DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Deep Learning:
Tarea 2

Alumno: Rodolfo Fariña Reisenegger

Profesores: Julio Godoy Del Campo
Guillermo Cabrera Vives

Ayudantes: Alejandra Fernández
Constanza Vásquez

13 de Junio del 2021, Concepción

Introduccion

Con la llegada de grandes capacidades de cómputo accesible desde los 2010, las redes neuronales profundas han demostrado ser de gran utilidad para resolver problemas altamente complejos como por ejemplo la clasificación de distintas enfermedades.

Han habido muchos proyectos que utilizan redes neuronales profundas en el área de la medicina. Estas pueden ver patrones o identificar anomalías que resultan difíciles de ver para el ojo humano

En esta tarea se han utilizado las redes neuronales convolucionales para identificar la existencia de COVID-19 en los pacientes. Se han aplicado los conocimientos adquiridos a lo largo del semestre y también se ha investigado bastante por iniciativa propia para crear un programa que pueda predecir COVID dada una imagen de rayos-x del tórax de un paciente.

Sobre los archivos

Se ha adjuntado el código fuente bajo el nombre de *"Rodolfo_Farina_Tarea-2.ipynb"*, las predicciones para el conjunto de test están con el nombre de *"predicted_rodolfo.csv"*, y los pesos de la red que realizaron esa predicción están con el nombre de *"simpleCNN.h5"*.

Experimentacion

Para realizar esta tarea se fue experimentando con varias redes neuronales. Ya que se estaba lidiando con imágenes, se ha decido usar una red convolucional, para ir extrayendo features de la imagen a medida que se procesa por la red.

Desde un inicio se ha decidido clasificar por COVID o NO COVID, esto debido a el output pedido en el *"predicted.csv"*, y también porque no veía que valor podría agregar hacer diferencias entre neumonías y otras anomalías.

Al inicio se utilizó la red vista en la práctica 6, donde se vieron las redes convolucionales.

```
from keras import layers
model = keras.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Figura 1: Red vista en la practica 6

Esta red fue adaptada para recibir un input de tamaño 256x256x1, puesto que las imágenes de rayos-x no estaban a color. También se cambió la capa final a una capa densa de 1 solo neurona, con una función de activación sigmoide, para ir diferenciando entre covid y no covid.

Esta red obtiene resultados aceptables, con un 78-82% precisión utilizando el optimizador adam y binary-crossentropy (ya que solo teníamos COVID / NO-COVID)

Con la idea de aumentar la precisión de esta red neuronal, se creó un modelo distinto a este, que usaba las siguientes capas:

```
simpleCNN = Sequential()
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Flatten())
simpleCNN.add(layers.Dense(16, activation='relu'))
simpleCNN.add(Dropout(0.35))
simpleCNN.add(layers.Dense(1, activation='sigmoid'))
```

Figura 2: Primera red neuronal

Esta red neuronal daba mejores resultados, la precisión era baja, de un 76-80%, pero ya que eran los primeros intentos se consideró un éxito. Se usaron pocas capas debido a que el tiempo de ejecución aumentaba mucho si se agregan más.

Esta red neuronal usaba el optimizador SGD, y tenía el problema de que en las primeras épocas no cambiaba nada la precisión, a pesar de que había regularización y que debería estar aprendiendo. Modificando el learning rate obtuve algunos resultados, pero necesitaba demasiadas épocas para llegar a algo.

Esto fue resuelto cambiando al optimizador “adam”, lo cual hacía que la red aprendiera todas las épocas, y que también alcanzará un nivel más alto de precisión, de alrededor de 83%.

Sin embargo, se presentó otro problema.

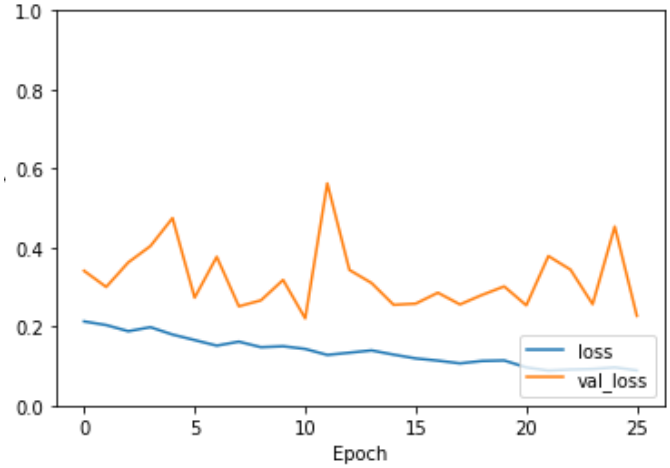

Figura 3: Problemas de la primera red neuronal

El gráfico de la función de pérdida de la red era muy extraño, por esto, se concluye que la red no estaba aprendiendo correctamente.

Como la red no estaba aprendiendo, se agregaron más capas convolucionales, con la idea de que aprendiera mejor. En este momento fue descubierta la posibilidad de activar aceleración de GPU, lo cual permite agregar todas estas capas y ejecutarlas en un tiempo razonable.

```
simpleCNN = Sequential()
simpleCNN.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(256,256,1)))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.40))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Flatten())
simpleCNN.add(layers.Dense(16, activation='relu'))
simpleCNN.add(Dropout(0.35))
simpleCNN.add(layers.Dense(1, activation='sigmoid'))
```

Figura 4: Red neuronal modificada

Esta red aprendía mucho mejor, en ciertos casos llegaba a 90% accuracy, pero aún tenía el problema de que la pérdida seguía siendo muy “saltarina”, tal cual como fue mostrado en la figura 3.

Al investigar en varias fuentes y otras redes neuronales que realizaban trabajos similares, fue posible darse cuenta de que la red neuronal estaba siendo regularizada demasiado, por esto fue reducido el dropout de las capas convolucionales a 25%.

Además, la capa densa final fue aumentada de 16 neuronas a 64, puesto que el output de la última capa de pooling antes del flatten era de 14x14x1, por esto me di cuenta que 16 neuronas no eran suficientes para eso.

```
simpleCNN = Sequential()
simpleCNN.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(256,256,1)))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Flatten())
simpleCNN.add(layers.Dense(64, activation='relu'))
simpleCNN.add(Dropout(0.35))
simpleCNN.add(layers.Dense(1, activation='sigmoid'))
```

Figura 5: Red neuronal final

Con lo cual se obtuvo la red neuronal de la figura 5. La pérdida de esta red neuronal ya no fluctúa constantemente, sino que se veía similar a las redes vistas previamente en clases (véase figura 6).

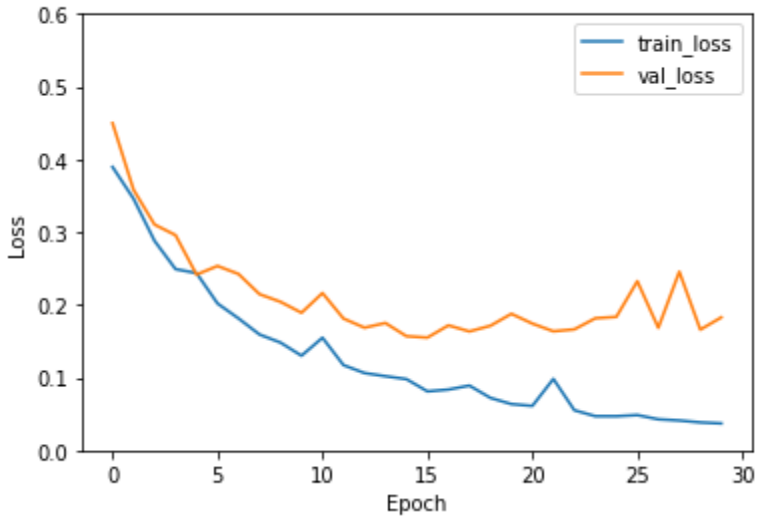

Figura 6: Gráfico de pérdida y épocas

Además, se llegaba a niveles mucho más altos de precisión para el conjunto de validación, de hasta un 95%. Esta fue la red que se utilizó finalmente.

Arquitectura de la red neuronal

Antes de hablar de la arquitectura de la red neuronal, mencionaremos la aumentación de los datos que se realizó.

Se tomó todo el conjunto de entrenamiento y se le aplicó un flip horizontal, para aumentar la cantidad de datos de entrenamiento al doble.

Esto generó ciertas preocupaciones, puesto que el tórax humano no es completamente simétrico, por ejemplo, el corazón está un poco hacia la izquierda. Sin embargo, considere que el beneficio de aumentar los datos valdría la pena, además, el conjunto de validación no se aumentó, para que aun así se pueda ver si el modelo está prediciendo correctamente.

La arquitectura de la red neuronal usada fue la siguiente (esta se mostró previamente en la sección anterior):

```
simpleCNN = Sequential()
simpleCNN.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(256,256,1)))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
simpleCNN.add(Dropout(0.25))
simpleCNN.add(MaxPooling2D(pool_size=(2, 2)))
#
simpleCNN.add(Flatten())
simpleCNN.add(layers.Dense(64, activation='relu'))
simpleCNN.add(Dropout(0.35))
simpleCNN.add(layers.Dense(1, activation='sigmoid'))
```

Figura 6: Arquitectura de la red

El proceso de cómo se llegó a esta red se explicó en la sección anterior, sin embargo, se explicará porqué se usó cada parte.

Se usaron 4 capas convolucionales con 64 filtros cada una. Se llegó a ese número final luego de ensayo y error probando distintas cantidades (32, 64, 128). 32 era más rápido,

pero no lograba extraer suficiente información, y 128 demoraba demasiado tiempo en ejecutarse.

Se usaron 4 layers para ir extrayendo features a medida que iban progresando los datos por la red, una vez más, se usaron solo 4 layers porque brindaban más accuracy que 2, pero usar mas hacia que el modelo se demorara demasiado en ejecutarse.

Tampoco se observó una mejora notable al probar con 6 layers, se sospecha que esto es por el uso de pooling.

El pooling fue usado para ir reduciendo la dimensionalidad de los datos sin perder información relevante.

Era necesario tener regularización, para esto, se usó dropout. Se había considerado usar ruido, pero se optó por dropout, puesto que fue visto en clases y en práctica, y es muy sencillo de incorporar en la red. Se usó 25% luego de probar con distintos valores, 40% y 35% mostraron curvas erráticas de pérdida.

También hay una capa densa de 64 neuronas, esta se usa para hacer la clasificación usando la información de las features analizadas en las capas convolucionales. Se usaron 64 por las razones descritas en la sección anterior.

La función de activación escogida fue ReLU, puesto que es no-lineal y funcionó bien a lo largo de toda la implementación del programa. No se vieron problemas de exploding o vanishing gradient.

También se presentan los callbacks y el optimizador usado.

```
early_s_p = tf.keras.callbacks.EarlyStopping(monitor = "val_loss", min_delta = 0.010, patience = 10, restore_best_weights=True)
#definimos un modelo con early stopping con paciencia y un min delta
simpleCNN.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
#usamos binary_crossentropy porque estamos clasificando un caso de 0 o 1
history = simpleCNN.fit(trainImages, trainLabels, epochs=100, validation_data=(valImages, valLabels), callbacks = [early_s_p])
```

Figura 7: Optimizador, callbacks y fit

En la última capa se usa una función de activación sigmoidea, esto porque usamos binary_crossentropy.

Se optó por binary_crossentropy porque estamos clasificando si pertenece o no a una clase (covid o no covid).

También se usó early stopping, puesto que si bien se ejecutaba por 100 épocas, más allá de la época 15 usualmente había sobre-ajuste y no se mejoraba nada.

Adam se usó luego de experimentar con SGD y obtener resultados no deseables. En las primeras épocas no cambiaba nada la validación ni la pérdida, y tampoco se llegaba a una accuracy alta.

Se uso batch size default.

Resultados

La métrica que más se analizó a medida que se fue desarrollando el programa fue la función de pérdida. Esta nos da una clara idea de cómo se va progresando el modelo a lo largo de las épocas, y nos permite ver si aprende bien.

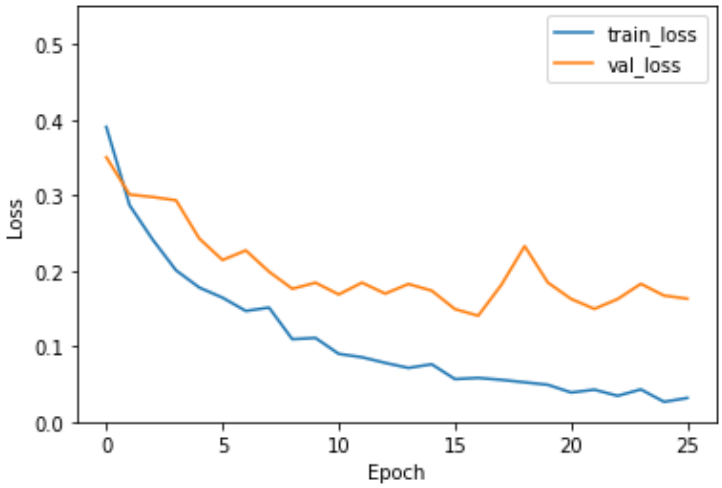

Figura 8: Función de pérdida

Podemos notar que se genera el cruce de ambas curvas como se espera usualmente, y que ambas pérdidas van disminuyendo hasta llegar a la época 15, donde se alcanza el mínimo para la pérdida de la validación. Se continua ejecutando el programa hasta la época 25, puesto que el early stopping tiene paciencia configurada a 10 épocas. Se obtuvo una pérdida de 0.149 en la época 17.

Nos damos cuenta que el modelo se detiene antes que ocurra sobreajuste, y se guardan los mejores pesos de la red neuronal.

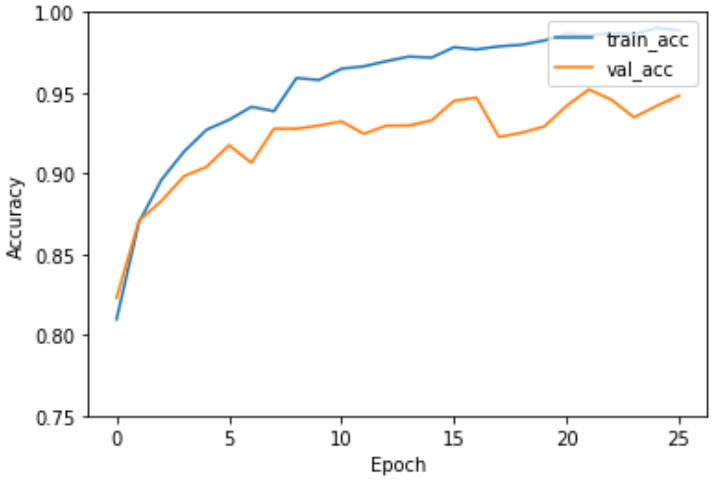

Figura 9: Accuracy

Con respecto a la accuracy, también nos damos cuenta que va aumentando según lo esperado, y que las curvas se van separando cada vez más, esto es por un leve sobreajuste a los datos, sin embargo, se evita que se sobreajuste demasiado debido al early stopping y la regularización implementada. Se alcanzó una accuracy de 94.5% en la época 17.

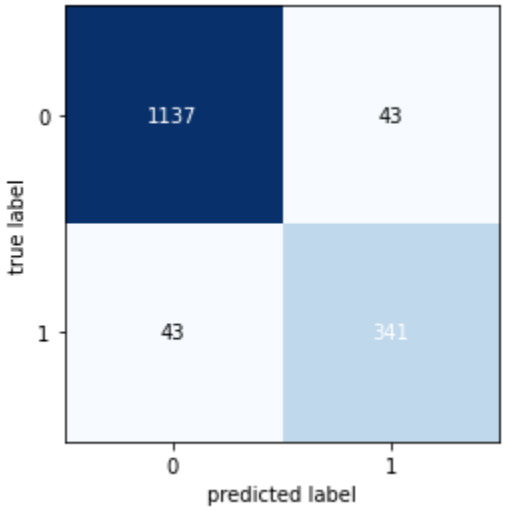

Figura 10: Matriz de confusión.

Ya sabemos que la precisión no indica todo con respecto al desempeño de una red. En este caso, podemos ver que la red neuronal entrega 43 falsos negativos y falsos positivos, lo cual es bastante bajo.

Sería deseable reducir más la cantidad de falsos negativos, puesto que estos son más importantes que los falsos positivos. A pesar de esto, el desempeño de la red es bastante bueno y muestra buen rendimiento con el conjunto de validación.

El f1 score obtenido, que fue calculado utilizando la ecuación mostrada a continuación, fue de 0.888, lo cual está en línea con lo visto en la matriz de confusión, la red se desempeña bastante bien, sin embargo, el hecho de que entregue falsos negativos es poco deseable.

$$f1 = 2 * (precision * recall)/(precision + recall)$$

$$accuracy = (TN + TP)/(TN + FP + TP + FN)$$

$$precision = TP/(TP + FP)$$

Conclusiones

La facilidad de implementar las redes neuronales profundas muestra porque se han popularizado tanto en los últimos años, sin embargo, esto requiere una gran capacidad de cómputo, que está fuera del alcance de la mayoría, a menos que se usen plataformas externas como colab.

Se intentó instalar tensor flow en el computador local, lo cual no fue exitoso, actualmente tengo una tarjeta gráfica RX580, lo cual implica que instalar tensorflow no iba a ser tan straight-forward, puesto que se requiere una versión específica de Ubuntu LTS, y también el proceso de instalación era bastante complicado, puesto que eran muchos requerimientos y ROCm no es fácil de instalar. No poder ejecutar localmente la red dificulto el progreso, puesto que era lento y no se podía usar aceleración de GPU constantemente.

Se alcanzó una accuracy bastante alta a pesar de tener una red bastante sencilla, esto muestra la efectividad de las redes neuronales en las tareas de clasificación.

Sería deseable seguir experimentando con redes neuronales, teniendo más conocimiento del trabajo que realiza cada capa en específico.

La red genera bastantes falsos negativos, lo cual no es deseable para un detector de enfermedades.

Como una apreciación personal, me fue bastante difícil avanzar en este proyecto, puesto que he tenido fuertes síntomas secundarios por la vacuna, sobretodo un dolor muy fuerte en el hombro.

Referencias

- https://www.youtube.com/watch?v=HBi-P5j0Kec
- https://medium.com/analytics-vidhya/confusion-matrix-accuracy-precision-recall-f1-score-ade299cf63cd
- https://stackoverflow.com/questions/58960358/generate-a-filename-in-python
- https://stackoverflow.com/questions/62654715/how-to-read-images-dataset-in-google-colab-for-deep-learning
- https://stackoverflow.com/questions/48285129/saving-best-model-in-keras/48924351
- https://github.com/as4401s/COVID-19-X_ray-image-classification/blob/master/DenseNet121.ipynb

- https://www.pyimagesearch.com/2019/10/14/why-is-my-validation-loss-lower-than-my-training-loss/
- https://appdividend.com/2020/12/02/python-join-list/
- https://stackoverflow.com/questions/48735600/file-download-from-google-drive-to-colaboratory
- https://github.com/faizancodes/COVID-19-X-Ray-Classification