# Tarea 2 - Procesamiento distribuido y redes neuronales profundas

MA6202 Laboratorio de Ciencia de Datos

Otoño 2020

## 1. Introducción

La siguiente evaluación corresponde a la segunda tarea del curso de laboratorio de ciencia de datos. A lo largo de la tarea se trabaja sobre un problema de clasificación binaria de imágenes torácicas de rayos X, buscando predecir si la imagen corresponde a neumonía.

Se evaluará la presentación de sus resultados por medio de un informe, las condiciones de entrega requeridas son:

- La extensión máxima del informe es de 8 planas a las que puede añadir 2 para anexos.
- Debe adjuntar un repositorio `git` donde se incluya todo su código.
  - A lo menos 1 `commit` por cada pregunta de la tarea
  - Por lo menos 1 `merge` a través de su trabajo.
- Incluya un documento `jupyter notebook` llamado `tarea2.ipynb` en el cual se exponga todo el procedimiento realizado.
- Por último es necesario también entregar un archivo *hdf* denominado `modelo.h5` que contenga los pesos del mejor modelo de red neuronal obtenido.

Tenga en mente que su informe será revisado por un equipo técnico que debe entender a cabalidad su metodología, ser capaz de replicarlo y evaluarlo a partir de su lectura.

## P1. Carga y transformación de datos

Al abordar un problema de clasificación de imágenes con redes neuronales profundas, es importante optimizar el proceso de carga de datos. Al tratarse de imágenes, es de esperar que no todas las muestras puedan ser almacenadas en memoria de manera simultanea, por lo que es necesario construir un *generador*. El objetivo de la sección es construir un *generador* de muestras eficiente, que minimice el tiempo de carga de datos y optimice el uso de memoria.

El conjunto de datos a utilizar está disponible en este link [1]. En la Figura 1 se observan algunas muestras de cada clase. La carpeta `train` consta de alrededor de 5.000 imágenes y la carpeta `test`, cerca de 600. Al ser un conjunto de datos pequeño, se emplean técnicas de aumentación de datos que serán detalladas a continuación.

1. Instancie objetos de la clase `torchvision.datasets.DatasetFolder` que le permitan cargar las imágenes de las carpetas `train` y `test`. Además, Construya la función `loader`, que permite cargar muestras de la base de datos. Note que existen imágenes que tienen 1 sólo canal. Estas deben ser transformadas a 3 canales mediante concatenación. Dicho procedimiento debe ocurrir en la función `loader`.

   Además, en el parámetro `transform`, instancie un objeto de la clase `torchvision.transforms.Compose` que componga las siguientes transformaciones:

   - Escalamiento de la imagen a un tamaño de $224 \times 224$ pixeles. Además escale los valores de brillo de los pixeles a valores entre 0 y 1, dividiendo por el valor máximo del tipo de dato `uint8`.
   - Con probabilidad $\frac{1}{2}$, voltee la imagen en el eje horizontal.
   - Rote la imagen, con respecto a su centro, con un ángulo aleatorio entre -20°y 20°.

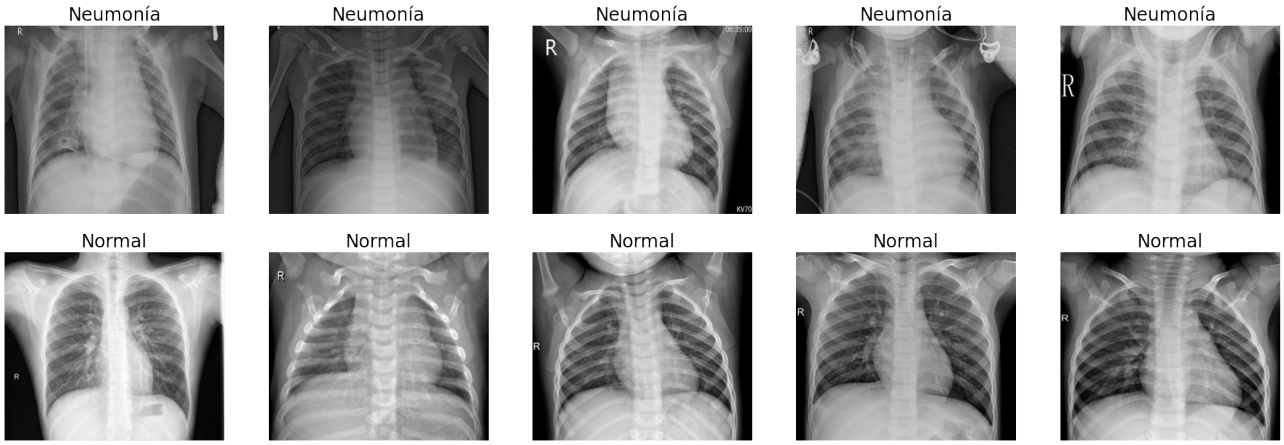



Figura 1: Muestras del conjunto de datos de radiografías de pecho[1]

- Multiplique los valores de brillo de cada canal por un número aleatorio entre 1,2 y 1,5. Cada pixel debe ser multiplicado por un número potencialmente distinto, es decir, a cada valor de brillo corresponde una número aleatorio potencialmente diferente.

Mediante perfilamiento de tiempo de cómputo seleccione las herramientas que le parezcan óptimas para implementar las transformaciones anteriores. Puede utilizar las librerías `pytorch, PIL, skimage` u `opencv`. Incluya en el reporte, una comparación con herramientas de 2 procesamientos de su elección.

**Observación**: Se recomienda realizar por completo las secciones **P1** y **P2** antes de fundamentar su selección.

2. Visualice la cantidad de muestras de cada clase en las carpetas `train` y `test`. Discuta sobre las implicancias de estas distribuciones.

Debido a que el balance de clases es diferente entre la carpeta `train` de la carpeta `test`, es necesario definir cual es la distribución de clases del *problema*. En lo que sigue, asumiremos que dicha distribución es la que presenta la carpeta `test`. Esto quiere decir que en un entorno de producción, se espera recibir muestras distribuidas de manera similar a tal conjunto, que llamaremos *conjunto de prueba*.

3. Separe los índices del objeto `DatasetFolder` de la carpeta `train` en un *conjunto de entrenamiento* y un *conjunto de validación*, con un 80 y 20 % de las muestras respectivamente. Construya la clase `ReplicarMuestreoDePrueba` que herede de `torch.utils.data.Sampler` y permita iterar sobre el conjunto de validación de tal forma que replique la distribución de clases del *conjunto de prueba*, mediante un sobremuestreo de la clase minoritaria. Esta clase debe poseer los métodos:

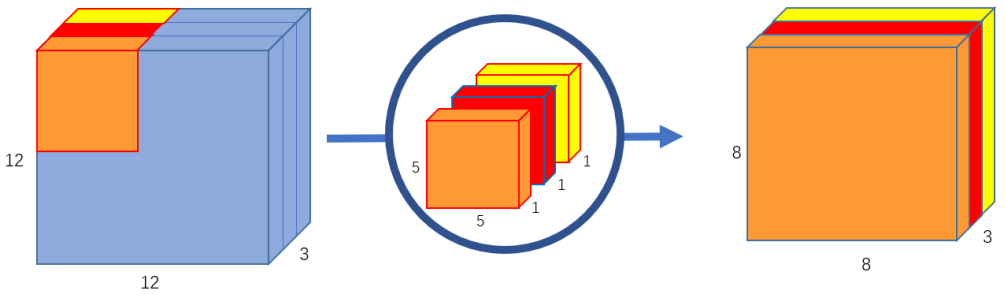
   - `__init__(self, etiquetas_prueba, indices_val, etiquetas_val)`: que guarde como atributos las variables necesarias para generar el muestreo deseado.
   - `__iter__(self)`: que entregue un `iterator` sobre los índices del muestreo deseado.

   *Hint*: Puede ser útil emplear `numpy.random.choice`. Observe que tendrá índices de validación duplicados.
   **Observación**: Asuma que se trata de un problema de clasificación binaria.

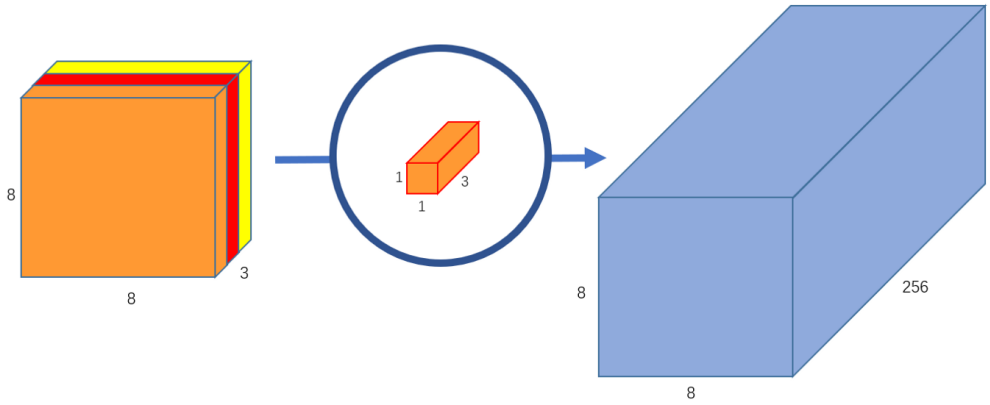
4. Instancie objetos de la clase `torch.utils.data.DataLoader` para recorrer sus conjuntos de entrenamiento, validación y prueba. Para ello utilice los objetos adecuados en el parámetro `sampler`. Discuta en el reporte las implicancias en tiempos de cómputo del parámetro `num_workers`

   *Hint*: Puede ser útil usar la clase. `torch.utils.data.sampler.SubsetRandomSampler` para el *conjunto de entrenamiento* y `torch.utils.data.RandomSampler` para el *conjunto de prueba*.

   **Observación**: Se recomienda realizar por completo las secciones **P1** y **P2** para fundamentar su discusión. Además, como referencia, en `Colaboratory` el parámetro `batch_size=16` es compatible con los recursos de la plataforma.




(a) Capa de convolución *depthwise*




(b) Capa de convolución *pointwise*

Figura 2: Componentes de la capa de convolución *Dephtwise Separable Convolution*
Fuente - *A Basic Introduction to Separable Convolutions*, Chi-Feng Wang, Medium.

## P2. Redes convolucionales profundas

El objetivo de esta sección es construir una red neuronal profunda para el problema de clasificación de imágenes de rayos X sobre neumonía. Dicha red debe ser implementada en `Pytorch`.

Se implementa un tipo de capa de convolución conocida como *Dephtwise Separable Convolution* [2]. Esta consiste en separar una capa de $k$ filtros de convolución de tamaño $n \times n$, *i.e.* definida por $k$ filtros de tamaño $n \times n \times c$ (donde $c$ representa el número de canales) en dos capas de convolución:

- Una capa de convolución llamada *Depthwise*, definida por $c$ filtros de tamaño $n \times n \times 1$, donde cada canal de entrada es convolucionado con su respectivo filtro, obteniéndose así un volumen de salida de $c$ canales. En la figura 2a se ilustra lo mencionado para $n = 5$, $c = 3$ y un *stride* $= 1$.
- Una capa de convolución llamada *pointwise*, definida por $k$ filtros de tamaño $1 \times 1 \times c$ que se aplica al volumen de salida de la capa *depthwise*. En la figura 2b se muestra su funcionamiento para $c = 3$ y $k = 256$. Cabe notar que la función de activación se aplica sólo en el volumen de salida de esta capa.

1. Construya la clase de `DWSepConv2d` que herede de `torch.nn` y haga overriding de los métodos `__init__` y `forward`. El primero debe recibir como parámetros: `in_channels, out_channels, kernel_size, padding, bias=True`. Estos tienen el significado usual empleados en los módulos `torch.nn`.

   *Hint*: Observe que para el ejemplo de la Figura 2 los parámetros tomarían los valores `in_channels=3, out_channels=256` y `kernel_size=5`.

2. Construya una red convolucional profunda mediante la clase `VGG16DWSep` que hereda `torch.nn` y hace overriding de los métodos `__init__` y `forward`, de tal manera que la red posea la estructura detallada en la Tabla 1. Cabe notar que única la función de activación utilizada en todas las capas ocultas de convolución y totalmente conectadas es ***Re**ctified **L**inear **U**nit (ReLU)*.

   Obtenga el número de parámetros que tiene esta estructura y el número de parámetros que habrían sido utilizados si todas las capas `DWSepConv2d` hubiesen sido `Conv2d`. Discuta la ganancia en tiempo de cómputo en entrenamiento y prueba.

| capa | tamaño filtro | padding | stride | # filtros | fn. activación |
|---|---|---|---|---|---|
| Conv2d | 3 | 1 | 1 | 64 | *ReLU* |
| Conv2d | 3 | 1 | 1 | 64 | *ReLU* |
| MaxPool2d | 2 | 0 | 2 | - | - |
| DWSepConv2d | 3 | 1 | 1 | 128 | *ReLU* |
| DWSepConv2d | 3 | 1 | 1 | 128 | *ReLU* |
| MaxPool2d | 2 | 0 | 2 | - | - |
| DWSepConv2d | 3 | 1 | 1 | 256 | *ReLU* |
| BatchNorm2d | - | - | - | - | - |
| DWSepConv2d | 3 | 1 | 1 | 256 | *ReLU* |
| BatchNorm2d | - | - | - | - | - |
| DWSepConv2d | 3 | 1 | 1 | 256 | *ReLU* |
| MaxPool2d | 2 | 0 | 2 | - | - |
| DWSepConv2d | 3 | 1 | 1 | 512 | *ReLU* |
| BatchNorm2d | - | - | - | - | - |
| DWSepConv2d | 3 | 1 | 1 | 512 | *ReLU* |
| BatchNorm2d | - | - | - | - | - |
| DWSepConv2d | 3 | 1 | 1 | 512 | *ReLU* |
| MaxPool2d | 2 | 0 | 2 | - | - |
| Flatten | - | - | - | - | - |
| Linear | - | - | - | 1024 | *ReLU* |
| Dropout(.7) | - | - | - | - | - |
| Linear | - | - | - | 512 | *ReLU* |
| Dropout(.5) | - | - | - | - | - |
| Linear | - | - | - | 2 | - |

Tabla 1: Estructura de la red `VGG16DWSep`

3. Transfiera los pesos de las dos primeras capas de convolución de la red `VGG16` preentrenada en *imageNet* a las dos primeras capas de la red `VGG16DWSep` construida y manténgalos constantes durante el entrenamiento.

Al entrenar redes neuronales profundas, es usual emplear una heurística de regularización llamada *Early Stopping* que consiste en monitorear alguna métrica de desempeño (usualmente la función de costo) en un conjunto de validación, para así detener el entrenamiento cuando dicha métrica empeora de forma sostenida.

En el siguiente fragmento de código ilustramos el uso de la heurística implementada en la clase `EarlyStopping`:

```
es = EarlyStopping(...)

for epoch in range(num_epochs):
    # ciclo de entrenamiento
    ...

    # ciclo de validacion
    ...
    metrica_validacion = ...

    if es.deberia_parar(metrica_validacion):
        # se cumple el criterio de "early stop"
        break
```

Listing 1: Funcionamiento de la heurística *Early Stopping*

4. Programe la clase `EarlyStopping`. Cuyo objetivo es implementar la heurística mencionada. Esta debe poseer los siguientes métodos:

   - `__init__(self, modo='min', paciencia=5, porcentaje=False, tol=0)`: donde los parámetros:
     - `modo`: toma valores en `'min'` o `'max'`. Este define si la métrica obtenida en el conjunto de validación es considerada mejor al ser más pequeña o más grande según respectivamente.
     - `paciencia`: define el número de épocas en la que la métrica de validación puede empeorar sin detener el entrenamiento.
     - `porcentaje`: define si la comparación entre métricas de desempeño en validación, deben realizarse en términos relativos (como porcentaje de a la mejor métrica de desempeño observada) o absolutos.

- `tol`: define la diferencia mínima que debe existir con respecto la mejor métrica de desempeño observada en validación, para considerar si existe un empeoramiento del desempeño y actualizar el valor del contador asociado a `paciencia`.

Se deja a criterio del equipo la definición de atributos para el correcto funcionamiento de la clase.

- `mejor(self, metrica_validacion)`: que compare la `metrica_validacion` con la mejor métrica de desempeño ya observada. Dicha comparación debe realizarse considerando los parámetros `modo`, `porcentaje` y `tol`, y debe retornar `True` o `False`.
- `deberia_parar(self, metrica_validacion)`: que llame al método `mejor` y retorne `True` cuando la cantidad de épocas en que `metrica_validacion` empeora con respecto a la mejor. métrica de desempeño observada sea igual a `paciencia`. En caso contrario retorna `False`.

5. Implemente el ciclo de entrenamiento de la red `VGG16DWSep` utilizando una instancia de la clase `EarlyStopping` según el esquema de Listing 1 con sus parámetros por defecto y además vaya guardando los pesos del mejor modelo obtenido a través de las épocas. Para dicho ciclo de entrenamiento use:

   - *Entropía cruzada* como función de costo.
   - *Adam* cómo algoritmo de optimización, con parámetros `lr=1e-4` y `weight_decay=1e-5`.

   Reporte sus resultados en términos de *accuracy* y *f1-score*. Compruebe que obtiene resultados superiores a ,75 y ,8 respectivamente.
   *Hint*: Por la magnitud del problema se recomienda usar GPU. Recuerde que en Colaboratory tiene acceso gratuito a dicho recurso.

6. Implemente aumentación de datos en el conjunto de prueba: para una observación $x_i$, obtenga $d$ muestras $\{x_i^{(j)}\}_{j=1}^{d}$ para calcular la predicción $\hat{y}_i = \arg\max_{j=1,\dots,d} f(x_i^{(j)})$, donde $f(\cdot)$ representa la probabilidad asociada la predicción de su red. Discuta si es correcto o no realizar esto y soporte su argumento con los resultados obtenidos en términos de *accuracy* y *f1-score*.
   *Hint*: Puede ser útil implementar una clase que herede de `torch.utils.data.Sampler` y que reciba como parámetros la cantidad de muestras para una misma observación $x_i$.

## P3. Interpretabilidad

El objetivo de esta pregunta es que implemente un modelo auxiliar de interpretabilidad local sobre las predicciones que genera una red neuronal. Esto consiste en generar perturbaciones sobre datos de entrada, con el fin de comprender la importancia de las variables en los procesos de predicción. Para ello deberá implementar el método LIME (**L**ocal **I**nterpretable **M**odel-**A**gnostic **E**xplanations [3]).

### Desarrollo teórico

El procedimiento LIME consiste en una metodología diseñada para otorgar *interpretabilidad* a modelos de aprendizaje que suelen ser denotados como "caja negra". Por interpretabilidad se entiende, la capacidad de establecer relaciones claras entre las variables de un fenómeno y la respuesta que producen. LIME es "agnóstico en el modelo", esto se refiere a que pueda ser utilizado para cualquier tipo de modelo de predicción.

La idea central de LIME consiste en aproximar localmente el comportamiento de un predictor, utilizando un modelo que sea interpretable como por ejemplo regresión lineal o arboles de decisión. En términos concretos, dada una instancia a predecir $x \in \mathbb{R}^d$ (dato de entrada), se utilizará un vector $x' \in \{0,1\}^{d'}$ como representación interpretable. Se define así, una **explicación** cómo un modelo $g \in G$, donde $G$ corresponde a una familia de modelos potencialmente interpretable, el dominio de cada $g \in G$ será $\{0,1\}^{d'}$. Para asegurar que la aproximación buscada sea interpretable por un humano, se utiliza una medida de complejidad $\Omega(g)$ sobre cada $g \in G$, considerando el grado de complejidad en contraposición a la interpretabilidad de un modelo.

Sea un modelo predictor $f : \mathbb{R}^d \to \mathbb{R}$, sea además $x \in \mathbb{R}^d$ en el conjunto de datos de entrada, y $x' \in \{0,1\}^{'d}$ su representación interpretable. Para $x$ se define $\pi_x(z') : \{0,1\}^{'d} \to \mathbb{R}$ como una medida de similitud entre $x'$ y $z' \in \{0,1\}^{'d}$. Finalmente, se define $\mathcal{L}(f, g, \pi_x)$ en una vecindad inducida localmente por $\pi_x$. Para asegurar interpretabilidad y fidelidad local (asociada a $x$), la explicación producida por LIME se obtiene resolviendo la siguiente expresión:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g) \tag{1}$$

En esta pregunta, se utilizará regresión logística cómo familia de explicaciones $G$, es decir para cada $g \in G$ este será de la forma $g(x') = \sigma(w_g \cdot x')$. Cómo función de fidelidad $\mathcal{L}$ se usa el la verosimilitud asociada a la regresión logística, ponderada localmente por $\pi_x$, es decir:

$$\mathcal{L}(f, g, \pi_x) = \sum_{z,z'} \pi_x(z)(f(z)\log(g(z')) + (1 - f(z))\log(1 - g(z')) \tag{2}$$

En este caso $\pi_x$ será un kernel exponencial definido por una medida de similitud, se utilizará la distancia coseno:

$$\pi_x(z') = \exp(-d(x', z')^2/\sigma^2) \tag{3}$$

$$d(x', z') = 1 - \frac{x' \cdot z'}{\|x'\| \, \|z'\|} \tag{4}$$

Implemente los paso iniciales para trabajos con LIME, para esto:

1. Instancie un objeto `torchvision.transforms.Compose`, este opera sobre imágenes `PIL` y le aplique las siguientes transformaciones:
   - Escalamiento de la imagen a un tamaño de $229 \times 229$ píxeles.
   - Opere por medio de `CenterCrop(299)`.
   - Transforme la imagen en un objeto `Tensor`.
   - Normalice con las medias `[0.485, 0.456, 0.406]` y, desviaciones estándar `[0.229, 0.224, 0.225]`.
2. Cargue la red `inception_v3` entrenada sobre *imageNet* (en Pytorch). Utilice esta red para hacer predicción sobre una imagen de control a la cual se aplican las transformaciones antes definidas. Obtenga la clase más probable asignada por la red.

   *Hint*: Puede ser útil la función `decode_predictions` del módulo `keras.applications.imagenet_utils` sobre las predicciones de la red cargada.
3. Segmente la imagen de control utilizando la función `slic` del módulo `skimage.segmentation`, para los parámetros `start_label=0, n_segments=80`. El resultado de esta segmentación es un arreglo de dimensión $299 \times 299$ que asigna una categoría para cada píxel de la imagen procesada. Todos los píxeles en la imagen que comparten etiqueta conforman un super-píxel dentro de la imagen. Utilice la función `mark_boundaries` del módulo `skimage.segmentation` en conjunción con `imshow` del módulo `skimage.io` para visualizar los bordes inducidos por el conjunto de super-píxeles.

Al representar una imagen $x$ por medio de la presencia y ausencia de los super-píxeles se logra una representación interpretable $x'$ según un vector de entradas binarias.

Genere perturbaciones en la imagen de control, para esto siga los siguientes pasos:

4. Defina un número de *perturbaciones* a realizar (al menos 1,000). Cada perturbación consiste en arreglo binario, donde cada componente es asociada a un super-píxel. Estos arreglos serán las representaciones interpretables de la imagen de control ($x'$ asociado a $x$). Considere cada entrada de su arreglo de perturbaciones como una variable aleatoria `Bernoulli` con $p = 0,5$.
5. Genere tantas versiones perturbadas de la imagen de control como perturbaciones haya construido. Obtener una imagen perturbada consiste en asignar el valor 0 en cada canal de color en aquellos píxeles cuyos super-píxeles asociados tengan su componente nula en el vector de perturbaciones. Obtenga una visualización de una imagen perturbada.
6. Haga predicción sobre las imágenes perturbadas utilizando la red `inception_v3`. Asocie el valor 1 como etiqueta a las imágenes perturbadas que sean clasificadas a la misma categoría de la imagen de control y 0 en caso contrario, el arreglo binario correspondiente se denotará $y$.
7. Calcule $\pi_x$ según la expresión (3). Para ello, obtenga la distancia de coseno entre las perturbaciones asociadas a cada imagen perturbada y el vector de perturbación de la imagen de control $x$ según lo indica (4).

Una vez obtenido un conjunto de representaciones para la imagen de control $x$ y el vector de pesos asociados $\pi_x$, se pasa a minimizar la función de fidelidad, para esto:

8. Genere un conjunto de entrenamiento $D_p$. Este consta de vectores de perturbación como observaciones. La variable de respuesta será el arreglo $y$ generado anteriormente.

9. Utilice la clase `LogisticRegression` del módulo `sklearn.linear_model` para entrenar un clasificador sobre conjunto de entrenamiento $D_p$. Haga uso de $\pi_x$ al momento de utilizar el método `.fit()`. ¿Es posible agregar una medida de complejidad $\Omega(g)$ con este esquema? argumente al respecto.

10. Utilice los coeficientes del clasificador anterior para inferir los super-píxeles de mayor importancia en la clasificación de la imagen de control. Obtenga una visualización y discuta al respecto.

La segmentación antes utilizada se hace de *manera espacial*. Es decir, se realiza una clusterización sobre la escala de grises según su posición en la imagen. Del procedimiento recién explicado para implementar LIME reemplace la etapa de segmentación de la imagen por 2 segmentaciones espaciales utilizando 2 modelos de clustering a su elección, para ello:

11. Clusterice sobre un conjunto de entrenamiento `X` con $299^2$ observaciones, es decir, una observación por píxel en la imagen de control escalada. Cada observación de `X` consta de 3 componentes, donde la primera y segunda son espaciales (posición del píxel en la imagen) y la última es el valor de intensidad asociado al píxel (escala de grises). Utilice los clusters descubiertos para generar super-píxeles.

    *Hint:* En Scikit-learn el método `fit_predict` en algoritmos de clustering puede ser de ayuda para generar super-píxeles.

12. Aplique el esquema LIME desarrollado anteriormente sobre sus super-píxeles. Interprete los resultados.

Finalmente aplique un esquema LIME sobre una predicción de la red `VGG16DWSep` implementada en la parte 2 de la tarea. Discuta sus resultados y observaciones.

**(Bonus)** Se puede hacer segmentación en el espacio de colores, en este caso, el conjunto a segmentar consiste en una transformación de la imagen de control escalada en un arreglo de dimensión $(\text{ancho} \cdot \text{largo}) \times 3$ (en el caso de `inception_v3`, la dimesión sería $299^2 \times 3$). Sobre este nuevo arreglo implemente un algoritmo de clustering y genere super-píxeles. Aplique un esquema LIME con una familia $G$ no basada en modelos lineales y compare con la misma familia sobre un método de segmentación espacial.

# Referencias


[1] D. Kermany, K. Zhang, and M. Goldbaum, "Labeled optical coherence tomography (oct) and chest x-ray images for classification," *Mendeley data*, vol. 2, 2018.

[2] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[3] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?."[ex]plaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144, 2016.