Evaluación de redes neuronales convolucionales para la clasificación de imágenes histológicas de cáncer colorrectal mediante transferencia de aprendizaje.

Nombre Estudiante
John Marturet Rodrigo
Máster en Bioinformática y Bioestadística
Área de *Machine Learning*

Nombre Director, Consultor
Edwin Santiago Alférez Baquero

Enero de 2018

Cuanto más entreno, más suerte tengo.

Gary Player.

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Evaluación de redes neuronales convolucionales para la clasificación de imágenes histológicas de cáncer* colorrectal *mediante transferencia de aprendizaje*
Nombre del autor:	*John Marturet Rodrigo*
Nombre del consultor/a:	*Edwin Santiago Alférez Baquero*
Nombre del PRA:	
Fecha de entrega (mm/aaaa):	*01/2017*
Titulación:	*Máster en Bioinformática y Bioestadística*
Área del Trabajo Final:	*Machine learning*
Idioma del trabajo:	*Español*
Palabras clave	*Transferencia de aprendizaje, Keras, redes convolucionales.*

Resumen del Trabajo (máximo 250 palabras):

Una de las dificultades que se presentan en la clasificación de imágenes histológicas es determinar descriptores de texturas hechos a mano sobre los que poder modelizar algoritmos de clasificación automática. En el presente trabajo se pretende abordar el problema desde otro enfoque, implementando distintas arquitecturas de redes neuronales convolucionales sobre un conjunto de imágenes de cáncer de colon y recto.

Dos características con respecto al entrenamiento de redes convolucionales se han de tener en cuenta, por una parte, se debe tener un gran número de imágenes y por otra, el tiempo necesario para completarlo, debido a estos motivos se recurrirá al uso de arquitecturas pre-entrenadas como VGG16, VGG19 e InceptionV3.

Una vez implementadas se efectuarán dos tipos de transferencia de aprendizaje, por una parte, se extraerán descriptores profundos para su clasificación mediante algoritmos clásicos de clasificación, y por otra, se ajustarán los pesos en algunas capas de las redes con objetivo de volver a evaluar los resultados y establecer una comparativa entre ambos métodos.

Abstract (in English, 250 words or less):

One of the difficulties arising in histological slides classification is to set up the hand-made texture descriptors that could be used to create automatic classification models. In this study we try a different approach, implementing a few convolutional neural network architectures and feeding them a colorectal set of histological slides.

Two main characteristics regarding the training of convolutional networks must be taken into account, the need for a large set of images and the amount of time necessary to complete it. Pre-trained architectures as VGG16, VGG19 and InceptionV3 will be used in order to overcome those problems.

Once those architectures are implemented, two transfer learning techniques will be evaluated. The first, deep features will be extracted to be classified by classical algorithms, secondly, in which the weights in some layers will be adjusted so the convolutional network classifies the histological slides. Finally, a comparison between both methods will be shown.

Índice

Lista de figuras

Lista de tablas

Lista de códigos

Lista de archivos

1. Introducción

1.1 Contexto y justificación del trabajo

En el interior del colon y del recto humano se hallan una serie de glándulas que segregan una capa de tejido, denominada mucosa. En ella se desarrollan los pólipos que cuando son adenomatosos poseen la capacidad de evolucionar en tumor maligno, es decir en cáncer de colon y recto, solo un porcentaje aproximado del 10% de los mismos evolucionan a tal estado [1]. Las tasas de supervivencia dependen de muchos factores, principalmente del estadio de la enfermedad, pero cuanto antes se detecta y se incide en su tratamiento, mejores resultados en cuanto a su supervivencia se obtienen.

En este sentido, el cáncer de colon y recto puede ser diagnosticado precozmente, antes incluso de que el paciente padezca los síntomas habituales mediante el test de sangre oculta en heces (TSOH). Esta prueba no invasiva, busca restos de sangre en las heces del paciente y un positivo en él, puede ser un indicador precoz del cáncer. En esta situación el facultativo puede proceder extrayendo una biopsia para su análisis y obtener una imagen histológica del tejido [2].

Por otra parte, el análisis de texturas en imágenes histológicas es una herramienta a la que los investigadores recurren para evaluar patologías de este tipo [3]. Habitualmente en estos análisis de texturas intervienen descriptores hechos a mano ("*hand-crafted features"*) que requieren de cierta experiencia y que, frecuentemente, no pueden ser reutilizados en otro tipo de estudios [4]. Por ejemplo, en Kather JN et al. [3], estos investigadores liberaron un conjunto de imágenes histológicas de cáncer de colon y recto. En este artículo se presentan clasificaciones de imágenes mediante patrones locales binarios (LBP), filtros Gabor y otros descriptores de textura. Realizaron dos tipos de clasificación, una logística tumor-estroma con un resultado de 98.6% y una **multi categoría**, que es con la que se compara este trabajo, **con un 87.4% de acierto**.

En el presente trabajo, en lugar de clasificar las imágenes mediante descriptores hechos a mano, se pretende abordar el problema desde otro enfoque, evaluando el uso de redes neuronales convolucionales (*convnets*) para la clasificación automática de las imágenes histológicas de cáncer de colon y recto. Dos características se pueden destacar respecto del entrenamiento de las redes neuronales. La primera, es la necesidad de tener un gran número de imágenes y la segunda, es el tiempo de cómputo. Por ello se recurrirá a implementar arquitecturas pre-entrenadas con un conjunto de imágenes llamado *imagenet* [5] y cuyos pesos (*weights*) se encuentran disponibles en [6].

Tras la implementación de las arquitecturas de las convnets, con los respectivos pesos asociados a cada capa de la arquitectura, se realizan dos procedimientos (*pipeline*). En el primero se extraen descriptores profundos (*deep features*) calculados por la convnet,

para luego clasificarlos mediante técnicas clásicas como máquinas de soporte vectorial (SVM) entre otras. Un factor a tener en cuenta en este pipeline es que debido al alto número de deep features que se obtienen, se evalúa el uso de técnicas de reducción de dimensionalidad [7]. En el segundo pipeline, es la propia convnet quien clasifica de forma automática las imágenes histológicas de cáncer de colon y recto, mediante el entrenamiento de una o varias capas de la arquitectura implementada obteniendo así una mejor adaptación de la convnet al problema.

El citado conjunto de imágenes liberado por el equipo de investigadores en [3] ha sido utilizado en varios estudios como [7-9]. Por lo que se comparan los resultados obtenidos con otros similares de la literatura encontrada. En definitiva, como resultado final, se obtiene una idea de cómo de útil puede resultar el uso de redes neuronales convolucionales en este tipo de problemas.

1.2 Objetivos generales y específicos

A continuación, se exponen el objetivo general, los específicos y las tareas asociadas.

Objetivo general

1. Evaluar las redes neuronales convolucionales para la clasificación de imágenes histológicas de cáncer colorrectal mediante la transferencia de aprendizaje.

Objetivos específicos

1. Elaborar un estado del arte sobre estudios relacionados la clasificación automática de imágenes histológicas.
2. Explorar las imágenes histológicas de cáncer de colon y recto.
3. Extraer descriptores profundos de varias arquitecturas de redes neuronales convolucionales para la clasificación de imágenes histológicas de cáncer de colon.
4. Clasificar mediante algoritmos clásicos el conjunto de imágenes histológicas de cáncer de colon usando los descriptores profundos extraídos.
5. Ajustar los pesos de algunas capas de las redes neuronales pre-entrenadas para la clasificación de imágenes histológicas de cáncer de colon.
6. Comparar y discutir los resultados de clasificación obtenidos en este trabajo con otros estudios de la literatura basados en el mismo conjunto de imágenes histológicas de cáncer de colon o con trabajos similares.

1.3 Enfoque y método seguido

Para realizar el trabajo se han tenido en cuenta los conocimientos previos del autor.

Áreas con experiencia previa	Áreas sin experiencia previa
Algoritmos de clasificación	Redes neuronales convolucionales
Conocimientos básicos de redes neuronales	Pre-procesado de imágenes
Programación básica en lenguaje Python	

Metodología

Tras estudiar las características del conjunto de imágenes histológicas, se implementan distintas arquitecturas mediante Keras, como VGG16, VGG19 e InceptionV3. La plataforma de desarrollo elegida es Anaconda, instalada en un sistema operativo Linux.

En este punto se crean dos pipelines:

- Pipeline 1: Se extraen las deep features de la convnet, se reduce su dimensionalidad, para finalmente clasificarlos mediante algoritmos clásicos, como máquinas de soporte vectorial.

- Pipeline 2: Se crean muestras de entrenamiento, validación y test. Con las dos primeras de ellas se entrenan una o varias capas de las convnet y se obtiene una clasificación sobre la tercera de ellas.

En ambos pipelines se obtienen medidas de exactitud y una matriz de confusión, a partir de las cuál se evalúa el desempeño de las convnets en las tareas que se proponen. A su vez, se guardan en archivos los resultados intermedios, pesos entre otros, para evitar en la medida de lo posible entrenar las convnets en repetidas ocasiones. Asimismo, se efectúan iteraciones sobre los códigos generados en Python para mejorar su calidad y legibilidad (refactorización).

1.4 Planificación con hitos y temporización

1.4.1 Tareas

El tiempo del que se dispone es de 300h, el autor tiene disponibilidad total durante el desarrollo del mismo, por lo que, si hubiera que modificar las horas asignadas a los objetivos o tareas, en base a algún imprevisto, podría realizarlo sin problema.

Teniendo en cuenta lo anterior, se seguirá la siguiente asignación de tiempo:

- Definición de contenidos y plan de trabajo → 60h
- Desarrollo de la investigación → 180h
- Redacción de memoria, presentación y defensa → 60h

A continuación, se detallan las tareas asociadas a cada objetivo específico, la asignación de horas es proporcional a cada uno de ellos en función de los días asignados como se pueden observar en el cronograma expuesto en la sección 4.3.- Calendario.

Objetivo específico 1.

- Elaborar el estado del arte.

Objetivo específico 2.

- Entender el significado de las imágenes histológicas y su patología.
- Realizar una división del conjunto de datos para obtener muestras para el entrenamiento, validación y test del segundo pipeline.

Objetivo específico 3.

- Estudiar información sobre redes neuronales convolucionales.
- Instalar y configurar Tensorflow-Keras, así como otras herramientas relacionadas.
- Familiarizarse con el uso de Keras.
- Ordenar y pre-procesar las imágenes histológicas de cáncer de colon.
- Aplicar distintas arquitecturas de redes neuronales convolucionales.
- Extraer descriptores profundos para la caracterización de las imágenes histológicas de cáncer de colon.

Objetivo específico 4.

- Estudiar información sobre varios algoritmos clásicos.
- Clasificación de las imágenes histológicas.
- Validar la clasificación mediante diferentes medidas de desempeño.

- Justificar y evaluar los mejores modelos con sus respectivas medidas de desempeño.

Objetivo específico 5.

- Estudiar información sobre *fine tuning* de las redes neuronales convolucionales.
- Seleccionar arquitecturas y redes pre-entrenadas.
- Validar la clasificación mediante diferentes medidas de desempeño.

Objetivo específico 6.

- Hacer cuadro comparativo entre los diferentes métodos y el propuesto, teniendo como base las medidas de desempeño.
- Realizar una discusión de los resultados obtenidos.

1.4.2 Herramientas

Se enumeran las herramientas que se han contemplado para el desarrollo del trabajo, se promociona el uso de herramientas libres o gratuitas.

Software

- Ms Word, Google docs para editar los textos.
- GanttProject para la temporización de los objetivos.
- Python v.3.x y la plataforma Anaconda, en la que incluyen diversas librerías como NumPy, SciPy, Scikit-learn etc. Instalado todo ello en un sistema Linux.
- Tensorflow una librería para el cálculo numérico. Keras una librería que facilita la implementación de Tensorflow.
- Webapp www.draw.io para realizar mapas conceptuales.
- OBS – Open Broadcaster Software para la grabación de la presentación.

Hardware

- tarjeta gráfica (GPU) Nvidia GeForce GTX 960, detallada en los anexos.

1.4.3 Calendario

Se presenta a continuación un diagrama de Gantt de lo expuesto en la sección 4.1.- Tareas, teniendo en cuenta además los siguientes aspectos:

- Los hitos PEC-2 y PEC-3, son fechas en los que se hará entrega de los avances parciales obtenidos hasta ese momento.
- Los conocimientos previos indicados en la sección 3.- Enfoque y método a seguir, se promociona el periodo de documentación bibliográfica a tal efecto.
- Redacción de los avances parciales, de la memoria y sus respectivas revisiones por parte del director del trabajo.

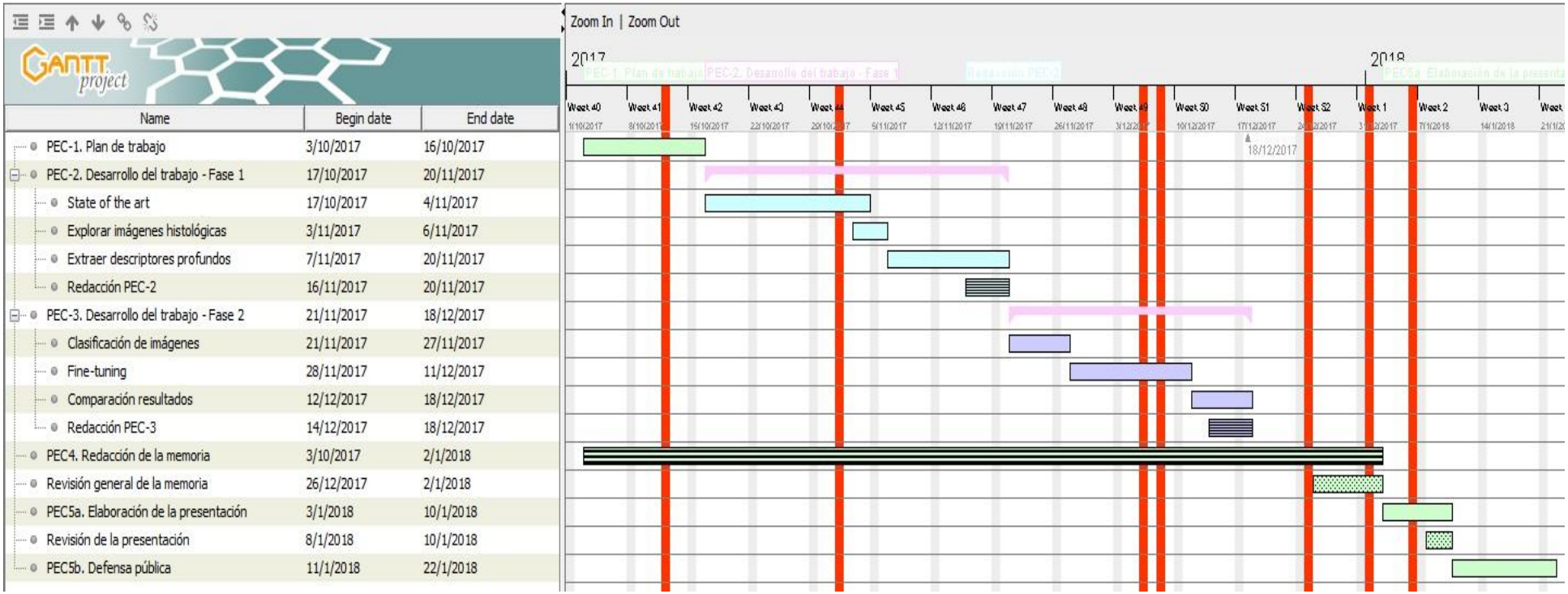

Figura 1 Planificación temporal

1.4.4 Análisis de riesgos

Los principales riesgos que se contemplan son, la capacidad de computación, el coste asociado al trabajo, el tiempo disponible y la privacidad de los datos manejados.

- Con respecto a la capacidad de computación, se dispone de una tarjeta gráfica (GPU) Nvidia GeForce GTX 960, que para realizar las primeras pruebas será suficiente. Al tratarse de una tarjeta con una capacidad de cálculo escasa, se tendrá en cuenta el uso de tarjetas profesionales, en servidores de la Universidad Politécnica de Cataluña o incluso recurrir a servicios de computación en la nube.

- El costo estimado es bajo debido a:
    - El equipo informático del que se hará uso es de personal.
    - No se requerirá la impresión de material.
    - Se hará uso de herramientas informáticas libres, incluido el sistema operativo, sin menoscabar la calidad del trabajo, salvo para la edición de la memoria que se recurrirá a Ms Word.

  En caso de recurrir a la computación en la nube, necesitar alguna licencia de *software* comercial se efectuaría una comparativa de las distintas opciones, quedando reflejado en la memoria.

- Con respecto al tiempo disponible, se considerarán los domingos como días de descanso y se tendrán en cuenta los siguientes días festivos,

    - ✓ 12 de octubre de 2017
    - ✓ 1 de noviembre de 2017
    - ✓ 6, 8 y 25 de diciembre de 2017
    - ✓ 1 y 6 de enero de 2018

  Se tendrán en cuenta como días efectivos para la revisión de la evolución del trabajo los días laborables de lunes a viernes.

- Un problema detectado es el periodo en el que se debe elaborar la memoria y su revisión, son fechas navideñas, por este motivo se decide de forma consensuada con el consultor escribir partes de la memoria conforme se desarrolla el trabajo.

- Las imágenes liberadas en [3] siguen los preceptos de la declaración de Helsinki [11], el cuál es un código ético que debe seguirse por parte de los científicos que trabajen en experimentación humana, por lo que los riesgos de violar la citada privacidad son nulos. En el anterior artículo, se cita de forma explícita la aprobación recibida por parte de la junta ética de la Universidad de Heidelberg en la realización de los experimentos llevados a cabo.

1.5 Breve sumario de los productos obtenidos

Se obtendrán los siguientes productos:

- Una memoria en la que se detallarán todos los procesos y resultados obtenidos durante el desarrollo del trabajo.
- Se incluirán los scripts programados para asegurar la reproducibilidad de los resultados. Todos ellos estarán alojados en un repositorio en Github, para dar mayor visibilidad y transparencia al trabajo.

1.6 Estructuración del trabajo

Estos son los capítulos en los que se estructurará el trabajo.

Capítulo 1. Introducción.

- Se trata de este capítulo, se presenta una descripción y justificación del trabajo, así como los objetivos y tareas del mismo, con su correspondiente calendario.

Capítulo 2. Conjunto de datos.

- Se describe el conjunto de datos liberados en [3]. Se muestran las características inherentes a cada imagen y se establecen las muestras para el entrenamiento y test de las convnets que se implementarán.

Capítulo 3. Redes convolucionales y sus arquitecturas.

Capítulo 4. Transferencia de aprendizaje.

- Se explicarán brevemente las arquitecturas que se implementarán.
- Pipeline 1: Extracción de descriptores profundos de redes neuronales convolucionales y clasificación de imágenes histológicas de cáncer de colon.
- Pipeline 2: Se ajustan los pesos de algunas capas de las redes neuronales, se procede a clasificarlas de nuevo para evaluar su desempeño comparando estos ajustes con los obtenidos en el pipeline anterior.

Capítulo 5. Discusión sobre los resultados.

- Se presentan los resultados obtenidos y se comparan con otros trabajos.

Capítulo 6. Conclusión.

- Se presentan las conclusiones obtenidas durante el trabajo realizado.

Capítulo 7. Referencias bibliográficas.

- Listado de todas las referencias consultadas, libros, artículos, webs, etc.

Capítulo 8. Anexos.

- Se incluye un glosario, la descripción del equipo, la instalación y configuración de las librerías necesarias, gráficas secundarias y el conjunto de códigos utilizados para el desarrollo de la memoria.

2. Conjunto de datos

Origen de los datos

El conjunto de datos consiste en un grupo de imágenes que fueron liberadas bajo Creative Commons Attribution 4.0 International License pudiéndose descargar en [10]. Cabe mencionar que la realización de los experimentos siguió la Declaración de Helsinki [11] y fue aprobada por el comité de ética de la Universidad de Heildelberg (Alemania).

Exploración del conjunto de datos[1]

Las imágenes liberadas por los investigadores, están divididas en dos subconjuntos, el presente trabajo se centra en el primero de ellos, aunque se describen a continuación ambos. Es importante tener en cuenta que las imágenes están ordenadas en subcarpetas cuyo nombre representa la categoría a la que pertenecen.

Conjunto de datos 1

En este conjunto se hallan 8 categorías de tejido, listadas a continuación. En cada una de ellas hay 625 imágenes que no se solapan entre sí. Cada imagen tiene un tamaño de 150 x 150 píxeles, que se corresponden con 74µm de tejido. En total son 5000 imágenes. Todas ellas tienen formato tif, con 3 canales de color RGB (Red – Green – Blue), que por defecto no está soportado por Keras y por lo tanto, no reconoce ninguna imagen de forma automática. Para evitar este problema, en el archivo **image.py** cuya ruta puede variar en cada instalación, y que se halla en el módulo *preprocessing* de Keras [12], hay que añadir el formato tif a la lista *white_list_formats*, tal y como se indica a continuación:

```
white_list_formats = {'png', 'jpg', 'jpeg', 'bmp', 'ppm', 'tif'}
```

Código 1 Incluir formato tif al white_list_format

Categorías:

- Epitelio canceroso: Tejido epitelial, se trata de una o varias capas de células que constituyen revestimiento interno de órganos y cavidades, así como la citada mucosa presente en el colon y recto.
- Estroma simple (de composición homogénea, estroma tumoroso, estroma extra-tumoral y tejido blando): Se trata de la matriz extracelular, es un medio de integración fisiológico, en el que están embebidas las células, en este caso la composición es homogénea.

[1] En el presente trabajo no se tendrá en cuenta que el conjunto de imágenes proviene de solo 10 pacientes, en un estudio real habría que tener en cuenta el sesgo que se comete debido a este hecho.

- Estroma complejo (células tumorosas y/o alguna célula inmune): En este caso, se hallan células tumorosas y alguna inmune.
- Células inmunes (conglomerado de células inmunes y folículos linfoides submucosos): Folículo linfoide es una agrupación de células sin organización o estructura, que se encuentra asociado a submucosas. Es decir, una capa por debajo del revestimiento interno de colon y recto.
- Material variado (mucosas, hemorragia, necrosado): los tejidos necrosados son un conjunto de material muerto por alguna patología.
- Glándulas mucosas normales
- Tejido adiposo: Se trata de un tejido conjuntivo cuyas células acumulan lípidos en su citoplasma, cumple tanto funciones mecánicas como fisiológicas.
- Imagen de fondo (no se corresponde con tejido).

Conjunto de datos 2

Se hallan 10 imágenes de gran tamaño, en concreto 5000 x 5000 píxeles, que provienen de una región distinta con respecto a las imágenes del conjunto anterior. Según los investigadores en [3], pueden ser usadas para comprobar distintas combinaciones de descriptores de textura o clasificadores en un entorno real.

En la Figura 2, se presenta una imagen del conjunto de datos 1 de la categoría Estroma, respetando sus dimensiones.

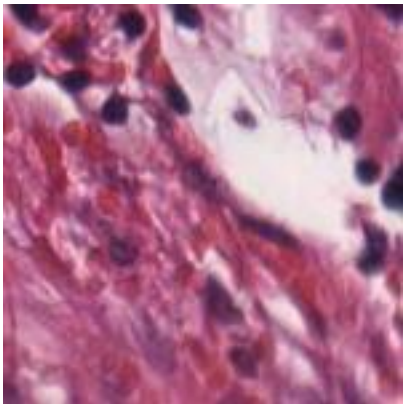

Figura 2 Ejemplo imagen sin reducción

En la Figura 3 se muestran ejemplos de imágenes de cada una de las categorías del conjunto de datos 1, en esta ocasión, y por razones estéticas se han reducido un 50% ambas dimensiones.

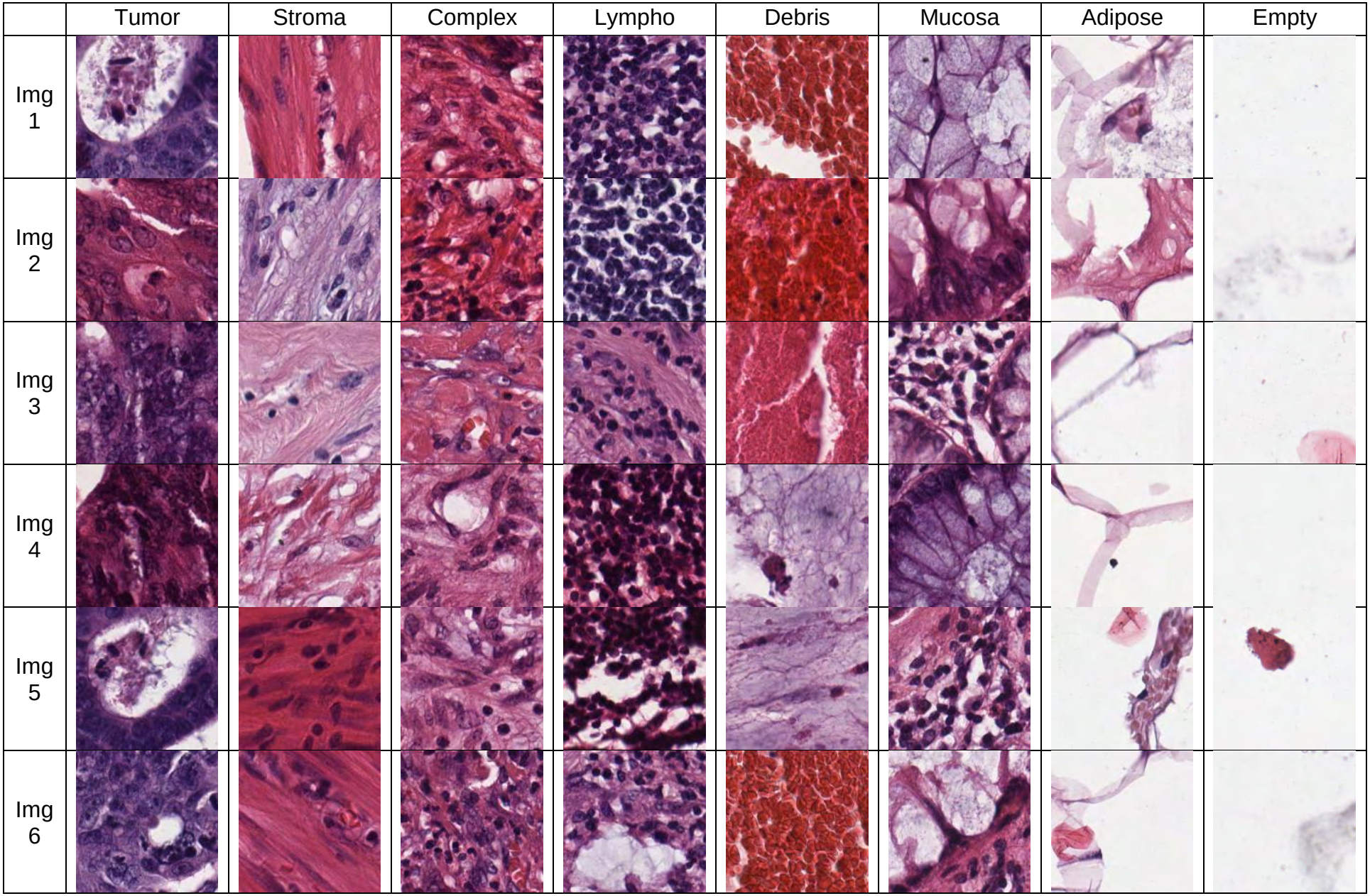

Figura 3 Ejemplos de imágenes histológicas de cáncer de colon y recto

3. Elementos básicos de una red convolucional

Así como las redes neuronales están inspiradas en la red neuronal del sistema nervioso, las convolucionales se inspiran más específicamente en la corteza visual del cerebro, gracias a las aportaciones de David H. Hubel y Torsten Wiesel en sus investigaciones [16], [17]. Sin embargo, no se restringe su uso solamente al reconocimiento de imágenes, pueden aplicarse a reconocimiento del habla [18] o para clasificación de frases [19] con las debidas transformaciones dependiendo de la naturaleza de los datos de entrada.

Las redes convolucionales, poseen características de las redes neuronales como las funciones de activación o las capas totalmente conectadas (*fully connected*), pero introducen dos conceptos, la capa convolucional (*convolutional layer*) y la capa de agrupación o de muestreo (*pooling layer*), las arquitecturas de las redes convolucionales se conforman apilando estos elementos, sin los cuales sería inviable debido a cuestiones técnicas computacionales y de memoria para una red neuronal, el procesamiento de imágenes de gran tamaño [20 – 22].

Capa convolucional

En la primera capa convolucional de este tipo de redes neuronales, las neuronas que la integran se conectan a una porción de la imagen de entrada y no a la totalidad de la misma, de esta forma cada neurona tiene un campo de visión particular. Al concatenar varias capas convolucionales, a una parte de la salida de una capa se conectan ciertas neuronas de la siguiente capa, pero no todas las que integran la segunda capa. Así, tras varias capas convolucionales concatenadas, se obtiene un gran detalle con respecto a las características observadas en la imagen, como formas o colores.

Dentro de cada capa convolucional se apilan a su vez los denominados mapas de características (feature map), un mapa de este tipo es una capa en la que todas sus neuronas utilizan un mismo filtro y comparten sus parámetros característicos como los pesos, por lo que en cada capa convolucional se aplican tantos filtros como mapas de características se apilen en ella. Todos los mapas de características tienen las mismas dimensiones m x n que la capa convolucional a la que pertenecen, por lo que una neurona situada en la posición (i,j) recibe la influencia de todos los sucesivos filtros hallados en la capa convolucional [20 – 22].

Capa de agrupación o muestreo

Este tipo de capas reducen las dimensiones de la imagen de entrada, suelen apilarse tras una capa convolucional, limitando el número de parámetros y operaciones a calcular, reduciendo así el uso de memoria y evitando el sobre ajuste (overfitting) [21].

4. Transferencia de aprendizaje

Para el entrenamiento de redes neuronales convolucionales se ha de contar con un gran número de imágenes catalogadas, del orden de cientos de miles [5], y de equipos con gran capacidad de cómputo [21]. Investigadores con acceso a estos equipos ceden los pesos de las arquitecturas anteriormente citadas a la comunidad, denominadas arquitecturas pre-entrenadas, para su adaptación a problemas específicos.

La transferencia de aprendizaje, se puede producir bien mediante extracción de descriptores profundos, pesos internos de una capa, o mediante ajuste de pesos en una o varias capas internas de las convnet. Este ajuste se produce a través del entrenamiento y validación sobre el conjunto de imágenes del problema específico en cuestión, valiéndose del ajuste previo sobre imágenes de tipo generalista de un conjunto como *imagenet*. En las siguientes secciones se presentan sendos mapas conceptuales que ayudan a entender los pipelines. La explicación se completa con las piezas de código más importantes de la arquitectura VGG16. El resto de los códigos pueden ser consultados en el repositorio de Github a tal efecto creado.

4.1 Pipeline 1 - Descriptores profundos

En la Figura 4 se presenta conceptualmente el pipeline 1.

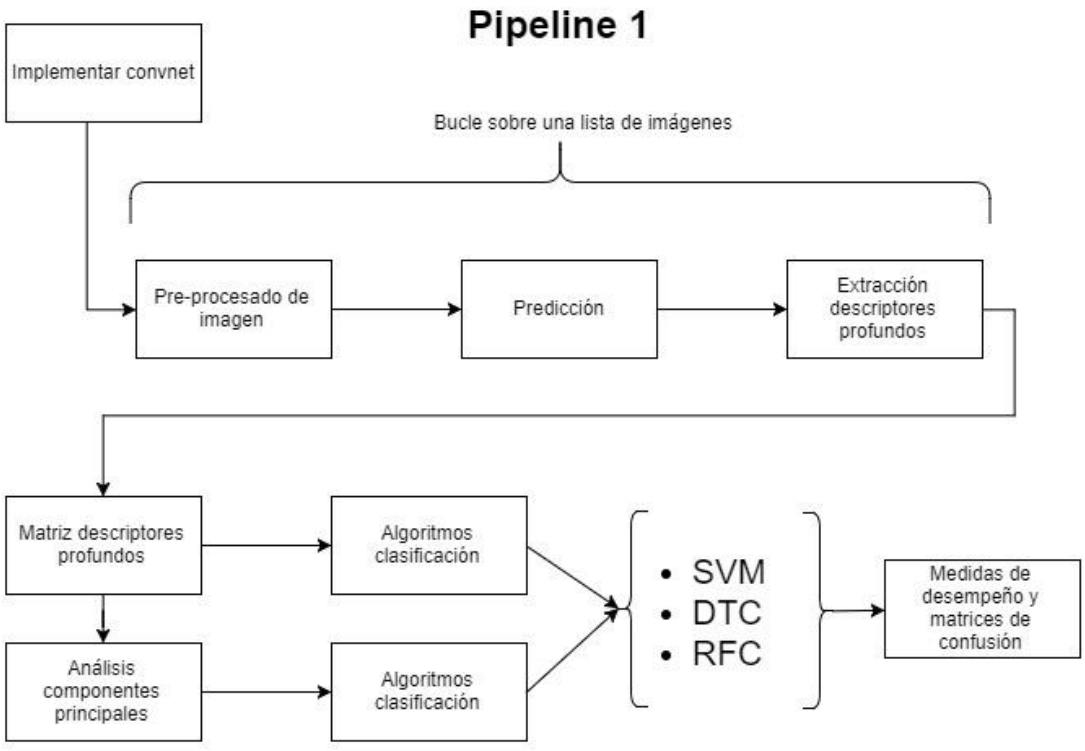

Figura 4 Mapa conceptual pipeline 1

Para la extracción de los descriptores profundos, primero debe declararse la arquitectura. En el Código 2 se muestra cómo se efectúa esta operación de manera secuencial. Pueden observarse, separados entre sí, los cinco bloques convolucionales que conforman la arquitectura. En el Código 3 que sigue siendo parte de esta declaración, se resalta un punto muy importante a tener en cuenta. Las capas declaradas en él, denominadas *fc1* y *fc2*, son las capas totalmente conectadas (*fully connected*). Siguiendo la idea de preservar la máxima variabilidad en los pesos, los descriptores profundos se extraen de la capa fc1, que es la que se declara primero en la arquitectura y se establece como argumento *None* como función de activación, de esta forma los pesos extraídos de esta capa no sufren transformación alguna. Finalmente se cargan los pesos de Imagenet y se compila el modelo.

```
model_no_relu = Sequential()
# bloque 1
model_no_relu.add(ZeroPadding2D((1,1),input_shape=(224,224,3)))
model_no_relu.add(Conv2D(64, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(64, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 2
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(128, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(128, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 3
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 4
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 5
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
```

```
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))

model_no_relu.add(Flatten())
```

Código 2 Implementación arquitectura VGG16

```
# top layer of the VGG net
model_no_relu.add(Dense(4096, activation=None,name="fc1"))
model_no_relu.add(Dropout(0.5))
model_no_relu.add(Dense(4096, activation=None,name="fc2"))
model_no_relu.add(Dropout(0.5))
model_no_relu.add(Dense(1000, activation='softmax'))

model_no_relu.load_weights("vgg16_weights_tf_dim_ordering_tf_kernels.h5")
model_no_relu.compile(optimizer=SGD(lr=0.01), loss='categorical_crossentropy')
```

Código 3 Bloque top layer

En este punto se crea una lista llamada *img_list,* que contiene las rutas de las imágenes. En el Código 4, primero se declara una matriz, *VGG16_dfmap_no_relU*, en la que se irán almacenando los descriptores profundos, tiene como dimensiones 5000 x 4096. Es decir, por filas se tienen las imágenes y por columnas los descriptores profundos de cada una de ellas.

Un bucle recorre la lista mencionada anteriormente, realizando varias operaciones.

1. Se define la extracción de los pesos de la capa *fc1* en la variable *model_fc1.*Tras esto, cada imagen sufre un pre-procesado en tiempo real.
2. Primero, Keras [14] que es la librería en la que se implementan las arquitecturas, carga la imagen mediante la función *load_image()* [12]. La función de forma interna transforma las imágenes, tomando como argumento las dimensiones (ancho y alto en píxeles) de la imagen que admiten las arquitecturas implementadas y un método de interpolación. Al redimensionar las imágenes, se recomienda utilizar, o bien *bicubic* o bien *lanczos*, frente a otros más simples como *nearest,* que es el método por defecto. Para evitar pérdida de información en este tipo de transformaciones, aun con el inconveniente del coste computacional, Lanczos presenta la mejor conservación de información y produce un menor número de impurezas *aliasing* [15], por lo que es el método de interpolación utilizado. En la tabla 1 se presentan las dimensiones utilizadas en el redimensionamiento efectuado en las arquitecturas implementadas.

3. Segundo, la imagen transformada se convierte a un vector de la librería Numpy que se pasa como argumento a la función *preprocess_input()* [13] otra función de la librería Keras.
4. Una vez pre-procesada la imagen mediante el método *predict* de la variable *model_fc1*, se realizan predicciones obteniéndose los descriptores profundos de la imagen. Basta almacenar los descriptores profundos en la matriz VGG*16_dfmap_no_relU* declarada con anterioridad para finalizar el proceso.

VGG16	224 x 224
VGG19	224 x 224
InceptionV3	299 x 299

Tabla 1 Dimensiones, ancho - alto de imagen en píxeles, utilizadas en las arquitecturas implementadas

```
# deep features matrix
VGG16_dfmap_no_relU = np.empty((len(img_list),deep_features))

for index, img_path in enumerate(img_list):

    model_fc1 = Model(input=model_no_relu.input, output=model_no_relu.get_layer('fc1').output)

    # img preprocessing
    img = image.load_img(img_path, target_size=(224, 224), interpolation='lanczos')
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    # model predict
    x = model_fc1.predict(x)

    VGG16_dfmap_no_relU[index,:] = np.squeeze(x, axis=0)
```

Código 4 Pre-procesado de imágenes y extracción de descriptores profundos

En este punto se crea una bifurcación en el pipeline para evaluar los posibles beneficios computacionales que se obtienen al realizar una reducción de dimensionalidad mediante análisis de componentes principales.

Se tienen dos matrices:

- Matriz de descriptores profundos, cuyas dimensiones son 5000 x 4096.

- Matriz de componentes principales, cuyas dimensiones son 5000 x n, donde n es el número de componentes establecidas, variando en cada arquitectura implementada.

Finalmente, a cada una de las matrices anteriores, se les aplican algoritmos de clasificación clásicos, como máquinas de soporte vectorial entre otros. Obteniendo medidas como exactitud o visualizando matrices de confusión se discute sobre el desempeño en cuanto a la calidad de la clasificación. La implementación de los algoritmos anteriores al no presentar dificultad alguna queda registrada en el anexo A.6.

4.2 Pipeline 2 - Ajustes de pesos

En la Figura 5, se presenta un mapa conceptual de este pipeline que se subdivide en dos procedimientos concatenados.

- En el primero, llamado Bottleneck, se entrena un bloque *top layer*.
- En el segundo, denominado ajuste de pesos (*fine tuning*), primero se apila el bloque *top layer* anteriormente diseñado, a una convnet pre-entrenada para después, realizar un entrenamiento de un solo bloque convolucional, quedando el resto de bloques convolucionales inmutables al entrenamiento.

A diferencia del pipeline 1, en este se determinan tres muestras sobre el conjunto de imágenes histológicas:

- Muestra de entrenamiento (*train*), que representa un 60% del total, cuya función es entrenar un bloque convolucional, ajustando los pesos en las capas seleccionadas a tal efecto, dejando al resto sin capacidad de modificarlos.

- Muestra de validación (*validation*), representa el 20%, que evalúa el mejor ajuste del modelo entrenado y evitar los posibles sobre ajustes (*overfitting*).

- Muestra de test (*test*), representa el 20% restante, que indica el error real de la convnet a la hora de clasificar las imágenes histológicas de cáncer de colon y recto.

Una consideración importante que se ha de tener en cuenta es que las funciones de Keras encargadas de suministrar las imágenes a la convnet, determinan las categorías de las mismas mediante los nombres de las carpetas en las que se hallan, por lo que se debe establecer la estructura mostrada en la Figura 6. Es decir, las muestras deben estar en el mismo directorio en el que se encuentren los scripts de Python que implementan las arquitecturas del pipeline 2.

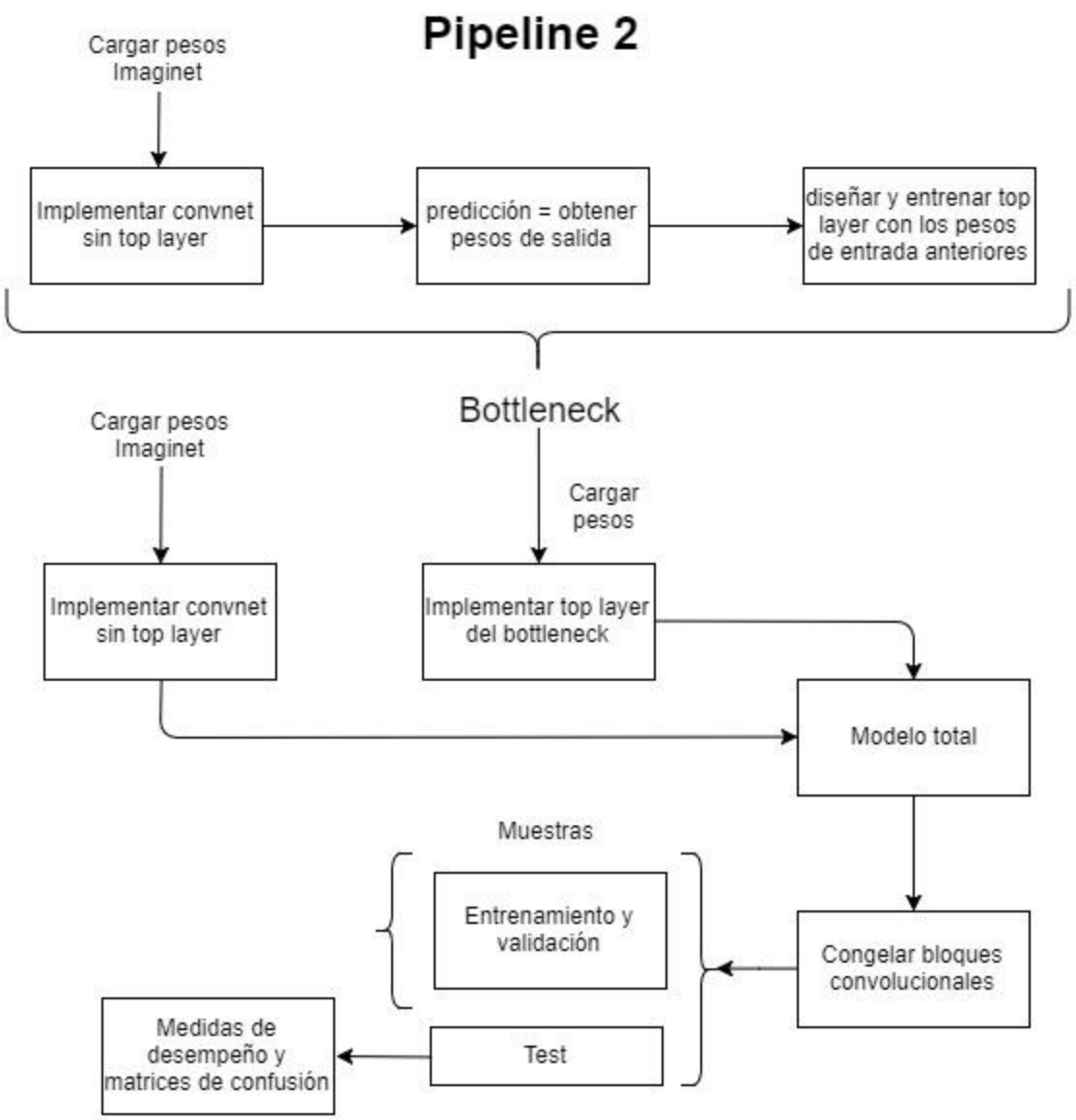

Figura 5 Mapa conceptual pipeline 2

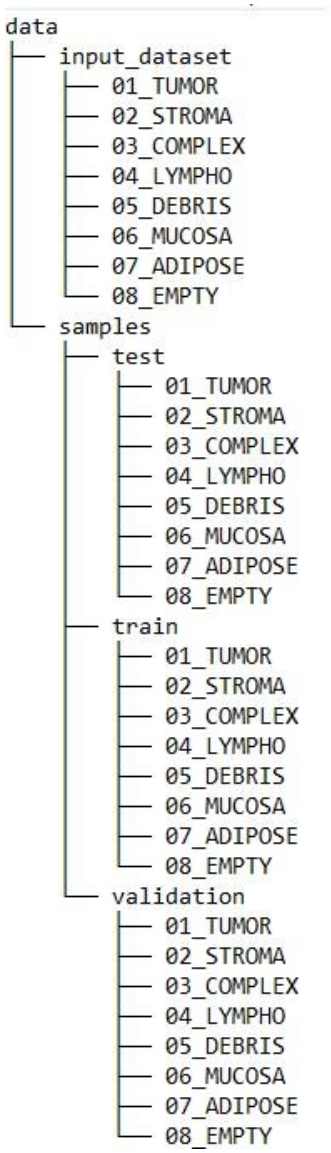

Figura 6 Estructura de carpetas conjunto de datos. En input_dataset se hallan las imágenes originales. En la carpeta samples, se tienen las subcarpetas para las muestras, y en cada una de ellas se encuentran las categorías de las imágenes.

En los Códigos 5 a 7, se detalla cómo obtener las muestras para el este pipeline. Más concretamente en el Código 5, se crea un array de longitud el tamaño de la variable *img_list* que es el listado de las rutas de todas las imágenes, que se recuerda son 5000, para después aleatorizar sus componentes. Tras esto, se reordenan los listados las rutas de todas las imágenes (*img_list_arr*) y sus respectivas categorías (*img_labels_arr*) de acuerdo a las componentes anteriormente calculadas.

```
import numpy as np

# index selection
index_selection = np.arange(len(img_list))

# random array
np.random.shuffle(index_selection)

# paired arrays
img_labels_arr_rand = img_labels_arr[index_selection]
img_list_arr_rand = img_list_arr[index_selection]
```

Código 5 Reorganización aleatoria de los vectores que contienen las rutas y las categorías de las imágenes

En el Código 6, se calculan los índices que serán los cortes del vector, de acuerdo a las proporciones de las muestras, al ser 3 muestras, debe haber 2 cortes.

```
# index cuts
sep_1 = int(TRAIN*len(img_list))
sep_2 = int(VALIDATION*len(img_list))
```

Código 6 Cálculo de los puntos de corte

En el Código 7, a los vectores de categorías y de imágenes, se les aplica los cortes designados por los índices calculados previamente. Estos vectores son usados para determinar a que muestra pertenece cada imagen. Las muestras son las mismas en todas las arquitecturas implementadas.

```
# training
labels_train = img_labels_arr_rand[:sep_1]
img_list_train = img_list_arr_rand[:sep_1]

# validation
labels_val = img_labels_arr_rand[sep_1:sep_1+sep_2]
img_list_val = img_list_arr_rand[sep_1:sep_1+sep_2]

# test
labels_test = img_labels_arr_rand[sep_1+sep_2:]
```

```
img_list_test = img_list_arr_rand[sep_1+sep_2:]
```

Código 7 Asignación de las muestras

Dos comentarios generales con respecto a la obtención de las muestras. Primero, existen funciones como *train_test_split* implementada en *Scikit-learn*, que facilitan este proceso y segundo, se destaca que no se ha buscado explícitamente que dentro de cada muestra las categorías estuvieran balanceadas.

4.2.1 Bottleneck

La idea principal de esta parte del segundo pipeline, es entrenar un pequeño bloque top layer declarado en el Código 9 para que, al apilarlo a una arquitectura pre-entrenada, que se expone en la Sección 4.2.2, no se asignen pesos aleatorios facilitando de esta forma el entrenamiento del bloque convolucional.

En el Código 8 se declara un modelo de la arquitectura VGG16, sin incluir bloque *top layer* y cargando los pesos *imagenet (pre-trained architecture)*. Tanto para la muestra de entrenamiento como la de validación se actúa de igual forma:

- Mediante la función definida en Keras *flow_from_directory* y asignada a la variable *generator*, se pre-procesan las imágenes que se encuentran en los directorios previamente declarados. Además, se especifican como argumentos, entre otros, el tamaño al que se redimensionarán las imágenes, así como la interpolación que se usa a tal efecto.
- El modelo declarado (*model*), se realiza una predicción tomando como argumento la variable *generator*. En definitiva, en *bnfeatures_train* (*bnfeagures_val*) se tienen los pesos de salida de la arquitectura VGG16 sin *top layer* para cada una de las imágenes procesadas.
- El resto de las sentencias sirven para guardar las categorías de cada imagen y los pesos de salida anteriores.

```
# pre-trained model without top layer
model = VGG16(include_top=False, weights='imagenet')
datagen = ImageDataGenerator(rescale=1. / 255)

# train sample
generator = datagen.flow_from_directory(
    train_data_dir,
    target_size = (img_width, img_height),
    batch_size = batch_size,
```

```
    class_mode = None,
    shuffle = False,
    interpolation = 'lanczos')

max_queue_size_train = int(math.ceil(n_train_samples / batch_size))
bnfeatures_train = model.predict_generator(generator, max_queue_size_train)
np.save('../data/output_convnet/VGG16/VGG16_bnfeatures_train_aux.npy',
bnfeatures_train)

# ref attribute classes --> https://keras.io/preprocessing/image/
train_labels = generator.classes # the key attribute
train_labels = to_categorical(train_labels, num_classes=n_classes)

# validation sample
generator = datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode=None,
    shuffle=False,
    interpolation = 'lanczos')

max_queue_size_val = int(math.ceil(n_validation_samples / batch_size))
bnfeatures_val = model.predict_generator(generator, max_queue_size_val)
np.save('../data/output_convnet/VGG16/VGG16_bnfeatures_val_aux.npy',
bnfeatures_val)

val_labels = generator.classes # the key attribute
val_labels = to_categorical(val_labels, num_classes=n_classes)
```

Código 8 Obtención pesos de entrada al top layer

En el Código 9, se cargan en las variables *train_data* y *val_data* los pesos anteriormente calculados. Se diseña un top layer cuyo nombre es *model*, se declara capa a capa de manera secuencial, compilándose al final, de manera similar a como se hizo en el Código 3. Finalmente, se entrena mediante el método *fit* del objeto *model*. Finalmente se salvan los pesos de salida en un archivo tipo h5.

```
train_data = np.load('../data/output_convnet/VGG16/VGG16_bnfeatures_train_aux.npy')
val_data = np.load('../data/output_convnet/VGG16/VGG16_bnfeatures_val_aux.npy')
```

```
# top model, could be with a diff dense, optimizer, momentum ->
https://keras.io/optimizers/
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))


model.compile(optimizer=SGD(lr=0.001),
              loss='categorical_crossentropy', metrics=['accuracy'])


historical_data = model.fit(train_data, train_labels,
                    epochs=n_epochs,
                    batch_size=batch_size,
                    validation_data=(val_data, val_labels))

model.save_weights('../data/output_convnet/VGG16/VGG16_bn_model_aux.h5')
```

Código 9 Declaración y entrenamiento del top layer

Los pesos obtenidos con casi total seguridad pueden variar entre diseños distintos, se destaca que el diseño empleado es personal. Tanto en las arquitecturas VGG16 y VGG19, se ha optado por un diseño igual, al tener una arquitectura similar. En cambio, en InceptionV3 el top layer es ligeramente distinto.

4.2.2 Ajustes de pesos en una capa convolucional

En el Código 10 se declaran dos arquitecturas:

- En *base_model*, una arquitectura pre-entrenada VGG16, sin top layer y con los pesos de la base de datos *imagenet* [5].
- En *top_model*, se declara la misma arquitectura *top layer* que en *Bottleneck*. Se cargan los pesos previamente calculados, de esta forma no se inicializan de forma aleatoria.

Estos dos modelos deben unirse en uno solo de forma que, la salida de *base_model* se corresponda con la entrada de *top_model*. Esto se consigue en la declaración de la variable *model_total* en el Código 11.

Por lo tanto, la entrada de model_total, son las imágenes histológicas y tiene como salida la clasificación de las mismas en las distintas categorías determinadas anteriormente.

```
# base model
base_model = applications.VGG16(weights='imagenet', include_top=False,
input_tensor=input_tensor)

# top model
top_model = Sequential()
top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
top_model.add(Dropout(0.5))
top_model.add(Dense(8, activation='softmax'))

# base model has its weights, now we load the weights on the top layer
top_model.load_weights("../data/output_convnet/VGG16/VGG16_bn_model_aux.h5")
```

Código 10 Carga de modelos y sus pesos

```
# we join base and top it has to be updated to api2
model_total = Model(input= base_model.input, output= top_model(base_model.output))
```

Código 11 Unión de ambos modelos en uno solo

En este punto se ha de determinar el bloque convolucional que recibe entrenamiento. Como ayuda para decidir el número de capas a entrenar, puede consultarse el Código 2 donde se muestran las capas que pertenecen a los distintos bloques convolucionales. En el Código 12 con un simple bucle y recurriendo al atributo *trainable* de las capas que componen la arquitectura *model_total*, se establece el bloque convolucional que recibe entrenamiento. Es importante destacar, que el bloque top layer diseñado también recibe este entrenamiento.

```
for layer in model_total.layers[:15]:
    layer.trainable = False
```

Código 12 Selección de capas a entrenar

Una vez compilado el modelo, se procede a preparar el conjunto de imágenes para el entrenamiento tal y como se indica en el Código 13. En este punto y para evitar un ligero sobre ajuste, se recurre a técnicas de aumento de datos (*data augmentation*). Para ello, se declara *train_datagen*, en la que se aplican transformaciones como rotaciones o volteados de imagen entre otras.

```
# diff with bottleneck, we have to use data augmentation here
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      vertical_flip = True,
      fill_mode='nearest')

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width,img_height),
    batch_size=batch_size,
    class_mode='categorical')

val_datagen = ImageDataGenerator(rescale=1. / 255) # not in the val data

val_generator = val_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical')
```

Código 13 Entrenamiento del modelo (model_total)

A su vez las variables *train_generator* y *val_generator*, declaradas en el Código 13, tienen la misma función que la variable *generator* en el Código 8. Y son pasadas como argumentos a la función *fit_generator*, que es la encargada de entrenar el modelo.

```
historical_data = model_total.fit_generator(
    train_generator,
    samples_per_epoch=n_train_samples,
    epochs=n_epochs,
    verbose = 1,
    validation_data=val_generator,
```

```
    validation_steps=n_validation_samples)
```

Código 14 Entrenamiento del bloque convolucional y el top layer

Una vez el entrenamiento finaliza, se recomienda guardar los pesos para no tener que volver a repetirlo. Para evaluar la calidad del mismo, se procede a declarar un generador para la muestra de test, llamado *test_generator*, pasándolo como argumento al método *predict_generator* del modelo implementado y entrenado anteriormente.

El resultado *predictions*, almacena las categorías que la convnet predice para cada imagen de la muestra test. En el anexo A.6 se definen unas operaciones no muy complejas para obtener los vectores que son necesarios para calcular la matriz de confusión.

```
test_convnet = ImageDataGenerator(rescale=1. / 255)

test_generator = test_convnet.flow_from_directory(
    test_data_dir,
    target_size=(img_width,img_height),
    batch_size=batch_size,
    shuffle =False,
    class_mode='categorical')

predictions = model_total.predict_generator(test_generator, steps = steps)
```

Código 15 Predicción de las imágenes de la muestra de test

5. Resultados y discusión

Tras exponer la metodología empleada en ambos pipelines, se presentan los resultados obtenidos y se comparan, primero entre sí y después con los resultados de los grupos de investigación que hicieron uso del mismo conjunto de imágenes histológicas [3] y [7-9]. En esta sección se presentan las gráficas más representativas o en las que se han obtenido mejores resultados, encontrándose el resto en los anexos A.4 y A.5.

5.1 Análisis de Componentes Principales

Tanto en la Figura 7 mostrada a continuación, como en la Figura 21 y Figura 22, mostradas en el anexo A.4, se presentan las gráficas de la variabilidad acumulada frente al número de componentes principales. El número de componentes principales seleccionadas se obtiene al redondear la cantidad de componentes que acumulan el 95% de la variabilidad explicada.

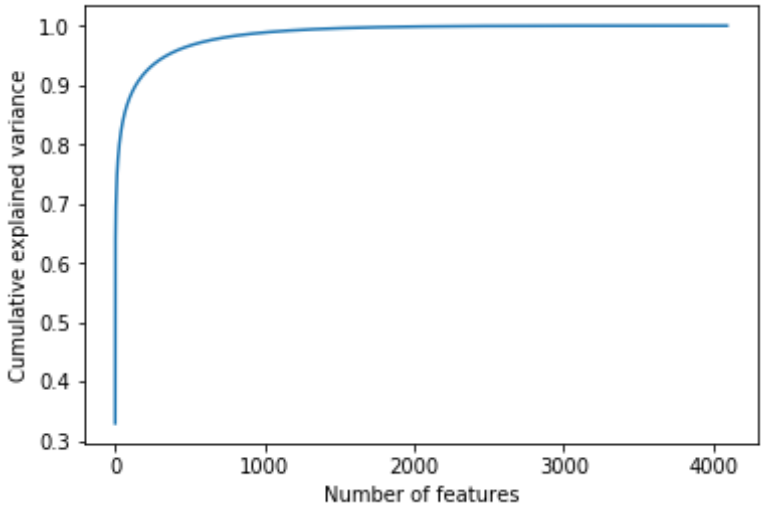

Figura 7 Análisis de componentes principales arquitectura VGG16 - 200 componentes seleccionadas

En la Tabla 2 vienen reflejadas el número de componentes seleccionadas en cada arquitectura. Tanto VGG16 como VGG19, debido a la similitud en su arquitectura, cuentan con el mismo número de componentes. Lo interesante de este análisis es que se requiere de pocas componentes para superar el umbral del 95% de la variabilidad explicada.

Arquitectura	Número de componentes
VGG16	200
VGG19	200
InceptionV3	400

Tabla 2 Número de componentes principales halladas en cada arquitectura

En la Tabla 3 se tienen los tamaños de los ficheros que contienen los descriptores profundos (No PCA) y los que contienen las componentes principales (PCA). Ambos archivos son guardados mediante la librería Pickle. Queda contrastada la diferencia en los tamaños entre ambos, así como la cantidad de memoria requerida para trabajar con ellos.

Tamaño fichero	No PCA	PCA
VGG16	156 Mb	8 Mb
VGG19	156 Mb	8 Mb
InceptionV3	156 Mb	8 Mb

Tabla 3 Tamaño de los ficheros deep features y los ficheros de componentes principales

5.2 Clasificación de los descriptores profundos

5.2.1 Métricas de los descriptores profundos y componentes principales

En las Tablas 4 a 6 se tienen, ordenados por arquitectura y por algoritmo de clasificación, los resultados de la exactitud y tiempos obtenidos para las clasificaciones mediante: máquinas de soporte vectorial (SVM), árboles de decisión (DTC) y bosques aleatorios (RFC); tanto para descriptores profundos (NO PCA) como para las componentes principales (PCA). En negrita se destacan los valores que mejoran significativamente los obtenidos por los investigadores en la tarea de la clasificación multi-categoría sobre el mismo conjunto de imágenes histológicas [3].

VGG16	SVM		DTC		RFC	
	NO PCA	PCA	NO PCA	PCA	NO PCA	PCA
Accuracy	**89.52%**	**89.04%**	72.80%	73.82%	83.12%	77.09%
Tiempo	5’51’’	5’’	2’22’	6’’	15’’	2’’

Tabla 4 Arquitectura VGG16 resultados clasificación primer pipeline

VGG19	SVM		DTC		RFC	
	NO PCA	PCA	NO PCA	PCA	NO PCA	PCA
Accuracy	**88.40%**	**88.18%**	72.03%	73.76%	82.83%	75.74%
Tiempo	5’53’’	6’’	2’15’’	6’’	15’’	3’’

Tabla 5 Arquitectura VGG19 resultados clasificación primer pipeline

InceptionV3	SVM		DTC		RFC	
	NO PCA	PCA	NO PCA	PCA	NO PCA	PCA
Accuracy	**93.84%**	**93.18%**	79.51%	82.56%	87.42%	84.42%
Tiempo	3’26’’	11’’	1’59’’	13’’	15’’	4’’

Tabla 6 Arquitectura InceptionV3 resultados clasificación primer pipeline

5.2.2 Matrices de confusión de los descriptores profundos y de las componentes principales

En las Figuras 8 y 9 se presentan las matrices de confusión de la arquitectura InceptionV3 obtenidas mediante máquinas de soporte vectorial, al haber sido estas las que mejor resultado han arrojado. El resto de matrices de confusión, que se obtienen con otros algoritmos y arquitecturas implementadas se pueden observar en el anexo A.5. Todas estas figuras, muestran dos matrices, en la izquierda se tiene la matriz de confusión sin normalizar, estando a la derecha normalizada.

En la Figura 8 se observan las matrices de los descriptores profundos y en la Figura 9 de las componentes principales. La categoría estroma complejo (*03_COMPLEX*) presenta una baja exactitud en todas las arquitecturas, algoritmos, tanto en descriptores profundos, como en componentes principales, por lo que se puede inferir un problema inherente al conjunto de imágenes histológicas que pertenecen a esta categoría.

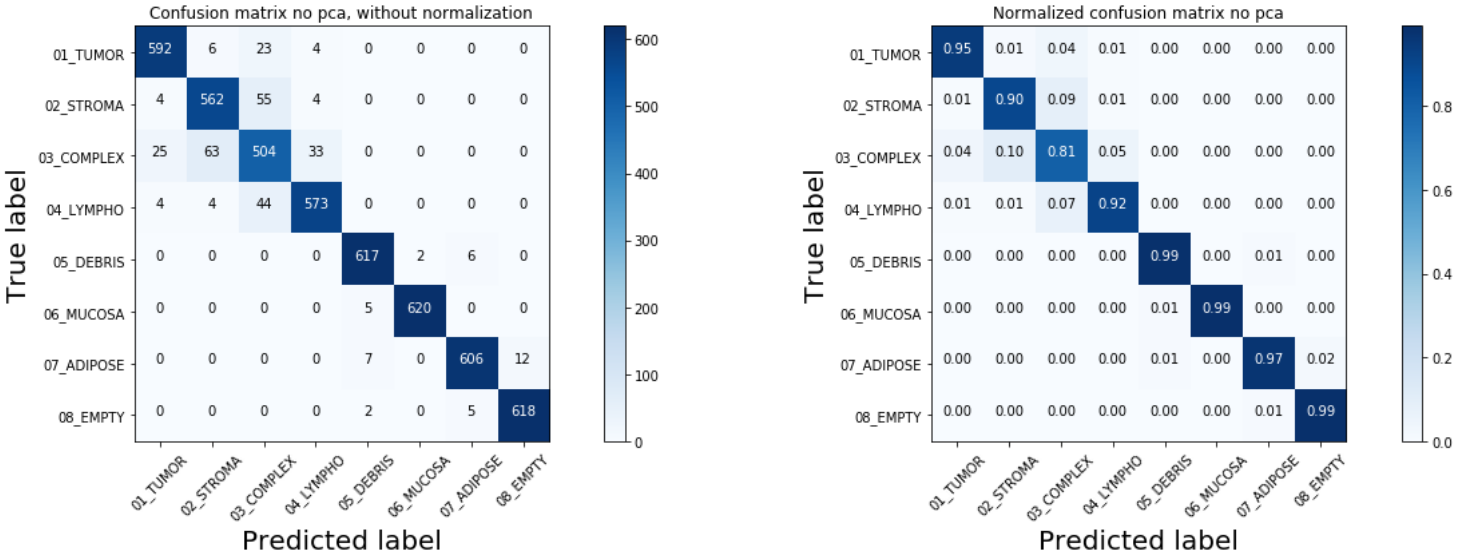

Figura 8 Arquitectura InceptionV3. Algoritmo SVM. Matriz de confusión de los descriptores profundos

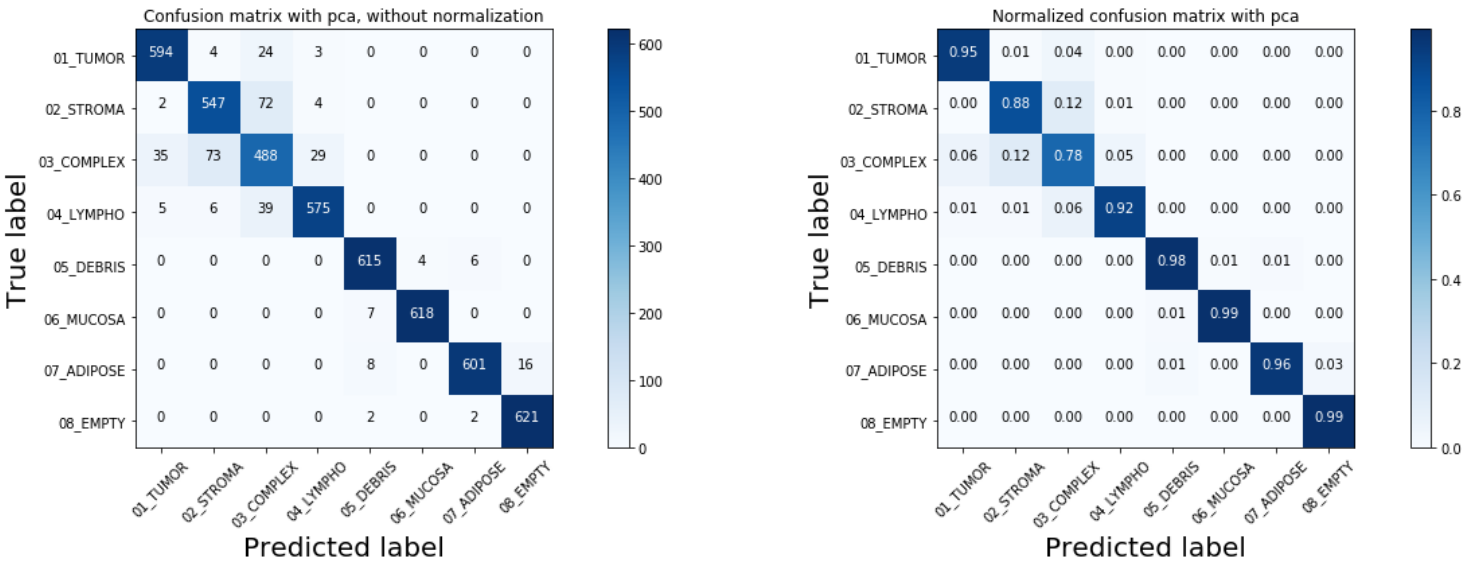

Figura 9 Arquitectura InceptionV3. Algoritmo SVM. Matriz de confusión de las componentes principales

5.2.2 Conclusiones sobre los resultados del pipeline 1

Las conclusiones que se desprenden de este primer pipeline, se agrupan conceptualmente en:

1.- Tiempos de cálculo. Hay que tener en cuenta que la metodología que se usa tomar tiempos de ejecución es la más simple posible y no tiene en cuenta muchos factores. Por lo que los resultados son orientativos y representan una prueba de concepto. Así como lo fueron los tamaños de los ficheros de la Tabla 3.

- Los tiempos de cálculo de las clasificaciones son significativamente menores sobre las componentes principales que sobre los descriptores profundos.
- Las diferencias entre arquitecturas VGG16 y VGG19 no son significativas, debido a que son de estructura similar.

2.- Resultados de exactitud (accuracy %) obtenidos entre algoritmos de clasificación. Para cada arquitectura implementada se tiene que:

- Los mejores resultados se obtienen con máquinas de soporte vectorial. Tanto con descriptores profundos como con componentes principales.
- Por lo general, con descriptores profundos se obtienen mejores resultados en bosques aleatorios que en árboles de decisión. Aunque en ambos algoritmos los resultados no comportan una mejoría con respecto a las máquinas de soporte vectorial.

3.- Resultados de exactitud (accuracy %) obtenidos entre arquitecturas implementadas.

- Debido a la similitud comentada entre las arquitecturas VGG16 y VGG19, las diferencias en los resultados de la clasificación son mínimos.
- Con la arquitectura InceptionV3 se han obtenido los mejores resultados. Con un porcentaje de **93.84%** mediante descriptores profundos **frente a un 87.4%** obtenido por los investigadores en [3]. Incluso con componentes principales se obtiene un resultado superior.

5.3 Bottleneck

En la Tabla 7 se muestra la exactitud (accuracy %) y valor loss obtenidos en el entrenamiento ejecutado en las distintas arquitecturas de la capa *top layer*, mediante las muestras de entrenamiento y validación. En todas las arquitecturas se tienen valores de exactitud similares, aunque el mejor comportamiento se observa en la arquitectura VGG16. Asimismo, el valor loss en todas ellas es similar, pero como en el caso de la exactitud, el mejor valor lo muestra la arquitectura VGG16.

Arquitectura	Exactitud	Loss
VGG16	87.10%	0.37
VGG19	85.10%	0.41
InceptionV3	84.90%	0.47

Tabla 7 Medidas de exactitud (accuracy) y loss, con las muestras de entrenamiento y validación, sobre el Bottleneck en las distintas arquitecturas implementadas

5.3.1 Gráficas de exactitud y loss en las iteraciones del entrenamiento del top layer

En las Figuras 10 a 12 se muestra la evolución de las variables anteriores, exactitud y loss, a lo largo de las 50 iteraciones. En las arquitecturas VGG16 y VGG19, Figuras 8 y 9, tienen un comportamiento similar y característico de un buen entrenamiento. Sin embargo, en la Figura 12, que pertenece a la arquitectura InceptionV3, presenta una gráfica que se puede corresponder con un ligero sobre ajuste (*overfitting*).

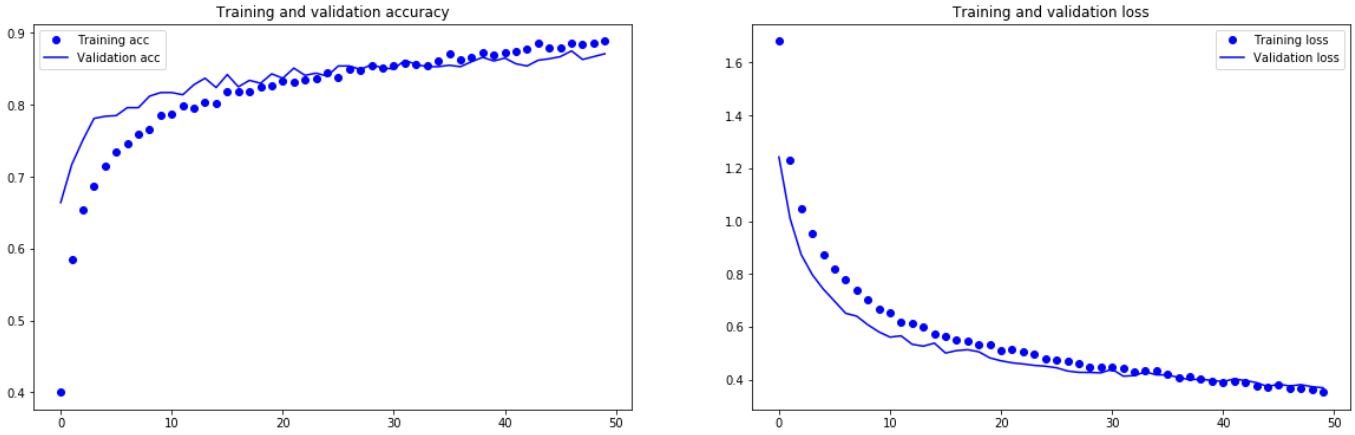

Figura 10 Evolución de las medidas de desempeño exactitud y loss en 50 iteraciones sobre la arquitectura VGG16

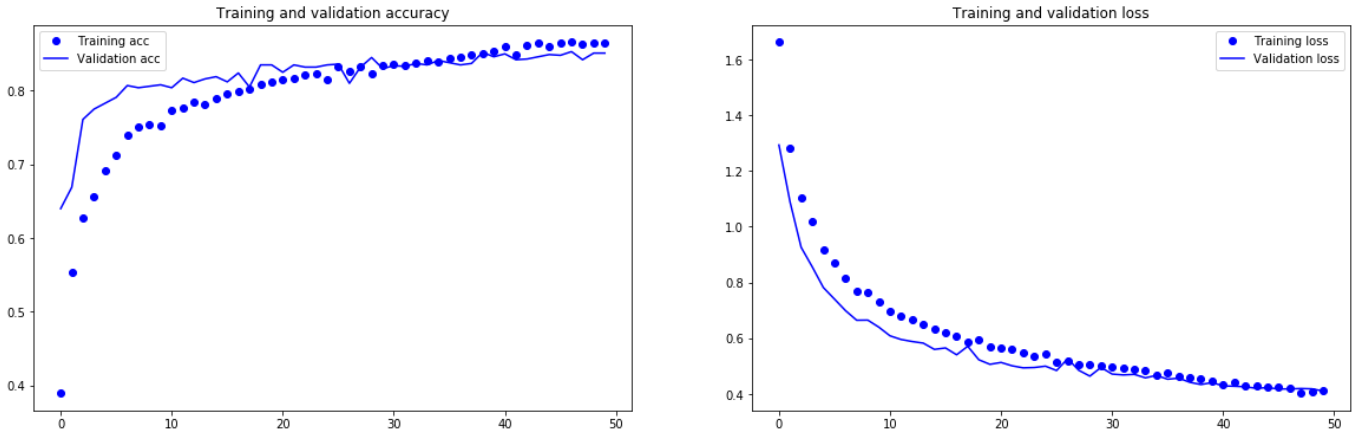

Figura 11 Evolución de las medidas de desempeño exactitud y loss en 50 iteraciones sobre la arquitectura VGG19

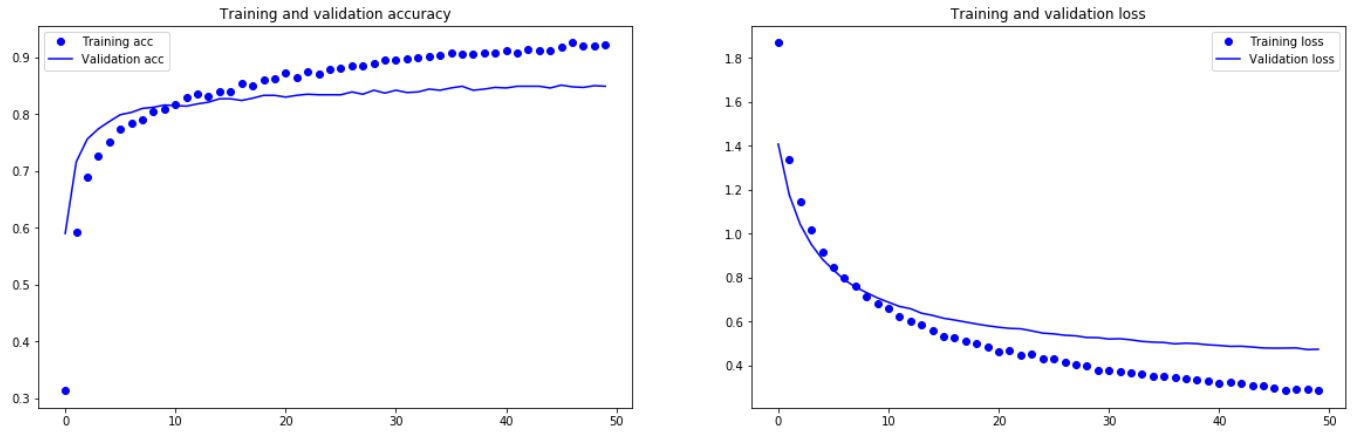

Figura 12 Evolución de las medidas de desempeño exactitud y loss en 50 iteraciones sobre la arquitectura InceptionV3

5.4 Ajustes de pesos

5.4.1 Métricas tras el ajuste de los pesos de un bloque convolucional

En la Tabla 8 se exponen los resultados de la clasificación que realiza la convnet de las imágenes histológicas de la muestra test. Las arquitecturas VGG16 y VGG19, tienen un comportamiento similar como se puede esperar tras los resultados observados del entrenamiento del bloque convolucional y teniendo en cuenta su arquitectura similar. Sin embargo, InceptionV3 arroja los peores resultados. En la Figura 13 puede observarse la evolución de las medidas de desempeño tras el entrenamiento de la arquitectura InceptionV3, este comportamiento anómalo puede deberse a un sobre ajuste. En la Sección 5.4.3 se discuten posibles mejoras para o bien mitigar o bien evitar el sobre ajuste.

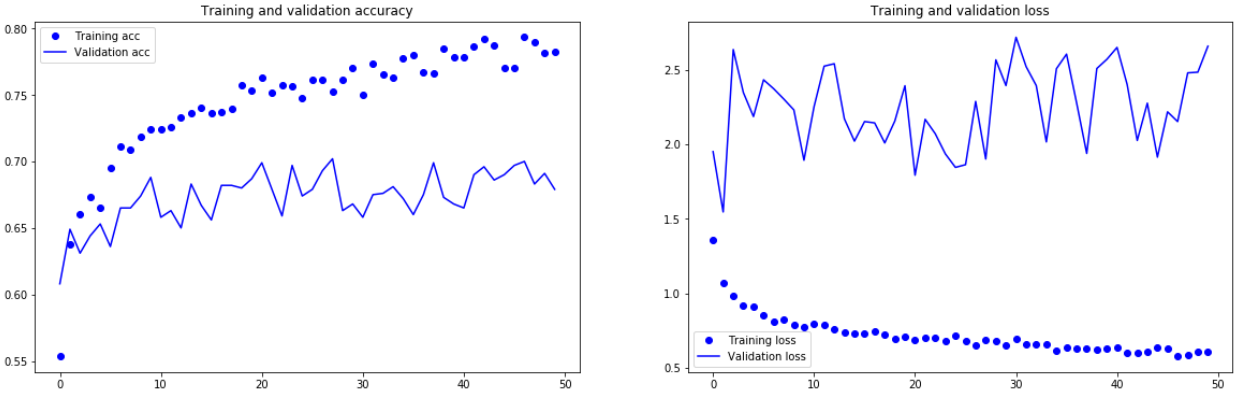

Figura 13 Evolución de las medidas de desempeño exactitud y loss en 50 iteraciones sobre la arquitectura InceptionV3 tras el ajuste de pesos en un bloque convolucional

Arquitectura	Exactitud	Loss
VGG16	91.10%	0.26
VGG19	92.00%	0.23
InceptionV3	67.90%	2.65

Tabla 8 Medidas de exactitud (accuracy) y loss, con la muestra test, sobre el ajuste de pesos, con las muestras de entrenamiento y validación, de las distintas arquitecturas implementadas

5.4.3 Matrices de confusión tras el ajuste de los pesos de un bloque convolucional

En la Figura 14 se muestran las matrices de confusión de la arquitectura VGG19, cuya arquitectura tiene un mejor desempeño en la clasificación de las imágenes de la muestra de test tras el entrenamiento del bloque convolucional. Al igual que en el primer pipeline, se obtiene una mala clasificación de la categoría estroma complejo con respecto al resto de categorías.

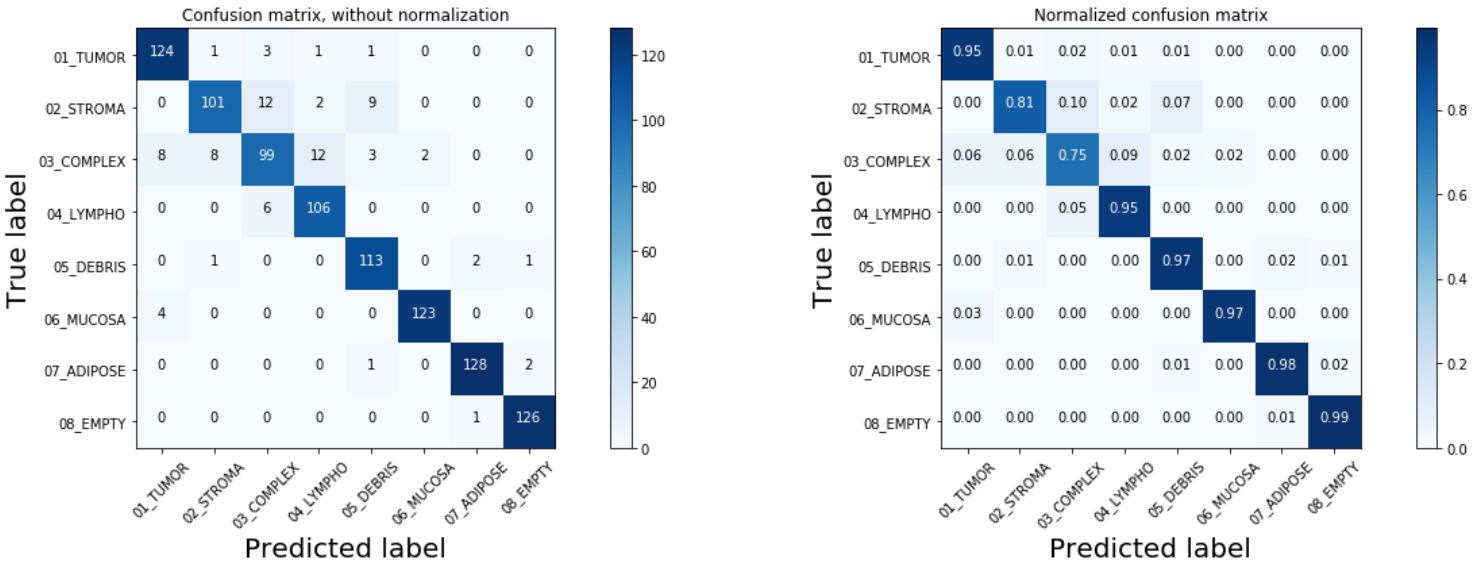

Figura 14 Matriz de confusión de la clasificación, sobre la muestra de test, tras el fine tuning de la arquitectura VGG19

5.4.3 Conclusión sobre los resultados del pipeline 2

En este pipeline destaca la arquitectura InceptionV3 por su mal comportamiento. En igualdad de implementación entre arquitecturas, del uso de las mismas muestras para el entrenamiento y validación, así como las mismas funciones de aumento de datos (data augmentation) este comportamiento puede deberse a un sobre ajuste.

Para mitigarlo o eliminarlo, existen posibles mejoras como reducir el tamaño de las muestras de validación y test aumentando la de entrenamiento, balancear de forma explícita el tamaño de las categorías dentro de cada muestra, utilizar funciones de aumento de datos más agresivas o recurrir al uso de regulizadores.

5.5 Comparación entre ambos pipelines

A la vista de los resultados, tanto por eficiencia como por resultados, el primer pipeline tiene las mejores prestaciones. En la Tabla 9 se muestran los mejores resultados

obtenidos en cada pipeline. En ambos pipelines se obtienen resultados que mejoran los obtenidos por los investigadores de [3].

Dos puntos importantes a tener en cuenta con respecto al conjunto de imágenes histológicas:

- Las imágenes que lo conforman, tienen unas características muy distintas al conjunto *imagenet*, que es el utilizado para entrenar las arquitecturas implementadas [22].
- El conjunto puede considerarse como muy reducido

Por lo que, suele ser de mayor utilidad el método implementado en el pipeline 1, es decir, extraer los pesos de una capa de la arquitectura y aplicarles un método clásico de clasificación. También hay que tener en cuenta el menor tiempo necesario para obtener resultados en el primer pipeline frente al segundo.

	Características de la implementación	Exactitud
Pipeline 1	InceptionV3 – SVM – descriptores profundos	93.84%
Pipeline 2	VGG19	92.00%

Tabla 9 Mejores resultados obtenidos en los dos pipelines

5.6 Comparación con otros artículos

En común con los experimentos mostrados en [7] se tienen el algoritmo de reducción de dimensionalidad, análisis de componentes principales y la arquitectura implementada VGG16. Las diferencias son variadas, pero fundamentalmente, la extracción de los descriptores profundos es de la última capa *(fully connected*) de la arquitectura mientras que, en el presente estudio es en la primera donde se realiza tal extracción. Con respecto a los parámetros utilizados en el algoritmo de reducción de dimensionalidad, en el presente estudio se implementó sin evaluar *kernels* distintos al que se tiene por defecto.

Pese a todas las diferencias expuestas, en ambos estudios se concluye que, mediante el análisis de componentes principales, la exactitud se reduce con respecto a la obtenida directamente con los descriptores originales pero los beneficios en cuanto a reducción de tiempos de ejecución o memoria necesaria son evidentes.

Con respecto al **87.4%** obtenido en el estudio de referencia [3], se destaca la mejoría en la clasificación multi-categoría en ambos pipelines. El pipeline 1 arroja la mejor clasificación con un **93.84%**, una mejora sustancial pero ciertamente mejorable.

5.7 Discusión final

La característica más destacable con respecto al conjunto de imágenes es la categoría estroma complejo. En ambos pipelines, con independencia de la arquitectura implementada, esta categoría presenta una baja exactitud en su clasificación. Por lo tanto, se puede considerar un problema inherente a la propia categoría. En ella se hallan tanto células tumorosas como células inmunes. Las categorías contiguas, estroma simple y linfoma, se clasifican incorrectamente como estroma complejo y viceversa, siendo un foco importante en cuanto a la reducción general de la exactitud.

Una idea interesante es obtener mapas de calor (*heatmaps*) para analizar qué zonas de la imagen histológica determinan que la convnet la clasifique incorrectamente. Quizás se obtengan conclusiones que mejoren la clasificación, ya no solo de la categoría estroma complejo sino de todas en general [23].

Por todo lo anteriormente expuesto y ante los resultados obtenidos en ambos pipelines, pueden considerarse futuras investigaciones en convnets en clasificación automática de imágenes histológicas. Pudiendo llegar a incorporarse al conjunto de herramientas utilizadas por investigadores en estudios patológicos.

El desempeño de las convnets puede verse mejorado en muchos aspectos, atendiendo a una búsqueda sistemática de los mejores hiperparámetros en ambos pipelines, en funciones de aumento de datos más agresivas (*data augmentation*). Mejorando la obtención de las muestras. Creando conjuntos con mayor número de imágenes histológicas, pueden apilarse sin llegar a sobre entrenarse, mayor número de capas o bloques convolucionales. También pueden considerarse otras arquitecturas de la literatura científica como ResNet en cualquiera de sus versiones, recurrir al uso de Autencoders [9], o incluso si las circunstancias son adecuadas crear nuevas arquitecturas adaptadas a este tipo de imágenes.

Como prueba de concepto, aspectos como eficiencia y reutilización de código, son secundarios, pero en caso de llevar a producción este tipo de redes neuronales, sería un factor a tener en cuenta. Sobre todo, a la hora de hallar los citados hiperparámetros y encontrar la mejor exactitud en el modelo implementado.

6. Conclusiones

Tanto el objetivo general como los específicos se han cumplimentado satisfactoriamente. Queda reflejado que las convnets pueden ser de utilidad a la investigación patológica en la clasificación automática de imágenes histológicas.

En la elaboración del estado del arte se encuentran estudios similares basados en el mismo conjunto de imágenes [3] [7-9], cuyos resultados refuerzan la anterior afirmación. Ambos pipelines implementados evidencian puntos fuertes y puntos débiles respecto al uso de las convnets en este tipo de tareas [4]. Y por lo tanto posibles líneas de investigación.

En cuanto al desarrollo de los pipelines se alcanzaron los siguientes logros:

- Mejora en la puntuación de la clasificación (exactitud) con respecto al estudio de referencia [3].
- Uso de la herramienta de versionado de código Git – Github.
- Experiencia adquirida en el continuado proceso de depuración de los errores que se encontraron, tanto en instalación, configuración y uso, en las dos plataformas usadas Linux y Windows 10.
- Si bien es cierto que se ha desarrollado la memoria en Ms Word. Se inició el desarrollo en LaTeX, pero debido a los continuos retrasos se determinó usar Ms Word por la sencillez en su manejo.
- Previamente se tenía cierto conocimiento del lenguaje Python, básico por determinarlo de alguna manera. En el momento en el que se escriben estas líneas, este conocimiento ha aumentado, tanto en cantidad como en calidad.
- Conocimiento adquirido, tanto en la enfermedad de cáncer de colon y recto, así como en los procedimientos para la obtención de las imágenes histológicas y lo que representan las categorías. Así mismo y en esta línea, mejora en la búsqueda de información científica y otros aspectos que, aunque no calificables, al autor le reportan otros beneficios.
- La lectura de diversos artículos, libros y blogs, gracias a los cuales el conocimiento sobre convnets puede considerarse como algo muy positivo y con posibilidad de aplicarlo a proyectos personales del autor

Se proponen las siguientes perspectivas futuras:

- Búsqueda de mejores implementaciones en las convnets. En ambos pipelines. Uso directo de Tensorflow sin recurrir a Keras o Pytorch.
- Uso de otras técnicas de reducción de dimensionalidad [7] u otras metodologías como, por ejemplo, selección de descriptores profundos.
- Uso de técnicas de búsqueda iterada de hiperparámetros en modelos clásicos de clasificación. Válido este concepto para el segundo pipeline, aunque en el equipo

en el que se ha desarrollado era totalmente inviable computacionalmente hablando.

- Uso de funciones de aumento de datos más agresivas (*data augmentation*).
- Pruebas con otras combinaciones de capas superiores en el primer pipeline.
- Depuración de la implementación del segundo pipeline, con el objetivo de estudiar la evolución de los parámetros exactitud y *loss*. Este hecho puede tener origen en el conjunto de datos.
- Registro de las salidas de las capas internas de las convnets, y visualizar su contenido, para realizar mapas de calor (*heatmaps*) sobre las zonas que más identifican una categoría en una imagen histológica [23].
- Obtención de más medidas de evaluación, este punto no es de difícil implementación.
- Uso de métodos implementados en Keras para crear historiales (logs) una herramienta que habría dado mayor calidad al trabajo.
- Determinar con asistencia de investigadores patólogos en cáncer de colon y recto la baja exactitud en las clasificaciones con respecto a las categorías estroma simple y estroma complejo. O recibir información detallada sobre la obtención de las imágenes que forman parte en el conjunto de imágenes utilizado.
- Realizar una regresión logística tumor-estroma para comparar resultados con [3].

7. Bibliografía

[1] A. Eguino Villegas, A. I. Fernández Crespo, B. Fernández Sánchez, G. García Álvarez, and C. Pascual Fernández, “Cáncer colorrectal. Una guía práctica,” Madrid.

[2] “Diagnóstico precoz del cáncer de colon.” [Online]. Available: https://www.aecc.es/SobreElCancer/CancerPorLocalizacion/cancerdecolon/Paginas/diagnosticoprecoz.aspx. [Accessed: 20-Oct-2017].

[3] J. N. Kather et al., “Multi-class texture analysis in colorectal cancer histology,” Nat. Publ. Gr., 2016.

[4] F. A. Spanhol, L. S. Oliveira, C. Petitjean, and L. Heutte, “Breast Cancer Histopathological Image Classification using Convolutional Neural Networks,” Int. Jt. Conf. Neural Networks (IJCNN 2016), pp. 2560–2567, 2016.

[5] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255.

[6] F. Chollet, “weights.” [Online]. Available: https://github.com/fchollet/deep-learning-models/releases. [Accessed: 25-Oct-2017].

[7] S. Cascianelli et al., “Dimensionality Reduction Strategies for CNN-Based Classification of Histopathological Images,” Springer, Cham, 2018, pp. 21–30.

[8] F. Ciompi et al., “The importance of stain normalization in colorectal tissue classification with convolutional networks,” Feb. 2017.

[9] T. D. Pham, “Scaling of Texture in Training Autoencoders for Classification of Histological Images of Colorectal Cancer,” Springer, Cham, 2017, pp. 524–532.

[10] J. N. Kather et al., “Collection of textures in colorectal cancer histology,” doi.org, May 2016.

[11] Adopted, Assembly, and Helsinki, “World Medical Association Declaration of Helsinki Ethical Principles for Medical Research Involving Human Subjects,” WMA Gen. Assem. Repub. South Africa, vol. 35, 1964.

[12] F. Chollet, “img preprocessing 1.” [Online]. Available: https://github.com/keras-team/keras/blob/master/keras/preprocessing/image.py. [Accessed: 10-Nov-2017].

[13] F. Chollet, “img preprocessing 2.” [Online]. Available: https://github.com/fchollet/deep-learning-models/blob/master/imagenet_utils.py. [Accessed: 10-Nov-2017].

[14] F. Chollet, “Keras Documentation.” [Online]. Available: https://keras.io/. [Accessed: 10-Nov-2017].

[15] P. S. Parsania and P. V Virparia, “A Comparative Analysis of Image Interpolation Algorithms,” Int. J. Adv. Res. Comput. Commun. Eng., vol. 5, no. 1, 2016.

[16] D. H. HUBEL, “Single unit activity in striate cortex of unrestrained cats.,” J. Physiol., vol. 147, pp. 226–38, Sep. 1959.

[17] D. H. Hubel and T. N. Wiesel, “Receptive fields and functional architecture of monkey striate cortex.,” J. Physiol., vol. 195, no. 1, pp. 215–43, Mar. 1968.

[18] A. Hannun et al., “Deep Speech: Scaling up end-to-end speech recognition,” Dec. 2014.

[19] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” pp. 1746–1751.

[20] A. Géron, Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.

[21] “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/neural-networks-1/. [Accessed: 03-Nov-2017].

[22] “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/transfer-learning/. [Accessed: 03-Nov-2017].

[23] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” Nov. 2013.

[24] “GV-N960G1 GAMING-4GD (rev. 1.0) | Graphics Card - GIGABYTE Global.” [Online]. Available: https://www.gigabyte.com/Graphics-Card/GV-N960G1-GAMING-4GD-rev-10#sp. [Accessed: 28-Dec-2017]

A. Anexos

A.1 Glosario

- TSOH: Test de sangre oculta en heces.
- Hand-crafted features: descriptores diseñados mediante metodologías en ingeniería (traducción libre).
- Convnet: Convolutional network / redes neuronales convolucionales.
- Weights: Pesos de las arquitecturas de las redes neuronales convolucionales.
- Pipeline: Procedimiento.
- Deep Features: descriptores profundos.
- SVM: Support vector machine / máquinas de soporte vectorial.
- DTC: Decision tree classifier / árboles de decisión.
- RFC: Random forest classifier / bosques aleatorios.
- PCA: Principal component analysis / Análisis de componentes principales.
- Transfer learning: Transferencia de aprendizaje.
- Fine tuning: ajuste de pesos en redes convolucionales (en el contexto de este trabajo).
- GPU: Graphics processing unit / tarjeta gráfica.
- RGB: Red Green Blue / Rojo Verde Azul, son los canales de color de las imágenes.
- Convolutional layer: Capa convolucional.
- Pooling layer: Capa de muestreo.
- Feature map: mapas de características (en los códigos deep feature map / dfmap hace alusión a la matriz con los descriptores profundos).
- Overfitting: sobre ajuste, sobre entrenamiento (traducción libre).
- Top_layer: Última capa (bloque) en una arquitectura convolucional.
- Data augmentation: técnica para aumentar el conjunto de imágenes mediante trasformaciones (traducción libre).

A.2 Descripción equipo

Con ánimo de poder reproducir los resultados obtenidos en el presente trabajo, se especifican las características del equipo en el que se han desarrollado las simulaciones. De forma automática y no incluido en el código, se utiliza una semilla común en todas las simulaciones.

A.2.1 Tarjeta gráfica – Graphics Processing Unit – GPU [24]

- GPU – GeForce GTX 960 – año 2015
- Modelo GV-N960G1 Gaming-4GD
- Core clock
    - Boost: 1329 MHz/Base: 1266 MHz in OC Mode
    - Boost: 1304 MHz/Base: 1241 MHz in Gaming Mode
- Process Technology – 28nm
- Interface gráfica – PCI-E 3.0
- DDR5 SDRAM 4Gb – Clock: 7.01MHz – Bus: 128bit

A.2.2 Resto del sistema

- CPU - Intel Core i5-4460 @ 3.2GHz
- RAM - 16Gb (comprobar especificaciones)
- SSD - 256Gb (90Gb libres)
- Linux Mint 64bit

A.3 Instalación y configuración de herramientas

A.3.1 Sistema operativo – Python

Si bien en un principio se comenzó el trabajo con el sistema operativo Ms Windows 10, tras muchísimas dificultades encontradas tanto en la instalación como en la configuración de las distintas herramientas y librerías, se tomó la decisión de comenzar de nuevo el proceso en un entorno Linux.

En el archivo *linux.log* se puede obtener la versión completa del sistema Linux. Se trata de **Linux Mint 18.3 MATE 64-bit**, la versión exacta del kernel instalado se halla en *linux_2.log*.

```
Linux brain 4.10.0-42-generic #46~16.04.1-Ubuntu SMP Mon Dec 4 15:57:59 UTC 2017
x86_64 x86_64 x86_64 GNU/Linux
```

Figura 15 Versión del sistema operativo

En el sistema operativo Linux viene instalado por defecto el intérprete del lenguaje de programación Python. En *python.log* se hallan los detalles de la versión.

```
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct 13 2017, 12:02:49) [GCC 7.2.0] on linux
```

Figura 16 Versión Python instalada

A.3.2 Drivers GPU-Nvidia

Se comenzó por instalar el driver de la tarjeta gráfica para que el propio sistema operativo pueda trabajar con ella. Desde la página oficial y seleccionando las especificaciones de la tarjeta y sistema operativo instalado podemos bajar un archivo con los drivers.

Drivers

- Versión: 384.98
- Fecha publicación: 2 de noviembre de 2017
- Sistema: Linux 64-bit

Durante el desarrollo de las implementaciones se instalaron varios drivers debido a la última actualización cuya fecha aparece reflejada.

A.3.3 Drivers CUDA Toolkit 8.1 y cuDNN

La instalación de las distintas librerías y herramientas se basó en la ayuda oficial ofrecida por Tensorflow.

En esa guía se remite a la documentación de CUDA Toolkit. CUDA es una plataforma para realizar cálculos distribuidos, sobre todo beneficiándose de los núcleos (*cores*) de las tarjetas gráficas.

La instalación de CUDA no es compleja, pero se ha de seguir paso a paso para instalarla correctamente, en este punto no se encontraron dificultades añadidas. En el archivo *cuda.log* se observa la versión instalada. En diciembre del 2017, hubo una actualización que ocasionó problemas debido a que no era totalmente compatible con los whells que ofrece Tensorflow, se recurrió a unos compilados a tal efecto en el siguiente github, allí mismo, los desarrolladores ofrecen instrucciones precisas para su instalación.

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2015 NVIDIA Corporation
Built on Tue_Aug_11_14:27:32_CDT_2015
Cuda compilation tools, release 7.5, V7.5.17
```

Figura 17 Verión CUDA instalada

Tras tener los drivers de la GPU actualizados, se debe proceder a descargar Cuda-Toolkit un archivo ejecutable cuya versión es la 8.0.61. Se trata de unas librerías para poder utilizar la potencia de cálculo de la tarjeta gráfica.

Una vez terminado el punto anterior, se debe instalar cuDNN, una librería que permite aprovechar las capacidades de la tarjeta gráfica y que está diseñada para implementar redes neuronales profundas, entre otras puede trabajar con Tensorflow, Theano, etc. Para su descarga es necesario crear una cuenta gratuita. Durante esta instalación no se produjeron problemas.

A.3.4 Anaconda – Anaconda Navigator

Para instalar la librería Tensorflow se tienen varias posibilidades la elegida por diversos motivos es Anaconda. Anaconda es una plataforma en la que se pueden configurar entornos virtuales, desde la que se pueden lanzar servidores web para Jupyter, gestionar paquetes etc. Entre ellos, se hallan librerías muy útiles para el propósito de la investigación. En este enlace puede encontrarse una completa documentación sobre esta plataforma, a la que se ha recurrido para solventar problemas en la configuración y uso. La versión instalada es 4.3.30.

Finalmente, ya se tienen las herramientas necesarias para instalar Tensorflow que es una librería para cálculo numérico sobre grafos, que son las representaciones de las redes neuronales. Gracias a la interacción entre las librerías CUDA, cuDNN y Tensorflow los cálculos de las redes neuronales son posibles.

Tensorflow

Se crea un entorno virtual en Anaconda como se indica aquí. En una consola Linux, y mediante *conda install -c anaconda anaconda-navigator*, se instalará un entorno visual de Anaconda, desde el cual podremos seleccionar el entorno virtual en que se ha hecho la instalación de Tensorflow. Para arrancar Navigator, simplemente desde una consola Linux se ejecuta *anaconda-navigator*.

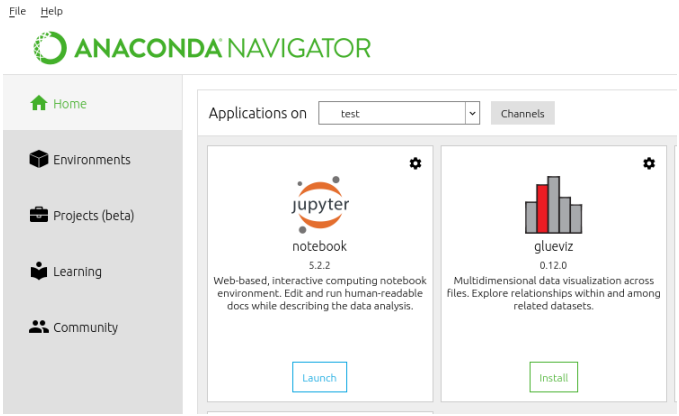

Figura 18 Selección entorno virtual conda

Una vez seleccionando el entorno conda en el que está instalada la librería Tensorflow, se puede lanzar un servidor web desde el cual, los Jupyter notebooks son accesibles. Una vez creado el primer archivo notebook en Jupyter, para comprobar que la que la instalación de Tensorflow es correcta, bastará cargar la librería. Se muestra la versión instalada.

```
In [1]: import tensorflow as tf
        print(tf.VERSION)

        1.4.1
```

Figura 19 Versión Tensorflow instalada

Keras

Keras es una librería escrita en lenguaje Python que simplifica la implementación de convnets. Incluyendo diversas funciones para el pre procesamiento de imágenes o para el aumento de datos (*data augmentation*) entre otras. La instalación, en el entorno virtual creado en pasos previos, a través de anaconda es sencilla, simplemente hay que activar el entorno virtual y ejecutar el siguiente comando *conda install -c conda-forge keras*. Bastará con importar la librería para saber si está correctamente instalada. Dos cosas hay que tener en cuenta con Keras, en cuanto a su configuración. En el archivo *keras.json* hay que especificar dos variables, la primera el backend a utilizar, en este caso Tensorflow, y segundo el orden de los atributos de las imágenes, alto x ancho x número de canales (al ser imágenes RGB son 3). Este es el archivo de configuración usado.

```
{
    "floatx": "float32",
    "epsilon": 1e-07,
    "backend": "tensorflow",
    "image_data_format": "channels_last"
}
```

Código 16 Archivo json de configuración Keras

```
In [1]: import keras
        Using TensorFlow backend.
```

Figura 20 Manera de cargar y comprobar la correcta instalación de Keras

A.3.5 Git – Github

Git es una herramienta de versionado de archivos. Se puede mantener una traza con los cambios hechos en uno o varios archivos. Se considera una herramienta muy útil y necesaria en cualquier proyecto como este. Por su parte, Github es un servidor remoto (en la nube) en el que se almacenan estos archivos con sus respectivas versiones, es necesaria una cuenta que es gratuita.

Git, al igual que el intérprete de Python, es una herramienta que viene instalada por defecto en Linux. Para iniciar el seguimiento de los cambios basta con iniciar git en la carpeta que contiene los scripts Python, mediante git init.

A.4 Gráficas análisis de componentes principales

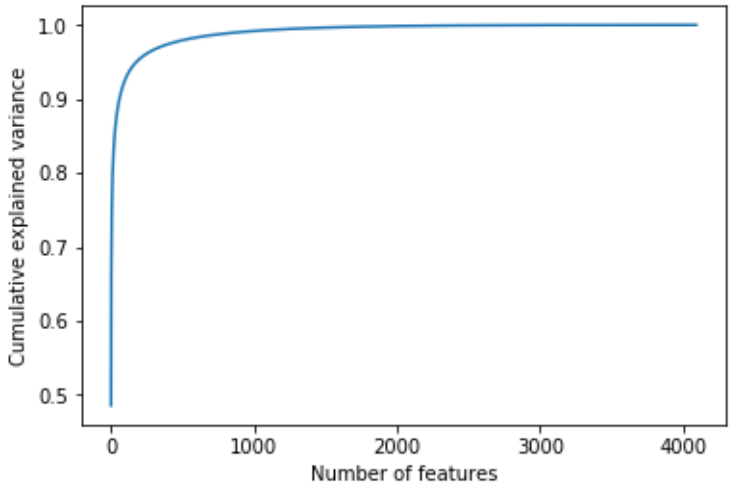

Figura 21 Análisis de componentes principales arquitectura VGG19 - 200 componentes seleccionadas

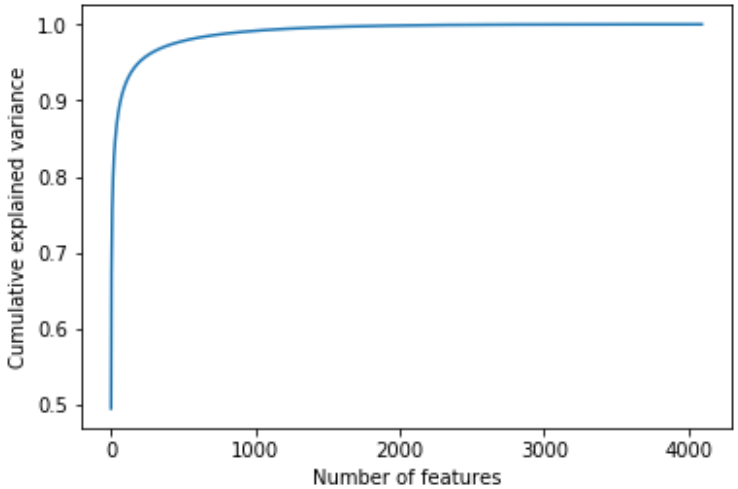

Figura 22 Análisis de componentes principales arquitectura InceptionV3 - 400 componentes seleccionadas

A.5 Matrices de confusión pipeline 1

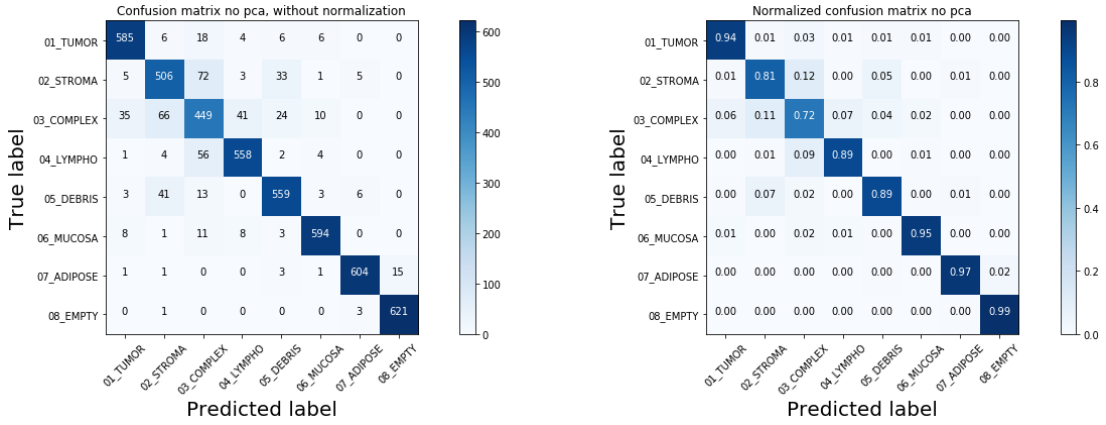

Figura 23 Arquitectura VGG16. Algoritmo SVM. Matriz de confusión de los descriptores profundos

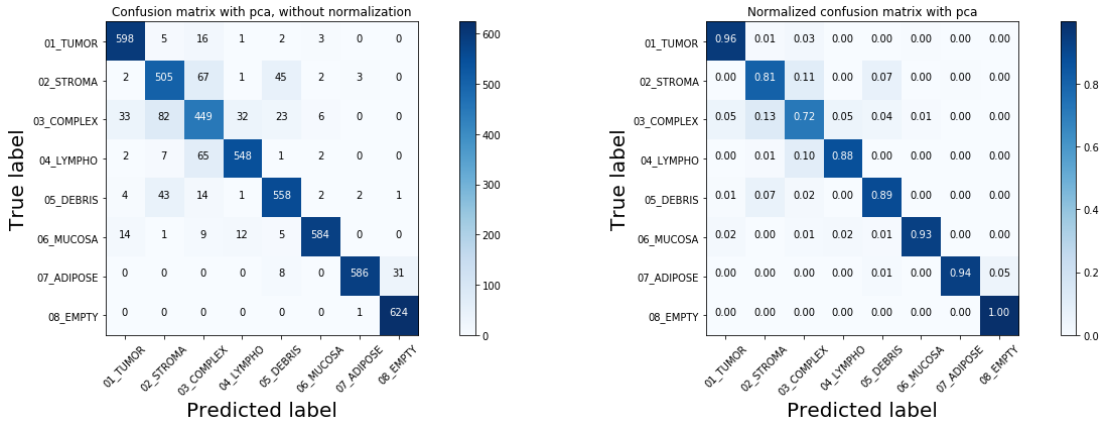

Figura 24 Arquitectura VGG16. Algoritmo SVM. Matriz de confusión de las componentes principales

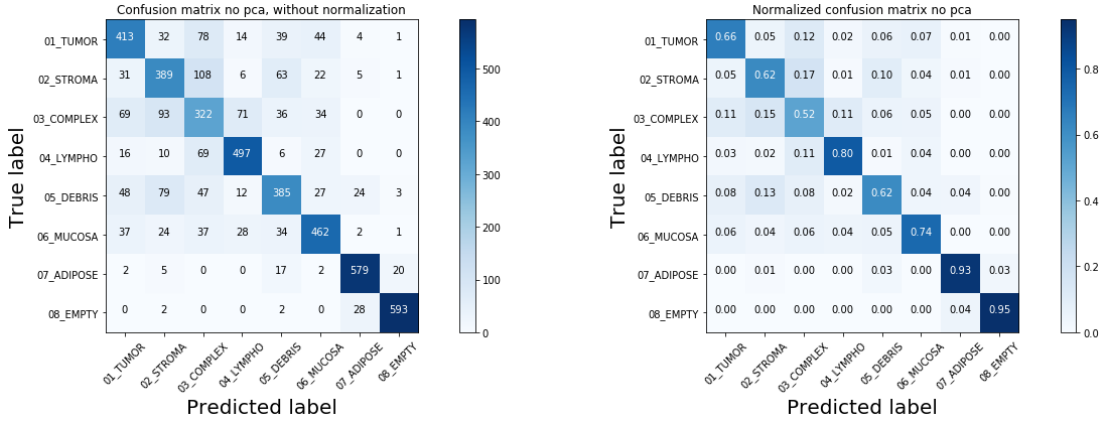

Figura 25 Arquitectura VGG16. Algoritmo DTC. Matriz de confusión de los descriptores profundos

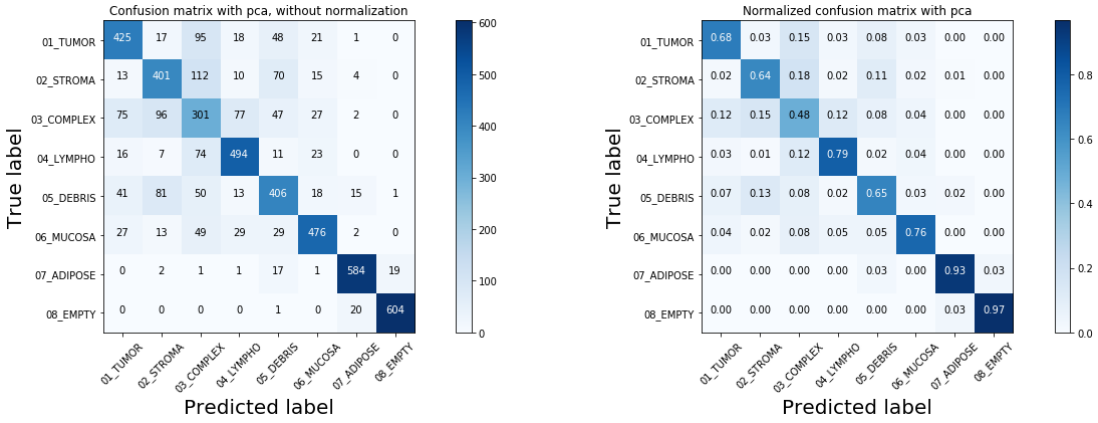

Figura 26 Arquitectura VGG16. Algoritmo DTC. Matriz de confusión de las componentes principales

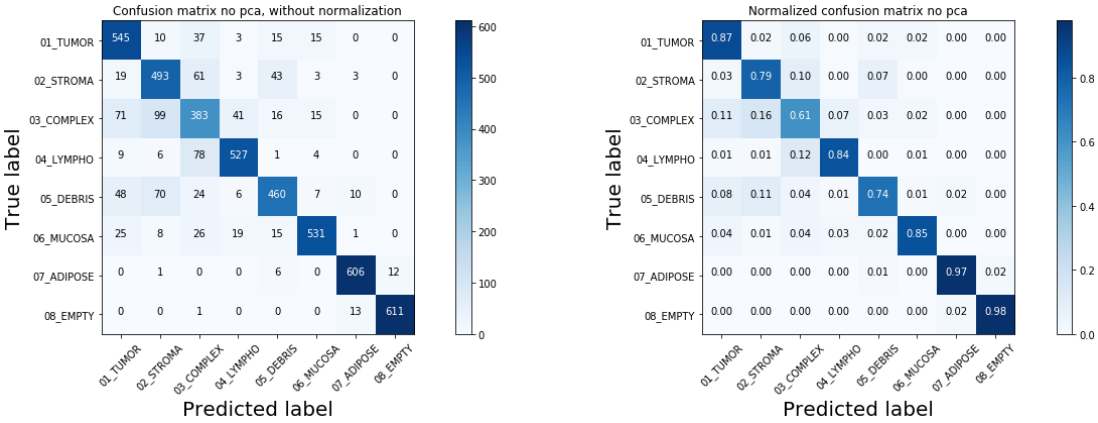

Figura 27 Arquitectura VGG16. Algoritmo RFC. Matriz de confusión de los descriptores profundos

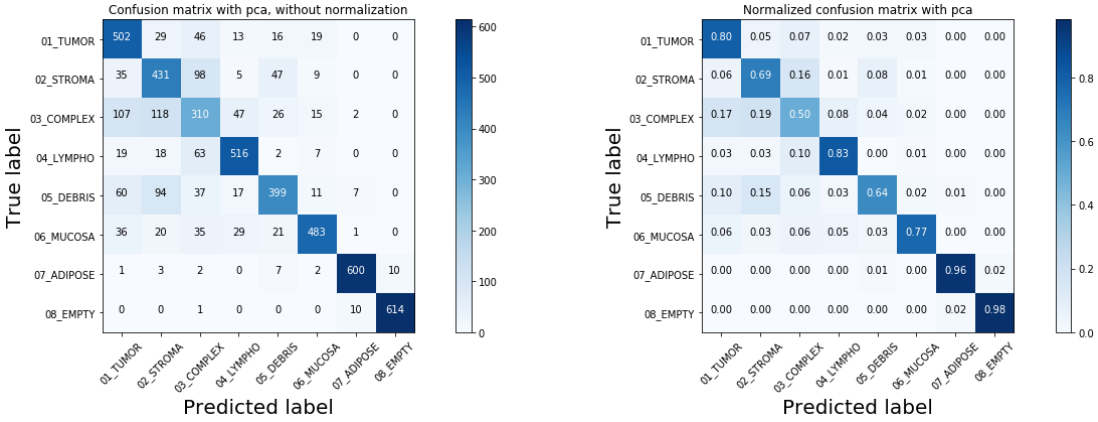

Figura 28 Arquitectura VGG16. Algoritmo RFC. Matriz de confusión de las componentes principales

h

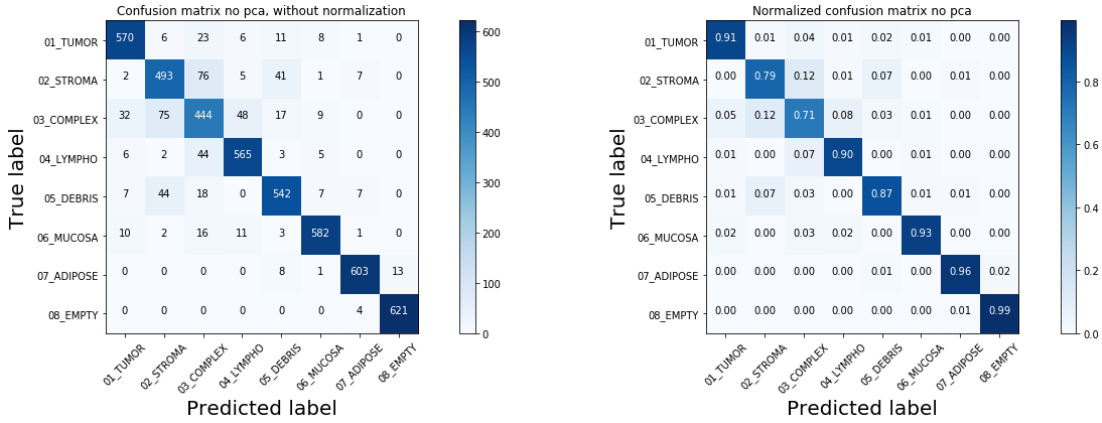

Figura 29 Arquitectura VGG19. Algoritmo SVM. Matriz de confusión de los descriptores profundos

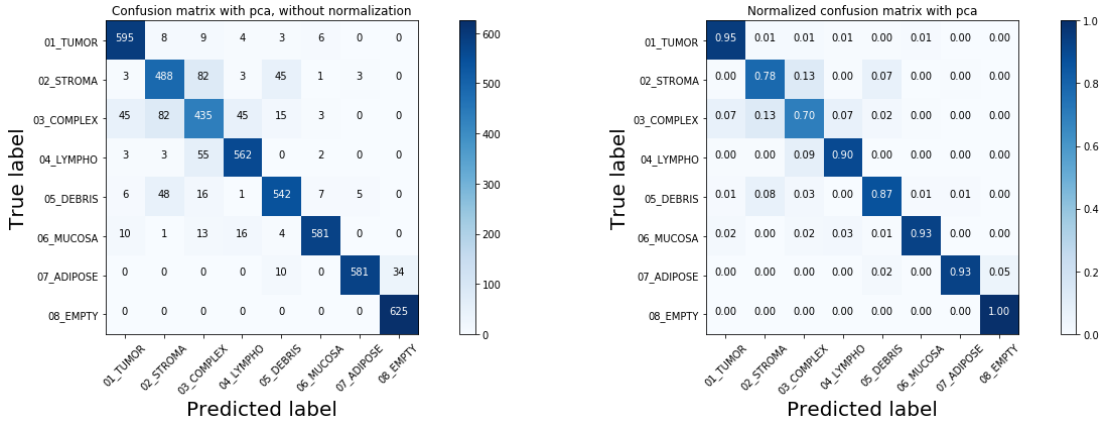

Figura 30 Arquitectura VGG19. Algoritmo SVM. Matriz de confusión de las componentes principales

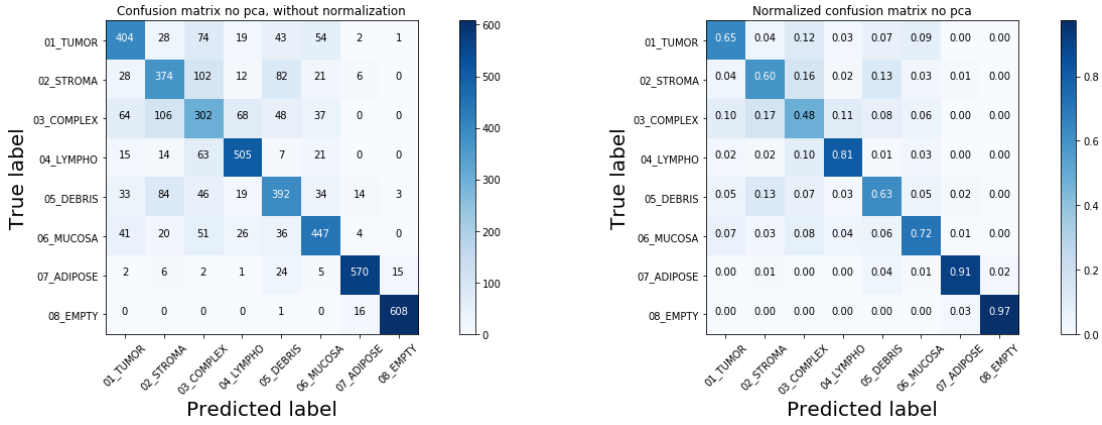

Figura 31 Arquitectura VGG19. Algoritmo DTC. Matriz de confusión de los descriptores profundos

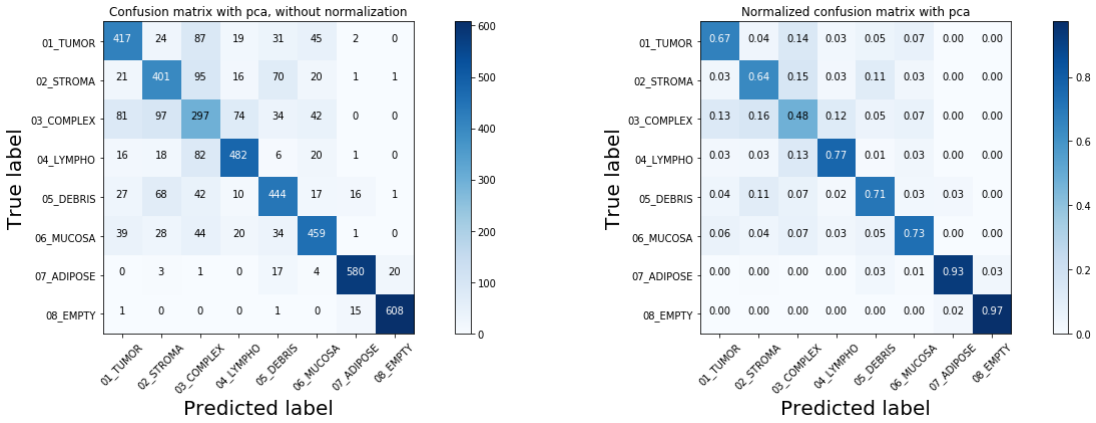

Figura 32 Arquitectura VGG19. Algoritmo DTC. Matriz de confusión de las componentes principales

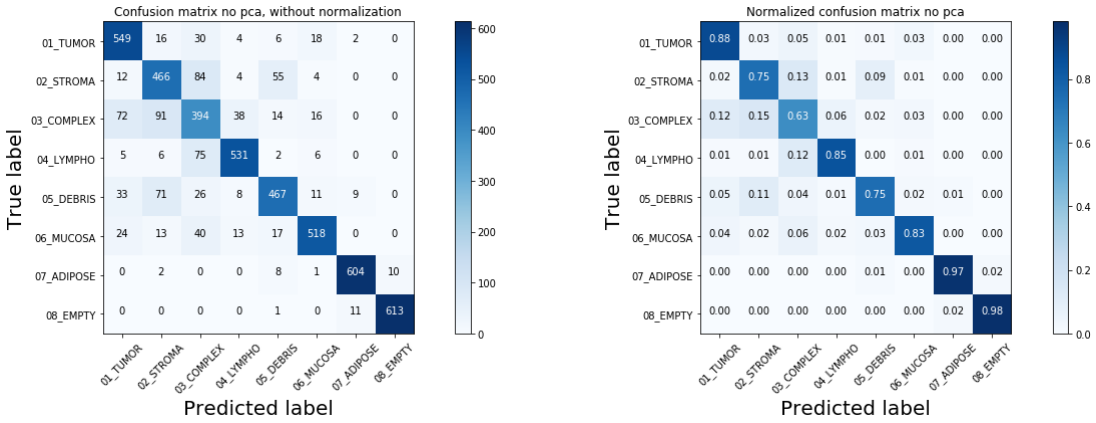

Figura 33 Arquitectura VGG19. Algoritmo RFC. Matriz de confusión de los descriptores profundos

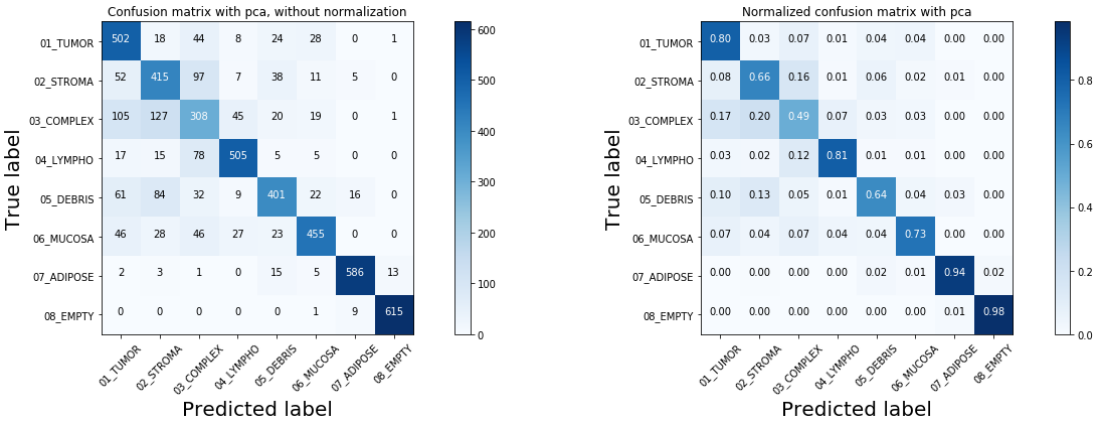

Figura 34 Arquitectura VGG19. Algoritmo RFC. Matriz de confusión de las componentes principales

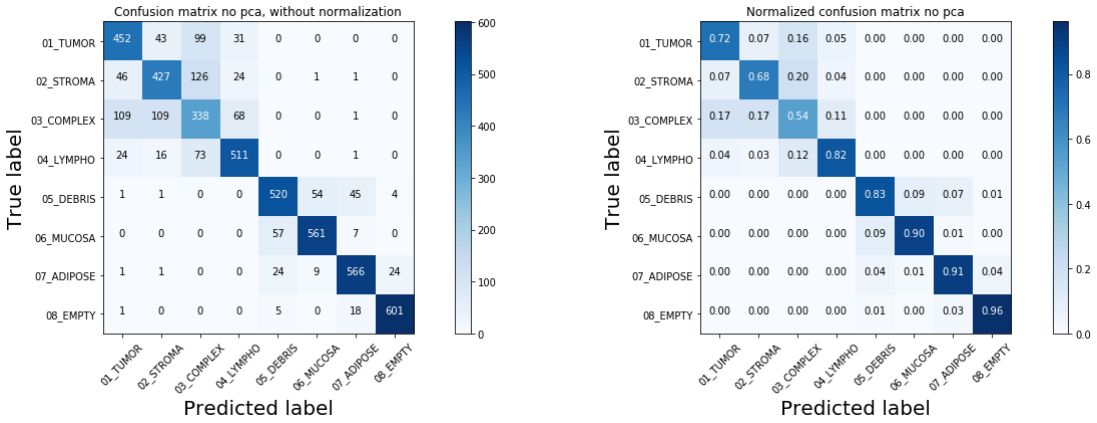

Figura 35 Arquitectura InceptionV3. Algoritmo DTC. Matriz de confusión de los descriptores profundos

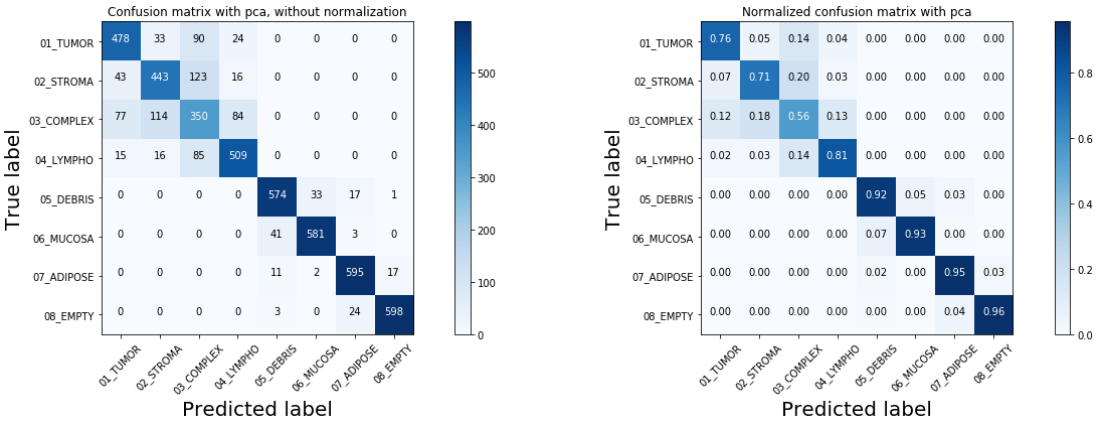

Figura 36 Arquitectura InceptionV3. Algoritmo DTC. Matriz de confusión de las componentes principales

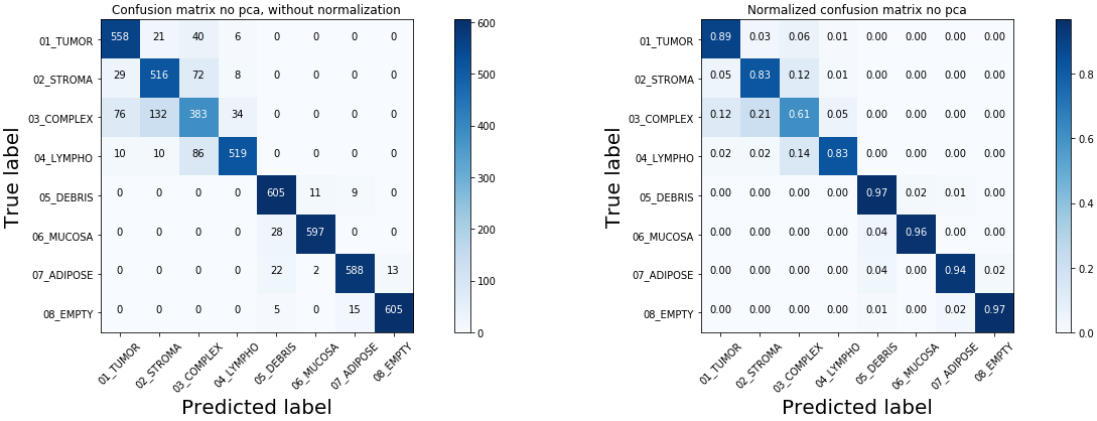

Figura 37 Arquitectura InceptionV3. Algoritmo RFC. Matriz de confusión de los descriptores profundos

k

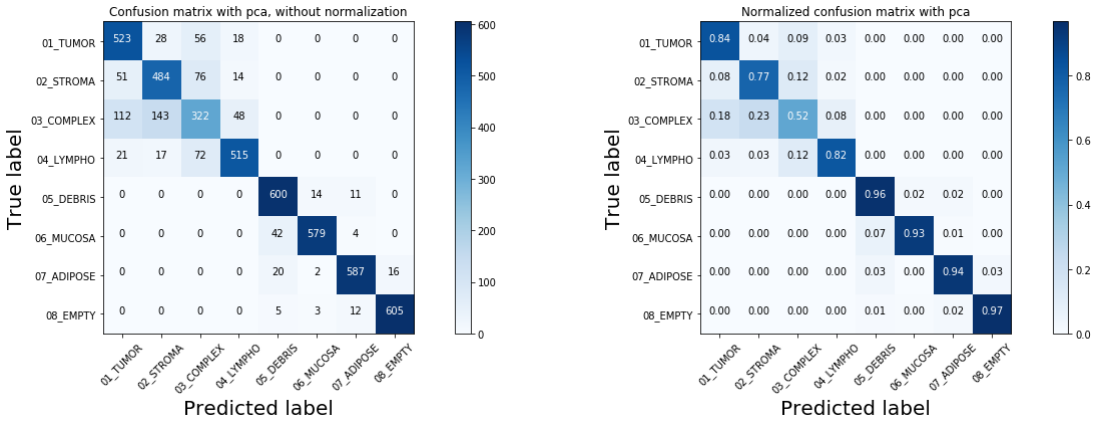

Figura 38 Arquitectura InceptionV3. Algoritmo RFC. Matriz de confusión de las componentes principales

A.6 Códigos

Los códigos asociados al este trabajo están alojados en un repositorio en Github. En esta sección se presentan los códigos de la arquitectura VGG16. El resto de los códigos pueden consultarse en el citado repositorio.

```
from keras.models import Model
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
from keras import backend as K
from keras.layers.core import Flatten, Dense, Dropout
from keras.layers.convolutional import Conv2D, MaxPooling2D, ZeroPadding2D
from keras.optimizers import SGD
from keras.models import Sequential
import numpy as np
import os
import pickle

PATH_TO_LABELS_FILE = "../data/output_dataset/img_labels"
PATH_TO_PATHS_FILE = "../data/output_dataset/img_list"
PATH_TO_OUTPUT = "../data/output_convnet"

img_labels = []
img_list = []

with open(PATH_TO_LABELS_FILE,'r') as f_img_labels:
    for line in f_img_labels:
        img_labels.append(line[:-1])

with open(PATH_TO_PATHS_FILE,'r') as f_img_list:
```

```
    for line in f_img_list:
        img_list.append("../data/"+line[:-1]) # get rid of the EOL

# https://github.com/keras-team/keras/blob/master/keras/applications/vgg16.py
model_no_relu = Sequential()
# bloque 1
model_no_relu.add(ZeroPadding2D((1,1),input_shape=(224,224,3)))
model_no_relu.add(Conv2D(64, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(64, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 2
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(128, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(128, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 3
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(256, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 4
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))
# bloque 5
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(ZeroPadding2D((1,1)))
model_no_relu.add(Conv2D(512, (3, 3), activation='relu'))
model_no_relu.add(MaxPooling2D((2,2), strides=(2,2)))

model_no_relu.add(Flatten())

# top layer of the VGG net
model_no_relu.add(Dense(4096, activation=None,name="fc1")) # activation function set to none
```

m

```
model_no_relu.add(Dropout(0.5))
model_no_relu.add(Dense(4096, activation=None,name="fc2")) # activation function set to none
model_no_relu.add(Dropout(0.5))
model_no_relu.add(Dense(1000, activation='softmax'))

model_no_relu.load_weights("keras_weights/vgg16_weights_tf_dim_ordering_tf_kernels.h5")
model_no_relu.compile(optimizer=SGD(lr=0.01), loss='categorical_crossentropy')

# deep features matrix
VGG16_dfmap_no_relU = np.empty((len(img_list),deep_features))

ti_dfmap_no_relU = time.time()

for index, img_path in enumerate(img_list):

    model_fc1 = Model(input=model_no_relu.input, output=model_no_relu.get_layer('fc1').output)

    # img preprocessing
    img = image.load_img(img_path, target_size=(224, 224), interpolation='lanczos')
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    # model predict
    x = model_fc1.predict(x)

    VGG16_dfmap_no_relU[index,:] = np.squeeze(x, axis=0)

tf_dfmap_no_relU = time.time()
tt_dfmap_no_relU = tf_dfmap_no_relU - ti_dfmap_no_relU
print()
print(time.strftime("%H:%M:%S", time.gmtime(tt_dfmap_no_relU)))
print()
print("Tamaño en memoria de la matriz de características profundas:
{0:.2f}Mb".format(getsizeof(VGG16_dfmap_no_relU)/float(1<<20)))
```

Archivo ipynb 1 VGG16_01_dfmap

```
import os
import pickle
import numpy as np
import time
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
%matplotlib inline
```

n

```
PATH_TO_LABELS_FILE = "../data/output_dataset"
LABELS = "img_labels"

PATH_TO_DF = "../data/output_convnet/VGG16"
# DF_MAP = "VGG16_dfmap_relU_pickle" # with relU activation function
DF_MAP = "VGG16_dfmap_no_relU_pickle" # without activation function

labels = []

with open(os.path.join(PATH_TO_LABELS_FILE,LABELS),'r') as f_img_labels:
    for line in f_img_labels:
        labels.append(line[:-1])

features = pickle.load(open(os.path.join(PATH_TO_DF,DF_MAP),'rb'))

features_arr = np.array(features)
labels_arr = np.array(labels)
features_arr = features_arr.T # n x m matrix!

ti_pca = time.time()

features_model_pca = PCA().fit(features_arr) # model
# plot
plt.plot(np.cumsum(features_model_pca.explained_variance_ratio_))
plt.xlabel('Number of features')
plt.ylabel('Cumulative explained variance')
plt.savefig(os.path.join(PATH_TO_DF, "VGG16_pca.png"), bbox_inches='tight') # png 70kb vs
jpg 135 kb
plt.show()

tf_pca = time.time()
tt_pca = tf_pca - ti_pca
print(time.strftime("%H:%M:%S", time.gmtime(tt_pca)))

for i in np.arange(0.7,1,0.05):
    print("With %d features we get %f of cumulative explicative variance." % \
          (np.argmax(features_model_pca.explained_variance_ratio_.cumsum() > i), i))

ti_pca = time.time()

features_model_pca = PCA(n_components=200) # fixed number of features
features_model_pca.fit(features_arr)

tf_pca = time.time()
tt_pca = tf_pca - ti_pca
```

```
print(time.strftime("%H:%M:%S", time.gmtime(tt_pca)))

# pickle.dump(features_model_pca.components_.T,\
#             open(os.path.join(PATH_TO_DF,"VGG16_dfmap_relU_pca_pickle"), 'wb'))
pickle.dump(features_model_pca.components_.T,\
            open(os.path.join(PATH_TO_DF,"VGG16_dfmap_no_relU_pca_pickle"), 'wb'))
```

Archivo ipynb 2 VGG16_02_dfmap_reduction

```
import os
import pickle
import numpy as np

PATH_TO_LABELS_FILE = "../data/output_dataset"
LABELS = "img_labels"

PATH_TO_DF = "../data/output_convnet/VGG16"

DF_MAP = "VGG16_dfmap_no_relU_pickle" # original data no relU no pca
DF_MAP_pca = "VGG16_dfmap_no_relU_pca_pickle" # original data no relU with pca

labels = []

with open(os.path.join(PATH_TO_LABELS_FILE,LABELS),'r') as f_img_labels:
    for line in f_img_labels:
        labels.append(line[:-1])

features = pickle.load(open(os.path.join(PATH_TO_DF,DF_MAP),'rb'))
features_pca = pickle.load(open(os.path.join(PATH_TO_DF,DF_MAP_pca),'rb'))

features_arr = np.array(features)
features_arr_pca = np.array(features_pca)
labels_arr = np.array(labels)

print("Deep features - deep feature map + relU")
print()
print("Structure dims: {0:d} x {1:d}".format(features_arr.shape[0], features_arr.shape[1]))
print()
print("N of deep features arrays (images): ", features_arr.shape[0])
print()
print("N of deep features: ",features_arr.shape[1])
print()
```

p

```
print(features_arr) # array with deep features
print()
print()
print("Deep features - deep feature map no relU")
print()
print("Structure dims: {0:d} x {1:d}".format(features_arr_pca.shape[0],
features_arr_pca.shape[1]))
print()
print("N of deep features arrays (images): ", features_arr_pca.shape[0])
print()
print("N of deep features: ",features_arr_pca.shape[1])
print()
print(features_arr_pca) # array with deep features
print()
print()
print("Image's labels")
print()
print("Structure: ", type(labels_arr))
print()
print("Nº of image labels (images): ", len(labels_arr))
print()
print(labels_arr) # img's label

from sklearn.model_selection import StratifiedKFold, KFold, cross_val_score,
cross_val_predict
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC
import time


cv_skf = StratifiedKFold(n_splits=10, shuffle=False, random_state=42)
# cv_kf = KFold(n_splits=5, shuffle=False, random_state=42)

# svm
SVM = LinearSVC()

ti_svm = time.time()

# data -> original dfmap no pca
scores = cross_val_score(SVM, features_arr, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred = cross_val_predict(SVM,features_arr, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat = confusion_matrix(labels_arr,y_pred)

tf_svm = time.time()
tt_svm = tf_svm - ti_svm
```

```
ti_svm_pca = time.time()

# data -> original dfmap pca
scores_pca = cross_val_score(SVM, features_arr_pca, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred_pca = cross_val_predict(SVM,features_arr_pca, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat_pca = confusion_matrix(labels_arr,y_pred_pca)

tf_svm_pca = time.time()
tt_svm_pca = tf_svm_pca - ti_svm_pca

# calc time
print(time.strftime("%H:%M:%S", time.gmtime(tt_svm))) # no pca
print(time.strftime("%H:%M:%S", time.gmtime(tt_svm_pca))) # pca

print("Feature map sin reducción de dimensiones")
print()

for i, score in enumerate(scores):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))

print()
print("media obtenida: {0:.2f}".format(scores.mean()*100))

print()
print()

print("Feature map con reducción de dimensiones")
print()

for i, score in enumerate(scores_pca):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))

print()
print("media obtenida: {0:.2f}".format(scores_pca.mean()*100))

# confusion matrix
conf_mat

conf_mat_pca

# reference http://scikit-
learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
import itertools
```

r

```
import matplotlib.pyplot as plt

def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label',size=20)
    plt.xlabel('Predicted label',size=20)

PATH_TO_IMG = "../data/input_dataset"
class_names = sorted([folder for folder in os.listdir(PATH_TO_IMG)
                      if os.path.isdir(os.path.join(PATH_TO_IMG, folder))])


fig = plt.figure(figsize=(20, 6))


fig.add_subplot(1,2,1)
plot_confusion_matrix(conf_mat, classes=class_names,
                      title='Confusion matrix no pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat, classes=class_names, normalize=True,
                      title='Normalized confusion matrix no pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_svm_no_pca.png"), bbox_inches='tight')
```

```
plt.show()

fig = plt.figure(figsize=(20, 6))


fig.add_subplot(1,2,1)
plot_confusion_matrix(conf_mat_pca, classes=class_names,
                      title='Confusion matrix with pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat_pca, classes=class_names, normalize=True,
                      title='Normalized confusion matrix with pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_svm_pca.png"), bbox_inches='tight')
plt.show()

from sklearn.tree import DecisionTreeClassifier

# DTC
DTC = DecisionTreeClassifier()

ti_dtc = time.time()

# data -> original dfmap no pca
scores = cross_val_score(DTC, features_arr, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred = cross_val_predict(DTC,features_arr, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat = confusion_matrix(labels_arr,y_pred)

tf_dtc = time.time()
tt_dtc = tf_dtc - ti_dtc


ti_dtc_pca = time.time()

# data -> original dfmap pca
scores_pca = cross_val_score(DTC, features_arr_pca, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred_pca = cross_val_predict(DTC,features_arr_pca, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat_pca = confusion_matrix(labels_arr,y_pred_pca)

tf_dtc_pca = time.time()
tt_dtc_pca = tf_dtc_pca - ti_dtc_pca

# calc time
print(time.strftime("%H:%M:%S", time.gmtime(tt_dtc))) # no pca
```

```
print(time.strftime("%H:%M:%S", time.gmtime(tt_dtc_pca))) # pca

print("Feature map sin reducción de dimensiones")
print()

for i, score in enumerate(scores):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))

print()
print("media obtenida: {0:.2f}".format(scores.mean()*100))

print()
print()

print("Feature map con reducción de dimensiones")
print()

for i, score in enumerate(scores_pca):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))

print()
print("media obtenida: {0:.2f}".format(scores_pca.mean()*100))

# confusion matrix
conf_mat

conf_mat_pca

fig = plt.figure(figsize=(20, 6))


fig.add_subplot(1,2,1)
plot_confusion_matrix(conf_mat, classes=class_names,
                      title='Confusion matrix no pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat, classes=class_names, normalize=True,
                      title='Normalized confusion matrix no pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_dtc_no_pca.png"), bbox_inches='tight')
plt.show()

fig = plt.figure(figsize=(20, 6))


fig.add_subplot(1,2,1)
```

u

```
plot_confusion_matrix(conf_mat_pca, classes=class_names,
                      title='Confusion matrix with pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat_pca, classes=class_names, normalize=True,
                      title='Normalized confusion matrix with pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_dtc_pca.png"), bbox_inches='tight')
plt.show()

from sklearn.ensemble import RandomForestClassifier

# RFC
RFC = RandomForestClassifier()

ti_rfc = time.time()

# data -> original dfmap no pca
scores = cross_val_score(RFC, features_arr, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred = cross_val_predict(RFC,features_arr, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat = confusion_matrix(labels_arr,y_pred)

tf_rfc = time.time()
tt_rfc = tf_rfc - ti_rfc


ti_rfc_pca = time.time()

# data -> original dfmap pca
scores_pca = cross_val_score(RFC, features_arr_pca, labels_arr, cv=cv_skf, n_jobs = -1)
# conf matrix
y_pred_pca = cross_val_predict(RFC,features_arr_pca, labels_arr,cv=cv_skf, n_jobs = -1)
conf_mat_pca = confusion_matrix(labels_arr,y_pred_pca)

tf_rfc_pca = time.time()
tt_rfc_pca = tf_rfc_pca - ti_rfc_pca

# calc time
print(time.strftime("%H:%M:%S", time.gmtime(tt_rfc))) # no pca
print(time.strftime("%H:%M:%S", time.gmtime(tt_rfc_pca))) # pca
print("Feature map sin reducción de dimensiones")
print()

for i, score in enumerate(scores):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))
```

```
print()
print("media obtenida: {0:.2f}".format(scores.mean()*100))

print()
print()
print("Feature map con reducción de dimensiones")
print()

for i, score in enumerate(scores_pca):
    print("acc fold nº {0:d}: {1:.2f}".format(i+1,score*100))

print()
print("media obtenida: {0:.2f}".format(scores_pca.mean()*100))

# confusion matrix
conf_mat

conf_mat_pca

fig = plt.figure(figsize=(20, 6))


fig.add_subplot(1,2,1)
plot_confusion_matrix(conf_mat, classes=class_names,
                      title='Confusion matrix no pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat, classes=class_names, normalize=True,
                      title='Normalized confusion matrix no pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_rfc_no_pca.png"), bbox_inches='tight')
plt.show()
fig = plt.figure(figsize=(20, 6))
fig.add_subplot(1,2,1)
plot_confusion_matrix(conf_mat_pca, classes=class_names,
                      title='Confusion matrix with pca, without normalization')

fig.add_subplot(1,2,2)
plot_confusion_matrix(conf_mat_pca, classes=class_names, normalize=True,
                      title='Normalized confusion matrix with pca')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_rfc_pca.png"), bbox_inches='tight')
plt.show()
```

```
from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
from keras.models import Sequential
from keras.layers import Dropout, Flatten, Dense
from keras.applications.vgg16 import preprocess_input, VGG16
from keras.optimizers import SGD
from keras.utils.np_utils import to_categorical
import time
import os
import numpy as np
import math
import matplotlib.pyplot as plt

PATH_TO_DF = "../data/output_convnet/VGG16"

img_width = 150
img_height = 150
n_epochs = 50
batch_size = 16
train_data_dir = "train"
validation_data_dir = "validation"
test_data_dir = "test"
n_train_samples = 3000
n_validation_samples = 1000
n_test_samples = 1000
n_classes = 8

# https://b/log.keras.io/building-powerful-image-classification-models-using-very-little-
data.html

ti_bn_features = time.time()

# pre-trained model without top layer
model = VGG16(include_top=False, weights='imagenet')
datagen = ImageDataGenerator(rescale=1. / 255)

# train sample
generator = datagen.flow_from_directory(
    train_data_dir,
    target_size = (img_width, img_height),
    batch_size = batch_size,
    class_mode = None,
    shuffle = False,
    interpolation = 'lanczos')
```

```
max_queue_size_train = int(math.ceil(n_train_samples / batch_size))
bnfeatures_train = model.predict_generator(generator, max_queue_size_train)
np.save('../data/output_convnet/VGG16/VGG16_bnfeatures_train_aux.npy', bnfeatures_train)

# ref attribute classes --> https://keras.io/preprocessing/image/
train_labels = generator.classes # the key attribute
train_labels = to_categorical(train_labels, num_classes=n_classes) # the key function

# validation sample
generator = datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode=None,
    shuffle=False,
    interpolation = 'lanczos')

max_queue_size_val = int(math.ceil(n_validation_samples / batch_size))
bnfeatures_val = model.predict_generator(generator, max_queue_size_val)
np.save('../data/output_convnet/VGG16/VGG16_bnfeatures_val_aux.npy', bnfeatures_val)

val_labels = generator.classes # the key attribute
val_labels = to_categorical(val_labels, num_classes=n_classes) # the key function

tf_bn_features = time.time()
tt_bn_features = tf_bn_features - ti_bn_features

print(time.strftime("%H:%M:%S", time.gmtime(tt_bn_features)))

# training top layer
ti_bn_train = time.time()

train_data = np.load('../data/output_convnet/VGG16/VGG16_bnfeatures_train_aux.npy')
val_data = np.load('../data/output_convnet/VGG16/VGG16_bnfeatures_val_aux.npy')

# top model, could be with a diff dense, optimizer, momentum ->
https://keras.io/optimizers/
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))
```

y

```
model.compile(optimizer=SGD(lr=0.001),
              loss='categorical_crossentropy', metrics=['accuracy'])


historical_data = model.fit(train_data, train_labels,
                    epochs=n_epochs,
                    batch_size=batch_size,
                    validation_data=(val_data, val_labels))

# h5py
model.save_weights('../data/output_convnet/VGG16/VGG16_bn_model_aux.h5')

tf_bn_train = time.time()
tt_bn_train = tf_bn_train - ti_bn_train

print(time.strftime("%H:%M:%S", time.gmtime(tt_bn_train)))

(loss, acc) = model.evaluate(val_data, val_labels, batch_size=batch_size, verbose=0)
print("acc: {0:.2f}% - loss: {1:f}".format(acc * 100, loss))

train_acc = historical_data.history['acc']
train_loss = historical_data.history['loss']
val_acc = historical_data.history['val_acc'] # validation
val_loss = historical_data.history['val_loss'] # validation
range_epochs = range(n_epochs)

fig = plt.figure(figsize=(20, 6))

fig.add_subplot(1,2,1)
plt.plot(range_epochs, train_acc, 'bo', label='Training acc')
plt.plot(range_epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
fig.add_subplot(1,2,2)
plt.plot(range_epochs, train_loss, 'bo', label='Training loss')
plt.plot(range_epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_bn_acc_loss.png"), bbox_inches='tight')
plt.show()
```

Archivo ipynb 4 VGG16_04_bottleneck

```
from keras import applications
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
from keras.models import Sequential, Model
from keras.layers import Dropout, Flatten, Dense, Input

top_model_weights_path = '../data/output_convnet/VGG16/VGG16_bn_model_aux.h5'

img_width = 150
img_height = 150

n_epochs = 50
batch_size = 5 # reduce to 5 in order to be able to compute the calcs

train_data_dir = "train"
validation_data_dir = "validation"
test_data_dir = "test"

n_train_samples = 3000
n_validation_samples = 1000
n_test_samples = 1000
n_classes = 8

# base model
input_tensor = Input(shape=(img_width,img_height,3)) # another way to shape the input
base_model = applications.VGG16(weights='imagenet', include_top=False,
input_tensor=input_tensor)

# top model
top_model = Sequential()
top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
top_model.add(Dropout(0.5))
top_model.add(Dense(8, activation='softmax'))

# base model has its weights, now we load the weights on the top layer
top_model.load_weights("../data/output_convnet/VGG16/VGG16_bn_model_aux.h5")

# we join base and top it has to be updated to api2
model_total = Model(input= base_model.input, output= top_model(base_model.output))

# sequential is the top layer a complex one
for i, layer in enumerate(model_total.layers):
    print (i, layer.name, layer.output_shape)
```

aa

```
for i, layer in enumerate(model_total.layers):
    if layer.trainable:
        print("layer {0:d}, {1:s} is trainable".format(i, layer.name))
    else:
        print("layer {0:d}, {1:s} is freezed".format(i, layer.name))

# freezing layers implies they will not update their weights over the training
for layer in model_total.layers[:15]:
    layer.trainable = False

# check for updates
for i, layer in enumerate(model_total.layers):
    if layer.trainable:
        print("layer {0:d}, {1:s} is trainable".format(i, layer.name))
    else:
        print("layer {0:d}, {1:s} is freezed".format(i, layer.name))

model_total.compile(optimizer=SGD(lr=1e-4, momentum=0.9),
              loss='categorical_crossentropy', metrics=['accuracy'])

# diff with bottleneck, we have to use data augmentation here
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      vertical_flip = True,
      fill_mode='nearest')

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width,img_height),
    batch_size=batch_size,
    class_mode='categorical')

val_datagen = ImageDataGenerator(rescale=1. / 255) # not in the val data

val_generator = val_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical')
```

bb

```
import time
# fine-tune the model

ti_ftuning = time.time()


historical_data = model_total.fit_generator(
    train_generator,
    samples_per_epoch=n_train_samples,
    epochs=n_epochs,
    verbose = 1,
    validation_data=val_generator,
    validation_steps=n_validation_samples)

tf_ftuning = time.time()
tt_ftuning = tf_ftuning - ti_ftuning

print(time.strftime("%H:%M:%S", time.gmtime(tt_ftuning)))

model_total.save_weights('../data/output_convnet/VGG16/VGG16_ft_testing_model_aux.h5')

import math
steps = int(math.ceil(n_validation_samples / batch_size))

(loss, acc) = model_total.evaluate_generator(val_generator, steps=steps)

print("acc: {0:.2f}% - loss: {1:f}".format(acc * 100, loss))

import numpy as np

train_acc = historical_data.history['acc']
train_loss = historical_data.history['loss']

val_acc = historical_data.history['val_acc'] # validation
val_loss = historical_data.history['val_loss'] # validation

print("train acc mean: {0:.2f} * train loss mean:
{1:.2f}".format(np.average(train_acc),np.average(train_loss)))
print("validation acc mean: {0:.2f} * validation loss mean:
{1:.2f}".format(np.average(val_acc),np.average(val_loss)))

# historical_data.history acc and loss data over the epochs (train and validation)
import matplotlib.pyplot as plt
import os

PATH_TO_DF = "../data/output_convnet/VGG16"
```

```
range_epochs = range(n_epochs)

fig = plt.figure(figsize=(20, 6))

fig.add_subplot(1,2,1)
plt.plot(range_epochs, train_acc, 'bo', label='Training acc')
plt.plot(range_epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
fig.add_subplot(1,2,2)
plt.plot(range_epochs, train_loss, 'bo', label='Training loss')
plt.plot(range_epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_ft_acc_loss_aux.png"), bbox_inches='tight')
# plt.savefig(os.path.join(PATH_TO_DF, "VGG16_ft_acc_loss.png"), bbox_inches='tight')
plt.show()

test_convnet = ImageDataGenerator(rescale=1. / 255)

test_generator = test_convnet.flow_from_directory(
    test_data_dir,
    target_size=(img_width,img_height),
    batch_size=batch_size,
    shuffle =False,
    class_mode='categorical')

steps = int(math.ceil(n_test_samples / batch_size))

(loss, acc) = model_total.evaluate_generator(test_generator, steps=steps)

print("acc: {0:.2f}% - loss: {1:f}".format(acc * 100, loss))

predictions = model_total.predict_generator(test_generator, steps = steps)

from sklearn.metrics import confusion_matrix

PATH_TO_IMG = "../data/input_dataset"

prediction_list = []
real_label_list = []
```

dd

```
cat_dict = test_generator.class_indices # the key attribute
inverse_coding = {value: key for key, value in cat_dict.items()} # dict of categories

for label in test_generator.classes:
    real_label_list.append(inverse_coding[label])

for prediction in predictions:
    prediction_list.append(inverse_coding[np.argmax(prediction)])

# in order to get the confusion matrix
class_names = sorted([folder for folder in os.listdir(PATH_TO_IMG)
                  if os.path.isdir(os.path.join(PATH_TO_IMG, folder))])
prediction_list_arr = np.array(prediction_list)
real_label_list_arr = np.array(real_label_list)

VGG16_cm_ft = confusion_matrix(real_label_list_arr,prediction_list_arr)

# reference http://scikit-
learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
import itertools
import matplotlib.pyplot as plt

def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
```

ee

```
    plt.ylabel('True label',size=20)
    plt.xlabel('Predicted label',size=20)

fig = plt.figure(figsize=(20, 6))
fig.add_subplot(1,2,1)
plot_confusion_matrix(VGG16_cm_ft, classes=class_names,
                      title='Confusion matrix, without normalization')
fig.add_subplot(1,2,2)
plot_confusion_matrix(VGG16_cm_ft, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.savefig(os.path.join(PATH_TO_DF, "VGG16_ft_test.png"), bbox_inches='tight')
plt.show()
```

Archivo ipynb 5 VGG16_05_fine_tuning