Comisión Nacional de Energía Atómica

Instituto Balseiro

UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

APRENDIZAJE PROFUNDO Y REDES NEURONALES ARTIFICIALES

Práctica N°4

Introducción a Keras

ALUMNA Sol Micaela Maldonado Betanzo
DNI 38799660
CARRERA Maestría en Ingeniería

11 de octubre de 2020

Índice

Ejercicio 1

En este ejercicio, se desea utilizar el dataset *Boston* de *sklearn* que contiene datos sobre los precios de los inmuebles en Boston. Con los mismos, se quiere entrenar un modelo de regresión lineal que permita predecir el precio de los inmuebles. Para tal fin, se utilizan ~25% de los datos para *testing* y ~75% de los datos para training. La red neuronal propuesta es la siguiente:

Tipo de capa	Cantidad de neuronas	Función de activación
Dense	1	lineal

Se estableció como función de costo *Adam* con un *learning rate* de 0.005, y la función de costo *mean squared error*. El *batch size* se determinó en 2, y se corrieron 20 *épocas.*

La *loss* obtenida es la siguiente:

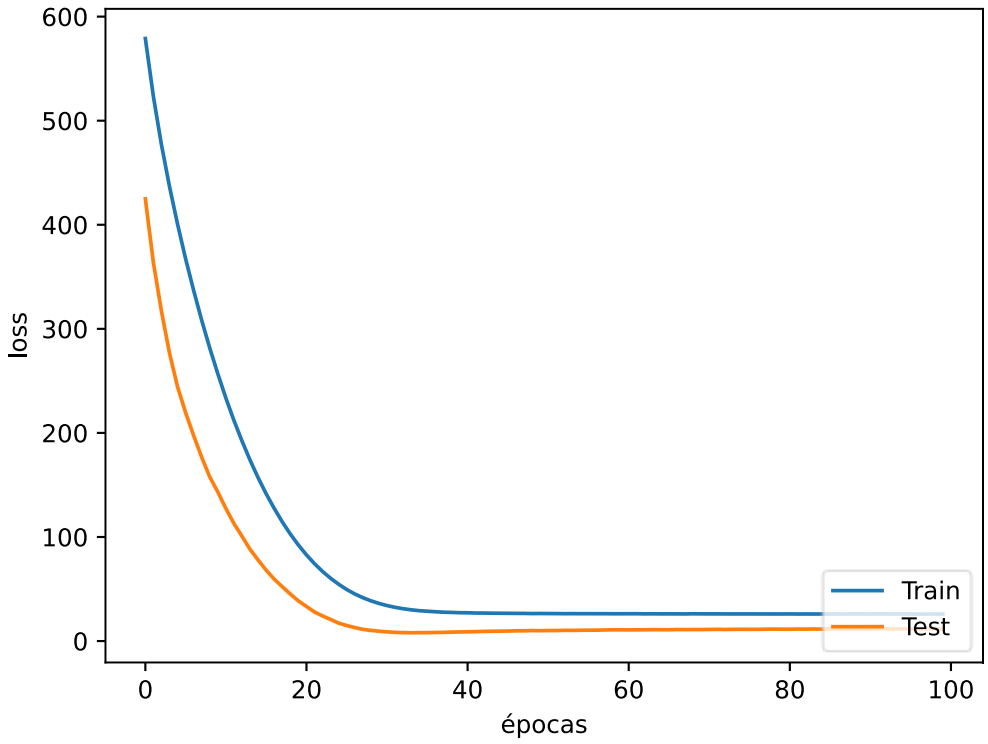


Figura 1: *loss del dataset Boston.*

Finalmente, se graficó la regresión lineal obtenida para los precios reales con respecto a los predichos, obteniendo el siguiente resultado:

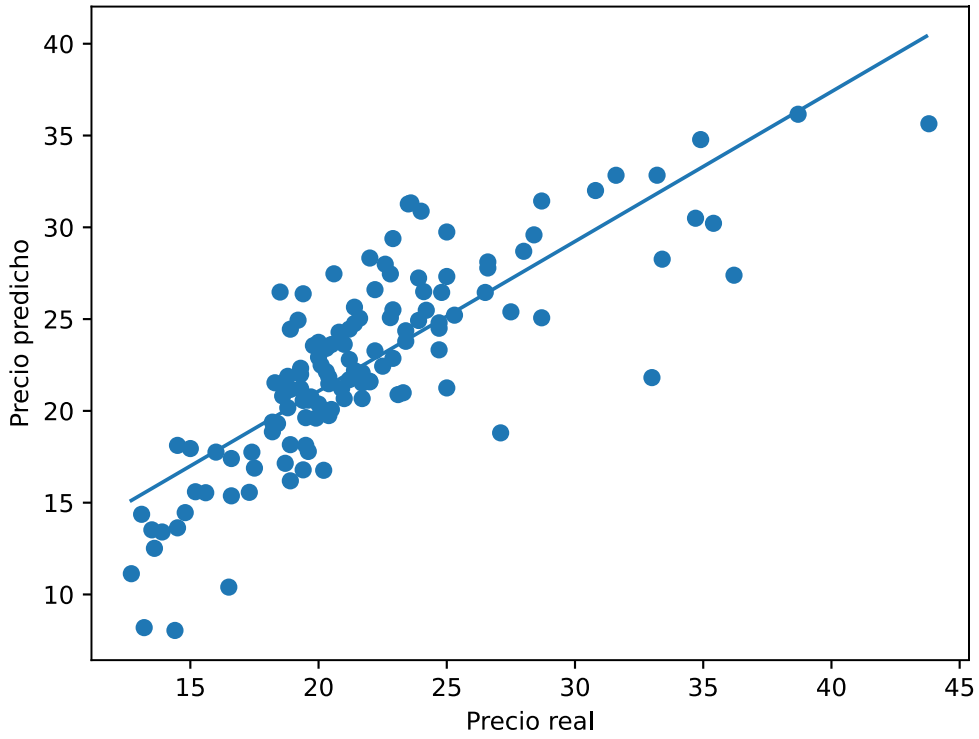


Figura 2: *regresión lineal de precio real con respecto al predicho de dataset Boston.*

El R^2 dio un valor de 0.66.

Ejercicio 2

Este ejercicio se puede dividir en dos partes. El objetivo de la primera, es clasificar las imágenes del dataset de CIFAR-10 utilizando como funciones de costo *mean squared error, support vector machine* y *softmax*. En el siguiente cuadro, se resumen las características de la red.

Tipo de capa	Cantidad de neuronas	Función de activación
Dense	100	Sigmoide
Dense	10	Lineal

Se utilizó como optimizador *stochastic gradient descent* con un *learning rate* de 1e-3, y se entrenó durante 50 épocas con un *batch size* de 128. Se muestran los resultados a continuación para las distintas funciones de costo, y al final, una comparación de las distintas *accuracy* obtenidas para cada función de costo para los datos de test.

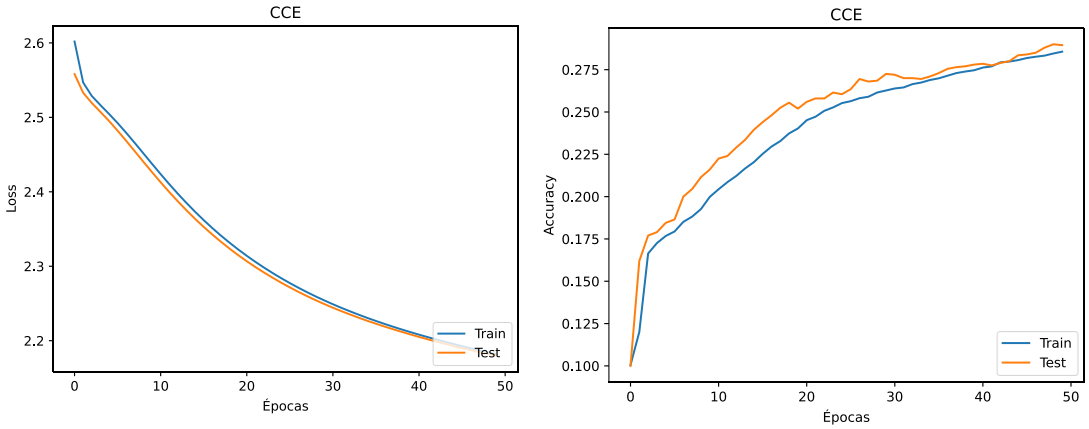


Figura 3: *loss y accuracy para función de costo categorical cross entropy.*

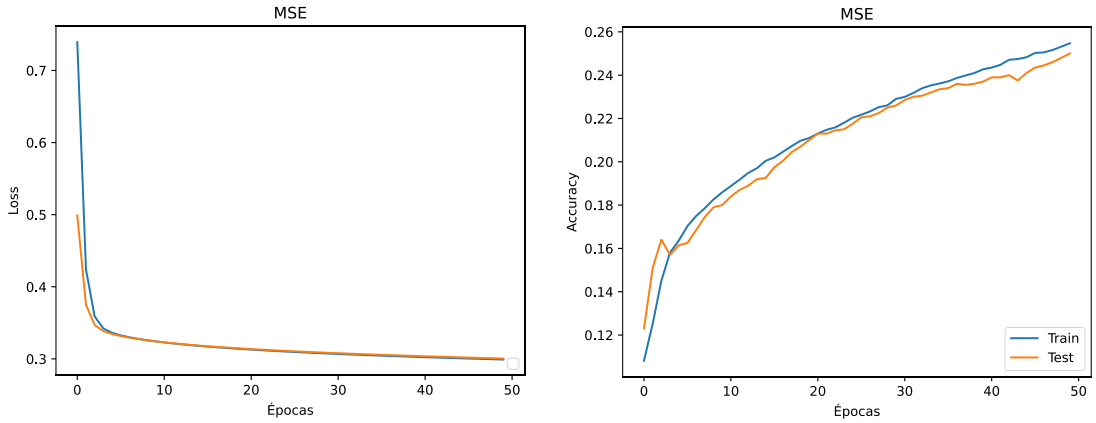


Figura 4*: loss y accuracy para función de costo mean squared error.*

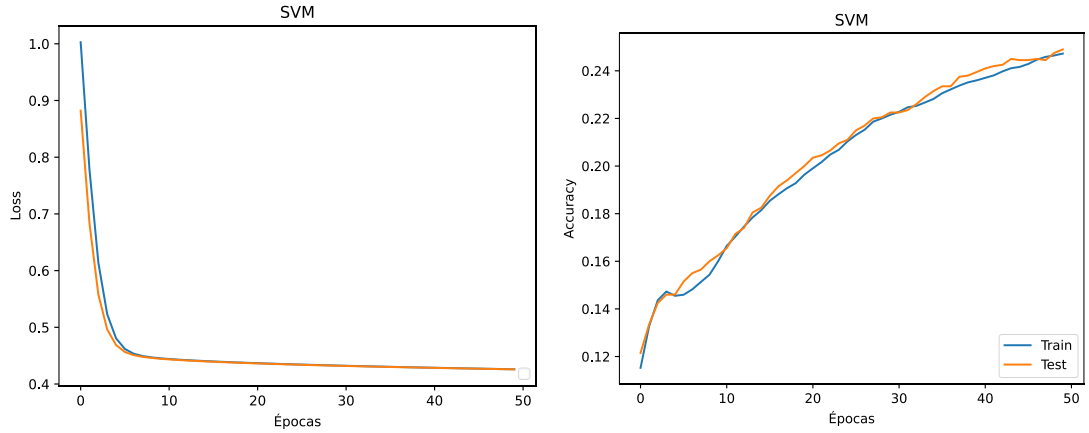


Figura 5: *loss y accuracy para función de costo hinge.*

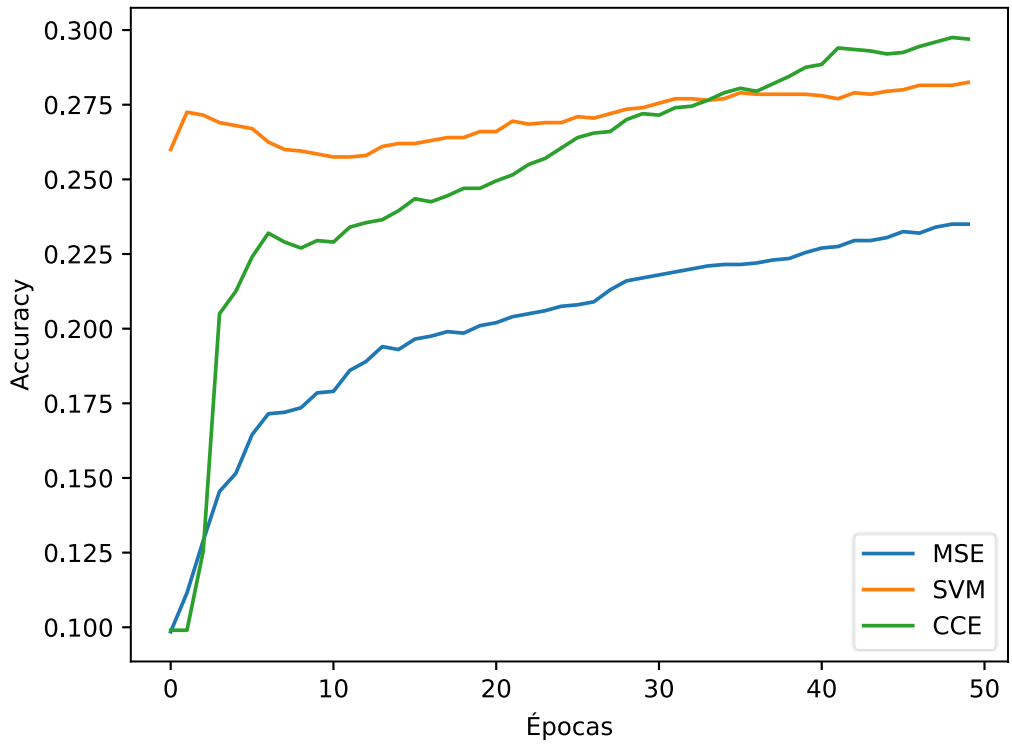


Figura 6: *accuracy para los datos de test para las distintas funciones de costo evaluadas.*

Como se puede observar, la mejor *accuracy* se obtiene con *categorical cross entropy,* seguido de *hinge* y por último *mean squared error.* Nuevamente como el práctico 2, el mejor resultado se obtuvo con CCE, cambiando el orden entre SVM y MSE.

La segunda parte de este ejercicio, consistió en implementar dos arquitecturas para resolver el clasificador XOR, la primera se puede resumir en la siguiente tabla:

Tipo de capa	Cantidad de neuronas	Función de activación
Densa	2	Tangente hiperbólica
Densa	1	Tangente hiperbólica

La segunda arquitectura, es la siguiente:

Tipo de capa	Cantidad de neuronas	Función de activación
Input	2	-
Dense	1	Tangente hiperbólica
Concatenate	-	-
Dense	1	Tangente hiperbólica

Cabe aclarar, que para ambas arquitecturas se debió definir la métrica “XOR_acc”, y se estableció un *learning rate* de 0.5 con el optimizador *stochastic gradient descent*, la función de activación fue *mean squared error* y el *batch size* de 4.

Los resultados se muestran a continuación:

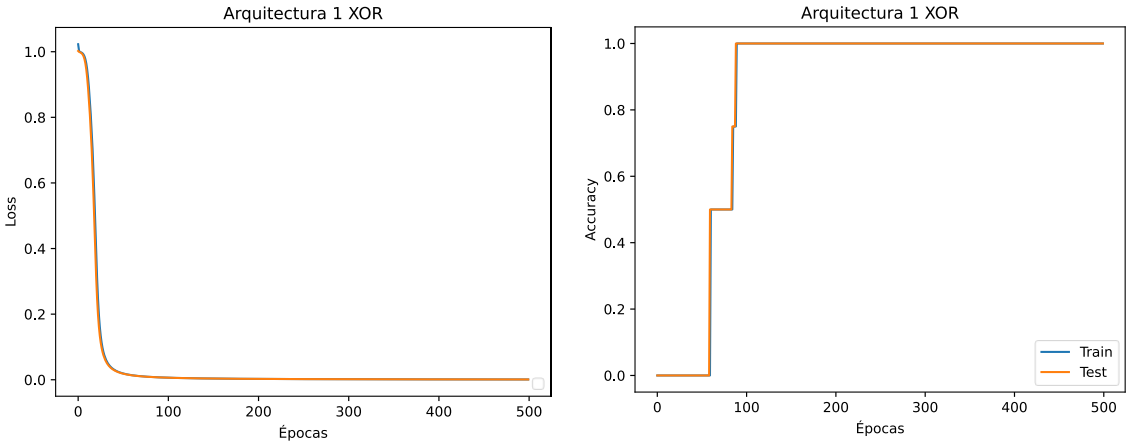


Figura 8: *loss y accuracy para arquitectura 1.*

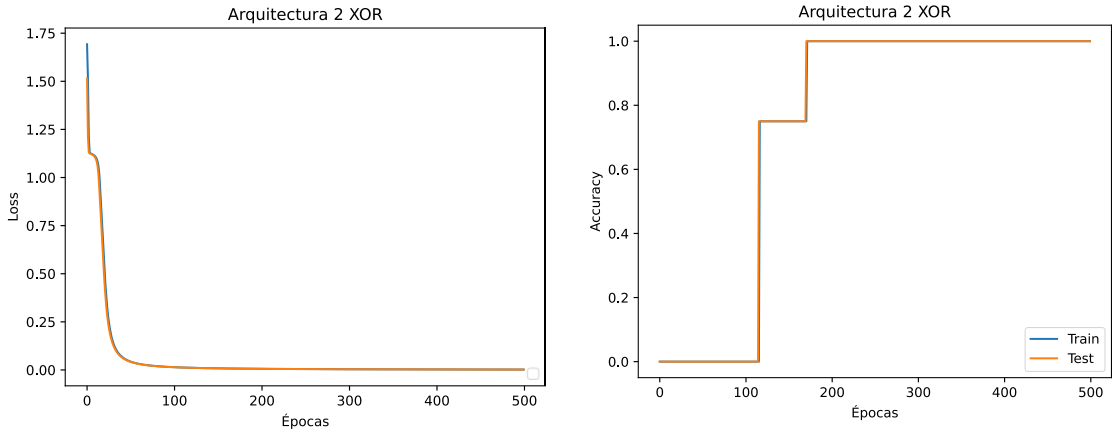


Figura 7: *loss y accuracy para arquitectura 2.*

Como se puede observar, nuevamente con ambas arquitecturas se llega a un *accuracy* de 100% y a una *loss* de 0.

Ejercicio 3

El objetivo de este punto es implementar una red neuronal para clasificar las revisiones de el dataset de Keras IMDB. Para tal finalidad, lo primero que se hizo fue considerar las 10000 palabras más comunes, se realizó un concatenado tanto para los datos (x) como los targets (y), y se realizó un preprocesado. En el mismo, se revisaron las reseñas y si las mismas contenían alguna de las 10000 palabras más comunes, se reemplazó dicho valor por 1, en caso contrario se dejó un 0. Luego, se dividieron los datos y tagets, dejando 40000 para *train* y 10000 para *test*. En primera instancia, no se aplicó ninguna regularización y se implementó la siguiente arquitectura:

Tipo de capa	Cantidad de neuronas	Función de activación
Dense	100	ReLU
Dense	50	ReLU
Dense	1	Sigmoide

Se utilizó el optimizador *Adadelta* con un *learning rate* de 0.1, como función de costo a *mean squared error*, y un *batch size* de 128. Los resultados fueron los siguientes:

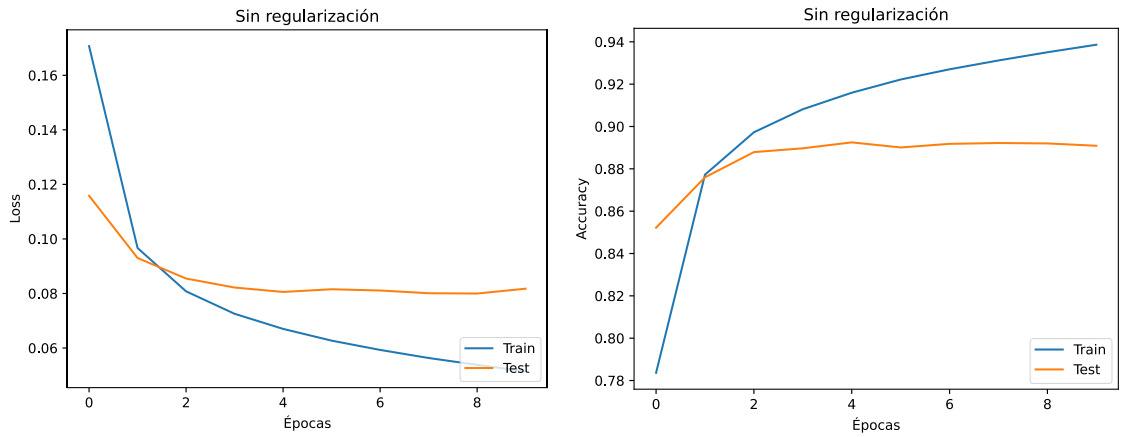


Figura 9: *loss y accuracy sin regularización.*

Como se puede observar, hay un gran *overfitting*, por lo que se procedió a implementar distintas técnicas de generalización. La primera fue *L2* en todas las capas densas con un factor de 1e-2. Los resultados fueron:

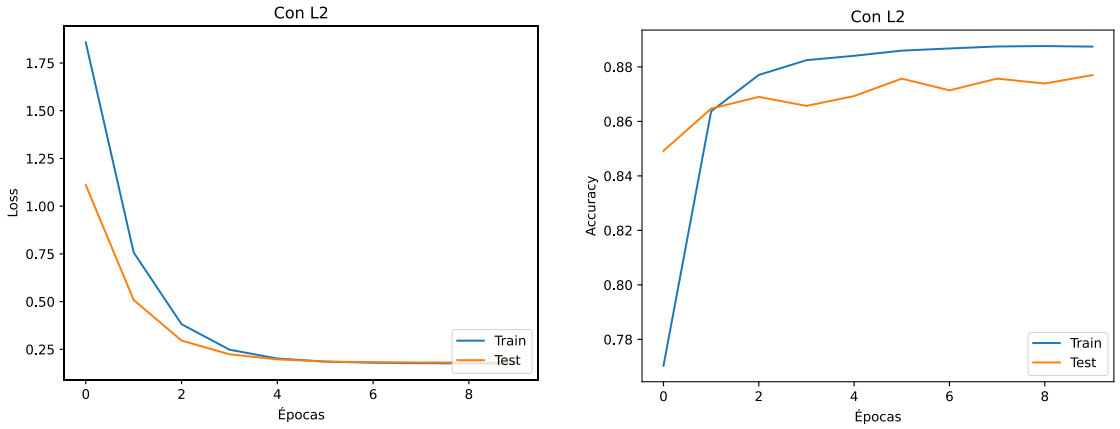


Figura 10: *loss y accuracy con L2.*

Como se observa, hay una gran mejoría con respecto a los resultados obtenidos sin regularización. Luego, se procedió a implementar *Batch Normalization* luego de la primera capa densa y de la capa oculta.

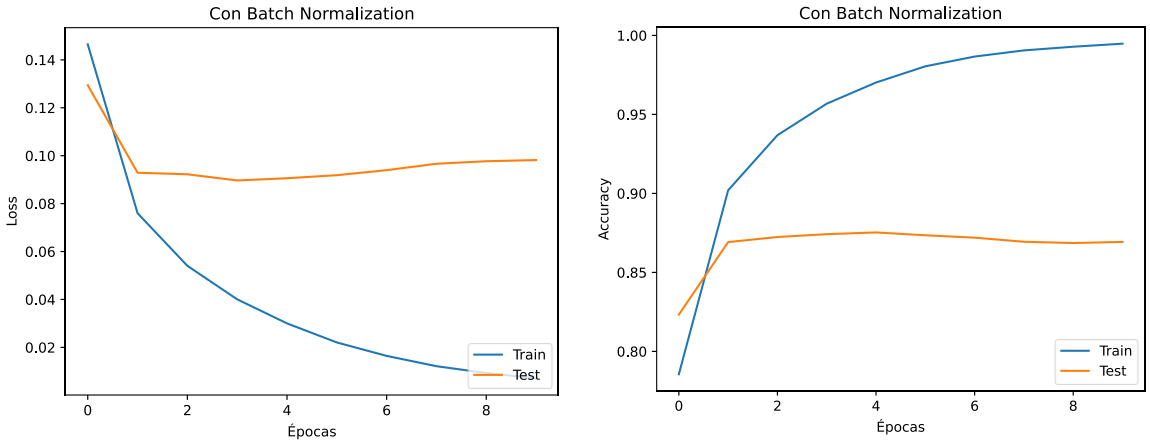


Figura 11: *loss y accuracy con batch normalization.*

En este caso, los resultados empeoran. Este es un resultado esperable considerando que la red neuronal implementada tiene muy pocas capas y esta técnica muestra mejores resultados en redes profundas. Por último, se implementaron dos capas de *Dropout* con un factor de 0.5 luego de la capa de entrada y la capa oculta. Los resultados fueron:

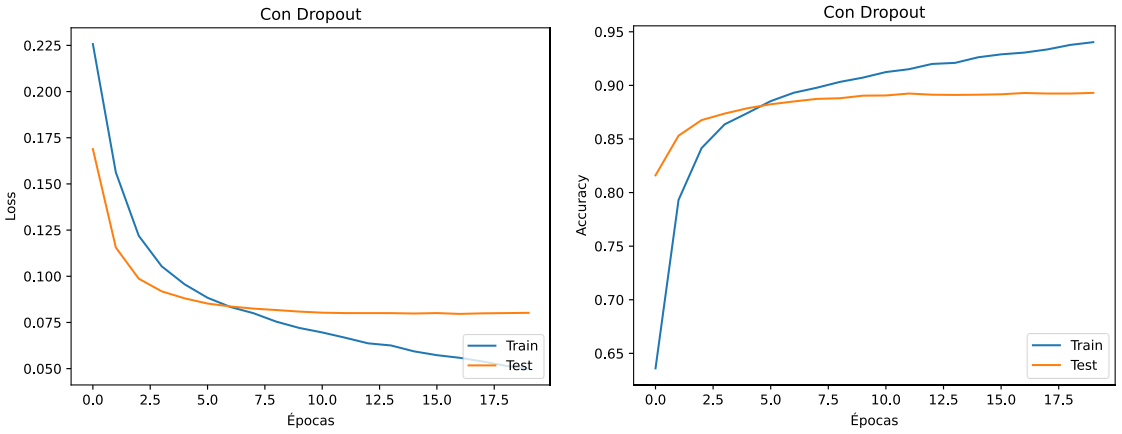


Figura 12: *loss y accuracy con dropout.*

Nuevamente, se observa una mejoría con respecto a sin regularización. Se llega a la conclusión que *L2* presenta mejores resultados con respecto a la generación, y *dropout* también mejora la generalización, llegando a una *accuracy* más alta que *L2*.

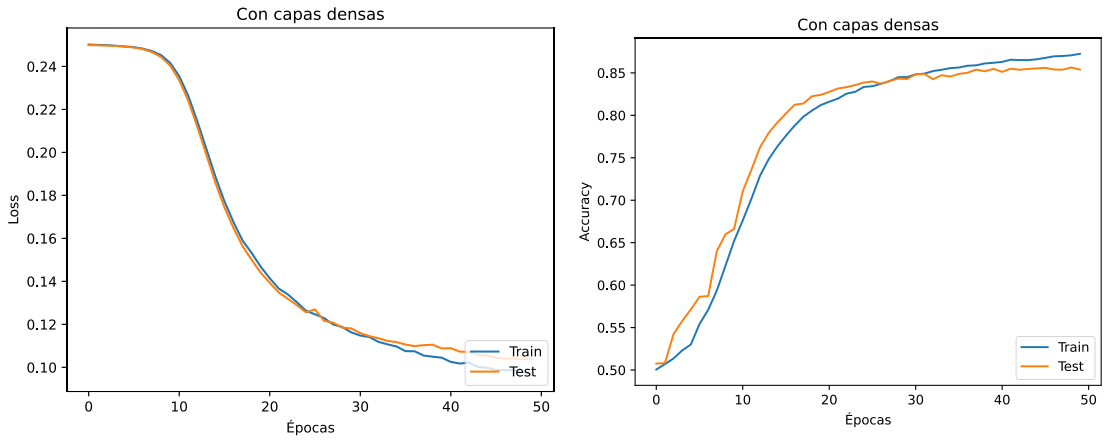
Ejercicio 4

El objetivo de este ejercicio es utilizar el código del anterior utilizando *Embeddings* para representar las palabras de las reseñas en vectores de números reales. Para tal finalidad, se utilizó la capa *Embedding* al inicio de la red, en donde se determinaron 10000 palabras distintas, 15 dimensiones de salida, y 100 palabras por reseña. Fue necesario, realizar previamente un *padding* para que las reseñas que no llegaban a 100 palabras, rellenarlas con 0, y recortar las reseñas de más de 100 palabras a ese máximo.

En primer lugar, se implementó una arquitectura con capas densas, que se resume a continuación:

Tipo de capa	Propiedades de la capa
Embedding	• input_dim: 10000 • output_dim: 15 • input_length: 100
Dropout	• rate:0.5
Dense	• units: 100 • activation: ReLU
Dropout	• rate: 0.5
MaxPooling1D	• pool_size: 2
Flatten	
Dropout	• rate: 0.5
Dense	• units: 1 • activation: sigmoide

Se utilizó como optimización *Adadelta* con un *learning rate* de 0.1, función de costo *mean squared error* y *batch size* de 128.



Como se observa, se obtiene una respuesta con respecto al *overfitting* mejor a las obtenidas en el inciso anterior. Quizá podría haberse inicializado los pesos de manera distinta para evitar la parte inicial sin respuesta, en donde las neuronas se activan luego de 10 épocas. Luego, se implementó la misma arquitectura, cambiando la capa densa de 100 neuronas por la siguiente:

Tipo de capa	Propiedades de la capa
Conv1D	• filters: 32 • kernel_size: 2 • padding: same • activation: ReLU

Los resultados obtenidos fueron los siguientes:

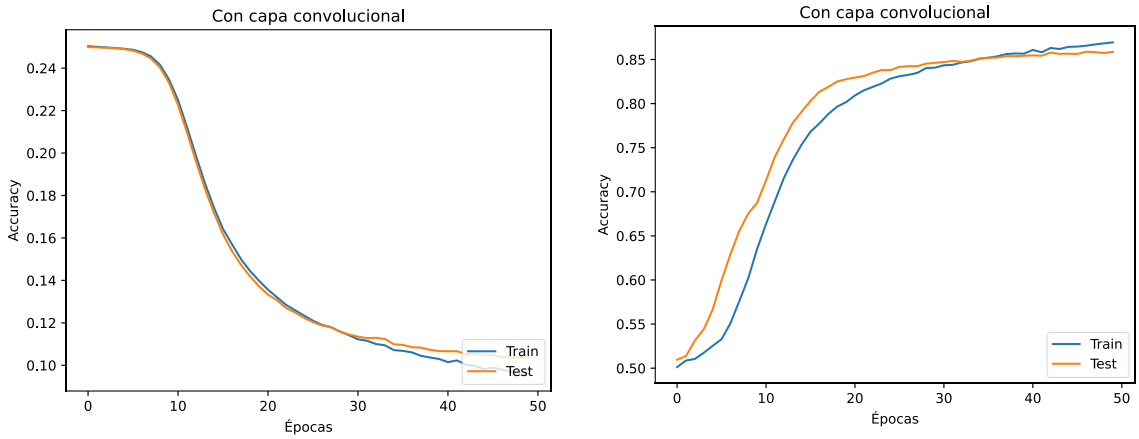


***Figura 13**: loss y accuracy con capa convolucional.*

Como se puede observar, la diferencia es mínima con respecto a la arquitectura anterior, presentándose levemente un mayor *overfitting* en la *accuracy* con la capa convolucional.

En conclusión, si se compara con los resultados obtenidos en el punto anterior, teniendo en cuenta la generalización y la *accuracy*, se obtuvieron mejores resultados utilizando *Embeddings.*

Ejercicio 5

En este ejercicio, se pretende implementar la siguiente arquitectura:

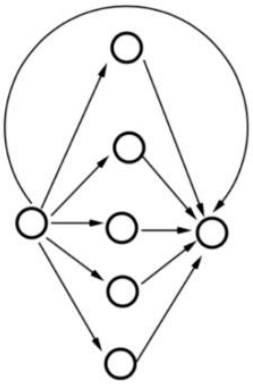


***Figura 14:** arquitectura ejercicio 5.*

Con la misma, se desea resolver el mapeo logístico:

$$x(t+1) = 4x(t)(1-x(t))$$

Para tal finalidad, se generaron 1000 datos, de los cuales se utilizaron 750 para *train* y 250 para *test*. Los datos, debieron tomarse considerando que la muestra sea representativa del conjunto. Los mismos se observan a continuación:

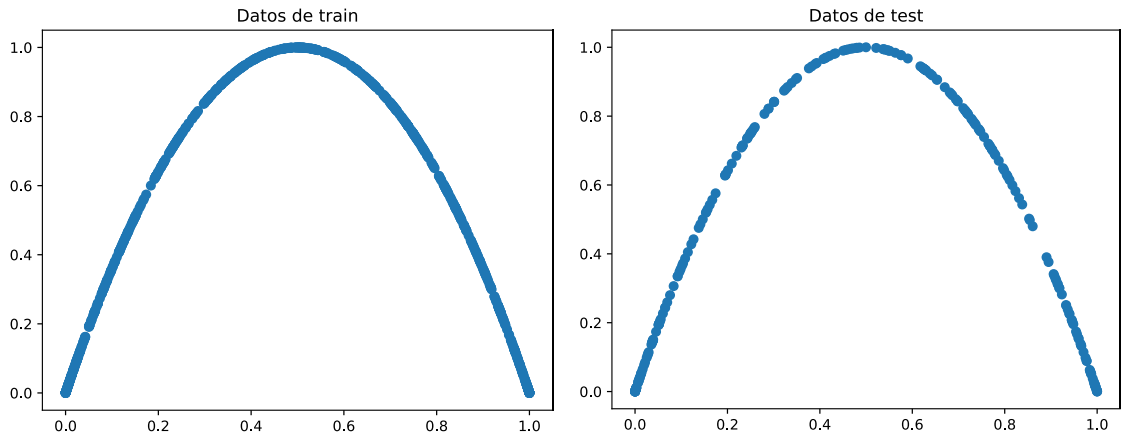


Figura 15: *conjuntos de datos para ejercicio 5.*

Luego, se implementó la siguiente arquitectura:

Tipo de capa	Propiedades de la capa
Input	• shape: [1]
Dense (capa1)	• units: 5 • activation: tangente hiperbólica
Concatenate	• [input,capa1]
Dense	• units: 1 • activation: lineal

El optimizador utilizado fue *stochastic gradient descent* con un *learning rate de 0.1*, y función de costo y métrica *mean squared error.* El *batch size* se determinó en 8, y el resultado obtenido fue el siguiente:

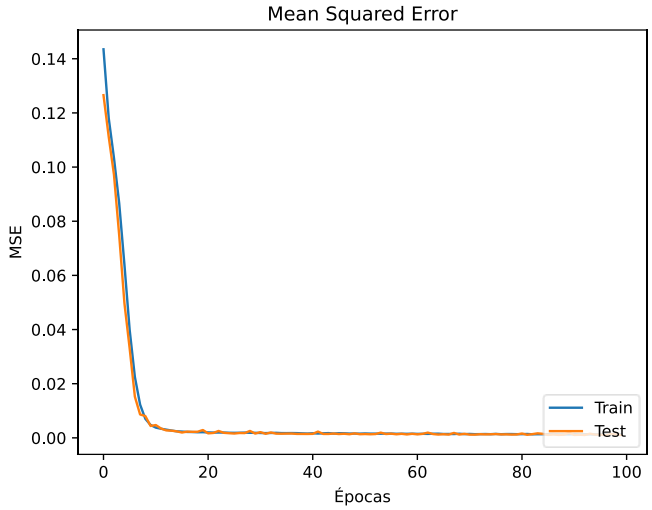


Figura 16: *mean squared error para ejercicio 5.*

Es decir, que luego de poco menos de 10 épocas, se llega a no tener error tanto en los datos de *test* como los de *train* para el mapeo logístico.

Ejercicio 6

En este ejercicio, se desea diseñar una red neuronal que permita predecir si los pacientes tienen posibilidad de tener diabetes o no a partir del dataset *"pima-indians-diabetes-csv".*

Para tal finalidad, se utiliza la técnica de validación cruzada *"K folds"*, 5 "pliegues", para evaluar 5 veces el modelo cambiando los datos de *test* y *train*. Para tal finalidad, se utiliza la librería *scikit-learn*, y se avalúa el modelo en sus 5 pliegues, obteniendo sus *accuracy* y *loss*. La arquitectura implementada fue la siguiente:

Tipo de capa	Propiedades de la capa
Dense	• units:50 • activation: ReLU
Dense	• unit:50 • activation: ReLU
Dense	• units:1 • activation: sigmoide

El optimizador utilizado fue *Adam* con un *learning rate* de 0.0001, la función de costo utilizada fue *binary cross entropy*, y el *batch size* de 128.

Finalmente, se evalúa cuales presentan las mayores y las menores *loss* y *accuracy,* y se obtiene el promedio de las mismas. Los resultados se pueden observar a continuación:

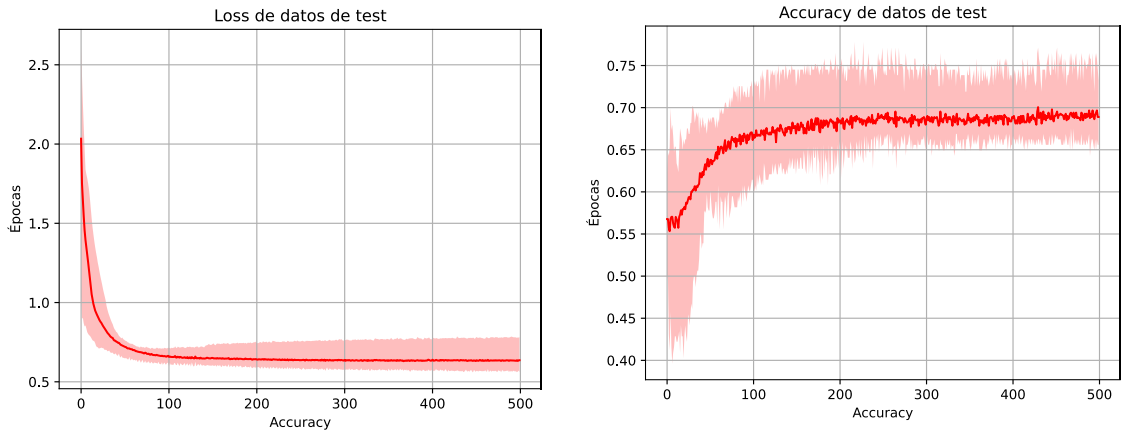


Figura 17: *loss y accuracy sin corrección.*

Luego, se analizaron los datos del dataset, observando que muchos de ellos eran 0, es decir que en ese paciente no se había medido ese parámetro. Por lo tanto, aquellos parámetros (excepto cantidad de embarazos, que puede ser que hayan sido 0), que eran 0, se modificaron por la media de su grupo, y se aplicó la siguiente arquitectura con más capas y más neuronas:

Tipo de capa	Propiedades de la capa
Dense	• units: 100 • activation: ReLU
Dense	• units: 100 • activation: ReLU
Dense	• units:50 • activation: ReLU
Dense	• units:50 • activation: ReLU
Dense	• units: 25 • activation: ReLU
Dense	• units: 25 • activation: ReLU
Dense	• units: 1 • activation: sigmoide

Los resultados fueron:

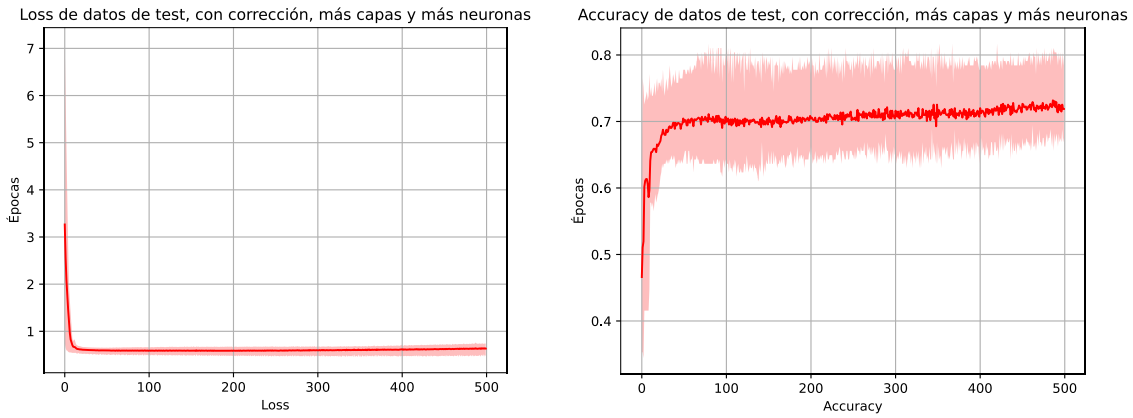


Figura 18: *loss y accuracy con corección, más capas y más neuronas.*

Como se observa, la *loss* de los 5 folds es prácticamente la misma, y la *accuracy* aumenta levemente llegando al 70%.

Ejercicio 7

El objetivo de este ejercicio es implementar un *autoencoder* para que haga de filtro de imágenes y les quite el ruido. Para tal fin, se utiliza el dataset MNIST de Keras, y lo primero que se hace es agregarle ruido blanco gaussiano con media en 0 y desviación estándar en 0.5, con un *noise factor* de 0.5. Luego, se implementa la siguiente arquitectura:

Tipo de capa	Propiedades de la capa
Dense - encoded	• units: 1000 • activation: ReLU
Dense – encoded	• units:500 • activation: ReLU
Dense - encoded	• units: 250 • activation: ReLU
Dense – decoded	• units: 500 • activation: ReLU
Dense – decoded	• units: 1000 • activation: ReLU

Se entrenó y evaluó con lo datos de las imágenes (x) con ruido y sin ruido, con optimizador *Adagrad, learning rate* de 1, función de activación *binary cross entropy* y *batch size* de 128. A continuación, se muestran los resultados:

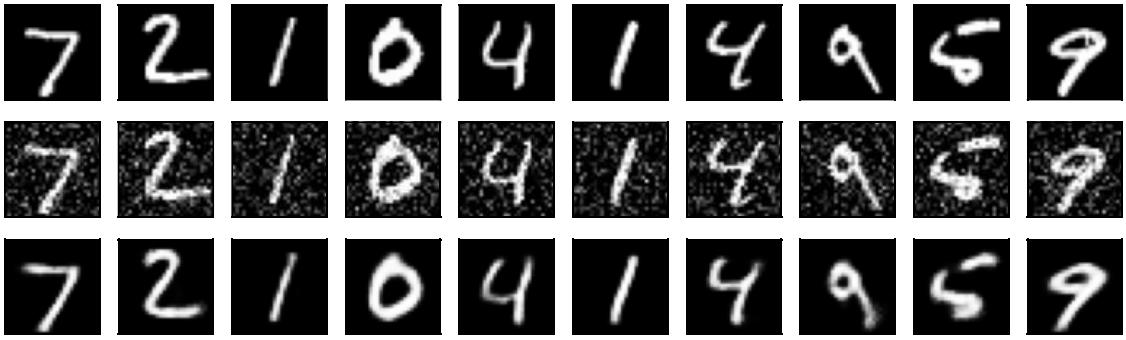


Figura 20: *primera fila, imágenes sin ruido. Segunda fila, imágenes con ruido. Tercera fila, imágenes luego del autencoder.*

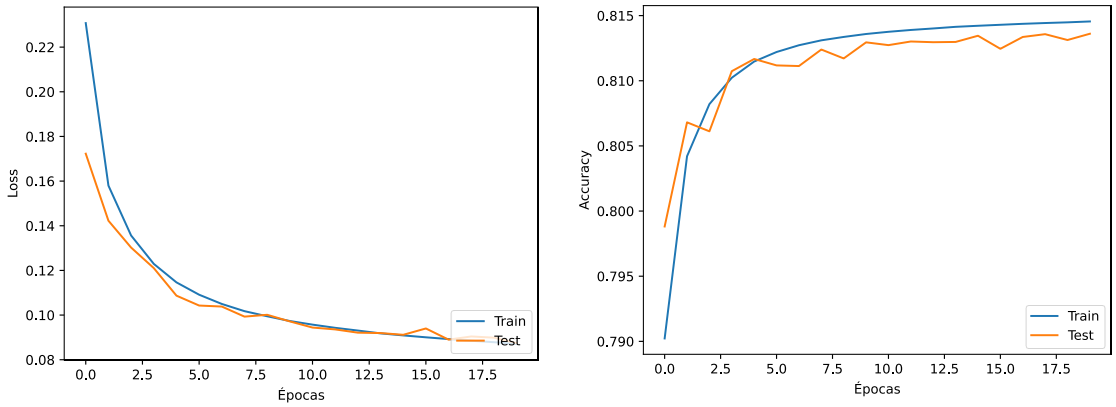


Figura 19: *loss y accuracy del ejercicio 7.*

Se concluye, que los resultados obtenidos son adecuados y que al haber poco *overfitting*, este *autoenconder* se podría utilizar en otras imágenes con ruido sin problemas.

Ejercicio 8

En este punto, se proponen dos arquitecturas para resolver la clasificación de datos del dataset de MNIST. La primera arquitectura, basada en capas densas, es la siguiente:

Tipo de capa	Propiedades de la capa
Dense	• units: 100 • activation: ReLU
Dropout	• rate: 0.15
Dense	• units: 50 • activation: ReLU
Dropout	• rate: 0.15
Dense	• units: 10 • activation: sigmoide

Con el optimizador *Adagrad* con un *learning rate* de 1e-2, función de costo *categorical cross entropy*, y *batch size* de 64, se obtuvieron los siguientes resultados:

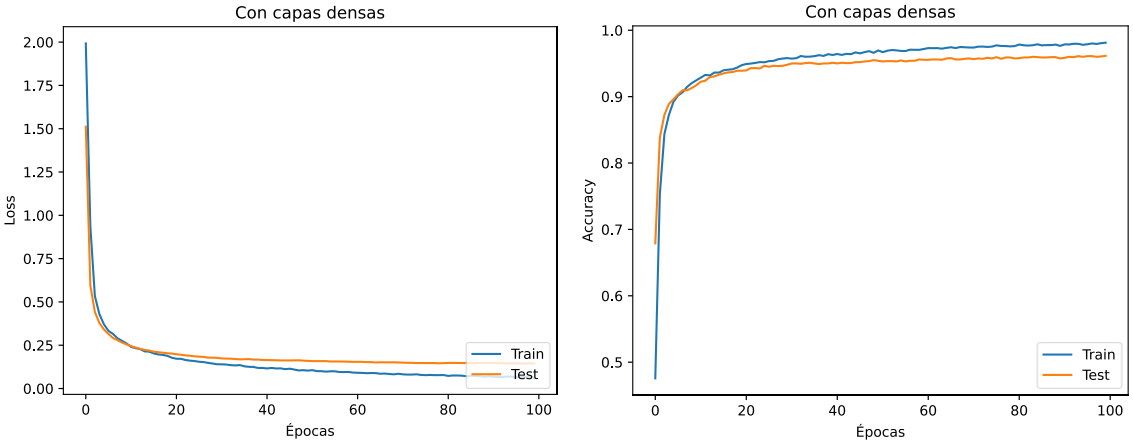


Figura 21: *loss y accuracy con arquitectura basada en capas densas.*

Luego, se propuso la siguiente arquitectura basada en capas convolucionales:

Tipo de capa	Propiedades de la capa
Conv2D	• filters: 4 • kernel_size: (3,3) • activation: ReLU
Dropout	• rate: 0.15
Conv2D	• filters: 4 • kernel_size: (3,3) • activation: ReLU
Dropout	• rate: 0.15
Flatten	
Dense	• units: 10 • activation: sigmoide

Los resultados fueron:

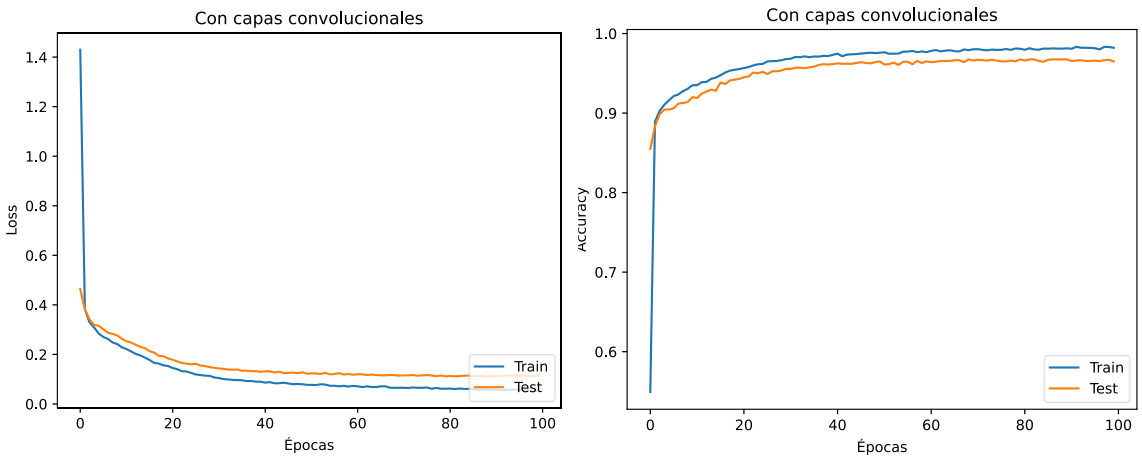


Figura 22: *loss y accuracy con capas convolucionales.*

Si se comparan los datos de *test* para ambas arquitecturas, se obtienen los siguientes gráficos:

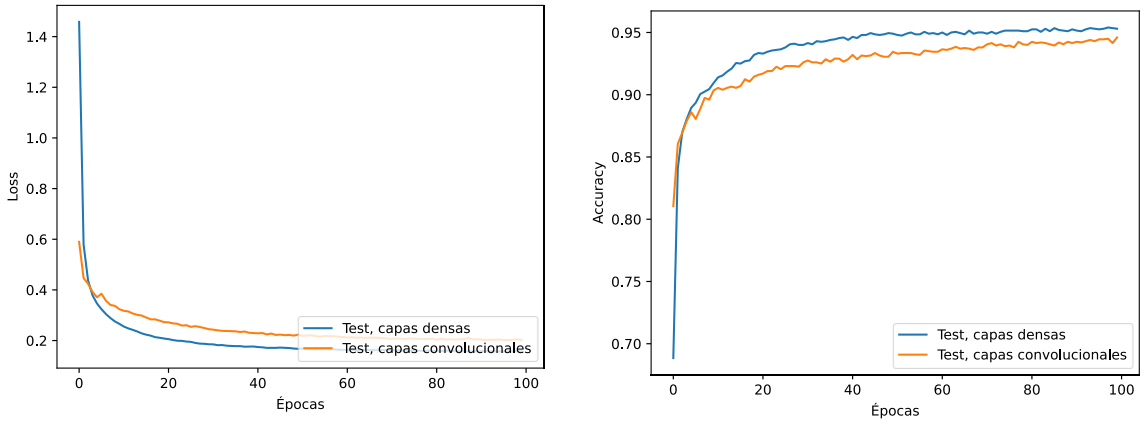


Figura 23: *comparación para datos de test de ambas arquitecturas.*

Como se observa, se obtuvieron mejores resultados aplicando capas densas. En una primera aproximación, se podría concluir que, aplicando mayor cantidad de filtros, quizá se podría haber llegado a mejores resultados con las capas convolucionales, las cuales aprenden características específicas de las imágenes (en este caso que son números, se esperaría que aprendan, por ejemplo, líneas horizontales, verticales, etcétera).

En cuanto a la aplicación de las capas de *Dropout*, fueron de utilidad para disminuir el *overfitting*, viendo que cumplen con su finalidad.

Ejercicio 9

En este ejercicio, se toman los códigos del ejercicio anterior, y se aplican permutaciones espaciales de los pixeles para los datos de *train* y de *test*. Los resultados obtenidos fueron:

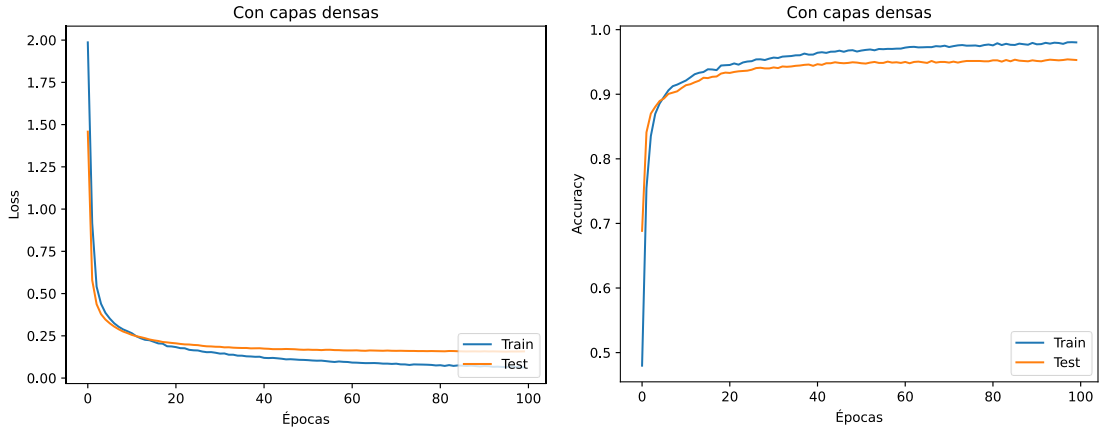


Figura 25: *loss y accuracy con capas densas.*

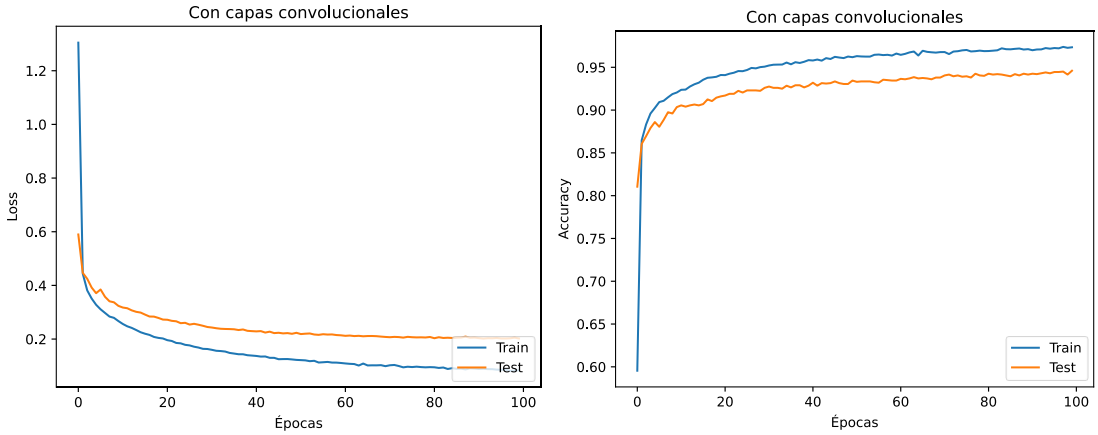


Figura 24: *loss y accuracy con capas convolucionales.*

A primera vista, se puede observar que la arquitectura con capas convolucionales presenta mayor *overfitting*. Se comparan ambos resultados para los datos de *test* y los resultados son:

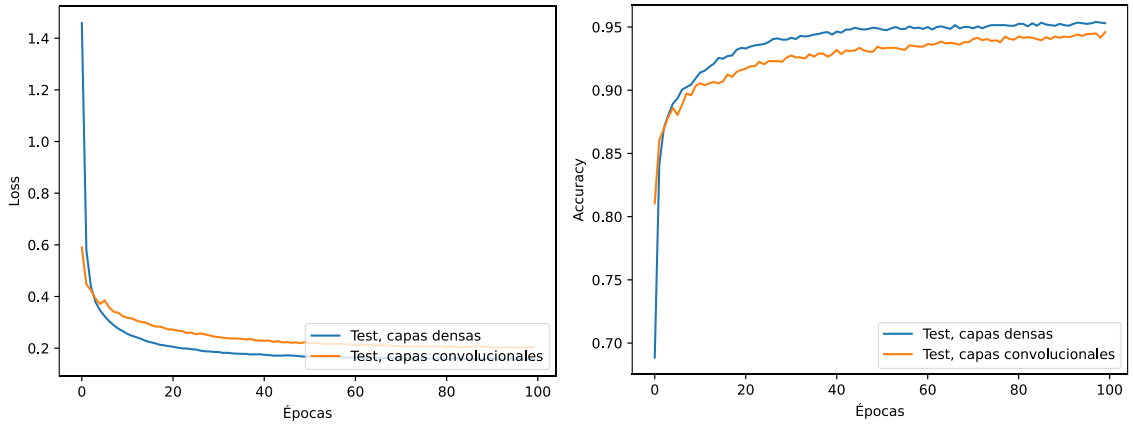


Figura 26: *comparación de resultados para capas densas y convolucionales.*

Al igual que en el punto anterior, se observa que las capas densas llegaron a mejores resultados a pesar de la permutación espacial.

Ejercicio 10

En este ejercicio, se pretende resolver la clasificación de imágenes de los datasets CIFAR-10 y CIFAR-100, utilizando las arquitecturas AlexNet y VGG16. En todos los casos, se utilizó aumentación de datos para mejorar la generalización con las siguientes características:

- rotation_range: 90
- horizontal_flip: True
- width_shift_range: 0.1
- height_shift_range: 0.1

También, en todas las arquitecturas se implementó *Batch Normalization, Dropout* y *L2* para mejorar la generalización.

La primera arquitectura implementada fue AlexNet para CIFAR-10, con *Adagrad* como optimizador, *learning rate* de 1e-2, función de costo *categorical cross entropy* y *batch size* de 64, y se obtuvieron los siguientes resultados:

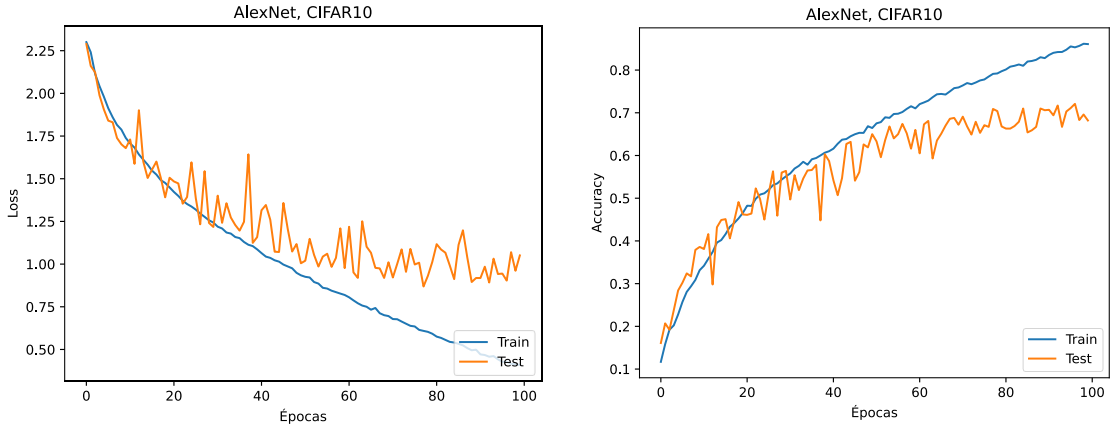


Figura 27: *AlexNet para CIFAR-10.*

Luego, se implementó VGG-16 para CIFAR-10, con optimizador *Adadelta*, *learning rate* de 1, función de costo *categorical cross entropy* y *batch size* de 64. Cabe aclarar que tanto para CIFAR-10 como CIFAR-100, con la arquitectura VGG-16, se corrieron sólo 20 épocas debido a cuestiones de tiempo (tardaba aproximadamente 98 segundos por época). De igual manera, se pudieron obtener resultados suficientes para comparar las arquitecturas. Los resultados para CIFAR-10 fueron:

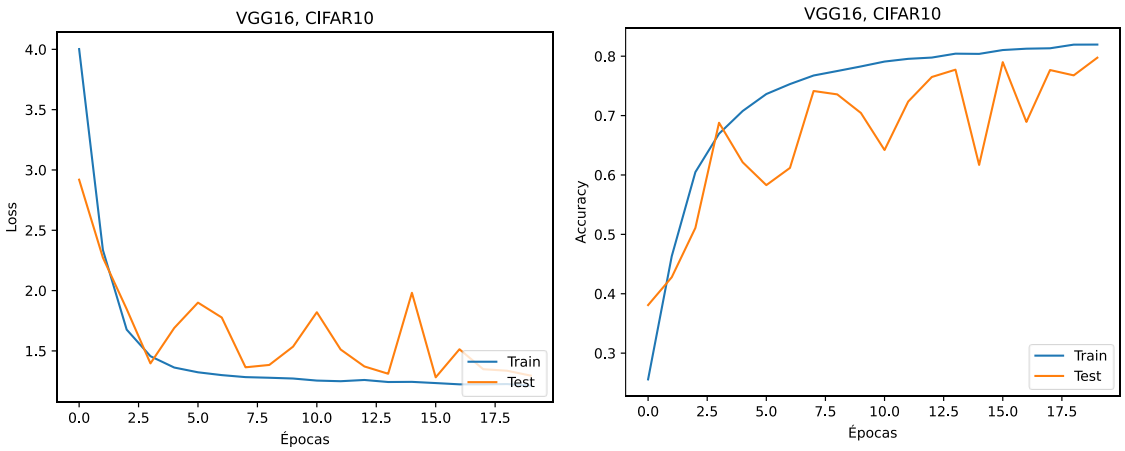


Figura 28: *VGG16 para CIFAR-10.*

Si se comparan los resultados para 20 épocas, VGG16 llega a una *accuracy* de 80%, mientras que AlexNet llega a 50%, llegando como máximo a 70% para 100 épocas.

Luego, se procede a implementar AlexNet para CIFAR-100 con los mismo hiperparámetros de AlexNet para CIFAR-10 y se obtiene:

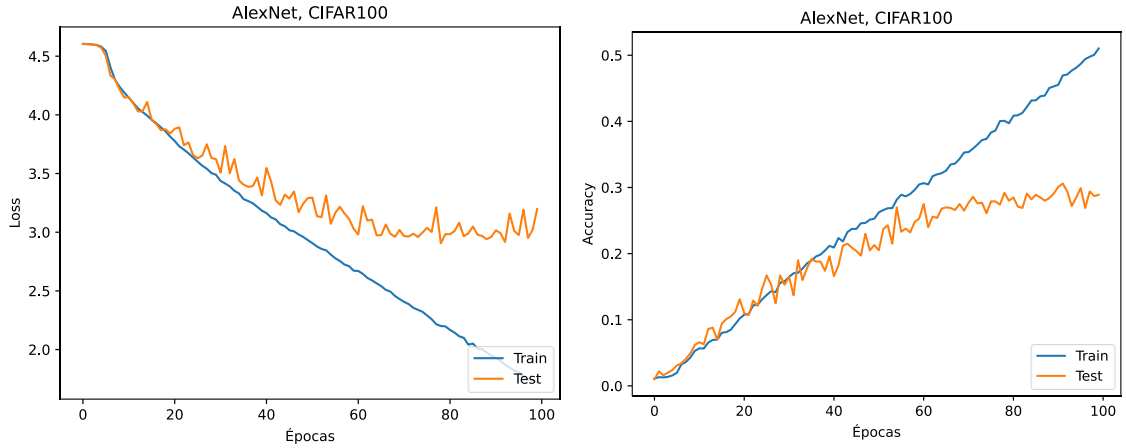


Figura 29: *AlexNet para CIFAR-100.*

Como se observa, se tiene mayor *overfitting* y menor *accuracy*.

Por último, se implementa VGG16 para CIFAR-100 con los mismos hiperparámetros que para CIFAR-10 y se obtiene:

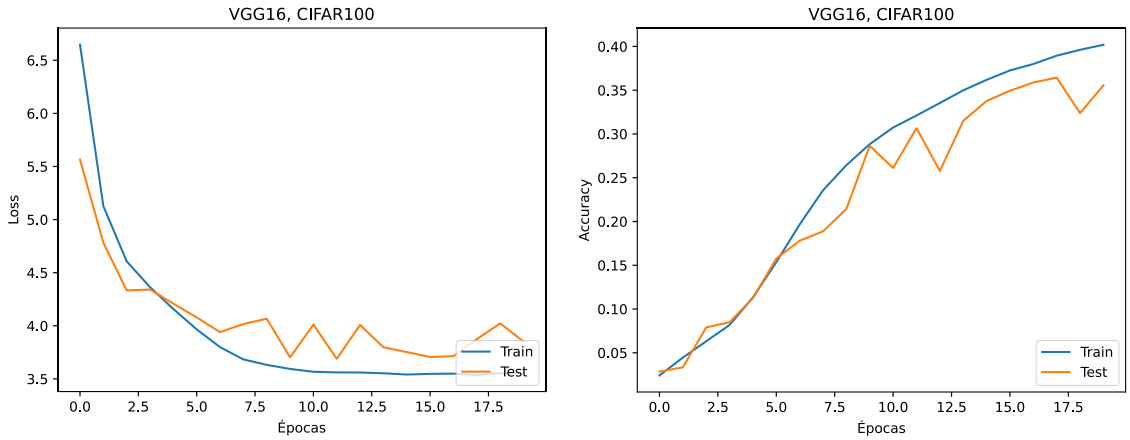


Figura 30: *VGG16 para CIFAR-100.*

Se concluye que VGG16 con tan sólo 20 épocas, llega a un *accuracy* de 40%, mientras que AlexNet para 20 épocas llega a 15%, llegando como máximo a 30% a partir de las 60 épocas.