Universidad Internacional de La Rioja (UNIR)

ESIT

Máster Universitario en Inteligencia Artificial

Sistemas cognitivos artificiales para la detección de *deepfakes* usando datos audio-visuales.

Trabajo Fin de Máster

Presentado por: Hazeu González, Mónica

Director/a: Torralba Elipe, Guillermo

Ciudad: Barcelona
Fecha: Julio 2021

Resumen

Los *deepfakes* se refieren a videos, audios u otros tipos de datos manipulados utilizando técnicas de *deep learning*. Pueden ser una fuente de desinformación y manipulación, y es por esto que en este trabajo se ha elaborado un modelo para detectar *deepfakes* a partir de datos audio-visuales. Con este propósito, se han entrenado dos modelos, uno de audio y otro de video, con datos procedentes de diversos *datasets*, buscando ante todo mejorar la capacidad de generalización, que escasea en los modelos existentes de detección de *deepfakes*. Se ha conseguido una capacidad de generalización relativamente buena comparada con el estado del arte. Además, se han combinado los dos modelos en una sencilla interfaz para que usuarios no expertos puedan detectar videos falsos.

Palabras Clave: audio-visual, deepfakes, transfer learning

Abstract

The term deepfakes refers to videos, audio or other types of data that have been manipulated using deep learning techniques. They can be a source of misinformation and manipulation, which is why in this project a model has been trained to detect deepfakes in audio-visual data. With this purpose, two separate models have been trained, one for audio and one for video, with data from a variety of datasets, looking above all to improve the generalization capacity, which is scarce in existing models. A relatively good generalization capability has been achieved compared with the state of the art. Additionally, the two models have been combined in a simple-to-use interface so non-expert users can detect deepfake videos.

Keywords: Audio-visual, Deepfakes, Transfer learning

Índice de contenidos

Índice de tablas

Índice de ilustraciones

1. Introducción

1.1 Motivación

El término *deepfakes* hace referencia a datos generados utilizando técnicas de *deep learning*. Generalmente, se refiere a audios, imágenes o videos generados mediante *autoencoders* o, más recientemente, *Generative adversarial networks* (GAN) (ver el capítulo 2.1). Se puede plantear la duda de si realmente esto supone un problema muy diferente al de las técnicas de edición de imágenes existentes (p.ej. *Photoshop*). La respuesta a esta duda es que las técnicas de generación de *deepfakes* no sólo son mucho más potentes para la generación de videos realistas, sino que esta potencia puede ser fácilmente puesta a disposición de un público no profesional, como ha sido el caso de la aplicación *FaceApp*. Esto puede ser una fuente de entretenimiento banal para algunos usuarios, o utilizarse con fines maliciosos. Un ejemplo de esto es una app desarrollada por un programador anónimo que permitía, a partir de fotos de mujeres vestidas, generar fotos realistas de ellas desnudas. La app fue rápidamente desmantelada, pero muestra la gravedad del problema de los *deepfakes*. (Samuel, 2019)

No es el único caso que ha habido de un uso malicioso de las técnicas de *deepfakes*. Existe una página denominada Mr. *DeepFakes* en la que se ponen a disposición del usuario videos pornográficos artificiales de famosos. En (Foley, 2021) se pueden ver otros 12 ejemplos de videos generados artificialmente. Uno de ellos muestra a Obama dando un discurso que en realidad nunca ha dado, con lo que se puede ver claramente el potencial para la desinformación que pueden tener esas técnicas.

Más allá de las falsificaciones de videos, recientemente el CEO de una compañía energética fue engañado para transferir 220000€ al que creía que era su jefe por teléfono, pero que en realidad era un audio generado mediante técnicas de *deepfakes* (Damiani, 2019). Esta capacidad de engaño y extorsión se puede ver ampliada aún más con las técnicas de *deepfakes* para videos.

1.2 Planteamiento del trabajo

Los *deepfakes* no sólo tienen fines maliciosos. Recientemente, en una campaña publicitaría de Cruzcampo, se ha “resucitado” a Lola Flores mediante técnicas de generación de *deepfakes*, por su puesto con el consentimiento de la familia, mandando un mensaje

motivador (García, 2021). Por su parte, la app *Deep Nostalgia* permite animar fotos antiguas de seres queridos, con gran realismo y facilidad (MyHeritage, n.d.). No se defiende por tanto en este trabajo que se deban prohibir los *deepfakes*, aunque tampoco se conseguiría si se quisiera. Se considera más acertado dotar a los usuarios de una forma de detectar si un video es real o sintético.

El objetivo de este trabajo es permitir esto mediante la creación de un modelo de detección de *deepfakes*. Existen ya trabajos anteriores, como (T. Chen et al., 2020; Guarnera et al., 2020; Güera & Delp, 2018), que han estudiado este problema, como se expondrán en detalle en la sección 2. Sin embargo, la capacidad de generalización de estos modelos es aún pobre y su accesibilidad es únicamente para un público técnico y formado en inteligencia artificial. El propósito de este trabajo es mejorar el estado del arte, con un foco especial en la capacidad de generalización del modelo, para posteriormente habilitar que un público con conocimientos básicos de Python pueda usar este modelo para inferir si un video concreto es artificial o real.

Con este propósito, se ha utilizado la técnica de *transfer learning* para entrenar modelos para detectar tanto videos como audios manipulados a partir de una variedad de *datasets*. Se ha entrenado estos modelos realizando previamente un pre-procesado de estos *datasets* para adecuarlos a la arquitectura el modelo. Tras el entrenamiento, se han evaluado estos modelos con datos de testeo ocultos para comprobar la capacidad de generalización del modelo. Además, se ha simplificado la parte de inferencia para que un usuario medio de Python pueda realizar predicciones sobre sus propios videos o audios.

1.3 Estructura de la memoria

En la sección 2 de este trabajo, se hará una revisión de la realidad actual en lo que a *deepfakes* se refiere, así como del estado del arte en su detección. Partiendo de esta revisión del estado del arte, en la sección 3 se describirán los objetivos concretos de este trabajo. En la sección 4, se explica la arquitectura que se ha elegido para solucionar el problema. Posteriormente, en la sección 5, se elabora acerca de la implementación concreta de la arquitectura, especificando las herramientas utilizadas en lo que a código se refiere. La sección 6 está dedicada a la evaluación de la arquitectura con los datos de entrenamiento, validación y testeo. Finalmente, la sección 7 expone las conclusiones finales y líneas futuras de trabajo, a lo que sigue en la sección 8 la bibliografía. Se cierra el trabajo con los anexos, que muestran el código utilizado, así como el artículo resumen de todo el trabajo.

2. Contexto y estado del arte

2.1. Técnicas de creación de *deepfakes*

2.1.1. Videos e imágenes

Para poder detectar *deepfakes*, antes conviene conocer cómo se generan. Primeramente, cabe distinguir en la actualidad tres tipos de *deepfakes* (según (Lyu, 2020)) en lo que respecta a la alteración de videos:

1. *Head puppetry*: se utiliza un actor "fuente" (*source*) y se sintetiza a una persona objetivo (*target*) para que parezca que está haciendo lo mismo que el actor fuente. El *deepfake* abarca la cabeza y zona de los hombros.
2. *Face swapping*: se superpone la cara del actor fuente sobre la del objetivo, alterando los rasgos faciales del objetivo, pero manteniendo sus expresiones.
3. *Lip syncing*: se altera solo la zona de los labios del objetivo a partir de la fuente, normalmente acompañado de audio falsificado para que parezca que dice algo que no dice.

De estos tres tipos de manipulaciones, se considera que la primera y la tercera son las más peligrosas, ya que la efectividad del *face swapping* es muy dependiente del parecido entre la persona fuente y la persona objetivo. Por otro lado, esta variante es la que está más comercializada en la actualidad, y la más disponible para un público no técnico. Como se expresa en (Lyu, 2020), existe código libre como *FakeApp* o *faceswap-GAN*, así como servicios online como *deepfakesweb* o incluso *start-ups* con herramientas para la creación de *deepfakes* como Synthesia[1].

Por otro lado, hay distintas tecnologías para la creación de estos tres tipos de *deepfakes*. Los *deepfakes*, como se ha mencionado, se crean por medio del *deep learning*. El primer modelo fue *FakeApp*, que usaba dos pares de *encoder-decoder* (codificadores y decodificadores), es decir, un modelo *autoencoder* (Rumelhart et al., 1985). Un *autoencoder* es una arquitectura de red neuronal en la que la función objetivo es el mismo input de la red. La primera mitad de la arquitectura es el *encoder*, que reduce la dimensionalidad de los datos de entrada. La segunda mitad es el *decoder*, que reconstruye la imagen a partir de la salida del *encoder*, comparando la función de coste la salida del *decoder* con la entrada del *encoder*. En el caso de *FakeApp*, el *encoder* se comparte entre las dos imágenes, la fuente y el objetivo, y se encarga de extraer las *features* similares entre las dos caras. Posteriormente, cada imagen

[1] https://www.synthesia.io/

tiene un *decoder* que se encarga de reconstruir la imagen a partir de la codificación del *encoder* (T. T. Nguyen et al., 2019). Así, aplicando el *decoder* de la imagen objetivo a la codificación de la imagen fuente, se genera una reconstrucción de la imagen objetivo con los rasgos en la zona de la cara de la imagen fuente, es decir, un *deepfake* de tipo *face swap*.

En este tipo de arquitectura se basan otros modelos como *DeepFaceLab*, *DFaker*, o *deepfake-tf* (T. T. Nguyen et al., 2019). Sin embargo, existe una versión mejorada, capaz de crear *deepfakes* de mayor calidad y más realistas, que se basa en añadir a esta arquitectura *adversarial loss* (Cycle-GAN (J.-Y. Zhu et al., 2017)) y *perceptual loss* (VGGFace (Parkhi et al., 2015)), además del MAE (*Mean Absolute Error*) del *autoencoder*. En resumen, la arquitectura es como se muestra en la Ilustración 1, donde se muestra la combinación de un *autoencoder* con una *Generative Adversarial Network* (GAN) (*Faceswap-GAN*, 2019). Esto último es un tipo de arquitectura que combina un generador con un discriminador, donde el primero intenta capturar la distribución de los datos de entrada para generar imágenes que parezcan reales y el segundo intenta diferenciar las imágenes reales de las falsas, generándose por tanto una competición entre ambas (*minmax*) (Guarnera et al., 2020).

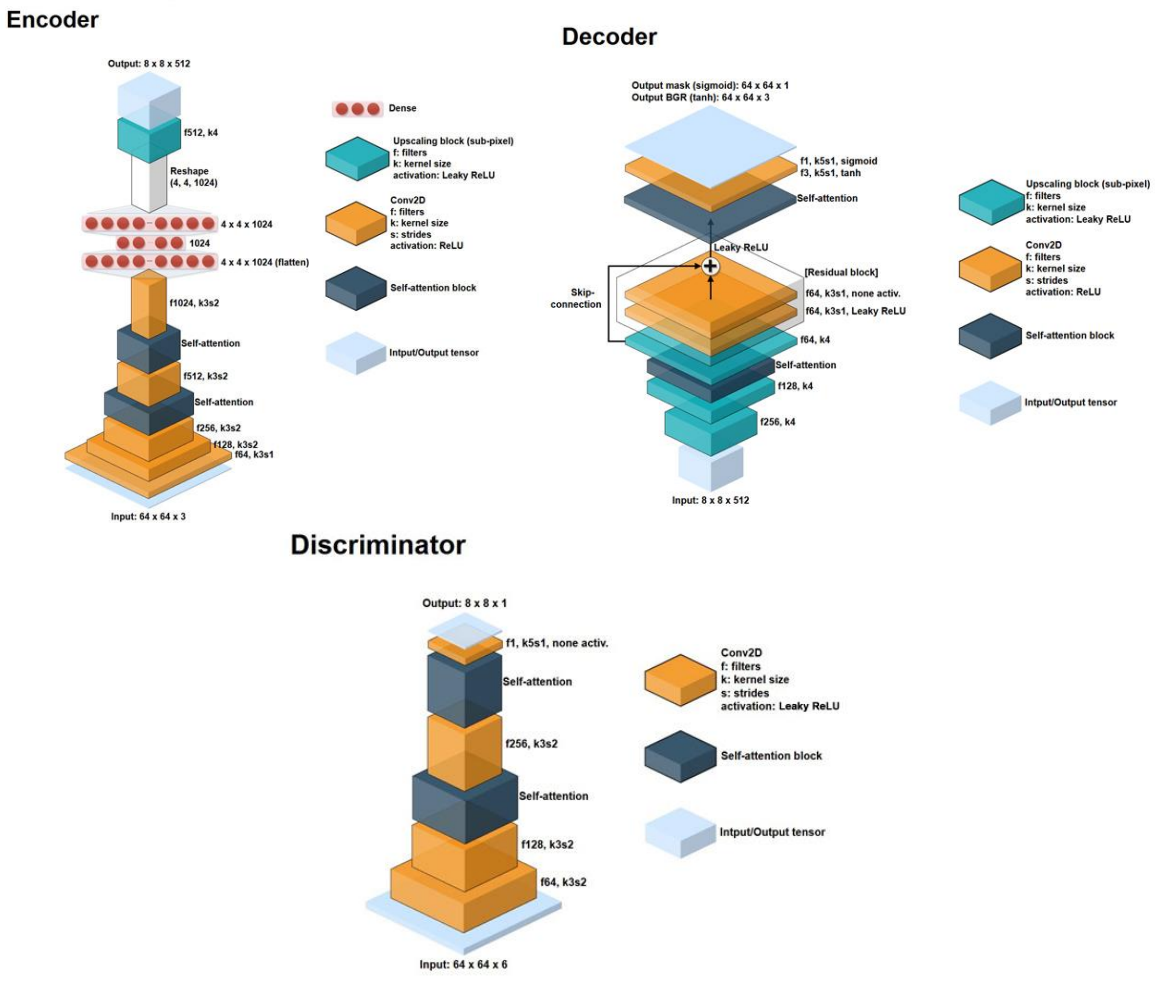


Ilustración 1. Arquitectura Faceswap-GAN. Fuente: (*Faceswap-GAN*, 2019)

Otras arquitecturas de generación de *deepfakes* basadas en GANs incluyen: CycleGAN (en la que se basa la arquitectura anterior), StarGAN, AttGAN, *Group-wise deep whitening and coloring method* (GDWCT), StyleGAN2 (se usó para crear la página "*This person does not exist*"), ProGAN, FaceForensics++, IMLE y Spade (Guarnera et al., 2020).

Con respecto a la calidad de los *deepfakes*, se pueden distinguir en cierto modo dos generaciones (Tolosana et al., 2020):

- La primera generación genera videos de baja calidad, con diferencias de color de piel entre el objetivo y la máscara de la fuente, bordes visibles entre estos, poca variación en las posturas y numerosos artefactos.
- En la segunda generación se mejoran muchos de estos artefactos, considerándose distintas escenas, posturas y distancias a la cámara.

2.1.2. Audio

En lo que respecta a la generación de audio sintético, se plantea como un problema de *Text-to-Sound* (TTS). Acercamientos iniciales se basaban en modelos ocultos de Markov (HMM) (Rabiner & Juang, 1986), utilizando síntesis de audio estadística paramétrica (SPSS) (Zen et al., 2009). Sin embargo, más recientemente se ha optado por utilizar modelos de *deep learning*.

Deep Voice 3 utiliza una arquitectura *encoder-decoder*, junto con un *converter*, para convertir texto escrito a una representación interna, que luego el *decoder* transforma a una representación de audio de baja dimensión (espectrograma en escala Mel[2]). Posteriormente, el *converter* predice los parámetros finales del *vocoder* (codificador de voz). La arquitectura es completamente convolucional. No se utilizan redes recurrentes con el objetivo de acelerar el entrenamiento. (Ping et al., 2017)

Por su parte, *Multi-task WaveNet* utiliza aprendizaje multi-tarea (MTL), con una arquitectura basada en *dilated convolution* (la salida de la convolución depende no sólo de los pixeles inmediatamente vecinos, sino que se concatenan convoluciones usando vecinos a distintas distancias al estilo de *wavelet decomposition*) y *quasi-recurrent neural networks* (QRNN) (arquitectura que alterna capas convolucionales paralelizables con una función de *recurrent*

[2] La escala Mel es una escala de frecuencia que tiene en cuenta la percepción subjetiva del audio por parte de los humanos, aplicando un escalado logarítmico, ya que el oído humano percibe mayores diferencias entre frecuencias más bajas que en frecuencias más altas. La ecuación de conversión de frecuencia en Hz a frecuencia en escala Mel es: $m = 2595 \cdot \log\left(1 + \frac{f}{500}\right)$

pooling) para aprender a la vez la generación del audio y las *features* acústicas. (Gu & Kang, 2018)

Como se verá más adelante, sin embargo, las técnicas de generación de audio sintético son extremadamente variadas, por lo tanto, lo más conveniente es tener un *dataset* en el que se incluya esta diversidad de técnicas.

2.2. Fuentes de datos

Como suele ser el caso, a raíz de estas técnicas de manipulación de la información han surgido esfuerzos para mejorar las técnicas de detección de *deepfakes*, también llamado *deepfake forensics*. Para intentar apoyar estos esfuerzos, se han generado una serie de *datasets* de videos reales y falsos para entrenar modelos capaces de detectar *deepfakes*. Una lista de los principales *datasets* incluye (Li et al., 2020):

Primera generación:

- UADFV: uno de los primeros datasets, incluye 49 videos reales y 49 falsos, siendo los segundos una superposición de la cara de Nicolas Cage sobre el video original. Los videos se generaron usando *FakeApp*.
- Deepfake-TIMIT: usando *faceswap-GAN* y basándose en vid-TIMIT, el *dataset* incluye 640 videos falsos, la mitad en alta calidad y la otra mitad en baja calidad de video.
- FaceForensics++: incluye 1000 videos reales y 1000 videos falsos generados con cuatro técnicas distintas.

Segunda generación:

- Celeb-DF: compuesto de 590 videos reales de entrevistas a famosos y 5639 videos falsos generados a partir de ellos. Los videos son de gran calidad, con pocos artefactos, una gran variedad de posturas y condiciones ambientales.
- *DeepFake Detection Challenge* (DFDC): este *dataset* se ha publicado como parte de una competición de *Kaggle* denominada *Facebook DeepFake Detection Challenge*. Contiene 1131 videos reales de 66 individuos y 4113 videos falsos basados en los reales y generados usando actores. En este caso, no se detallan las técnicas de generación de los *deepfakes* con el objetivo de que los modelos de detección no se sobre-ajusten a estas técnicas, mencionando sólo que se utilizan dos métodos. Además, tanto el audio como el video pueden estar manipulados, aunque no se especifica en qué videos el audio está también manipulado.
- *DARPA Media Forensics Challenge (MFC)*: organizado por el *National Institute of Standards and Technology* (NIST), el *Synthetic Data Detection Challenge* busca avanzar el ámbito de las técnicas de detección de manipulación proporcionando dos *datasets*: *Manipulation Detection and Localization* (MDL), con 50000 imágenes y 5000

videos, donde el objetivo es detectar si los videos o imágenes han sido manipulados y dónde, y el *Splice Detection and Localization* (SDL), con 50000 imágenes, donde el objetivo es, dadas dos imágenes, detectar si una región de una imagen ha sido introducida en la otra y, en tal caso, localizar dicha región. (NIST, 2018)

- *Deeper Forensics* 1.0: se compone de un total de 60000 videos manipulados y no manipulados de alta calidad, generados con una nueva técnica de generación de *deepfakes* e incluyendo un conjunto de test oculto. (Jiang et al., 2020)

Por la parte del audio, existe también una competición de detección de voz sintética, el *Global Automatic Speaker Verification* (ASV) *spoofing Challenge*, en concreto como una subcategoría de la competición. En este *dataset*, se incluyen audios generados con una gran variedad de técnicas, y además se garantiza que se comprueba la capacidad de generalización porque el set de testeo está generado mediante 11 técnicas desconocidas y que no se han utilizado en los datos de entrenamiento (ASV, 2021).

2.3. Técnicas de detección de *Deepfakes*

2.3.1. Estado del Arte

La mayoría de los proyectos de investigación consideran la detección de *DeepFakes* como un problema de clasificación binaria (verdadero o falso). Para (Lyu, 2020), existen principalmente tres acercamientos para esta clasificación:

1. Basados en inconsistencias físicas o fisiológicas (frecuencia de parpadeo, postura de la cabeza, etc.)
2. Artefactos a nivel de señal (usando p.ej. la transformada de Fourier)
3. Basado en datos (entrenamiento con un *dataset*)

En (T. T. Nguyen et al., 2019), se hace una revisión del estado del arte (septiembre 2019) en cuanto a detección de *deepfakes*. Ahí, se dividen las técnicas de detección en las categorías que se muestran en la Ilustración 2. Se va a hacer una revisión del estado del arte en base a esta división.

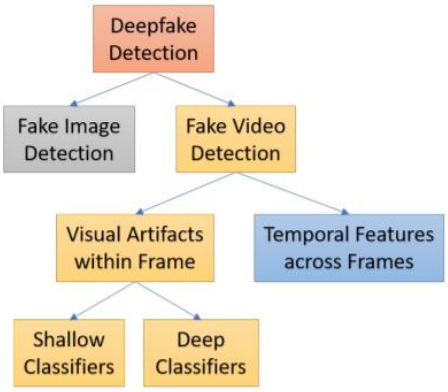


Ilustración 2. Categorías de técnicas de detección de DeepFakes según (T. T. Nguyen et al., 2019)

De las técnicas de generación de *DeepFakes*, se considera que las que generan imágenes de mayor calidad son las basadas en GANs. Sin embargo, muchos métodos de detección de *fakes* no tienen en cuenta la capacidad de generalización de sus modelos (T. T. Nguyen et al., 2019). Por tanto, y en el ámbito de detección de imágenes falsas, (Xuan et al., 2019) introducen un paso de pre-procesado de las imágenes con *Gaussian blur* (emborronado de imágenes con una máscara gaussiana aplicada a la vecindad de cada pixel) para evitar el sobreajuste del modelo y obligarlo a aprender *features* más allá de los errores de la GAN concreta.

También en el ámbito de la detección de imágenes falsas, existen otros acercamientos:

- *Common Fake Feature Network* (CFFN) (Hsu et al., 2020). Una arquitectura que denominan *Common Fake Feature Network* (CFFN), que consiste en un *back-bone* de *DenseNet* (Iandola et al., 2014) y una *Siamese network architecture* (Sem-Jacobsen et al., 2005). En concreto, se emparejan imágenes falsas con imágenes verdaderas y se entrena la primera parte de la red (convolucional) mediante *contrastive learning* para que aprenda las *features* diferenciadoras entre ambas imágenes (*pairwise learning*). Como segundo paso de entrenamiento, se entrena la última capa convolucional y la *fully-connected*, que constituye la parte clasificadora de la red, mediante *binary cross-entropy loss* (ver ecuación (1)). Esto se hace en este orden porque se ve empíricamente que es más importante aprender las *features* discriminadoras que el entrenamiento de la parte clasificadora. Los resultados en cuanto a precisión (*precision*) (ecuación (2)) y exhaustividad (*recall*) (ecuación (3)) son superiores a los métodos con los que se compara la arquitectura, pero los mismos autores expresan la preocupación de que el modelo no generalice bien frente a GANs que produzcan otro tipo de artefactos, y señalan que en ese caso probablemente el modelo debería ser reentrenado.

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log\big(p(y_i)\big) + (1 - y_i) \cdot \log\,(1 - p(y_i)) \quad (1)$$

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

- *Hypothesis testing* (Agarwal & Varshney, 2019). Se plantea la detección de *deepfakes* como un problema de *hypothesis testing*, en el que el objetivo es discernir si la imagen proviene de la distribución de probabilidad real (H_0) o de la distribución de probabilidad de una GAN (H_1). Se propone por tanto un *framework* estadístico para el análisis. Este *framework* será más efectivo cuanto peor sea la calidad de la GAN, ya que, en esencia, una GAN intenta minimizar la diferencia entre la distribución de probabilidad real y la falsa.

- Trazas convolucionales (Guarnera et al., 2020). Se busca extraer una huella de cada arquitectura GAN que ha generado la imagen falsa usando el algoritmo *Expectation-Maximization* (EM) (Moon, 1996). En concreto, se busca encontrar las trazas convolucionales que dejan las GANs para detectar las imágenes falsas. Se analizan imágenes generadas por 10 arquitecturas GAN, generando 2000 imágenes para cada una. En cierto modo es similar al acercamiento de *Hypothesis Testing*, en el sentido de que se busca calcular la probabilidad de que la imagen pertenezca a un modelo (real) u otro (falso). Mediante el algoritmo EM, se extrae un vector de características, que posteriormente se clasifica usando *Random Forest*.

Las técnicas de detección de imágenes falsas pueden ser fuente de inspiración para las técnicas de detección de videos falsos. Esto es especialmente cierto en el caso de las técnicas basadas en artefactos dentro de cada *frame*. Durante la generación de *DeepFakes*, se generan una serie de artefactos, como se mencionaba anteriormente, y estas técnicas buscan entrenar al modelo para identificar dichos artefactos. Existen propuestas usando técnicas de *deep learning*:

- Diferencias de resolución entre la zona de la cara y los alrededores (Li & Lyu, 2018). Se propone detectar estas inconsistencias mediante modelos de redes neuronales convolucionales (CNN). Para mejorar la capacidad de generalización, en el entrenamiento no se usan *deepfakes*, sino que se generan imágenes de forma dinámica durante el entrenamiento a las que se aplica *Gaussian blurring* en la zona de la cara, simulando las transformaciones afines que debe realizar una GAN para

introducir la cara de una persona en la de otra. Se evalúa este método con *datasets* de la primera generación (UADFC y *DeepfakeTIMIT*), obteniendo buenos resultados pues estos presentan artefactos de este tipo y son *deefakes* de tipo *face swap*.

- MesoNet (Afchar et al., 2018). Se proponen dos arquitecturas: Meso-4 es una pequeña red convolucional de 4 capas, con *batch normalization* y *max pooling*, seguida de dos capas *fully connected* con *dropout* de 0.5; MesoInception-4 sustituye las dos primeras capas convolucionales por modulos *inception*, el cual usa *dilated convolutions*. Se entrenan las dos arquitecturas por separado con un dataset de *Deepfake* (*face swap*) y uno de *Face2Face* (*head-puppetry*), pues esta separación es la que obtiene mejores resultados.

- *Capsule Forensics* (H. H. Nguyen et al., 2019). La intención es evitar las limitaciones de las CNNs al aplicarse a tareas de *inverse graphics* (hallar los procesos que dan lugar a las imágenes). Se basa en utilizar un *backbone* compuesto por las primeras capas de VGG-19 (véase la Ilustración 3), añadiendo después 3 o 10 bloques de *capsule networks* (en función de la arquitectura ligera o más precisa). Estas últimas son bloques de convoluciones independientes, generando algo parecido a un *ensemble* de modelos y utilizando *dynamic routing* para mapear las salidas de cada *capsule* a las *capsules* clasificadoras (una por clase). La ventaja principal de esta arquitectura es la consecución de resultados equiparables a las basadas en CNN con menos parámetros y un entrenamiento más eficiente. Este método se prueba con 4 *datasets*, obteniendo mejores resultados que todos sus competidores.

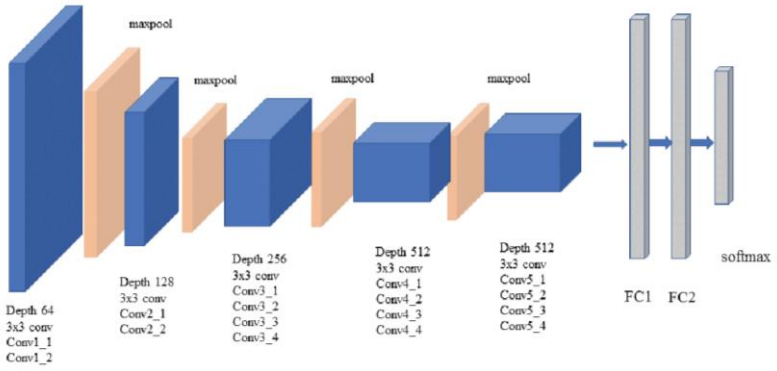


Ilustración 3. Arquitectura VGG-19. Fuente: (Satterfield, 2020)

Por otro lado, las técnicas basadas en redes neuronales profundas pueden dar lugar fácilmente al sobreajuste, y es por esto que otros investigadores prefieren utilizar técnicas *shallow*:

- *Support Vector Machines* (SVM) basado en la diferencia de posturas de la cabeza (X. Yang et al., 2019). Se basa en 68 *landmarks* (“marcadores”) faciales que luego el modelo SVM utiliza para clasificar el video. Las pruebas se realizan sobre el UADFV y una selección de videos de otro *dataset* (DARPA MediFor GAN), obteniendo muy buenos resultados en comparación con sus competidores.

- *Photo response non uniformity* (PRNU) (Koopman et al., 2018). PRNU es un patrón de ruido diferente para cada cámara digital que se genera por las inconsistencias en la fabricación. A menudo se considera una huella digital de imágenes (T. T. Nguyen et al., 2019). Así, la manipulación de la zona facial al generar un d*eepfake* alteraría el patrón de ruido en esa zona. Por tanto, en el método propuesto se selecciona la región de la cara y se calcula el PRNU medio de cada grupo de 8 *frames*. Comparando estos grupos se haya una diferencia estadísticamente significativa, mostrando potencial, aunque huelga decir que este método se prueba sólo en 10 videos reales y 16 falsos, por lo que los resultados son prematuros.

Por otro lado, existen acercamientos en la detección de videos falsos que consideran más adecuado tener en cuenta las inconsistencias temporales (*interframe*) que se derivan de la generación de videos falsos. Esencialmente, estas técnicas se basan en el uso de redes neuronales recurrentes (RNN):

- CNN combinada con LSTM (red *Long Short Term Memory*) (Güera & Delp, 2018). En este acercamiento, la CNN se utiliza para extraer artefactos dentro de cada frame, pasando posteriormente esas *features* a la LSTM para extraer inconsistencias entre *frames*, y después a una *fully-connected* para clasificar el video.

- Frecuencia de parpadeo (Li et al., 2018). Esta técnica se basa en la incapacidad de los métodos de generación de *deepfakes* de producir un patrón de parpadeo acorde con la frecuencia de un ser humano, por lo general parpadeando con mucha menor frecuencia. Se cree que esta incapacidad se debe a que los algoritmos se basan en imágenes, donde las personas suelen tener los ojos abiertos. Se utiliza por tanto la región de los ojos y una *long-term recurrent convolutional network* (LRCN) para hallar la frecuencia de parpadeo y detectar así los videos falsos.

- Redes recurrentes convolucionales (Sabir et al., 2019). El *pipeline* del modelo consiste en un pre-procesado en el que se detecta la cara, se recorta y se alinean los frames. Posteriormente, se pasan los recortes a una red convolucional, que se entrena para extraer *features* importantes. La salida de la convolución en cada *frame* se utiliza como input de la red recurrente, que se utiliza para detectar las inconsistencias temporales.

Se utiliza recurrencia bi-direccional, mostrándose mediante experimentos que es superior a la recurrencia uni-direccional. Se muestra además experimentalmente que los resultados son mejores utilizando la salida de la red convolucional como entrada a la recurrente que utilizando salidas en distintas dimensiones de la red convolucional (capas anteriores) para entrenar múltiples redes recurrentes. Se utiliza DenseNet como *backbone* (convolucional).

Finalmente, existe otro acercamiento que utiliza información no sólo visual, como los anteriores, sino que suma a esta la disonancia entre audio e imagen para detectar los *deepfakes*. Entre los estudios que optan por este acercamiento se encuentran:

- Uso de *affective cues* (señales emocionales) (Mittal et al., 2020). Se basa en la hipótesis de que las distintas modalidades de un video (en este caso audio e imagen) se pueden mapear a un espacio en el que la distancia entre ambas modalidades será muy pequeña para videos reales y más grande para videos falsos. Además, este será el caso también para las emociones reconocidas a partir del audio y a partir del video. Por lo tanto, el acercamiento se basa en una red *Siamese*, a la cual se le introduce durante el entrenamiento a la vez un video falso y un video verdadero del mismo sujeto. Utilizando *OpenFace* (Amos et al., 2016) y *pyAudio Analysis* (Giannakopoulos, 2015) se extraen los *features* visuales y de audio, respectivamente, los cuales pasan por una red convolucional para audio y otra para imagen, concluyendo con cuatro vectores finales de características, dos para el video real y dos para el falso. A la vez, se utiliza una *Memory Fusion Network* (MFN) (Zadeh et al., 2018) para reconocer emociones de ambas modalidades, pre-entrenada con el *dataset* CMU-MOSEI, el cual describe 6 emociones. Con todo, se compara la distancia euclídea entre los vectores de características para los videos reales y falsos, estableciendo un umbral que se usará en el testeo para clasificar los videos como reales o falsos en función de la distancia calculada para un único video.

- Sincronización entre video y audio (Chugh et al., 2020). El objetivo es detectar la falta de sincronización (*dissonance*) entre audio y video en *deepfakes*, asignando un *Modality Dissonance Score* (MDS, disimilitud acumulada). Se utilizan segmentos de 1 segundo del video, a los cuales se recorta la zona de la cara para la parte visual utilizando S3FD (detector de caras). Se procesa la parte auditiva, transformándola a *Mel-frequency cepstral coefficients* (MFCC) y representando las 13 bandas de frecuencia en segmentos de un segundo como un mapa de calor (*heatmap*). Por tanto, tanto la parte auditiva como la visual se utilizan para entrenar redes neuronales independientes diseñadas para el análisis de imágenes, en concreto 3D-ResNet para

la parte visual y una red convolucional específica para la parte de audio. La red se entrena utilizando *cross-entropy loss* (ver ecuación (1)) para cada una de las modalidades por separado para permitir el análisis de videos únicamente o audio únicamente. Además, se añade *contrastive loss* (ecuación (4)) para comparar la similitud entre audio y video y detectar así los *deepfakes*.

$$Loss_{contr} = (1 - Y) \cdot \frac{1}{2} \cdot (D_W^i)^2 + Y \cdot \frac{1}{2} \cdot \{\max(0, m - D_W^i\}^2 \quad (4)^3$$

Huelga decir que estos métodos que combinan audio e imagen usan los *datasets Deepfake-TIMIT* y DFDC, que incluyen ambas modalidades, con el inconveniente de que en los videos falsos el audio no necesariamente es falso, por lo que la clasificación del audio por separado como falso no tiene por qué ser correcta. Esto se observa claramente en el segundo de los dos estudios, donde la red clasificadora de audio por sí sola tiene un AUC del 50%, que es esencialmente un clasificador aleatorio. Como métodos dedicados a la detección de audio sintético, se pueden destacar:

- ResNets (ver Ilustración 4) con *data augmentation* (T. Chen et al., 2020). Utilizando los datos del ASVspoof 2019, junto con diversas técnicas como *frequency masking* y la adición de ruido, entrenan una red ResNet usando *large margin cosine loss* (ver ecuación (7)) para aumentar la varianza entre clases (real, falso) y disminuir la varianza dentro de la clase. Esta competición garantiza una capacidad de generalización bastante mayor, ya que el *dataset* de testeo está generado con 11 técnicas que no se utilizan para generar el *dataset* de entrenamiento.

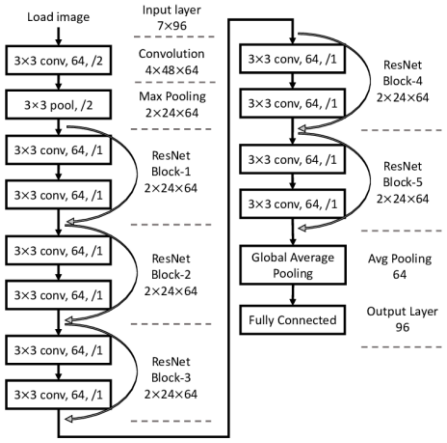


Ilustración 4. Arquitectura ResNet. Fuente: (Choi et al., 2018)

- Características artificiales del audio sintético (Gao et al., 2021). Es un estudio estadístico en el que se basan en las diferencias entre el audio sintético y el audio real,

[3] D_w es la distancia entre dos puntos de la entrada al modelo. Y son las etiquetas.

como por ejemplo el número de pausas y la transición hacia estas (más marcada en el audio sintético), encontrando diversas *features* de la señal con diferencias estadísticamente significativas entre audios sintéticos y reales.

2.3.2. Humanos en la detección de *deepfakes*

Es interesante comparar la precisión de las máquinas en la detección de *deepfakes* con la precisión de los humanos. Con este propósito, (Korshunov & Marcel, 2020) realizan un estudio en el cual piden a 60 personas que clasifiquen 120 videos (la mitad falsos y la mitad verdaderos), clasificando cada uno de los sujetos un *subset* de 40 videos elegidos aleatoriamente (para reducir el tiempo invertido por cada sujeto). El estudio pide que los sujetos clasifiquen los videos como verdaderos, falsos o “no lo sé”. Los sujetos son miembros del *Indiap Research Institute* (estudiantes de doctorado, científicos, administrativos, etc.) y son conscientes de que están viendo videos potencialmente manipulados.

Por otro lado, estos mismos 120 videos se evalúan con dos clasificadores, Xception (Chollet, 2017) y EfficientNet (Tan & Le, 2019), entrenados cada uno por separado con FaceForensics++ y Celeb-DF, siendo por tanto estos 120 videos un test set oculto. Esto es importante, ya que estos *datasets* tienen AUCs altos en los *datasets* sobre los que han sido entrenados, cosa que, como se verá, no sigue siendo el caso en el test set oculto.

Los investigadores dividen los 60 videos falsos en cinco categorías, según su criterio, que clasifican la facilidad de detección del *deepfake*, siendo estas, por tanto: muy fácil, fácil, moderado, difícil y muy difícil. En la Ilustración 5 se pueden ver los resultados obtenidos por parte de los humanos en la clasificación de *deepfakes*. Como se puede ver, la intuición de los investigadores era correcta y en los videos considerados fáciles se obtiene la mayor precisión, mientras que en los difíciles no se supera el 50% de precisión.

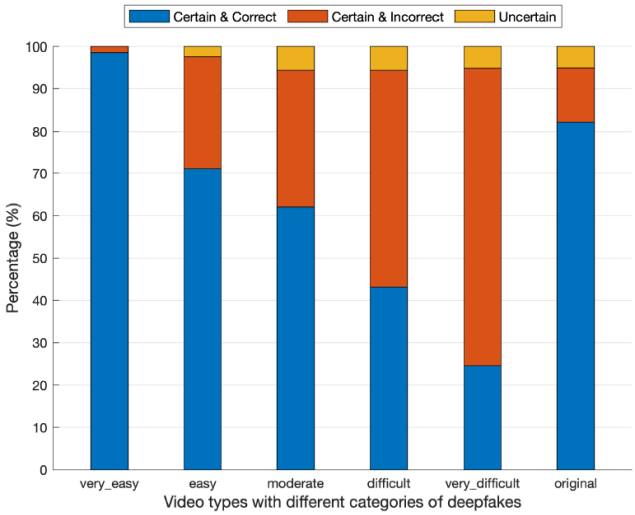
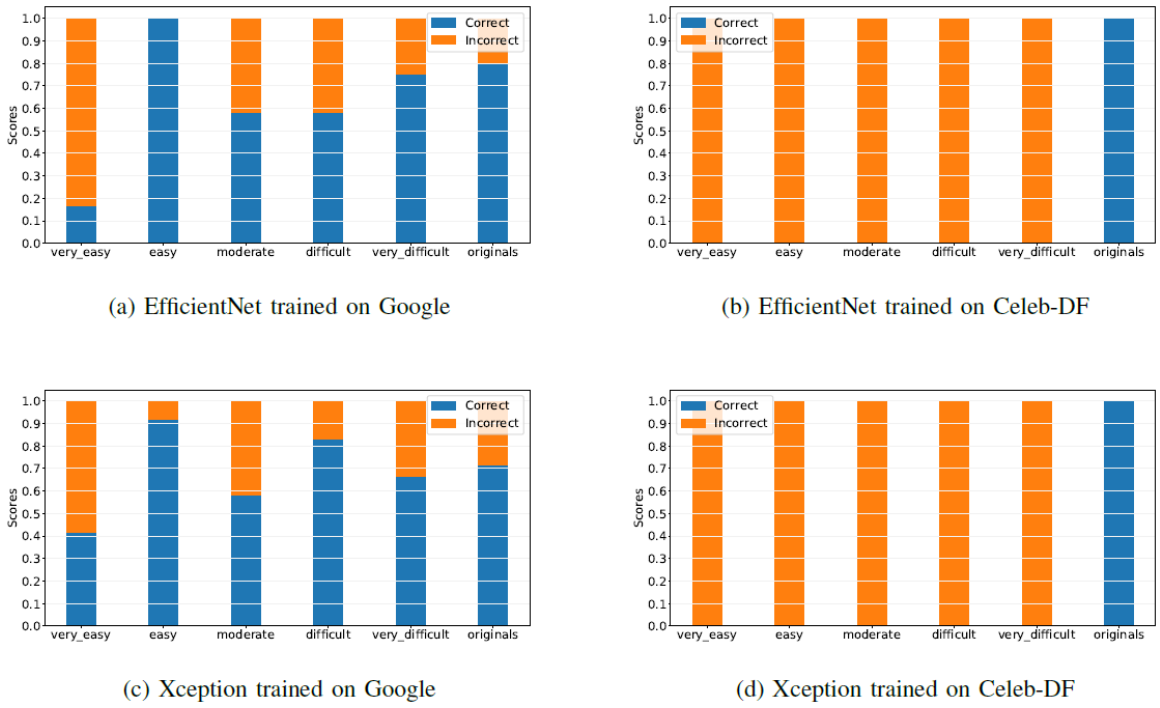


Ilustración 5. Resultados de la clasificación por parte de humanos. Fuente: (Korshunov & Marcel, 2020)

Por otra parte, esto se puede comparar con los resultados de los algoritmos, mostrados en la Ilustración 6. Como se puede ver, existe de hecho una menor precisión en la clasificación de videos que un humano fácilmente identifica como falsos, habiendo una alta precisión en videos muy difíciles. Además, cabe destacar que los modelos entrenados sobre Celeb-DF esencialmente están clasificando todos los videos como verdaderos, demostrando una baja capacidad de generalización.

Además, cabe destacar que los cuatro modelos mostrados obtuvieron un AUC de entre 70 y 74%, mientras que los sujetos humanos obtuvieron un AUC del 87.47%.



(a) EfficientNet trained on Google

(b) EfficientNet trained on Celeb-DF

(c) Xception trained on Google

(d) Xception trained on Celeb-DF

Ilustración 6. Resultados de la clasificación por parte de los modelos artificiales. Fuente: (Korshunov & Marcel, 2020)

2.3.3. Herramientas comerciales existentes

A raíz de las técnicas que se han expuesto anteriormente, han surgido una serie de estrategias para intentar combatir la desinformación. La primera que se va a comentar es Sensity, una compañía de inteligencia artificial que intenta combatir la desinformación. Tienen a disposición del público una plataforma en la que los usuarios pueden comprobar si una imagen o video han sido manipulados. Es una plataforma de pago, pero permite probar cinco videos para que sean clasificados. (Sensity, n.d.)

Lo primero que se prueba es el famoso *deepfake* de Obama[4], en el que se le ve aparentemente diciendo cosas que nunca diría en público. Este video se clasifica correctamente en Sensity como un video manipulado. Viendo este buen resultado, se decide probar con un video real y aleatoriamente seleccionado para comprobar que la clasificación no está siendo aleatoria. El resultado es bastante más decepcionante, como se puede ver en la Ilustración 7. Aparece un mensaje de que este link no puede ser testeado porque no se tiene un registro de él.

[4] Se puede encontrar en: https://www.youtube.com/watch?v=cQ54GDm1eL0

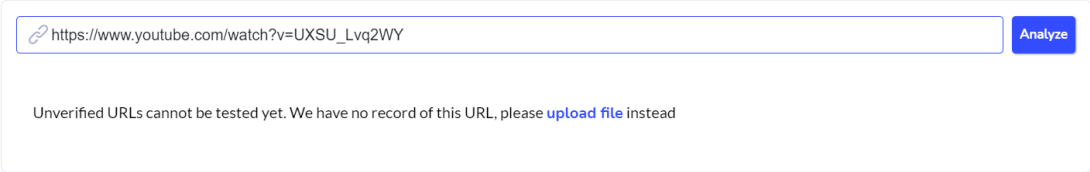


Ilustración 7. Video aleatorio testeado en la plataforma de Sensity. Fuente: elaboración propia

Parece que, por tanto, la parte de links de la plataforma no comprueba en vivo el link, sino que tienen un registro de links de videos manipulados. Se prueba por tanto a introducir un video descargado en local, en concreto un video real del *dataset* Celeb-DF. El resultado se puede ver en la Ilustración 8. Como se puede ver, se clasifica el video como manipulado, con un 71.8% de certeza. Esta es una certeza bastante alta para un video mal clasificado.

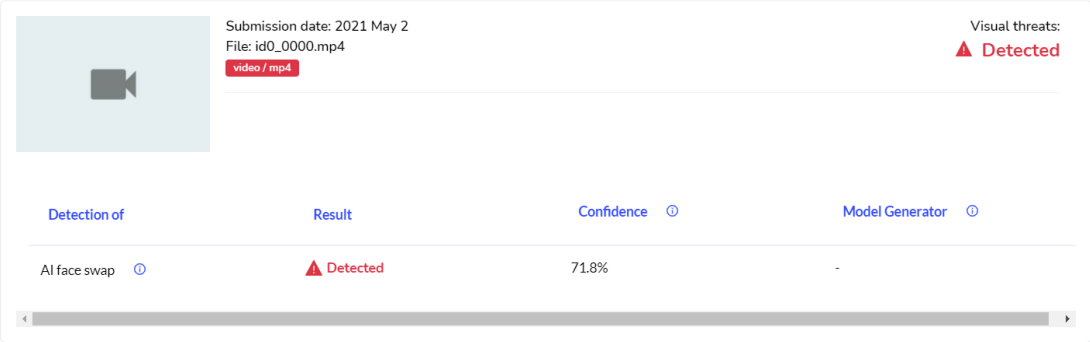


Ilustración 8. Video real de Celeb-DF en la plataforma de Sensity. Fuente: elaboración propia

Dado que no se puede hacer un experimento más amplio, se han intentado buscar datos de precisión en la web de Sensity, pero no se ha encontrado dicha información. Por lo tanto, no se puede juzgar la eficacia de la herramienta de Sensity más formalmente. Sólo se puede juzgar su interfaz visual como sencilla de utilizar.

Por otro lado, Microsoft también va a lanzar una herramienta llamada *Microsoft Video Authenticator*, que es una herramienta de detección de *deepfakes*, principalmente orientada a periodistas. Esta herramienta la está lanzando en colaboración con *Reality Defender* 2020, un esfuerzo no comercial para combatir la desinformación. (Kelion, 2020)

Una vez más, no hay datos públicos de la precisión que tiene esta herramienta, y tampoco se puede juzgar su facilidad de uso ya que no es accesible al público general. Lo único que se sabe es que se ha entrenado con FaceForensics++.

2.4. Conclusión Estado del Arte

Pese a que los *deepfakes* han cobrado importancia solo recientemente, se puede ver que ya hay varios trabajos enfocados a la detección de estos. Estos métodos son variados y han

supuesto un gran avance en esta área. También están apareciendo las primeras herramientas comerciales para detectar *deepfakes*. Sin embargo, cabe destacar dos cosas. La primera se puede ver en la Tabla 1, que consiste en un recopilatorio por parte de (Li et al., 2020) de los AUC de distintos modelos del estado del arte en diferentes *datasets* de *deepfakes*. Como se puede observar, los *fakes* de primera generación se detectan con relativa facilidad, con AUCs cercanos al 100%. Sin embargo, en *datasets* más avanzados los resultados son bastante peores, agravándose estos resultados en *datasets* sobre los que no se ha entrenado.

Tabla 1. AUC (%) de las principales técnicas de detección de Deepfakes para distintos datasets. Fuente: (Li et al., 2020)

Methods↓ Datasets→	UADFV [53]	DF-TIMIT [25]		FF-DF [40]	DFD [15]	DFDC [14]	**Celeb-DF**
		LQ	HQ				
Two-stream [54]	85.1	83.5	73.5	70.1	52.8	61.4	53.8
Meso4 [6]	84.3	87.8	68.4	84.7	76.0	75.3	54.8
MesoInception4	82.1	80.4	62.7	83.0	75.9	73.2	53.6
HeadPose [53]	89.0	55.1	53.2	47.3	56.1	55.9	54.6
FWA [28]	97.4	99.9	93.2	80.1	74.3	72.7	56.9
VA-MLP [33]	70.2	61.4	62.1	66.4	69.1	61.9	55.0
VA-LogReg	54.0	77.0	77.3	78.0	77.2	66.2	55.1
Xception-raw [40]	80.4	56.7	54.0	**99.7**	53.9	49.9	48.2
Xception-c23	91.2	95.9	94.4	99.7	**85.9**	72.2	65.3
Xception-c40	83.6	75.8	70.5	95.5	65.8	69.7	**65.5**
Multi-task [34]	65.8	62.2	55.3	76.3	54.1	53.6	54.3
Capsule [36]	61.3	78.4	74.4	96.6	64.0	53.3	57.5
DSP-FWA	**97.7**	**99.9**	**99.7**	93.0	81.1	**75.5**	64.6

Este mismo problema se ha visto recientemente en la *Deepfake Detection Challenge*, donde el modelo con mejor puntuación en el *dataset* público obtuvo un *Average Precision* (AP) de 82.56%, pero el mejor modelo en el *dataset* privado (oculto o *black-box*), que no fue el mismo que el mejor del *dataset* público, obtuvo un AP de 65.18% (Facebook, 2020). Por lo tanto, se puede ver que la principal problemática en la actualidad es aumentar la capacidad de generalización de los modelos.

El otro punto que cabe destacar es que las técnicas de generación de *deepfakes* están evolucionando todos los días, aumentando la capacidad de generar *fakes* más realistas y con menores tiempos de entrenamiento. Esto significa que las técnicas de detección de *deepfakes* también deben evolucionar todos los días, mejorando a la par que las de generación o incluso intentando adelantarse.

Tras analizar el estado del arte, se considera que la línea de investigación más prometedora y relativamente menos explorada es la de la utilización de audio e imagen para la detección de *deepfakes*. Sin embargo, lo que se ha encontrado es que en trabajos anteriores se ha utilizado DF-TIMIT y DFDC para entrenar video y audio. Se considera que este acercamiento no es correcto, ya que no todos los videos clasificados como *fake* tienen audio manipulado, por lo que la información que se pasa al modelo de audio es incorrecta, resultando en una

capacidad de discriminación muy baja. En este trabajo se plantea que este entrenamiento debe ir por separado, con audio y video bien clasificados.

No es el objetivo de este trabajo crear una herramienta comercial de detección de *deepfakes* ni competir con empresas como Sensity o Microsoft. No se puede juzgar la precisión de estas herramientas y además no es el objetivo diseñar una interfaz visual. Sin embargo, se considera que para mantenerse actualizado en la detección de *deepfakes* es necesario un esfuerzo colaborativo en el que cada propuesta construya sobre la anterior, por lo que se quiere poner a disposición de usuarios con conocimientos mínimos de Python una sencilla herramienta con la que puedan usar el modelo para detectar *deepfakes*. Esto ayudará a combatir la desinformación democratizando estas herramientas y haciéndolas accesibles al público. Además, potencialmente aumentará el interés general en este campo, atrayendo más investigadores que aporten al esfuerzo continuo de mejorar los modelos de detección de *deepfakes*.

3. Objetivos y metodología de trabajo

3.1. Objetivo general

El objetivo de este trabajo es el desarrollo de un modelo de detección de *deepfakes* basado en video y audio que permita detectar dichas manipulaciones con una precisión mayor al estado del arte, evaluando dicho modelo sobre un *dataset* oculto y poniéndolo posteriormente a disposición de usuarios con conocimientos mínimos de Python (Van Rossum & Drake Jr, 1995) que puedan utilizarlo para detectar videos falsos. Se va a centrar este trabajo en la detección de *deepfakes* de humanos.

3.2. Objetivos específicos

Más concretamente, este trabajo pretende conseguir los siguientes objetivos específicos:

- Analizar las técnicas existentes de detección de *deepfakes*
- Determinar entre estas el acercamiento más apropiado mediante una evaluación a alto nivel
- Desarrollar, en base a lo anterior, un modelo capaz de detectar *deepfakes*
- Evaluar dicho modelo sobre un *dataset* oculto para verificar la capacidad de generalización
- Facilitar el acceso al modelo para usuarios no técnicos

3.3. Metodología del trabajo

Para alcanzar los objetivos específicos, y por tanto el general, se deben seguir una serie de pasos:

1. Evaluar el estado del arte
2. Definir la arquitectura a utilizar para la detección de *deepfakes*
3. Implementar y entrenar el modelo
    a. Seleccionar el *framework* que se va a utilizar para implementar el código

   b. Elegir los datos de entrada para el modelo e implementar el código para pre-procesarlos
   c. Decidir las *augmentations* que se van a realizar sobre los datos
   d. Implementar el modelo y entrenarlo
   e. Validar el modelo
4. Demostrar la capacidad de generalización del modelo mediante experimentación
5. Habilitar el acceso al modelo para inferencia a usuarios de Python

4. Desarrollo del modelo

4.1. Preparación de los datos - video

Como se ha podido ver en el capítulo del estado del arte, existen dos posibles acercamientos al problema de detección de *deepfakes* en videos, que se pueden basar en *frames* individuales o en *features* temporales. En algunos casos, el segundo acercamiento ha dado mejores resultados que el primero. Sin embargo, esto es a costa de usar redes recurrentes o convoluciones 3D, lo cual es computacionalmente mucho más costoso. Debido a que también se han obtenido buenos resultados con los acercamientos basados en *frames* y que se busca utilizar una arquitectura ligera que luego puedan utilizar usuarios con un tiempo de inferencia razonable, en este proyecto se va a utilizar este tipo de modelo.

El principal objetivo es mejorar la capacidad de generalización de la red, por lo que el sobreajuste se debe evitar. Se extraerán *frames* individuales de los distintos videos y se utilizarán estos como entrenamiento.

4.1.1. *Face-cropping*

Los algoritmos de creación de *deepfakes* de humanos alteran principalmente la zona de la cara. El resto de la imagen se mantiene inalterada. Se considera adecuado, por tanto, ayudar al modelo a focalizarse en esta zona por medio de utilizar recortes de las caras de los *frames* para entrenar y predecir, en lugar de utilizar la imagen entera.

Lo primero que es necesario para poder extraer recortes de las caras es detectarlas. Existen varios tipos de método para este propósito (M.-H. Yang et al., 2002):

1. Basados en conocimiento: se definen una serie de reglas que definen una cara humana (una nariz, dos ojos, etc.).
2. *Feature invariant*: busca encontrar características estructurales que definen una cara independientemente de la postura, la iluminación, etc. Su uso es principalmente para localizar caras.
3. *Template matching*: se guardan patrones de caras que se comparan con la imagen en cuestión para determinar si hay una cara y dónde.
4. Basados en apariencia: se crean modelos basados en imágenes de entrenamiento que posteriormente se usan para detectar caras

En la actualidad, la mayor parte de la investigación se está centrando en el cuarto tipo de método. En la Ilustración 9 se puede ver la evolución de los algoritmos de detección de caras

a lo largo de los años, basados en apariencia. Como se puede ver, la evolución en esta área es constante.

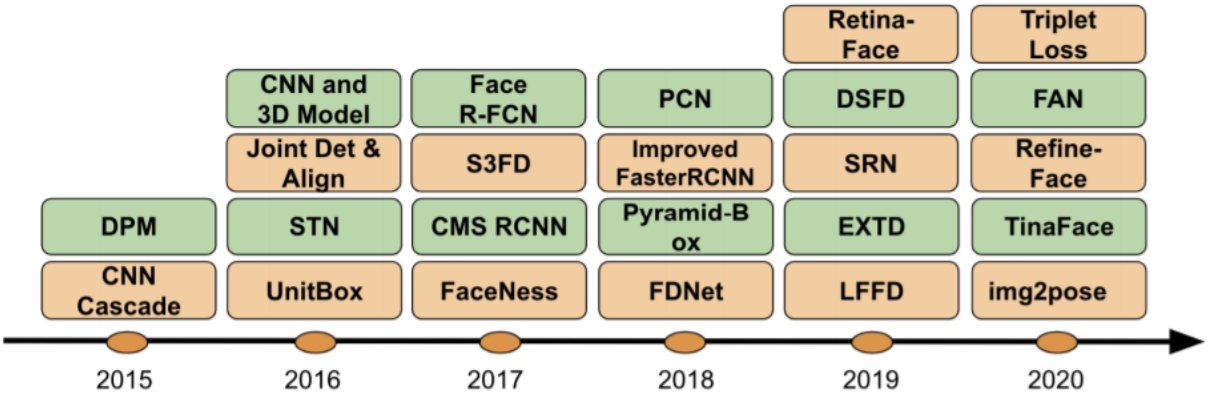


Ilustración 9. Algoritmos de detección de caras del 2015 al 2020. Fuente: (Minaee et al., 2021)

Realizar una comparativa entre los distintos métodos de detección de caras es ligeramente complicado, ya que no hay un único *benchmark* con el que todos los métodos realicen su evaluación. Sin embargo, los requerimientos que un detector de caras debe cumplir para este trabajo, en orden de importancia, son:

1. Rapidez y ligereza: el detector de caras no es la parte principal del modelo, sino que se busca que la mayor parte de los recursos estén dedicados a la clasificación en video falso o verdadero.

2. Precisión (ver ecuación (2)): la localización de la cara debe ser correcta para asegurar que se está alimentando el modelo con imágenes correctas focalizadas en la zona de mayor manipulación. Es preferible que el detector no detecte algunas caras (falso negativo (FN)) y ese frame no se utilice en la predicción a que detecte algo que no es una cara (falso positivo (FP)). Es por esto que se valora la precisión por encima de la exhaustividad.

3. Exhaustividad (ver ecuación (3)): es importante que se detecten las caras en la mayoría de los frames, particularmente de cara a la clasificación de un video en la fase de *testing*. Esta métrica también es importante, aunque menos que la anterior.

Para cumplir con estos requisitos, se evalúan distintos detectores del estado del arte. Ya que no se puede hacer una comparativa entre todos los algoritmos por métricas de un único *dataset*, se evalúan distintas fuentes para poder compararlos. En la Tabla 2, la Tabla 3, la Tabla 4 y la Ilustración 10 se puede ver la información utilizada para dicha comparativa.

Tabla 2. mAP de distintos detectores en el Face Detection Dataset and Benchmark y velocidad de inferencia en frames per second. "Ours" se refiere al algoritmo FaceBoxes. Fuente: (Zhang et al., 2017)

Approach	CPU-model	mAP(%)	FPS
ACF [40]	i7-3770@3.40	85.2	20
CasCNN [16]	E5-2620@2.00	85.7	14
FaceCraft [26]	N/A	90.8	10
STN [4]	i7-4770K@3.50	91.5	10
MTCNN [45]	N/A@2.60	94.4	16
Ours	E5-2660v3@2.60	**96.0**	**20**

Tabla 3. AP de distintos algoritmos en el dataset Wider Face. Fuente: (Y. Zhu et al., 2020)

Method	Backbone	Val Easy	Val Medium	Val Hard	Test Easy	Test Medium	Test Hard
AInnoFace [53] †	ResNet-152	0.970	0.961	0.918	0.965	0.957	0.912
RetinaFace [6] †	ResNet-152	0.969	0.961	0.918	0.963	0.956	0.914
RefineFace [57] †	ResNet-152	0.972	0.962	0.920	0.966	0.958	0.914
ASFD-D6 [51] †	ResNet-152	**0.972**	**0.965**	0.925	**0.967**	**0.962**	0.921
HAMBox [23] †	ResNet-50	0.970	0.964	0.933	0.959	0.955	0.923
TinaFace **(ours)**	ResNet-50	0.963	0.957	0.930	0.952	0.947	0.921
TinaFace **(ours)** †	ResNet-50	0.970	0.963	**0.934**	0.958	0.953	**0.924**

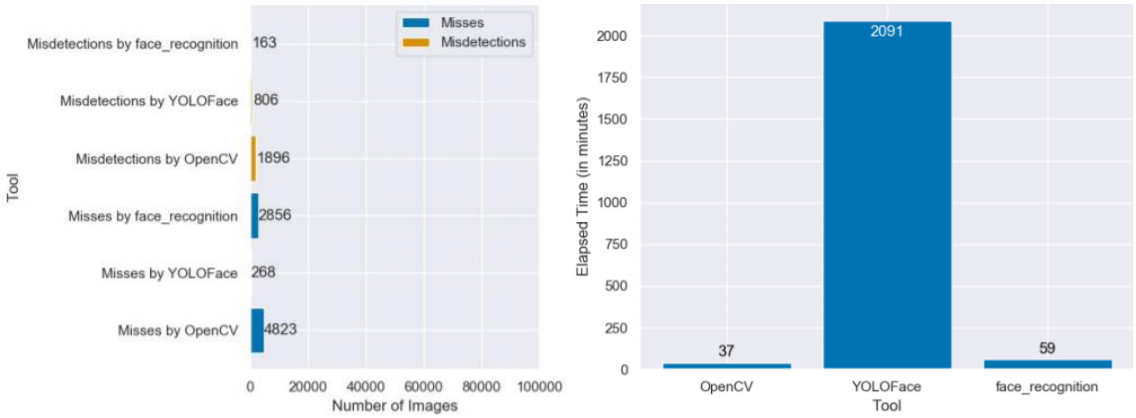


Ilustración 10. Falsos Positivos (misdetections) y Falsos Negativos (misses) de distintos algoritmos en CelebA (iquierda). Tiempos de detección por cada 100000 imágenes (derecha). Fuente: (Kabakus & others, 2019)

Tabla 4. Tiempos de inferencia por frame para distintos algoritmos. Fuente: (Rosaj, 2020)

Model	Seconds per frame
CenterFace	1
DSFD	48
FaceBoxes	0.3
Haarcascade	1
MTCNN	4
RetinaFace	90
S3FD	20
YoloFace	10

Como se puede ver, *FaceBoxes* (0.3 s/frame, ver Tabla 4) y *face_recognition* (0.04 s/frame, ver Ilustración 10[5]) son los dos algoritmos más rápidos y con buenas precisiones comparando con el estado del arte, aunque las precisiones no se pueden comparar entre los dos ya que se han evaluado sobre distintos *datasets*. Realmente, como se puede observar, la mayoría de los detectores dan buenos resultados, por lo que se va a priorizar la velocidad de inferencia. Además, cabe destacar que *face_recognition* es una librería basada en dlib, la cual no está oficialmente soportada en Windows. Es cierto que se puede conseguir que funcione, pero dado que los resultados son muy buenos con otros modelos, se considera que es preferible usar un modelo compatible con todos los sistemas operativos. Es por esta razón que se va a utilizar *FaceBoxes* (Zhang et al., 2017) como algoritmo de detección de caras.

4.1.2. Datos y *augmentations*

Hay disponible una gran variedad de *datasets* con *deepfakes* para entrenar modelos en su detección. Como se ha comentado, se van a utilizar *frames* individuales como entrenamiento para la red. Dado que se dispone de muchos datos, se quieren utilizar pocos *frames* por video para evitar que el modelo sobreajuste a videos o sujetos concretos. Se extraerán, por tanto, únicamente 10 *frames* aleatorios por video.

En este proyecto se han elegido los siguientes *datasets* del conjunto disponible:

- *DeepfakeTIMIT* (Korshunov & Marcel, 2018; Sanderson & Lovell, 2009): Este *dataset* es de primera generación y está compuesto por 320 videos falsos de alta calidad y 320 de baja calidad. En estos videos aparecen 32 sujetos diferentes, con 10 videos de

[5] $59\ {}^{min}/_{100000\ frames} \cdot 60\ s = 0.04\ {}^{s}/_{frame}$

cada calidad para cada sujeto. Se parte del *dataset VidTIMIT*, que es un *dataset* con 430 videos reales de 43 sujetos. A partir de estos, se han seleccionado 16 parejas de sujetos de aspecto parecido para generar los videos falsos de *DeepfakeTIMIT* utilizando *faceswap-GAN*. En los videos, el sujeto ocupa una porción importante de la imagen y tiene poca variabilidad en su postura. La iluminación también es bastante consistente. Se puede ver un ejemplo en la Ilustración 11.

Ilustración 11. Ejemplos de DeepfakeTIMIT en alta calidad (izquierda) y baja calidad (derecha). Fuente:(Korshunov & Marcel, 2018)

- *FaceForensics*++ (Rössler et al., 2019): Se compone de 1000 videos originales extraídos de YouTube. Los videos están filtrados para garantizar que las caras de los sujetos son completamente visibles y poco rotadas, ya que estos son los videos en los que las técnicas de manipulación funcionan mejor. A partir de estos videos, se generan 4000 videos manipulados con 4 técnicas distintas: *FaceSwap* y *DeepFakes*, que son técnicas de *face swapping*, *Face2Face* como técnica de *head puppetry*, y *NeuralTextures*, que los autores utilizan únicamente para *lip syncing*. Se utilizarán todos los videos originales, pero se seleccionarán aleatoriamente 400 videos manipulados de cada tipo para evitar el sobre ajuste a los mismos sujetos en múltiples videos y para intentar igualar en cierta manera el número de datos de las dos clases. Se pueden ver ejemplos de este *dataset* en la Ilustración 12. Como se podrá ver, los videos con más artefactos corresponden a técnicas de *face swapping*, ya que se intenta insertar la cara de una persona en la de otra.

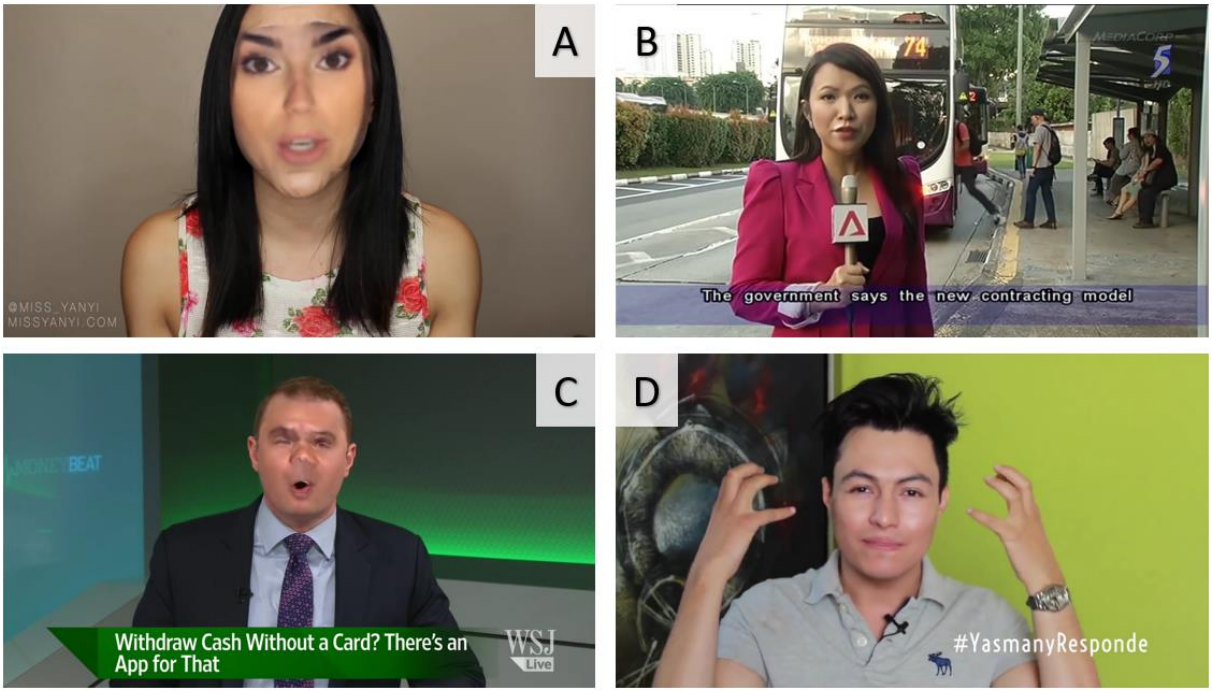


Ilustración 12. Ejemplos de FaceForensics++: DeepFake (A), Face2Face (B), Faceswap (C), NeuralTextures (D). Fuente: (Rössler et al., 2019)

- *Celeb*-DF-v2 (Li et al., 2020): Compuesto por 890 videos reales extraídos de YouTube, 590 de los cuales son videos de personas famosas. A partir de los videos de personas famosas, se han generado 5639 videos manipulados utilizando un algoritmo mejorado de *face swapping*. Los videos también aseguran que la cara no se oculta en ningún momento, aunque con una variedad de posturas. En el caso de este *dataset*, se utilizará sólo una parte de los videos manipulados (2000 videos), ya que las dos clases están fuertemente desequilibradas. Se plantea la posibilidad de aumentar esta cantidad si se viese necesario durante el entrenamiento, pero, como se verá más adelante, esto no termina siendo necesario. Se puede ver un ejemplo de este *dataset* en la Ilustración 13. Por lo general, los *deepfakes* generados son de calidad bastante buena, con pocos artefactos.

Ilustración 13. Ejemplo de Celeb-DF-v2. Fuente: (Li et al., 2020)

- *DeepFake Detection Challenge* (Dolhansky et al., 2020): Este *dataset* está compuesto de 104500 videos falsos y 23654 videos reales de 960 actores pagados. Los videos tienen gran variabilidad en cuanto a la calidad del video, la iluminación, la cercanía del actor a la cámara y su postura. Los videos falsos han sido generados con dos técnicas de generación de *deepfakes*, aunque estas técnicas no son públicas con el objetivo de conseguir que los modelos no sobre-ajusten a los artefactos de técnicas concretas. En el artículo de este *dataset*, se considera que este *dataset* pertenece a una tercera generación de *datasets*, diferenciado por el tamaño y la longitud de los videos. Se pueden ver ejemplos de este *dataset* en la Ilustración 14.

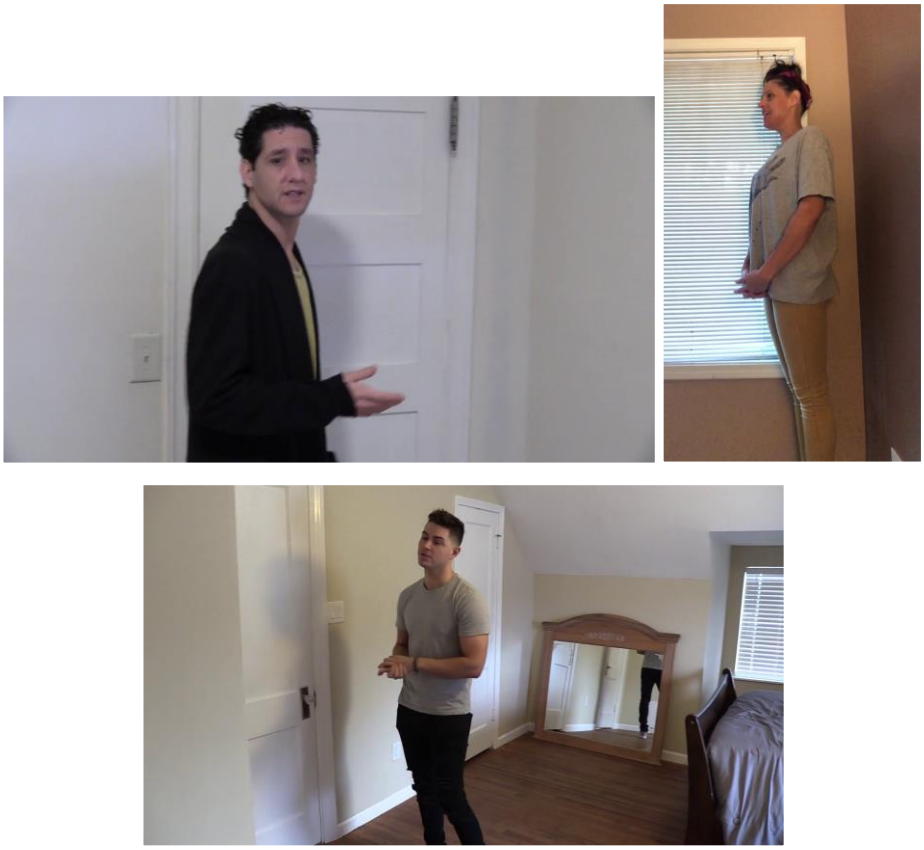


Ilustración 14. Ejemplos de DFDC. Fuente: (Dolhansky et al., 2020)

Estos *datasets* se van a utilizar para el entrenamiento y la validación del modelo de video. Por otro lado, el testeo se va a realizar con los siguientes datos:

- *DeeperForensics1.0* (Jiang et al., 2020): Este *dataset*, también considerado de tercera generación en lo que a su tamaño respecta, está compuesto de 60000 videos reales y falsos, con una razón de real a falso de 5 a 1. Los videos fuente se graban a partir de 100 actores pagados. Se utiliza un *variational auto-encoder* para generar los videos falsos. Se utilizará únicamente una parte de estos datos como set de testeo oculto.

Durante el entrenamiento, para mejorar la capacidad de generalización del algoritmo, se van a utilizar una serie de *augmentations*. En (P. Chen et al., 2020), se dividen los tipos de *augmentations* de imágenes en transformaciones espaciales, distorsiones de color y eliminación de información (*information dropping*). En este proyecto se van a utilizar:

- Transformaciones espaciales:
  - Traslación
  - Escalado
  - Rotación

    - Espejo horizontal
    - Redimensionar (al tamaño necesario como input de la red)
- Distorsiones de color:
    - Cambios de iluminación
    - Cambios de contraste
    - *FancyPCA*: se realiza *Principal Component Analysis* sobre el set de pixeles RGB de *ImageNet*. A cada imagen se le añaden múltiplos de estos componentes multiplicados por una variable aleatoria con el mismo valor para cada *batch*. (Krizhevsky et al., 2012)
    - Transformación a escala de grises
    - Normalización (con los parámetros de media y desviación estándar de ImageNet)
- Eliminación de información:
    - Compresión de imagen
    - Ruido Gaussiano
    - *Grid Dropout*: se ponen grupos de píxeles a negro en un patrón de rejilla, demostrando mejores resultados que alternativas como *AutoAugment* y *Cutout*. (P. Chen et al., 2020)

Para la fase de validación y testeo, únicamente se realizarán sobre las imágenes las operaciones de redimensionamiento y normalización.

4.2. Preparación de los datos - audio

Como se vio en la sección de estado del arte, los acercamientos anteriores a detección de *deepfakes* por audio y video utilizaban *datasets* como el DFDC, en los cuales el audio no necesariamente está manipulado, dando lugar a clasificadores de audio esencialmente aleatorios. En este trabajo se quiere entrenar el clasificador de audio con información correcta, es decir, que el audio que se utilice para entrenar como audio falso sea realmente sintético; de otra forma el clasificador no será capaz de discriminar correctamente. Es por esto que el clasificador de audio se va a entrenar por separado con el *dataset* del ASVspoof2019, en

concreto con la parte del *dataset* que corresponde al *logical access* (LA), que está formado por audios reales y sintéticos, estos últimos generados por técnicas del estado del arte de *text-to-speech* (TTS, conversión de texto escrito en voz natural) y *voice conversion* (VC, modificar la voz de una persona (*source*) para que suene como la de otra (*target*)). (Yamagishi et al., 2019)

Como se mencionaba anteriormente, en este *dataset* se utilizan técnicas distintas para generar los audios de entrenamiento y validación respecto de las de los audios de testeo. Además, las tres particiones están compuestas de audios de sujetos distintos, lo cual permite evaluar que el modelo no se está sobre ajustando a sujetos concretos. Se considera, por tanto, que este *dataset* será una buena forma de comprobar la capacidad de generalización.

4.2.1. *Features* de audio

En (Das et al., 2019), se realiza un análisis de distintas formas de representación del audio para el entrenamiento de redes neuronales en el ASVspoof 2019. Estas formas de representación se dividen en:

- *Long range features*: características que se extraen mediante transformaciones con una ventana de tiempo larga. Estos métodos son derivaciones de la *constant-Q transform*, que “proporciona mayor resolución frecuencial a las frecuencias más bajas y mayor resolución temporal a las frecuencias más altas” (Das et al., 2019) respecto a la transformada de Fourier.

- *Deep embeddings*: se entrena una red neuronal a partir de la transformada discreta de Fourier (TDF) de la señal y su conversión a escala logarítmica, con el objetivo de que clasifique el audio como real o manipulado. Posteriormente, extrayendo la última capa de la red, que actúa como clasificador, se obtiene una red que transforma una entrada de audio en un *embedding*.

- *Contrast features*: otras técnicas de representación de audio como son los *Mel-frequency cepstral coefficients* (MFCC) o *Linear frequency cepstral coefficients* (LFCC). Estas técnicas se basan en la discretización del espectro de potencia mediante bancos de filtros, aplicando también compresión logarítmica. Posteriormente, se utiliza la transformada discreta del coseno (TDC) para obtener el *cepstrum*. Se puede ver gráficamente esta conversión en la Ilustración 15. Las técnicas mencionadas difieren en el tipo de filtro que se utiliza, donde MFCC utiliza filtros en la escala Mel y LFCC utiliza filtros triangulares linealmente distribuidos en todas las frecuencias (véase la Ilustración 16).

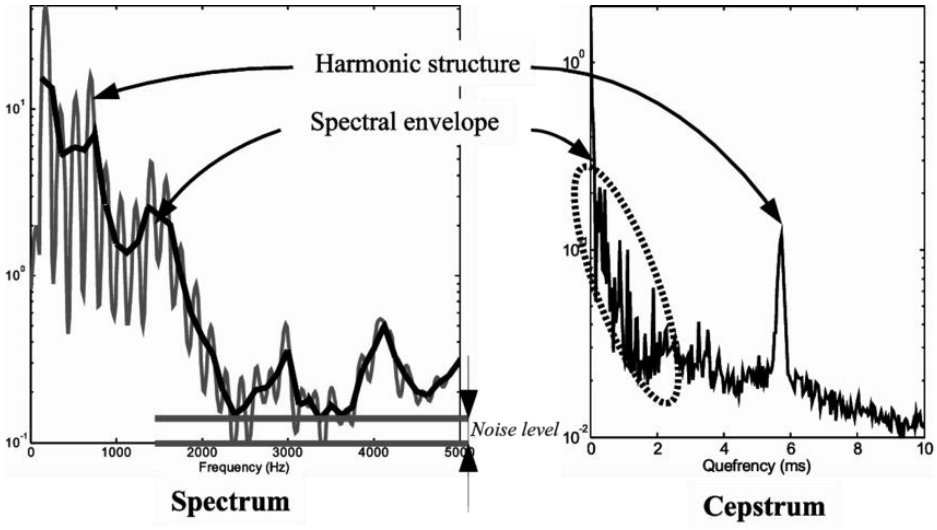


Ilustración 15. Representación gráfica de la conversión del spectrum *al* cepstrum*. El* spectral envelope *corresponde a la representación de los fonemas concretos, es decir, "qué se dice", mientras que la* harmonic structure *corresponde al denominado pulso glotal, definido por las características físicas del tracto vocal de la persona, es decir, "quién lo dice". Fuente:* (Fraile et al., 2009)

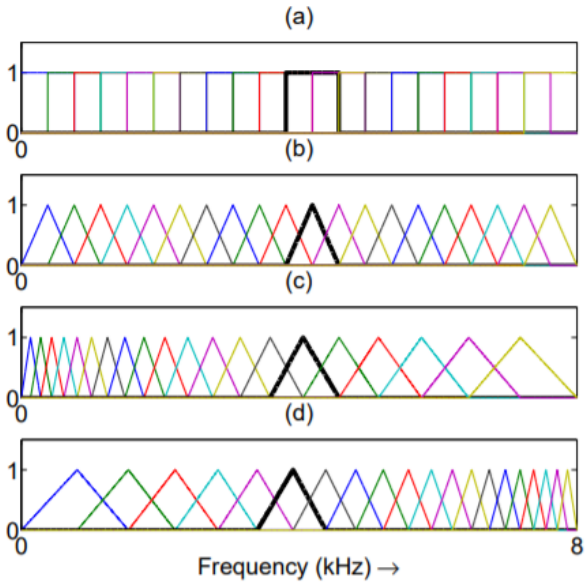


Ilustración 16. Diferentes tipos de filtros utilizados en la transformación al cepstrum*. (a) Rectangular (RFCC), (b) Lineal (LFCC), (c) Escala Mel (MFCC), (d) Escala Mel Invertida (IMFCC). Fuente:* (Sahidullah et al., 2015)

En el artículo mencionado, se utilizan estos métodos para representar el audio, utilizando estas representaciones para entrenar un modelo de mezclas gaussiano[6] (GMM) por un lado y una red neuronal (DNN) por otro, obteniendo los siguientes resultados:

[6] Modelo de *clustering* que incorpora información sobre la covarianza al modelo de k-means.

Tabla 5. Resultados obtenidos en ASVspoof 2019 con las diferentes técnicas de representación de audio en el set de validación (izquierda) y el de evaluación (derecha). Fuente: (Das et al., 2019)[7]

Front-ends	Back-ends			
	GMM		DNN	
	EER (%)	t-DCF	EER (%)	t-DCF
CQCC	0.431	0.012	**0.000**	**0.000**
CQ-EST	4.983	0.167	0.040	0.013
CQ-OST	3.845	0.129	0.042	0.001
eCQCC	0.941	0.025	**0.000**	**0.000**
CQSPIC	4.428	0.143	0.039	0.013
ICQCC	0.393	0.010	**0.000**	**0.000**
CMC	9.965	0.224	0.071	0.001
LFCC	2.827	0.077	1.569	0.046
IFCC	0.042	0.001	0.004	0.001
MFCC	7.062	0.169	12.087	0.221

Front-ends	Back-ends			
	GMM		DNN	
	EER (%)	t-DCF	EER (%)	t-DCF
CQCC	9.572	0.236	13.149	0.301
CQ-EST	10.263	0.251	17.131	0.422
CQ-OST	8.403	**0.201**	9.694	0.272
eCQCC	**7.043**	**0.160**	11.081	0.285
CQSPIC	**7.220**	**0.199**	9.585	0.262
ICQCC	**6.854**	**0.164**	12.481	0.295
CMC	11.748	0.266	13.368	0.335
LFCC	8.320	0.227	9.654	0.234
IFCC	10.402	0.298	16.275	0.362

Como se puede observar, la capacidad de generalización de estos métodos es muy baja. Por otro lado, en (Sahidullah et al., 2015) se analizan algunas de las anteriores técnicas de representación de audio, además de algunas adicionales. Se plantea que técnicas como MFCC, que se mencionaba en el capítulo del estado del arte, aunque son adecuadas para reconocimiento del habla, no lo son tanto para la detección de audio manipulado, siendo más adecuados los filtros lineales o de frecuencia Mel inversa. Además, plantean que el uso de coeficientes dinámicos, es decir, deltas y doble deltas (Δ-*Cepstrum* and Δ^2 -*Cepstrum*), puede ser más acertado, ya que las técnicas de VC y TTS no pueden modelar completamente las características temporales del habla. Esto lo demuestran experimentalmente, como se puede observar a continuación:

[7] *Equal Error Rate*: Punto en el que se cortan las curvas de ratio de falsa aceptación (*false acceptance rate*) y ratio de falso rechazo (*false rejection rate*), es decir, misma cantidad de falsas aceptaciones que de falsos rechazos. Es una métrica comúnmente utilizada en sistemas biométricos.

Tabla 6. Resultados (EER%) obtenidos en ASVspoof 2015 con las diferentes técnicas de representación de audio en el set de validación (izquierda) y el de evaluación (derecha). Fuente: (Sahidullah et al., 2015)

Feature	GMM-ML			LBP-SVM		
	Static	Static+ $\Delta\Delta^2$	$\Delta\Delta^2$	Static	Static+ $\Delta\Delta^2$	$\Delta\Delta^2$
RFCC	2.41	0.75	0.21	6.25	2.12	3.38
LFCC	2.46	0.66	0.12	5.05	1.56	2.37
MFCC	3.46	1.09	0.64	7.71	4.78	7.99
IMFCC	1.33	0.48	0.20	6.03	**1.50**	2.10
LPCC	2.44	0.68	0.14	5.44	2.47	3.94
PLPCC	2.95	1.61	1.51	9.09	5.48	8.07
SSFC	0.96	0.60	0.49	4.57	2.80	5.82
SCFC	1.77	**0.25**	**0.05**	23.43	1.87	**1.91**
SCMC	2.76	0.95	0.20	5.62	1.85	2.85
MGDF	4.71	2.24	2.69	7.15	3.81	7.41
APGDF	2.44	0.75	0.19	5.67	2.42	4.20
CosPhase	0.82	1.11	1.89	15.45	10.83	13.30
RPS	**0.21**	0.37	6.44	**2.45**	1.80	13.21
FDLP	5.71	2.18	1.99	12.17	6.50	9.44
MHEC	7.69	3.30	2.01	11.88	6.54	8.09
SDC-MFCC	4.37	-	-	-	7.06	-
ModSpec	4.41	-	-	-	5.92	-

	GMM-ML		LBP-SVM	
	Known	Unknown	Known	Unknown
MFCC (Static-$\Delta\Delta^2$)	0.83	5.17	4.35	17.18
RPS (Static)	0.10	10.51	**1.66**	20.04
RFCC ($\Delta\Delta^2$)	0.12	1.92	3.20	19.96
LFCC ($\Delta\Delta^2$)	0.11	**1.67**	2.13	19.45
MFCC ($\Delta\Delta^2$)	0.39	3.84	7.78	19.22
IMFCC ($\Delta\Delta^2$)	0.15	1.86	1.96	**9.97**
LPCC ($\Delta\Delta^2$)	0.11	2.31	3.54	13.90
SSFC ($\Delta\Delta^2$)	0.30	1.96	5.22	14.91
SCFC ($\Delta\Delta^2$)	**0.07**	8.84	1.81	17.54
SCMC ($\Delta\Delta^2$)	0.17	1.71	2.36	19.10
APGDF ($\Delta\Delta^2$)	0.16	2.34	3.74	13.10

Como se puede observar, la técnica de representación LFCC $\Delta\,\Delta^2$ es la que consigue la mejor capacidad de generalización, en particular con un modelo GMM con criterio *maximum likelihood*. Se puede observar también que los coeficientes estáticos empeoran los resultados, y se obtienen mejores *equal error rate* (EER) cuando sólo se usan los coeficientes dinámicos. Basado en este descubrimiento, se va a usar esta representación del audio como forma de reducir la dimensionalidad de la transformada de Fourier de tiempo reducido (STFT).

Una señal de voz durante un tiempo del orden de 5 a 100 ms se considera prácticamente estacionaria (Hasan et al., 2004). Este es el intervalo de tiempo que se va a plantear para la elección del tamaño de *frame*, ya que no se quiere capturar información sobre lo que la persona está diciendo, sino sobre si el audio está manipulado o no. Como punto de partida, se va a usar el tamaño de ventana de 32 ms[8], con una superposición entre ventanas de 10 ms, al igual que (Sahidullah et al., 2015), utilizando además los 20 primeros coeficientes, que es lo común al usar *cepstrum*. Dado que se van a utilizar solo los coeficientes dinámicos, esto supondrá tener 40 coeficientes por *frame*.

El procedimiento para obtener los LFCC a partir de la señal de audio es el siguiente:

[8] Corresponde a una longitud de ventana de 512 muestras para una frecuencia de muestreo de 16kHz. Es el número de muestras recomendado para el procesado de voz (Librosa, n.d.).

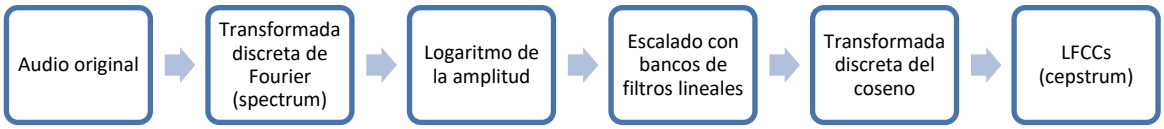


Ilustración 17. Procedimiento para obtener los LFCCs. Fuente: elaboración propia

Los filtros lineales se pueden visualizar en la Ilustración 16 b y se expresan en forma matricial, donde la matriz tiene dimensiones (nº de bandas, nº de ventanas de la STFT / 2 + 1).

4.2.2. *Data augmentation*

Para mejorar la capacidad de generalización del modelo, se va a utilizar *data augmentation*. En (T. Chen et al., 2020) se plantean una serie de formas de aumentar el ASVspoof 2019:

- Reverberación a partir de respuestas al impulso de una sala (*room impulse responses* o RIR)[9]
- Ruido de fondo a partir de música, televisión, charla de fondo (*babble*), and *freesound*
- Simulación de audio en un centro de llamadas.
- Enmascaramiento frecuencial (poner bandas frecuenciales a 0), aleatoriamente y de forma dinámica en cada epoch

Dado que estas alteraciones dan excelentes resultados, se van a utilizar transformaciones similares en este trabajo. Adicionalmente, se van a utilizar las siguientes alteraciones para comprobar si mejoran aún más los resultados:

- Alteración del momento de inicio o fin. Esencialmente consiste en poner a 0 la amplitud durante x segundos del inicio o del final.
- Alterar el tono (pitch)
- Alterar la velocidad

Las alteraciones se realizarán de forma dinámica y aleatoria.

[9] Respuesta de un sistema a la función de impulso δ(t)

4.3. Arquitectura del modelo

La clasificación de audio y de video se van a hacer con dos redes independientes, y para ambos modelos se va a utilizar *transfer learning* (Bozinovski & Fulgosi, 1976). Se busca utilizar un modelo pre-entrenado con un *dataset* muy amplio, que haya "aprendido" representaciones de dicho *dataset*. Ese conocimiento posteriormente se puede utilizar para resolver problemas similares con un entrenamiento mucho más rápido.

4.3.1. Video

En (Tan & Le, 2019), se presenta una familia de modelos denominado *EfficientNet*. Este artículo está principalmente focalizado en plantear un método para el correcto escalado de redes convolucionales. Se plantea que la forma más eficiente de escalar dichas redes es aumentando proporcionalmente a un valor constante el número de filtros y el número de convoluciones a la vez, y no sólo aumentando uno de estos dos hiperparámetros. El modelo que presentan, basado principalmente en bloques *Mobile Inverted Bottleneck* (MBConv), obtiene mejores resultados en ImageNet que el estado del arte con muchos menos parámetros, como se puede ver en la Ilustración 18 (izquierda).

En marzo de 2021, los mismos autores han publicado una versión mejorada de esta familia de modelos que llaman *EfficientNetV2*, utilizando Fused-MBConv y *progressive learning* (aprendizaje progresivo), donde se empieza el entrenamiento de la red con imágenes sencillas y de pequeño tamaño y, según van avanzando los epochs, se añade más regularización, se aumenta el tamaño de imagen y se les realizan más *augmentations*. Se puede ver la mejora en precisión en la Ilustración 18 (derecha).

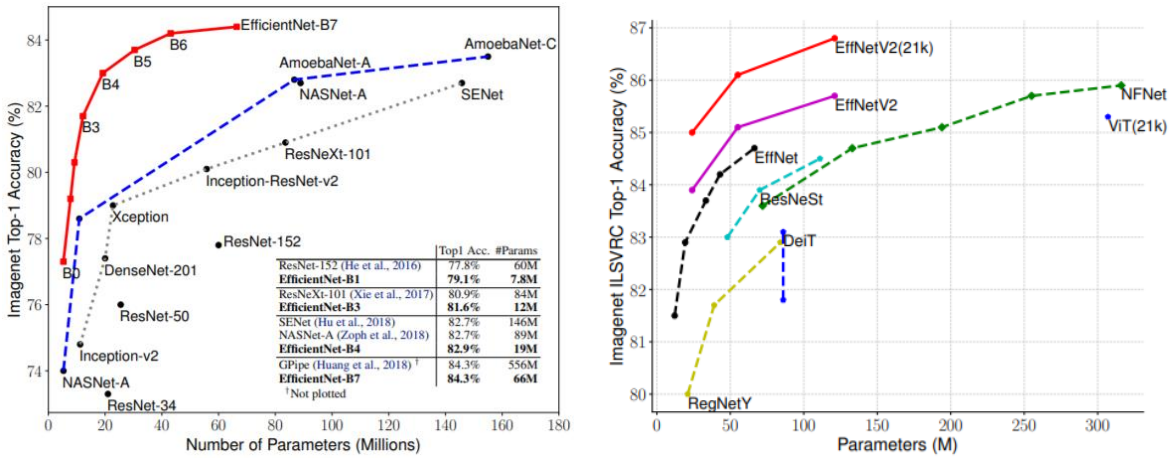


Ilustración 18. Comparativa de EfficientNet (izquierda, fuente: (Tan & Le, 2019)*) y EfficientNetV2 (derecha, fuente:* (Tan & Le, 2021)*) con el estado del arte.*

Dada la clara mejora en eficiencia y la alta precisión, se va a utilizar este modelo para hacer *transfer learning*, haciendo posteriormente *fine-tunning*.

En la ilustración Ilustración 19 se puede ver el resumen de la arquitectura de *EfficientNet* V2. Como se puede ver, está compuesto principalmente por bloques de convoluciones denominados *MBConv* y *Fused-MBConv*. Este segundo tipo de bloques es la diferencia en la arquitectura de la versión 1 y la de la 2. Se puede ver en qué consisten estos bloques en la Ilustración 20. Según los autores, el ratio de estos dos tipos de bloque es el correcto para lograr un buen equilibrio entre la mejora en la velocidad de entrenamiento de *Fused-MBConv* y el menor número de parámetros de *MBConv*.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Ilustración 19. Arquitectura de EfficientNet V2. Fuente: (Tan & Le, 2021)

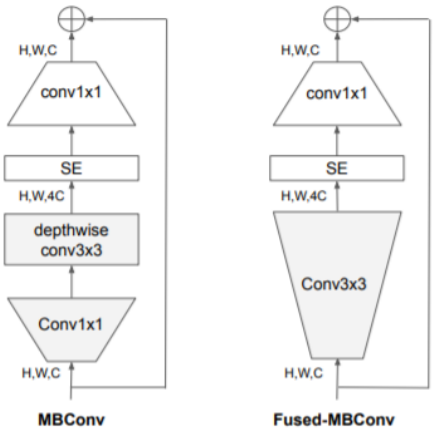


Ilustración 20. Bloque MBConv frente a Fused-MBConv. Fuente: (Tan & Le, 2021)

4.3.2. Audio

Como se ha explicado anteriormente, se van a utilizar LFCCs como representaciones de audio. Dado que esto es esencialmente una matriz, es común que estas representaciones se traten como imágenes (en concreto *heatmaps*) y se analicen utilizando arquitecturas de redes

neuronales como las convolucionales 2D (Koike et al., 2020; Palanisamy et al., 2020; Zhou et al., 2018). Sin embargo, es cierto que la clasificación de audio e imágenes son problemas muy distintos, por lo que un modelo pre-entrenado con *ImageNet* no necesariamente podrá clasificar audio correctamente.

En (Palanisamy et al., 2020; Zhou et al., 2018) se utilizan redes convolucionales pre-entrenadas con *ImageNet* para clasificar audio, obteniendo resultados satisfactorios. Se demuestra que el uso de *transfer learning* hace que se consigan mejores resultados que con una inicialización aleatoria de los pesos de distintas redes. Cabe destacar, por otro lado, que esta comparativa se hace, precisamente, contra una inicialización aleatoria de los pesos. No se compara el pre-entrenamiento con *ImageNet* frente al pre-entrenamiento con un *dataset* de audio. Esto se estudia en (Koike et al., 2020), donde se comparan arquitecturas de redes neuronales pre-entrenadas con *ImageNet* o con audio en la tarea de clasificación de señales de audio cardiacas. La red pre-entrenada con audio (Large-Scale Pretrained Audio Neural Networks (PANN)) obtiene los mejores resultados, pero se debe destacar que sólo hay una red pre-entrenada con audio en la comparativa y que *MobileNet*, pre-entrenada con *ImageNet* y con un orden de magnitud menos de parámetros que PANN, obtiene resultados relativamente cercanos. Se pueden ver dichos resultados en la Tabla 7. Se considera, por tanto, que esta evaluación no es del todo concluyente.

Tabla 7. Comparativa de los resultados del transfer learning con distintos modelos para tareas de audio. Fuente: (Koike et al., 2020)

		Test UAR [%]			
Model Name	# Params.	Log Mel		Spectrogram	
		mean	std.	mean	std.
VGG16	14.7	85.6	1.3	85.3	1.3
VGG19	20.0	86.0	0.9	85.7	1.3
MobileNet V2	2.2	86.1	2.2	84.5	2.1
ResNet18	11.1	82.1	10.3	74.9	8.6
ResNeXt-50	22.9	76.2	7.4	78.0	4.3
ResNeXt-101	86.7	65.3	1.6	79.9	1.3
+ WSL	86.7	71.0	2.5	59.9	9.2
PANNs CNN14	80.7	**89.7**	1.5	—	—

Dados estos resultados, se va a hacer un primer modelo utilizando MobileNetV2 (Sandler et al., 2018), dado que es una arquitectura más ligera y sencilla de implementar. Si está arquitectura no diese buenos resultados, se plantea la utilización de la red PANNs, pre-entrenada con audio. Como se verá más adelante, esto finalmente no acaba siendo necesario.

La arquitectura de *MobileNet* V2 se puede ver resumida en la Ilustración 21. Principalmente, está compuesta de bloques *bottleneck*, que se pueden ver en la Ilustración 22. Es como un

bloque residual invertido, manteniendo *skip-connections* para mejorar el flujo del gradiente, y sustituyendo las funciones de activación RELU por RELU6 (RELU con valor máximo de 6).

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Ilustración 21. Arquitectura MobileNetV2. Fuente: (Sandler et al., 2018)

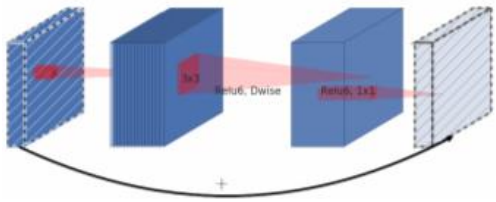


Ilustración 22. Bloque Bottleneck. Fuente: (Sandler et al., 2018)

4.3.3. Función de coste (*loss*)

Para poder entrenar los anteriores modelos se debe definir una función de coste con la cual juzgar la dirección en la que debe mejorar el modelo. Este problema se va a analizar como uno de clasificación binaria. Se considera que clasificar los distintos métodos de generación de *deepfakes* por separado no aportará valor al objetivo del trabajo, que es mejorar la capacidad de generalización.

Tradicionalmente, para problemas de clasificación binaria se ha utilizado la función de coste de entropía cruzada binaria (*binary cross-entropy*) (ecuación (5)). Sin embargo, algunos estudios recientes como (Wen et al., 2016) muestran que la capacidad de discriminación al utilizar esta función de coste es limitada. Se han planteado, por tanto, una serie de alternativas cuyo objetivo es, principalmente, maximizar la diferencia entre clases y minimizar la diferencia dentro de las clases. En el contrastive y *triplet loss*, se plantea la utilización de un margen en

el espacio euclídeo para maximizar la diferencia entre clases. En (Liu et al., 2016), plantea medir la similitud en un espacio angular, maximizando la distancia angular entre clases y minimizándola dentro de la clase. Este acercamiento tiene mayor consistencia con la función de *softmax*, dado que la distancia euclídea depende no sólo de la dirección sino también de la magnitud del vector, mientras que en el espacio angular se mide la similitud con independencia de la magnitud, lo cual es más coherente con la normalización que realiza *softmax*. Se denomina a esta función *large-margin sofmax loss* o L-softmax. Posteriormente, (Liu et al., 2017) introduce *angular softmax* (A-softmax), que normaliza los pesos para mejorar L-softmax.

$$Loss_{softmax} = \frac{1}{N} \cdot \sum_{i=1}^{N} \ln \frac{e^{f_{y_i}}}{\sum_{j=1}^{C} e^{f_j}} \tag{5}$$

Donde, asumiendo *bias = 0:*

$$f_j = \left\|W_j\right\| \cdot \|x\| \cdot \cos\theta_j \tag{6}$$

Tras la función A-softmax, (Wang et al., 2018) introduce el *Large Margin Cosine Loss* (LMCL), que utilizan para el reconocimiento de caras. Este se formula primeramente como una normalización de las matrices de entradas a la última capa (vector de representación) y de pesos de dicha capa, de tal forma que:

$$\left\|W_j\right\| = 1$$

$$\|x\| = s$$

Donde s es un hiperparámetro. Por medio de esta normalización, fuerzan a que la función de coste dependa exclusivamente del ángulo entre los vectores de pesos y entrada, traduciendo por tanto el problema al espacio angular. Sin embargo, como aún así la función de coste no es lo suficientemente discriminativa, introducen un margen (m) entre las representaciones vectoriales de las clases, que es otro hiperparámetro, de tal forma que la función final de coste se formula como se muestra en la ecuación (7). Además, se puede ver una representación gráfica de las representaciones que se aprenden con esta función en la Ilustración 23.

$$Loss_{LMCL} = \frac{1}{N} \cdot \sum_{i=1}^{N} \ln \frac{e^{s\cdot(\cos(\theta_{y_i,i})-m)}}{e^{s\cdot(\cos(\theta_{y_i,i})-m)} + \sum_{j\neq y_i} e^{s\cdot\cos(\theta_{y_i,i})}} \tag{7}$$

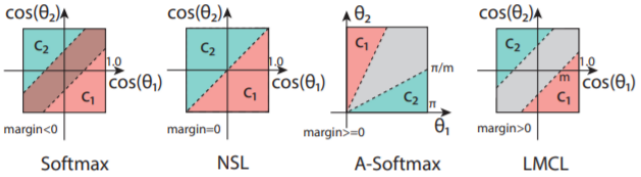


Ilustración 23. Espacios de representación para distintas funciones de coste. Fuente: (Wang et al., 2018)

Tanto para el modelo de audio como para el de video, se va a comparar la función LMCL con la *Softmax Loss* estándar.

4.3.4. Métricas

Para poder evaluar correctamente el modelo, se necesita elegir una métrica adecuada. En el capítulo del estado del arte se ha visto que distintos trabajos han usado distintas métricas:

- Exactitud (*accuracy*) (H. H. Nguyen et al., 2019; Sabir et al., 2019): definida por:

$$Accuracy = \frac{Predicciones\ correctas}{Predicciones\ totales} = \frac{TP + TN}{TP + FP + TN + FN} \quad (8)$$

  Lo cual devuelve un valor entre 0 y 1. Esta métrica es muy común en el ámbito de la clasificación. Sin embargo, en el caso de clases no balanceadas puede no ser el mejor indicador, ya que, si una clase tiene muchos aciertos y es la clase mayoritaria, enmascarará los fallos en la clase minoritaria.

- Área debajo de la curva (AUC) (Tolosana et al., 2020): Corresponde al área debajo de la curva de ratio de verdaderos positivos (TPR) frente al ratio de falsos positivos (FPR), definidos por:

$$TPR = \frac{TP}{FN + TP}$$

$$FPR = \frac{FP}{TN + FP}$$

  El AUC también devuelve un valor entre 0 y 1.

- EER (T. Chen et al., 2020): Punto en el que se cortan las curvas de ratio de falsa aceptación (f*alse acceptance rate* - FAR) y ratio de falso rechazo (*false rejection rate*

- FRR), es decir, misma cantidad de falsas aceptaciones que de falsos rechazos, donde:

$$FAR = \frac{Admisiones\ erróneas}{Nº\ de\ intentos\ de\ identificación}$$

$$FRR = \frac{Rechazos\ erróneos}{Nº\ de\ intentos\ de\ identificación}$$

Es una métrica comúnmente utilizada en sistemas biométricos. En este caso, el FAR sería un indicador del número de errores de tipo II (Webopedia, 2021), donde se aceptaría un video manipulado como verdadero.

Durante el entrenamiento del modelo, la función de coste será el mejor indicador del entrenamiento, más aun teniendo en cuenta que las métricas calculadas durante el entrenamiento con *batches* no dan una visión global precisa. Además, se utilizará la exactitud como métrica absoluta, ya que da un valor más fácilmente cuantificable (se sabe que el máximo es 1, cosa que no aplica para la función de coste, que sólo se mide de forma relativa a valores anteriores), pero únicamente como orientación adicional, monitorizando principalmente el *loss*.

Tras el entrenamiento, siempre que las clases estén balanceadas, se utilizará también la exactitud. Sin embargo, para clases desbalanceadas, se evaluará también la AUC, ya que es una métrica más adecuada para ámbitos con clases desbalanceadas.

5. Descripción de la herramienta software desarrollada

5.1. Módulo de pre-procesado

5.1.1. Video

Para la extracción de los *frames* y los recortes de las caras, se va a utilizar la librería *OpenCV* (OpenCV, 2021), que permite de forma sencilla las operaciones con imágenes. Con esta librería se extraen *frames* aleatorios de la selección de videos y, a continuación, se extraen los recortes de las caras con *Faceboxes*. En la Ilustración 24, se pueden ver ejemplos de estas imágenes con sus etiquetas.

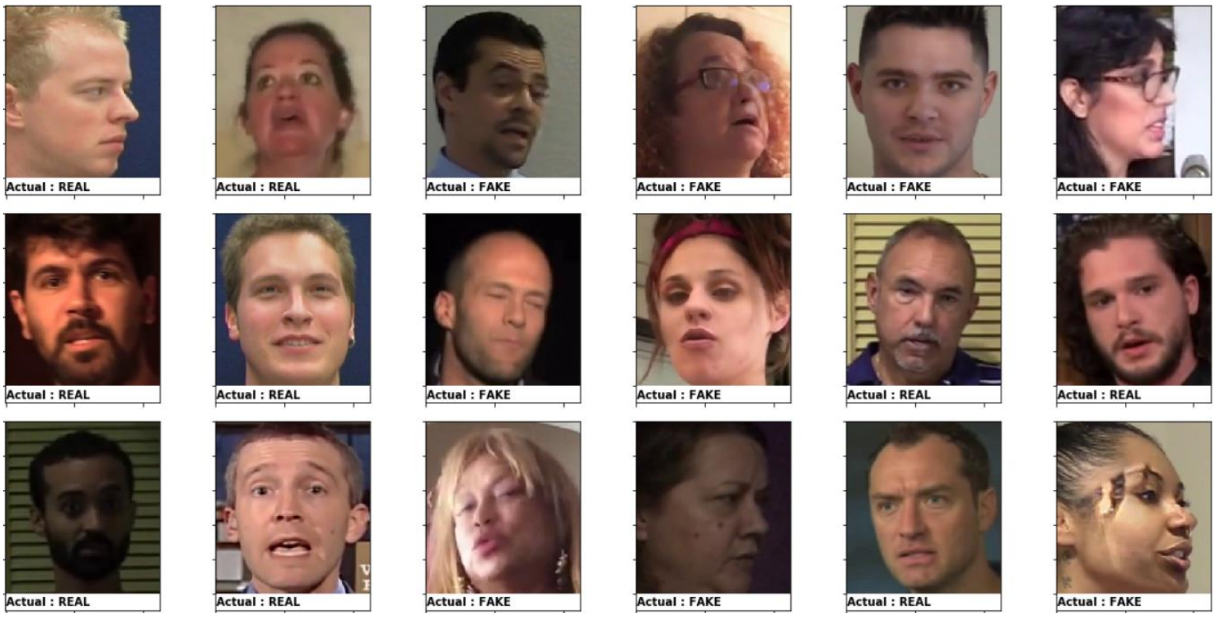


Ilustración 24. Ejemplos de imágenes manipuladas y reales. Fuente: elaboración propia

Para realizar *augmentations*, se va a utilizar la librería *Albumentations* (Buslaev et al., 2020), una librería muy popular en el ámbito de *augmentation* de imágenes. Esta librería incluye más de 70 *augmentatios* distintas y es más rápida que alternativas como las *augmentations* nativas de *torchvision* y *keras* (más de 3 veces más rápido). Además, permite definir una probabilidad para cada tipo de operación, de tal forma que la generación de imágenes es estocástica y el modelo no ve nunca la misma imagen. A continuación, se muestra el código para las operaciones de *augmentation* para entrenamiento (*AUGMENTATIONS_TRAIN*) y para validación y testeo (*AUGMENTATIONS_TEST*)

```
AUGMENTATIONS_TRAIN = Compose([
```

```
        ImageCompression(quality_lower=60, quality_upper=100, p=0.4),
        GaussNoise(p=0.1),
        HorizontalFlip(p=0.5),
        OneOf([RandomBrightnessContrast(), FancyPCA()], p=0.7),
        ShiftScaleRotate(border_mode=BORDER_CONSTANT, p=0.6),
        GridDropout(p=0.2),
        ToGray(p=0.1),
        Resize(275, 225, p=1),
        Normalize()
    ])

    AUGMENTATIONS_TEST = Compose([
        Resize(275, 225, p=1),
        Normalize()
    ])
```

En la Ilustración 25 se pueden ver algunos ejemplos de las imágenes con las transformaciones que se han mencionado para el entrenamiento.

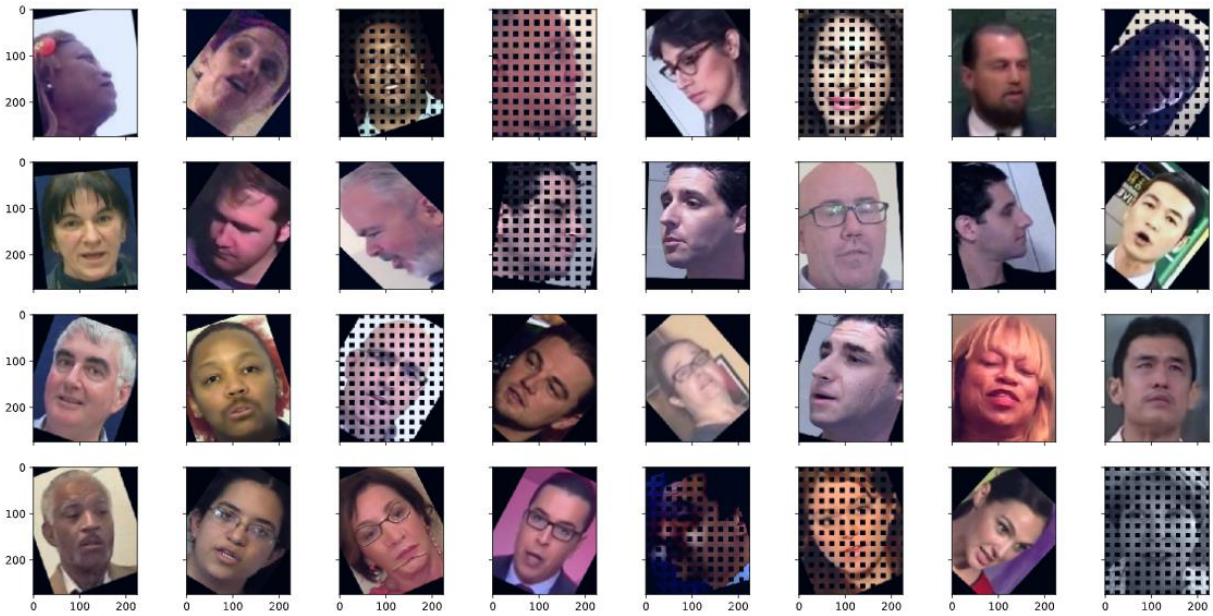


Ilustración 25. Ejemplos de imágenes aumentadas. Fuente: elaboración propia

Se puede ver el código completo para la carga de imágenes en el Anexo I. Módulo de pre-procesado.

5.1.2. Audio

Para la preparación del audio, se va a utilizar la librería *librosa* (McFee et al., 2015) de Python. Esta es una popular librería para análisis de audio en Python, y proporciona fácilmente el

cálculo de los MFCCs. Sin embargo, no proporciona los LFCCs, y es por esto por lo que se creará una versión personalizada de alguna de las funciones que calculan los MFCCs, esencialmente cambiando los filtros en frecuencia Mel a filtros lineales y eliminando las transformaciones a escala Mel. Los coeficientes dinámicos se pueden obtener directamente de la librería.

En la Ilustración 26, se puede ver un audio de ejemplo del *dataset* ASVspoof 2019, dibujado en como la amplitud en función del tiempo. A partir de este audio, en la Ilustración 27 se puede ver de forma gráfica los coeficientes LFCC, Δ-LFCC y Δ^2-LFCC, mostrando además a modo de comparación los MFCC. Como se puede observar, los LFCC y MFCC son parecidos, pero no idénticos.

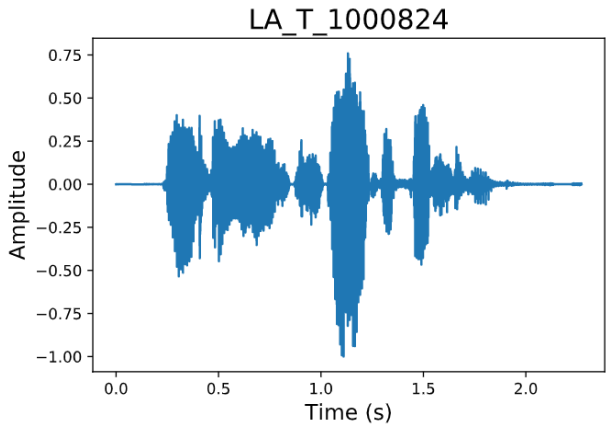


Ilustración 26. Audio de ejemplo del dataset ASVspoof2019. Fuente: elaboración propia

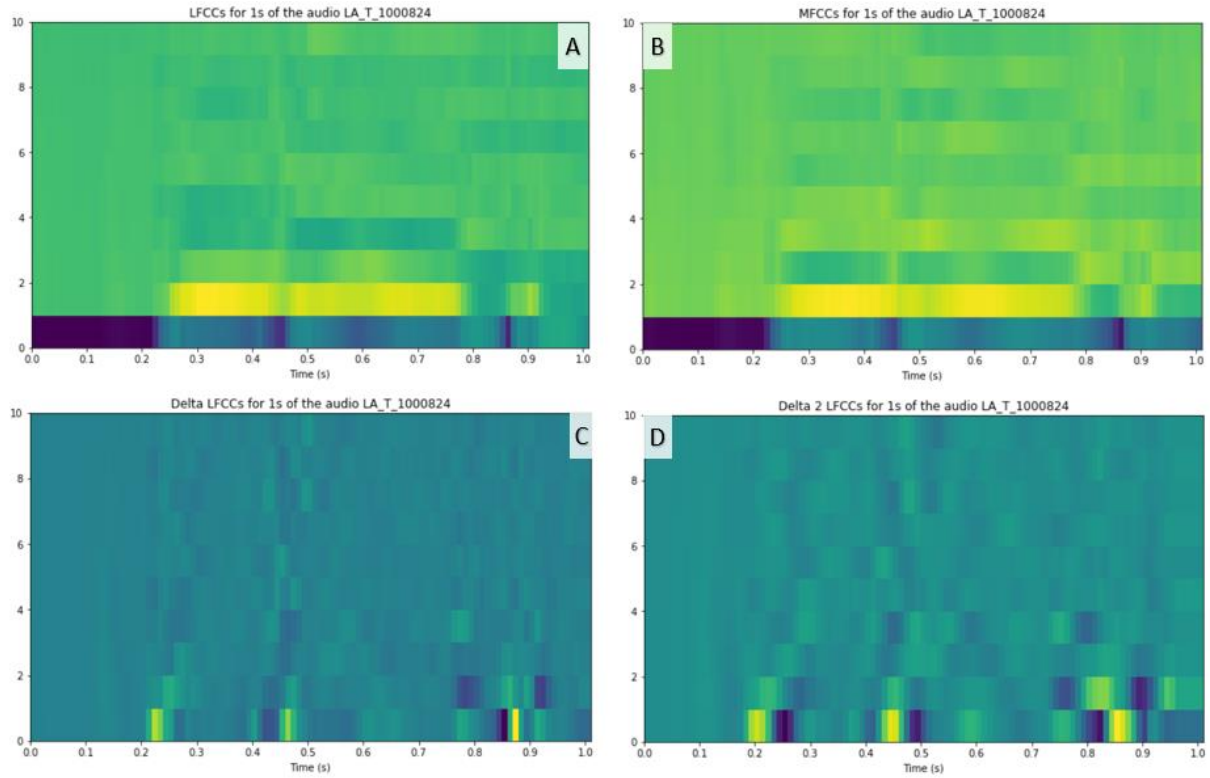


Ilustración 27. LFCCs (A), MFCCs (B), Delta LFCCs (C) y Delta2 LFCCs (D) para un segundo de un audio de ejemplo. Fuente: elaboración propia

En la Ilustración 28, se pueden ver los coeficientes finales que se utilizarán para el entrenamiento del modelo de audio. Los 20 primeros coeficientes corresponden a los Δ-LFCC y los 20 últimos corresponden a los Δ^2-LFCC. Aunque en la imagen no se aprecian demasiado las diferencias entre los valores de los últimos coeficientes, en la Ilustración 29 se pueden ver los coeficientes desglosados en distintos gráficos, empezando arriba a la izquierda por el coeficiente 0 y avanzando hacia la derecha primero y hacia abajo. Como se puede observar, los últimos coeficientes de la imagen son bastante estables y cercanos a 0, por lo que no está claro que la red pueda aprender información correcta a partir de estos coeficientes. Por esta razón, durante el entrenamiento se probará también a utilizar LFCCs no dinámicos en el caso de que los coeficientes dinámicos no den buenos resultados.

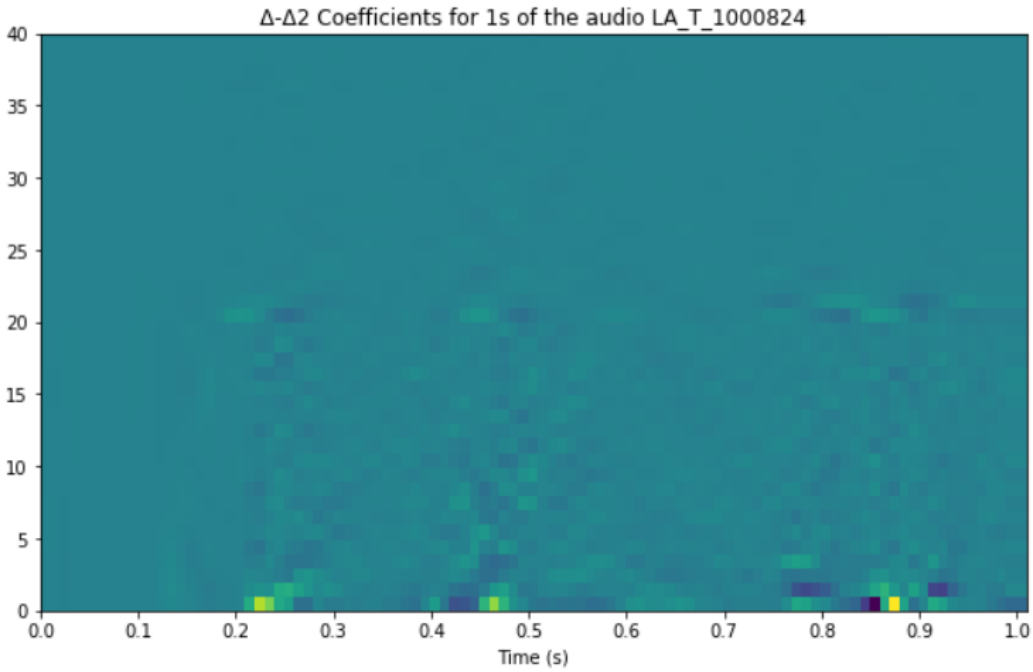


Ilustración 28. Δ-Δ2 LFCCs para 1 segundo de audio. Fuente: elaboración propia

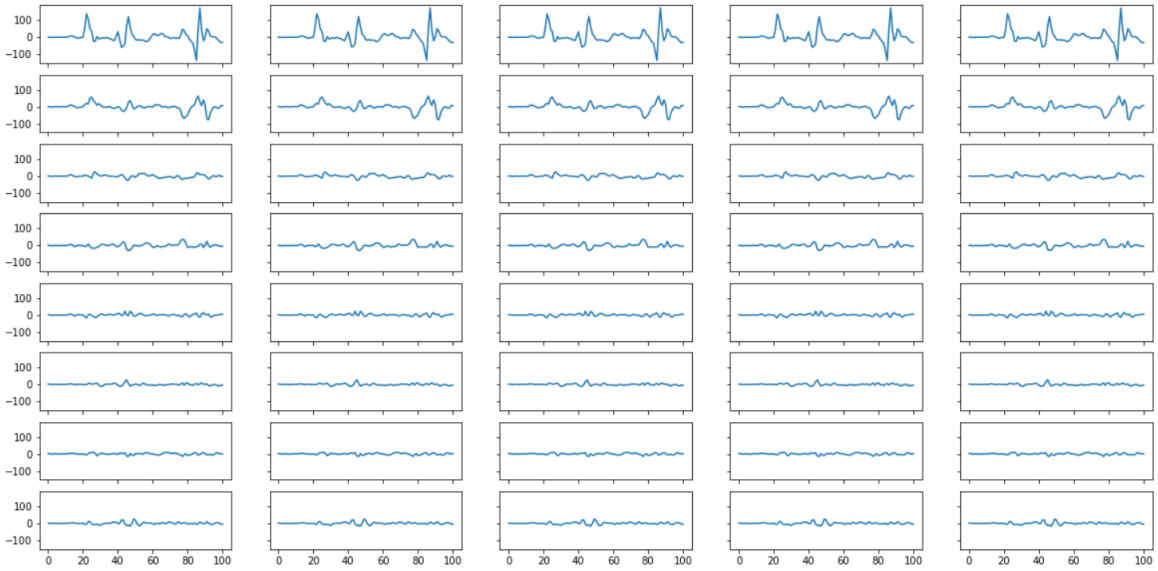


Ilustración 29. Desglose de Δ-Δ2 LFCCs para 1 segundo de audio. Fuente: elaboración propia

A los LFCCs también se les van a realizar *augmentations* para mejorar la capacidad de generalización de la red. Inspirada en la librería *Albumentations*, hay una librería para realizar *augmentations* de forma dinámica para audio llamada *Audiomentations* (Jordal, 2021). Funciona de forma muy similar a *Albumentations*, pero con transformaciones específicas para audio.

En el caso del audio, las *augmentations* se van a realizar tanto antes, como después de convertir el audio a LFCCs. Previamente a convertir el audio a LFCCs, se van a realizar las siguientes transformaciones:

```
AUGMENTATIONS_PRE_TRAIN = Compose([
    AddGaussianNoise(p=0.1),
    ApplyImpulseResponse(ir_path='', p=0.5,leave_length_unchanged=True),
    Shift(p=0.3, fade=True),
    TimeStretch(p=0.5),
    PitchShift(p=0.3),
    Normalize(p=1.0)
])

AUGMENTATIONS_PRE_TEST = Compose([
    Normalize(p=1.0)
])
```

Los sonidos de RIR se obtienen del *dataset* SLR28 (Ko et al., 2017; OpenSLR, 2017). La normalización, por otro lado, se realiza en valores de amplitud entre -1 y 1.

Tras la conversión del audio a LFCCs siguiendo el procedimiento explicado anteriormente, en la fase de entrenamiento, se realiza también enmascaramiento de bandas frecuenciales y enmascaramiento de franjas de tiempo, usando:

```
AUGMENTATIONS_SPEC_TRAIN = Compose([
    SpecFrequencyMask(p=0.5, fill_mode='constant', fill_constant=0.0)
])
```

Las frecuencias y franjas de tiempo enmascaradas se ponen a 0. Este tipo de *augmentation* está basado en *SpecAugment* (Park et al., 2019). Tras esta *augmentation*, se redimensiona la imagen al tamaño que le correponde para poder ser procesada por *MobileNet* y se convierte a RGB. También se normaliza la matriz de salida con los valores de *ImageNet*. Estas operaciones se realizan utilizando *OpenCV*. En la Ilustración 30 se pueden ver algunos ejemplos de los delta-LFCCs que recibe el modelo.

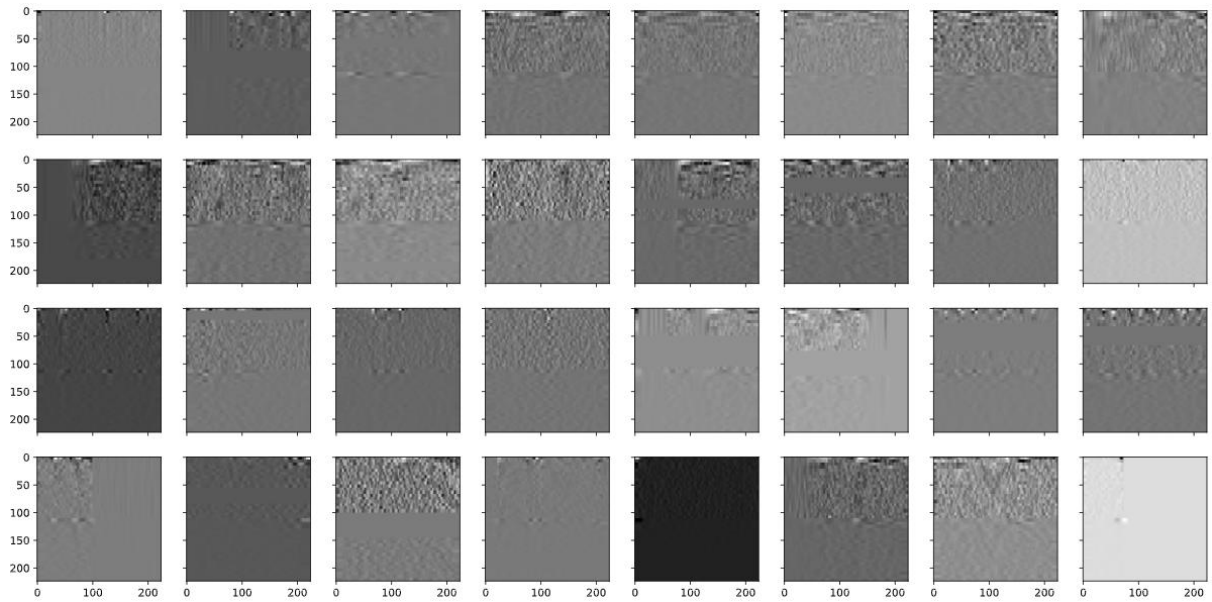


Ilustración 30. Ejemplos de fragmentos de audio aumentados. Fuente: elaboración propia

Se puede ver el código completo en el Anexo I. Módulo de pre-procesado.

5.2. Módulo de entrenamiento

5.2.1. Video

El modelo *efficientNet* ha sido desarrollado utilizando el *framework tensorflow*, por lo que lo más lógico es utilizar la interfaz de *Keras* para el *transfer learning*. Esto se puede hacer de manera sencilla añadiendo una "capa" *efficientNet-v2* a un modelo *Sequential*, especificando *include_top* a falso para que no se incluya la capa *fully connected* de clasificación para *ImageNet* e indicando que no se entrenen los parámetros. Estos no se deben entrenar ya que se va a añadir una capa completamente no entrenada encima de este modelo, y primero se debe entrenar esta capa antes de poder hacer *fine-tunning*.

A modo de comparativa, se van a hacer pruebas con dos modelos distintos, uno con *softmax loss* y otro con *large margin cosine loss*, para comprobar si realmente el segundo es superior al primero en esta aplicación. Debido a la naturaleza de la segunda función de coste, los modelos deben ser ligeramente diferentes.

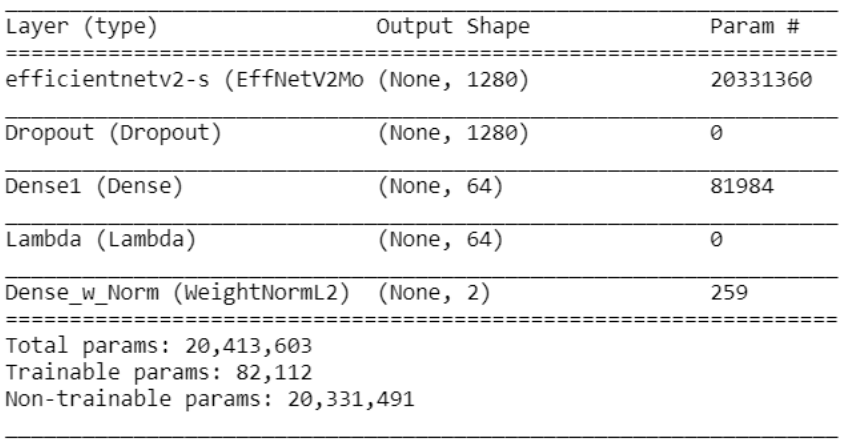
El modelo con *softmax loss* consiste del modelo *efficientNet*, para el que se ha definido el tamaño de imagen de entrada como 225x275. Para las primeras pruebas se usará el modelo *small*, y no se ha aumentado el tamaño porque no se ha visto que fuera necesario. A este modelo le sigue una capa *drop-out*, seguida por dos *fully connected* de 64 y dos neuronas

respectivamente, teniendo la primera capa activación *RELU* y la segunda activación *softmax*. Ambas usan como inicialización de los pesos *glorot uniform*. Se puede ver el resumen del modelo en la Ilustración 31. La función de coste, en concreto, es *sparse categorical cross entropy*, ya que se proveen las etiquetas como un único vector de 0 (*fake*) o 1 (*real*). Este modelo se modificará durante el entrenamiento en función de lo que dé mejores resultados, pero el punto de partida es el que se muestra.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
efficientnetv2-s (EffNetV2Mo (None, 1280)              20331360
_________________________________________________________________
dropout_1 (Dropout)          (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 64)                81984
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 130
=================================================================
Total params: 20,413,474
Trainable params: 82,114
Non-trainable params: 20,331,360
_________________________________________________________________
```

Ilustración 31. Resumen del modelo de video con softmax loss. Fuente: elaboración propia

El modelo con *large margin cosine loss*, por su parte, comienza con el modelo *efficientNet* también, seguido de *drop-out* y una *fully connected* de 64 neuronas y activación *RELU*. Sin embargo, a esta última capa le sigue una capa *lambda*, que realiza la normalización a 1 de las entradas a la siguiente capa, es decir, x. A esto le sigue una capa *fully connected* de 2 neuronas como parámetro de una capa de normalización de los pesos a 1. Dado que la normalización L2 de los pesos a 1 no está implementada, se ha adaptado una implementación de tensorflow para crear una capa personalizada. Tensorflow tiene una capa denominada *WeightNormalization* basada en (Salimans & Kingma, 2016), donde se plantea que la normalización de los pesos puede acelerar la convergencia de la red. En esta capa, sin embargo, se fija la norma a un valor entrenable. Para la presente aplicación, se va a adaptar este código para poner el parámetro de la norma a *trainable* = *False*, de tal forma que se mantenga el valor de inicialización de 1. Se puede ver un resumen de este modelo en la Ilustración 32. Al igual que en el caso anterior, se modificarán los parámetros de este modelo según se vea necesario.



Layer (type)	Output Shape	Param #
efficientnetv2-s (EffNetV2Mo	(None, 1280)	20331360
Dropout (Dropout)	(None, 1280)	0
Dense1 (Dense)	(None, 64)	81984
Lambda (Lambda)	(None, 64)	0
Dense_w_Norm (WeightNormL2)	(None, 2)	259

Total params: 20,413,603
Trainable params: 82,112
Non-trainable params: 20,331,491

Ilustración 32. Resumen del modelo de video con Large Margin Cosine Loss. Fuente: elaboración propia

Además de esta arquitectura diferente, también se ha implementado la función de coste, dado que *Tensorflow* no la proporciona y los investigadores tampoco han proporcionado el código. La implementación se ha basado parcialmente en (MachineLP, n.d.), pero se han tenido que hacer varias modificaciones, ya que esta implementación ha dado varios errores.

Nótese que, en la presente implementación, las capas de *loss* se han incluido dentro del mismo modelo, en lugar de como una parte del cálculo de la función de coste. Esto se debe a que, en este proyecto, el objetivo es tener como salida del modelo la clasificación como real/falso, en lugar de tener como salida un vector de representación para identificar caras contra una base de datos mediante la distancia del coseno. La implementación es equivalente, pero la salida de interés es diferente. Esta implementación se puede ver en el Anexo II. Módulo de entrenamiento.

5.2.2. Audio

El modelo de *MobileNet* también ha sido desarrollado utilizando T*ensorflow*. Esto es una gran ventaja para este proyecto, no sólo porque implica que las implementaciones de ambos modelos pueden compartir bastantes características, sino también porque reduce la complejidad de la arquitectura final. Una vez más, se puede añadir *MobileNetV2* como una "capa" de un modelo secuencial.

La implementación es parecida a la del modelo de video. Para el modelo con la función *softmax*, la arquitectura de partida será la que se muestra en la Ilustración 33, modificándose durante el entrenamiento según las pruebas que se vayan realizando. La capa *fully connected* tiene inicialización de los pesos *glorot uniform* y función de activación *softmax*, siendo la función de coste una vez más *sparse categorical cross entropy*, ya que se proveen las etiquetas como un único vector de 0 (*fake*) o 1 (*real*). También se probarán variantes como la que se muestra en la Ilustración 34.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 1280)              2257984
_________________________________________________________________
Dense (Dense)                (None, 2)                 2562
=================================================================
Total params: 2,260,546
Trainable params: 2,562
Non-trainable params: 2,257,984
_________________________________________________________________
```

Ilustración 33. Arquitectura de uno de los modelos de audio con softmax loss. Fuente: elaboración propia

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 1280)              2257984
_________________________________________________________________
dropout_11 (Dropout)         (None, 1280)              0
_________________________________________________________________
Dense (Dense)                (None, 64)                81984
_________________________________________________________________
Prediction (Dense)           (None, 2)                 130
=================================================================
Total params: 2,340,098
Trainable params: 82,114
Non-trainable params: 2,257,984
_________________________________________________________________
```

Ilustración 34. Arquitectura de otro de los modelos de audio con softmax loss. Fuente: elaboración propia

Por otro lado, la arquitectura de partida del modelo con LCML se muestra en la Ilustración 35. Esta implementación es equivalente a la del modelo de video.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 1280)              2257984
_________________________________________________________________
dropout_3 (Dropout)          (None, 1280)              0
_________________________________________________________________
Dense1 (Dense)               (None, 64)                81984
_________________________________________________________________
Lambda2 (Lambda)             (None, 64)                0
_________________________________________________________________
Dense_w_Norm (WeightNormL2)  (None, 2)                 259
=================================================================
Total params: 2,340,227
Trainable params: 82,112
Non-trainable params: 2,258,115
_________________________________________________________________
```

Ilustración 35. Arquitectura del modelo de audio con LMCL. Fuente: elaboración propia

Como se ha comentado, la implementación del modelo es prácticamente idéntica a la del modelo de video. Esta implementación se puede ver en el Anexo II. Módulo de entrenamiento.

5.3. Módulo de inferencia

El módulo de inferencia va a reutilizar parte del código que se utilice para entrenar los modelos. Sin embargo, este código experimental y pensado para hacer pruebas se va a

optimizar para la predicción y se va a simplificar, manteniendo sólo lo esencial, de tal forma que sea más sencillo de utilizar.

Para este módulo, se deben cargar los modelos finales a partir de los archivos en los que se han guardado. También se deben crear funciones de carga de datos específicas, que no van a funcionar por *batches*. Además, la creación del modelo debe ser automática, sin que el usuario tenga que crear la arquitectura ni compilar el modelo explícitamente. Este comportamiento se abstraerá por medio de programación orientada a objetos.

En concreto, se utilizarán modificaciones de las funciones de *data loading*. Además de estas funciones, se usará MoviePy (Zulko, 2017) para cargar los videos y separar el audio, pudiendo así alimentar los dos modelos por separado.

Por lo demás, la implementación de los modelos será parecida a durante el entrenamiento, utilizando la función *predict* de *Tensorflow* para clasificar los videos. Simplemente, se abstraerá esto para que el usuario vea una interfaz más amigable en la que se diga si el video está manipulado o no.

Para la predicción, se va a utilizar un número concreto de *frames*, a elegir por el usuario (por defecto 10). El parámetro del número de *frames* esencialmente ayuda a aumentar la confianza en la predicción, a costa de un mayor tiempo de computación. Con las predicciones de estos *frames*, se va a realizar una media de los valores de salida del modelo para cada *frame*, considerando por separado cada una de las dos neuronas de salida. Esto se utilizará como predicción final del modelo, donde el valor máximo de las dos neuronas indicará si el video es falso o verdadero.

Por otro lado, para el modelo de audio se extraerán fragmentos de 1 segundo del audio, donde el usuario podrá modificar el número de fragmentos y la longitud de los audios. La predicción final se definirá de la misma forma que en el modelo de video.

Ambas predicciones, se devolverán al usuario de forma clara. Se puede ver el código de esta implementación en el Anexo III. Módulo de inferencia.

6. Evaluación

6.1. Modelo de video

El *dataset* generado siguiendo los pasos anteriormente descritos contiene más de 200 000 imágenes. El entrenamiento del modelo utilizando *transfer learning*, es decir, actualizando únicamente la última capa *fully connected*, para tantas imágenes lleva aproximadamente dos horas por *epoch,* para un *batch size* de 32. Se fija el *batch size* a este tamaño porque es lo máximo que puede tratar el ordenador en el que se está entrenando para este *dataset*. Para las pruebas iniciales que ayudan a definir los hiperparámetros se utiliza únicamente un subset de 500 imágenes de entrenamiento y 100 de validación. Esto reduce el tiempo a un minuto y medio por *epoch*, lo cual permite hacer pruebas iniciales muy rápidamente.

Primeramente, se observa que algunas de las imágenes que se han generado utilizando FaceBoxes no son imágenes de caras. Se realiza por tanto un filtrado visual, eliminando aquellas imágenes que no son caras. Debido a la gran cantidad de imágenes, este filtrado indudablemente no es perfecto, pero se elimina la mayoría de imágenes erróneas, habiendo por tanto sólo un pequeño ruido. Se reduce el número de imágenes de 207294 a 200315, teniendo en cuenta que en todos los *datasets* salvo en DFDC se deja una sola cara por cada frame, ya que sólo una cara está manipulada en estos *datasets*.

Tras eliminar las imágenes incorrectas o no manipuladas, se observa que las dos clases están fuertemente desequilibradas. Hay únicamente 54869 imágenes reales, frente a 152425 imágenes manipuladas, es decir, un ratio de aproximadamente 1:3. Esto era de esperar, ya que los *datasets* originales también están fuertemente desequilibrados. Las clases desbalanceadas, al usar el algoritmo de *backpropagation*, hacen que el gradiente de la clase minoritaria sea muy pequeño con respecto al de la clase mayoritaria, lo que hace que el error de la mayoritaria se reduzca muy rápidamente pero la minoritaria converja muy despacio y mantenga un error alto (Anand et al., 1993). Por otra parte, en (Lee et al., 2016) se experimentó con clases de imágenes fuertemente desequilibradas (ratios de más de 650 a 1), usando distintos métodos como el *Random Under Sampling* (RUS) y el *Random Over Sampling* (ROS) con *augmentation*. Demostraron experimentalmente que el método con mejores resultados fue un entrenamiento inicial con RUS, reduciendo las clases mayoritarias a un umbral, para después hacer *fine-tunning* con todos los ejemplos disponibles. En este proyecto, dado que el desequilibrio no es tan grande como en el experimento mencionado, se va a hacer un entrenamiento inicial de las capas de clasificación de la red, congelando los pesos del modelo pre-entrenado *efficientNet*, con el mismo número de imágenes para las dos

clases, haciendo RUS de la clase FAKE. Posteriormente, se hará *fine-tunning* de todo el modelo utilizando todas las imágenes.

A continuación, se muestran algunos ejemplos de entrenamiento de los modelos con las 500 imágenes de entrenamiento. No se muestran todas las imágenes de las pruebas que se han hecho por limitaciones de espacio. En la Ilustración 36 se muestra el entrenamiento de la red con *softmax loss* y un dropout de 0.3, mientras que en la Ilustración 37 se ve el entrenamiento del modelo con LMCL con *dropout* de 0.2, s = 2 y m = 0.25 (ver ecuación (7)). En las imágenes, se puede ver que en ambos modelos se está cayendo en *overfitting*, aunque el modelo de *softmax* tiene una divergencia algo mayor entre la *loss* de validación y la de entrenamiento.

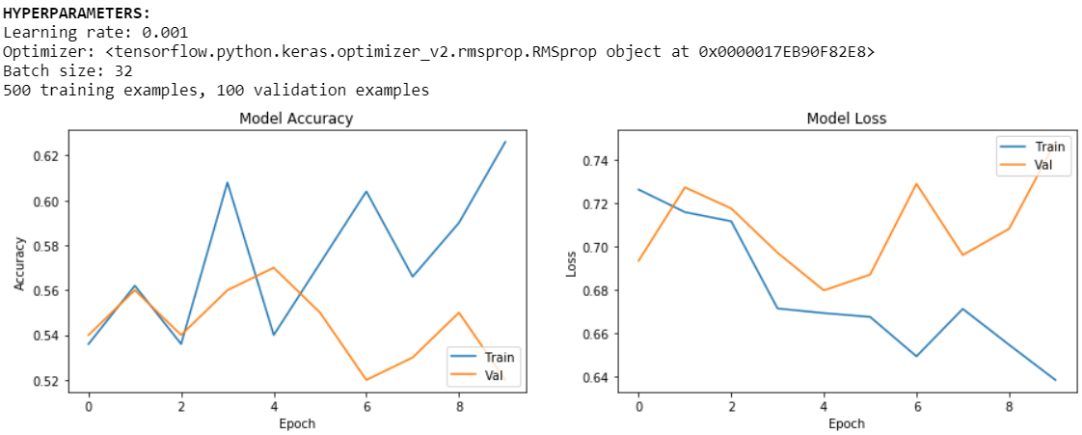


Ilustración 36. Entrenamiento del modelo con sparse categorical cross entropy. Modelo con dropout 0.3. Fuente: elaboración propia

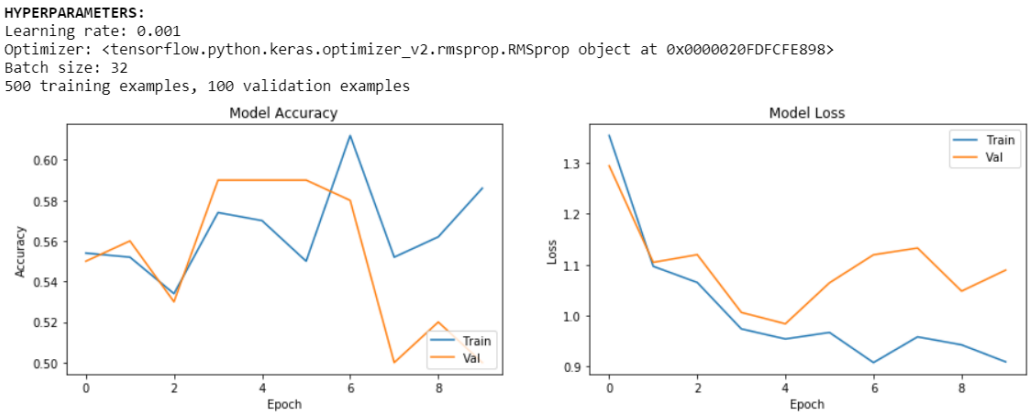


Ilustración 37. Entrenamiento del modelo con LMCL. Modelo con dropout 0.2, m = 0.25, s = 2. Fuente: elaboración propia

En el caso del modelo de *softmax*, se hacen pruebas con distintos valores de *dropout*, *learning rate* y tamaños de capa oculta. Finalmente, lo que mejor resultado da para evitar el sobreajuste es la normalización L2 de los pesos de la capa oculta. Se puede ver este entrenamiento en la Ilustración 38. Se encuentra por tanto que el *dropout* ayuda a reducir el sobreajuste, pero que la técnica de regularización que más aporta es la normalización de los pesos de la capa oculta.

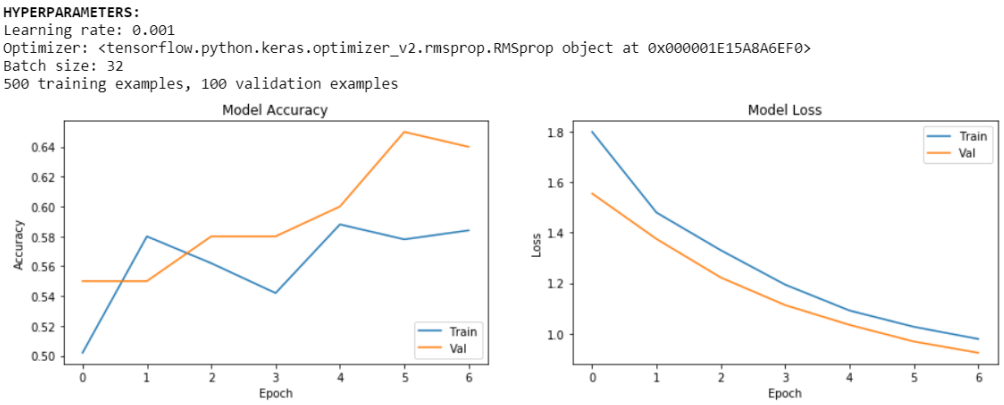


Ilustración 38. Entrenamiento del modelo con sparse categorical cross entropy. Modelo con dropout 0.3 y normalización l2 de los pesos de la capa oculta. Fuente: elaboración propia

En el caso de la función de coste LMCL, se prueban distintos tamaños del vector de representación, así como distintos valores de *learning rate*, *dropout*, s y m. En la Ilustración 39 se puede ver el entrenamiento con un vector de representación de 512, con un *dropout* de 0.2, una m de 0.25 y una s de 64. Como se puede ver, el coste está descendiendo correctamente, sin embargo, esto no se está traduciendo en un incremento en la exactitud. Se observa durante el entrenamiento que el valor de margen tiene un efecto reducido en el entrenamiento siempre que se mantenga en valores pequeños, como recomiendan los autores de (Wang et al., 2018). El tamaño del vector de representación sí ayuda a mejorar el entrenamiento cuando se incrementa su tamaño, mientras que el valor de s tiene influencia en el entrenamiento relativa al *learning rate* que se escoja, es decir, es parecido el entrenamiento con una *s* pequeña y un *learning rate* alto a un entrenamiento con una *s* grande y un *learning rate* más bajo, dado que la *s* tiene un efecto directo sobre la magnitud del coste.

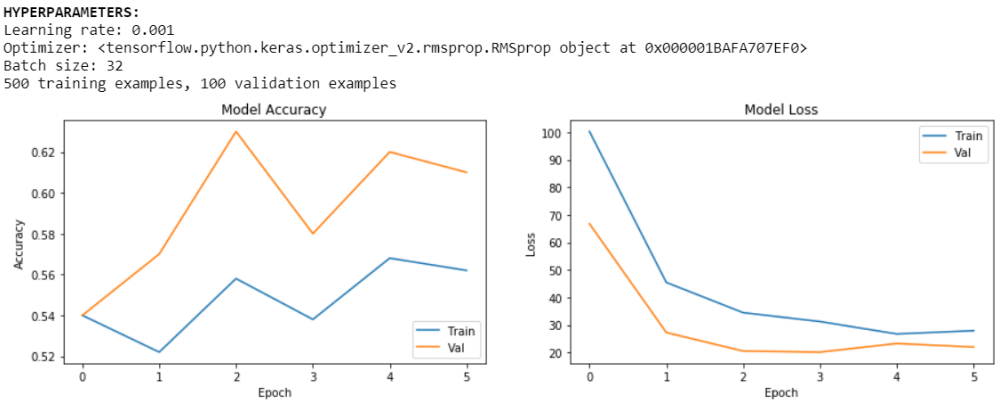


Ilustración 39. Entrenamiento del modelo con LMCL. Modelo con vector de representación de 512, dropout 0.3, m= 0.25, s= 64. Fuente: elaboración propia

Tras evaluar estos entrenamientos, se considera que lo más apropiado para este problema es utilizar la función de coste *softmax*, ya que está dando mejores resultados. Se van a utilizar

los hiperparámetros del entrenamiento que se muestra en la Ilustración 38 para entrenar un modelo con todos los datos.

Como se mencionaba anteriormente, primeramente, se hace un entrenamiento con los pesos de *efficientNet* congelados, entrenando únicamente las nuevas capas de clasificación, y utilizando RUS para igualar el número de ejemplos de las clases. Se entrena este modelo con un *callback* que haga *early stopping* cuando la *loss* de validación lleve 3 *epochs* sin decrecer. El modelo tarda en entrenar unas 3 horas por *epoch*, haciendo 2685 iteraciones por *epoch* para un *batch size* de 32. Se puede observar este entrenamiento en la Ilustración 40. Este modelo está configurado con 0.3 de *dropout*, normalización l2 de los pesos de la capa oculta de 64 neuronas, inicialización *GlorotUniform* de ambas capas, activación RELU de la capa oculta y *softmax* de la segunda capa. Además, se mantiene el mejor modelo de todas las *epochs*. Finalmente:

- *Loss* entrenamiento: 0.67756 | *Loss* validación: 0.63918
- Exactitud entrenamiento: 58.94% | Exactitud validación: 65.03%

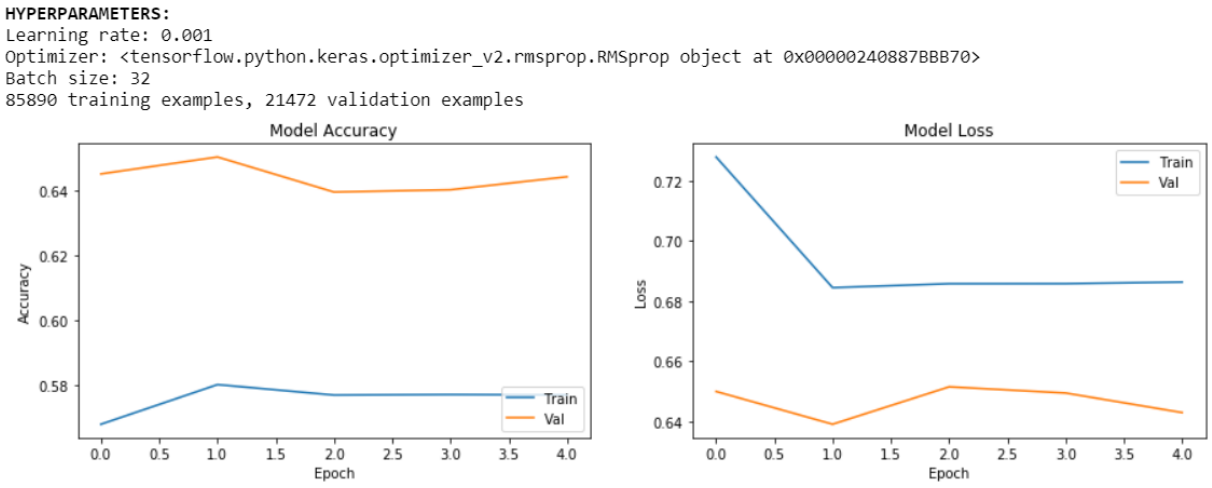


Ilustración 40. Entrenamiento con transfer learning (pesos efficientNet congelados) y función de coste softmax. Fuente: elaboración propia

Es coherente que la precisión de validación sea más alta que la de entrenamiento, ya que las imágenes de validación no están aumentadas, por lo que son más sencillas.

Tras este entrenamiento, se va a utilizar este mismo modelo con sus pesos finales para hacer *fine-tunning*. Se cambia el parámetro de *trainable* de *efficientNet* a *True* y se vuelve a compilar el modelo. Para este entrenamiento, se utilizan todos los datos, por lo que hay 5008 iteraciones por *epoch*. Esto, sumado a que se tienen que calcular los gradientes de todo el modelo de *efficientNet*, hace que el tiempo de entrenamiento por *epoch* suba a unas 32 horas por *epoch*.

En la Ilustración 41, se puede ver el entrenamiento del modelo. Como se ve, a partir de la *epoch* 4 el modelo converge e incluso la *loss* de validación sube ligeramente, por lo que se para el entrenamiento. Así, se obtiene:

- *Loss* entrenamiento: 0.2473 | *Loss* validación: 0.2009
- Exactitud entrenamiento: 90.26% | Exactitud validación: 92.35%

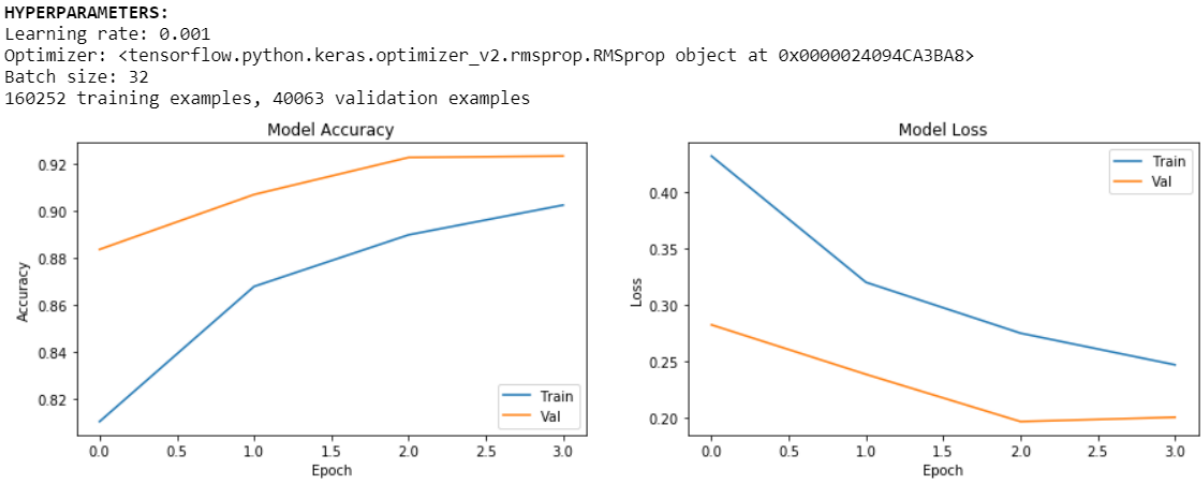


Ilustración 41. Fine-tunning del modelo de video. Fuente: elaboración propia

Dado que en este caso se tienen clases desbalanceadas, se calcula también la AUC. En concreto, se obtiene:

- *AUC score* validación: 0.88

Como se puede ver, el valor de AUC también es bastante alto. En concreto, se puede ver la matriz de confusión en la Tabla 8. Como se puede ver en la tabla, el modelo tiene un cierto prejuicio hacia la clase mayoritaria, la clase falsa. De las imágenes reales, un 21% se clasifican como falsas, mientras que sólo un 2% de las imágenes falsas se clasifican erróneamente como verdaderas. Este prejuicio, aunque reduce la precisión de las clases minoritaria, implica una mayor seguridad frente a videos falsos, a costa de hacer que una mayor cantidad de ellos deban ser erróneamente investigados. Con todo, sólo un 7% de los videos clasificados como falsos no lo son en realidad, y un 7% de los videos clasificados como reales en realidad son falsos.

Tabla 8. Matriz de confusión del modelo de video en el set de validación. Fuente: elaboración propia

	Predicción	
Ground truth	**FAKE**	**REAL**
FAKE	28788	663
REAL	2276	8336

Con estos resultados finales, se va a evaluar el modelo con los datos de testeo del *dataset Deeperforensics1.0*. Para evaluar el modelo, se va a usar un ratio de datos verdaderos frente a falsos más cercano a lo que se encuentra en la realidad, aunque sin reducir excesivamente la cantidad de datos falsos, ya que se busca evaluar correctamente el modelo. En concreto, se usa un ratio de 1.5 imágenes no manipuladas por cada imagen manipulada. Tras extraer las imágenes pertinentes, se obtienen los siguientes resultados:

- *AUC score* testeo: 0.75

Tabla 9. Matriz de confusión del modelo de video en el set de testeo. Fuente: elaboración propia

	Predicción	
Ground truth	**FAKE**	**REAL**
FAKE	17945	10977
REAL	6098	41446

Como se ve en la Tabla 9, los resultados en el set de testeo son peores que los de validación. Un 37% de los videos manipulados han sido clasificados como verdaderos, mientras que el 13% de los videos reales han sido clasificados como falsos. En la Ilustración 42, se pueden ver algunos ejemplos de imágenes erróneamente clasificadas por la red. Se muestra la etiqueta de *ground truth*, donde la predicción ha sido lo contrario.

Ilustración 42. Muestra de imágenes erróneamente clasificadas por el modelo de video. Fuente: elaboración propia

Como se puede ver, la mayoría de las imágenes que se muestran son bastante complicadas. Ninguna tiene artefactos flagrantes, pese a que algunas sí serían fácilmente identificables como falsas por un humano. Las imágenes reales erróneamente clasificadas como falsas también tienen una iluminación desde ángulos no muy comunes. Con todo, un 20% de los videos que se han clasificado como reales en realidad son falsos, frente a un 25% de videos clasificados como falsos que en realidad son reales. La exactitud del modelo es del 77.6%. Pese a que este resultado es peor al conseguido en los datos de entrenamiento, se puede comparar con los resultados en (Du et al., 2020), que se muestra en la Tabla 10. Aquí se compara la capacidad de generalización de distintos modelos. La comparación de exactitudes no es 100% directa, ya que los datos en el caso de este trabajo se han evaluado con FaceForensics++, CelebA manipulado y un tercer grupo de imágenes personalizadas. Sin embargo, sí se aprecia que los mejores resultados que se muestran en esta comparativa son de una exactitud de 96.8% en validación frente a un 68.06% en testeo, mientras que en este trabajo se ha obtenido una exactitud de 92.35% en validación frente a un 77.6% en testeo.

Tabla 10. Comparativa de distintos modelos en diferentes datasets. En cada una de las tres columnas, la sub-columna de la izquierda indica la exactitud en validación, mientras que la de la derecha indica exactitud en testeo. Fuente: (Du et al., 2020)

	Swap		Attribute		Inpainting	
Models	Face2face	FaceSwap	StarGAN	Glow	G&L	ContextAtten
SuppressNet	93.86	50.92	99.98	49.94	99.08	49.98
ResidualNet	86.67	61.54	99.98	49.86	98.96	58.45
StatsNet	92.94	57.74	99.98	50.04	96.17	50.12
MesoInception	94.38	47.32	100.0	50.01	86.90	61.34
XceptionNet	98.02	49.94	100.0	49.67	99.86	50.16
ForensicTransfer	93.91	52.81	100.0	50.08	99.65	50.05
LAE_100	96.84	61.09	99.91	59.05	99.04	57.64
LAE_400	96.82	65.24	99.75	60.08	98.95	60.67
LAE_800	96.80	**68.06**	99.67	**62.11**	98.94	**64.42**

Como se menciona, no se pueden comparar de forma directa los resultados de otros trabajos con este, como tampoco se puede comparar de forma directa con los malos resultados en cuanto a generalización en la DFDC, ya que no se están evaluando los resultados sobre el mismo *dataset*. Sin embargo, cualitativamente se puede ver que este modelo ha generalizado mejor que los mencionados anteriormente.

6.2. Modelo de audio

El *dataset* ASVspoof2019 LA está compuesto por 25380 audios de entrenamiento, de los cuales 22800 son manipulados y tan solo 2580 son verdaderos. Por otra parte, contiene 24844 audios de validación, de los cuales 22296 son manipulados y 2548 son verdaderos. Una vez más, las clases están fuertemente desbalanceadas, por lo que se usará la misma estrategia de RUS para entrenar el modelo con *transfer learning*. Sin embargo, esta vez el ratio no es de 1:3 sino de 1:10. Este fuerte desequilibrio hace que no sea buena idea utilizar todos los ejemplos incluso aunque sólo sea para el *fine-tunning*. Sin embargo, tampoco se quiere renunciar a una cantidad tan grande de ejemplos para equilibrar las dos clases. Es por esto que se va a optar por utilizar RUS para el *transfer learning*, al igual que con el modelo de video, pero para la parte de *fine-tunning*, se va a utilizar ROS de la clase minoritaria, multiplicando por dos el número de ejemplos. Dado que se está utilizando *data augmentation*, el modelo no verá dos veces exactamente el mismo ejemplo. Por otro lado, se hará también RUS de la clase mayoritaria, pero no para igualar la clases, sino para dejar un número final de ejemplos de 4 veces los ejemplos de la clase minoritaria. Esto hace que el ratio final para *fine-tunning* entre las dos clases sea de 1:2, lo cual es mucho más aceptable que un ratio 1:10.

El modelo con *MobileNetV2* es mucho más ligero que el modelo *EfficientNet*. Esto, sumado al menor tamaño del *dataset*, resulta, como cabía esperar, en tiempos de entrenamiento mucho menores. En concreto, se tardan unos 60s por *epoch* para un set de 500 audios de entrenamiento y 100 de validación, y unos 60 minutos por *epoch* para el *dataset* completo en *fine tunning*. Estos tiempos son para un *batch size* de 16. Se ha reducido el tamaño de *batch size* porque parte de la función de *data loading* consiste en dividir el audio en fragmentos de 1 segundo, por lo que de cada audio sale más de una imagen. Salvo que se indique lo contrario, los entrenamientos que se muestran se han optimizado usando RMSprop.

A continuación, se muestran las gráficas de algunas de las pruebas realizadas con este set de audios. No se muestran todas las pruebas por limitaciones de espacio. En la Ilustración 43 se puede ver el entrenamiento del modelo con la función de coste *softmax* y una única capa de salida con dos neuronas tras el modelo de *MobileNet*. En el entrenamiento que se muestra ya se ha ajustado el *learning rate*. Como se puede ver, el modelo mejora, aunque lentamente, pero es el máximo *learning rate* que se puede usar antes de que el modelo caiga en un fuerte sobreajuste.

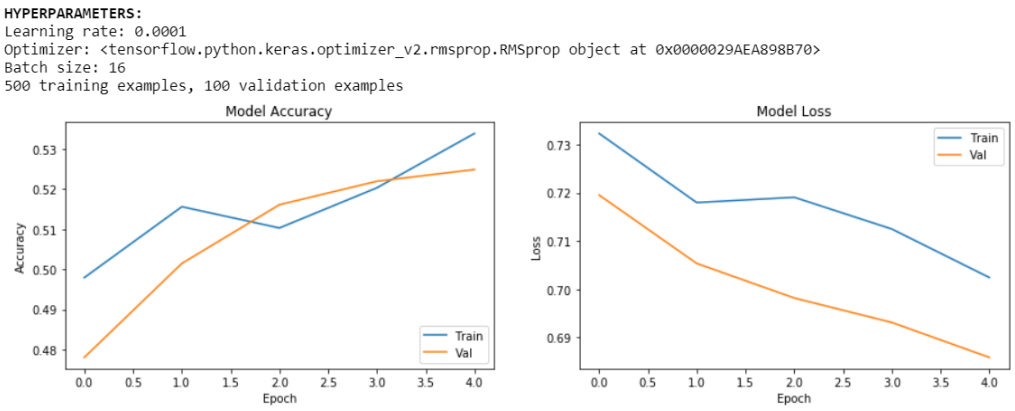


Ilustración 43. Entrenamiento del modelo de audio con función softmax y arquitectura simple (una sola capa tras el modelo MobileNet). Fuente: elaboración propia

En la Ilustración 44, se muestra el mismo modelo, pero añadiendo *dropout* de 0.3 y una capa oculta de 64 neuronas con normalización de los pesos L2. Esto se puede comparar con la Ilustración 45, donde se muestra el entrenamiento del modelo con función de coste LMCL, que tiene un vector de características (capa oculta) de 64 y el mismo *dropout*. Se puede ver que el modelo de LMCL obtiene peores resultados e incluso comienza a aumentar el coste de validación. El modelo con *softmax*, por otro lado, tiene margen todavía para que el coste siga bajando durante algunas *epochs*, aunque se observa que subir el *learning rate* no da mejores resultados.

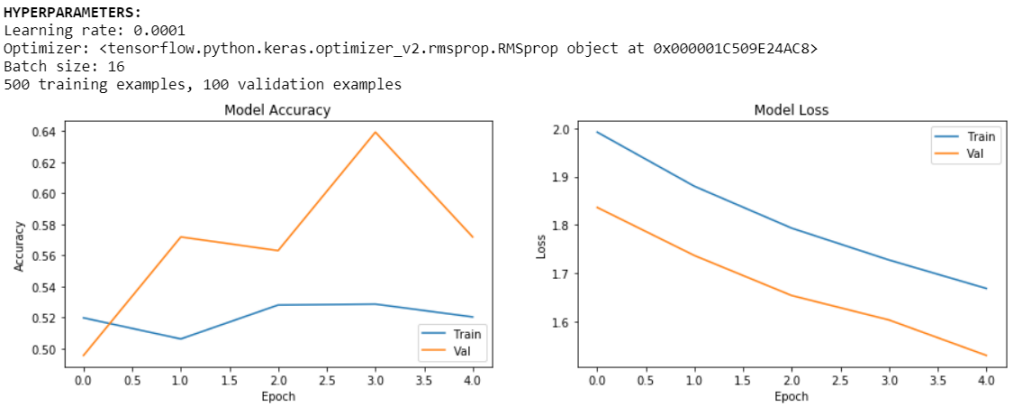


Ilustración 44. Entrenamiento del modelo de audio con función softmax y arquitectura más compleja (se añade dropout de 0.3 y una capa oculta de 64 neuronas). Fuente: elaboración propia

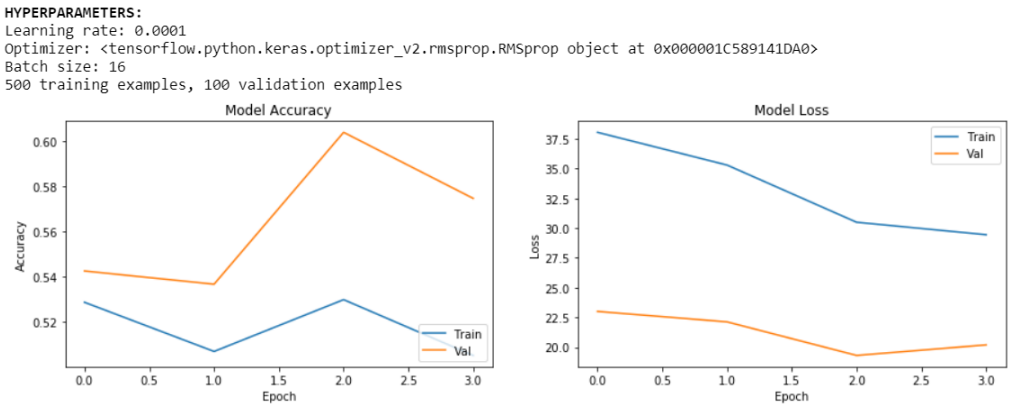


Ilustración 45. Entrenamiento del modelo de audio con función LMCL y vector de representación de 64 neuronas, dropout de 0.3, s=64 y m=0.35. Fuente: elaboración propia

Para el modelo de LMCL, se hacen pruebas variando la *s*. Este parámetro tiene principalmente influencia en el tamaño del espacio disponible para representar los datos, por lo que debería ser grande para un número de clases grande, pero se puede reducir en el caso de pocas clases, como es el caso para este proyecto. Se muestra una de esas pruebas en la Ilustración 46. Como se puede ver, la evolución de la función de coste es parecida, aunque se consigue una exactitud algo más alta. Por otro lado, se observa que aumentar el tamaño del vector de representación no ayuda a mejorar el entrenamiento incluso para *learning rates* bajos.

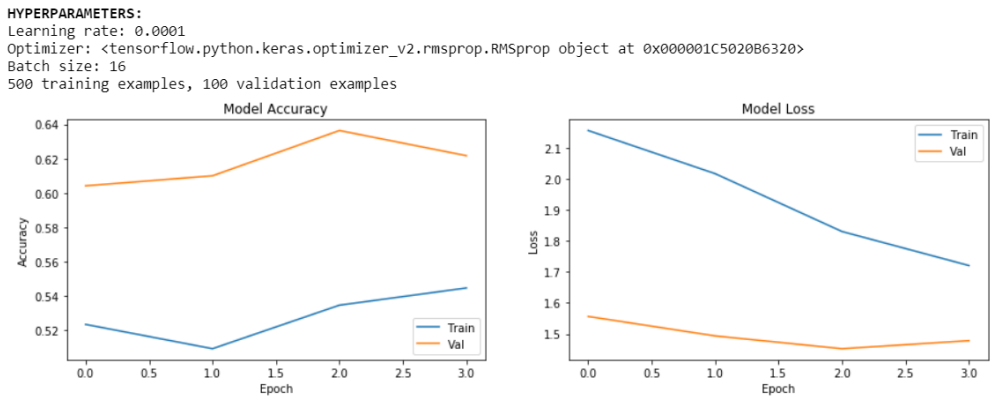


Ilustración 46. Entrenamiento del modelo de audio con función LMCL y vector de representación de 64 neuronas, dropout de 0.3, s=4 y m=0.35. Fuente: elaboración propia

Para el modelo con función de coste *softmax*, se hacen pruebas también con tamaños de capa oculta, cantidad de capas ocultas, *learning rates* y niveles de *dropout*. En la Ilustración 47 se muestra el entrenamiento con una capa oculta de 512 neuronas y *dropout* de 0.3 (*dropout* menores que esto no dan buenos resultados). Incrementar el *learning rate* para este modelo no mejora la exactitud ni la evolución del coste. Por el contrario, en la Ilustración 48 se muestra el entrenamiento con *dropout* de 0.4, lo cual permite aumentar el *learning rate* y obtener resultados parecidos, aunque sí que se observa que la exactitud de entrenamiento está subiendo más rápidamente, un claro signo de sobreajuste.

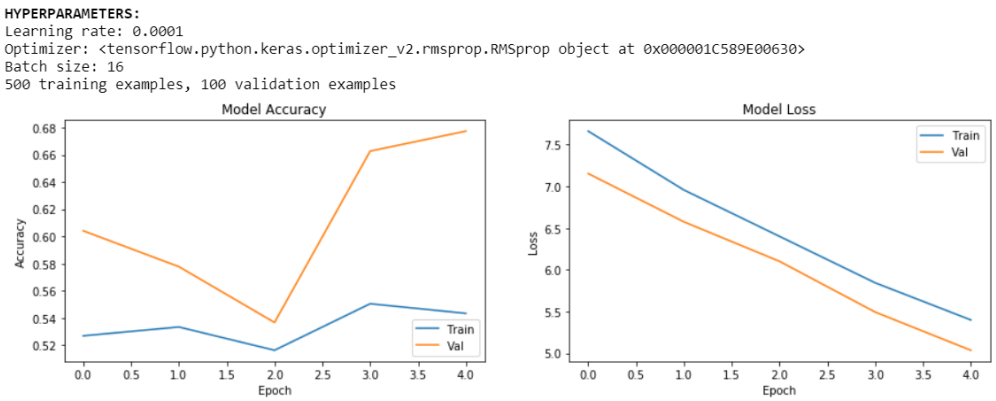


Ilustración 47. Entrenamiento del modelo de audio con función softmax con dropout de 0.3 y una capa oculta de 512 neuronas. Fuente: elaboración propia

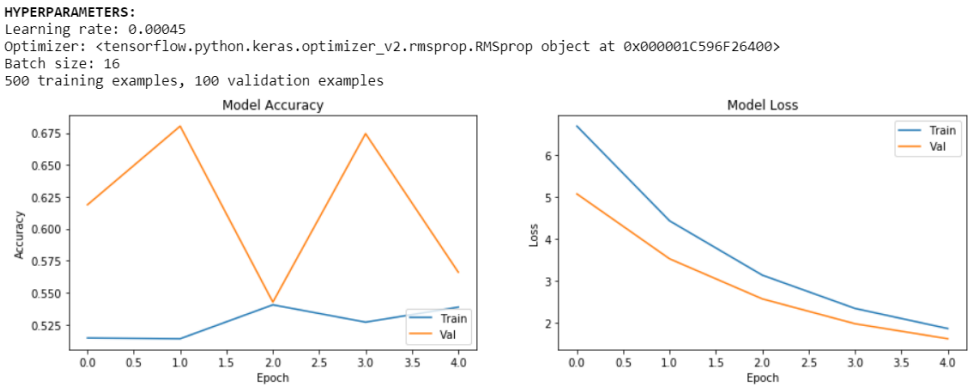


Ilustración 48. Entrenamiento del modelo de audio con función softmax con dropout de 0.4 y una capa oculta de 512 neuronas. Fuente: elaboración propia

Finalmente, tras comparar los modelos, se opta por entrenar un modelo final con función de coste *softmax*, con una capa oculta de 512 neuronas, activación RELU y normalización L2 de los pesos. Además, se utiliza un *dropout* de 0.3 y un *learning rate* 0.0001. Como nota, destacar que se hacen pruebas también con el optimizador Adam, obteniendo resultados parecidos.

En la Ilustración 49 se muestra el entrenamiento con los pesos de MobileNetV2 congelados. Como se puede observar, se alcanza la convergencia en pocas *epochs*, restaurándose como modelo final, según la configuración que se ha hecho, los pesos de la *epoch* 2 (empezando por 0). Esto implica que:

- *Loss* entrenamiento: 1.8800 | *Loss* validación: 1.572
- Exactitud entrenamiento: 56.06% | Exactitud validación: 68.89%

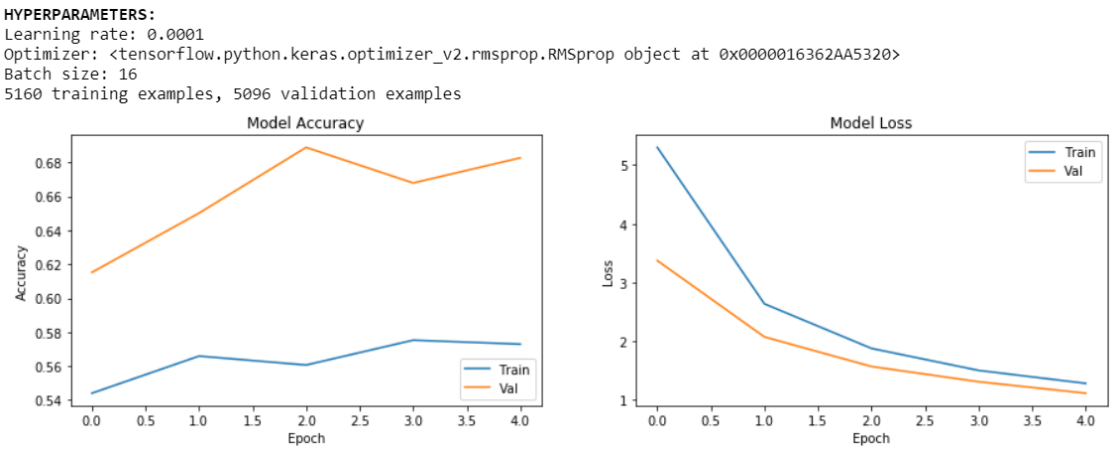


Ilustración 49. Entrenamiento con todos los ejemplos del modelo de audio. Fuente: elaboración propia

Sin embargo, cuando se pasa a la parte de *fine tunning*, el modelo sobreajusta inmediatamente, manteniéndose la exactitud de validación alrededor del 50%. Tras estas pruebas, se plantea que posiblemente los coeficientes dinámicos no son suficientemente

informativos para que el modelo pueda aprender a diferenciar los audios manipulados de los originales.

Por este motivo, se va a probar a entrenar el mismo modelo, pero con imágenes a partir de LFCCs no dinámicos. Además, para reducir el sobre-ajuste y, dado que los audios del *dataset* tienen una duración baja, en lugar de utilizar fragmentos de un segundo de audio se van a utilizar los audios completos para generar la "imagen" de LFCCs. Los resultados se muestran en la Ilustración 50 y a continuación:

- *Loss* entrenamiento: 1.1242 | *Loss* validación: 0.9477
- Exactitud entrenamiento: 69.30% | Exactitud validación: 85.32%

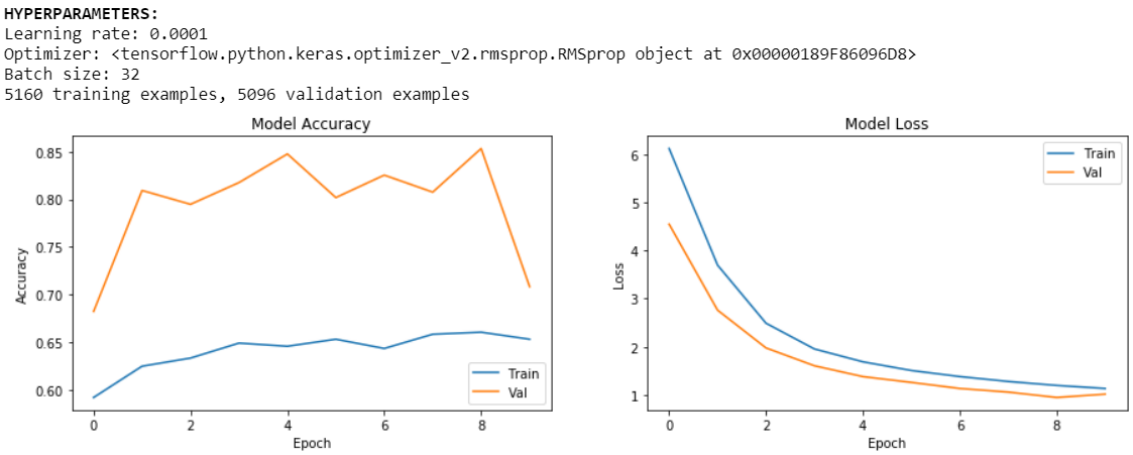


Ilustración 50. Entrenamiento del modelo de audio con coeficientes LFCC no dinámicos. Fuente: elaboración propia

Como se puede ver, los resultados son mucho mejores que con los coeficientes dinámicos, consiguiéndose una precisión mucho mayor. Dados estos resultados, se decide entrenar la red con el ratio de datos que se ha mencionado anteriormente, con ROS para la clase minoritaria y RUS para la clase mayoritaria, pero sin descongelar los pesos de MobileNet. Tras este entrenamiento, y reduciendo el *learning rate*, se consiguen los resultados que se ven en la Ilustración 51. Los resultados que se mantienen son los de la *epoch* 3, donde se logra la mayor exactitud de validación, en concreto un 88.7%. Por otro lado, se obtiene:

- *AUC score* validación: 0.86

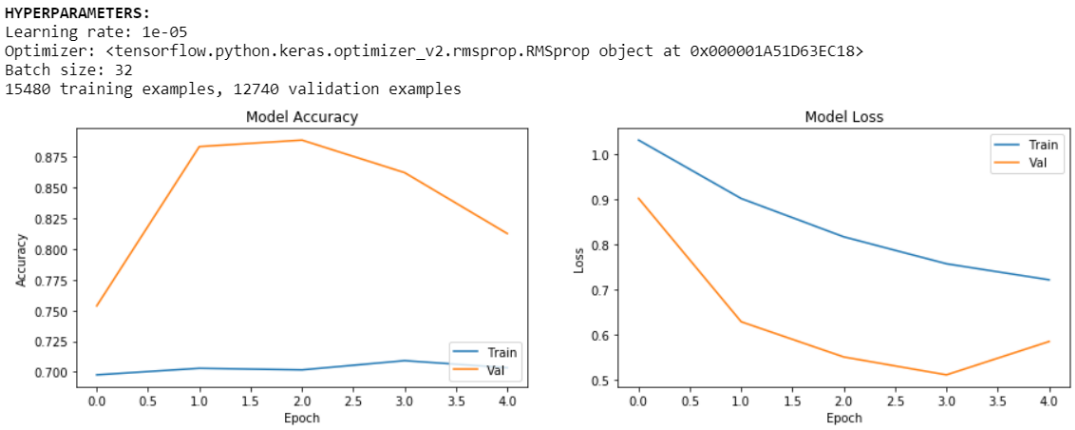


Ilustración 51. Entrenamiento del modelo de audio con más datos. Fuente: elaboración propia

El AUC de validación es por lo tanto también bastante alto, señal de que el modelo no ha tomado preferencia por la clase mayoritaria. Además, puede verse en la Tabla 11 la matriz de confusión. Como se puede observar, el 16% de los audios manipulados se han clasificado como reales, mientras que el 13% de los audios no manipulados han sido clasificados como manipulados.

Tabla 11. Matriz de confusión del modelo de audio en el set de validación. Fuente: elaboración propia

	Predicción	
Ground truth	**FAKE**	**REAL**
FAKE	8570	1622
REAL	331	2217

Se considera que estos resultados son bastante satisfactorios considerando la ligereza y rapidez del modelo. Además, viendo la facilidad con la que el modelo sobreajusta a los datos, no se considera que utilizar un modelo más potente, añadiendo por tanto capacidad de representación (y sobreajuste), vaya a ayudar a obtener mejores resultados. Por tanto, se decide mantener este modelo como modelo final y se evalúa su capacidad de generalización con los datos de testeo. Los resultados se pueden ver en la Tabla 12. En resumen:

- *AUC score* testeo: 0.77

Tabla 12. Matriz de confusión del modelo de audio en el set de testeo. Fuente: elaboración propia

	Predicción	
Ground truth	**FAKE**	**REAL**
FAKE	53386	10496
REAL	2180	5175

Como se puede ver, el AUC de testeo es un 10% menor al de validación. Esto quiere decir que en algún aspecto el modelo no está generalizando de forma del todo correcta. Si se observa la matriz de confusión, se puede ver que un 16% de los audios manipulados han sido clasificados como verdaderos, lo cual es idéntico al porcentaje de fallos en el set de validación. Sin embargo, un 29% de los ejemplos reales han sido clasificados como falsos, lo cual es un porcentaje considerablemente mayor que en el set de validación. Esto tiene una explicación clara y es que no hay suficientes ejemplos no manipulados para que el modelo pueda generalizar correctamente.

Por otro lado, es bueno ver que el modelo está sabiendo detectar el mismo porcentaje de videos falsos, incluso teniendo en cuenta que las técnicas de generación de audios utilizadas en el set de testeo son completamente distintas a las del set de entrenamiento y el de validación.

6.3. Combinación de modelos

Los dos modelos que se han mencionado se han combinado en una única herramienta, cuya interfaz es un *jupyter notebook*. El *notebook* tiene instrucciones claras para que los usuarios puedan ejecutar de forma sencilla el modelo y obtengan una predicción clara. La interfaz se muestra a continuación, en la Ilustración 52.

1 Deepfake Detector

1.0.1 Import necessary libraries

[1]:
```
#custom
from models.inference import DeepfakeDetector
df_det = DeepfakeDetector()
```

1.0.2 Provide the following data:

[2]:
```
#Path to the video:
video_path = r'C:\Users\Monica\Downloads\Cruzcampo Con Mucho Acento_360P.mp4'

#Number of video frames to use for the prediction:
video_frames = 10 #int

#Number of audio frames to use for the prediction:
audio_frames = 5 #int

#Duration of the audio frames in seconds:
audio_len = 1.0 #int or float
```

Bear in mind that a larger amount of frames will give a more accurate result, but will result in longer computation times.

1.0.3 Run the following cell to get your prediction:

[3]:
```
df_det.deepfake_detector([video_path], video_frames, audio_frames, audio_len)
```

```
The image for this video looks manipulated with a certainty of 56.27%
The audio for this video looks manipulated with a certainty of 73.95%
```

Ilustración 52. Ejecución de ejemplo de la herramienta de predicción. Fuente: elaboración propia

Se muestra cómo quedaría una ejecución de ejemplo, en la que se le transmite al usuario que tanto la imagen como el audio de un video han sido manipulados, indicándole el porcentaje de confianza, que es la salida de la neurona del modelo con mayor puntuación. La salida que se muestra corresponde al video que se mencionaba en la introducción del anuncio con Lola Flores (García, 2021). Como se puede ver, se detecta correctamente que el video es falso, aunque la confianza es baja, ya que este video es extremadamente realista.

7. Conclusiones y trabajo futuro

7.1. Conclusiones

Los *deepfakes* son una potencial fuente de desinformación y manipulación, pero también de entretenimiento si se crean con fines no maliciosos. Se han utilizado en el pasado para engañar (Damiani, 2019), pero también para ayudar a recordar a seres queridos (García, 2021). En este trabajo, se ha buscado generar un modelo de detección de *deepfakes* que pueda empoderar a sus usuarios para distinguir manipulaciones de videos o audios potencialmente muy realistas.

El problema se ha abordado utilizando dos modelos separados, uno para video y otro para audio. Esto se ha hecho considerando que ningún *dataset* en la actualidad contiene tanto video como audio manipulado y etiquetado de forma correcta. A partir de una variedad de *datasets*, se ha entrenado ambos modelos utilizando *transfer learning*, con distintos modelos pre-entrenados para audio y video. A los *datasets*, se les ha aplicado una serie de transformaciones o *augmentations* para mejorar la capacidad de generalización de los modelos. Posteriormente, los modelos entrenados se han evaluado con datos no utilizados ni para entrenamiento ni para validación con el fin de evaluar la capacidad de generalización del modelo.

En el modelo de video, se han conseguido muy buenos resultados en el set de validación, aunque con un cierto prejuicio del modelo hacia la clase falsa. Posteriormente, al evaluar el modelo con el set de testeo, se ha visto que los resultados son peores, como cabía esperar para una distribución de datos no vista durante el entrenamiento. Sin embargo, se ha podido comprobar que el modelo generaliza mejor que el estado del arte, aunque la comparativa se haya podido realizar solo cualitativamente, dado que no se ha evaluado el modelo con los mismos datos que otros trabajos.

En lo que respecta al modelo de audio, los resultados obtenidos han sido buenos, teniendo en cuenta que se ha utilizado un modelo muy ligero. El modelo ha sabido generalizar bien en lo que respecta a la detección de audios falsos. Sin embargo, no ha podido generalizar tan bien en el caso de los audios verdaderos. Se cree que esto se debe a que los datos de entrenamiento incluían muy pocos ejemplos de audios no manipulados.

En general, lo que se ha visto en ambos modelos es que los *datasets* están fuertemente desequilibrados hacia los videos manipulados, cosa que no es el caso fuera de este entorno

controlado, ya que en el día a día la gran mayoría de los videos y audios no son *deepfakes*. Se cree por tanto que una mejora de estos modelos pasaría primeramente por aumentar la población de ejemplos no manipulados. También se ha visto, en el caso de la clasificación de video, que el uso de una población variada de videos manipulados, que han sido producidos con diferentes técnicas, permite conseguir una mejor capacidad de generalización.

Los dos modelos se han integrado en una única herramienta de detección de *deepfakes*, que es sencilla de utilizar y se espera que sea fuente de inspiración para otros trabajos.

En lo que respecta a los objetivos de este trabajo, se considera que se han alcanzado satisfactoriamente. Se han analizado las técnicas existentes de detección de *deepfakes*. A partir de estas técnicas, se ha evaluado el acercamiento más apropiado para solucionar el problema. A continuación, se ha desarrollado un modelo capaz de detectar *deepfakes* de video y de audio y se ha evaluado el modelo sobre un *dataset* oculto. Además, se ha facilitado el acceso al modelo para usuarios no técnicos mediante una interfaz sencilla.

7.2. Líneas de trabajo futuro

A partir de este proyecto, existe una serie de líneas futuras de trabajo. Primeramente, y como se mencionaba en capítulos anteriores, se debe seguir trabajando continuamente para mejorar los modelos de detección de *deepfakes*, ya que también mejoran constantemente las técnicas de generación de *deepfakes*. Se espera que la metodología usada en este trabajo pueda servir de punto de partida para esas mejoras futuras.

Como ejemplo, los modelos presentados podrían mejorarse realizando el entrenamiento en un dispositivo más potente. Esto permitiría hacer más iteraciones y comparativas, sin duda llegando a un modelo con una exactitud mayor. Se podrían entrenar redes de mayor tamaño que, en el caso de video, podrían lograr una mayor exactitud.

Otra posible línea que no se ha implementado por falta de tiempo es el entrenamiento de los modelos con datos que contengan *adversarial noise*. Este tipo de ruido puede confundir fácilmente a una red neuronal, ya que se entrena al generador del ruido específicamente para engañar al clasificador. El entrenamiento con ruido de este tipo podría mejorar la robustez del modelo frente a este tipo de ruido.

En lo que respecta a la herramienta, una clara línea de trabajo sería mejorar la interfaz para hacerla accesible incluso a usuarios que no sepan utilizar Python. Se podría desarrollar una

aplicación. Esto requeriría, además, hacer un mantenimiento constante del modelo, ya que si no puede degradarse con el tiempo.

Otra mejora en lo que respecta a la herramienta sería la implementación de un *tracker*. Esto haría que la detección de las caras fuese más estable, e incluso podría mejorar las predicciones realizando las medias de las predicciones individualmente para cada entidad.

En definitiva, este trabajo abre muchas posibilidades futuras de trabajo para mejorar la convivencia con los *deepfakes*, que sin duda han venido para quedarse.

8. Bibliografía

Afchar, D., Nozick, V., Yamagishi, J., & Echizen, I. (2018). Mesonet: a compact facial video forgery detection network. *2018 IEEE International Workshop on Information Forensics and Security (WIFS)*, 1–7.

Agarwal, S., & Varshney, L. R. (2019). Limits of deepfake detection: A robust estimation viewpoint. *ArXiv Preprint ArXiv:1905.03493*.

Amos, B., Ludwiczuk, B., Satyanarayanan, M., & others. (2016). Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, *6*(2).

Anand, R., Mehrotra, K. G., Mohan, C. K., & Ranka, S. (1993). An improved algorithm for neural network classification of imbalanced training sets. *IEEE Transactions on Neural Networks*, *4*(6), 962–969.

ASV. (2021). *ASVspoof 2021*. https://www.asvspoof.org/

Bozinovski, S., & Fulgosi, A. (1976). The influence of pattern similarity and transfer learning upon training of a base perceptron b2. *Proceedings of Symposium Informatica*, 3–121.

Buslaev, A., Iglovikov, V. I., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). Albumentations: Fast and Flexible Image Augmentations. *Information*, *11*(2). https://doi.org/10.3390/info11020125

Chen, P., Liu, S., Zhao, H., & Jia, J. (2020). Gridmask data augmentation. *ArXiv Preprint ArXiv:2001.04086*.

Chen, T., Kumar, A., Nagarsheth, P., Sivaraman, G., & Khoury, E. (2020). Generalization of audio deepfake detection. *Proceedings of the Odyssey Speaker and Language Recognition Workshop, Tokyo, Japan*, 1–5.

Choi, H., Ryu, S., & Kim, H. (2018). Short-term load forecasting based on ResNet and LSTM. *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, 1–6.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1251–1258.

Chugh, K., Gupta, P., Dhall, A., & Subramanian, R. (2020). Not made for each other-Audio-

Visual Dissonance-based Deepfake Detection and Localization. *Proceedings of the 28th ACM International Conference on Multimedia*, 439–447.

Damiani, J. (2019). *A Voice Deepfake Was Used To Scam A CEO Out Of $243,000*. https://www.forbes.com/sites/jessedamiani/2019/09/03/a-voice-deepfake-was-used-to-scam-a-ceo-out-of-243000/

Das, R. K., Yang, J., & Li, H. (2019). Long range acoustic and deep features perspective on ASVspoof 2019. *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 1018–1025.

Dolhansky, B., Howes, R., Pflaum, B., Baram, N., & Ferrer, C. C. (2020). The deepfake detection challenge (dfdc) dataset. *ArXiv Preprint ArXiv:2006.07397*, *1*(2).

Du, M., Pentyala, S., Li, Y., & Hu, X. (2020). Towards generalizable deepfake detection with locality-aware autoencoder. *Proceedings of the 29th ACM International Conference on Information \& Knowledge Management*, 325–334.

Facebook. (2020). *Deepfake Detection Challenge Dataset*. Facebook AI. https://ai.facebook.com/datasets/dfdc/

Faceswap-GAN. (2019). https://github.com/shaoanlu/faceswap-GAN

Foley, J. (2021). *12 deepfake examples that terrified and amused the internet*. https://www.creativebloq.com/features/deepfake-examples

Fraile, R., Godino-Llorentea, N. S.-L. J. I., & Fredouilleb, V. O.-R. C. (2009). Automatic detection of laryngeal pathologies in records of sustained vowels by means of MFCC parameters and differentiation of patients by sex. *Folia Phoniatrica et Logopaedica*, *61*(3), 146–152.

Gao, Y., Lian, J., Raj, B., & Singh, R. (2021). Detection and Evaluation of human and machine generated speech in spoofing attacks on automatic speaker verification systems. *2021 IEEE Spoken Language Technology Workshop (SLT)*, 544–551.

García, J. (2021). *Lola Flores vuelve a escena gracias a los deepfakes en el nuevo anuncio de Cruzcampo*. https://www.xataka.com/robotica-e-ia/lola-flores-vuelve-a-escena-gracias-a-deepfakes-nuevo-anuncio-cruzcampo

Giannakopoulos, T. (2015). pyaudioanalysis: An open-source python library for audio signal analysis. *PloS One*, *10*(12), e0144610.

Gu, Y., & Kang, Y. (2018). Multi-task WaveNet: A multi-task generative model for statistical parametric speech synthesis without fundamental frequency conditions. *ArXiv Preprint ArXiv:1806.08619*.

Guarnera, L., Giudice, O., & Battiato, S. (2020). Fighting Deepfake by Exposing the Convolutional Traces on Images. *IEEE Access*, *8*, 165085–165098.

Güera, D., & Delp, E. J. (2018). Deepfake video detection using recurrent neural networks. *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 1–6.

Hasan, M. R., Jamil, M., Rahman, M., & others. (2004). Speaker identification using mel frequency cepstral coefficients. *Variations*, *1*(4).

Hsu, C.-C., Zhuang, Y.-X., & Lee, C.-Y. (2020). Deep fake image detection based on pairwise learning. *Applied Sciences*, *10*(1), 370.

Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). Densenet: Implementing efficient convnet descriptor pyramids. *ArXiv Preprint ArXiv:1404.1869*.

Jiang, L., Li, R., Wu, W., Qian, C., & Loy, C. C. (2020). Deeperforensics-1.0: A large-scale dataset for real-world face forgery detection. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2889–2898.

Jordal, I. (2021). *Audiomentations*. https://github.com/iver56/audiomentations

Kabakus, A. T., & others. (2019). *An Experimental Performance Comparison of Widely Used Face Detection Tools*.

Kelion, L. (2020). *Deepfake detection tool unveiled by Microsoft*. BBC. https://www.bbc.com/news/technology-53984114

Ko, T., Peddinti, V., Povey, D., Seltzer, M. L., & Khudanpur, S. (2017). A study on data augmentation of reverberant speech for robust speech recognition. *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5220–5224.

Koike, T., Qian, K., Kong, Q., Plumbley, M. D., Schuller, B. W., & Yamamoto, Y. (2020). Audio for audio is better? An investigation on transfer learning models for heart sound classification. *2020 42nd Annual International Conference of the IEEE Engineering in*

Medicine \& Biology Society (EMBC), 74–77.

Koopman, M., Rodriguez, A. M., & Geradts, Z. (2018). Detection of deepfake video manipulation. *The 20th Irish Machine Vision and Image Processing Conference (IMVIP)*, 133–136.

Korshunov, P., & Marcel, S. (2018). Deepfakes: a new threat to face recognition? assessment and detection. *ArXiv Preprint ArXiv:1812.08685*.

Korshunov, P., & Marcel, S. (2020). Deepfake detection: humans vs. machines. *ArXiv Preprint ArXiv:2009.03155*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, *25*, 1097–1105.

Lee, H., Park, M., & Kim, J. (2016). Plankton classification on imbalanced large scale database via convolutional neural networks with transfer learning. *2016 IEEE International Conference on Image Processing (ICIP)*, 3713–3717.

Li, Y., Chang, M.-C., & Lyu, S. (2018). In ictu oculi: Exposing ai created fake videos by detecting eye blinking. *2018 IEEE International Workshop on Information Forensics and Security (WIFS)*, 1–7.

Li, Y., & Lyu, S. (2018). Exposing deepfake videos by detecting face warping artifacts. *ArXiv Preprint ArXiv:1811.00656*.

Li, Y., Yang, X., Sun, P., Qi, H., & Lyu, S. (2020). Celeb-df: A large-scale challenging dataset for deepfake forensics. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 3207–3216.

Librosa. (n.d.). *STFT*. https://librosa.org/doc/main/generated/librosa.stft.html?highlight=stft#librosa.stft

Liu, W., Wen, Y., Yu, Z., Li, M., Raj, B., & Song, L. (2017). Sphereface: Deep hypersphere embedding for face recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 212–220.

Liu, W., Wen, Y., Yu, Z., & Yang, M. (2016). Large-margin softmax loss for convolutional neural networks. *ICML*, *2*(3), 7.

Lyu, S. (2020). Deepfake detection: Current challenges and next steps. *2020 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, 1–6.

MachineLP. (n.d.). *Cos loss (Large Margin Cosine Loss) implementation in CosFace (tensorflow)*. https://www.programmersought.com/article/59094640230/

McFee, B., Raffel, C., Liang, D., Ellis, D. P. W., McVicar, M., Battenberg, E., & Nieto, O. (2015). librosa: Audio and music signal analysis in python. *Proceedings of the 14th Python in Science Conference*, *8*, 18–25.

Minaee, S., Luo, P., Lin, Z., & Bowyer, K. (2021). Going Deeper Into Face Detection: A Survey. *ArXiv Preprint ArXiv:2103.14983*.

Mittal, T., Bhattacharya, U., Chandra, R., Bera, A., & Manocha, D. (2020). Emotions Don't Lie: An Audio-Visual Deepfake Detection Method using Affective Cues. *Proceedings of the 28th ACM International Conference on Multimedia*, 2823–2832.

Moon, T. K. (1996). The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, *13*(6), 47–60.

MyHeritage. (n.d.). *Deep Nostalgia*. https://www.myheritage.es/deep-nostalgia

Nguyen, H. H., Yamagishi, J., & Echizen, I. (2019). Use of a capsule network to detect fake images and videos. *ArXiv Preprint ArXiv:1910.12467*.

Nguyen, T. T., Nguyen, C. M., Nguyen, D. T., Nguyen, D. T., & Nahavandi, S. (2019). Deep learning for deepfakes creation and detection. *ArXiv Preprint ArXiv:1909.11573*, *1*.

NIST. (2018). *Media Forensics Challenge 2019*. https://www.nist.gov/itl/iad/mig/media-forensics-challenge-2019-0

OpenCV. (2021). *OpenCV*. https://github.com/opencv/opencv/tree/4.5.2

OpenSLR. (2017). *Room Impulse Response and Noise Database*. http://www.openslr.org/28/

Palanisamy, K., Singhania, D., & Yao, A. (2020). Rethinking cnn models for audio classification. *ArXiv Preprint ArXiv:2007.11154*.

Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., & Le, Q. V. (2019). Specaugment: A simple data augmentation method for automatic speech recognition. *ArXiv Preprint ArXiv:1904.08779*.

Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). *Deep face recognition*.

Ping, W., Peng, K., Gibiansky, A., Arik, S. O., Kannan, A., Narang, S., Raiman, J., & Miller, J. (2017). Deep voice 3: Scaling text-to-speech with convolutional sequence learning. *ArXiv Preprint ArXiv:1710.07654*.

Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. *Ieee Assp Magazine*, *3*(1), 4–16.

Rosaj. (2020). *face_detection*. Github. https://github.com/rosaj/face_detection

Rössler, A., Cozzolino, D., Verdoliva, L., Riess, C., Thies, J., & Nießner, M. (2019). Face{F}orensics++: Learning to Detect Manipulated Facial Images. *International Conference on Computer Vision (ICCV)*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*.

Sabir, E., Cheng, J., Jaiswal, A., AbdAlmageed, W., Masi, I., & Natarajan, P. (2019). Recurrent convolutional strategies for face manipulation detection in videos. *Interfaces (GUI)*, *3*(1).

Sahidullah, M., Kinnunen, T., & Hanilçi, C. (2015). *A comparison of features for synthetic speech detection*.

Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *ArXiv Preprint ArXiv:1602.07868*.

Samuel, S. (2019). A guy made a deepfake app to turn photos of women into nudes. It didn't go well. *Vox*. https://www.vox.com/2019/6/27/18761639/ai-deepfake-deepnude-app-nude-women-porn

Sanderson, C., & Lovell, B. C. (2009). Multi-region probabilistic histograms for robust and scalable identity inference. *International Conference on Biometrics*, 199–208.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4510–4520.

Satterfield, A. (2020). *Visualize Filters and Feature Maps in VGG16 and VGG19 CNN Models*. https://morioh.com/p/383582dc31a6

Sem-Jacobsen, F. O., Skeie, T., Lysne, O., Tørudbakken, O., Rongved, E., & Johnsen, B.

(2005). Siamese-twin: A dynamically fault-tolerant fat-tree. *19th IEEE International Parallel and Distributed Processing Symposium*, 10--pp.

Sensity. (n.d.). *Sensity*. Retrieved May 3, 2021, from https://sensity.ai/

Tan, M., & Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *International Conference on Machine Learning*, 6105–6114.

Tan, M., & Le, Q. V. (2021). Efficientnetv2: Smaller models and faster training. *ArXiv Preprint ArXiv:2104.00298*.

Tolosana, R., Romero-Tapiador, S., Fierrez, J., & Vera-Rodriguez, R. (2020). Deepfakes evolution: Analysis of facial regions and fake detection performance. *ArXiv Preprint ArXiv:2004.07532*.

Van Rossum, G., & Drake Jr, F. L. (1995). *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.

Wang, H., Wang, Y., Zhou, Z., Ji, X., Gong, D., Zhou, J., Li, Z., & Liu, W. (2018). Cosface: Large margin cosine loss for deep face recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5265–5274.

Webopedia. (2021). *Equal Error Rate*. https://www.webopedia.com/definitions/equal-error-rate/

Wen, Y., Zhang, K., Li, Z., & Qiao, Y. (2016). A discriminative feature learning approach for deep face recognition. *European Conference on Computer Vision*, 499–515.

Xuan, X., Peng, B., Wang, W., & Dong, J. (2019). On the generalization of GAN image forensics. *Chinese Conference on Biometric Recognition*, 134–141.

Yamagishi, J., Todisco, M., Sahidullah, M., Delgado, H., Wang, X., Evans, N., Kinnunen, T., Lee, K. A., Vestman, V., & Nautsch, A. (2019). *Asvspoof 2019: The 3rd automatic speaker verification spoofing and countermeasures challenge database*.

Yang, M.-H., Kriegman, D. J., & Ahuja, N. (2002). Detecting faces in images: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(1), 34–58.

Yang, X., Li, Y., & Lyu, S. (2019). Exposing deep fakes using inconsistent head poses. *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 8261–8265.

Zadeh, A., Liang, P. P., Mazumder, N., Poria, S., Cambria, E., & Morency, L.-P. (2018). Memory fusion network for multi-view sequential learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1).

Zen, H., Tokuda, K., & Black, A. W. (2009). Statistical parametric speech synthesis. *Speech Communication*, *51*(11), 1039–1064.

Zhang, S., Zhu, X., Lei, Z., Shi, H., Wang, X., & Li, S. Z. (2017). Faceboxes: A CPU real-time face detector with high accuracy. *2017 IEEE International Joint Conference on Biometrics (IJCB)*, 1–9.

Zhou, H., Bai, X., & Du, J. (2018). An investigation of transfer learning mechanism for acoustic scene classification. *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, 404–408.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE International Conference on Computer Vision*, 2223–2232.

Zhu, Y., Cai, H., Zhang, S., Wang, C., & Xiong, Y. (2020). TinaFace: Strong but Simple Baseline for Face Detection. *ArXiv Preprint ArXiv:2011.13183*.

Zulko. (2017). *MoviePy*. https://zulko.github.io/moviepy/

Anexos

Anexo I. Módulo de pre-procesado

Video

A continuación, se muestra el listado de funciones que se utilizan para preprocesar los videos, extrayendo *frames* con *frames_videos* y posteriormente realizando los recortes de las caras de esos *frames* con *crop_faces*. También se incluyen funciones auxiliares y funciones para visualizar los datos y comprobar que todo está funcionando correctamente.

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
from os.path import split, join, exists, dirname
from os import makedirs

#Face detection
from face_boxes import FaceBoxes
from PIL import ImageDraw

def frames_videos(video_paths, images_path, frame_num, file_name='frames.csv')
:
    '''Extract individual frames from multiple videos

    Parameters
    ----------
    video_paths : [str]
        Paths to the videos
    images_path : str
        Path to store the generated images
    frame_num : int
        Number of individual frames to randomly select
    file_name : str
        Name of the file in which the paths will be stored

    Returns
    -------
    frames_list : list
        paths to all the images generated
    '''

    if not exists(dirname(file_name)):
        makedirs(dirname(file_name))
```

```
    if not exists(str(images_path)):
        makedirs(str(images_path))

    frames_list = []

    for video_path in video_paths:
        frames = extract_frames(video_path, images_path, frame_num)
        if not frames:
            continue

        with open(file_name, 'a') as file:
            for path in frames:
                file.write(path + '\n')

        frames_list.extend(frames)

    return frames_list

def extract_frames(video_path, images_path, frame_num):
    '''Extract individual frames from a video to have images for the data load
er

    Parameters
    ----------
    video_path : str
        Path to the video
    images_path : str
        Path to store the generated images
    frame_num : int
        Number of individual frames to randomly select

    Returns
    -------
    image_list : list
        paths to all the images generated
    '''

    cap = cv2.VideoCapture(video_path)

    # get total number of frames
    try:
        totalFrames = cap.get(cv2.CAP_PROP_FRAME_COUNT)
    except:
        print(video_path)
        return 0
    else:
        if not totalFrames:
            print(video_path)
```

```
            return 0

        image_list = []

        frame_indices = np.random.randint(0, totalFrames, frame_num)

        for frame_number in frame_indices:
            # set frame position
            cap.set(cv2.CAP_PROP_POS_FRAMES, frame_number)
            ret, frame = cap.read()

            if not ret:
                continue

            fr_name =  str(split(video_path)[-
1]).split('.')[0] + '_' + str(frame_number) + '.png'
            image_path = str(join(images_path, fr_name))
            cv2.imwrite(image_path, frame)
            image_list.append(image_path)

        # When everything done, release
        # the video capture object
        cap.release()
        cv2.destroyAllWindows()

        return image_list

def view_frame(video_path):
    '''View a random frame

    Parameters
    ----------
    video_path : str
        Path to the video
    '''
    cap = cv2.VideoCapture(video_path)

    # get total number of frames
    totalFrames = cap.get(cv2.CAP_PROP_FRAME_COUNT)

    frame_index = np.random.randint(0, totalFrames)
    # set frame position
    cap.set(cv2.CAP_PROP_POS_FRAMES, frame_index)
    ret, frame = cap.read()

    # When everything done, release
    # the video capture object
    cap.release()
    cv2.destroyAllWindows()
```

```
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(15, 15))
    plt.imshow(frame)

    return frame


def view_video(video_path):
    '''View video

    Parameters
    ----------
    video_path : str
        Path to the video
    '''
    cap = cv2.VideoCapture(video_path)

    while cap.isOpened():
        # Capture frame-by-frame
        ret, frame = cap.read()
        if ret == True:
            # Display the resulting frame
            cv2.imshow('Frame', frame)

            # Press Q on keyboard to  exit
            if cv2.waitKey(25) & 0xFF == ord('q'):
                break

        # Break the loop
        else:
            break

    # When everything done, release
    # the video capture object
    cap.release()
    # Closes all the frames
    cv2.destroyAllWindows()


def draw_predict(image, score, box):
    '''Draw the bounding box for faces detected

    Parameters
    ----------
    image: PIL image
    score: float, confidence score assigned to face
    box: [xmin, ymin, xmax, ymax]

    Returns
```

```
    -------
    image_copy: PIL image
    '''
    image_copy = image.copy()
    draw = ImageDraw.Draw(image_copy, 'RGBA')
    width, height = image.size


    xmin, ymin, xmax, ymax = box
    ymin = int(ymin)
    xmin = int(xmin)
    ymax = int(ymax)
    xmax = int(xmax)
    fill = (255, 0, 0, 45)
    outline = 'red'
    draw.rectangle(
        [(xmin, ymin), (xmax, ymax)],
        fill=fill, outline=outline
    )
    draw.text((xmin, ymin), text='{:.3f}'.format(score))
    return image_copy


def crop_faces(frame_list, path_out, buffer=50, file_name='crops.csv'):
    '''Crop faces from frames and save them

    Parameters
    ----------
    frame_list: [str]
        list of frames from which to extract faces
    path_out: str
        path to save images
    buffer: int
        amount by which to increase the crop (equally in all directions)
    file_name: str
        file in which to save crop paths and data for traceability

    Returns
    -------
    crop_list: list of crops
    '''
    if not exists(dirname(file_name)):
        makedirs(dirname(file_name))

    with open(file_name, mode='a') as file:
        file.write('Path, xmin, xmax, ymin, ymax, confidence\n')

    if not exists(str(path_out)):
        makedirs(str(path_out))
```

```
    crop_list = []

    for frame_path in frame_list:
        img = cv2.imread(frame_path)
        faces = FaceBoxes.detect_faces(img)

        if not faces:
            cv2.imwrite(str(join(path_out, 'review', crop_name)), img)
            continue

        for i, face in enumerate(faces):
            if face.conf<0.9:
                continue

            if not face.bbox:
                cv2.imwrite(str(join(path_out, 'review', crop_name)), img)
                continue

            xmin, ymin, xmax, ymax = face.bbox
            ymin = int(ymin)
            xmin = int(xmin)
            ymax = int(ymax)
            xmax = int(xmax)

            if xmin==xmax or ymin==ymax:
                cv2.imwrite(str(join(path_out, 'review', crop_name)), img)
                continue

            height, width, _ = img.shape

            if ymin<0:
                ymin = 0
            if xmin<0:
                xmin = 0
            if ymax>height:
                ymax = height
            if xmax>width:
                xmax = width

            ylow = ymin-buffer//2
            yhigh = ymax+buffer//2
            xlow = xmin-buffer//2
            xhigh = xmax+buffer//2

            if ylow<0:
                ylow = ymin
            if xlow<0:
                xlow = xmin
```

```
            if yhigh>height:
                yhigh = ymax
            if xhigh>width:
                xhigh = xmax

            crop = img[ylow:yhigh, xlow:xhigh]

            if crop is None:
                cv2.imwrite(str(join(path_out, 'review', crop_name)), img)
                continue

            crop_name =  str(split(frame_path)[-
1]).split('.')[0] + '_face' + str(i) + '.png'
            crop_path = str(join(path_out, crop_name))
            cv2.imwrite(crop_path, crop)
            crop_list.append(crop_path)
            with open(file_name, 'a') as file:
                file.write(f'{crop_path}, {xmin}, {xmax}, {ymin}, {ymax}, {fac
e.conf}\n')

    return crop_list
```

Por otro lado, a continuación se muestra el código para realizar las *augmentations* de las imágenes:

```
from tensorflow.python.keras.utils.data_utils import Sequence
import numpy as np
import cv2

class DeepfakeSeq(Sequence):
    def __init__(self, x_set, y_set, batch_size, augmentations, absolute_path)
:
        self.x, self.y = x_set, y_set
        self.batch_size = int(batch_size)
        self.augment = augmentations
        self.absolute = absolute_path

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))

    def __getitem__(self, idx):

        batch_x_path = self.x[idx * self.batch_size:(idx + 1) * self.batch_siz
e]
```

```
        batch_x_path = [path.replace('..', self.absolute) for path in batch_x_
path]
        batch_x = [cv2.cvtColor(cv2.imread(path), cv2.COLOR_RGB2BGR) for path
in batch_x_path]

        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]

        return np.stack([
            self.augment(image=x)["image"] for x in batch_x
        ], axis=0), np.array(batch_y)
```

Audio

Para el módulo de audio, la conversión a LFCCs se ha realizado partiendo de las funciones de conversión a MFCCs de librosa, por lo que no se reproduce el código aquí, que consiste únicamente en modificar los filtros de escala Mel a filtros triangulares. Por otro lado, sí se reproduce aquí el código utilizado para realizar las *augmentations*, mostrando únicamente la función en su estado final, es decir, generando LFCCs del audio entero.

```
from tensorflow.python.keras.utils.data_utils import Sequence
import numpy as np
from cv2 import resize, INTER_AREA, cvtColor, COLOR_GRAY2RGB, normalize, NORM_
MINMAX, CV_32F
import librosa
from audio_lin import lfcc

class DeepfakeAudSeq(Sequence):
    def __init__(self, x_set, y_set, batch_size, augmentations_pre, augmentati
ons_post, absolute_path):
        self.x, self.y = x_set, y_set
        self.batch_size = int(batch_size)
        self.augment_pre = augmentations_pre
        self.augment_post = augmentations_post
        self.absolute = absolute_path

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))

    def __getitem__(self, idx):

        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = [1 if label=='bonafide' else 0 for label in batch_y]

        batch_x_path = self.x[idx * self.batch_size:(idx + 1) * self.batch_siz
e]
```

```
        batch_x_path = [self.absolute+file_name+'.flac' for file_name in batch
_x_path]

        batch_x = [librosa.load(path, sr=None) for path in batch_x_path]
        sample_rate = batch_x[0][1]
        batch_x = [wave[0] for wave in batch_x]
        batch_x = [self.augment_pre(samples=x, sample_rate=sample_rate) for x
in batch_x]

        pad_batch_x = []
        #pad_batch_y = []
        for wave in batch_x:
            lf_wave = lfcc(y=wave, sr=sample_rate, n_lfcc=20, n_fft=512, hop_l
ength=160)
            pad_batch_x.append(lf_wave)



        if self.augment_post!=None:
            pad_batch_x = [self.augment_post(magnitude_spectrogram=x) for x in
 pad_batch_x]

        dim = (224, 224)
        pad_batch_x = [cvtColor(resize(img, dim, interpolation = INTER_AREA),
COLOR_GRAY2RGB) for img in pad_batch_x]
        pad_batch_x = [normalize(img, None, alpha=0, beta=1, norm_type=NORM_MI
NMAX, dtype=CV_32F) for img in pad_batch_x]
        return np.stack(pad_batch_x, axis=0), np.array(batch_y)
```

Anexo II. Módulo de entrenamiento

Video

A continuación, se muestra un *jupyter notebook* con el código utilizado en el entrenamiento del modelo de video.

[ ]:
```
#Common libraries
import numpy as np
import pandas as pd
from cv2 import BORDER_REPLICATE, BORDER_CONSTANT, normalize, NORM_MINMAX,␣
 ↪CV_32F
import json
import random
import matplotlib.pyplot as plt

#Model
import tensorflow as tf
import tensorflow_addons as tfa
import keras.backend as K
import effnetv2_model
from sklearn.metrics import confusion_matrix, roc_auc_score

#seeds
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

#augmentations
from albumentations import (Normalize,
    Compose, ImageCompression, GaussNoise,
    GaussianBlur, HorizontalFlip, OneOf,
    RandomBrightnessContrast, FancyPCA,
    ShiftScaleRotate, GridDropout,
    ToGray, Resize
)

#custom functions
from data_handling.augmentations import DeepfakeSeq #sequence to augment data
from data_handling.data_loading import display_samples #function to display data
from data_handling.loss_funct import CosFace, plot_acc_loss #loss
from data_handling.wgt_norm import WeightNormL2 #weight normalization
```

1 Transfer learning

Primeramente se entrena el modelo con los pesos de efficientNet congelados, utilizando el dataset equilibrado.

[ ]:
```
data = pd.read_csv(r'D:
↪\UNIR\Master_Inteligencia_Artificial\TFM\clean_data\crops\reduced_train_val_data
↪csv')

X = data.Path.to_list()
y = data.Label.to_list()

#división de los datos en validación y entrenamiento 80/20:
X_train = X[:int(np.ceil(len(X)*0.8))]
X_val = X[int(np.ceil(len(X)*0.8)):]

y_train = y[:int(np.ceil(len(y)*0.8))]
y_val = y[int(np.ceil(len(y)*0.8)):]
```

[ ]:
```
#Carpeta en la que se encuentran los datos
absolute_path = 'D:\\UNIR\\Master_Inteligencia_Artificial\\TFM'
```

[ ]:
```
#plot samples
display_samples(X_train, y_train, absolute_path)
```

[ ]:
```
#Augmentations
AUGMENTATIONS_TRAIN = Compose([
    ImageCompression(quality_lower=60, quality_upper=100, p=0.4),
    GaussNoise(p=0.1),
    HorizontalFlip(p=0.5),
    OneOf([RandomBrightnessContrast(), FancyPCA()], p=0.7),
    ShiftScaleRotate(border_mode=BORDER_CONSTANT, p=0.6),
    GridDropout(p=0.2),
    ToGray(p=0.1),
    Resize(275, 225, p=1),
    Normalize()
])

AUGMENTATIONS_TEST = Compose([
    Resize(275, 225, p=1),
    Normalize()
])
```

[ ]:
```
train_batch_size = 32
val_batch_size = 32

train_examples = len(X_train)
```

```
valid_examples = len(X_val)

# Training data
train_gen = DeepfakeSeq(X_train[:train_examples], y_train[:train_examples],␣
↪train_batch_size, augmentations=AUGMENTATIONS_TRAIN,
                        absolute_path=absolute_path)

# Validation data
valid_gen = DeepfakeSeq(X_val[:valid_examples], y_val[:valid_examples],␣
↪val_batch_size, augmentations=AUGMENTATIONS_TEST,
                        absolute_path=absolute_path)

# Define callbacks
checkpoint_filepath = absolute_path + '\\models\\tmp\\checkpoint_video_model'

callbacks = [
            tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,␣
↪mode='min', restore_best_weights=True),
            tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath,␣
↪save_weights_only=True, monitor='val_accuracy',
                                               mode='max', save_freq='epoch'
                                               )
            ]
```

[ ]:
```
#Batch 0 del dataset con augmentations:
batch_0, label = train_gen.__getitem__(0)

#Mostrar imágenes augmented
rows=4
columns=8
fig, ax = plt.subplots(rows, columns, figsize=(20,10), sharex=True, sharey=True)
images = iter(batch_0)
for i in range(rows):
    for j in range(columns):
        ax[i, j].imshow(normalize(next(images), None, alpha=0, beta=1,␣
↪norm_type=NORM_MINMAX, dtype=CV_32F))
plt.show()
```

1.0.1 Modelo softmax

[ ]:
```
#Softmax model
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=[225, 275, 3]),
    effnetv2_model.get_model('efficientnetv2-s', include_top=False,␣
↪pretrained=True, training=False),
    tf.keras.layers.Dropout(rate=0.3),
```

```
    tf.keras.layers.Dense(64, activation='relu',␣
 ↪kernel_initializer='GlorotUniform',
                          kernel_regularizer='l2', name='Dense1'),
    tf.keras.layers.Dense(2, activation='softmax',␣
 ↪kernel_initializer='GlorotUniform', name='Dense2')
])

loss_f = "sparse_categorical_crossentropy"
```

1.0.2 Modelo LMCL

[ ]:
```
#LMCL
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=[225, 275, 3], name='inputs'),
    effnetv2_model.get_model('efficientnetv2-s', include_top=False,␣
 ↪pretrained=True, training=False),
    tf.keras.layers.Dropout(rate=0.3, name='Dropout'),
    tf.keras.layers.Dense(64, kernel_initializer='GlorotUniform',␣
 ↪name='Representation'),
    tf.keras.layers.Lambda(lambda x: K.l2_normalize(x, axis=1), name='Lambda'),
    WeightNormL2(tf.keras.layers.Dense(2, use_bias=False), name='Dense_w_Norm')
])

loss_f = CosFace(m=0.35, s=64.0, n_classes=2)
```

1.0.3 Entrenamiento

[ ]:
```
model.get_layer('efficientnetv2-s').trainable = False

model.summary()
```

[ ]:
```
learning_rate=0.001
epochs = 7

optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)␣
 ↪#optimizador

model.compile(loss=loss_f, optimizer=optimizer, metrics=["accuracy"])

#entrenar
history = model.fit(train_gen,
                    epochs = epochs,
                    validation_data=valid_gen,
                    callbacks=callbacks,
                    workers=1)
```

```
[ ]: plot_acc_loss(history, learning_rate, optimizer, train_batch_size,␣
     ↪train_examples, valid_examples)
```

```
[ ]: #Evaluación final
     loss, acc = model.evaluate(train_gen, verbose = 0)
     loss_val, acc_val = model.evaluate(valid_gen, verbose = 0)

     print("Training loss {:.5f} and accuracy {:.2f}%".format(loss,acc*100))
     print("Validation loss {:.5f} and accuracy {:.2f}%".
     ↪format(loss_val,acc_val*100))
```

Guardar el modelo:

```
[ ]: model_path = absolute_path + '\\models\\eff_net_1.h5py'
     model.save(
         model_path,
         overwrite=True,
         include_optimizer=True,
         save_traces=True,
     )
```

```
[ ]: metadata = {'summary': 'Dropout 0.3, Dense 64 w/ RELU & l2 norm, Dense 2 w/␣
     ↪softmax',
                 'train_data': train_examples,
                 'val_data': valid_examples,
                 'learning_rate': learning_rate,
                 'batch_size': train_batch_size,
                 'epochs': epochs,
                 'optimizer': 'RMSprop',
                 'loss function': loss_f,
                 'tr_loss': loss,
                 'tr_acc': acc,
                 'val_loss': loss_val,
                 'val_acc': acc_val}

     json_path = f'{absolute_path}\\models\\model_metadata_eff_net_1.json'
     with open(json_path, "w+") as outfile:
         json.dump(metadata, outfile)
```

1.1 Fine-Tunning

```
[ ]: data = pd.read_csv(r'D:
     ↪\UNIR\Master_Inteligencia_Artificial\TFM\clean_data\crops\train_val_data_face0.
     ↪csv')
     data = data.sample(frac=1).reset_index(drop=True)

     X = data.Path.to_list()
```

```
y = data.Label.to_list()

X_train = X[:int(np.ceil(len(X)*0.8))]
X_val = X[int(np.ceil(len(X)*0.8)):]

y_train = y[:int(np.ceil(len(y)*0.8))]
y_val = y[int(np.ceil(len(y)*0.8)):]
```

[ ]:
```
train_batch_size = 32
val_batch_size = 32

train_examples = len(X_train)
valid_examples = len(X_val)

# Training data
train_gen = DeepfakeSeq(X_train[:], y_train[:], train_batch_size,␣
 ↪augmentations=AUGMENTATIONS_TRAIN, absolute_path=absolute_path)

# Validation data
valid_gen = DeepfakeSeq(X_val[:], y_val[:], val_batch_size,␣
 ↪augmentations=AUGMENTATIONS_TEST, absolute_path=absolute_path)

# Define callbacks
checkpoint_filepath = absolute_path +␣
 ↪'\\models\\tmp\\checkpoint_video_model_fin_tune'

callbacks = [
            tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3,␣
 ↪mode='min', restore_best_weights=True),
            tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath,␣
 ↪save_weights_only=True, monitor='val_accuracy',
                                               mode='max', save_freq='epoch'
                                               )
            ]
```

[ ]:
```
model.get_layer('efficientnetv2-s').trainable = True

model.summary()
```

[ ]:
```
learning_rate=0.001
epochs = 7

optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)␣
 ↪#optimizador

model.compile(loss=loss_f, optimizer=optimizer, metrics=["accuracy"])
```

```
#entrenar
history = model.fit(train_gen,
                    epochs = epochs,
                    validation_data=valid_gen,
                    callbacks=callbacks,
                    workers=1)
```

[ ]:
```
plot_acc_loss(history, learning_rate, optimizer, train_batch_size,␣
↪train_examples, valid_examples)
```

[ ]:
```
loss, acc = model.evaluate(train_gen, verbose = 0)
loss_val, acc_val = model.evaluate(valid_gen, verbose = 0)

print("Training loss {:.5f} and accuracy {:.2f}%".format(loss,acc*100))
print("Validation loss {:.5f} and accuracy {:.2f}%".
↪format(loss_val,acc_val*100))
```

1.1.1 Guardar el modelo

[ ]:
```
model_path = absolute_path + '\\models\\eff_net_1_fine_tune.h5py'
model.save(
    model_path,
    overwrite=True,
    include_optimizer=True,
    save_traces=True,
)

metadata = {'summary': 'Dropout 0.3, Dense 64 w/ RELU & l2 norm, Dense 2 w/␣
↪softmax',
            'train_data': train_examples,
            'val_data': valid_examples,
            'learning_rate': learning_rate,
            'batch_size': train_batch_size,
            'epochs': epochs,
            'optimizer': 'RMSprop',
            'loss function': loss_f,
            'tr_loss': loss,
            'tr_acc': acc,
            'val_loss': loss_val,
            'val_acc': acc_val}

json_path = f'{absolute_path}\\models\\model_metadata_eff_net_1_fine_tune.json'
with open(json_path, "w+") as outfile:
    json.dump(metadata, outfile)
```

2 Evaluación

2.0.1 Validación

```
[ ]: predicciones = model.predict(valid_gen)
     preds = np.argmax(predicciones, axis=1)
```

```
[ ]: auc_score = roc_auc_score(y_val[:100], preds)
     print(f'\033[1mAUC score:\033[0m {np.round(auc_score, 2)}\n')

     print('\033[1mConfusion Matrix:\033[0m')
     print('               | True Labels      |')
     print('False Labels   |  FAKE   |  REAL  |')
     conf_m = confusion_matrix(y_val[:100], preds)
     print(f'   FAKE        |  {conf_m[0][0]}  |   {conf_m[0][1]}  |')
     print(f'   REAL        |   {conf_m[1][0]}  |  {conf_m[1][1]}  |')
```

2.0.2 Testeo

```
[ ]: data = pd.read_csv(r'D:
      ↪\UNIR\Master_Inteligencia_Artificial\TFM\clean_data\crops\test_data.csv')

     X_test = data.Path.to_list()
     y_test = data.Label.to_list()

     AUGMENTATIONS_TEST = Compose([
         Resize(275, 225, p=1),
         Normalize()
     ])

     test_batch_size = 32
     test_examples = len(X_test)

     # Validation data
     test_gen = DeepfakeSeq(X_test, y_test, test_batch_size,␣
      ↪augmentations=AUGMENTATIONS_TEST,
                            absolute_path=absolute_path)
```

```
[ ]: predicciones = model.predict(test_gen)
     preds = np.argmax(predicciones, axis=1)
```

```
[ ]: auc_score = roc_auc_score(y_test, preds)
     print(f'\033[1mAUC score:\033[0m {np.round(auc_score, 2)}\n')

     print('\033[1mConfusion Matrix:\033[0m')
     print('               | True Labels      |')
     print('False Labels   |  FAKE   |  REAL  |')
```

```
     conf_m = confusion_matrix(y_test, preds)
     print(f'   FAKE        |  {conf_m[0][0]}  |   {conf_m[0][1]}  |')
     print(f'   REAL        |   {conf_m[1][0]}  |  {conf_m[1][1]}  |')
```

Audio

A continuación, se muestra un *jupyter notebook* con el código utilizado en el entrenamiento del modelo de audio.

[ ]:
```
#Common libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import json
import random

#Model
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
import keras.backend as K
from sklearn.metrics import confusion_matrix, roc_auc_score
#seeds
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

#augmentations
from audiomentations import (Compose, SpecCompose, AddGaussianNoise,␣
↪ApplyImpulseResponse, AddBackgroundNoise, AddShortNoises,
                             Shift, TimeStretch, PitchShift, Mp3Compression,␣
↪TimeMask, SpecFrequencyMask, Normalize)

#custom functions
from audio_augm import DeepfakeAudSeq
from audio_loss import CosFace, plot_acc_loss
from weight_norm import WeightNormL2
```

1 Transfer learning 1

Importar datos y hacer RUS

[ ]:
```
#Training data:
train_data = pd.read_csv('..
↪\\datasets\\LA\\LA\\ASVspoof2019_LA_cm_protocols\\ASVspoof2019.LA.cm.train.
↪trn.csv', delimiter=' ', header=None)
```

```
train_data.columns = ['Subject', 'Video_name', '-', 'Spoof_technique',␣
↪'Bonafide_Spoof']
true_d_train = len(train_data[train_data.Bonafide_Spoof=='bonafide'])
fake_train = train_data[train_data.Bonafide_Spoof=='spoof'].
↪sample(n=true_d_train).reset_index(drop=True)
real_train = train_data[train_data.Bonafide_Spoof=='bonafide'].sample(frac=1).
↪reset_index(drop=True)
train_reduced = pd.concat([fake_train, real_train]).reset_index(drop=True)
train_reduced = train_reduced.sample(frac=1).reset_index(drop=True)
X_train = train_reduced.Video_name.to_list()
y_train = train_reduced.Bonafide_Spoof.to_list()

#Validation data
val_data = pd.read_csv('..
↪\\datasets\\LA\\LA\\ASVspoof2019_LA_cm_protocols\\ASVspoof2019.LA.cm.dev.trl.
↪csv', delimiter=' ', header=None)
val_data.columns = ['Subject', 'Video_name', '-', 'Spoof_technique',␣
↪'Bonafide_Spoof']
true_d_val = len(val_data[val_data.Bonafide_Spoof=='bonafide'])
fake_val = val_data[val_data.Bonafide_Spoof=='spoof'].sample(n=true_d_val).
↪reset_index(drop=True)
real_val = val_data[val_data.Bonafide_Spoof=='bonafide'].sample(frac=1).
↪reset_index(drop=True)
val_reduced = pd.concat([fake_val, real_val]).reset_index(drop=True)
val_reduced = val_reduced.sample(frac=1).reset_index(drop=True)
X_val = val_reduced.Video_name.to_list()
y_val = val_reduced.Bonafide_Spoof.to_list()
```

[ ]:
```
#Carpetas en las que se encuentran los datos
train_path = '..\\datasets\\LA\\LA\\ASVspoof2019_LA_train\\flac\\'
val_path = '..\\datasets\\LA\\LA\\ASVspoof2019_LA_dev\\flac\\'

absolute_path = 'D:\\UNIR\\Master_Inteligencia_Artificial\\TFM'
```

[ ]:
```
#Augmentations entrenamiento
AUGMENTATIONS_PRE_TRAIN = Compose([
    AddGaussianNoise(p=0.1),
    ApplyImpulseResponse(ir_path=r'D:
↪\UNIR\Master_Inteligencia_Artificial\TFM\clean_data\rirs_noises\RIRS_NOISES\real
↪p=0.5, leave_length_unchanged=True),
    Shift(p=0.3, fade=True),
    TimeStretch(p=0.5),
    PitchShift(p=0.3),
    Normalize(p=1.0)
])
```

```
AUGMENTATIONS_POST_TRAIN = SpecCompose([
    SpecFrequencyMask(p=0.5, fill_mode='constant', fill_constant=0.0)
])

#augmentations validación
AUGMENTATIONS_PRE_TEST = Compose([
    Normalize(p=1.0)
])
```

[ ]:
```
train_batch_size = 32
val_batch_size = 32

train_examples = len(X_train)
valid_examples = len(X_val)

# Training data
train_gen = DeepfakeAudSeq(X_train, y_train, train_batch_size,
                           augmentations_pre=AUGMENTATIONS_PRE_TRAIN,␣
 ↪augmentations_post=AUGMENTATIONS_POST_TRAIN,
                           absolute_path=train_path)

# Validation data
valid_gen = DeepfakeAudSeq(X_val, y_val, val_batch_size,
                           augmentations_pre=AUGMENTATIONS_PRE_TEST,␣
 ↪augmentations_post=None,
                           absolute_path=val_path)

# Define callbacks
checkpoint_filepath = absolute_path +␣
 ↪'\\models\\tmp_audio\\checkpoint_audio_model_LFCC{epoch}'

callbacks = [
            tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=1,␣
 ↪mode='min', restore_best_weights=True),
            tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath,␣
 ↪save_weights_only=True, monitor='val_accuracy',
                                               mode='max', save_freq='epoch'
                                               )
            ]
```

[ ]:
```
#Batch 0 del dataset con augmentations:
batch_0, label = train_gen.__getitem__(0)

#Mostrar imágenes augmented
rows=4
columns=8
```

```
fig, ax = plt.subplots(rows, columns, figsize=(20,10), sharex=True, sharey=True)
images = iter(batch_0)
for i in range(rows):
    for j in range(columns):
        ax[i, j].imshow(next(images))
plt.show()
```

1.0.1 Modelo softmax

[ ]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=[224, 224, 3]),
    MobileNetV2(weights='imagenet', include_top=False, alpha=1.0,␣
 ↪pooling='avg'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Dense(512, activation='relu',␣
 ↪kernel_initializer='GlorotUniform',
                          kernel_regularizer='l2', name='Dense'),
    tf.keras.layers.Dense(2, activation='softmax',␣
 ↪kernel_initializer='GlorotUniform', name='Prediction')
])

loss_f = "sparse_categorical_crossentropy"
```

1.0.2 Modelo LMCL

[ ]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=[224, 224, 3]),
    MobileNetV2(weights='imagenet',include_top=False, alpha=1.0, pooling='avg'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Dense(64, kernel_initializer='GlorotUniform',␣
 ↪name='Dense1'),
    tf.keras.layers.Lambda(lambda x: K.l2_normalize(x, axis=1), name='Lambda2'),
    WeightNormL2(tf.keras.layers.Dense(2, use_bias=False), name='Dense_w_Norm')
])

loss_f = CosFace(m=0.35, s=4.0, n_classes=2)
```

1.0.3 Entrenamiento

[ ]:
```
model.get_layer('mobilenetv2_1.00_224').trainable = False

model.summary()
```

[ ]:
```
learning_rate=0.0001
epochs = 10
```

```
optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)␣
 ↪#optimizador
#optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

model.compile(loss=loss_f, optimizer=optimizer, metrics=["accuracy"])

#entrenar
history = model.fit(train_gen,
                    epochs = epochs,
                    validation_data=valid_gen,
                    callbacks=callbacks,
                    workers=2)
```

[ ]:
```
plot_acc_loss(history, learning_rate, optimizer, train_batch_size,␣
 ↪train_examples, valid_examples)
```

[ ]:
```
loss, acc = model.evaluate(train_gen, verbose = 0)
loss_val, acc_val = model.evaluate(valid_gen, verbose = 0)

print("Training loss {:.5f} and accuracy {:.2f}%".format(loss,acc*100))
print("Validation loss {:.5f} and accuracy {:.2f}%".
 ↪format(loss_val,acc_val*100))
```

2 Entrenamiento con LR menor y más datos

[ ]:
```
train_data = pd.read_csv('..
 ↪\\datasets\\LA\\LA\\ASVspoof2019_LA_cm_protocols\\ASVspoof2019.LA.cm.train.
 ↪trn.csv', delimiter=' ', header=None)
train_data.columns = ['Subject', 'Video_name', '-', 'Spoof_technique',␣
 ↪'Bonafide_Spoof']
true_d_train = 4*len(train_data[train_data.Bonafide_Spoof=='bonafide'])
fake_train = train_data[train_data.Bonafide_Spoof=='spoof'].
 ↪sample(n=true_d_train).reset_index(drop=True)
real_train = train_data[train_data.Bonafide_Spoof=='bonafide'].sample(frac=1).
 ↪reset_index(drop=True)
train_reduced = pd.concat([real_train, real_train, fake_train, real_train]).
 ↪reset_index(drop=True)
train_reduced = train_reduced.sample(frac=1).reset_index(drop=True)
X_train = train_reduced.Video_name.to_list()
y_train = train_reduced.Bonafide_Spoof.to_list()

val_data = pd.read_csv('..
 ↪\\datasets\\LA\\LA\\ASVspoof2019_LA_cm_protocols\\ASVspoof2019.LA.cm.dev.trl.
 ↪csv', delimiter=' ', header=None)
```

```
val_data.columns = ['Subject', 'Video_name', '-', 'Spoof_technique',␣
↪'Bonafide_Spoof']
true_d_val = 4*len(val_data[val_data.Bonafide_Spoof=='bonafide'])
fake_val = val_data[val_data.Bonafide_Spoof=='spoof'].sample(n=true_d_val).
↪reset_index(drop=True)
real_val = val_data[val_data.Bonafide_Spoof=='bonafide'].sample(frac=1).
↪reset_index(drop=True)
val_reduced = pd.concat([fake_val, real_val]).reset_index(drop=True)
val_reduced = val_reduced.sample(frac=1).reset_index(drop=True)
X_val = val_reduced.Video_name.to_list()
y_val = val_reduced.Bonafide_Spoof.to_list()
```

[ ]:
```
train_batch_size = 32
val_batch_size = 32

train_examples = len(X_train)
valid_examples = len(X_val)

# Training data
train_gen = DeepfakeAudSeq(X_train, y_train, train_batch_size,
                           augmentations_pre=AUGMENTATIONS_PRE_TRAIN,␣
↪augmentations_post=AUGMENTATIONS_POST_TRAIN,
                           absolute_path=train_path)

# Validation data
valid_gen = DeepfakeAudSeq(X_val, y_val, val_batch_size,
                           augmentations_pre=AUGMENTATIONS_PRE_TEST,␣
↪augmentations_post=None,
                           absolute_path=val_path)

# Define callbacks
checkpoint_filepath = absolute_path +␣
↪'\\models\\tmp_audio\\checkpoint_audio_model2_LFCC{epoch}'

callbacks = [
            tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=1,␣
↪mode='min', restore_best_weights=True),
            tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath,␣
↪save_weights_only=True, monitor='val_accuracy',
                                               mode='max', save_freq='epoch'
                                               )
            ]
```

[ ]:
```
learning_rate=0.00001
epochs = 10
```

```
optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)␣
 ↪#optimizador
#optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

model.compile(loss=loss_f, optimizer=optimizer, metrics=["accuracy"])

#entrenar
history = model.fit(train_gen,
                    epochs = epochs,
                    validation_data=valid_gen,
                    callbacks=callbacks,
                    workers=2)
```

[ ]:
```
plot_acc_loss(history, learning_rate, optimizer, train_batch_size,␣
 ↪train_examples, valid_examples)
```

[ ]:
```
loss, acc = model.evaluate(train_gen, verbose = 0)
loss_val, acc_val = model.evaluate(valid_gen, verbose = 0)

print("Training loss {:.5f} and accuracy {:.2f}%".format(loss,acc*100))
print("Validation loss {:.5f} and accuracy {:.2f}%".
 ↪format(loss_val,acc_val*100))
```

3 Evaluación

3.1 Validación

[ ]:
```
predicciones = model.predict(val_gen)
preds = np.argmax(predicciones, axis=1)
```

[ ]:
```
y_labels = [1 if i=='bonafide' else 0  for i in y_val]
auc_score = roc_auc_score(y_labels, preds)
print(f'\033[1mAUC score:\033[0m {np.round(auc_score, 2)}\n')

print('\033[1mConfusion Matrix:\033[0m')
print('              | Pred Labels      |')
print('True Labels  |  FAKE   |  REAL  |')
conf_m = confusion_matrix(y_labels, preds)
print(f'   FAKE       |  {conf_m[0][0]}  |   {conf_m[0][1]}  |')
print(f'   REAL       |   {conf_m[1][0]}  |  {conf_m[1][1]}  |')
```

4 Testeo

```
[ ]: test_data = pd.read_csv('..
     ↪\\datasets\\LA\\LA\\ASVspoof2019_LA_cm_protocols\\ASVspoof2019.LA.cm.eval.
     ↪trl.csv', delimiter=' ', header=None)
     test_data.columns = ['Subject', 'Video_name', '-', 'Spoof_technique',␣
     ↪'Bonafide_Spoof']
     X_test = test_data.Video_name.to_list()
     y_test = test_data.Bonafide_Spoof.to_list()

     test_path = '..\\datasets\\LA\\LA\\ASVspoof2019_LA_eval\\flac\\'

     AUGMENTATIONS_PRE_TEST = Compose([
         Normalize(p=1.0)
     ])

     test_batch_size = 32

     test_examples = len(X_test)

     # Test data
     test_gen = DeepfakeAudSeq(X_test[:], y_test[:], test_batch_size,
                               augmentations_pre=AUGMENTATIONS_PRE_TEST,␣
     ↪augmentations_post=None,
                               absolute_path=test_path)
```

```
[ ]: predicciones = model.predict(test_gen)
     preds = np.argmax(predicciones, axis=1)
```

```
[ ]: y_labels = [1 if i=='bonafide' else 0  for i in y_test]
     auc_score = roc_auc_score(y_labels, preds)
     print(f'\033[1mAUC score:\033[0m {np.round(auc_score, 2)}\n')

     print('\033[1mConfusion Matrix:\033[0m')
     print('              | Pred Labels      |')
     print('True Labels  |  FAKE   |  REAL  |')
     conf_m = confusion_matrix(y_labels, preds)
     print(f'   FAKE       |  {conf_m[0][0]}  |   {conf_m[0][1]}  |')
     print(f'   REAL       |   {conf_m[1][0]}  |  {conf_m[1][1]}  |')
```

5 Guardar el modelo

```
[ ]: model_path = absolute_path + '\\models\\mobilenet_3.h5py'
     model.save(
         model_path,
         overwrite=True,
```

```
    include_optimizer=True,
    save_traces=True,
)
```

[ ]:
```
metadata = {'summary': 'Dropout 0.3, Dense 512 w/ RELU & l2 norm, Dense 2 w/␣
↪softmax',
            'train_data': train_examples,
            'val_data': valid_examples,
            'learning_rate': learning_rate,
            'batch_size': train_batch_size,
            'epochs': epochs,
            'optimizer': 'RMSprop',
            'loss function': loss_f,
            'tr_loss': loss,
            'tr_acc': acc,
            'val_loss': loss_val,
            'val_acc': acc_val}

json_path = f'{absolute_path}\\models\\model_metadata_mobilenet_3.json'
with open(json_path, "w+") as outfile:
    json.dump(metadata, outfile)
```

Anexo III. Módulo de inferencia

Para la parte de inferencia de video, se adaptan las funciones y *data loaders* anteriormente mostrados para que puedan trabajar a partir de video, en lugar de a partir de imágenes pre-procesadas. Esto se consigue combinando las funciones que se han mostrado anteriormente. Para la parte de audio, también se hacen modificaciones al *data loader* para que pueda extraer el audio directamente del video. A continuación, se muestran los fragmentos importantes del código.

Clase de clasificación

```
import tensorflow as tf
from efficientnetv2 import effnetv2_model
from tensorflow.keras.applications import MobileNetV2

from models.Video_class import extract_image
from models.Audio_class import extract_lfcc
import numpy as np

import warnings
warnings.filterwarnings("ignore")

import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"


class DeepfakeDetector:
    def __init__(self):

        #Video Model
        self.video_model = tf.keras.models.Sequential([
                                            effnetv2_model.EffNetV2Model('effi
cientnetv2-s', include_top=False),
                                            tf.keras.layers.Dropout(rate=0.3),
                                            tf.keras.layers.Dense(64, activati
on='relu', kernel_initializer='GlorotUniform',
                                                                  kernel_reg
ularizer='l2', name='Dense1'),
                                            tf.keras.layers.Dense(2, activatio
n='softmax', kernel_initializer='GlorotUniform', name='Dense2')
                                        ])

        self.video_model.load_weights('models\\video_model\\variables')
        self.video_model.compile()

        #Audio Model
```

```
        self.audio_model = tf.keras.models.Sequential([
                                                tf.keras.layers.InputLayer(inp
ut_shape=[224, 224, 3]),
                                                MobileNetV2(weights='imagenet'
, include_top=False, alpha=1.0, pooling='avg'),
                                                tf.keras.layers.Dropout(rate=0
.3),
                                                tf.keras.layers.Dense(512, act
ivation='relu', kernel_initializer='GlorotUniform',
                                                                      kernel_reg
ularizer='l2', name='Dense'),
                                                tf.keras.layers.Dense(2, activ
ation='softmax', kernel_initializer='GlorotUniform', name='Prediction')
                                            ])

        self.audio_model.load_weights('models\\audio_model\\checkpoint_audio_m
odelLFCC3')
        self.audio_model.compile()


    def deepfake_detector(self, path, video_frames=10, audio_frames=5, audio_l
ength=1):
        '''Function to detect whether a video is manipulated. Detects both man
ipulated audio and images

        Parameters
        ----------
        path: str, path to the video
        video_frames: int, number of video frames to use
        audio_frames: int, number of audio frames to use
        audio_length: int or float, length of audio frame
        '''
        if not isinstance(video_frames, int) or not isinstance(audio_frames, i
nt):
            print('Number of frames must be an integer!')
            return 0

        if not isinstance(audio_length, int) and not isinstance(audio_length,
float):
            print('Audio length must be a number!')
            return 0

        label_encoding = {0:'manipulated', 1:'pristine'}

        imgs = extract_image(path, video_frames)
        if isinstance(imgs, int) and not imgs:
            return 0

        v_pred_percent = self.video_model.predict(imgs)
```

```
        video_cert = np.mean(v_pred_percent, axis=0)
        video_pred = np.argmax(video_cert)

        audios = extract_lfcc(path, audio_length, audio_frames)
        a_pred_percent = self.audio_model.predict(audios)
        audio_cert = np.mean(a_pred_percent, axis=0)
        audio_pred = np.argmax(audio_cert)

        print(f'The image for this video looks {label_encoding[video_pred]} wi
th a certainty of {np.round(video_cert[video_pred]*100, 2)}%')
        print(f'The audio for this video looks {label_encoding[audio_pred]} wi
th a certainty of {np.round(audio_cert[audio_pred]*100, 2)}%')
```

Data loader de video

```
import numpy as np
from cv2 import cvtColor, COLOR_RGB2BGR
from albumentations import Normalize, Compose, Resize

from data_handling.video_handling import frames_videos, crop_faces


def extract_image(X, frames=10):
    augment = Compose([Resize(275, 225, p=1.0), Normalize(p=1.0)])

    batch_x = frames_videos(X, frames)
    if not batch_x:
        return 0

    batch_x = crop_faces(batch_x, 60)
    batch_x = [cvtColor(frame, COLOR_RGB2BGR) for frame in batch_x]


    return np.stack([augment(image=x)["image"] for x in batch_x], axis=0)
```

Data loader de audio

```
from moviepy.editor import VideoFileClip
import numpy as np
from cv2 import resize, INTER_AREA, normalize, NORM_MINMAX, CV_32F, merge

from data_handling.audio_lin import lfcc
from audiomentations import Compose
from audiomentations import Normalize
```

```
def extract_lfcc(X, lengths=1, n_frames=5):
    augment_pre = Compose([Normalize(p=1.0)])

    lengths = int(lengths*16000)
    videoclip = VideoFileClip(X)
    audioclip = videoclip.audio
    sample_rate = 16000
    sound = audioclip.to_soundarray(fps=sample_rate)[:, 0]

    frame_indices = np.random.randint(0, len(sound)-lengths, n_frames)
    batch_x = [sound[i:i+lengths] for i in frame_indices]

    batch_x = [augment_pre(samples=x, sample_rate=sample_rate) for x in batch_
x]

    pad_batch_x = []
    for wave in batch_x:
        lf_wave = lfcc(y=wave, sr=sample_rate, n_lfcc=20, n_fft=512, hop_lengt
h=160)
        pad_batch_x.append(lf_wave)

    dim = (224, 224)
    pad_batch_x = [resize(img, dim, interpolation = INTER_AREA) for img in pad
_batch_x]
    pad_batch_x = [merge((img, img, img)) for img in pad_batch_x]
    pad_batch_x = [normalize(img, None, alpha=0, beta=1, norm_type=NORM_MINMAX
, dtype=CV_32F) for img in pad_batch_x]
    return np.stack(list(pad_batch_x), axis=0)
```

Anexo IV. Artículo de investigación

Deepfake detection using audio-visual data.

Monica Hazeu Gonzalez

Universidad Internacional de la Rioja, Logroño (España)

22/07/2021

ABSTRACT

The term deepfakes refers to videos, audio or other types of data that have been manipulated using deep learning techniques. They can be a source of misinformation and manipulation, which is why in this project a model has been trained to detect deepfakes in audio-visual data. With this purpose, two separate models have been trained, one for audio and one for video, with data from a variety of datasets, looking above all to improve the generalization capacity, which is scarce in existing models. A relatively good generalization capability has been achieved compared with the state of the art. Additionally, the two models have been combined in a simple-to-use interface so non-expert users can detect deepfake videos.

KEYWORDS

Audio-visual, Deepfakes, Transfer learning

I. INTRODUCTION

THE term deepfake refers to data generated using deep learning techniques. In general, this means audio, images or videos generated using autoencoders or, more recently, Generative Adversarial Networks (GANs). This is a more troubling issue than other image editing techniques like Photoshop in the sense that deepfake techniques are widely available to a non-professional audience. This has led certain people to use deepfakes for malicious purposes, an example being a recent scam to the CEO of an energy company, who was tricked out of 220000€ through a deepfake phone call acting the part of his boss. [1]

However, it is also worth noting that deepfakes don't only have malicious purposes, but can be used for entertainment or other harmless endeavors. All in all, it is not possible to ban deepfakes. This project wishes instead to provide Python users with a tool to discern whether a video or audio has been manipulated or not. To this purpose, two separate models have been trained, one for audio and one for video, using transfer learning. These models have been trained on a variety of data, with a main focus on the generalization capacity of the model. Then, the models have been made available through an easy-to-use interface.

This paper begins with a review of the current state of the art, acknowledging the existing approaches to deepfake detection and using them as a starting point upon which to build. After this review, the objectives of the project are mentioned, followed by a detailed description of the contribution. Then, this contribution is evaluated and discussed, providing a final conclusion.

II. STATE OF THE ART

Deepfake generation

Three types of deepfakes can be distinguished [2] with respect to video manipulation:

1. Head puppetry: source actor is used to make it seem as though the target person is doing the same things as the source actor.
2. Face swapping: the face of the source actor is superposed over the face of the target actor, altering their facial features.
3. Lip syncing: only the lip area is altered.

The first deepfake generation methods for videos were based on autoencoders [3], which is an encoder-decoder pair that reduces the dimensions of the input image and then maps it back onto its original space. Therefore, the objective function is the input image itself. [4] Training the encoder on all the identities and then the decoder on source identity means that the target identity can then be passed onto the common encoder and be decoded with the source decoder, generating a face swap deepfake.

Newer deepfake generation techniques are based on adding to this architecture adversarial loss [5] and perceptual loss [6], in short, using an autoencoder as the generator for a Generative Adversarial Network (GAN) [7], and then adding another network as a discriminator, converting the problem into a minmax optimization, where the generator tries to generate realistic images to fool the discriminator.

With respect to the quality of the deepfakes, they can be grouped into two generations [8]:

- First generation deepfakes are of low quality, with stark differences in skin color between the target and the source, numerous artifacts and little pose variation.
- Second generation deepfakes greatly improve a lot of these artifacts, considering different scenes, postures and distances to the camera.

Audio deepfake generation, on the other hand, is handled as a text-to-sound (TTS) problem. Initial approaches were based on hidden Markov models [9], but more recently, deep learning models have gained attention.

Deep voice 3 uses an encoder-decoder architecture with a converter to transform written text into an internal representation, which the decoder then transforms into a lower dimensional audio representation (Mel scale spectrogram). Then, the converter predicts the final vocoder parameters, all using a fully convolutional architecture. [10]

Multi-task WaveNet, on the other hand, uses multi-task learning, with an architecture based on dilated convolution and quasi-recurrent neural networks to learn at the same time how to generate the audio and the acoustic features. [11]

All in all, audio deepfake generation techniques are very varied, and the most appropriate approach is to have a dataset generated with this variety of techniques.

Deepfake detection

The great majority of investigations consider deepfake detection as a binary classification problem (real or fake). For [2], there exist mainly three approaches to this classification:

- Based on physical or physiological inconsistencies
- Based on signal-level artifacts
- Based on data

Using these techniques, some investigation projects have focused on deepfake image detection:

- Common Fake Feature Network (CFFN) [12]: made up of a DenseNet [13] backbone and a Siamese Network architecture [14]. Pairs of fake and real images are used to train a convolutional network using contrastive learning so it learns the differentiating features. As a second training step, another convolutional layer plus a fully-connected layer are trained as a classifier using binary cross-entropy loss. Although results are good, the authors express concerns about the ability of the model to generalize to different GAN artifacts.
- Convolutional traces [15]: the goal is to extract a fingerprint for each deepfake generation GAN architecture. Specifically, the convolutional traces left by GANs are sought for.

Deepfake image detection techniques can serve as a starting point for deepfake video detection techniques, especially if these techniques are frame-based. Some examples using deep learning are:

- Detecting the difference between the facial area and its surroundings [16]: these inconsistencies are detected using Convolutional Neural Networks (CNN) and, in order to improve the generalization capacity, the network isn't trained using deepfakes, but instead using dynamically generated images to which Gaussian blur has been applied on the facial region. This technique gives good results on first generation datasets.
- MesoNet [17]: Two network architectures are proposed: Meso-4, a small convolutional network and MesoInception-4, which changes the first two convolutional layers of the former for inception modules. The two networks are trained separately with a face swap dataset and a head-puppetry dataset.
- Capsule Forensics [18]: the intention of this network is to avoid the limitations of CNNs when applied to inverse graphics tasks. It is based on a VGG-19 backbone, adding 3 or 10 (depending on the desired network size) capsule network blocks after this backbone. The main advantage of the architecture is the achievement of comparable results to CNNs with less parameters and a more efficient training.

Since deep neural networks can easily overfit, other approaches are based on shallow classifiers:

- Support Vector Machines (SVM) based on head pose difference [19]: using 68 facial landmarks, an SVM model classifies the video as fake/real.
- Photo response non uniformity (PRNU) [20]: this is a noise pattern that is different for each digital camera, as it is generated due to the manufacturing variabilities. It is often considered to be the digital fingerprint for images. This means that any manipulation to the facial region in order to generate a deepfake would alter the noise pattern in this area, allowing the detection of a tampered image.

Some approaches are based on interframe temporal inconsistencies, using mainly Recurrent Neural Networks (RNN) to detect deepfakes:

- CNN combined with LSTM (Long-Short Term Memory network) [21]: the CNN extracts the within-frame artifacts, whereas the LSTM extracts inter-frame inconsistencies.
- Blinking frequency [22]: this technique is based on the inability of deepfake generation methods to generate a blinking pattern similar to a human being, generally blinking much less frequently. The eye region is fed into a Long-Term Recurrent Convolutional Network (LRCN) to find the blinking frequency and detect fake videos.
- Recurrent convolutional networks [23]: this model pipeline is based on a video pre-processing, in which the face is detected and cropped and the frames are aligned. Then, the crops are passed onto a convolutional network, trained to extract important features. The output of the convolutions is used as an input to a bi-directional recurrent network, used to find temporal inconsistencies.

Finally, another approach is based not on visual information, but on audio-visual dissonance:

- Use of affective cues [24]: based on the hypothesis that different video modalities (audio and image) can be mapped onto a space where the distance between them will be small for real videos and large for fake videos. This will also be the case for emotions recognized in the audio and image. The approach uses a Siamese network, into which, during training, a fake and a real video of the same subject are fed. Four feature vectors are generated using convolutions, two for the real video and two for the fake one. Emotions are detected and the Euclidean distance between the vectors is calculated, defining a threshold above which a video is classified as fake during testing.
- Audio and image synchrony [25]: the goal is to detect the dissonance between audio and image in deepfakes, assigning a Modality Dissonance Score (MDS). The audio is represented as Mel-frequency cepstral coefficients (MFCCs), so both the audio and image are analyzed using convolutional architectures, using 2D

convolution for the audio and 3D convolution for the video (to capture time discrepancies).

It is worth noting that these two methods use as data sources Deepfake-TIMIT and Deepfake Detection Challenge (DFDC) datasets, because they include audio. However, none of the two datasets are correctly labelled with regards to audio, since they have fake videos where only the image is manipulated and the audio is pristine. This means that the networks are being trained with incorrect information.

With regards to specific audio deepfake detection, some of the previous methodologies are:

- ResNets with data augmentation [26]: the ASVspoof 2019 dataset is used in combination with a variety of techniques like frequency masking and noise addition, training a ResNet using Large Margin Cosine Loss (LMCL) in order to increase variance between classes and decrease intra-class variance.
- Artificial characteristics of synthetic audio [27]: a statistical study based on the differences between synthetic and pristine audio, like the number of pauses and the transition to said pauses (sharper in synthetic audio).

Although deepfakes have gained importance only recently, it can be seen that there are already numerous projects focused on detecting them. These methods are varied and have brought great advancements to this field. Also, the first commercial deepfake detection tools are starting to appear [28][29]. However, there are two things which should be highlighted. The first is that the current generalization capability of deepfake detection methods on data they have not been trained on is extremely poor, with a drop of 25-35% in AUC [30]. This was also seen in the Kaggle Deepfake Detection Challenge (DFDC) where the best model achieved an Average Precision (AP) of 82.56% on the public dataset, but the best model on the private (hidden) dataset obtained an AP of 65.18% [31].

The second point to highlight is the fact that deepfake generation methods improve daily, generating more realistic deepfakes with less training time. This means that deepfake detection techniques must also improve daily in order to keep up.

After analyzing the state of the art, it is considered that the most promising line of investigation is using audio and image for deepfake detection. However, DF-TIMIT and DFDC will not be used to train the audio model for the reasons mentioned previously. Instead, audio and video will be trained separately with properly labelled data.

Additionally, in order to increase interest in this area and incentivize investigation projects, a simple Python tool will be implemented in order to allow users with some Python knowledge to use the developed models to detect deepfakes in new data. This will also help raise awareness about deepfakes and make these users less susceptible to being fooled by them.

III. Objectives and Methodology

The general objective of this project is to develop a deepfake detection model based on audio and video, evaluating said model with a hidden dataset and putting it at the disposal of users with minimal knowledge of Python. More specifically, the specific objectives are:

- Analyze existing deepfake detection techniques
- Determine amongst these techniques the most appropriate one
- Develop, based on this, a model capable of deepfake detection
- Evaluate said model on a hidden dataset
- Enable the access to the model for non-technical users

In order to accomplish these goals, the steps to follow are:

1. Evaluate the state of the art
2. Define the architecture
3. Implement and train the model
4. Demonstrate the generalization capability through experimentation
5. Enable the access to the model

IV. Contribuiton

Data preparation

Video

As reviewed in the chapter regarding the state of the art, there are two possible approaches to deepfake detection in videos: based on individual frames or based on temporal features. In some cases, this second approach has had better results, but at the expense of using recurrent neural networks or 3D convolutional networks, which is very costly computationally speaking. Since good results have been obtained using individual frames, this approach will be chosen for this project, which also wishes to give importance to the inference time that the user will have to endure.

Deepfake generation algorithms mainly alter the facial region, leaving the rest of the image unaltered. For this reason, after extracting the individual frames, the facial region will be cropped from said frames, and this will be the image fed into the model.

In order to crop the face from the image, it is first necessary to detect the face. For this purpose, an appropriate face detector must be chosen from those available. The main requirements for this detector, in order of importance, are:

1. Speed and lightweight: since the detector is not the main part of the model, ideally most of the resources will not be devoted to this detection.
2. Precision: the facial detection must be correct to ensure the model is fed with correct images.
3. Recall: it is important to detect faces in the majority of the frames, particularly in the testing phase, although this is less important than detecting correct faces.

In order to satisfy these requirements, different state of the art detectors are evaluated. During this evaluation, it is found that most of the face detection algorithms are highly accurate, and so the main criteria for choosing a detector ends up being speed. FaceBoxes (0.3 s/frame) [32] and face_recognition (0.04 s/frame) end up as the two top options. However, the former is chosen over the latter because face_recognition is a library based on dlib, which is not officially supported on Windows, and therefore adds unnecessary complexity to the model.

After the faces have been cropped from the frames, these images will be augmented before being fed into the model. The augmentations used will be:

- Spatial transformations: translation, scaling, rotation, mirroring, resizing
- Color distortions: changes in lighting, changes in

contrast, FancyPCA [33], gray scale transformation, normalization

- Information removal: image compression, Gaussian Noise, Grid Dropout [34]

As a source of data, a number of datasets have been chosen in order to improve the generalization capacity of the model. In particular, the datasets used for training were:

- DeepfakeTIMIT [35][36]: made up of 640 manipulated videos and 430 pristine videos of 43 subjects in total. There is little variability in pose and lighting.
- FaceForensics++ [37]: made up of 1000 original videos from youtube, out of which 4000 manipulated videos have been generated using 4 different deepfake generation techniques.
- Celeb-DF-v2 [30]: 890 original videos from youtube, 590 of them picturing famous people. 5639 manipulated videos are generated from the videos of famous people using face swapping techniques. Only a subset of these videos will be used.
- DeepFake Detection Challenge [38]: this dataset is made up of 104500 manipulated videos and 23654 pristine videos, with great variability in poses, lighting and video quality.

The data extracted from these datasets will be divided into a training and a validation set. In order to test the generalization capability of the model, the test set is composed of images extracted from videos that don't belong to these datasets. A formal evaluation of the model is done using videos from the dataset DeeperForensics 1.0 [39].

With regards to external libraries, OpenCV [40] is used to handle the images and Albumentations [41] is used to augment them. An example of the augmented images can be seen in Image 1.

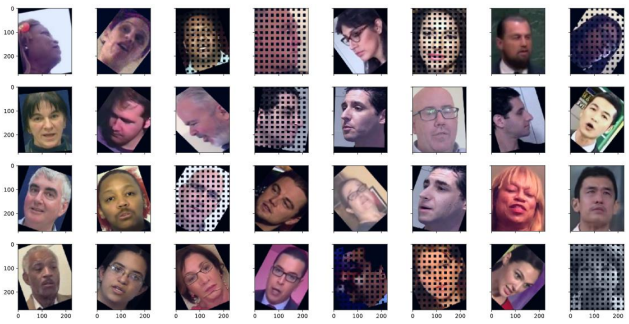


Image 1. Example of augmentations done on images. Source: self-made

Audio

In order to train the audio model, properly labelled data must be used. This is why the ASVspoof2019 dataset has been chosen, in particular de Logical Access (LA) part, which is made up of real and synthetic audios, the latter generated with state-of-the-art text-to-speech and voice conversion techniques [42]. In this dataset, different techniques are used to generate the audios of the training and validation sets with respect to the test set.

In order to feed the audio data to the model, an appropriate representation feature set must be chosen. In [43], a number of representation features for audio are tested with an earlier version of the ASVspoof. They find that the features which allow for the best generalization capability are Δ Δ^2 LFCCs. Linear Frequency Cepstral Coefficients (LFCC) are a transformation of the frequency spectrum back into the time domain, to which a linear filter bank is applied in order to discretize the representation into frequency bins. The Δ coefficients are the dynamic variant, where the variation between time steps is calculated. Seeing the results from this paper, dynamic LFCCs will be used as audio representations for this model. In particular, a 32 ms window size is chosen, with a 10 ms window overlap. Additionally, 20 cepstrum coefficients will be used, giving therefore 40 coefficients per frame (originating from the combination of Δ and Δ^2 coefficients).

In order to increase the generalization capacity of the model, and inspired by [26], the following augmentations will be used:

- Reverberation using Room Impulse Responses (from [44][45])
- Frequency masking
- Time shifting
- Pitch modifications
- Speed modifications

Librosa [46] will be used as an external library to process the audio. Although it does not provide the computation of the LFCCs, it does provide the MFCCs, so the implementation of the LFCC will simply need the Mel-scale filter to be changed for a triangular filter bank. An example of this audio representation can be seen on Image 2.

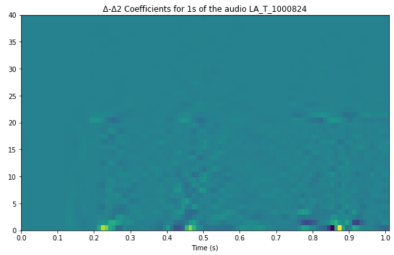


Image 2. Example of $\Delta\Delta^2$ LFCC coefficients. Source: self-made

Model architecture

For both models, transfer learning techniques will be used [47]. The idea behind this is to use a model pretrained on a very large dataset and which has learnt representations from that dataset. These can then be used to learn a new problem much quicker.

Video

In [48], a family of models named EfficientNet is presented. This article presents a novel technique for convolutional network scaling. In particular, they use an architecture based on Mobile Inverted Bottleneck (MBConv) blocks to attain above state-of-the-art results on ImageNet.

In March of 2021, the same authors have presented an improved version of this model, EfficientNetV2 [49], which uses Fused-MBConv blocks and progressive learning to improve previous performance on ImageNet with less parameters. The comparison can be seen in Image 3.

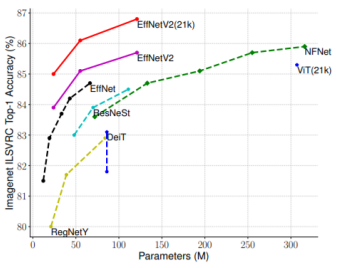


Image 3. Comparison of different model accuracies on ImageNet versus number of parameters. Source: [49]

Seeing the clear improvement in efficiency and accuracy, this model has been chosen to do transfer learning on the video dataset, after which fine-tunning on the model will be done. In Table 1, a summary of the model architecture can be seen.

Table 1. Summary of the EfficientNetV2 model architecture. Source: [49]

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Audio

As mentioned above, dynamic LFCCs will be used as features for the audio model. Since these are essentially a matrix, it is quite common to treat them as images (heatmaps) and analyze them using 2D convolutional neural networks [50][51][52].

[51] and [52] prove using convolutional neural networks that transfer learning allows the achievement of models with better results than those with randomly initialized weights. However, it must be noted that this comparison is done against a randomly initialized network, and there is no comparison between networks pretrained on ImageNet or on an audio dataset. [50] carry out this comparison and the results they find are considered to be non-conclusive. They do find that the model pretrained on audio gets slightly better results than models pretrained on ImageNet, but there is only one audio model in the comparison. MobileNetV2, trained on ImageNet, achieves only slightly worse results with an order of magnitude less parameters.

Considering these results, the initial audio model will be developed using MobileNetV2 [53], since it is a much lighter and easier to implement network. Only if the results are unsatisfactory will the other network be tested. The model architecture can be seen in Image 4.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Image 4. MobileNetV2 model architecture. Source: [53]

Loss function

In order to train the models, a loss function must be defined. The deepfake detection problem will be treated in this project as a binary classification problem. Usually, for these types of problems, binary cross-entropy is used (see equation 1). However, recent studies like [54] have shown that the discriminative capacity of this function is limited. Other alternatives have been proposed whose objective is to maximize the interclass variance and minimize the intraclass variance.

$$Loss_{softmax} = \frac{1}{N} \cdot \sum_{i=1}^{N} \ln \frac{e^{f_{y_i}}}{\sum_{j=1}^{c} e^{f_j}} \quad (1)$$

The majority of the alternatives proposed are modifications of the softmax loss. Most recently, [55] introduced Large Margin Cosine Loss (LMCL), which proposed to measure the angular distance between classes instead of the Euclidean distance, normalizing the weight and feature vectors to make the prediction independent from their magnitude, and introducing an interclass margin to increase the discriminative power of the function. This cost function can be seen in equation 2.

$$Loss_{LMCL} = \frac{1}{N} \cdot \sum_{i=1}^{N} \ln \frac{e^{s \cdot (\cos(\theta_{y_i,i}) - m)}}{e^{s \cdot (\cos(\theta_{y_i,i}) - m)} + \sum_{j \neq y_i} e^{s \cdot \cos(\theta_{y_i,i})}} \quad (2)$$

Given that this cost function is used in the original paper for a different type of task (face recognition), both LMCL and softmax will be compared for the audio and video model, choosing the most appropriate one for the task.

Metrics

Among the available metrics to judge the performance of the model and guide the training, the following metrics will be chosen:

- Accuracy (used in [18][23]): defined by equation 3. It returns a value between 0 and 1.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (3)$$

- Area Under the Curve (AUC) (used in [8]): corresponds to the area under the true positive rate versus false positive rate curve.

During training, mainly the loss function will be monitored to see that the training is progressing appropriately, but the accuracy will be used to evaluate the performance on the validation set. Also, for unbalanced classes, AUC will be used as a metric, since accuracy is not a very appropriate indicator in these cases.

V. Evaluation and Results

Training

Video

The dataset generated following the steps detailed above is made up of more than 200000 images. This dataset is then processed to remove image crops that do not correspond with faces.

Additionally, the classes are imbalanced with a ratio of 1:3, with the FAKE class being the majority one. This was expected, as the original datasets are also quite imbalanced. This imbalance is problematic for the backpropagation algorithm, since it results in a smaller gradient for the minority class, which makes the majority class's gradient decrease very rapidly and the minority class converge very slowly [56]. To solve this problem, in [57], a training procedure was proposed where the initial training was done with Random Under Sampling (RUS) of the majority class to a threshold, and fine-tunning was done after this with all the examples available. This was proven experimentally to give the best results, even with an imbalance of 650 to 1. In this project, this was the procedure used, using RUS for the transfer learning training and then using all the examples for the fine-tunning training.

During the training, different hyperparameters were trialed with a small subset of the data in order to speed up the iterations. In the end, the best training and the best results were obtained using the softmax function, with the EfficientNetV2-small architecture followed by a dropout of 0.3. After this, there is a hidden layer of 64 neurons with *GlorotUniform* weight initialization, L2 weight normalization and RELU activation. The final layer is made up of two neurons with softmax activation. The specific cost function implementation is sparse categorical cross entropy, as the labels are provided with label encoding. This model is trained with an early stopping callback, obtaining a best accuracy of 65.03% on the validation set.

This best model is then fine-tuned by turning the EfficientNet part of the model into *trainable*. This part of the training takes significantly longer (32 hours per epoch versus 3 for transfer learning). The model converges at a validation accuracy of 92.35%. For this final model, since it is now using all of the data and is therefore imbalanced, the AUC is calculated, obtaining an AUC score of 88%. The confusion matrix can be seen in Table 2.

Table 2. Confusion matrix for the video model on the validation set. Source: self-made

	Prediction	
Ground truth	FAKE	REAL
FAKE	28788	663
REAL	2276	8336

This model is then evaluated on the testing set to judge its generalization capability. On this dataset, the ratio of real to fake images is 1.5 real images per fake image, which is closer to the current reality, where the majority of the videos are real, but not quite the same as the reality, since it is still necessary to judge the capacity of the model to detect fake videos. In this case, an accuracy of 77.6% is obtained, with an AUC of 75%. The confusion matrix can be seen in Table 3.

Table 3. Confusion matrix for the video model on the testing set. Source: self-made

	Prediction	
Ground truth	FAKE	REAL
FAKE	17945	10977
REAL	6098	41446

Audio

The audio data is also imbalanced in favor of the FAKE class, even more so than the video one, with a ratio of 1:10. This imbalance is too strong to use all the examples even if it is just for fine-tunning. For this reason, RUS will be used for the first part of the training, but RUS will be combined with Random Over Sampling (ROS) of the minority class for the second part of the training, reducing the number of majority class examples to 4 times the number of minority class examples, and doing ROS of the minority class until the ratio is 1:2.

After trialing a number of hyperparameters, the best model obtained is one with the softmax loss function, with a dropout of 0.3 after the MobileNetV2 model, followed by a 512-neuron hidden layer with RELU and L2 weight normalization. The final layer is also a 2-neuron layer with softmax activation. However, the best results, even for this model, are below 70% accuracy, and the fine-tunning process, where MobileNet weights are unfrozen, only results in overfitting and a reduction in accuracy. It is therefore seen that perhaps dynamic coefficients are not sufficiently expressive, so this same model is trained with static LFCC coefficients. The results obtained are much better, with 85.32% accuracy on the validation set.

This model is then tunned without unfreezing MobileNet weights, but using the RUS plus ROS mentioned previously, with an initial learning rate one order of magnitude lower than the first training[10]. The final results are an accuracy of 88.7% and an AUC score of 86%. The confusion matrix can be seen in Table 4.

[10] Initial learning rate only, as RMSprop is used as the optimizer.

Table 4. Confusion matrix for the audio model on the validation set. Source: self-made

	Prediction	
Ground truth	FAKE	REAL
FAKE	8570	1622
REAL	331	2217

This model is then used on the testing set, where an AUC score of 77% is obtained. The confusion matrix can be seen in Table 5.

Table 5. Confusion matrix for the audio model on the testing set. Source: self-made

	Prediction	
Ground truth	FAKE	REAL
FAKE	53386	10496
REAL	2180	5175

VI. Discussion

In Image 5, some examples of the images that the video model misclassified are shown (the true label is shown below the image). It can be seen that these images are quite hard even for a human, where none of them have obvious artifacts.

Image 5. Examples of misclassifications for the video model. Source: self-made

If the confusion matrices are reviewed, it can be seen that, for the validation set, 7% of the videos classified as fake were in fact real, with the same percentage of videos classified as real being fake. 2% of fake images were misclassified, versus 21% of real images being misclassified. On the other hand, for the testing set, 37% of fake videos were misclassified as real, whereas 13% of real videos were misclassified as fake. It is therefore clear that the results on the testing set are worse than those on the validation set. However, this is worth comparing with the results obtained in [58], where the generalization capability of a number of deepfake detection models was compared. The best results obtained where a 96.8% accuracy on the validation set versus a 68.06% accuracy on the test set. Although it is not possible to quantitatively compare this project to [58], since both projects are evaluated on different datasets ([58] is evaluated on FaceForensics++ and some additional custom data), it is possible to qualitatively judge that the generalization capability of this project's model seems better.

With respect to the audio model, 16% of fake audios were misclassified on the validation set, whereas 13% of real audios were misclassified. In contrast to this, 16% of fake audios on the testing set were misclassified, versus 29% of misclassified real audios. It can be clearly seen that, although the AUC score for the

testing set is lower than for the validation set, this is mainly due to the real class. A possible cause for this could be the lack of real examples, as they are quite scarce in the dataset used. On the other hand, a similar percentage of fake videos was correctly classified, which is good sign, as the techniques to generate the deepfakes of the testing set are completely different to those on the training and validation sets.

With these results, the code used for training and validation is adapted into a single tool, which can be easily run by a non-expert user through a Jupyter Notebook.

VII. Conclusion

This project has developed an audio-visual deepfake detection tool. This has been accomplished by training a model for audio and a model for video, using a variety of existing datasets. Transfer learning was used for both models and data augmentation was used on the training data in order to increase the generalization capability of the models. The models were later evaluated on a hidden test set to judge said generalization capability.

The results obtained on the video model were, as could be expected, better on the validation set than on the testing set. However, it seemed apparent that the model had a better generalization capability than the state of the art.

On the audio model, the results were also good, especially for the FAKE audio class. The generalization on the REAL class was not as good, which seemed to be a clear indicator that there were not enough examples belonging to this class.

In general, both the audio and video datasets were heavily skewed towards the FAKE class, which is not the case in real life. A possible future line of work could increase the number of REAL examples to see if this provides better results. In addition to this, it would be a good idea to train on a more powerful device, which would allow faster and more iterations, and might arrive at a better combination of hyperparameters.

Another possible line of work would be to train the models with data containing adversarial noise. This type of noise can easily confuse a neural network, so including it during the training could help improve the model robustness.

With respect to the deepfake detection tool, a possible improvement to it would be to make it accessible to non-Python users. It would also be a good idea to implement a tracker into the face detection part of the process, which might make this detection more stable and could allow individual predictions for each of the detected entities.

To summarize, this project opens up a lot of future possibilities to improve coexistence with deepfakes, which have no doubt come to stay.

References

[1] Damiani, J. (2019). A Voice Deepfake Was Used To Scam A CEO Out Of $243,000. https://www.forbes.com/sites/jessedamiani/2019/09/03/a-voice-deepfake-was-used-to-scam-a-ceo-out-of-243000/

[2] Lyu, S. (2020). Deepfake detection: Current challenges and next steps. 2020 IEEE International Conference on Multimedia & Expo Workshops (ICMEW), 1–6.

[3] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). Learning internal representations by error propagation.

[4] Nguyen, T. T., Nguyen, C. M., Nguyen, D. T., Nguyen, D. T., & Nahavandi, S. (2019). Deep learning for deepfakes creation and detection. ArXiv Preprint ArXiv:1909.11573, 1.

[5] Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. Proceedings of the IEEE International Conference on Computer Vision, 2223–2232.

[6] Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). Deep face recognition.

[7] Faceswap-GAN. (2019). https://github.com/shaoanlu/faceswap-GAN

[8] Tolosana, R., Romero-Tapiador, S., Fierrez, J., & Vera-Rodriguez, R. (2020). Deepfakes evolution: Analysis of facial regions and fake detection performance. ArXiv Preprint ArXiv:2004.07532.

[9] Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. Ieee Assp Magazine, 3(1), 4–16.

[10] Ping, W., Peng, K., Gibiansky, A., Arik, S. O., Kannan, A., Narang, S., Raiman, J., & Miller, J. (2017). Deep voice 3: Scaling text-to-speech with convolutional sequence learning. ArXiv Preprint ArXiv:1710.07654.

[11] Gu, Y., & Kang, Y. (2018). Multi-task WaveNet: A multi-task generative model for statistical parametric speech synthesis without fundamental frequency conditions. ArXiv Preprint ArXiv:1806.08619.

[12] Hsu, C.-C., Zhuang, Y.-X., & Lee, C.-Y. (2020). Deep fake image detection based on pairwise learning. Applied Sciences, 10(1), 370.

[13] Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). Densenet: Implementing efficient convnet descriptor pyramids. ArXiv Preprint ArXiv:1404.1869.

[14] Sem-Jacobsen, F. O., Skeie, T., Lysne, O., Tørudbakken, O., Rongved, E., & Johnsen, B. (2005). Siamese-twin: A dynamically fault-tolerant fat-tree. 19th IEEE International Parallel and Distributed Processing Symposium, 10--pp.

[15] Guarnera, L., Giudice, O., & Battiato, S. (2020). Fighting Deepfake by Exposing the Convolutional Traces on Images. IEEE Access, 8, 165085–165098.

[16] Li, Y., & Lyu, S. (2018). Exposing deepfake videos by detecting face warping artifacts. ArXiv Preprint ArXiv:1811.00656.

[17] Afchar, D., Nozick, V., Yamagishi, J., & Echizen, I. (2018). Mesonet: a compact facial video forgery detection network. 2018 IEEE International Workshop on Information Forensics and Security (WIFS), 1–7.

[18] Nguyen, H. H., Yamagishi, J., & Echizen, I. (2019a). Use of a capsule network to detect fake images and videos. ArXiv Preprint ArXiv:1910.12467.

[19] Yang, X., Li, Y., & Lyu, S. (2019). Exposing deep fakes using inconsistent head poses. ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 8261–8265.

[20] Koopman, M., Rodriguez, A. M., & Geradts, Z. (2018). Detection of deepfake video manipulation. The 20th Irish Machine Vision and Image Processing Conference (IMVIP), 133-136.

[21] Güera, D., & Delp, E. J. (2018). Deepfake video detection using recurrent neural networks. 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), 1–6.

[22] Li, Y., Chang, M.-C., & Lyu, S. (2018). In ictu oculi: Exposing ai created fake videos by detecting eye blinking. 2018 IEEE International Workshop on Information Forensics and Security (WIFS), 1–7.

[23] Sabir, E., Cheng, J., Jaiswal, A., AbdAlmageed, W., Masi, I., & Natarajan, P. (2019). Recurrent convolutional strategies for face manipulation detection in videos. Interfaces (GUI), 3(1).

[24] Mittal, T., Bhattacharya, U., Chandra, R., Bera, A., & Manocha, D. (2020). Emotions Don't Lie: An Audio-Visual Deepfake Detection Method using Affective Cues. Proceedings of the 28th ACM International Conference on Multimedia, 2823–2832.

[25] Chugh, K., Gupta, P., Dhall, A., & Subramanian, R. (2020). Not made for each other-Audio-Visual Dissonance-based Deepfake Detection and Localization. Proceedings of the 28th ACM International Conference on Multimedia, 439–447.

[26] Chen, T., Kumar, A., Nagarsheth, P., Sivaraman, G., & Khoury, E. (2020). Generalization of audio deepfake detection. Proceedings of the Odyssey Speaker and Language Recognition Workshop, Tokyo, Japan, 1–5.

[27] Gao, Y., Lian, J., Raj, B., & Singh, R. (2021). Detection and Evaluation of human and machine generated speech in spoofing attacks on automatic speaker verification systems. 2021 IEEE

Spoken Language Technology Workshop (SLT), 544–551.
[28] Sensity. (n.d.). Sensity. Retrieved May 3, 2021, from https://sensity.ai/
[29] Kelion, L. (2020). Deepfake detection tool unveiled by Microsoft. BBC. https://www.bbc.com/news/technology-53984114
[30] Li, Y., Yang, X., Sun, P., Qi, H., & Lyu, S. (2020). Celeb-df: A large-scale challenging dataset for deepfake forensics. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 3207–3216.
[31] Facebook. (2020). Deepfake Detection Challenge Dataset. Facebook AI. https://ai.facebook.com/datasets/dfdc/
[32] Zhang, S., Zhu, X., Lei, Z., Shi, H., Wang, X., & Li, S. Z. (2017). Faceboxes: A CPU real-time face detector with high accuracy. 2017 IEEE International Joint Conference on Biometrics (IJCB), 1–9.
[33] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems, 25, 1097–1105.
[34] Chen, P., Liu, S., Zhao, H., & Jia, J. (2020). Gridmask data augmentation. ArXiv Preprint ArXiv:2001.04086.
[35] Korshunov, P., & Marcel, S. (2018). Deepfakes: a new threat to face recognition? assessment and detection. ArXiv Preprint ArXiv:1812.08685.
[36] Sanderson, C., & Lovell, B. C. (2009). Multi-region probabilistic histograms for robust and scalable identity inference. International Conference on Biometrics, 199–208.
[37] Rössler, A., Cozzolino, D., Verdoliva, L., Riess, C., Thies, J., & Nießner, M. (2019). Face{F}orensics++: Learning to Detect Manipulated Facial Images. International Conference on Computer Vision (ICCV).
[38] Dolhansky, B., Howes, R., Pflaum, B., Baram, N., & Ferrer, C. C. (2020). The deepfake detection challenge (dfdc) dataset. ArXiv Preprint ArXiv:2006.07397, 1(2).
[39] Jiang, L., Li, R., Wu, W., Qian, C., & Loy, C. C. (2020). Deeperforensics-1.0: A large-scale dataset for real-world face forgery detection. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2889–2898.
[40] OpenCV. (2021). OpenCV. https://github.com/opencv/opencv/tree/4.5.2
[41] Buslaev, A., Iglovikov, V. I., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). Albumentations: Fast and Flexible Image Augmentations. Information, 11(2). https://doi.org/10.3390/info11020125
[42] Yamagishi, J., Todisco, M., Sahidullah, M., Delgado, H., Wang, X., Evans, N., Kinnunen, T., Lee, K. A., Vestman, V., & Nautsch, A. (2019). Asvspoof 2019: The 3rd automatic speaker verification spoofing and countermeasures challenge database.
[43] Sahidullah, M., Kinnunen, T., & Hanilçi, C. (2015). A comparison of features for synthetic speech detection.
[44] Ko, T., Peddinti, V., Povey, D., Seltzer, M. L., & Khudanpur, S. (2017). A study on data augmentation of reverberant speech for robust speech recognition. 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 5220–5224.
[45] OpenSLR. (2017). Room Impulse Response and Noise Database. http://www.openslr.org/28/
[46] McFee, B., Raffel, C., Liang, D., Ellis, D. P. W., McVicar, M., Battenberg, E., & Nieto, O. (2015). librosa: Audio and music signal analysis in python. Proceedings of the 14th Python in Science Conference, 8, 18–25.
[47] Bozinovski, S., & Fulgosi, A. (1976). The influence of pattern similarity and transfer learning upon training of a base perceptron b2. Proceedings of Symposium Informatica, 3–121.
[48] Tan, M., & Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. International Conference on Machine Learning, 6105–6114.
[49] Tan, M., & Le, Q. V. (2021). Efficientnetv2: Smaller models and faster training. ArXiv Preprint ArXiv:2104.00298.
[50] Koike, T., Qian, K., Kong, Q., Plumbley, M. D., Schuller, B. W., & Yamamoto, Y. (2020). Audio for audio is better? An investigation on transfer learning models for heart sound classification. 2020 42nd Annual International Conference of the IEEE Engineering in Medicine \& Biology Society (EMBC), 74–77.
[51] Palanisamy, K., Singhania, D., & Yao, A. (2020). Rethinking cnn models for audio classification. ArXiv Preprint ArXiv:2007.11154.
[52] Zhou, H., Bai, X., & Du, J. (2018). An investigation of transfer learning mechanism for acoustic scene classification. 2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP), 404–408.
[53] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 4510–4520.
[54] Wen, Y., Zhang, K., Li, Z., & Qiao, Y. (2016). A discriminative feature learning approach for deep face recognition. European Conference on Computer Vision, 499–515.
[55] Wang, H., Wang, Y., Zhou, Z., Ji, X., Gong, D., Zhou, J., Li, Z., & Liu, W. (2018). Cosface: Large margin cosine loss for deep face recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 5265–5274.
[56] Anand, R., Mehrotra, K. G., Mohan, C. K., & Ranka, S. (1993). An improved algorithm for neural network classification of imbalanced training sets. IEEE Transactions on Neural Networks, 4(6), 962–969.
[57] Lee, H., Park, M., & Kim, J. (2016). Plankton classification on imbalanced large scale database via convolutional neural networks with transfer learning. 2016 IEEE International Conference on Image Processing (ICIP), 3713–3717.
[58] Du, M., Pentyala, S., Li, Y., & Hu, X. (2020). Towards generalizable deepfake detection with locality-aware autoencoder. Proceedings of the 29th ACM International Conference on Information \& Knowledge Management, 325–334.