Proyecto 6

1st Ian Augusto Cortez Gorbalan
ian.cortez@utec.edu.pe
970895554

2nd Plinio Matías Avendaño Vargas
plinio.avendano@utec.edu.pe
927144823

I. Introducción

Este proyecto busca utilizar un *autoencoder* basado en CNN con el propósito de transformar imágenes de baja a alta resolución. Para este se va a utilizar un *dataset* compuesto por fotografías de ambas categorías anteriores de diferentes objetos. El objetivo principal del proyecto es desarrollar un modelo de *autoencoder* que sea capaz de producir imágenes de alta resolución en base a una imagen de baja resolución. Como objetivos secundarios, se tiene generar imágenes creadas por el modelo a través del espacio latente y mostrar las 10 mejores, asimismo determinar la pérdida producida por el entrenamiento y validación del modelo junto con el error promedio para el *dataset* de *testing* y *validation*.

II. Explicación del Modelo empleado

A. Autoencoders

Los *autoencoders* son un tipo de red neuronal que se entrenan con el principal objetivo de generar un espacio latente o vector Z, también conocido como *code*, el cual es de una dimensión sustancialmente menor al input, en el que se espera que se mantengan las características suficientes para representar correctamente la entrada. Su arquitectura se compone de dos partes, un *encoder* y un *decoder*. El *encoder* se encarga de reducir el *input* y producir el vector Z. El *decoder* se encarga de reconstruir el *input* utilizando el vector Z, para esto realiza el proceso inverso al *encoder*.

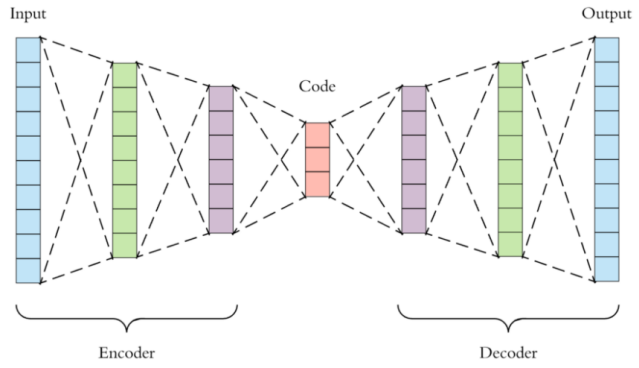

Fig. 1: *Autoencoder*

1) Convolutional Autoencoders: Un *Convolutional Autoencoder* es un *autoencoder* basado en la arquitectura de un CNN. El componente del *encoder* funciona de forma similar que un CNN, donde se utilizan capas *hidden* que contienen una función convolución, seguida por una función de activación y de un *max pooling* para reducir la dimensionalidad de la imagen y obtener el *code*. El componente del *decoder* realiza un *up convolution*. *Up convolution* o *deconvolution* es el proceso inverso a una convolución donde se extrae un elemento y mediante un matriz de pesos, se multiplica el valor por esta propagando el mismo a una matriz nueva de mayores dimensiones.

$$W = \begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \\ W_7 & W_8 & W_9 \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$$

$$W \times C_1 = \begin{bmatrix} (W_1 * C_1) & (W_2 * C_1) & (W_3 * C_1) \\ (W_4 * C_1) & (W_5 * C_1) & (W_6 * C_1) \\ (W_7 * C_1) & (W_8 * C_1) & (W_9 * C_1) \end{bmatrix}$$

Se repite el proceso anterior para cada elemento de la matriz C. Esto permite aumentar las dimensiones de C y obtener C' e ir reconstruyendo la imagen de entrada del *encoder*. Al construir C' utilizando el *kernel*, habrá un momento donde se sobrepongan las matrices resultantes, en esta situación se suman los valores de los elementos que se sobrepongan.

$$W = \begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \\ W_7 & W_8 & W_9 \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$$

$$C' = \begin{bmatrix} C'_1 & C'_2 & (W_3 * C_1 + W_1 * C_2) & C'_4 & C'_5 \\ C'_6 & C'_7 & (W_6 * C_1 + W_4 * C_2) & C'_9 & C'_{10} \\ C'_{11} & C'_{12} & (W_9 * C_1 + W_7 * C_2) & C'_{14} & C'_{15} \\ C'_{16} & C'_{17} & C'_{18} & C'_{19} & C'_{20} \\ C'_{21} & C'_{22} & C'_{23} & C'_{24} & C'_{25} \end{bmatrix}$$

2) Generación de imágenes a través del espacio latente: Al comprimir las imágenes al espacio latente, se espera que este almacene toda la información relevante, por lo que solo conservaremos las características significativas, es debido a lo anterior que, si graficamos los puntos correspondientes a cada imagen en el plano n-dimensional, es lógico que las fotografías similares se agrupen o se encuentren cerca euclidianamente, por lo que podemos asumir que las imágenes de la misma categoría se distribuyen normalmente, por consiguiente, para generar nuevas fotografías solo basta generar un valor aleatorio adentro de la campana.

III. Experimentos

A. Modificación y separación del dataset

Previa a la experimentación, se agregaron imágenes al *dataset* de fotografías de alta definición y con un *script* de Python se redujo la calidad de estas y se añadieron al grupo de baja resolución, cabe destacar que estas fotografías fueron elegidas manualmente para satisfacer las condiciones de la data original. Posteriormente, se separó el *dataset* con 85% de los elementos para entrenamiento y 15% de los elementos para validación y testing. Contando con un total de 955 elementos.

B. Definición de hiper-parámetros

El *batch size* se establecerá en 32 para todos los experimentos, esto es debido a que un tamaño mayor agota la VRAM de la tarjeta gráfica, de igual manera el *learning rate* (0.001) sera constante, sumado a estos el número de épocas se estableció en 40 y el *size* del espacio latente en 200.

C. Primer experimento

Se empleo una arquitectura de cuatro capas.
Primera capa:
-Kernel size 7, stride 2, padding 1, output de 32 canales.
Segunda capa:
-Kernel size 6, stride 2, padding 1, output de 64 canales.
Tercera capa:
-Kernel size 4, stride 2, padding 1, output de 128 canales.
Cuarta capa:
-Kernel size 3, stride 2, padding 1, output de 256 canales.

Todas con funcion de activacion Relu, el decoder exactamente igual pero a la inversa, se obtuvo resultados bastante malos,

Dataset de Prueba	*Loss*
Testing	0.3244
Validation	0.3549
Training	0.3631

Por lo que podemos concluir que el modelo cae en un *underfitting*.

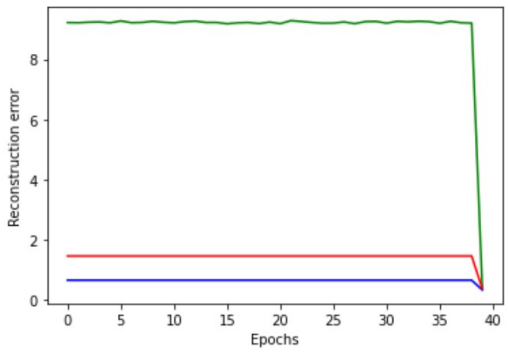

Fig. 2: Gráfica del experimento 1

Al intentar generar una imagen, resulta en una textura negra, sin relación visual alguna con la original.

D. Segundo experimento

Se empleo una arquitectura de cuatro capas, con la adición de un *maxpool* en la última capa, además en esa capa el *stride* de la convolución se redujo a 1.
Primera capa:
-Kernel size 7, stride 2, padding 1, output de 32 canales.
Segunda capa:
-Kernel size 6, stride 2, padding 1, output de 64 canales.
Tercera capa:
-Kernel size 4, stride 2, padding 1, output de 128 canales.
Cuarta capa:
-Kernel size 3, stride 1, padding 1, output de 256 canales.
-Maxpool, kernel size 3 y stride 2.

Todas con función de activación Relu, el *decoder* exactamente igual pero a la inversa, se obtuvo una mejora respecto al resultado anterior,

Dataset de Prueba	*Loss*
Testing	0.25409
Validation	0.256645
Training	0.282136

Por lo que, podemos concluir que el modelo aun cae en un *underfitting*, sin embargo el *max pooling* contribuyo a una mejora significativa,

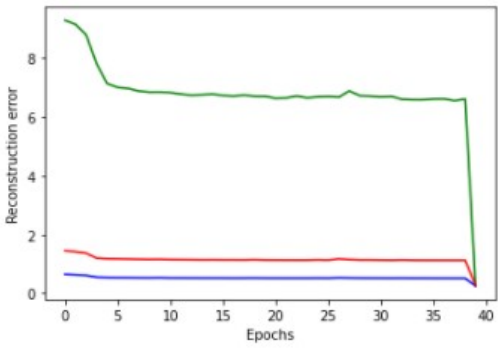

Fig. 3: Gráfica del experimento 2

Como podemos observar en la siguiente imagen el modelo, ya reconoce texturas y formas básicas.

Fig. 4: Imagen original

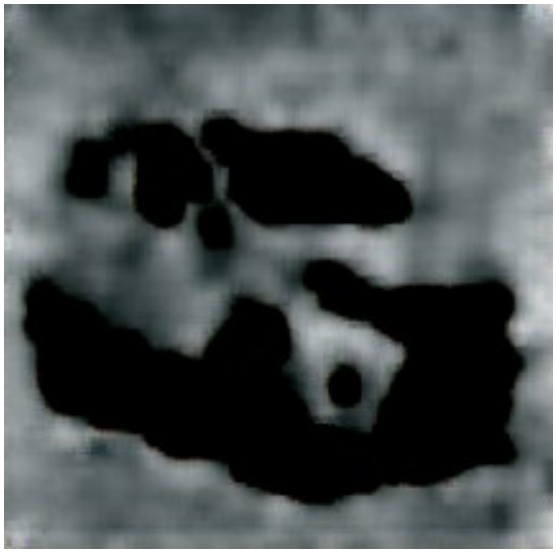

Fig. 5: Imagen generada basada en la distribución de Z

E. Tercer experimento

Se empleo la misma arquitectura, con diferencia que se agrego *batch normalization* en todas las capas.
Se obtuvo una gran mejora respecto al resultado anterior,

Dataset de Prueba	*Loss*
Testing	0.062432
Validation	0.065119
Training	0.043038

Por lo que, podemos concluir que el modelo aun cae en un *underfitting*, sin embargo el *batch normalization* significo una mejora exponencial ya que ahora reconoce colores, esto es debido a que es probable que el modelo se halla visto afectado por el problema de *vanishing gradient*, además podemos notar como es esperado que la gráfica es más suave.

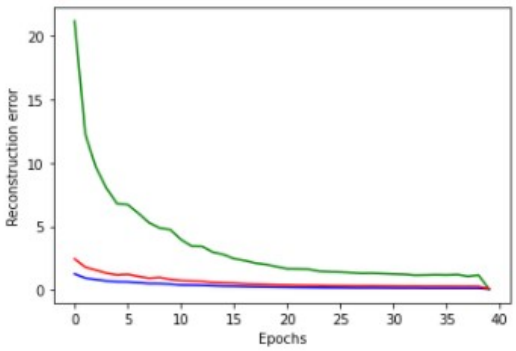

Fig. 6: Gráfica del experimento 3

Como podemos observar en la siguiente imagen el modelo, ya reconoce colores, en adición a lo anterior.

Fig. 7: Imagen original

Fig. 8: Imagen generada basada en la distribución de Z

F. Cuarto experimento

Se agrego dos capas más con la finalidad de aumentar la complejidad, resultando en la siguiente arquitectura.
Primera capa:
-Kernel size 7, stride 2, padding 1, output de 32 canales.
Segunda capa:
-Kernel size 6, stride 2, padding 1, output de 64 canales.
Tercera capa:
-Kernel size 4, stride 2, padding 1, output de 128 canales.
Cuarta capa:
-Kernel size 4, stride 1, padding 1, output de 256 canales.
Quinta capa:
-Kernel size 4, stride 1, padding 1, output de 512 canales.
Sexta capa:
-Kernel size 3, stride 1, padding 1, output de 1024 canales.
-Maxpool, kernel size 3 y stride 2.

Cabe destacar que todas las capas poseen *batch normalization*, este modelo resulto menos adecuado que el anterior, aunque por bastante poco.

Dataset de Prueba	*Loss*
Testing	0.0913
Validation	0.0921
Training	0.0593

Por lo que, podemos concluir que el modelo aun no es lo suficientemente óptimo.

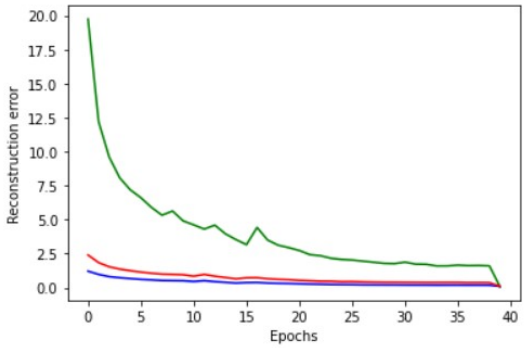

Fig. 9: Gráfica del experimento 4

Ya que esta arquitectura no supera la anterior, seguiremos probando con la arquitectura del experimento 3.

G. Quinto experimento

Bajo la arquitectura del experimento 3, se agregará un capa más de maxpooling, resultando en lo siguiente.
Primera capa:
-Kernel size 7, stride 1, padding 1, output de 32 canales.
-Maxpool, kernel size 3 y stride 2
Segunda capa:
-Kernel size 5, stride 2, padding 1, output de 64 canales.
Tercera capa:
-Kernel size 4, stride 2, padding 1, output de 128 canales.
Cuarta capa:
-Kernel size 3, stride 1, padding 1, output de 256 canales.
-Maxpool, kernel size 3 y stride 2.

Dataset de Prueba	*Loss*
Testing	0.0535
Validation	0.055373
Training	0.034454

Se observo una mejora respecto al modelo anterior, por lo que en el siguiente experimento se le agregara a cada función de convolución una de pool.

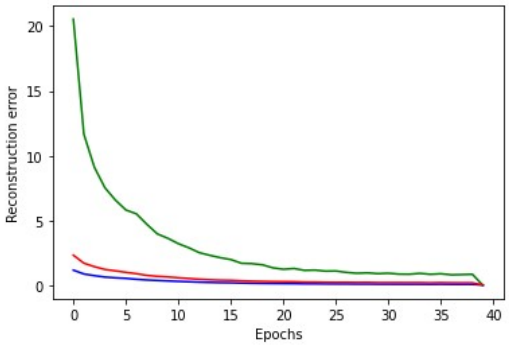

Fig. 10: Gráfica del experimento 5

Como podemos observar en las siguientes imágenes, el modelo reconoce detalles más pequeños como la placa las texturas y formas más suaves.

Fig. 11: Imagen original

Fig. 12: Imagen generada basada en la distribución de Z

H. Sexto experimento

Bajo la arquitectura del experimento 5, se le agregara a cada función de convolución una de pooling, adicionalmente en la última capa se obviara el padding.

Primera capa:
-Kernel size 7, stride 1, padding 1, output de 32 canales.
-Maxpool, kernel size 3 y stride 2

Segunda capa:
-Kernel size 6, stride 1, padding 1, output de 64 canales.
-Maxpool, kernel size 2 y stride 2.

Tercera capa:
-Kernel size 4, stride 1, padding 1, output de 128 canales.
-Maxpool, kernel size 2 y stride 2.

Cuarta capa:
-Kernel size 3, stride 1, padding 0, output de 256 canales.
-Maxpool, kernel size 2 y stride 2.

Dataset de Prueba	*Loss*
Testing	0.0479
Validation	0.0498
Training	0.026318

Se observo una mejora respecto al modelo anterior, en todos los datasets.

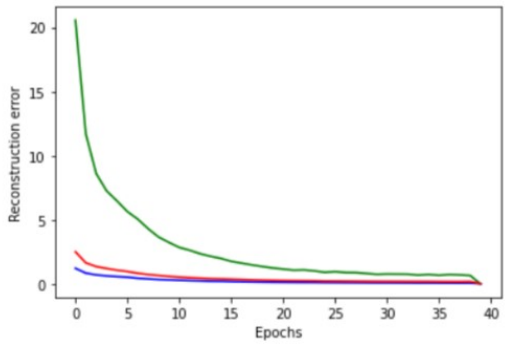

Fig. 13: Gráfica del experimento 6

Como podemos observar en las siguientes imágenes, el modelo reconoce prácticamente todos los colores, junto a la silueta de la persona.

Fig. 14: Imagen original

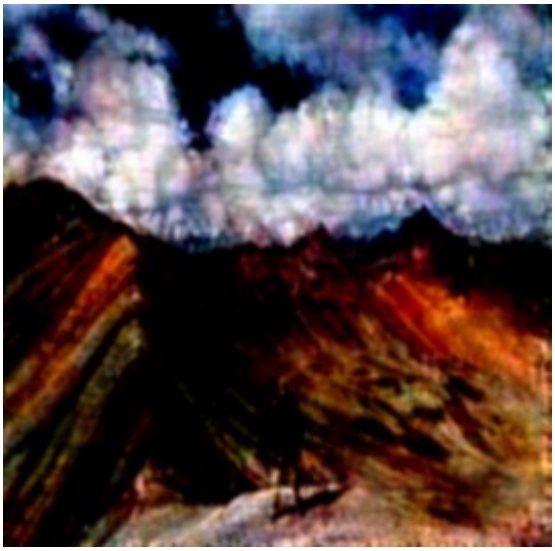

Fig. 15: Imagen generada basada en la distribución de Z

I. Séptimo experimento

La única diferencia con respecto a la anterior arquitectura fue el empleo de funciones sigmoideas a excepción de la ultima capa en la que se empleo Relu, esto modelo resulto en un aumento del error, pero muy pequeño, casi ínfimo.

Dataset de Prueba	*Loss*
Testing	0.051005
Validation	0.055532
Training	0.026902

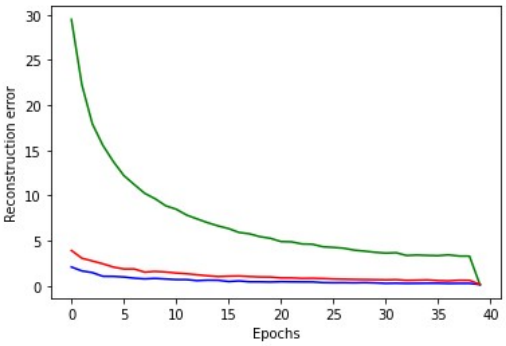

Fig. 16: Gráfica del experimento 7

J. Octavo experimento

Se empleo la misma arquitectura que el sexto experimento, sin embargo se amplio el espacio latente a 500.

Dataset de Prueba	*Loss*
Testing	0.04805
Validation	0.050138
Training	0.025751

Se observa, una mejora respecto a los datos de entrenamiento, sin embargo, el error aumenta en validation y testing, por lo que existe un ligero overfitting.

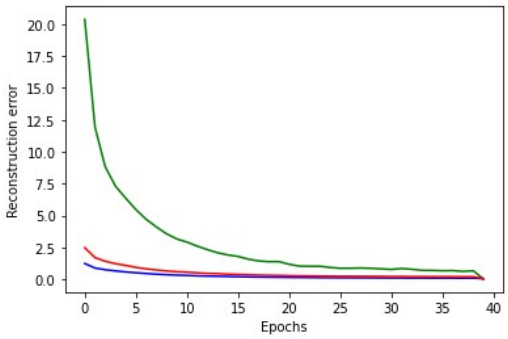

Fig. 17: Gráfica del experimento 8

K. Noveno experimento

Se aplico dropout al 0.15, en las dos primeras capas, esto resulto en una reducción de la calidad respecto al modelo anterior.

Dataset de Prueba	*Loss*
Testing	0.0977
Validation	0.1046
Training	0.0847

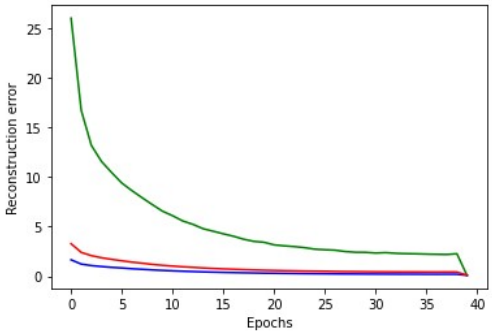

Fig. 18: Gráfica del experimento 10

L. Generación de imágenes

Emplearemos el modelo del experimento 6 ya que es el que tiene la menor pérdida, toda estas imágenes son basadas en el vector Z de los datasets de validation y testing.

Fig. 19: Figura generada por el modelo

Fig. 20: Figura generada por el modelo

Fig. 21: Figura generada por el modelo

Fig. 22: Figura generada por el modelo

Fig. 23: Figura generada por el modelo

Fig. 24: Figura generada por el modelo

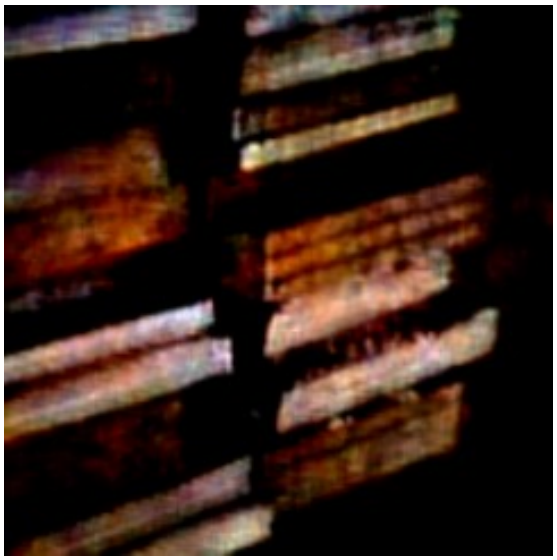

Fig. 25: Figura generada por el modelo

Fig. 26: Figura generada por el modelo

Fig. 27: Figura generada por el modelo

Fig. 28: Figura generada por el modelo

IV. Conclusión

Después de realizar varias pruebas, se ha identificado que la arquitectura empleada en el experimento 6 ha sido la mejor y que nos ha permitido obtener el menor error. Asimismo, se toma este experimento como el mejor resultado ya que el error disminuye significativamente en todos los dataset tanto *training*, *testing* y *validation*.

Se debe destacar que esta arquitectura ha tenido estos buenos resultados porque se ha utilizado varias capas de *pooling* y *batch normalization*. Se llegó a utilizar el segundo porque demostró una gran mejora sobre el comportamiento visible en la gráfica de errores para los conjuntos de datos probados, esto se evidencia en el experimento 3. Asimismo, se utiliza esta cantidad de capas debido a que como se mostró en el experimento 4, tener más de 4 capas aumenta el error sobre la arquitectura empleada. Finalmente cabe destacar que utilizar *Max Pooling* en cada capa nos da una mayor capacidad para extraer con más detalles los elementos de cada una de las imágenes utilizadas para el entrenamiento.

References

[1] Dertat, A. (2018, 21 junio). Applied Deep Learning - Part 3: Autoencoders - Towards Data Science. Medium. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[2] Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep Learning with Pytorch: Build, Train, and Tune Neural Networks Using Python Tools. Manning Publications.

[3] Alberti, M. (2018, 14 mayo). Introducción al autoencoder. DeepLearningItalia.