Proyecto 5

1st Barrios, Alonso
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
alonso.barrios@utec.edu.pe

2nd Rodriguez, Mauricio
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
mauricio.rodriguez@utec.edu.pe

3rd Tenazoa, Renzo
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
renzo.tenazoa@utec.edu.pe

***Abstract*—Este proyecto está desarrollado para poner a prueba lo aprendido acerca de redes neuronales convolucionales. El objetivo del proyecto es crear una red neuronal capaz de clasificar un *set* de datos de *testing* para determinar si el pulmon tiene Covid, *Lung Opacity*, *Viral pneumonia* o si es normal. La red neuronal deberá ser entrenada con un *set* de datos de *training* del mismo *dataset*.**

Index Terms*—redes neuronales, cnn, error, *COVID-19 Radiography

I. Introducción

Con el objetivo de poner a prueba la red neuronal se utilizó el *dataset* COVID-19 Radiography, el cual contiene un conjunto de 21165 imágenes, entre las que se encuentran: Covid, *Lung Opacity*, *Viral pneumonia* y normal. Se buscó que el modelo pueda clasificar de manera correcta entre los distintos tipos de pulmones. Además se mostrarán las gráficas de errores de cada caso.

II. Explicación

Para construir los modelos de CNN se utilizaron las funciones implementadas por la librería de *torch* en Python. Todos los modelos contienen los siguientes elementos en común

Dado el módulo torch.nn que se nombrará solo como nn En primer lugar, para generar un capa

- **nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding):** Aplica una convolución 2d a los valores de entrada.
- **nn.ReLU()** Aplica una función de activación RELU sobre el modelo convolucionado.
- **nn.MaxPool2d(kernel_size, stride):** Aplica un *max pooling* 2d para reducir las dimensiones, en este caso las reduce a la mitad.
- **nn.Sequential()** Este método permite generar un módulo, en este caso, una capa que ejecutará de forma secuencial las funciones pasadas como parámetros y guardará los resultados.

Luego de haber generado los capas necesarios aprovechando las funciones mencionadas. Se guarda también como atributo del modelo una instancia de la función nn.Linear(). Esta permite tener *fully connected layers*.

Finalmente, cada modelo CNN tiene hace uso de la función **forward(x)**. Esta función solo recibe x, la data inicial, como parámetro. La función, como el nombre lo dice, se encargará de realizar el proceso de *forwarding* de la red neuronal.

III. Experimentos

Se realizaron experimentos con distintas implementaciones de CNN, cada una con distintas características de forma que se pueda apreciar la utilidad de ellas.

A. CNN

Para este experimento se tomó el mismo modelo usado para clasificar números y se obtuvo la siguiente tabla de errores.

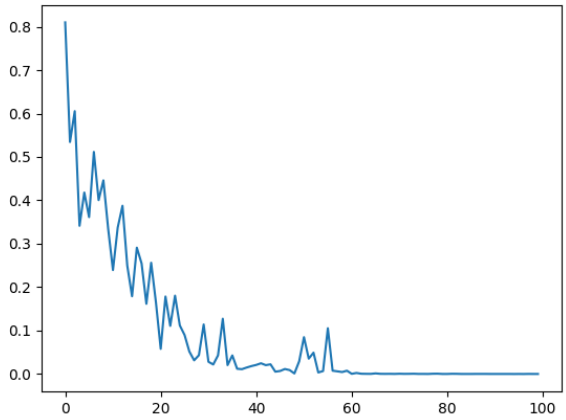

Fig. 1. Error CNN con 2 capas

Como se puede observar, el error va bajando hasta llegar a un valor muy cercano a 0. Además, al pasar el *test_loader* por el modelo ya entrenado se obtuvo un *accuracy* de 95.39%.

B. CNN con 5 capas

Para este experimento, se incrementó la cantidad de capas del modelo estándar. En cada capa, la cantidad de canales se incrementa. Siendo que inicialmente la cantidad de canales es 1 y al terminar el procesamiento de la 5ta capa la cantidad de canales es de 512.

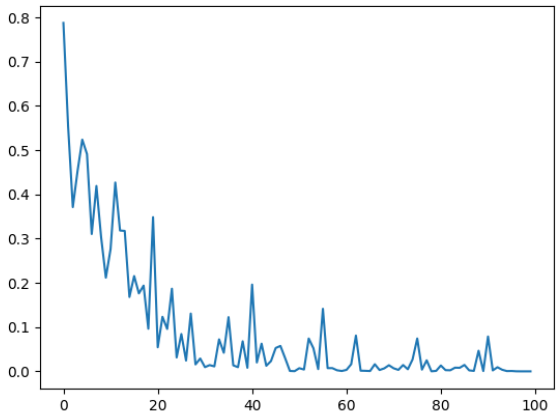

Fig. 2. Error en CNN estándar multilayer

Como se puede observar, el error va bajando hasta incluso llegar a un valor de 0. Además, al pasar el *test_loader* por el modelo ya entrenado se obtuvo un *accuracy* de 97.42%. Cabe resaltar que, a comparación del modelo anterior, consumió mucha menos memoria RAM y demoró menos tiempo en entrenar (aproximadamente 1 minuto por época).

C. CNN con batch normalization en cada capa

Este experimento, al igual que el modelo estándar, cuenta con 2 capas en las que se obtiene luego de las convoluciones una salida de 32 canales. Este experimento utiliza *batch normalization* en ambas capas. El *batch normalization* se aplica luego del proceso de convolución.

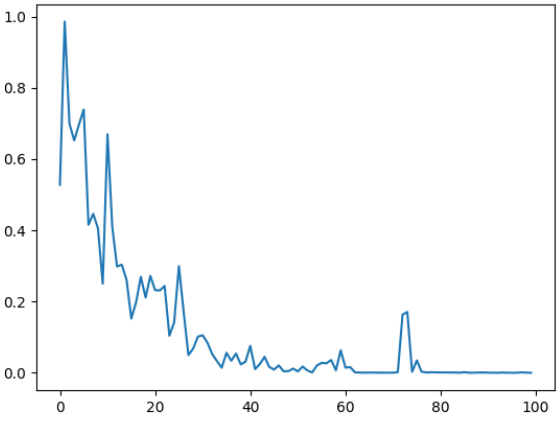

Fig. 3. Error en CNN con batch normalization por capa

Como se puede observar, al igual que los modelos anteriores, el error va bajando hasta llegar a valores cercanos a 0. Además, al pasar el *test_loader* por el modelo ya entrenado se obtuvo un *accuracy* de 95.05%.

D. CNN con batch normalization y dropout en cada capa

Este modelo también cuenta con 2 capas. Utiliza *batch normalization* en cada capa luego de realizar el proceso de convolución y posteriormente, luego de realizar el *max pooling* de la capa se define un *dropout(0.5)*, es decir, se define que la probabilidad para que una neurona se ”apague” o tenga sus valores en 0 es del 50%

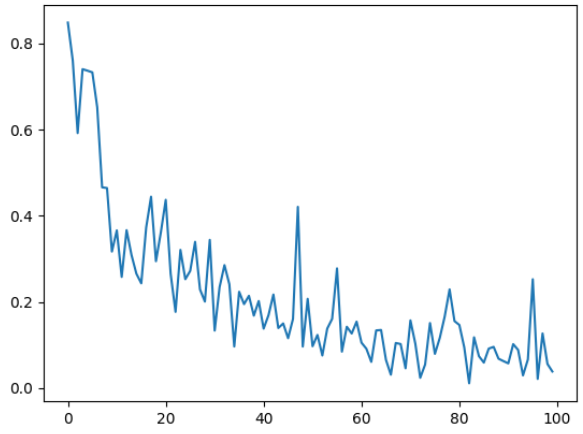

Fig. 4. Error en CNN con batch normalization y dropout por capa

En la gráfica se observa que el error es altamente volátil y a comparación de los anteriores modelo el error mínimo se encuentra lejos de ser 0. Además, al pasar el *test_loader* por el modelo ya entrenado se obtuvo un *accuracy* de 93.42%, siendo este modelo el *accuracy* más bajo que se tuvo.

E. CNN con batch normalization y dropout solo en la primer capa

Este modelo es similar al anterior. Sin embargo, en lugar de aplicar *batch normalization* y establecer un *dropout* en todas las capas solo se realiza esto en la primera.

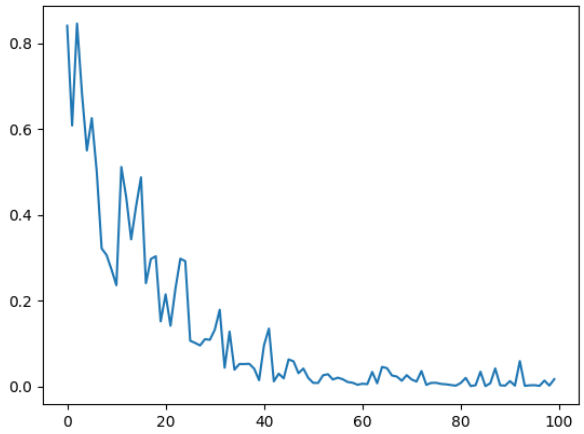

Fig. 5. Error en CNN con batch normalization y dropout: primera capa

En la gráfica se observa que, a diferencia del modelo anterior que es el más similar, el error ya no es tan volátil, incluso el error sí se acerca a 0. Además, al pasar el *test_loader* por el modelo ya entrenado se obtuvo un *accuracy* de 94.85%,

habiendo una ligera mejora con respecto al modelo más parecido.

Conclusiones

- Durante la experimentación, al ejecutar cada uno de los modelos CNN, se observó que el modelo que menos tiempo demoró en ejecutarse fue el modelo CNN con 5 capas.
- Durante la experimentación se pudo observar que el mejor modelo es el que cuenta con 5 capas, teniendo un *accuracy* de más del 97%. Siendo este uno de los más altos, incluyendo las implementaciones hechas en *kaggle*.
- El uso de VRAM de la GPU para cada modelo fue de aprox. 6GB, a diferencia del modelo de 5 capas que fue solo de 4GB aproximadamente.
- Durante la experimentación se pudo observar que el "peor" modelo es el que cuenta con *batch normalization* y *dropout* en cada capa.
- El tiempo promedio que tomó la fase de *training* entre todos los modelos es de 3 horas sin contar el modelo de 5 capas.
- El tiempo que tomó el entrenamiento en el modelo de 5 capas fue de menos de 1 hora y usó menos memoria VRAM que los demás modelos.
- Se intentó entrenar el modelo en Khipu, sin embargo el tiempo que tomaba era similar, posiblemente no se estuvieron usando todos los recursos disponibles.

Repositorio

Enlace al repositorio de Github:
https://github.com/alonso804/ai-project-5

References

[1] Neural networks: ¿de qué son capaces las redes neuronales artificiales? (2020, 20 octubre). IONOS Digitalguide. https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/que-es-una-neural-network/.

[2] Rosebrock, A. (2021). PyTorch: Training your first Convolutional Neural Network (CNN) - PyImageSearch. PyImageSearch. Retrieved 6 December 2021, from https://www.pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/.

[3] torch.nn — PyTorch 1.10.0 documentation. Pytorch.org. (2021). Retrieved 6 December 2021, from https://pytorch.org/docs/stable/nn.html.

[4] Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." stat 1050 (2016): 23