Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento Ingeniería Matemática

Laboratorio de Ciencia de Datos

Tarea 2

Redes convolucionales profundas e interpretabilidad

Integrantes: Gabriel Ortega
Pablo Paredes
Kurt Walsen
Profesor: Nicolás Caro
Auxiliares: Rodrigo Lara
Fecha: 17 de julio de 2020
Santiago, Chile

1. Contexto

A continuación se presenta la segunda entrega del curso *MA6202 Laboratorio de Ciencia de Datos*. En esta entrega se busca llevar a cabo un problema de ciencia de datos implementando técnicas y metodologías aprendidas en el curso, en particular la carga y transformación de datos, la implementación y entrenamiento de una red neuronal profunda para un problema de clasificación, junto con la implementación del metodo LIME(Local interpretable Model-Agnostic Explanations) para darle interpretabilidad a los resultados obtenidos por el modelo implementado.

1.1. Objetivo

Se busca implementar en `Pytorch` una red neuronal profunda para el problema de clasificación de imágenes de rayos X sobre neumonía.

1.2. Data

El conjunto de datos consiste en dos carpetas, una para entrenamiento y otra para test, que cointienen imágenes de radiografías torácicas, separadas en dos carpetas dependiendo si la radiografía corresponde o no a un paciente con neumonía. El conjunto de entrenamiento consta de al rededor de 5000 imágenes y el de test cerca de 600.

2. Carga y transformación de datos

El objetivo de está sección es cargar los datos de manera ótima, usando herramientas de las librerias `Torch` y `Pytorch`, que están diseñadas para el uso de redes neuronales y tratamiento de imágenes en redes neuronales respectivamente. Para esto, comparamos el rendimiento de `Pytorch` frente a `Skimage`, otra herramienta de procesamiento de imágenes en el proceso de carga y transformación de imágenes. Luego implementamos la clase `ReplicarMuestreoDePrueba` para replicar el la carga de un batch de datos en el conjunto de validación, la distribución que tienen los datos en el conjunto de test. Por último, implementamos un `data_loader` para samplear muestras de los conjuntos de entrenamiento, validación, y test, que será usado en el entrenamiento de la red de la próxima sección.

2.1. Funciones de carga de imágenes y transformaciones

En esta sección se cargan las imágenes usando la función `DatasetFolder`, que toma como argumentos: la dirección de una carpeta, una función que carga una imagen dada la dirección de ésta en la carpeta, y una lista de funciones que se aplican a la imagen que se carga con la función anterior. Para las funciones de carga dos librerias distintas, `PIL` y `Skimage`. En ambas funciones fue implementado el método de carga de la libreria respectiva, con una concatenación de canales en caso de que fuera 1 sólo canal, así, ambas funciones entregan imágenes en formatos distintos (formato `PIL` y `Numpy array`). Las funciones fueron implementadas para transformar la imagen directamente del formato en que las entrega la función de carga anterior. Para el caso de la imagen `PIL`, se usaron transformaciones de la libreria `Torchvision`(que implementan los valores aleatorios internamente), y para el caso de la imagen en formato `array`, se usaron funciones de la libreria `Skimage` junto con funciones de `Numpy` para agregar la parte aleatoria de las funciones.

Función	Tiempo total (seg)	Llamados a funciones internas
pil_loader	0.001	175
skimage_loader	0.006	684
Transform_Resize('PyTorch')	0.005	50
Transform_Resize('skimage')	0.020	541
Transform_Flip('PyTorch')	0.000	17
Transform_Flip('skimage')	0.000	12
Transform_Rotation('PyTorch')	0.001	63
Transform_Rotation('skimage')	0.014	154
Transform_Scale('PyTorch')	0.001	6
Transform_Scale('skimage')	0.001	12
Transform_Mult('PyTorch')	0.002	6
Transform_Mult('skimage')	0.005	5

Tabla 2.1: Comparación de rendimientos de distintas funciones de carga y manejo de imágenes.

Luego de esto se hace un análisis de rendimiento en las funciones de ambas librerias, cuyos resultados son presentados en la tabla 2.1.

En la tabla se puede notar que en todas las funciones, la función implementada para la imagen en formato `PIL` es más rápida que la función implementada para la imagen en formato `Numpy array`. Así, en este punto, se decide usar las funciones de la libreria `Torchvision` para las siguientes partes. También se crea la función `loader` para cargar los distintos datasets con usando la función `DatasetFolder` y las transformaciones elegidas anteriormente.

2.2. Visualización de distribuciones

En esta parte se muestra el gráfico 2.1, que muestra la distribución de clases en los datasets de entrenamiento y test.

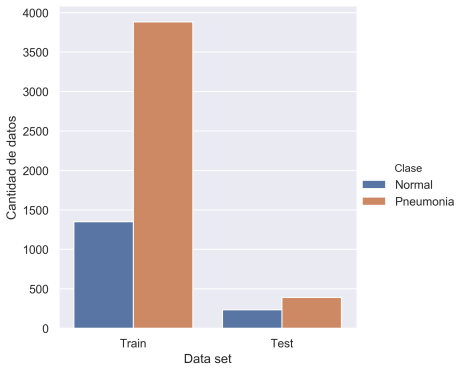

Figura 2.1: Distribuciones en la cantidad de datos en los distintos datasets

2.3. Implementación de la clase ReplicarMuestreoDePrueba

En esta sección se separa el conjunto de general de entrenamiento en uno de entrenamiento y uno de validación, y, por medio de la clase `ReplicarMuestreoDePrueba` se busca que la carga de datos en el `DataLoader` final tenga la misma distribución en el set de validación y el set de test. En concreto, la clase calcula el porcentaje de imagenes normales con respecto al total de imagenes (en el conjunto de validación). Si aquel porcentaje es menor que el del conjunto de test, se repiten los indices de tantas imagenes normales (aleatoriamente) hasta sobrepasar o igualar el porcentaje del conjunto de test, en el otro caso, el procedimiento es agregar indices de las imagenes de pneumonia. Finalmente la clase es implementada con metodo iter que entrega un iterador sobre los indices antes cambiados.

2.4. Implementación del Dataloader

En esta parte se usa la función `DataLoader` de la clase `torch.utils.data`. Con ella, y entregandole el parámetro sampler adecuado: un objeto SubsetRandomSampler o RandomSampler en el caso de los datasets de entrenamiento y test respectivamente; o en el caso del conjunto de validación la clase ReplicarMuestreoDePrueba con el iterador integrado.

3. Redes convolucionales profundas

El objetivo de esta sección es construir una red convolucional profunda para el problema de diagnosticar neumonia en pacientes a través de su radiografía. Esta red será implementada en `Pytorch`.

3.1. Depthwise Separable Convolution

Primero, se construye una clase llamada `DWSepConv2d`, que hereda de la clase `nn.Module`, la cual será una capa de la red consistente de dos capas de convolución:

- La primera es la capa Depthwise, la cual no cambia el número de canales del input, y tiene *padding*, y tamaño de filtro de acuerdo a los parámetros que recibe en su input.
- La segunda es la capa Pointwise, la cual modifica el número de canales de acuerdo a lo puesto en el input, pero con tamaño de filtro igual a 1 y sin *padding*. Además, a esta capa se le aplica la función de activación ReLU, mientras a la primera no se le aplica.

3.2. Construcción de la red

Se construye la red convolucional profunda mediante la clase `VGG16DWSep`, la cual hereda de `nn.Module`. En esta red, se consideran capas de convolución en dos dimensiones, además de la capa construida en la sección anterior, sumado a capas de *pooling*, *flattening*, *linear* y *dropouts*. Todas estas capas se construyen de acuerdo a los parámetros dados en el enunciado. Además, se aplica la función de activación ReLU, únicamente, a las capas de convolución.

3.3. Transferencia de pesos

En esta parte, se desea transferir los pesos de las primeras dos capas de la red preentrenada `VGG16` a las primeras dos capas de la red construida en la sección anterior.

Para esto, inicializamos una instancia de la red `VGG16` y extraemos los pesos de las primeras dos capas asignándolos a las variables `pesos1` y `pesos2`. Luego, creamos la variable `model` e inicializamos una instancia de la red `VGG16DWSep`, para después, asignarle los pesos extraídos de la red anterior a sus dos primeras capas, además de asignarle `False` al atributo `require_grad` para mantener los pesos constantes durante el entrenamiento.

3.4. EarlyStopping

Se crea la clase `EarlyStopping`, la cual servirá para detener el entrenamiento del modelo cuando ya no esté mejorando según los parámetros dados por el usuario.

Además de los atributos requeridos, se añaden `best` y `stopstep`, donde el primero guarda la mejor métrica de validación mientras entrena el modelo, para así poder compararla con la métrica actual y poder decidir si es mejor o peor, de acuerdo a la tolerancia especificada; mientras el segundo es un contador que se reiniciará cada vez que la métrica actual sea mejor que la guardada en `best`, y sumará uno cada vez que la métrica sea peor. Así, cuando `stopstep` llegue a la paciencia indicada, el programa sabrá que hay que parar.

El método `mejor` retornará `True` si es que la métrica de validación actual no es peor que la actual (de acuerdo a la tolerancia y modo indicado, además de si es comparación relativa o absoluta). Si no es peor, además de retornar `False`, reinicia el atributo `stopstep`, mientras que, en caso contrario, le suma uno al contador.

Luego, el método `deberia_parar` retorna `True` si el entrenamiento debería parar, de acuerdo a la paciencia indicada; y `False` en caso contrario.

3.5. Entrenamiento de la red

Se ejecuta el entrenamiento de la red `VGG16DWSep` creada previamente. Para esto, optimizaremos el tiempo de ejecución moviendo el modelo inicializado a la GPU. En este caso, se usó *Google Colaboratory* para poder acceder a la GPU sin problemas.

Se usa una instancia de `EarlyStopping` con los parámetros por defecto, para detener si es necesario. Además, se usará *cross entropy* como función de costo y *Adam* como algoritmo de optimización. La red entrenará por 20 épocas.

Los resultados obtenidos en términos de *accuracy* y *f1-score* son:

Época	*accuracy*	*f1-score*
1	74.684 %	0.850
5	89.636 %	0.929
10	90.902 %	0.935
15	93.670 %	0.953
20	92.801 %	0.945

Notamos que se realizaron las 20 épocas, por lo que el `EarlyStopping` no detuvo el entrenamiento. Esto se explica ya que las mejoras se hacían en un intervalo de menos de 5 épocas (que es la paciencia por defecto), por lo que no se tuvo la necesidad de parar, ya que la red estuvo siempre mejorando. Sin embargo, en este caso, observamos que de la época 15 no mejoró, entonces si hubiera tenido un número mayor de épocas, el entrenamiento hubiera parado en la 20 de todas formas.

4. Interpretabilidad(LIME)

Se procede a darle interpretabilidad a los resultados obtenidos por el modelo previamente desarrollado. Para ello se empleó LIME como un modelo auxiliar, que consiste en aproximar de manera local el comportamiento de un modelo de predicción(que se asume de caja negra) mediante la utilización de un modelo interpretable. Para esta entrega se utilizó regresión logística como familia de explicaciones, esto es, cada explicación $g \in G$ es de la forma $g(x') = \sigma(\omega_g \cdot x')$, donde x' es una representación interpretable. Se utilizó como funcion de fidelidad como sigue:

$$\mathcal{L}(f, g, \pi_x) = \sum_{z,z'} \pi_x(z)(f(z)\log(g(z')) + (1 - f(z))\log(1 - g(z'))$$

esto es, la funcion de verosimilitud asociada a la regresión logística bajo una ponderación local por $\pi_x(z') = exp(\frac{-d(x',z')^2}{\sigma^2})$, que corresponde a un kernel exponencial definido por una métrica de similitud d.

La construcción del método y algunos análisis aplicados sobre una imagen de prueba a modo de entender su funcionamiento pueden encontrarse en la sección de Anexos.

Se realiza el procedimiento de interpretación utilizando la red preentrenada `VGG16` de `torchvision.models`, la imagen corresponde a una de las radiografias disponibles en el conjunto de entrenamiento(ver Figura (4.1)). Notamos que en base a esta red no será posible clasificar de forma binaria (como lo hace `VGG16DWSep`), por lo cual trataremos el problema mediante la función `softmax` para realizar predicciones.

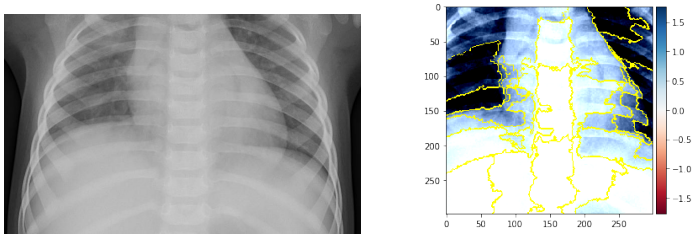

Figura 4.1: A la izquierda, radiografía torácica de un paciente con neumonía, sin procesar. A la derecha, imagen procesada y segmentada.

Se aplicó el esquema LIME siguiendo la siguiente metodología:

1. Se reescaló la imagen mediante el esquema de transformaciones detallado en Anexos 1. Luego, usando la red `VGG16` y la función `softmax` se realizaron predicciones y se obtuvieron las probabilidades de pertenencia a cada clase, las 5 de probabilidad más alta se encuentran en la Tabla (4.2) . La predicción obtenida corresponde a la clase 'Shower_curtain' con probabilidad de 0.159, bastante baja. Le siguen las clases 'window_shade', 'mosquito_net', 'fountain' y 'lampshade' con probabilidades 0.159,0.061,0.030,0.029 y 0.024 respectivamente.

2. Se utiliza la función `slic` del módulo `skimage.segmentation` para obtener una segmentación de la imagen (ver Figura (4.1)). La segmentación nos da un total de 22 superpixeles.

3. Mediante la función `bernoulli.rvs` del módulo `scipy.stats` se geneararon 1000 arreglos binarios de largo igual al numero de superpixeles obtenidos de la segmentación anterior, donde cada entrada corresponde a una realización de una Bernoulli(0.5). Cada arreglo indica una configuracion de presencia o ausencia de superpixeles dentro de la imagen de prueba. Con este arreglo se procede a generar versiones perturbadas de la imagen de prueba, sobre las cuales se realizan predicciones bajo la red `VGG16`, y se construye un arreglo binario $\mathbf{y}$ de largo igual al número total de perturbaciones, donde catalogará con valor 1 aquellas predicciones que coincidan con la predicción hecha a la imagen de prueba, y valor 0 las que no. De las 1000 perturbaciones solo 10 coincidieron en predicción con la de control.

4. Considerando la imagen original como un arreglo con todas sus entradas iguales a 1, se procede a calcular la distancia coseno entre las perturbaciones generadas y éste arreglo. Con el vector de distancias obtenido, utilizando un valor de $\sigma = 0.25$ se calcula π_x como un kernel exponencial sobre el vector de distancias. Se obtiene entonces un arreglo de largo igual al número de perturbaciones, donde cada entrada corresponde al peso que tiene una perturbación.

5. Se construye el conjunto de entrenamiento para ver los pesos que cada superpixel aporta en la predicción de la clase de la imagen de control. Consideramos como conjunto de entrenamiento todas las perturbaciones como variable explicativa, y las predicciones respectivas $\mathbf{y}$ como variable de respuesta. Luego se entrena un modelo de regresión logística mediante una instancia de la clase `LogisticRegression`, obteniendo los pesos.

6. Se consideran perturbaciones que consistan en los superpixeles con los 10 pesos más altos y 10 pesos más bajos respectivamente(ver Figura (4.2)) y se utiliza la red `VGG16` para hacer predicciones. Notamos que fijando los 10 superpixeles con mayor peso no logra obtenerse la misma predicción que la imagen de prueba, es decir, 'shower_curtain', sin embargo, se obitene la quinta con mayor probabilidad(aunque de todas maneras baja), es decir, 'lampshade'. Mientras que para los superpixeles de menor peso evidentemente tampoco logra obtenerse la misma predicción que la imagen de prueba. Esto puede ser debido a la proporción elegida a la hora de tomar los superpixeles más relevantes con respecto al total, la cual corresponde a casi la mitad. Tomando una cantidad menor puede ser posible notar diferencias significativas, pero debido al bajo peso que ponderan los superpixeles es dificil determinar con seguridad tal grado de importancia.

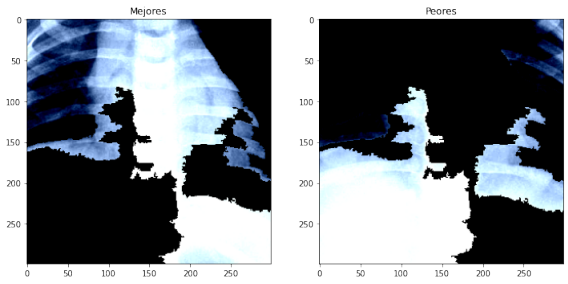

Figura 4.2: A la izquierda, imagen perturbada según los 10 superpixeles de mayor peso. A la derecha, imagen perturbada según los 10 superpixeles de menor peso.

Con esto podemos notar que el uso de la red `VGG16` no es suficiente para la correcta clasificación de casos que naturalmente son binarios (tener o no neumonía), además podemos comprobar que tampoco fue suficiente para la interpretación de la clasificacón. Por lo tanto, las modificaciones en la red `VGG16DWsep` son importantes para generar el buen clasificador y tener una métrica mas fiable a la hora de ver cuáles son los resultados que me entrega el modelo, y de esta forma poder asociar de manera más precisa una interpretación a las clasificaciones que el modelo realiza.

5. Conclusiones

En la implementación de la red neuronal con la heurística del `EarlyStopping`, la precisión de la mejor época del modelo fue de 93.67 % (sobre 75 %), mientras el mejor *f1-score* fue de 0.953 (sobre 0.8), lo cual es relativamente bueno, ya que puede ser de utilidad para optimizar el tiempo de diagnóstico de la enfermedad y abarcar a muchos más pacientes en un intervalo de tiempo más pequeño.

Además, los valores por defecto para los parámetros del `EarlyStopping` fueron adecuados para este entrenamiento, ya que el programa se computó una vez más, en el cual el programa se detuvo en la época 14, ya que no mejoró durante 5 épocas seguidas, con una precisión máxima de 93.829 %. Entonces, esto nos permite deducir que, dada la arquitectura de la red, la mejor precisión que puede obtener está entre 93 y 95 %.

En esta parte se presentó un nuevo desafío, el cual fue usar *Google Colaboratory* como nueva plataforma para ejecutar el programa usando la GPU como motor. La mayor dificultad fue, principalmente, al momento de cargar los datos y saber cómo y dónde guardar los archivos requeridos.

En la implementación de LIME notamos su eficiacia para poder darle interpretabilidad a los resultados obtenidos luego de un proceso de clasificación, donde hace posible tener una idea o visualización de qué componentes interpretables de un dato son las que más influyen en cómo el modelo lo clasifica. Se estudió además como esta interpretación puede variar según el método de segmentación elegido, el cual determina a su vez las componentes que serán una representación interpretable del dato, donde dependiendo del método la explicación de la clasificación puede variar.

Sin embargo, notamos que el método implementado es sensible al modelo, cambiar de modelo sugiere también modificaciones en el proceso de obtencion de representaciones. Se debe estudiar a cabalidad el funcionamiento de un modelo para poder realizar este esquema de validación de resultados, puesto que en caso de no hacerlo pueden llegarse a resultados erroneos y hasta contradictorios.

Durante el desarrollo de esta entrega se presentaron complicaciones, entre las cuales se encuentran la falta de tiempo para poder corroborar los resultados obtenidos y poder trasladar éstos a las secciones posteriores, la magnitud del problema y la falta de coordinación fueron relevantes a la hora de juntar todos los resultados y obtener resultados coherentes y significativos.

Como futuros pasos se espera ser capaces de profundizar y refinar las técnicas empleadas en esta entrega. El conocimiento y manejo de modelos complejos de aprendizaje de máquinas suponen una competencia importante a la hora de resolver problemáticas actuales. Se espera, además, ser capaces de definir un mejor hilo conductivo a la hora de realizar un proyecto, definir bien los lineamientos y cumplir las tareas solicitadas de manera prolija, pues, en el mundo laboral, tales habilidades son necesarias para llevar a cabo un proyecto.

6. Anexos

Se procede a implementar los pasos iniciales para trabajar con LIME en la red empleada, siguiendo la siguiente metodología:

1. Se instanció un esquema de transformaciones a operar sobre imagenes `PIL` bajo un objeto de la librería `torchvision.transforms.Compose`. Primero, reescala la imagen a un tamaño de 299x299, luego se realiza un corte a la imagen desde el centro mediante y se transforma a tensor para poder procesar por medio de una red neuronal. Finalmente el tensor obtenido se estandariza con medias [0.485, 0.456, 0.406] y desviaciones estándar [0.229, 0.224, 0.225].

2. Se carga la red `inception_v3` de `torchvision.models`, preentrenada sobre el conjunto de imagenes *ImageNet* en `Pytorch`. Luego, se carga una imagen de control(ver Figura (6.1)) para realizar análisis posteriores con LIME, la imagen se procesa mediante la composición de transformaciones implementadas y se procesa en la red para obtener una predicción. La predicción obtenida corresponde a la clase 'Labrador_retriever' con una probabilidad de 0.72.

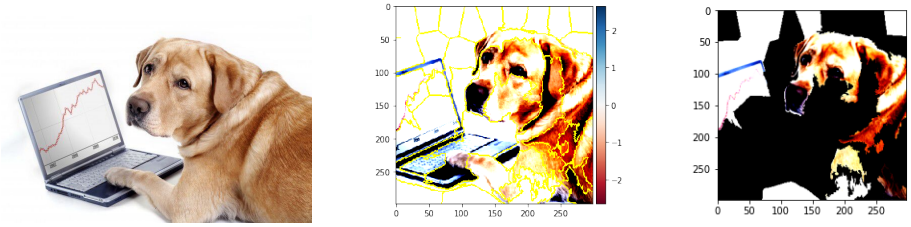

Figura 6.1: A la izquierda, imagen de control sin procesar. Al centro, imagen de control procesada y segmentada. A la derecha, imagen perturbada.

3. Mediante la función `slic` del módulo `skimage.segmentation` se realiza una segmentación de la imagen(ver Figura (6.1)), que consiste en asignar una categoria(que llamaremos superpixel) a cada pixel de la imagen procesada. La segmentación por superpixeles permite separar la imagen en parches que permitiran representar la imagen original mediante la presencia o ausencia de éstos. Se obtuvo un total de 37 superpixeles.

4. Mediante la función `bernoulli.rvs` del módulo `scipy.stats` se generaron 1000 arreglos binarios de largo igual al numero de superpixeles, donde cada entrada corresponde a una realización de una Bernoulli(0.5). Cada arreglo indica una configuración de presencia/ausencia de superpixeles, generando las representaciones interpretables. Se procede a generar una versión perturbada de la imagen de control por cada perturbación. Luego, para cada versión perturbada se realiza una predicción mediante la red `inception_v3` y las predicciones que coinciden con la predicción de la imagen de control, esto es, 'Labrador_retriever', se les asignará el valor 1, mientras que las que predigan una clase distinta serán etiquetadas con el valor 0. Se construye así un arreglo binario `y` que codifica el acierto en la predicción para cada imagen perturbada.

5. Conderando la imagen original como una perturbación donde todos los coeficientes del arreglo de superpixeles es 1, se procede a calcular la distancia coseno entre cada perturbacion y este

arreglo, obteniendo un vector de distancias. Luego, utilizando un valor de $\sigma = 0.25$ se calcula π_x como un kernel exponencial sobre el vector de distancias, obteniendo un arreglo de largo 1000 donde cada entrada de π_x corresponde a un peso sobre una de las perturbaciones.

6. Se procede a construir el conjunto de entrenamiento, dado que nos interesa ver los pesos(importancia) que cada superpixel aporta en la predicción sobre la imagen de control, consideraremos como conjunto de entrenamiento todas las perturbaciones como variable explicativa, y las predicciones respectivas y como variable de respuesta. Luego se procede a entrenar el modelo de regresión logística, obteniendo los pesos.

   Dado que el metodo `.fit_predict` de `LogisticRegression` no permite agregar argumentos sumados además de la función de pesos, luego no es posible discriminar de manera general la cantidad de superpixeles de soporte. Tenemos entonces que para el contexto de regresión logística no es necesario agregar tal medida de complejidad $\Omega(g)$. Sin embargo, cambiando el esquema de optimización es posible implementar una función objetivo que permita dimensionar la complejidad de $\Omega(g)$, por ejemplo, la cantidad de superpixeles que tiene la representación interpretable, de manera que ésta no presente 'pesos' en todas sus componentes y se obtenga una representación más evidente de la predicción.

   Se consideran perturbaciones que consistan en los superpixeles con los 10 pesos más altos y 10 pesos más bajos respectivamente(ver Figura (6.2)) y se utiliza la red `inception_v3` para hacer predicciones.

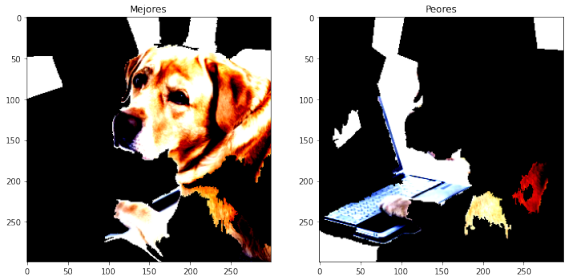

Figura 6.2: A la izquierda, imagen perturbada según los 10 superpixeles de mayor peso. A la derecha, imagen perturbada según los 10 superpixeles de menor peso.

Notamos que filtrando los 10 superpixeles con mayor peso dentro de la regresión logística, junto con los 10 superpixeles de menor peso, incluyendo negativos pues también representan los casos donde se desfavorece la predicción de 'Labrador_retriever', obtenemos 2 representaciones de la imagen de control. De las representaciones obtenidas, se tiene que la obtenida con los pesos más alto representa la predicción de 'Labrador_retriever', luego los superpixeles asociados a tales pesos son los que más aportan en la probabilidad de pertenecer a la clase. Por otra parte, notamos que para los pesos más bajos (negativos) encontramos representaciones que desfavorecen en la predicción de la clase, donde si realizamos una predicción sobre esta representación obtenemos que su clase más probable es 'notebook'. Así, la regresión logistica implementada, dada una segmentación de la imagen de control, es posible representar ésta de una manera interpretable por nosotros(qué superpixeles de la imagen considerados), para obtener una descripción de la predicción, validando de ésta manera la predicción obtenida inicialmente.

7. Para contrastar con el uso de `slic` en la generación de superpixeles, se utilizan dos métodos útiles en segmentación espacial de imágenes, los cuales consisten en utilizar métodos de clusterizacion sobre los arreglos de imágenes. Para ello, se reescalan los colores de la imagen de control a escala de grises y se construye el conjunto de entrenamiento X de dimensiones $(299 * 299) \times 3$, donde la primera dimension esta asociada a cada pixel listado, y en la segunda, las primeras dos componentes corresponden a la posición del pixel en la imagen mientras que la última corresponde a su intensidad de color en la escala de grises. Se utiliza para esta entrega los clasificadores `KMeans` y `MiniBatchKMeans` de la libreria `sklearn.clusters`, con número de clusters igual al número de superpixeles encontrados previamente mediante `slic`. Los clasificadores se ajustan a los datos y luego se realiza un proceso análogo al anterior. Los resultados se ilustran en la Figura (6.3).

   Para `KMeans` obtenemos que la representación con los 10 superpixeles de mayor peso en la regresión logística son clasificados como 'Labrador_retriever', donde a partir de la visualización es facil notar el por qué de la elección, ésta rescata los aspectos más característicos dentro de una raza de perro, esto es, las formas del hocico, las orejas y cráneo. Notamos además que para los 10 superpixeles con menor peso se obtiene como predicción 'conch', que corresponde a las conchas de mar, esto puede deberse a que entre los superpixeles considerados, el tercer y quinto más negativos (la oreja y el cuello) presentan una forma similar a la de una concha boca abajo, luego el clasificador pudo haber considerado como importantes esos pixeles de la imagen para hacer la predicción.

   Para `MiniBatchKMeans` tenemos que la representación con los 10 superpixeles de mayor peso obtienen la predicción esperada, es decir, 'Labrador_retriever'. Notamos que este método de clusterización logró captar el hocico y orejas en la imagen, lo cual ciertamente influyó en la predicción. Además si consideramos los 10 superpixeles de menor peso(negativos incluidos) tenemos que se obtiene como predicción 'Italian_greyhound', perro de pelaje claro y hocico delgado, como muestra la representación.

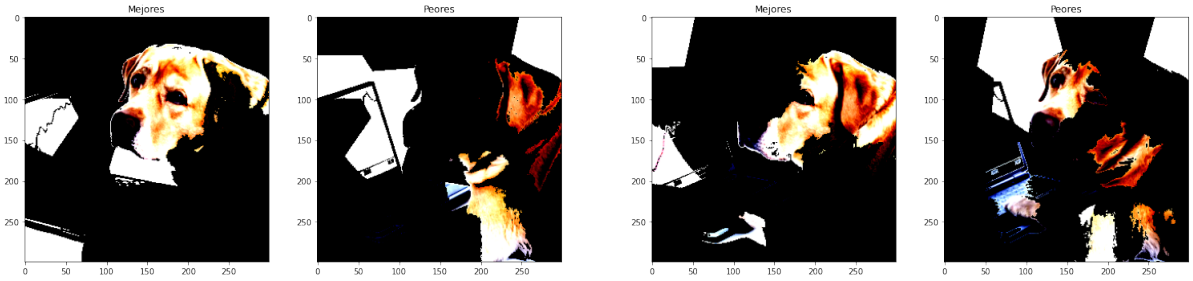

Figura 6.3: Perturbaciones de la imagen de control en base a los 10 mejores y peores superpixeles. A la izquierda, imagenes perturbadas según `KMeans`. A la derecha, imagenes perturbadas según `MiniBatchKMeans`.