	Adam	0.0001	True	256

Tabla 3.6: Tercer modelo VGG

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.5946	0.6147	0.5777	0.5859
1	0.5831	0.5917	0.5831	0.5757
2	0.611	0.5958	0.5635	0.571
3	0.5782	0.5647	0.5237	0.5273
4	0.6091	0.5830	0.557	0.5609
5	0.5981	0.5967	0.5667	0.5696
6	0.6115	0.5937	0.5524	0.5606
7	0.6073	0.5859	0.5662	0.5680
Media	0.5991	0.5908	0.5613	0.5649
Media (%)	59.91%	59.08%	56.13%	56.49%
Desviación típica	0.0130	0.0142	0.0182	0.0172

Tabla 3.7: Validación cruzada tercer modelo VGG

En este caso obtenemos una F1-score de 0.5649 y una accuracy de 0.5991. Esto sitúa a este modelo como el segundo mejor obtenido, por encima del segundo mejor modelo obtenido durante la búsqueda.

Mejor modelo obtenido:

Versión	Optimizador	LR	Aumento datos	Batch size
VGG16	Adam	0.0001	False	256

Tabla 3.8: Mejor modelo VGG

En este caso el mejor modelo obtenido en la búsqueda se corresponde con el mejor modelo obtenido durante la validación cruzada. En la tabla 3.9 podemos ver el resultado de ejecutar 10 veces el modelo, entrenándolo con el conjunto completo de entrenamiento (80%) y validando con el conjunto de test (20%). Por cada iteración, como se introdujo en el apartado 3.1, se aleatoriza el reparto del conjunto de datos entre los conjuntos de entrenamiento y validación.

Iteración	Accuracy	Precisión	Recall	F1
0	0.6461	0.6102	0.6321	0.6121
1	0.6133	0.5679	0.6136	0.5783
2	0.6167	0.5779	0.6299	0.5857
3	0.6095	0.5695	0.5962	0.572
4	0.6241	0.5874	0.6321	0.5999
5	0.6275	0.6068	0.6419	0.6107
6	0.6244	0.5909	0.6352	0.5997
7	0.6174	0.5962	0.5996	0.5876
8	0.6266	0.5836	0.6171	0.5917
9	0.5972	0.5528	0.5706	0.5511
Media	0.6203	0.5843	0.6168	0.5889
Media (%)	62.03%	58.43%	61.68%	58.89%
Desviación típica	0.0129	0.0179	0.0223	0.0185

Tabla 3.9: Iteraciones modelo final VGG

Como podemos observar en la tabla 3.9, obtenemos una F1-score media de 0.5889 y una accuracy de 0.6203, resultados muy consistentes teniendo en cuenta los obtenidos en la validación cruzada. La fila de la tabla de resultados marcada en amarillo corresponde al

mejor modelo obtenido, que en este caso se obtuvo en la iteración 0.

Tras la elección del modelo óptimo podemos analizar el rendimiento del clasificador para cada una de las 7 clases, para ello utilizaremos dos métricas adicionales, **la matriz de confusión** (introducida en apartado 2.4.2) y las **curvas ROC** (introducidas en apartado 2.4.2). Para esta evaluación utilizaremos el mejor modelo obtenido de entre las 10 iteraciones de la validación final (modelo marcado en amarillo en tabla de resultados 3.9).

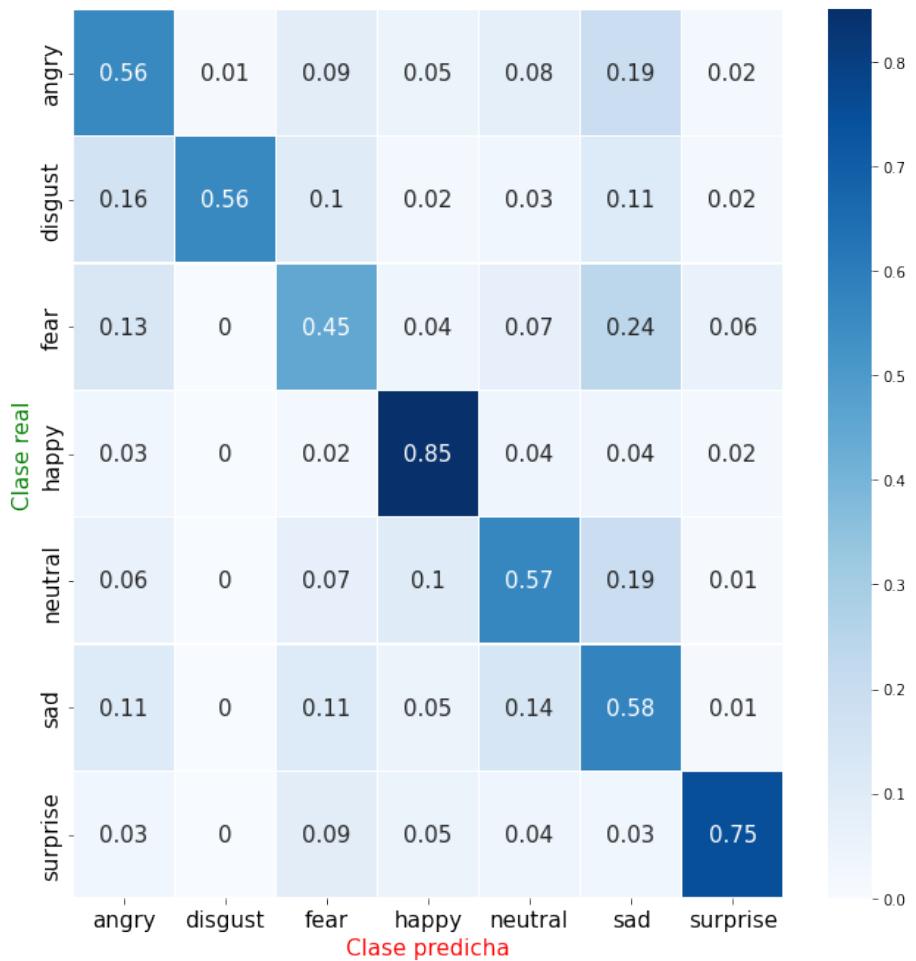


Figura 3.14: VGG matriz de confusión

En la figura 3.14 se muestra la matriz de confusión obtenida, el eje vertical representa los verdaderos valores de las clases mientras que el eje horizontal representa los valores predichos por el clasificador.

La matriz está representada en formato de mapa de calor, por lo que cuanto más cercano sea el valor de la celda al valor óptimo de 1, el color de la celda intersección será más oscuro.

Como podemos observar, se percibe una clara diagonal en la matriz, esta diagonal representa la intersección entre el valor de la clase real y la predicha de manera correcta y es

indicativa del buen rendimiento del modelo.

Como podemos observar, la clase "happy" obtiene el mejor resultado de todos (0.85), esto era de esperar puesto que, como se mostró en 2.3, es la clase mayoritaria en la base de datos (25.05% del total). También destaca el buen resultado obtenido para la clase "surprise", que pese a no ser una clase mayoritaria (solo representa el 11.15% del total), obtiene una puntuación de 0.75. En cuanto al resto de las clases, todas se mueven entorno al 0.57 salvo la clase "fear", obteniendo esta última una puntuación por debajo del 0.5.

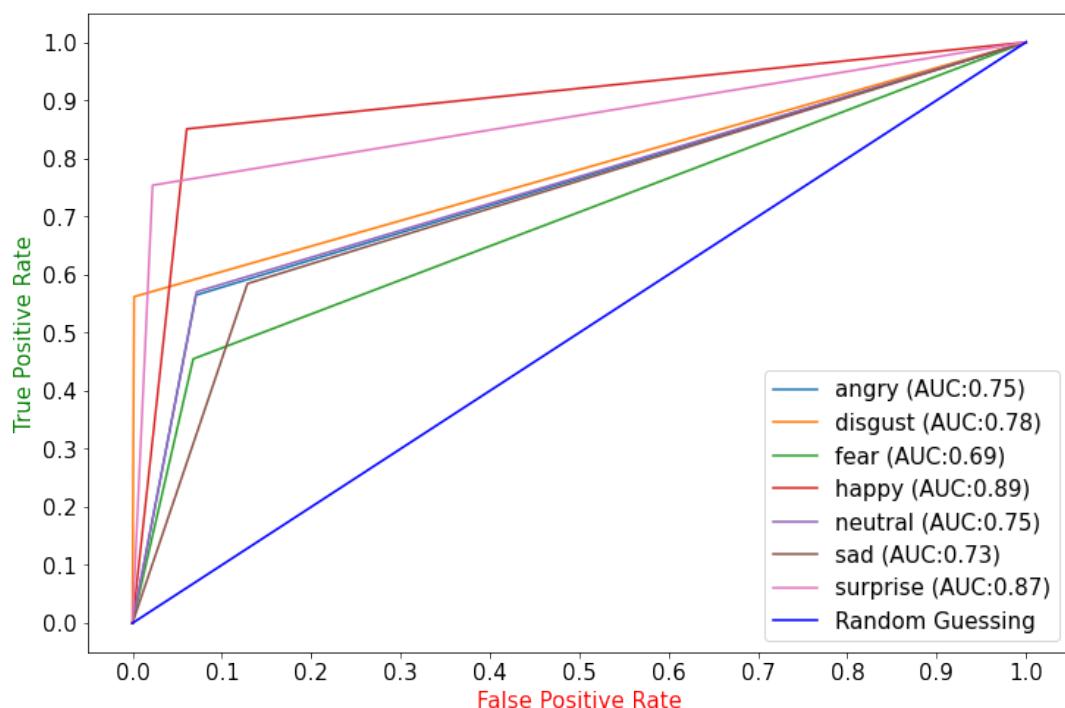


Figura 3.15: VGG curvas ROC

En cuanto a las curvas ROC (figura 3.15), podemos apreciar que la clase minoritaria "fear" obtiene la menor tasa de falsos positivos (la mas cercana a 0), esto, junto con el mal resultado obtenido para esta misma clase en la matriz de confusión, refleja la tendencia del modelo a no clasificar las imágenes como pertenecientes a la clase "fear".

La puntuación AUC media de todas las clases para este clasificador es de 0.7882, lo que lo posiciona, según la tabla 2.1 en el rango de discriminador aceptable.

ResNet

Para la búsqueda, además de los hiperparámetros comunes a la otras búsquedas, consideraremos tres versiones de la versión V2 de ResNet, cada una caracterizada por la profundidad de la red: **ResNet50V2**, **ResNet101V2**, **ResNet152V2**.

Los 5 mejores modelos obtenidos se muestran en la tabla 3.10.

Versión	Optimizador	LR	Aumento datos	Batch size	Accuracy	Precisión	Recall	F1
ResNet152V2	Adam	0.0001	True	256	0.6057	0.5880	0.5357	0.5416
ResNet152V2	Adam	0.0001	False	256	0.5848	0.5619	0.5396	0.5409
ResNet50V2	Adam	0.0001	True	256	0.6010	0.5734	0.5282	0.5391
ResNet50V2	SGD	0.01	True	256	0.5776	0.5345	0.5234	0.5190
ResNet50V2	Adam	0.0001	False	256	0.5662	0.5317	0.5210	0.5163

Tabla 3.10: Mejores modelos ResNet

Como se puede apreciar en la tabla de resultados 3.10, el mejor modelo corresponde al modelo de mayor profundidad ResNet152V2, con una *F1-score* de 0.5416 y una accuracy de 0.6057. También podemos apreciar que la diferencia en cuanto a la *F1-score* entre el primer y segundo mejor modelo es mínima, con únicamente 0.0007 unidades de diferencia. Por lo que en este caso cobra especial importancia la ejecución de la segunda validación para asegurar resultados consistentes.

Podemos apreciar que de nuevo los hiperparámetros dominantes son el optimizador *Adam* con un batch size de 256, también se observa que la arquitectura de más profundidad (ResNet152V2) es la más óptima para el problema, seguida por ResNet50V2, siendo Resnet101V2 la que peores resultados obtiene. También observamos por primera vez el uso del optimizador *SGD* en uno de los mejores modelos obtenidos (en este caso el cuarto), que como podemos ver, se beneficia de un mayor learning rate (0.01).

Ahora pasamos a ejecutar validación cruzada con 8 carpetas sobre los tres mejores modelos obtenidos.

Validación del primer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
ResNet152V2	Adam	0.0001	True	256

Tabla 3.11: Primer modelo ResNet

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.5954	0.5779	0.5416	0.5468
1	0.6141	0.5942	0.5628	0.5682
2	0.6169	0.576	0.5497	0.5545
3	0.6186	0.6143	0.5885	0.5923
4	0.6085	0.5855	0.557	0.5594
5	0.6037	0.5818	0.5452	0.5461
6	0.6001	0.5762	0.5452	0.553
7	0.5934	0.533	0.5247	0.5209
Media	0.6063	0.5799	0.5518	0.5552
Media (%)	60.63%	57.99%	55.18%	55.52%
Desviación típica	0.0097	0.0229	0.0186	0.0203

Tabla 3.12: Validación cruzada primer modelo ResNet

Como podemos observar en la tabla 3.12, obtenemos una F1-score de 0.5552, y una accuracy de 0.6063, métricas algo superiores a las obtenidas para este modelo durante la búsqueda. También cabe resaltar que la desviación típica obtenida, en especial en precision y F1-score (0.0229 y 0.0203 respectivamente) es algo alta si la comparamos con otros modelos, lo que puede ser indicativo de una baja consistencia del modelo a la hora de variar las particiones.

Validación del segundo modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
ResNet152V2	Adam	0.0001	False	256

Tabla 3.13: Tercer modelo VGG

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.5913	0.5759	0.5486	0.548
1	0.6091	0.617	0.5829	0.5879
2	0.5929	0.5822	0.5441	0.5533
3	0.5913	0.5883	0.56	0.5665
4	0.6105	0.5919	0.5777	0.5769
5	0.6087	0.5796	0.5757	0.5681
6	0.612	0.6065	0.5565	0.5698
7	0.5964	0.5989	0.5622	0.5671
Media	0.6015	0.5925	0.5635	0.5672
Media (%)	60.15%	59.25%	56.35%	56.72%
Desviación típica	0.0093	0.0142	0.0141	0.0125

Tabla 3.14: Validación cruzada segundo modelo ResNet

En la validación de este segundo modelo (tabla 3.14) podemos observar que obtiene una F1-score media de 0.5672, superior a la obtenida en el primer modelo de 0.5552, y una accuracy media de 0.6063 que resulta ligeramente inferior a la obtenida en el primer modelo (0.6015). Además, en este caso encontramos valores más bajos de desviación típica que los obtenidos durante la validación del primer modelo, lo que nos indica una mayor consistencia del modelo y lo convierte en un mejor candidato como representante más óptimo de la arquitectura.

Validación del tercer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
ResNet50V2	Adam	0.0001	True	256

Tabla 3.15: Tercer modelo VGG

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.5868	0.5687	0.5417	0.5437
1	0.6049	0.5692	0.5575	0.5498
2	0.6124	0.5766	0.5432	0.5455
3	0.5904	0.5681	0.5338	0.5378
4	0.6261	0.606	0.5753	0.5809
5	0.6054	0.6069	0.5567	0.5694
6	0.604	0.5754	0.5454	0.5481
7	0.6154	0.5982	0.5668	0.5705
Media	0.6057	0.5836	0.5525	0.5557
Media (%)	60.57%	58.36%	55.25%	55.57%
Desviación típica	0.0128	0.0171	0.0140	0.0156

Tabla 3.16: Validación cruzada tercer modelo ResNet

Como observamos en 3.16, en este caso obtenemos una F1-score de 0.5557 y una accuracy media de 0.6057, valores muy parecidos a los obtenidos en la primera validación, con la diferencia de que en este caso, las desviaciones típicas obtenidas y en consecuencia la consistencia del modelo son algo más razonables, lo que convierte a este en el segundo mejor modelo obtenido.

Como hemos podido observar durante la validación, en este caso la superioridad del primer mejor modelo con respecto a los dos siguientes obtenidos en la búsqueda se debió, entre otras cosas, a la bondad de las particiones utilizadas como conjuntos de entrenamiento y la validación de los modelos, esto explica la elevada desviación típica obtenida para este modelo.

Por este motivo, se pasa a considerar el segundo modelo obtenido durante la búsqueda bayesiana como el representante más óptimo para la arquitectura y problema concretos.

Mejor modelo obtenido:

Versión	Optimizador	LR	Aumento datos	Batch size
ResNet152V2	Adam	0.0001	False	256

Tabla 3.17: Mejor modelo ResNet

Iteración	Accuracy	Precisión	Recall	F1
0	0.6009	0.5617	0.5742	0.5592
1	0.6092	0.5731	0.6095	0.5804
2	0.6134	0.5723	0.6135	0.5799
3	0.6028	0.5635	0.6153	0.5724
4	0.5978	0.5783	0.5983	0.5747
5	0.5979	0.5888	0.6036	0.5884
6	0.5861	0.5784	0.5939	0.5726
7	0.6087	0.568	0.588	0.5684
8	0.6014	0.5522	0.6013	0.5622
9	0.5861	0.559	0.5692	0.5532
Media	0.6004	0.5695	0.5967	0.5711
Media (%)	60.04%	56.95%	59.67%	57.11%
Desviación típica	0.0091	0.0108	0.0157	0.0107

Tabla 3.18: Iteraciones modelo final Resnet

Tras entrenar el modelo utilizando el conjunto completo de entrenamiento, podemos observar que obtenemos una F1-score media de 0.5711, algo superior a la obtenida du-

rante las anteriores fases del estudio, y una accuracy de 0.6004, con desviaciones típicas y consistencia razonables.

Ahora pasamos a analizar la matriz de confusión y curvas ROC-AUC obtenidas para cada una de las clases para el mejor modelo tras las 10 iteraciones finales mostradas en 3.18 (modelo resaltado en amarillo).

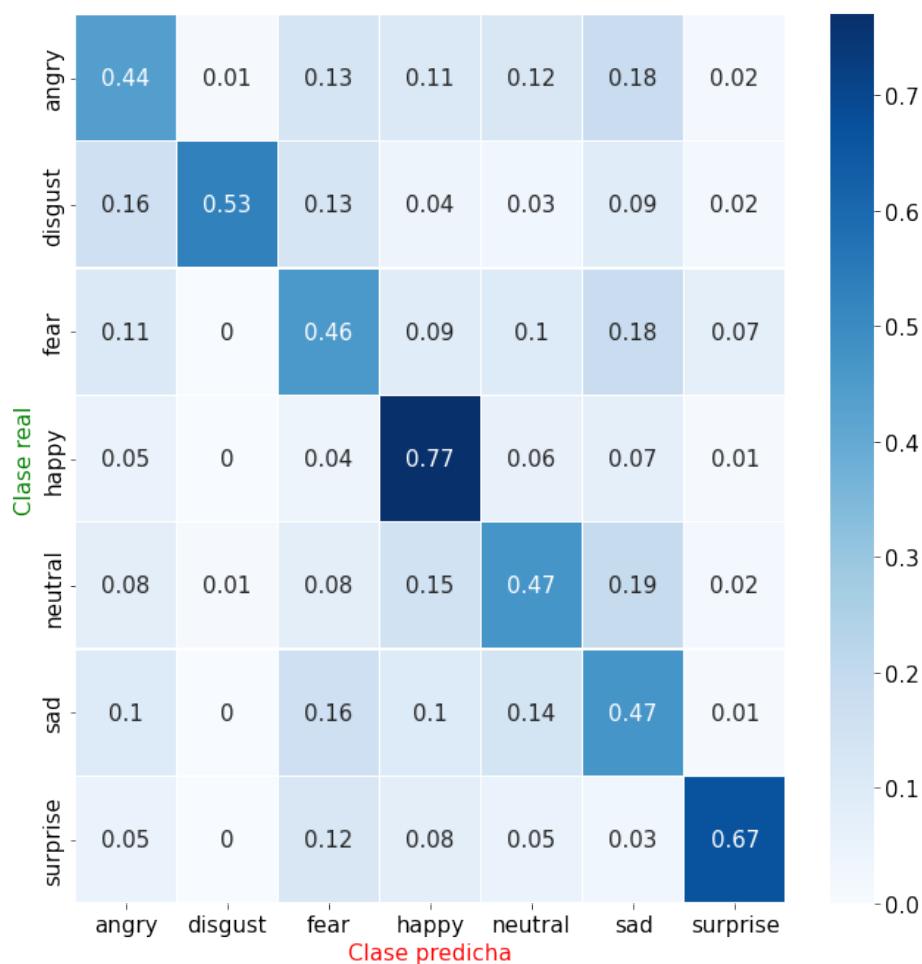


Figura 3.16: ResNet matriz de confusión

En la figura 3.16 se muestra la matriz de confusión obtenida. En este caso, como era de esperar, se observan peores resultados que los obtenidos en VGG. Observamos que de nuevo el mejor valor (0.77) corresponde a la clase "happy" y el peor (0.46) a la clase "fear".

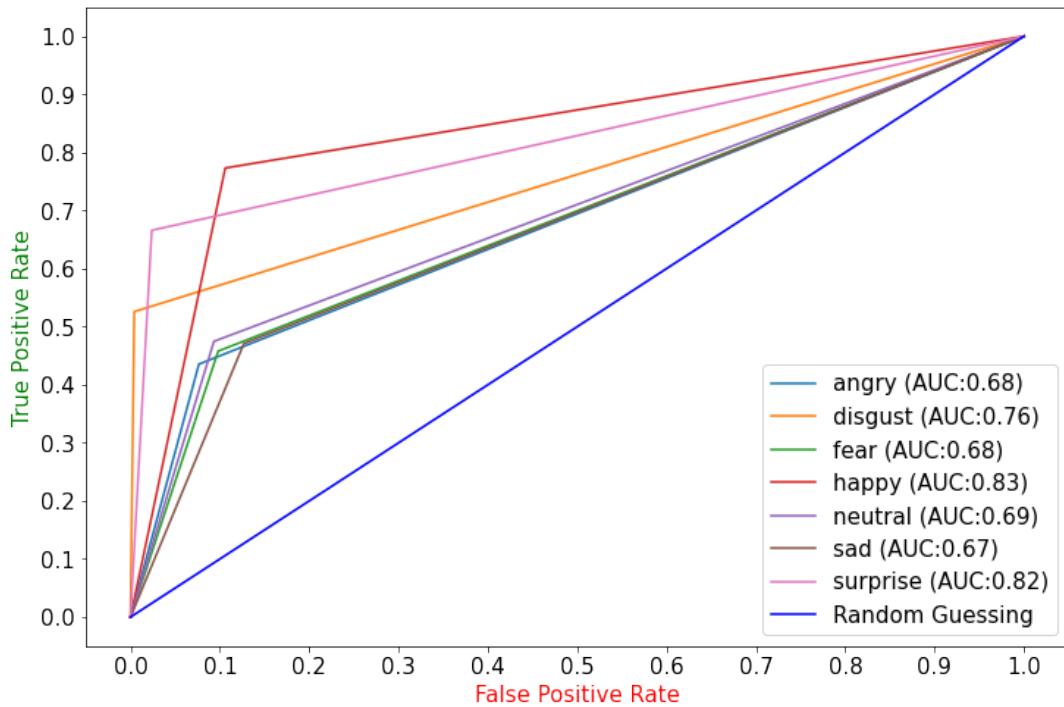


Figura 3.17: ResNet curvas ROC

En la figura 3.17 se muestran las curvas ROC obtenidas para cada clase. El modelo obtiene una puntuación AUC media de 0.7445, que a pesar de ser menor que la obtenida para el modelo anterior VGG, según la tabla 2.1 puede considerarse como un discriminador aceptable.

DenseNet

En esta sección aplicaremos la búsqueda de optimización a la arquitectura DenseNet. En este caso consideraremos las versiones **DenseNet121**, **DenseNet169** y **DenseNet201**.

Versión	Optimizador	LR	Aumento datos	Batch size	Accuracy	Precisión	Recall	F1
DenseNet201	Adam	0.0001	False	256	0.6447	0.6338	0.5999	0.5636
DenseNet201	Adam	0.0001	True	256	0.6524	0.6399	0.5391	0.5422
DenseNet121	Adam	0.0001	True	256	0.6319	0.6272	0.5808	0.5380
DenseNet121	Adam	0.0001	False	256	0.6285	0.6063	0.5886	0.5371
DenseNet121	Adam	0.0001	True	32	0.6701	0.5929	0.5850	0.5368

Tabla 3.19: Mejores modelos DenseNet

Como podemos observar en la tabla 3.19, los dos mejores modelos corresponden a la versión con y sin aumento de datos de la arquitectura de mayor profundidad DenseNet201, tras esta se encuentran de nuevo las versiones con y sin aumento de datos de la versión de menor profundidad DenseNet121, en cuanto al resto de hiperparámetros, se observa de nuevo la superioridad del optimizador Adam con learning rate de 0.0001.

En esta ocasión el primer modelo ha obtenido una F1-score de 0.5636, bastante superior a la obtenida por el segundo modelo (0.5422), y una accuracy de 0.6447, ligeramente inferior a la del segundo modelo.

Tras esta primera búsqueda pasamos de nuevo a realizar la validación cruzada sobre los tres mejores modelos para obtener una comparación más consistente.

Validación del primer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
DenseNet201	Adam	0.0001	False	256

Tabla 3.20: Primer modelo DenseNet

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6258	0.6235	0.5947	0.6027
1	0.645	0.6095	0.6047	0.6006
2	0.6300	0.598	0.578	0.5785
3	0.5837	0.5848	0.5458	0.5508
4	0.6422	0.6118	0.6099	0.6045
5	0.638	0.6368	0.6036	0.6088
6	0.6388	0.6183	0.5983	0.6019
7	0.6268	0.5953	0.5923	0.5854
Media	0.6287	0.6098	0.5909	0.5917
Media (%)	62.87%	60.98%	59.09%	59.17%
Desviación típica	0.0195	0.0168	0.0206	0.0194

Tabla 3.21: Validación cruzada primer modelo DenseNet

Como podemos observar en la tabla 3.21, este primer modelo obtiene una F1-score media de 0.5917 y una accuracy media de 0.6287, se puede observar algo de inconsistencia entre el entrenamiento con distintas carpetas, en especial con la tercera de ellas, lo que da lugar a una desviación típica algo elevada pero dentro de lo razonable.

Validación del segundo modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
DenseNet210	Adam	0.0001	True	256

Tabla 3.22: Segundo modelo DenseNet

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6361	0.5902	0.5884	0.5805
1	0.6436	0.6316	0.6072	0.6088
2	0.6088	0.5836	0.5573	0.5613
3	0.6367	0.6245	0.6006	0.6019
4	0.64	0.6236	0.606	0.6080
5	0.6361	0.5902	0.5884	0.5805
6	0.6157	0.5954	0.5343	0.5483
7	0.6544	0.6403	0.5993	0.6046
Media	0.6348	0.6130	0.5856	0.5884
Media (%)	63.48%	61.30%	58.56%	58.84%
Desviación típica	0.0152	0.0208	0.0261	0.0229

Tabla 3.23: Validación cruzada segundo modelo DenseNet

En este caso, como podemos ver en la tabla 3.23 obtenemos una F1-score media de 0.5884, algo inferior a la obtenida en el primer modelo. Sin embargo a pesar de ello, de nuevo la accuracy es superior en este segundo modelo (0.6348 frente a 0.6287). También observamos un aumento en las desviaciones típicas de precisión, recall y F1-score debido en gran parte a su mal rendimiento con la segunda y sexta carpeta, por lo que este modelo es algo más inconsistente que el primero.

Validación del tercer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
DenseNet121	Adam	0.0001	True	256

Tabla 3.24: Tercer modelo DenseNet

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6074	0.5794	0.5519	0.5502
1	0.6294	0.6231	0.5712	0.5793
2	0.6303	0.585	0.5521	0.553
3	0.6177	0.6007	0.5611	0.5681
4	0.6219	0.6136	0.5664	0.5761
5	0.6154	0.6151	0.5648	0.5778
6	0.6377	0.633	0.5858	0.5947
7	0.604	0.5523	0.5519	0.5451
Media	0.6205	0.6003	0.5631	0.5680
Media (%)	62.05%	60.03%	56.31%	56.80%
Desviación típica	0.0117	0.0266	0.0118	0.0172

Tabla 3.25: Validación cruzada tercer modelo DenseNet

Como observamos en la tabla 3.25, en este tercer modelo, aunque algo más consistentes, obtenemos las peores puntuaciones (F1-score de 0.5680 y accuracy de 0.6205).

En este caso el mejor modelo obtenido en la búsqueda es consistente con el mejor obtenido tras la validación cruzada, siendo este la versión con mayor profundidad. Pese a esto, cabe destacar que estos tres modelos han obtenido mayor *F1-score* en la validación cruzada que en la búsqueda, por lo que podemos deducir que en este caso se ha seleccionado una partición del conjunto de datos que no beneficiaba a esta arquitectura en particular.

Mejor modelo obtenido:

Versión	Optimizador	LR	Aumento datos	Batch size
DenseNet201	Adam	0.0001	False	256

Tabla 3.26: Mejor modelo DenseNet

Iteración	Accuracy	Precisión	Recall	F1
0	0.6452	0.6252	0.6454	0.6264
1	0.641	0.6032	0.6383	0.6098
2	0.6489	0.619	0.6519	0.6241
3	0.6421	0.6142	0.644	0.6185
4	0.5879	0.5543	0.6006	0.5627
5	0.6562	0.6325	0.6578	0.6358
6	0.6441	0.6215	0.647	0.6213
7	0.6477	0.6082	0.6224	0.6026
8	0.6446	0.611	0.6323	0.6142
9	0.6321	0.6007	0.6267	0.6023
Media	0.6390	0.6090	0.6366	0.6118
Media (%)	63.90%	60.90%	63.66%	61.18%
Desviación típica	0.0190	0.0216	0.0168	0.0202

Tabla 3.27: Iteraciones modelo final DenseNet

En la tabla 3.19 se muestra el resultado de las 10 iteraciones para la validación incluyendo el conjunto de test. Como podemos ver, hemos obtenido una *F1-score* media de 0.6118 con una accuracy de 0.6390, resultados muy consistentes con los obtenidos durante la validación cruzada y los más prometedores teniendo en cuenta el resto de arquitecturas evaluadas.

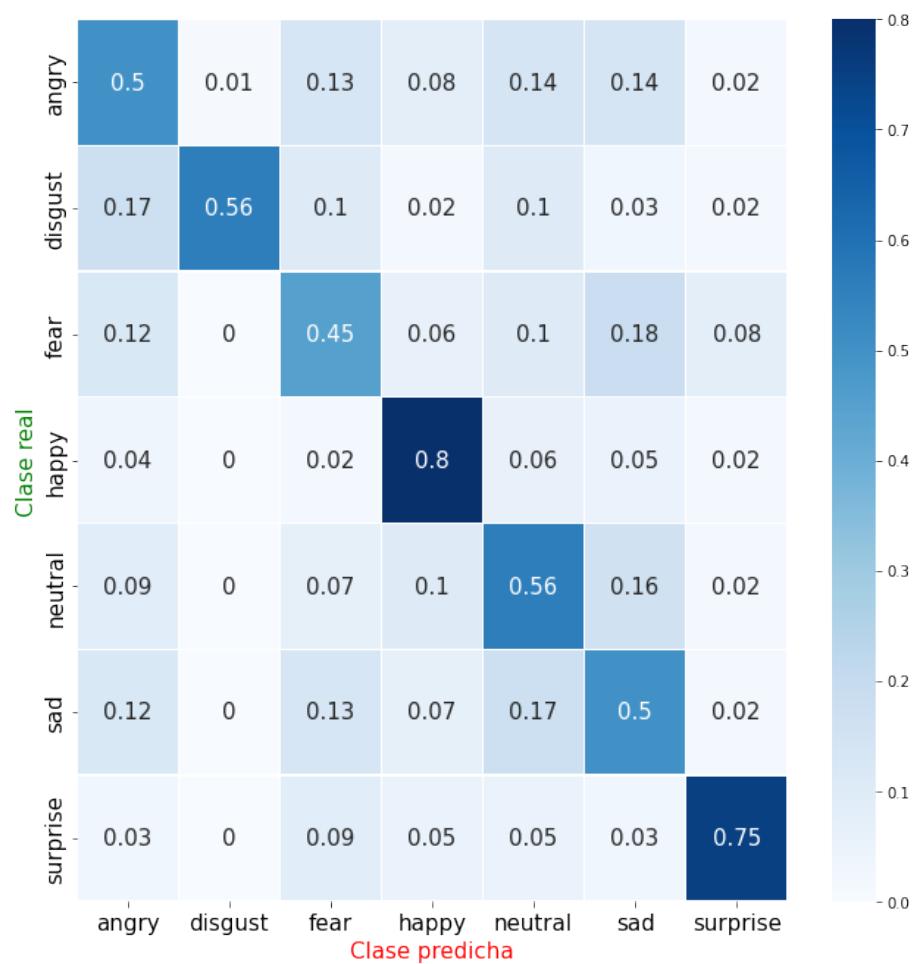


Figura 3.18: DenseNet matriz de confusión

En la figura 3.18 se muestra la matriz de confusión obtenida para el mejor modelo tras las 10 iteraciones finales. Si observamos la diagonal, podemos destacar una puntuación máxima de 0.8 para la clase "happy" y de 0.75 para la clase "surprise". El resto de puntuaciones oscilan entorno a una puntuación de 0.5, siendo de nuevo la más baja la obtenida para la clase "fear" con un 0.45.

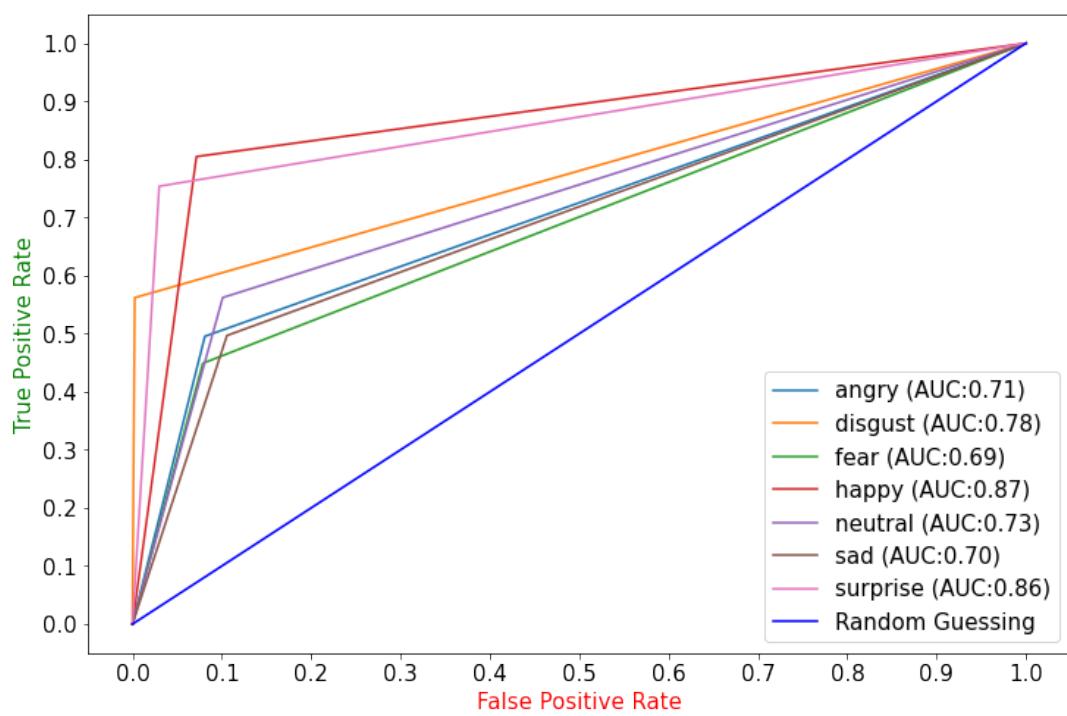


Figura 3.19: DenseNet curvas ROC

En la figura 3.19 se muestran las curvas ROC obtenidas para cada clase. Finalmente obtiene una puntuación AUC media de 0.7716, que de nuevo se encuentra en el rango de "discriminador aceptable" según la tabla 2.1.

Inception

Para la optimización de la arquitectura Inception valoraremos, además de los hiperparámetros comunes, las versiones **InceptionV3** e **InceptionResNetV2** de la arquitectura.

Versión	Optimizador	LR	Aumento datos	Batch size	Accuracy	Precisión	Recall	F1
InceptionV3	Adam	0.0001	True	256	0.6121	0.6022	0.5661	0.5707
IncepResnetV2	Adam	0.0001	True	256	0.6029	0.5561	0.5734	0.5521
IncepResnetV2	SGD	0.01	True	32	0.6019	0.5501	0.5699	0.5417
IncepResnetV2	Adam	0.0001	False	32	0.5885	0.5353	0.5401	0.5295
InceptionV3	Adam	0.0001	True	32	0.5701	0.5229	0.5309	0.5282

Tabla 3.28: Mejores modelos Inception

Como podemos observar en la tabla 3.28, obtenemos que el primer mejor modelo corresponde a la versión InceptionV3 con optimizador Adam, learning rate de 0.0001, aumento de datos y batch size de 256, obteniendo una F1-score de 0.5707 y una accuracy de 0.6121, seguido por la misma configuración pero en este caso para la versión InceptionV3.

Como peculiaridades, en este caso podemos volver a encontrar el optimizador SGD favorecido con un learning rate relativamente alto de 0.01 como tercer mejor modelo. También podemos observar que el aumento de datos parece favorecer en mayor medida a este modelo si tenemos en cuenta el resto de arquitecturas analizadas, pues 4 de los 5 mejores modelos cuentan con ella. También podemos encontrar que un batch size pequeño parece favorecer al modelo en casos concretos.

Tras esta primera búsqueda pasamos a realizar la validación cruzada de los tres mejores modelos obtenidos para asegurar la consistencia de los resultados.

Validación del primer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
InceptionV3	Adam	0.0001	True	256

Tabla 3.29: Primer modelo Inception

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6113	0.5973	0.5573	0.5665
1	0.6211	0.6054	0.5955	0.5905
2	0.6096	0.5746	0.5533	0.555
3	0.6102	0.6061	0.5715	0.5785
4	0.5578	0.5108	0.4961	0.4937
5	0.6076	0.5919	0.5539	0.5647
6	0.6185	0.6011	0.5747	0.5811
7	0.6243	0.5943	0.5733	0.5739
Media	0.6076	0.5852	0.5595	0.5630
Media (%)	60.76%	58.52%	55.95%	56.30%
Desviación típica	0.0210	0.0317	0.0292	0.0301

Tabla 3.30: Validación cruzada primer modelo Inception

En la tabla 3.30 mostramos los datos de la validación del primer mejor modelo obtenido. Este obtiene una puntuación F1-score de 0.5630 y una accuracy de 0.6076, resultados muy consistentes con los obtenidos durante la búsqueda, sin embargo, podemos observar una desviación típica considerable que en dos de las cuatro métricas evaluadas llega a superar 0.03. Esto se debe a que, como podemos observar, el modelo resulta muy inconsistente en función de la partición de los datos, en especial resalta lo perjudicado que se ve el modelo utilizando la cuarta partición, con la que obtiene una F1-score de 0.4937, muy alejada de la media de 0.56. Por tanto debemos tratar esta inconsistencia como un punto negativo del modelo a tener en cuenta.

Validación del segundo modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
InceptionResnetV2	Adam	0.0001	True	256

Tabla 3.31: Segundo modelo Inception

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6024	0.599	0.581	0.5826
1	0.6333	0.6218	0.6097	0.6107
2	0.6255	0.5985	0.5793	0.5759
3	0.6333	0.6338	0.608	0.6123
4	0.6261	0.6099	0.5864	0.5907
5	0.6293	0.5985	0.5855	0.5829
6	0.6076	0.5855	0.5679	0.5686
7	0.6157	0.5773	0.5652	0.5591
Media	0.6217	0.6030	0.5854	0.5853
Media (%)	62.17%	60.30%	58.54%	58.53%
Desviación típica	0.0118	0.0184	0.0164	0.0188

Tabla 3.32: Validación cruzada segundo modelo Inception

En esta segunda validación, cuyos resultados se muestran en la tabla (3.34), el modelo obtiene una F1-score de 0.5853 y una accuracy de 0.6217, puntuaciones bastante superiores a las obtenidas en el primer modelo con, además, una desviación típica mucho más razonable por debajo del 0.02 en todas las métricas, lo que lo convierte en un modelo mucho más consistente y un mejor candidato como representante óptimo de la arquitectura aplicada al reconocimiento de emociones.

Validación del tercer modelo:

Versión	Optimizador	LR	Aumento datos	Batch size
InceptionResNetV2	SGD	0.01	True	32

Tabla 3.33: Tercer modelo Inception

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6055	0.5441	0.5327	0.5147
1	0.6119	0.5344	0.5322	0.5069
2	0.6297	0.5602	0.5514	0.5313
3	0.6358	0.5603	0.5639	0.5378
4	0.623	0.5433	0.5405	0.5179
5	0.1385	0.0199	0.1416	0.0342
6	0.1385	0.0196	0.1378	0.0337
7	0.1382	0.0196	0.1403	0.0335
Media	0.4401	0.3502	0.3925	0.3388
Media (%)	44.01%	35.02%	39.25%	33.88%
Desviación típica	0.2500	0.2738	0.2095	0.2527

Tabla 3.34: Validación cruzada tercer modelo InceptionV3

En esta tercera validación, cuyos resultados se muestran en la tabla 3.33, obtenemos una puntuación F1-score de 0.3388 y una accuracy de 0.4401, siendo este el peor de los 3 modelos con diferencia. Si observamos los resultados carpeta por carpeta, podemos ver que el modelo obtiene resultados relativamente buenos y cercanos a los obtenidos en la búsqueda en las carpetas de la 0 a la 4. Sin embargo, estos resultados empeoran drásticamente en las carpetas de la 5 a la 7, en las que la accuracy de 0.13 nos da a entender que el modelo se limita a clasificar aleatoriamente las imágenes. Esto pone de manifiesto la baja consistencia que proporciona el patrón formado por el optimizador SGD con learning rate de 0.01 para este modelo y conjunto de datos concretos.

Mejor modelo obtenido:

Versión	Optimizador	LR	Aumento datos	Batch size
InceptionResnetV2	Adam	0.0001	True	256

Tabla 3.35: Mejor modelo Inception

En esta ocasión, de nuevo el mejor modelo obtenido durante la búsqueda no era realmente el más adecuado. Esto es debido, como ya ha sido comentado, a la naturaleza inconsistente de la validación holdout con una única carpeta, que en este caso concreto perjudicó al verdadero mejor modelo (el segundo mejor en la búsqueda), que obtuvo unas mejores y más consistentes métricas durante el proceso de validación cruzada.

Iteración	Accuracy	Precisión	Recall	F1
0	0.6422	0.6124	0.6326	0.6126
1	0.6364	0.6111	0.6173	0.6058
2	0.6468	0.6203	0.6267	0.6157
3	0.6491	0.6212	0.6538	0.6238
4	0.6328	0.6219	0.646	0.6207
5	0.6211	0.606	0.6409	0.6119
6	0.633	0.6075	0.6316	0.6064
7	0.5972	0.5716	0.5942	0.5694
8	0.6369	0.6034	0.6288	0.6026
9	0.6368	0.6169	0.6246	0.6141
Media	0.6332	0.6092	0.6296	0.6083
Media (%)	63.32%	60.92%	62.96%	60.83%
Desviación típica	0.0149	0.0148	0.0164	0.0152

Tabla 3.36: Iteraciones modelo final Inception

En la tabla 3.36 se muestran las 10 iteraciones del entrenamiento del modelo utilizando los conjuntos finales de entrenamiento y test, en este caso observamos una gran mejora con respecto a los resultados obtenidos durante la búsqueda y validación cruzada, obteniendo este una F1-score y accuracy medias de 0.6083 y 0.6332 respectivamente. Esta mejora puede deberse al beneficio que obtiene este modelo en concreto de la incorporación de un mayor

número de imágenes al conjunto de entrenamiento.

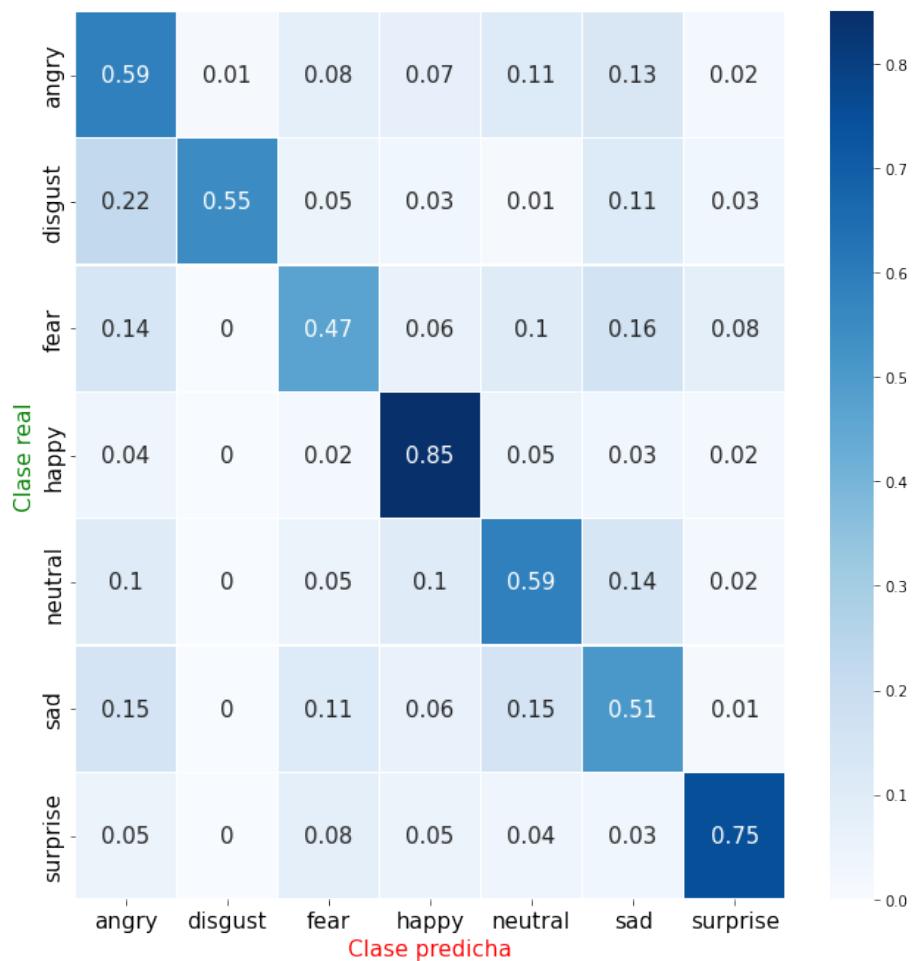


Figura 3.20: Inception matriz de confusión

En la figura 3.20 se muestra la matriz de confusión obtenida para el mejor modelo obtenido tras las 10 iteraciones finales de la arquitectura (fila resaltada en amarillo en tabla 3.36). Cabe resaltar la puntuación de 0.47 obtenida para la clase "fear" que si bien continúa siendo baja, se trata del mejor valor obtenido hasta el momento para esta clase. El resto de valores resultan muy similares a los obtenidos para DenseNet.

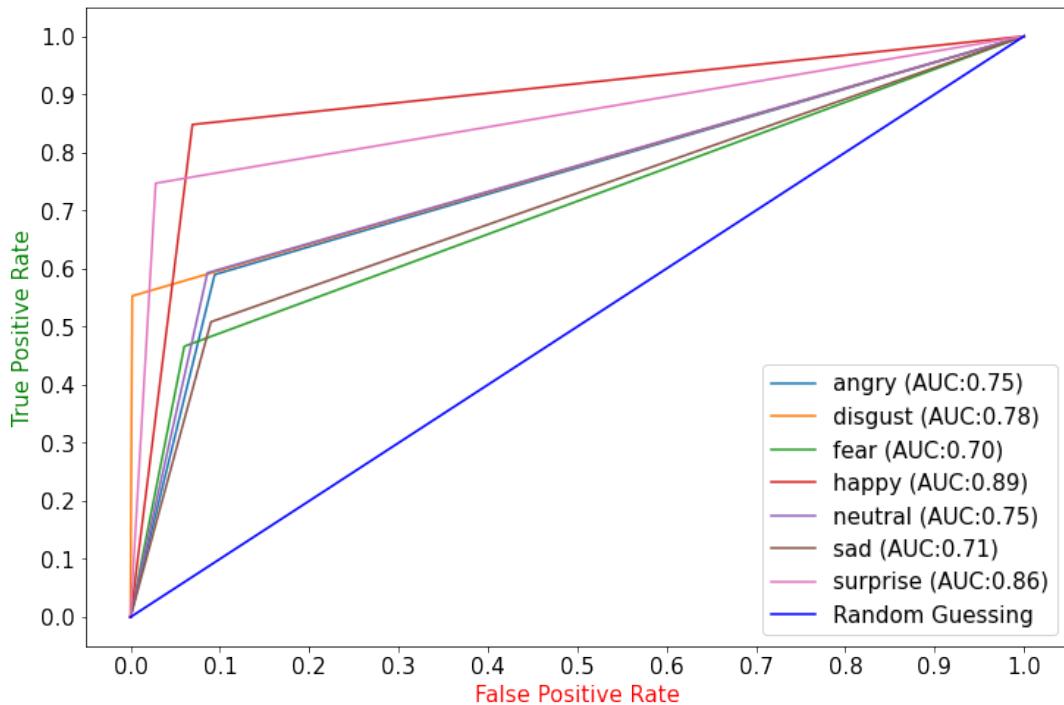


Figura 3.21: Inception curvas ROC

En cuanto a las curvas ROC (figura 3.21), no encontramos ninguna singularidad a resaltar, siendo la puntuación AUC media obtenida de 0.7911, lo que la clasifica según los rangos mostrados en la tabla 2.1 como un discriminador aceptable.

3.2.3. Comparativa

Versión	Optimizador	LR	Aumento datos	Batch size	Accuracy	Precisión	Recall	F1
DenseNet201	Adam	0.0001	False	256	0.6390	0.6090	0.6366	0.6118
IncepResnetV2	Adam	0.0001	True	256	0.6332	0.6092	0.6296	0.6083
VGG16	Adam	0.0001	False	256	0.6202	0.5843	0.6168	0.5889
ResNet152V2	Adam	0.0001	False	256	0.6004	0.5695	0.5967	0.5711

Tabla 3.37: Mejores modelos finales

En la tabla 3.39 mostramos los mejores modelos obtenidos ordenados de mayor a menor puntuación obtenida para la métrica *F1-score* en su versión final. Las puntuaciones corresponden a la media de las obtenidas para el modelo final entrenado con el 80% de la base de datos completa en 10 iteraciones. Como podemos observar, el mejor modelo corresponde a la arquitectura DenseNet201, con una *F1-score* de 0.6118, seguida por InceptionResNetV2, que obtuvo una puntuación *F1-score* ligeramente peor (0.6083). El tercer mejor modelo se trata de la arquitectura VGG16, que a pesar de su simplicidad y poca profundidad en comparación con el resto de arquitecturas, obtuvo una puntuación *F1-score* de 0.5889. Finalmente el peor modelo se trata de la optimización de la arquitectura ResNet, concretamente ResNet152V2, que obtuvo una puntuación de 0.5711.

En cuanto al resto de métricas, no se aprecia ningún desequilibrio destacable entre la precision y el recall, siendo estas métricas, como cabe esperar, muy proporcionales a la *F1-score* obtenida, en cuanto a accuracy, podemos observar que decrece proporcionalmente a la puntuación *F1-score* obtenida por los modelos.

Todas las arquitecturas analizadas parecen beneficiarse del patrón formado por el optimizador Adam con learning rate de 0.0001 y batch-size de 256. En cuanto al aumento de datos, parece haber beneficiado únicamente al modelo Inception.

En las figuras 3.22, 3.23, 3.24, 3.25 se muestran las curvas de aprendizaje de la mejor iteración de cada uno de los modelos.

Estas curvas están representadas en forma de gráficas en las que el eje horizontal representa los epochs de entrenamiento, mientras que el eje vertical representa la *F1-score* obtenida. En dichas gráficas, la línea naranja muestra cómo aumenta la puntuación *F1-score* obtenida sobre el conjunto de entrenamiento mientras que la azul muestra la obtenida sobre el conjunto de validación.

A primera vista podemos observar que cada una de las redes alcanzó su valor óptimo tras ejecutar un número de epochs distinto. Así, podemos observar que mientras que la red VGG alcanzó su valor óptimo tras un número relativamente reducido de 12 epochs (3.22), la red Inception lo alcanzó tras ejecutar 28 (3.25). En cuanto a Resnet y DenseNet, ejecutaron, respectivamente, 18 y 20 epochs (figuras 3.23 y 3.24).

Como podemos observar, todas comparten un patrón común: ambas curvas cuentan

con tendencia creciente, pero mientras que la curva de aprendizaje sobre los datos de entrenamiento (línea azul) crece hasta alcanzar puntuaciones muy cercanas al 1 (puntuación perfecta), la curva de validación presenta un crecimiento cada vez menos pronunciado. Esto es debido a que mientras que la curva de aprendizaje sobre el conjunto de entrenamiento muestra la puntuación obtenida sobre los mismos datos con los que se entrenó a la red, la curva sobre el conjunto de validación lo hace sobre datos que nunca se utilizaron para el entrenamiento. Cuanto más lejanas se encuentren estas dos curvas, mayor tendencia tendrá el modelo a sobreajustarse a los datos de entrenamiento, y menor capacidad tendrá de obtener buenos resultados cuando se utilice para la inferencia de nuevos datos.

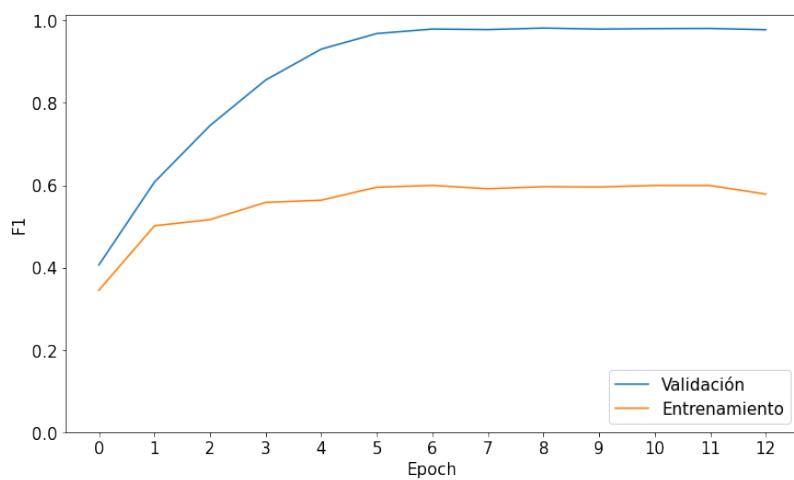


Figura 3.22: Curvas F1-score VGG

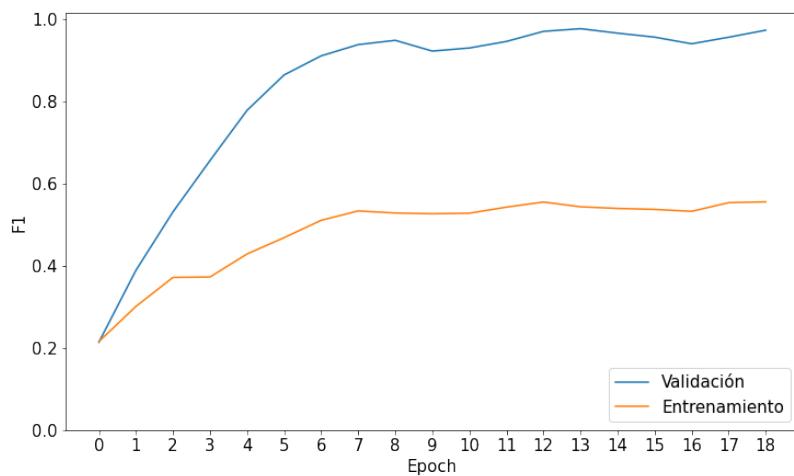


Figura 3.23: Curvas F1-score ResNet

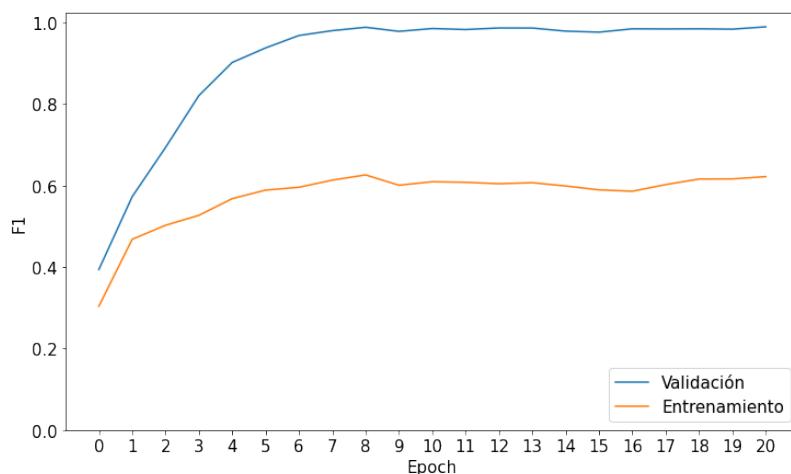


Figura 3.24: Curvas F1-score DenseNet

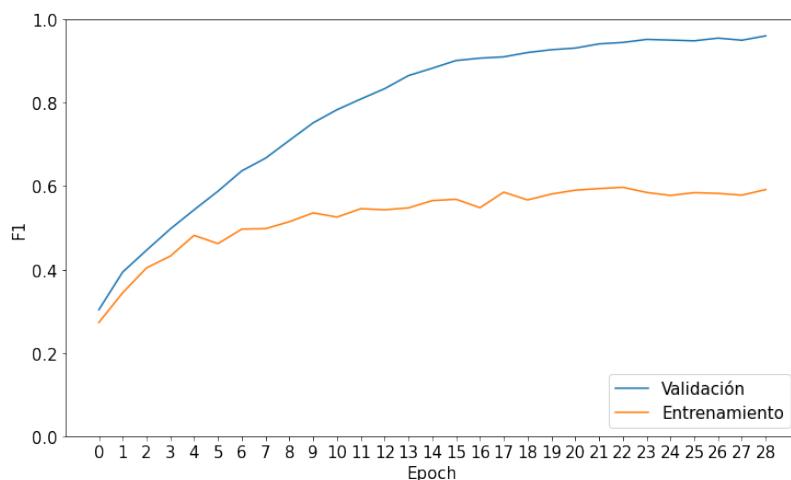


Figura 3.25: Curvas F1-score Inception

Conclusiones

Tras la realización del estudio podemos extraer las siguientes conclusiones del mismo:

- Las arquitecturas que mejores resultados obtuvieron se trata de arquitecturas relativamente profundas con capas residuales, siendo DenseNet201 la arquitectura que mejores resultados obtuvo.
- Los modelos parecen beneficiarse de la reducción de un learning rate relativamente pequeño de 0.0001. Además del uso del optimizador Adam frente al SGD, que como

comentamos en D.2, aplica un learning rate adaptativo al entrenamiento, por lo que como posible mejora sería interesante estudiar el uso de learning rates todavía más bajos con distintas técnicas de reducción del mismo.

- El aumento de datos aplicado no parece aumentar la *F1-score* obtenida en ninguno de los casos, a excepción de la arquitectura Inception-ResNetV2.
- Todos los modelos se han visto beneficiados por el mayor batch-size evaluado durante la búsqueda (256), lo que nos lleva a deducir que los modelos podrían beneficiarse de batch-sizes todavía más grandes.
- Observando las curvas de aprendizaje, todos los modelos muestran sobreajuste, por lo que ciertas técnicas para la reducción del mismo podrían beneficiar dichos modelos.

3.3. Arquitectura Propia

Según hemos observado, los modelos preentrenados analizados son capaces de devolver buenos resultados en el reconocimiento de emociones con FER2013, es por ello que a la hora de diseñar una arquitectura que mejore a las anteriores, lo más eficiente es crear versiones optimizadas de las mismas para el problema en concreto.

3.3.1. Modificación de VGG16

Para esta primera iteración se experimentará con la modificación de la red **VGG16**, esto es debido a que, a pesar de no ser el mejor modelo obtenido como conclusión del estudio, sí que obtuvo unos buenos resultados considerando su simpleza en comparación con el resto de modelos. Esta simplicidad es la que nos permitirá realizar pequeñas modificaciones en el cuerpo de la red para tratar de mejorar sus resultados.

La idea es tratar de combinar esta arquitectura con uno de los conceptos que sustentan la arquitectura **DenseNet**, puesto que fue esta la que obtuvo mejores resultados tras el estudio.

Como ya vimos en 2.2, durante el entrenamiento de un modelo de deep learning para visión artificial, los primeros bloques con capas convolucionales del modelo tienden a aprender filtros para la detección de formas simples como líneas rectas y curvas. Conforme la profundidad de la red aumenta, las capas tienden a detectar atributos cada vez más complejos, en nuestro caso se trataría de formas que correspondan a rasgos faciales más reconocibles como los ojos o la nariz. Estos rasgos, a pesar de no representar la imagen al completo, sí comienzan a arrojar cierta información acerca de la emoción expresada en la imagen.

La idea, por tanto, sería no solo tener en cuenta los filtros generados por el último bloque de la red, sino que por el contrario, aprovechar también los generados por algunos de los bloques anteriores.

Esta reutilización de los mapas de características puede lograrse, como ya introdujimos, agregando, al igual que se hizo en DenseNet, con capas residuales que conecten algunos de los últimos bloques de la red directamente con la capa final del cuerpo de la red.

En la figura 3.26 se muestra la arquitectura propuesta. Como podemos ver, la parte izquierda se trata del modelo VGG16 tal y como se entrenó durante el estudio, mientras que en la parte derecha podemos apreciar la modificación propuesta: esta consiste en tres conexiones residuales que conectan la salida de los tres bloques anteriores al bloque final. Cada una de estas conexiones está formada por la siguiente secuencia de capas:

- **Capas convolucionales** con filtros 1x1 para, al igual que se hizo en DenseNet, agregar un nivel más de profundidad y controlar el número parámetros. Esto es conveniente para evitar generar modelos demasiado pesados sin que sea estrictamente necesario para su rendimiento, ya que todos los pesos generados por las capas residuales se concatenarán al final.

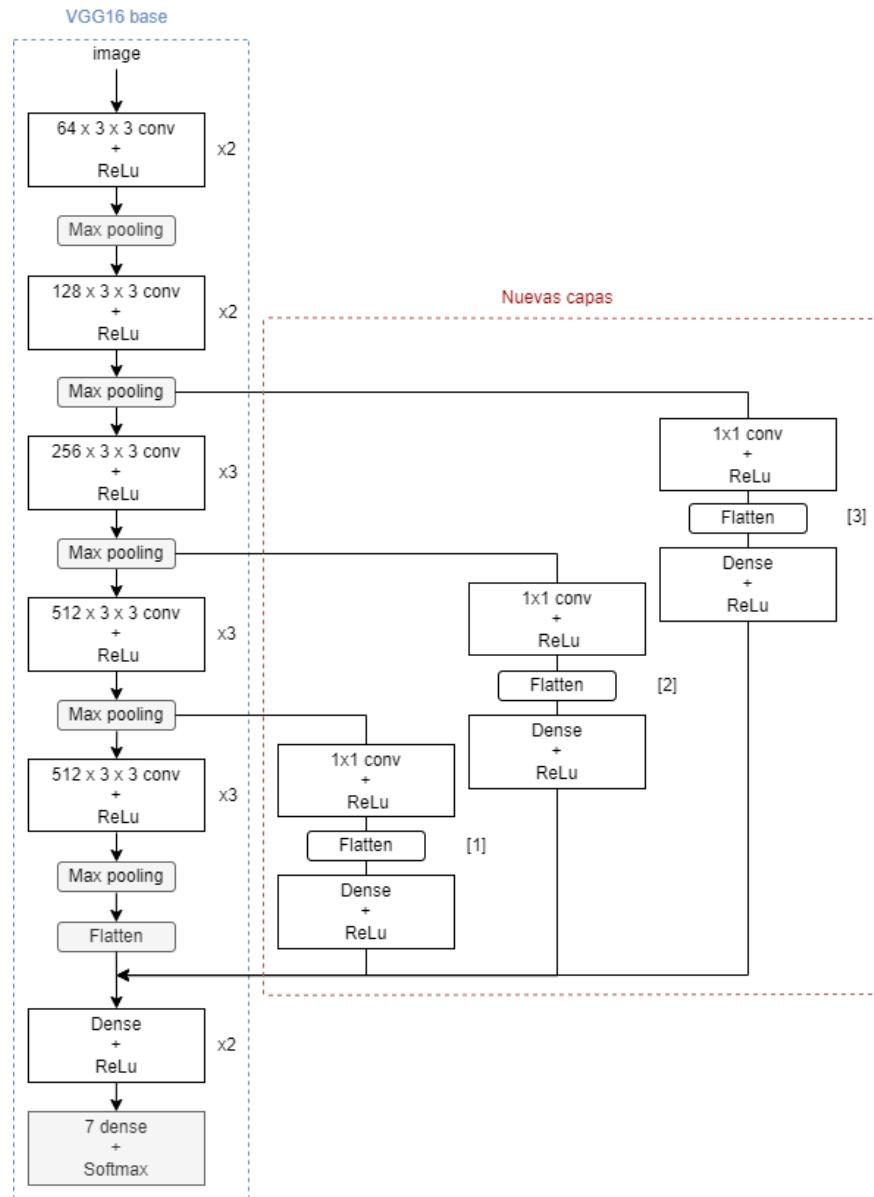


Figura 3.26: Arquitectura VGG16 con capas residuales

- **Capas flatten** para reducir la dimensionalidad y posteriormente sea posible concatenar los pesos al último tramo de la red.
- **Capas densas** para, junto con las capas convolucionales, dotar de profundidad y capacidad de aprendizaje a las conexiones residuales.

Búsqueda

Para la optimización de la arquitectura propuesta, se tendrán en cuenta los siguientes hiperparámetros:

-
- **Número de bloques residuales (BR)** a incluir en la arquitectura. Atendiendo a la figura 3.26, se valorará la inclusión de los bloques residuales 1, los bloques 1 y 2, los bloques 1, 2 y 3 o de ninguno de ellos.
 - **Número de filtros (NF)** generados por las capas convolucionales de tamaño 1x1 previas a los bloques residuales. En concreto se evaluará el rendimiento con 64, 128 y 256 filtros.
 - **Capas densas en la cabeza de los bloques (CDC)**. Como vimos en la figura 3.26, cada uno de los bloques residuales agregados culmina con una serie de capas densas para dotar a dichos bloques de capacidad de aprendizaje adicional. Para esta parte final valoraremos la inclusión de entre 1 y 2 capas de 128, 256, 512 y 1024 neuronas (**CDCN**).
 - El espacio de búsqueda del **learning rate (LR)** variará con respecto al utilizado durante el estudio. En éste, se observó que en todos los casos el optimizador Adam se veía favorecido por el learning rate más bajo a evaluar (0.0001), por lo que en este caso se incluirán en el espacio de búsqueda los learning rates 0.0001, 0.00001, 0.000001.
 - De nuevo se evaluará la inclusión o no del **aumento de datos (AD)** descrito en 3.3.
 - **Capas densas finales (CDF)** de la red. En este caso, puesto que el número de pesos recibidos será mayor que los de la arquitectura original, se valorará la inclusión de 0 a 4 capas densas con 512, 1024, 2048 o 4096 neuronas (**CDFN**),
 - En cuanto al **optimizador**, según lo que dedujimos del estudio de arquitecturas existentes, únicamente valoraremos el desempeño del modelo con el optimizador *Adam*.

En la figura 3.38 se muestran los resultados obtenidos tras ejecutar la búsqueda mediante el algoritmo de optimización bayesiana.

BR	LR	AD	BS	FDL	FDU	BDL	BDU	NF	Acc	P	R	F1
0	0.0001	False	256	0					0.6390	0.6370	0.6079	0.6098
0	0.0001	True	256	0					0.6330	0.6354	0.6064	0.6077
0	0.0001	False	256	4	4096				0.6314	0.6140	0.6082	0.6022
1	0.0001	True	256	1	512	1	128	64	0.5893	0.5863	0.5340	0.5521
2	0.0001	False	256	1	512	1	128	64	0.5740	0.5710	0.5221	0.5397

Tabla 3.38: Mejores modelos VGG con bloques residuales

Si observamos con detenimiento la tabla 3.38, podemos observar que el mejor modelo corresponde a la arquitectura VGG base sin la adición de ningún bloque residual, con learning rate de 0.0001, sin aumento de datos y sin ninguna capa densa a la cabeza de la red. Esta arquitectura logra un 0.6098 de F1-score

En cuanto a las siguientes dos mejores configuraciones, se trata de nuevo de la arquitectura sin bloques residuales, pero en este caso variando el aumento de datos en un caso, y el

número de capas densas en otro. Estos tres mejores modelos obtienen unas métricas muy similares entre ellas. En cuanto al cuarto y quinto mejor modelo, se trata de la arquitectura con uno y dos bloques residuales respectivamente, donde se observa un decremento considerable de las métricas con respecto a los modelos sin capas residuales. En ambos casos el modelo parece beneficiarse de una capa densa final de 512 neuronas, una capa densa al final de cada bloque de 128 neuronas, y una capa convolucional al inicio de cada bloque residual de 64 filtros 1x1. Estas dos últimas configuraciones obtienen una *F1-score* de 0.55, considerablemente peor a la obtenida por los modelos sin bloques residuales.

Esta búsqueda nos arroja una conclusión clara: en ningún caso el modelo se beneficia de los bloques residuales añadidos. Podemos hipotetizar que en este caso los bloques residuales perjudicarían uno de los puntos clave del buen rendimiento de estas arquitecturas: el preentrenamiento con Imagenet, que dota a los primeros bloques de la red de una gran capacidad para el reconocimiento de formas simples al mismo tiempo que los últimos bloques tratan de refinarse y adaptarse a la base de datos FER2013.

Tras esta primera búsqueda, llegamos a la siguiente conclusión: puesto que la elección del modelo VGG16 como base para la modificación se debió en gran parte a la sencillez para modificar el cuerpo de la arquitectura, y éste ha resultado ineficaz, lo más sensato es descartar esta iteración para la búsqueda de un modelo propio para tratar de encontrar una versión óptima de la mejor arquitectura obtenida en 3.3.8, siendo esta la arquitectura **DenseNet201**.

3.3.2. Optimización de DenseNet201

Para la realización de esta optimización, tendremos en cuenta los siguientes factores, extraídos de las conclusiones tras todo el estudio previo.

- En primer lugar, y como que ya se ha comentado, durante el estudio previo de las arquitecturas, cuyo resultado y conclusiones se mostraron en el apartado 3.2.3, se observó que la red DenseNet201 fue la que mejor se adaptaba al problema, por lo que lo lógico será realizar la optimización sobre el cuerpo de esta arquitectura.
- Por otra parte, tras la realización de la primera iteración de la búsqueda del modelo propio en 3.3.1, se observó que las modificaciones realizadas sobre el cuerpo de la red, lejos de beneficiar al modelo, lo perjudicaban, en parte por el beneficio que aportan al modelo los pesos preentrenados con Imagenet, por lo que en este caso decidimos mantener el cuerpo de la red lo más intacto posible.
- Por último, se ha de tener en cuenta que todas las arquitecturas existentes bajo estudio fueron diseñadas y entrenadas sobre el conjunto de datos Imagenet, que como se nombró en el apartado 2.3, cuentan con una resolución de 224x224, y puesto que las imágenes de FER2013 son de resolución 48x48, el uso de técnicas para el aumento de dicha resolución podría resultar beneficioso para el modelo.

Redimensionamiento

Existen varios métodos de redimensionamiento de imágenes en base a qué método de interpolación se utilice. Concretamente la biblioteca de Keras nos ofrece los métodos de interpolación bilineal, bicúbica, vecino más cercano, área, lanza3, lanza5, gausiana y mitchelcubic.

Para estimar cual de estos métodos es el más adecuado, al igual que en el caso de la elección de aumento de datos, podemos realizar una primera estimación visual del resultado de la aplicación de los métodos.

Como podemos ver en la figura 3.27, dependiendo del tipo de redimensionamiento utilizado, obtendremos distinto resultado. A simple vista, algunas técnicas como el redimensionamiento por vecinos más cercanos o por área, parecen obtener imágenes de peor calidad que el resto, que obtienen resultados muy similares entre ellos.

Optimización

En este caso, a diferencia de las búsquedas realizadas para las optimizaciones anteriores, no se realizará una búsqueda pura mediante el algoritmo de optimización bayesiana. Esto es debido a que, como mencionaremos más adelante, se realizará un aumento de las dimensiones de las imágenes de la base de datos. Esto incrementará de forma considerable tanto el tiempo de entrenamiento como las necesidades computacionales de la red, lo que hace inviable la idea de realizar un número considerable de entrenamientos. Por contra, se realizarán una serie de pruebas manuales para determinar hasta qué punto es posible aumentar la resolución de las imágenes para que un entrenamiento que el entrenamiento sea viable.

En primer lugar, utilizaremos como base la versión óptima de la arquitectura DenseNet201. Esta, como se muestra en la tabla de resultados finales de la comparativa (tabla ??), se trata de la versión DenseNet201, con optimizador *Adam*, learning rate de 0.0001, sin aumento de datos y con un batch size de 256. Sobre esta configuración, aplicaremos 4 redimensionamientos distintos, el cual se trata del redimensionamiento de tipo bilineal en cuatro resoluciones distintas: redimensionamiento a resolución 100x100, 125x125, 150x150 y por último 175. En este caso no se continuará incrementando la resolución resultante debido a limitaciones computacionales, pero se considera suficiente para comprobar la aportación de este tipo de redimensionamiento al preprocesamiento de las imágenes. Para esta primera búsqueda, de nuevo recurriremos a la técnica de validación holdout.

Tras este primer refinamiento, podemos observar que efectivamente el redimensionamiento parece afectar de manera muy positiva al modelo a las métricas del modelo, obteniéndose el mejor resultado para el mayor redimensionamiento evaluado (175x175), que obtiene un puntuación *F1-score* de 0.6243 y una accuracy de 0.6829. Estos resultados, a falta de evaluar la consistencia del modelo, resultan muy prometedores en comparación con los obtenidos en el resto de modelos. Una vez seleccionado el mejor redimensionamiento de entre los 4, podemos pasar a evaluarlo de la misma manera que se procedió durante el estudio de arquitecturas existentes: una validación cruzada con 8 carpetas seguido de 10 iteraciones incorporando en este caso el conjunto de test.



Figura 3.27: Tipos de redimensionamiento

Redimensionamiento	Accuracy	Precisión	Recall	F1
175	0.6829	0.6470	0.6242	0.6243
150	0.6400	0.6025	0.5752	0.5756
125	0.6707	0.6423	0.6125	0.6141
100	0.6322	0.6106	0.5710	0.5754

Tabla 3.39: Mejores modelos finales

En la tabla 3.40 se muestran los resultados de la validación cruzada. Podemos observar

que se han obtenido valores medios muy similares a los obtenidos en la validación holdout ($F1$ -score media de 0.6255 frente a 0.6243, y accuracy media de 0.6720 frente a 0.6829). Esto, junto con el hecho de haber obtenido una desviación típica dentro de lo razonable, es indicativo de una buena consistencia por parte del modelo.

Carpeta validación	Accuracy	Precisión	Recall	F1
0	0.6643	0.6649	0.6236	0.6288
1	0.6712	0.6418	0.6175	0.6199
2	0.6760	0.6301	0.5979	0.6014
3	0.6656	0.6595	0.6330	0.634
4	0.6715	0.6352	0.6276	0.6267
5	0.6870	0.6630	0.6387	0.6421
6	0.6577	0.6463	0.6073	0.6154
7	0.6826	0.6620	0.6324	0.6358
Media	0.6720	0.6223	0.6503	0.6255
Media (%)	67.20%	62.23%	65.03%	62.55%
Desviación típica	0.0097	0.0139	0.0137	0.0130

Tabla 3.40: Validación cruzada DenseNet optimizado

En la tabla 3.41 se muestran las 10 iteraciones del modelo entrenado sobre el conjunto de datos con el conjunto de test incluido. En este caso, se ha obtenido una $F1$ -score media de 0.6645 y una accuracy de 0.6935 con una desviación típica dentro de lo razonable. Estas métricas son superiores a las obtenidas tanto en la validación holdout como cruzada, por lo que la incorporación de nuevas imágenes tanto al modelo como al conjunto de validación parece afectar muy positivamente al desempeño del modelo. Estas métricas también resultan superiores a las obtenidas por el mejor de los modelos obtenidos durante el estudio de arquitecturas existentes, el cual correspondía a esta misma arquitectura (DenseNet201) sin la aplicación del redimensionamiento, que como vimos, obtuvo una $F1$ -score de 0.6118 y una accuracy de 0.6390.

Finalmente, mostramos tanto la matriz de confusión como las curvas ROC del mejor modelo obtenido en la tabla de resultados 3.41.

En la figura 3.28 se muestra la matriz de confusión obtenida para el modelo DenseNet optimizado, en ella podemos observar una mejora en todos los valores obtenidos con respecto a la versión DenseNet sin redimensionado. Destacamos un valor máximo de 0.87 para la clase "happy" y un valor mínimo de 0.51 para la clase "fear".

En la figura 3.29 mostramos las curvas ROC obtenidas, el valor AUC obtenido es de

Iteración	Accuracy	Precisión	Recall	F1
0	0.6851	0.6699	0.6513	0.6551
1	0.6948	0.6976	0.6472	0.6582
2	0.6989	0.709	0.6629	0.6746
3	0.7003	0.702	0.6668	0.6723
4	0.7005	0.7023	0.6677	0.6750
5	0.6886	0.6891	0.6661	0.6654
6	0.6936	0.6683	0.6669	0.6676
7	0.6803	0.6681	0.644	0.646
8	0.697	0.6896	0.6595	0.6645
9	0.6955	0.6868	0.6627	0.6667
Media	0.6935	0.6883	0.6595	0.6645
Media (%)	69.35%	68.83%	65.95%	66.45%
Desviación típica	0.0068	0.0160	0.0115	0.0113

Tabla 3.41: Iteraciones modelo final DenseNet optimizado

0.8156, el más alto obtenido durante el estudio, que según la tabla de rangos 2.1 posiciona al modelo como un discriminador excelente.

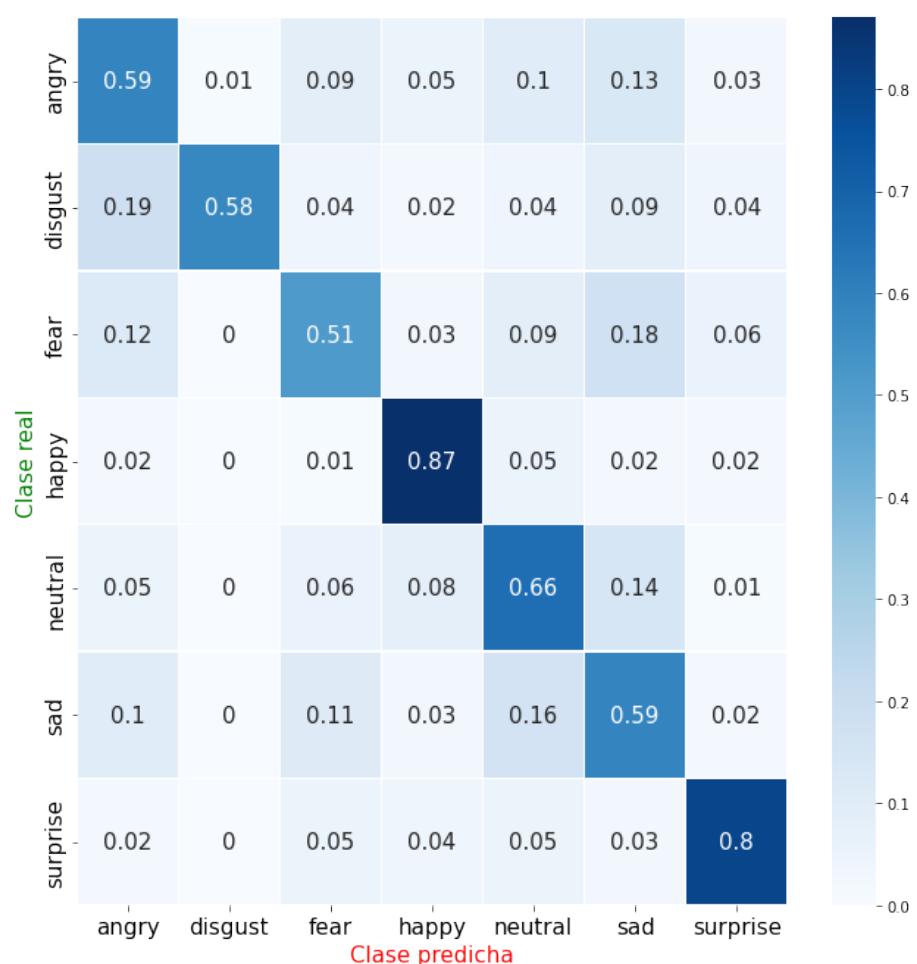


Figura 3.28: DenseNet optimizado matriz de confusión

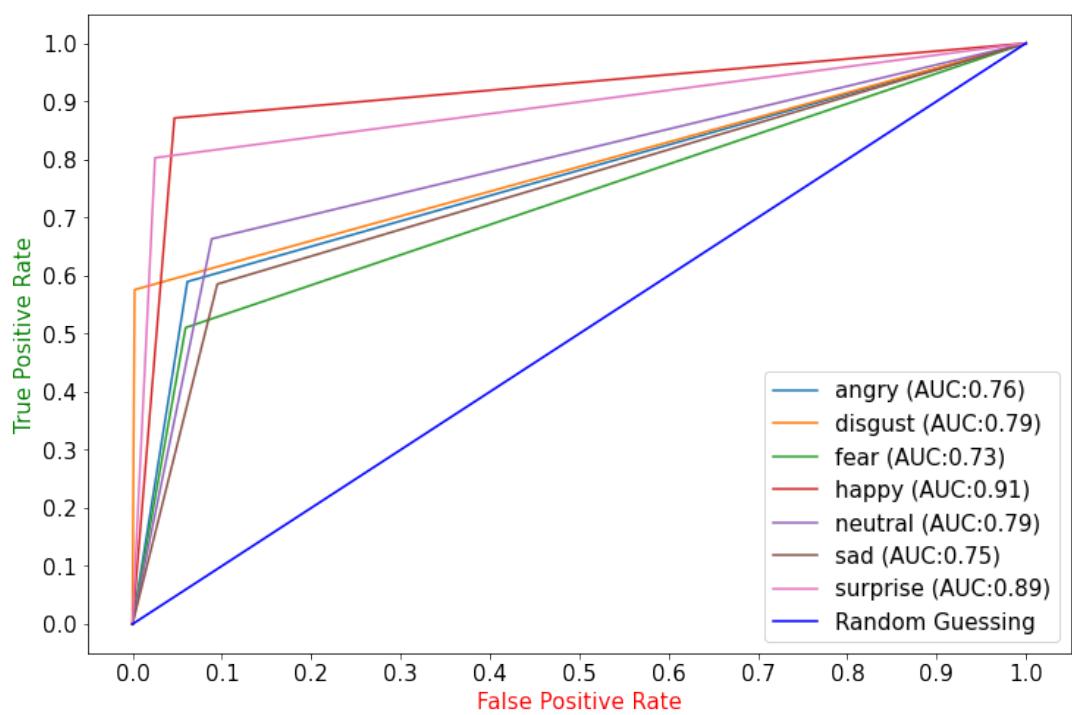


Figura 3.29: DenseNet optimizado curvas ROC

4. Conclusiones y trabajos futuros

Tras la elaboración de este proyecto, podemos destacar las siguientes conclusiones:

En primer lugar, hemos aprendido los conceptos fundamentales tras las redes neuronales y el *deep learning*. Tras el estudio y diseño de varias arquitecturas, podemos concluir que a la hora de diseñar arquitecturas para este tipo de modelos, la metodología basada en prueba y error resulta la más acertada, puesto que, a diferencia de otros modelos de machine learning, en este caso resulta imposible prever los resultados del mismo antes de ejecutarlo sobre la base de datos concreta. Esta poca intuitividad a la hora de diseñar las arquitecturas es la que pone de manifiesto la relevancia que tienen los algoritmos de búsqueda en el proceso de optimización de las redes. En nuestro caso, el uso de un algoritmo de búsqueda informada como el algoritmo de optimización bayesiana nos permitió encontrar las configuraciones óptimas para los hiperparámetros basándose únicamente en los resultados de las iteraciones anteriores.

A pesar de esto, cabe mencionar que un conocimiento sobre los fundamentos de estos modelos resulta útil a la hora de tanto acotar el espacio de búsqueda de hiperparámetros a aquellos realmente relevantes como de extraer conclusiones útiles sobre los mejores modelos obtenidos.

En segundo lugar, hemos aprendido los fundamentos tras las arquitecturas VGG, ResNet, DenseNet e Inception en sus distintas versiones, optimizándolas para obtener el mejor modelo posible para FER2013. Durante la validación de dichos modelos, se observó la importancia de utilizar métodos de validación que aseguren la consistencia del modelo, introduciendo conceptos como la validación holdout y cruzada, y los conjuntos de entrenamiento, validación y test.

En cuanto a evaluación de los modelos, hemos aprendido a interpretar las distintas métricas como la accuracy y la *F1-score*, y a seleccionar cada una de ellas en base a la naturaleza tanto del problema a resolver como de la base de datos que se utilizará para ello. En nuestro caso, puesto que contábamos con una base de datos desbalanceada, decidimos optar por la optimización de la métrica de *F1-score*. Este punto es una diferencia clave en cuanto a otros

estudios realizados, en los que se utiliza la accuracy como métrica base para tanto la evaluación como la optimización de los mismos, obteniendo resultados algo menos honestos.

Finalmente, tras tratar de diseñar un modelo propio para este problema, llegamos a la conclusión de que a la hora de tratar de aplicar uno de estos modelos a la resolución de un caso real, vale la pena valorar la utilización de métodos de transfer learning para tratar de reutilizar las arquitecturas existentes antes de diseñar una arquitectura desde cero, no solo por que dichas arquitecturas cuentan con un estudio y fundamento previos que sustentan su desempeño, sino también por la aprovechabilidad del conocimiento previo de estas redes en su versión preentrenada. Estas arquitecturas pueden ser una base sólida sobre la que aplicar mejoras, ya sea en forma de pequeñas modificaciones o, como en este caso, adecuar el preprocesamiento de los datos.

Fruto de todo el estudio previo, hemos logrado optimizar la arquitectura DenseNet201 hasta obtener unas métricas finales de 69.35% de *accuracy*, 68.83% de precisión, 65.95% de *recall* y 66.45% de *F1-score*.

Como idea para un posible **trabajo futuro**, se podría proponer la implementación de nuestra arquitectura final entrenada en una aplicación para el reconocimiento y análisis de emociones en vídeo. Dicha aplicación podría ser capaz de detectar las distintas emociones expresadas por las personas que aparezcan en el vídeo y utilizar dicha información para generar informes para su posterior análisis. Esto podría resultar útil por parte de ciertos locales de ocio como restaurantes o cines, los cuales podrían implementar cámaras para recoger información acerca de las emociones expresadas por los clientes que prueben el producto y que hayan dado su previo consentimiento, para posteriormente tratar de utilizar los datos para generar un análisis de, entre otras cosas, la aceptación o rechazo ante el producto.

Todo el código utilizado junto con los resultados obtenidos en formato de jupyter notebooks se encuentra adjuntado en el soporte digital.

A. Anexo 1: Algoritmos de búsqueda

A.1. Optimización bayesiana

El algoritmo de búsqueda bayesiana para la optimización de hiperparámetros de un modelo [Frazier, 2018] es una alternativa a otros algoritmos de búsqueda tradicionales como búsqueda en rejilla o búsqueda aleatoria.

Este algoritmo, a diferencia de los otros, realiza una **búsqueda informada** a la hora de optimizar los hiperparámetros, es decir, prioriza la búsqueda con los hiperparámetros que mejores resultados han devuelto en búsquedas pasadas para llegar a la configuración óptima de manera mas rápida.

El funcionamiento del algoritmo es el siguiente:

En primer lugar deberemos definir la **función objetivo**, que es la función que relaciona las distintas configuraciones de los hiperparámetros con el error obtenido de la red (ya sea el error cuadrático medio, la entropía cruzada, etc.). En esta función, a diferencia de la función de pérdida (introducida en el apartado 2.1.3), no conocemos las distribuciones de la función de error, por lo que no podemos recurrir al método del descenso del gradiente para encontrar el mínimo global.

Como alternativa al descenso del gradiente, el algoritmo de optimización bayesiana crea un modelo de probabilidad de la función objetivo utilizando varias muestras de la misma, al que llamaremos modelo sustituto, que será la representación de probabilidad de la función objetivo, es decir, $P(\text{puntuación en función objetivo} \mid \text{hiperparámetro})$.

El modelo sustituto más común se trata de un modelo de proceso gaussiano que modela $P(x|y)$, utilizando del historial de pares (hiperparámetro, puntuación en función objetivo) como x e y para la construcción de las distribuciones gaussianas multivariadas.

Este modelo sustituto se utilizará para la selección de los siguientes hiperparámetros a evaluar. Para ello necesitaremos incluirlo en una función de adquisición o selección de manera que el siguiente hiperparámetro a seleccionar será aquel que maximice la función

de adquisición. La función de adquisición mas comúnmente utilizada es la mejora esperada, cuya ecuación se muestra en A.1.

$$ME_{y^*}(x) = \int_{-\infty}^{\infty} \max(y^* - y, 0) \cdot P_M(y|x) dy \quad (\text{A.1})$$

Siendo:

- $P_M(x|y)$: el modelo sustituto.
- y^* : la mínima puntuación en la función objetivo observada hasta el momento.
- y : la nueva puntuación sobre el modelo .

Por tanto, una vez entrenado el nuevo modelo, se utilizará el valor del error real obtenido como nueva muestra para actualizar el modelo sustituto

B. Anexo 2: Diagramas adicionales de bloques

B.1. Diagramas bloques ResNet

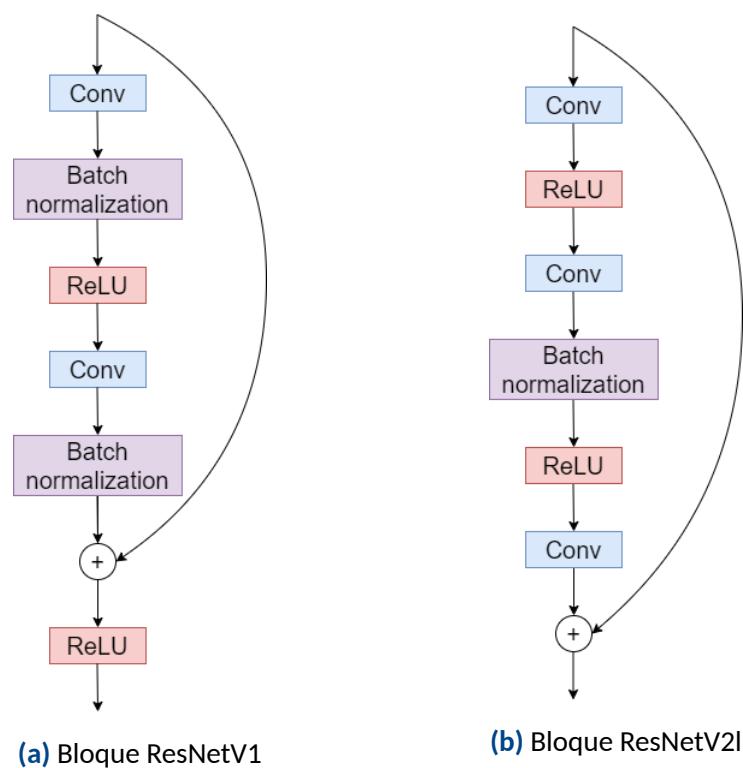
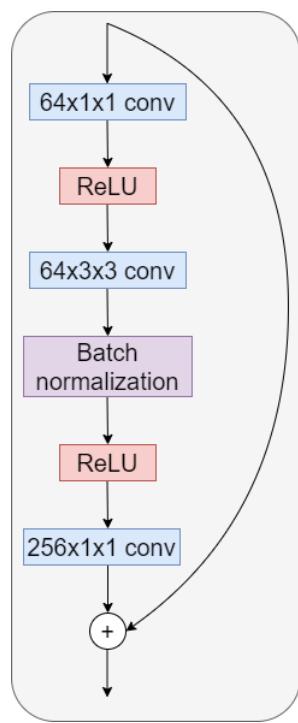
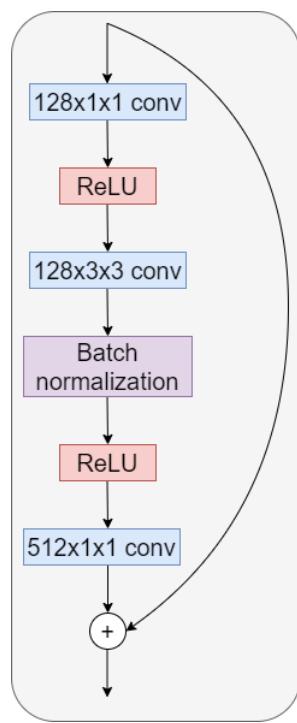


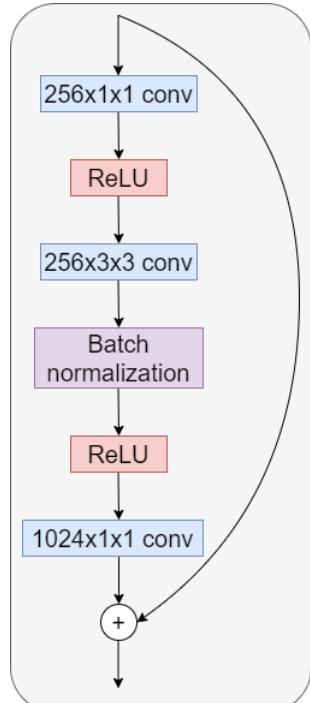
Figura B.1: Bloques ResNetV1 y ResNetV2



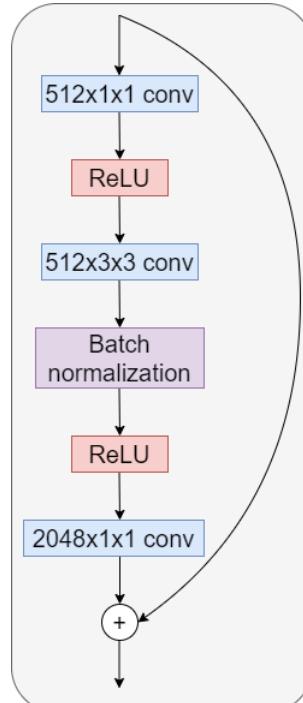
(a) ResNet bloque 1



(b) ResNet bloque 2



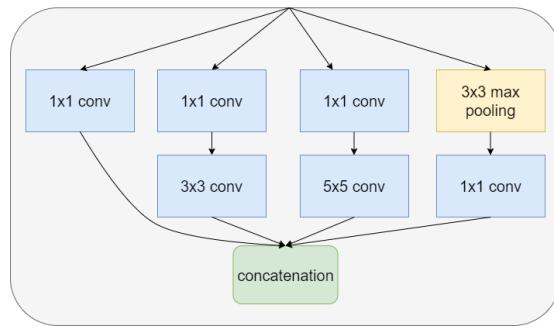
(c) ResNet bloque 3



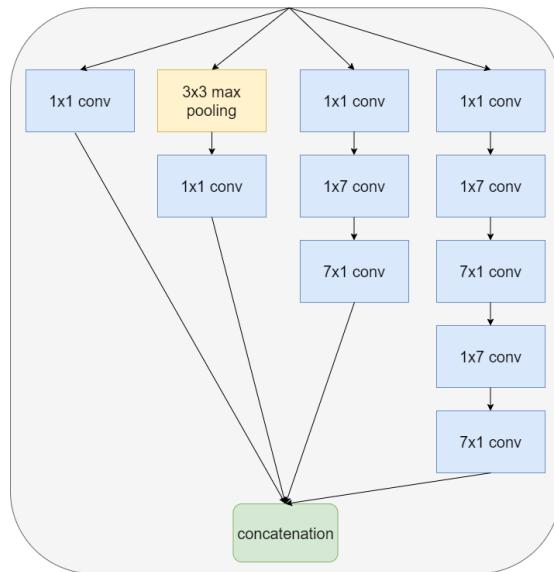
(d) ResNet bloque 4

Figura B.2: Bloques ResNet

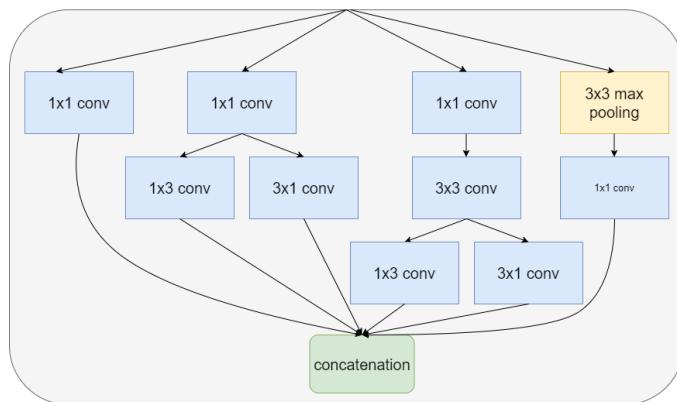
B.2. Diagramas bloques Inception



(a) InceptionV2 bloque A



(b) InceptionV2 bloque B



(c) InceptionV2 bloque C

Figura B.3: Bloques InceptionV2

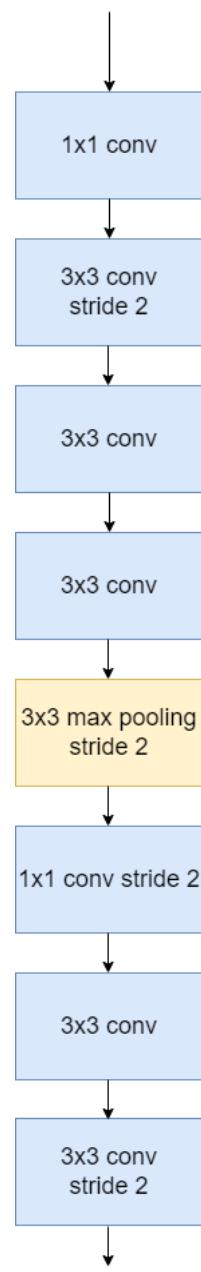
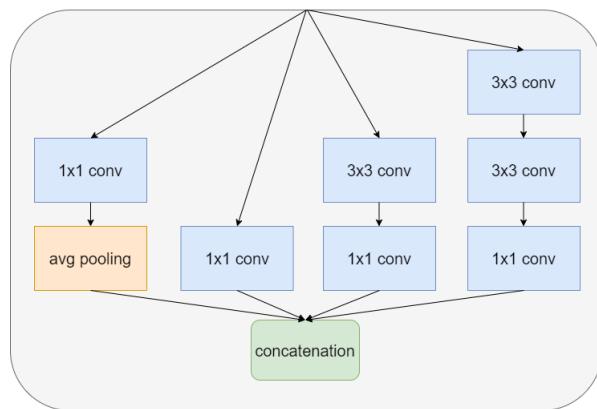
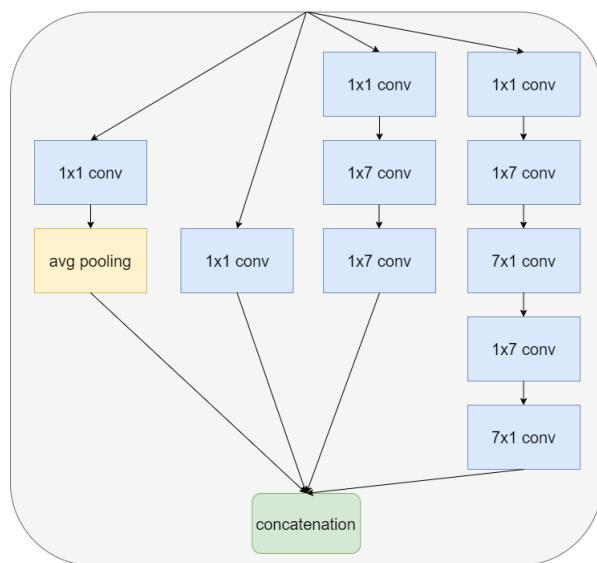


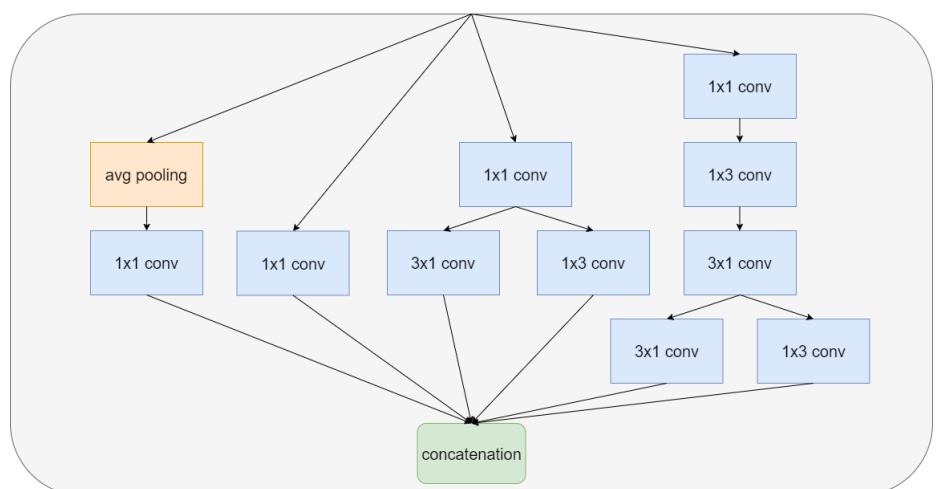
Figura B.4: InceptionV4 bloque inicial



(a) InceptionV4 bloque A



(b) InceptionV4 bloque B



(c) InceptionV4 bloque C

Figura B.5: Bloques InceptionV4

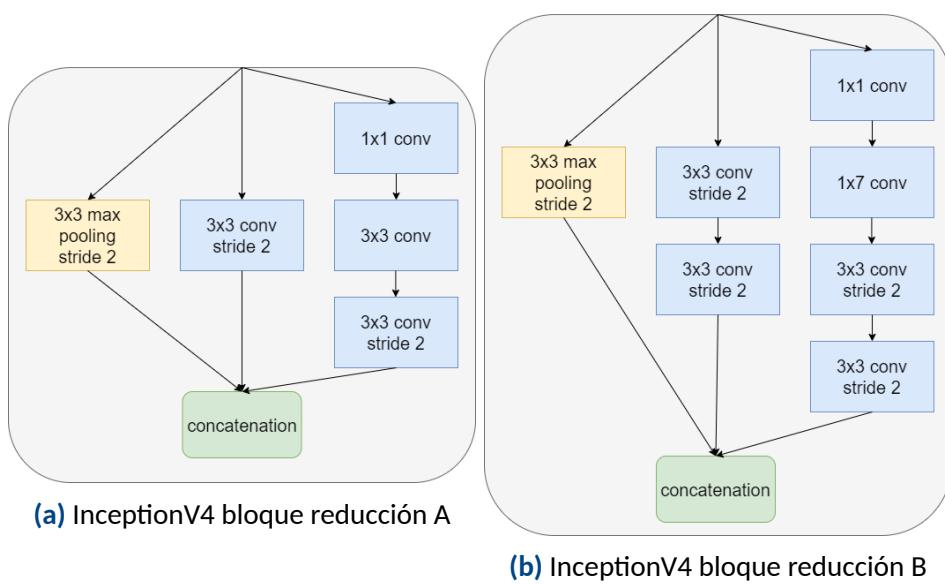


Figura B.6: Bloques de reducción InceptionV4

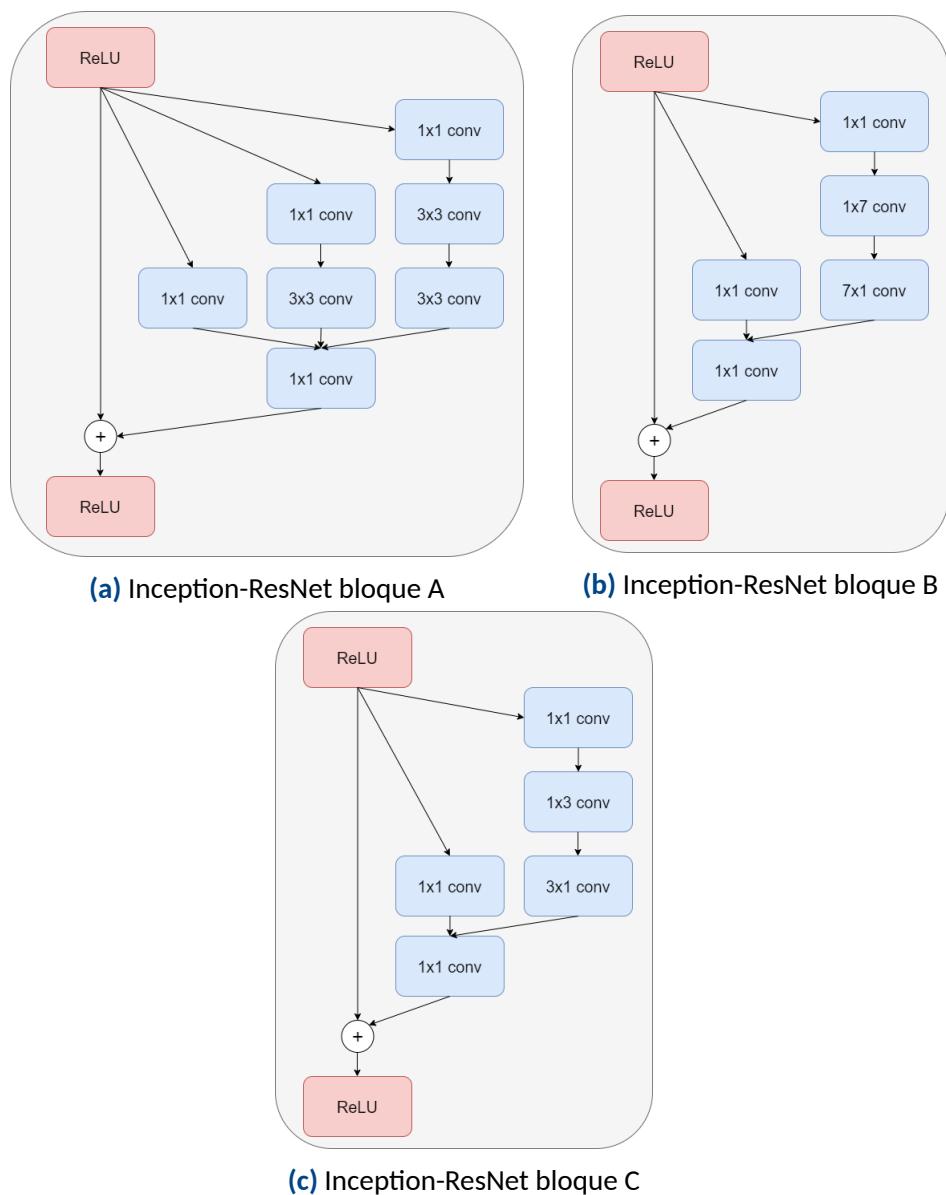


Figura B.7: Bloques Inception-ResNet

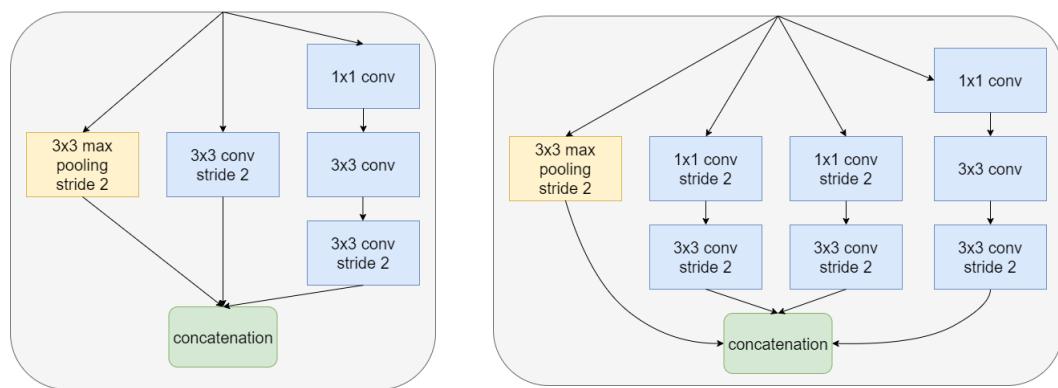


Figura B.8: Bloques de reducción Inception-ResNet

C. Anexo 3: Capas adicionales

C.1. Batch normalization

Las capas de batch normalization se aplican para evitar el desbalance entre las salidas de las diferentes neuronas durante el entrenamiento de la red, puesto que estas no tienen por qué estar en la misma escala (una neurona puede tener salidas en el rango [0-1] mientras que otra en el rango [1000-10000]).

Este desbalance puede ocasionar dos problemas relevantes. El primero se refiere a la ralentización de tanto el aprendizaje como la inferencia de la red, y el segundo está más relacionado con el desempeño de la red, ya que los altos valores de salida provocan la corrupción de los valores del gradiente (esto se conoce como el problema de la explotación del gradiente).

Las capas de batch normalization tienen la función de normalizar las salidas de la capa anterior de manera que la media corresponda a 0 con una desviación típica de 1.

Para ello, se aplica la siguiente fórmula para cada una de las salidas:

$$z = \frac{x - m}{s} \cdot g + b \quad (\text{C.1})$$

Siendo:

- z la nueva salida normalizada.
- x el valor de la salida actual.
- m la media de las salidas.
- s la desviación típica.
- g y b parámetros que se ajustarán durante el entrenamiento.

[Sergey Ioffe, 2015]

C.2. Dropout

El Dropout es una técnica utilizada para evitar el sobreajuste durante el entrenamiento de redes neuronales.

Esta se basa en aproximar una de las soluciones más prometedoras en cuanto a la evitación de este fenómeno: la combinación de modelos. Esta técnica consiste en inferir el resultado a partir del promedio del resultado devuelto por varias redes neuronales diferentes, lo que implicaría o bien entrenar varias arquitecturas diferentes, o bien entrenar la misma arquitectura varias veces, por lo que en la mayoría de las ocasiones este método es computacionalmente inabordable.

La técnica de Dropout es una manera de aproximar este método de una manera mucho más eficiente. La idea, como se muestra en la figura C.1, consiste en eliminar aleatoriamente ciertas neuronas de la red temporalmente durante el entrenamiento, ignorando tanto sus conexiones de entrada como de salida, variando así temporalmente la arquitectura de la red, de esta manera, de una red neuronal con n número de neuronas podremos obtener potencialmente 2^n distintos modelos.

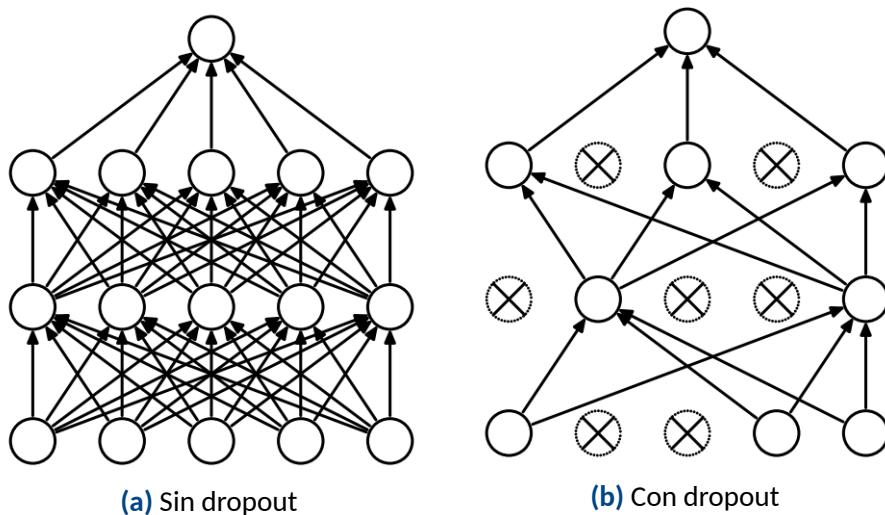


Figura C.1: Dropout

El comportamiento de las neuronas a las que se les aplique Dropout será distinto según si se trata de la fase de entrenamiento o la de inferencia.

- Durante la **fase de entrenamiento**, las neuronas con Dropout desaparecerán aleatoriamente en base al parámetro p , siendo p la probabilidad de que esa determinada neurona se mantenga en el entrenamiento, como se muestra en la figura C.2a.

- Durante la **fase de inferencia**, se requiere una manera de promediar el resultado y, puesto que resulta inviable realizar explícitamente la media de todos estos modelos, se utiliza una aproximación distinta. Esta se basa en utilizar una única red en validación con todas las neuronas activas. Los pesos de esta red serán una versión reducida de los pesos entrenados. Esto, como se muestra en la figura C.2b, se logra multiplicando cada uno de los pesos entrenados por la probabilidad de haber mantenido esa neurona durante el entrenamiento (p), lo que asegura que la salida esperada (bajo la distribución de Dropout) sea la misma que la salida real durante la validación. Este escalado permite combinar las 2^n posibles redes en una única a la hora de la validación.

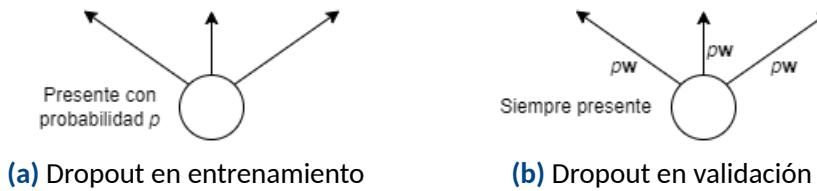


Figura C.2: Fases de dropout

Aplicando este método se consigue reducir en gran medida el sobre ajuste en una gran cantidad de problemas de clasificación [Nitish Srivastava, 2014].

D. Anexo 4: Optimizadores

D.1. SGD

El optimizador "Stochastic Gradient Descendent" [You et al., 2019], por sus siglas SGD, es un algoritmo basado en el gradiente descendiente. La diferencia radica en la manera de calcular el gradiente, ya que mientras que en el gradiente descendiente el gradiente se calcula en base a cada uno de los valores de los hiperparámetros, en SGD se calcula únicamente el gradiente de una serie de muestras aleatorias (de forma estocástica, de ahí su nombre), lo que reduce las necesidades computacionales.

D.2. Adam

El optimizador Adam [Bock et al., 2018], también conocido como "Estimación adaptativa del momento" es un algoritmo para optimización ampliamente utilizado por su rendimiento y eficiencia a la hora de entrenar modelos en base a bases de datos relativamente grandes. Se trata de un algoritmo relativamente complejo, ya que trata de combinar otros dos algoritmos de optimización: **AdaGrad** y **RMSProp**.

- **Adagrad** o algoritmo de gradiente adaptativo se trata de una adaptación de SGD (D.1) que utiliza tasas de aprendizaje distintas para cada variable. Para ello, tiene en cuenta el gradiente acumulado en ellas.
- **RMSProp** o algoritmo de propagación cuadrática media de la raíz es una variación de AdaGrad en la que, para cada hiperparámetro, se tendrán en cuenta únicamente los gradientes más recientes, en lugar de la acumulación de los mismos.

E. Anexo 5: Conceptos adicionales

E.1. Desvanecimiento del gradiente

Este fenómeno ocurre en redes neuronales profundas durante el aprendizaje de la red mediante **backpropagation**, y ocasiona que el factor del gradiente sea tan pequeño que las primeras capas de la red no puedan actualizar sus pesos [Hochreiter, 1998].

Para comprender este fenómeno debemos analizar la ecuación de backpropagation 2.9, introducida en la sección 2.1.3. En ella observamos que la salida de la neurona (h) multiplica al resto de la ecuación.

Este valor es el obtenido directamente de la **función de activación** de cada neurona, y será la naturaleza de dicha función la que determine el rango en el que se encontrará dicho valor.

Si observamos estas funciones de activación, podemos observar (especialmente en las de tipo sigmoide y tanh) que gran parte de los valores de salida estarán comprendidos entre 0 y 1 (o 0 y -1), y es aquí donde reside el problema: conforme el gradiente se propague hacia las primeras capas de la red, existirá una tendencia a que el mismo se reduzca hasta casi desaparecer en la primeras capas, lo que limitará la capacidad de aprender de redes relativamente profundas.

Referencia bibliográfica

- [Abril, 2013] Abril, R. R. (2013). Fer2013. <https://paperswithcode.com/dataset/fer2013>. Accedido en noviembre de 2022.
- [Abril, 2021] Abril, R. R. (2021). El descenso del gradiente. <https://lamaquinaoraculo.com/computacion/el-descenso-del-gradiente/>. Accedido en julio de 2022.
- [Baheti, 2022] Baheti, P. (2022). Why do neural networks need an activation function?]. <https://www.v7labs.com/blog/neural-networks-activation-functions>. Accedido en junio de 2022.
- [Baro, 2009] Baro, L. M. S. (2009). Compromiso sesgo-varianza.
- [Bock et al., 2018] Bock, S., Goppold, J., and Weiß, M. (2018). An improvement of the convergence proof of the adam-optimizer. *arXiv preprint arXiv:1804.10587*.
- [Chen, 2019] Chen, C. (2019). Emoción y sentimiento. <https://blogthinkbig.com/redes-neuronales-deep-learning>. Accedido en noviembre de 2022.
- [Chollet, 2014] Chollet, F. (2014). Documentación keras. <https://keras.io/api/>. Accedido en julio de 2022.
- [Christian Szegedy, 2015] Christian Szegedy, Vincent Vanhoucke, S. I. J. S. Z. W. (2015). Rethinking the inception architecture for computer vision. <https://arxiv.org/abs/1512.00567>. Accedido en julio de 2022.
- [Christian Szegedy, 2016] Christian Szegedy, Sergey Ioffe, V. V. A. A. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. <https://arxiv.org/abs/1602.07261>. Accedido en julio de 2022.
- [David W. Hosmer Jr., 2013] David W. Hosmer Jr., Stanley Lemeshow, R. X. S. (2013). *Applied Logistic Regression*. Wiley Series in Probability and Statistics. John Wiley Sons.
- [Education, 2020] Education, I. C. (2020). What are neural networks? <https://www.ibm.com/cloud/learn/neural-networks>. Accedido en noviembre de 2022.

-
- [Ekman, 1992] Ekman, P. (1992). An argument for basic emotions. *Cognition and Emotion*, 6(3-4):169–200.
- [Fawcett, 2005] Fawcett, T. (2005). An introduction to roc analysis. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4533825/>. Accedido en julio de 2022.
- [Frazier, 2018] Frazier, P. I. (2018). A tutorial on bayesian optimization. <https://arxiv.org/pdf/1807.02811.pdf>. Accedido en Noviembre de 2022.
- [Gao Huang, 2016] Gao Huang, Zhuang Liu, L. v. d. M. K. Q. W. (2016). Densely connected convolutional networks. <https://arxiv.org/abs/1608.06993>. Accedido en julio de 2022.
- [Hochreiter, 1998] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.
- [ichi.pro, 2021] ichi.pro (2021). Función de pérdida de entropía cruzada. <https://ichi.pro/es/funcion-de-perdida-de-entropia-cruzada-267783942726718>. Accedido en julio de 2022.
- [Kaiming He, 2015] Kaiming He, Xiangyu Zhang, S. R. J. S. (2015). Deep residual learning for image recognition. <https://arxiv.org/abs/1512.03385>. Accedido en julio de 2022.
- [Karen Simonyan, 2015] Karen Simonyan, A. Z. (2015). Very deep convolutional networks for large-scale image recognition. <https://arxiv.org/abs/1409.1556>. Accedido en julio de 2022.
- [Nitish Srivastava, 2014] Nitish Srivastava, Geoffrey Hinton, A. K. I. S. R. S. (2014). Dropout: A simple way to prevent neural networks from overfitting. <https://jmlr.org/papers/v15/srivastava14a.html>. Accedido en julio de 2022.
- [Paperswithcode, 2022] Paperswithcode (2022). Facial expression recognition on fer2013. <https://paperswithcode.com/sota/facial-expression-recognition-on-fer2013>. Accedido en julio de 2022.
- [Recuero, 2018] Recuero, P. (2018). ¿sabes en qué se diferencian las redes neuronales del deep learning? <https://blogthinkbig.com/redes-neuronales-deep-learning>. Accedido en julio de 2022.
- [Sammut and Webb, 2010] Sammut, C. and Webb, G. I., editors (2010). *Mean Squared Error*, pages 653–653. Springer US, Boston, MA.
- [Sergey Ioffe, 2015] Sergey Ioffe, C. S. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. <https://arxiv.org/abs/1502.03167>. Accedido en julio de 2022.

- [Taha AA, 2015] Taha AA, H. A. (2015). Metrics for evaluating 3d medical image segmentation. <https://people.inf.elte.hu/kiss/11dwhdm/roc.pdf>. Accedido en julio de 2022.
- [You et al., 2019] You, K., Long, M., Wang, J., and Jordan, M. I. (2019). How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878*.