Redes Neuronales - Práctico 3

Luis Biedma

10 de marzo de 2021

Ejercicio 1

Para resolver el ejercicio se implementó el autoencoder feed-forward convolucional usando el paquete Keras (junto con Tensorflow), al igual que se hizo con el práctico anterior. Dividimos la estructura en codificador y decodificador, para separar un poco y que no quede tan larga la descripción.

Para el **codificador**, tenemos la siguiente estructura:

1. Capa de entrada común con input que mantiene la forma de la imagen (28×28).
2. 16 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
3. Capa de Max Pooling 2×2, con el mismo padding.
4. 8 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
5. Capa de Max Pooling 2×2, con el mismo padding.
6. 16 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
7. Capa de Max Pooling 2×2, con el mismo padding.

Después de aplicar el codificador, tendremos que nuestros inputs se transformaron en representaciones $4 \times 4 \times 8$

Para el **decodificador**, haremos uso de las Capas UpSampling2D de Keras, que facilitan el trabajo de realizar la transformación inversa de los resultados de aplicarle el codificador a los inputs. Tenemos la siguiente estructura:

1. 8 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
2. Capa de UpSampling 2×2, con el mismo padding.
3. 8 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
4. Capa de UpSampling 2×2, con el mismo padding.

5. 16 convoluciones 3×3, con función de activación RELU, con un padding que nos mantiene el tamaño de la imagen.
6. Capa de UpSampling 2 × 2, con el mismo padding.
7. Capa de Salida: Una Convolución 2D 3 × 3, con función de activación sigmoide y mismo padding

El dataset utilizado es MNIST, que puede ser obtenido directamente desde Keras con la función $keras.datasets.mnist.load\_data()$. Nuestros conjuntos tienen 60000 imágenes de entrenamiento y 10000 de test o validación.

Para poder realizar comparaciones más justas entre este autoencoder y el del trabajo anterior, vamos a utilizar la función de pérdida (loss) del error cuadrático medio. Para hacer un poco más fácil el trabajo del entrenador, se optó por dividir todos los valores de los píxeles por el máximo posible (255), para tener valores entre 0 y 1 para nuestros inputs.

Para mantenernos en los mismos parámetros que en el trabajo anterior, debemos optimizar con Stochastic Gradient Descent (SGD) con la combinación Nesterov, learning rate de 1 y un momentum de 0.5, con 50 épocas de entrenamiento. Además, se utilizó el mismo batch_size que antes. A continuación, se muestran algunos gráficos de diferentes dígitos aleatorios del conjunto de test y su correspondiente transformación al ser pasados por el autoencoder (arriba: original, abajo: transformado). A modo cualitativo, se puede ver que el autoencoder funciona bien.

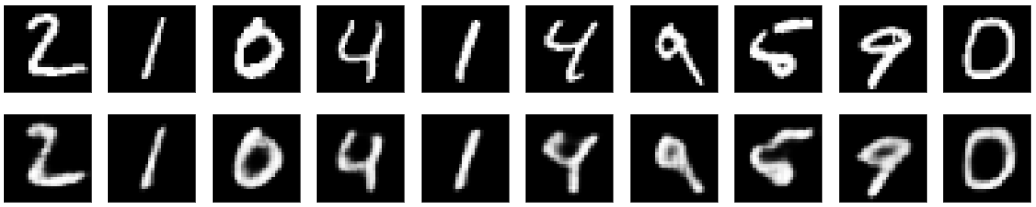

Figura 1: Comparación de dígitos aleatorios. Arriba original, abajo transformación.

Se puede visualizar en el avance de la función de pérdida que la misma tiene un gran decaimiento al comienzo y luego comienza a bajar de a poco. Se decidió parar en 50 épocas para mantenernos al mismo nivel que las últimas pruebas del trabajo anterior.

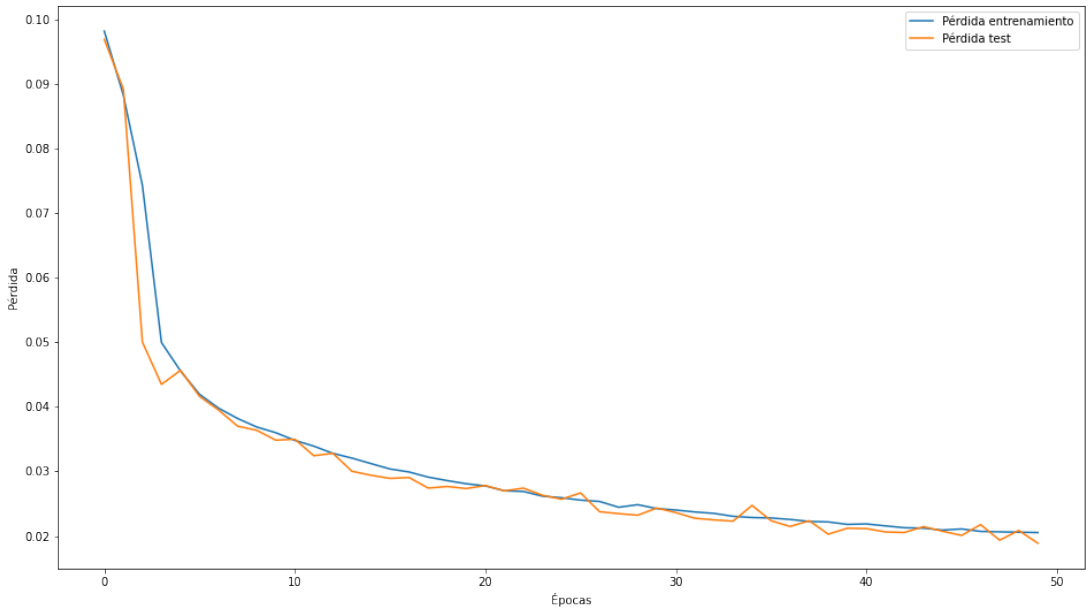

Figura 2: Pérdida (error cuadrático medio) en función de las épocas de entrenamiento

A continuación, se compara la performance de nuestro autoencoder convolucional con la del autoencoder con 512 neuronas intermedias, que fue el mejor modelo entrenado en el práctico anterior (con diferencias casi nulas con sus competidores), a través de dos figuras, que comparan el rendimiento en el conjunto de entrenamiento y en el de test, respectivamente.

Se puede ver que, utilizando la escala logarítmica en el eje Y, conseguimos una mejora del doble en nuestro error cuadrático medio, para los conjuntos de entrenamiento y para los de test. Esto nos deja ver que nuestro autoencoder convolucional es mejor, pero a costa de un tiempo de entrenamiento mayor.

Los tiempos de entrenamiento de las redes son de 28 segundos para la red convolucional y de 10 segundos para la red feedforward con 512 neuronas en la capa oculta. Para el entrenamiento de ambos modelos, se utilizó una GPU NVIDIA Geforce RTX 3060Ti, con la versión 2.4 de Tensorflow en un ambiente local.

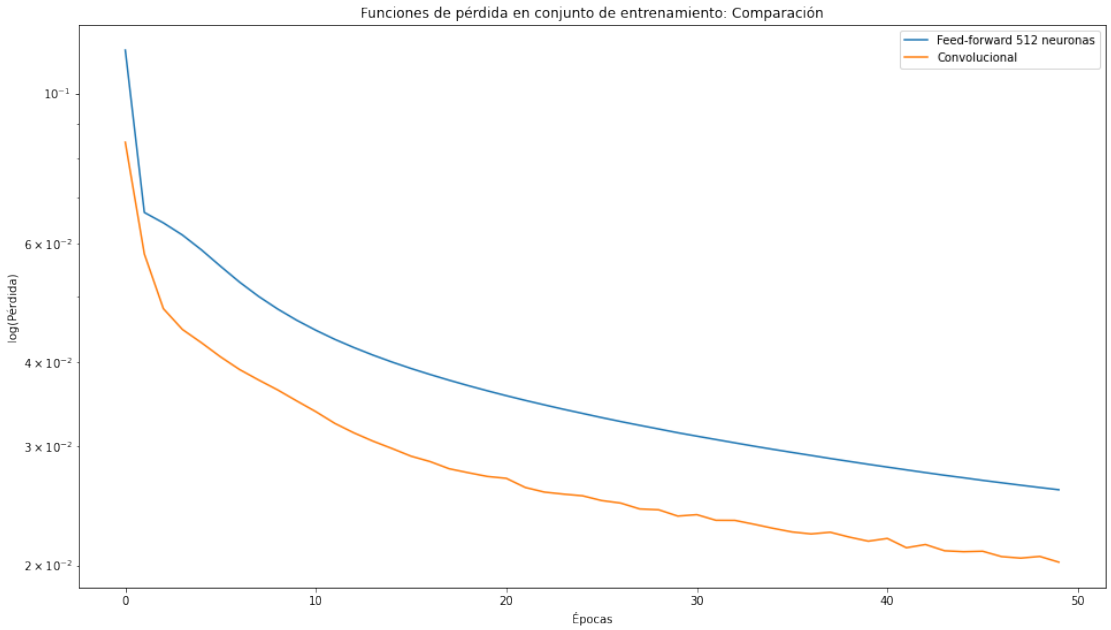

Figura 3: Pérdida en logaritmo en función de las épocas de entrenamiento

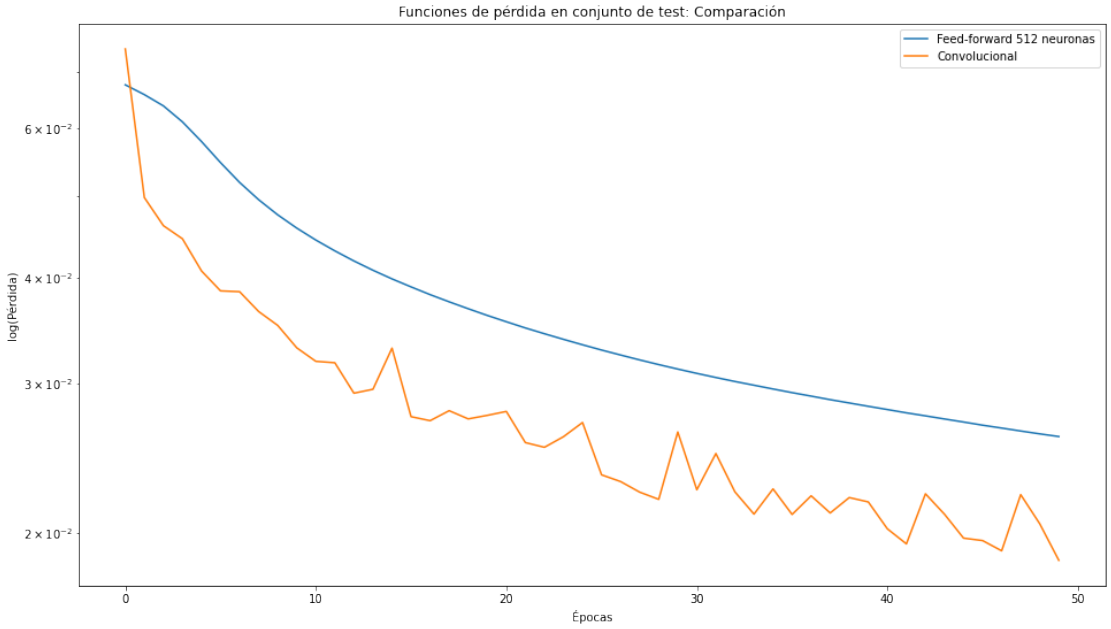

Figura 4: Pérdida en logaritmo en función de las épocas de entrenamiento (conjunto test)