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

Informe Tarea 2

Procesamiento distribuido y redes neuronales profundas

Integrantes: Pablo Araya
Benjamín Barrientos
Sebastián López
Profesor: Nicolas Caro
Auxiliar: Rodrigo Lara M.
Fecha de entrega: 18 de julio de 2020
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Introducción

Las redes neuronales son una herramienta potente de Machine Learning, que permiten resolver problemas de clasificación complejos donde otros clasificadores no llegan a los rendimientos deseados. Los problemas son de todo tipo: clasificación de imágenes, sonidos, palabras y un sin fin más; y son aplicables en diversas áreas de las ciencias.

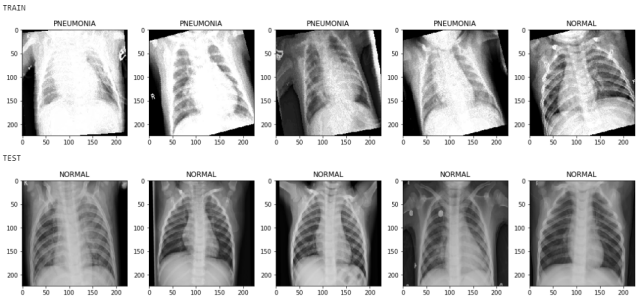
En el presente informe se presenta la Tarea 2 del curso Laboratorio en ciencia de datos el cual consta en el tratamiento de imágenes y modelos para predecir la categoría de estas imágenes. Estas imágenes son radiografías de pecho de personas enfermas y personas sanas en donde, a través de redes neuronales convolucionales, se intenta predecir la categoría de estas imágenes, es decir, si corresponden a una persona enferma o una persona sana.

1. Carga y transformación de datos

En esta sección se describe la carga y transformación de datos. El conjunto de datos a trabajar consta de imágenes de radiografía de pecho las cuales pueden ser de una persona enferma (Neumonia) o una persona sana (Normal). Estas vienen ya separadas en entrenamiento y testeo y se procede a aplicar una serie de transformaciones a cada imagen, las cuales se mencionan a continuación:

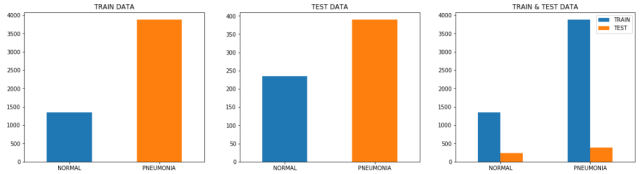
- Escalar cada imagen a un tamaño de 224×224 píxeles donde los valores de brillo de los píxeles deben estar escalados a valores entre 0 y 1 al dividir por el valor máximo del tipo de dato uint8.
- Con probabilidad $\frac{1}{2}$, voltear la imagen en el eje horizontal.
- Se rota la imagen, con respecto a su centro, con un ángulo aleatorio entre $-20°$ y $20°$.
- Se multiplican los valores de brillo de cada canal por un numero aleatorio entre 1,2 y 1,5 donde cada pixel es multiplicado por un numero potencialmente distinto.

A continuación se muestra un ejemplo de las imágenes luego de las transformaciones:



Lo anterior se hizo utilizando dos librerías distintas las cuales fueron `PIL` y `skimage`. El que tuvo mejor desempeño fue `PIL` el cual demoro aproximadamente 102 milisegundos y `skimage` demoro aproximadamente 1.79 segundos.

Posterior a aplicar las transformaciones se estudio la distribución del conjunto de testeo. Para esto es necesario ver la cantidad de imágenes por tipo en cada conjunto. A continuación se muestra un gráfico con la información deseada:



Se puede observar que en ambos conjuntos hay una mayor presencia de imágenes de tipo Neumonia que de tipo Normal pero que las distribuciones en el conjunto Train y Test son distintas.

Bajo el supuesto de que la distribución de clases del problema viene dada por la del conjunto test se quiere separar la data train en conjuntos entrenamiento y validación en donde se quiere generar una muestra a través del conjunto de validación que simule la distribución del conjunto de testeo. Para ello se programo una clase llamada `ReplicarMuestreoDePrueba` que itera sobre el conjunto de índices del conjunto de validación y agrega muestras hasta alcanzar una distribución cercana a la del conjunto de prueba. A continuación se muestra una tabla con las distribuciones mencionadas:

Tabla 1: Distribución de los conjuntos

Conjunto	Distribucion
Test	0.6250
Sample Val.	0.6251

Finalmente se procede a instanciar objetos de la clase `torch.utils.data.DataLoader` para recorrer los conjuntos de entrenamiento, validación y de prueba. Para estudiar el funcionamiento del parámetro `num_workers` se programo una función que extrae del conjunto especificado un sample de tamaño 16 con el `num_workers` es-

pecificado. A continuación se muestra una tabla con los tiempos de computo:

Tabla 2: Tiempos de computo del parametro `num_workers`

Num_workers	Tiempo
0	321 ms
1	388 ms
2	727 ms
3	1.08 s
4	1.8 s

Como se puede observar en la tabla a medida que aumenta el valor de `num_workers` el tiempo de ejecución aumenta.

2. Redes convolucionales profundas

El objetivo de esta sección es presentar como se construyó la red neuronal profunda para el problema de clasificación de imágenes de rayos X sobre neumonía, implementada en `Pytorch`.

2.1. Arquitectura

Se implementa un tipo de capa de convolucion conocida como *Dephtwise Separable Convolution*. 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.
- 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. Cabe notar que la función de activación se aplica sólo en el volumen de salida de esta capa.

Para esto, se crea la clase `DWSepConv2d` heredando de `torch.nn` que hace lo anteriormente mencionado. Se usa directamente la clase `torch.nn.Conv2d` aplicando el esquema anterior, es decir, la capa depthwise primero aplica una convolución 2D que recibe una imagen según `in.channels` y produce `in.channels` capas de salida, con `kernel_size`, `padding` y `bias` según lo seleccionado. Luego, la capa pointwise recibe las capas de la convolución y produce `out.channels` capas de salida con `kernel_size = 1` y `bias` según lo seleccionado.

A continuación se crea la red convolucional profunda usando la clase anterior, es decir aplicando *Dephtwise Separable Convolution*. Además de esta convolución, se presenta el resto de la estructura en la siguiente tabla.

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

El modelo anterior posee 110997378 parámetros, que serían 110925634 si en vez de `DWSepConv2d` se usara `Conv2d`.

Para terminar, se transfieren 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 se mantienen constantes congelando sus gradientes.

2.2. Entrenamiento

Para el entrenamiento se usó la heurística de regularización llamada *Early Stopping*. Se crea la clase con el mismo nombre, que posee los siguientes métodos y parámetros:

- `___init___(self, modo='min', paciencia=5, porcentaje=False, tol=0)`:
  - `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 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`.
  - `observacion`: define la última observación del modelo, mientras va a comparar con la métrica de validación.
  - `epocas`: define un contador de cuantas épocas se ha pasado sin una mejoría en la observación.
- `mejor(self, metrica_validacion)`: compara la métrica validación con la mejor métrica de desempeño ya observada. Dicha comparación se realiza considerando los parámetros `modo`, `porcentaje` y `tol`, y retorna `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`.

Seguidamente se implementa el ciclo de entrenamiento de la red `VGG16DWSep` utilizando una instancia de `EarlyStopping`. Para esto, se define una función `train(epochs, model, loss_func, optimizer, train_dl, valid_dl)`: que usa los *DataLoaders* de la sección anterior y además utiliza

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

La función `train` imprime como evoluciona el modelo en cada época, en términos de *accuracy* para el proceso de entrenamiento y validación, así como `f1-score`. Durante el entrenamiento se produce un *accuracy* máximo del 94.12 % en el conjunto de validación con 10 épocas. Mientras que probando el modelo se alcanzó un 70 % al trabajar con muestras del dataset de prueba.

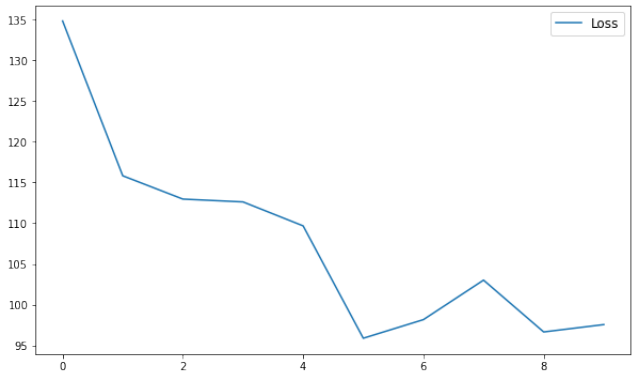


Figura 1: Función de pérdida.

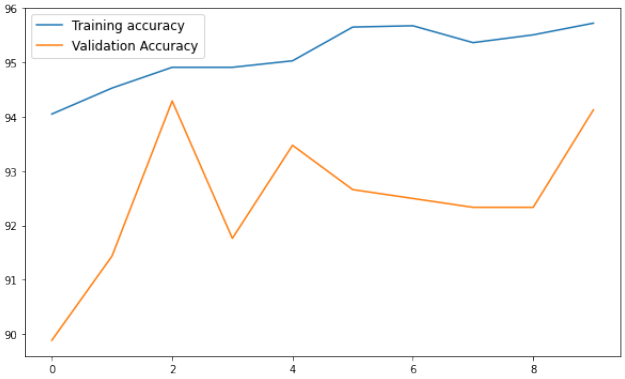


Figura 2: Training y Validation accuracies.

3. Interpretabilidad

Implementaremos **LIME**, el cuál es un modelo auxiliar que nos permitirá mejorar la interpretabilidad de las predicciones entregadas por la red neuronal en la sección anterior. Esto se hace localmente, es decir, de a una imagen a la vez. Se implementa tomando una imagen del dataset y perturbándola, lo que nos permite entender mejor cuales son las variables importantes para la red neuronal. Para su implementación, se usará una **imagen de control**, la cuál corresponde a una imagen de una llama como se puede ver a continuación. Además, se usará la predicción sobre esta imagen entregada por la red neuronal **inception v3**, la cuál está previamente entrenada sobre el dataset ImageNet.

Figura 3: Imagen de control

Para empezar, se transforma la imagen de control al formato adecuado para poder ser introducida en la red. Se la hacen las siguientes transformaciones:

- Escalamiento a 299 × 299 pixeles.
- Recorte de la imagen de 299 × 299 pixeles.
- Transformación a un objeto tipo Tensor
- Normalización con medias [0.485, 0.456, 0.406] y desviaciones estándar [0.229, 0.224, 0.225].

Una vez transformada, la imagen se le pasa a la red neuronal inception v3, la cuál predice que la imagen representa a una llama como era esperado.

A continuación, se procede a segmentar la imagen en 80 partes. A cada una de las partes de esta segmentación la llamaremos un **superpixel**, y cada pixel de la imagen estará contenido en un y solo un superpixel. La partición se realizó con la función **slic**, que implementa el método de k-means en el espacio de colores.

La partición hecha sobre la imagen se ve como sigue:

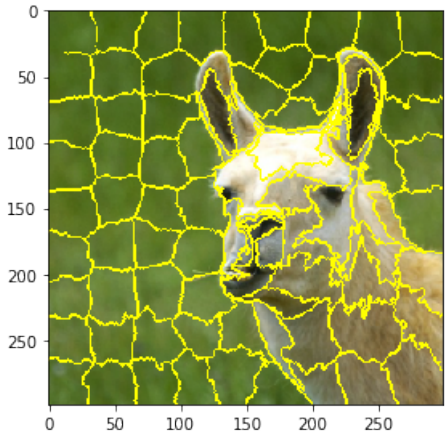


Figura 4: Imagen de control segmentada

Figura 5: Imagen de control perturbada

Con esta segmentación, se generarán las perturbaciones sobre la imagen de control de la siguiente manera:

Se generan 1000 arreglos binarios. Cada uno de estos arreglos se genera de forma aleatoria, pues cada componente de un arreglo viene dada por una variable aleatoria de distribución Bernoulli de parámetro 0.5. Cada arreglo es de largo la cantidad de superpixeles de la imagen, en este caso, 80.

Cada componente de este arreglo binario representa a un superpixel, donde si la componente es 0, diremos que el superpixel que representa está apagado, y por lo tanto, no aparecerá en la imagen. Por otro lado, si la componente es 1, el superpixel asociado aparecerá como es normal en la imagen. De esta forma, se procede a generar 1000 versiones perturbadas de la imagen de control, una por cada perturbación definida anteriormente.

Una de las perturbaciones realizadas sobre la imagen se ve como sigue:

Luego, cada una de estas perturbaciones se le entrega a la red inception v3. Si la predicción sobre la imagen perturbada es la misma que la que hizo para la imagen de control, es decir, si para la imagen perturbada la red predice que es la imagen de una llama, entonces guardamos el valor 1 en un arreglo. Si por el contrario, predice algo distinto a llama, guardamos un 0. Al arreglo donde guardamos estos valores lo llamaremos **hits**.

Definiremos una distancia, llamada π_x para medir que tan similar o distinta es una imagen perturbada de la imagen de control, dicho de otra manera, si tiene muchos o pocos superpixeles apagados.

Definimos entonces π_x, haciendo uso de la distancia coseno, de la siguiente manera:

$$\pi_x(z') = exp\left(\frac{-d(x', z')^2}{\sigma^2}\right) \quad (1)$$

$$d(x', z') = 1 - \frac{x' \cdot z'}{||x'||\ ||z'||} \quad (2)$$

Con σ un hiperparámetro definido como $\sigma = 0.25$. Se procede a calcular la distancia entre cada imagen perturbada y la imagen de control. Estos resultados se van almacenando en un arreglo que llamaremos **pesos**.

Utilizando los vectores de perturbación como datos de entrenamiento, y el vector hits como variable de respuesta, entrenaremos un **Regresor Logístico**. Para esto, se procede a minimizar la

función de fidelidad $\mathcal{L}$ con respecto a la función g, determinada por sus coeficientes w_g. En este caso la fidelidad será la verosimilitud asociada a la regresión logística, ponderada localmente por π_x. Se muestra en detalle en el anexo.

Dentro de este esquema, no es posible agregar una medida de complejidad $\Omega(g)$ sobre la familia de modelos interpretables, puesto que el ajuste del Regresor Logístico se hace mediante el método **.fit()** previamente implementado, el cuál no permite agregar nuevos términos como sumandos.

Una vez ajustado el regresor, obtenemos el vector con sus coeficientes w_g, los cuales nos inidican la importancia de cada superpixel. Mientras más grande sea el coeficiente, más importante será el superpixel en la imagen para poder calificarla correctamente. Obtenemos los 20 superpixeles de mayor importancia según el siguiente gráfico:

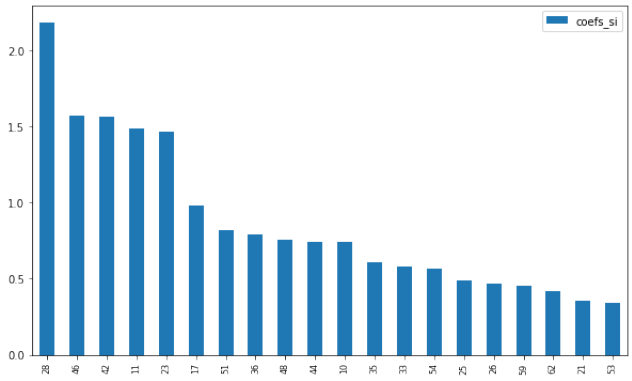


Figura 6: Superpixeles y sus coefs. de mayor a menor

La imagen, utilizando solo los 20 superpixeles más importantes, se ve como sigue:

Figura 7: Imagen con superpixeles de más importancia

Se puede apreciar que, a pesar de algunos errores pequeños, el esquema identifica aquellas zonas o superpixeles más importantes para la red a la hora de clasificar, y que aquellas zonas corresponden a lo que visualmente un humano interpreta como una llama.

Probaremos otros métodos de segmentación, y así poder evaluar si cambiar el método y la segmentación de la imagen cambia los resultados de interpretabilidad. Para esto probaremos segmentar mediante los métodos de **K-Means (KM)** y **Gaussian Mixture (GMM)**. La cantidad de segmentos también será de 80, de manera de que las tres particiones sean comparables.

Las particones generadas por ambos métodos se ven como sigue:

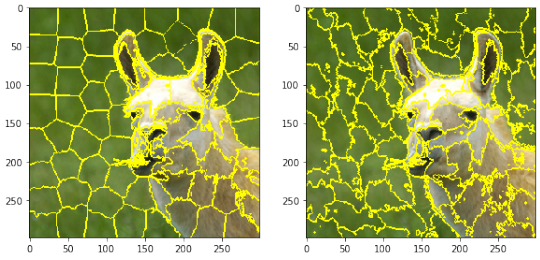


Figura 8: Partición por KM (izquierda) y GMM (derecha)

Aplicando el mismo esquema utilizado anteriormente, podemos obtener los superpixeles más importantes según estas dos nuevas particiones. Los 20 superpixeles más importantes se

ven como sigue:

Figura 9: Superpixeles más importantes según KM (izquierda) y GMM (derecha)

Se puede ver que la red prioriza ciertas zonas de la imagen a la hora de clasificar, y estas zonas son, a grandes rasgos, las mismas independiente de la partición. Además, estas zonas corresponden con lo que un humano interpreta como una llama.

En resumen, la red efectivamente logra capturar aquel valor abstracto de la imagen que podría denominarse "la llama". Como vimos que la partición escogida no provocaba grandes cambios en los resultados de interpretabilidad, seguiremos trabajando con la partición por GMM.

Con esto, el modelo LIME está listo para ser usado en nuestra red neuronal. Tomamos una imagen cualquiera del dataset, junto con la predicción entregada por la red **VGG16DWSep** entrenada en la sección anterior. La imagen seleccionada y su predicción son:

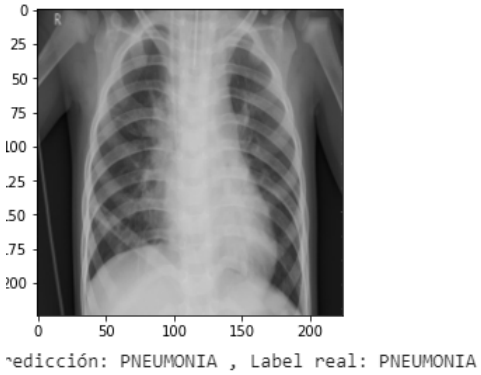


Figura 10: Imagen de prueba, de label 'Pneumonía'

Aplicando el esquema LIME sobre esta imagen y su predicción, obtenemos que los 20 superpixeles de mayor importancia se ven como sigue:

Figura 11: Superpixeles más importantes para identificar Pneumonía

Podemos notar que el modelo captura aquellos atributos importantes de la radiografía de pulmón, centrando su predicción en aquellos pixeles que están asociados al pulmón en sí. Más aún, los pixeles de mayor importancia son los asociados a las zonas blancas de la radiografía.

Conclusiones

Notamos que la red es suficientemente 'fuerte' por las transformaciones hechas gracias a la aumentación de datos vista en las primeras secciones. Esto se ve reflejado en el *accuracy* ya que a pesar de haber entrenado con pocas épocas se obtiene un valor aceptable del 70 %.

Del los resultados del modelo LIME, podemos concluir que la red es capaz de identificar los atributos abstractos de una imagen al momento de clasificar. Sin embargo, esta identificación de atributos es local, es decir, los atributos importantes son diferentes para cada imagen, por lo que no es posible generalizar un modelo de interpretabilidad que funcione para un dataset de gran tamaño. Más aún, no es posible siquiera generalizar un modelo de interpretabilidad para una imagen replicada varias veces, sobre la cual se apliquen transformaciones como rotaciones y traslaciones.

Se cumplen los objetivos de la tarea, al lograr implementar una red neuronal exitosamente y el esquema LIME para dar más interpretabilidad.

Anexo

La función de fidelidad $\mathcal{L}$ está dada por:

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

Donde σ representa a la función sigmoide, e $y_{z'}$ representa a la componente del vector hits asociada a la perturbación z'.