Trabajo Práctico 4

Camussoni Francesco[1,*]

[1]Instituto Balseiro, Universidad Nacional de cuyo
[*]camussonif@gmail.com

ABSTRACT

Se hizo uso de la librería Keras para resolver problemas dados como clasificación del set de imágenes CIFAR10 y de reseñas de IMDB, entre otros, a través de redes neuronales de distintas arquitecturas. En la mayoría de los ejercicios se utilizó el optimizador Adam debido a su velocidad y eficiencia de aprendizaje y la función de activación relu por su bajo costo computacional.

Ejercicio 1

Se implementó una red neuronal simple, de 1 sola neurona con función de activación lineal, para predecir el precio de una casa en la ciudad de Boston según ciertos atributos. Se entrenó durante 200 épocas con Adam como regularizador y MSE como función costo. Un resumen de dicha red se muestra a continuación:

```
Model: "Ejercicio 1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 14

=================================================================
Total params: 14
Trainable params: 14
Non-trainable params: 0
_________________________________________________________________
```

Así se obtuvieron los resultados que se muestran en las **Figuras 1** y **2**.

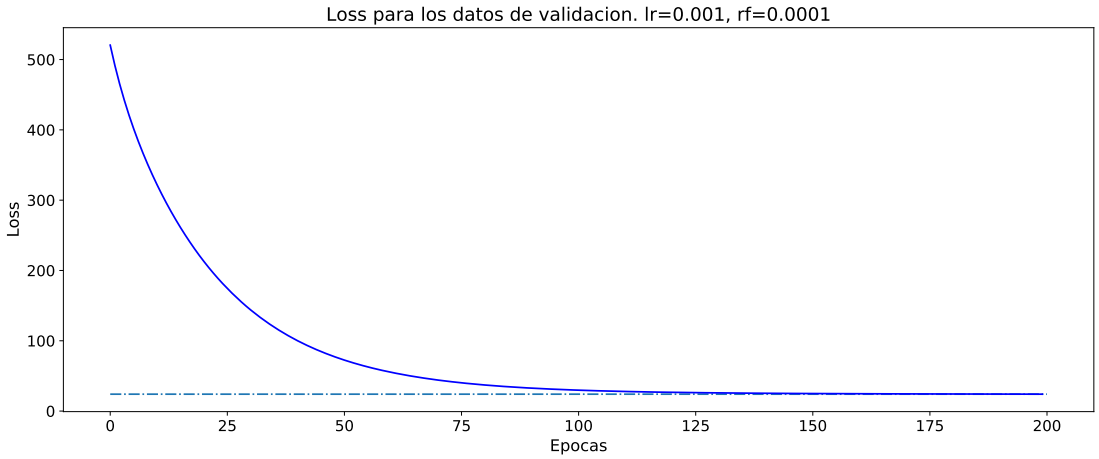

Figure 1. Función costo MSE en función de las epocas para los datos de test. Se alcanzo un costo mínimo de 24.04.

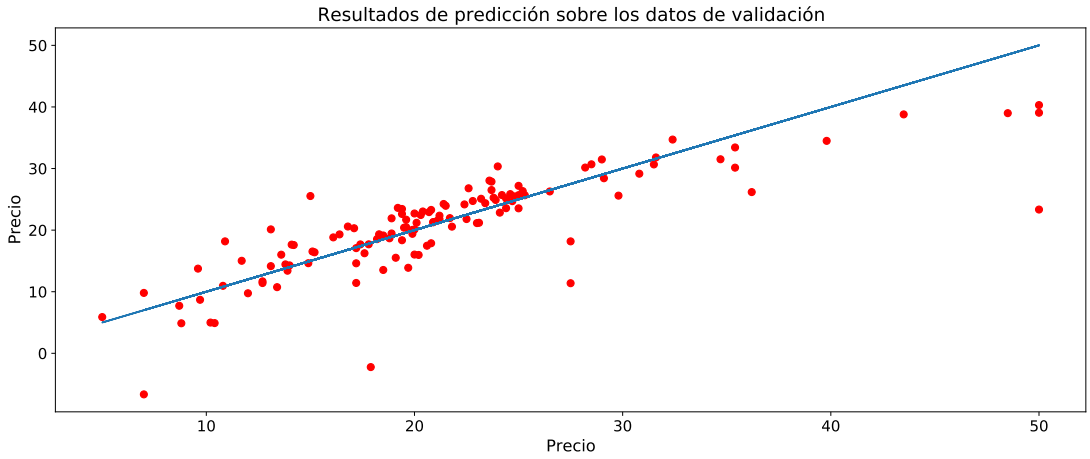

Figure 2. Costo de las casas en función del costo de las mismas y de los valores estimados para los datos de test.

Ejercicio 2

Se implementaron los problemas 3, 4 y 6 de la práctica 2 utilizando la librería de Keras.

- Para el problema 3 de la práctica 2 se utilizó una red neuronal de 2 capas densas de neuronas para implementar un clasificador de imagenes de CIFAR 10. La primera fue de 100 neuronas con una regularización L2 y una función de activación sigmoidal. Para la segunda se utilizaron 10 neuronas, con la misma regularización y con salida lineal. Finalmente, la función de costo fue MSE y el optimizador fue SGD para ser consistente con el práctico anterior. Un esquema de la misma se muestra a continuación:

```
Model: "Ejercicio 2 a y b"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               307300

_________________________________________________________________
dense (Dense)                (None, 10)                1010

=================================================================
Total params: 308,310
Trainable params: 308,310
Non-trainable params: 0
_________________________________________________________________
```

Así se obtuvieron los resultados que se muestran en las **Figuras** 3 y 4.

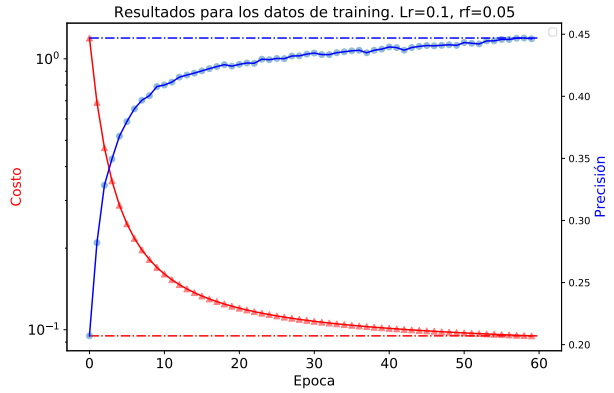

Figure 3. Costo y precisión de los datos de training para la red neuronal descripta.

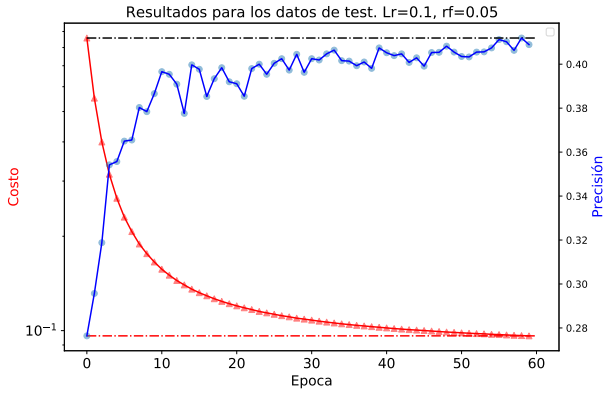

Figure 4. Costo y precisión de los datos de test para la red neuronal descripta.

- Para el problema 4 de la práctica 2 se utilizó la misma arquitectura pero se utilizó CCE como función de costo. Así se obtuvieron los resultados que se muestran en las **Figuras 7** y **8**.

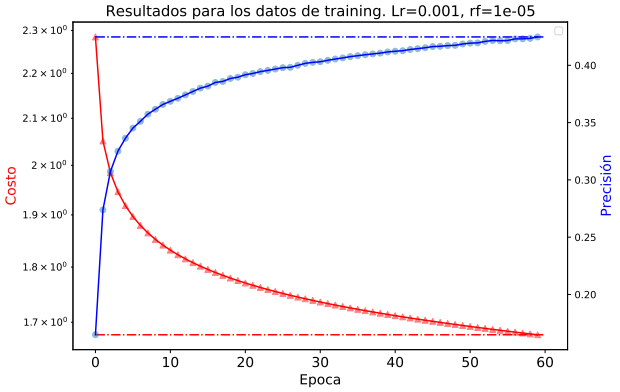

Figure 5. Costo y precisión de los datos de training para la red neuronal descripta.

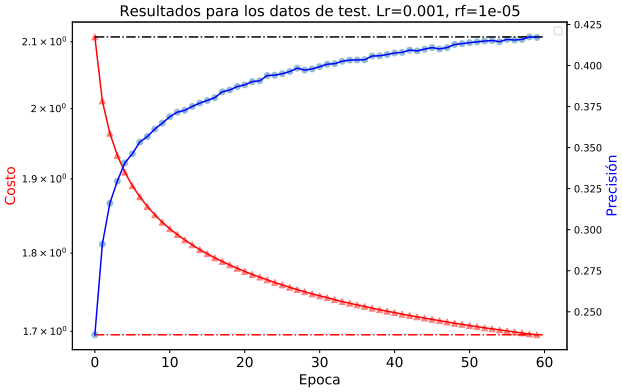

Figure 6. Costo y precisión de los datos de test para la red neuronal descripta.

- Para el problema 6 de la práctica 2 se implementó un clasificador binario para resolver el problema XOR bidimensional. Para esto se hizo uso de dos redes neuronales con funciones de activación tanh, función de costo MSE y el optimizador utilizado fue SGD. La primer red neuronal constó de dos capas de neuronas de 2 y 1 respectivamente mientras que la segunda red neuronal constó de 2 capas de 1 neurona con la ultima concatenada a los datos de entradas. Un esquema de la primer arquitectura se muestra a continuación:

```
Model: "Ejercicio 2 c"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(None, 2)]               0

_________________________________________________________________
dense (Dense)                (None, 2)                 6

_________________________________________________________________
dense (Dense)                (None, 1)                 3
```

```
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

Así se obtuvieron los resultados que se muestran en las **Figuras 7** y **8**.

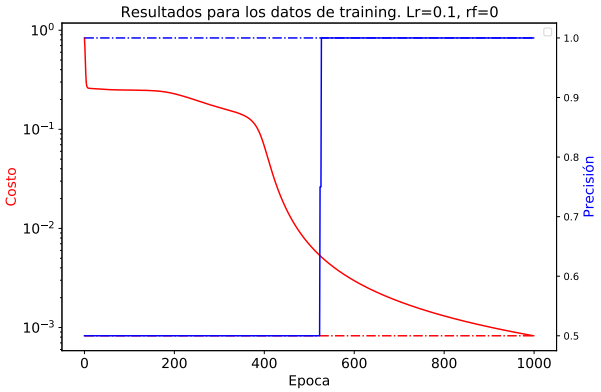

Figure 7. Costo y precisión de los datos de training para la primer red neuronal descripta, sin concatenación.

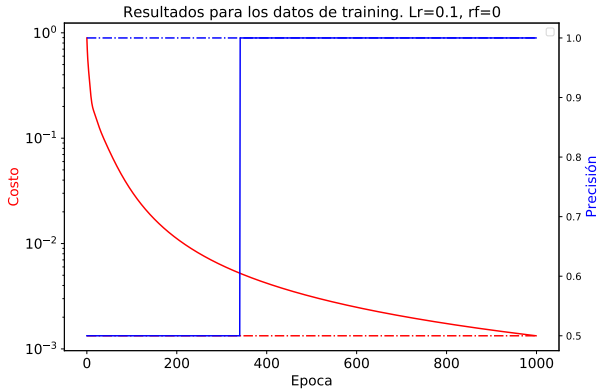

Figure 8. Costo y precisión de los datos de training para la segunda red neuronal descripta, con concatenación.

En ningún caso se uso regularización por no haber datos de test.

Ejercicio 3

Se implementó un clasificador de reseñas realizadas en IMDB como positivas o negativas. Para esto se utilizó la siguiente arquitectura: tres capas densas de 100, 10, 1 neuronas respectivamente. Se concatenó la entrada con la última capa oculta. Se utilizó la función de activación relu para todas las salidas, optimizador Adam y la función costo BinaryCrossentropy. Luego el accuracy se midió a través de BinaryClassifier. Se utilizó batch normalization, dropout y regularización como estrategias de regularización

Cada reseña contenía un numero distinto de palabras por lo que cada entrada era de longitud variable. Para solucionar esto se realizó un preprocesado mediante una variación de one hot encoder en donde en lugar de ubicar un 1 en la posición de la palabra presente correspondiente, se asignó la cantidad de veces que se repite en la oración dicha palabra

- Para la regularización simple se obtuvieron los resultados que se observan en la **Figura 9**, en donde se obtuvo una precisión para los datos de test de 0.88.

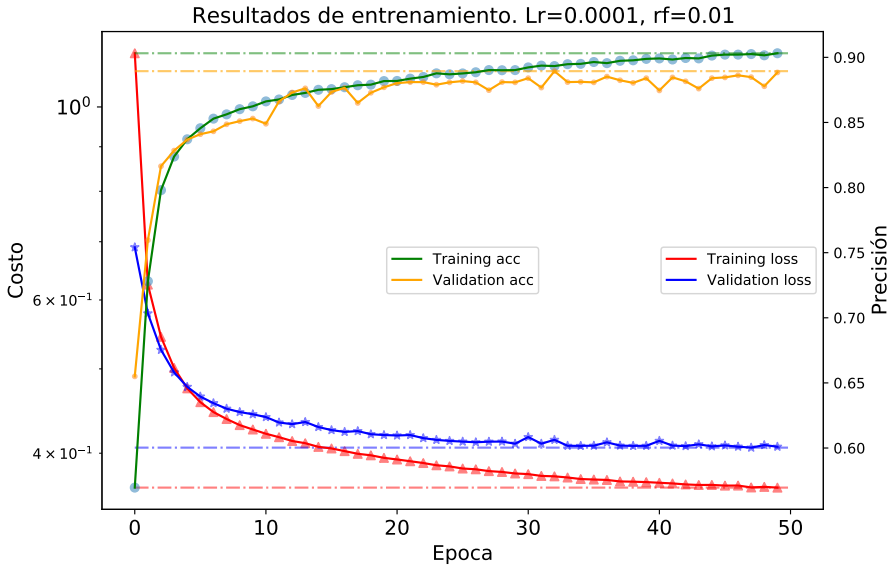

Figure 9. Resultados para el entrenamiento utilizando solo regularización L2, se muestran los datos de validacion y de test.

En donde la arquitectura fue la siguiente:

```
Model: "Ejercicio 3 solo L2"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #
   Connected to
==================================================================================================
input (InputLayer)           [(None, 10000)]      0


_________________________________________________________________________________________________
dense (Dense)                (None, 100)          1000100
   input[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)                (None, 10)            1010
   dense[0][0]
_________________________________________________________________________________________________
concatenate (Concatenate)        (None, 10010)         0
   input[0][0]

   dense_1[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)                (None, 1)             10011
   concatenate[0][0]
==================================================================================================
Total params: 1,011,121
Trainable params: 1,011,121
Non-trainable params: 0
_________________________________________________________________________________________________
```

- Ademas de la regularización se utilizo batch normalization. Por empezar, se mantuvieron los hiperparámetros anteriores para observar el comportamiento de la red. Luego se hizo una búsqueda de hiperparámetros adecuados. Así, se obtuvieron los resultados que se observan en las **Figuras** 11 y 10. Finalmente, se obtuvo una precisión para los datos de test también de 0.87.

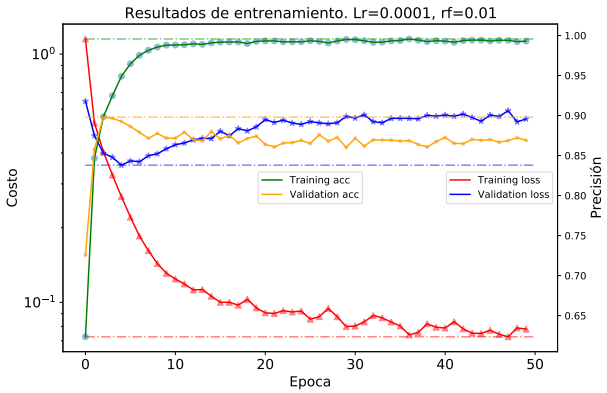

Figure 10. Resultados para los hiperparámetros anteriores

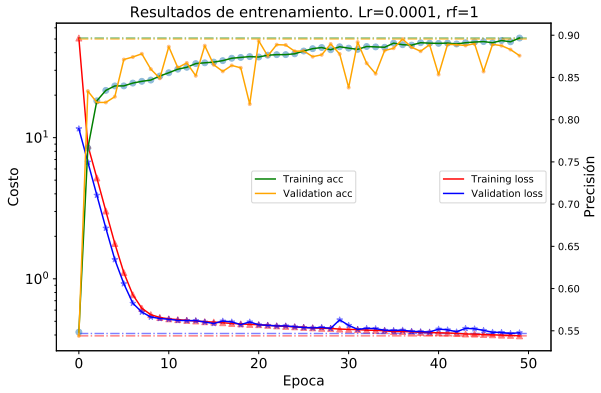

Figure 11. Resultados de entrenamiento utilizando regularizacion L2+batch normalization.

Como se observa, el batch normalization apresura el aprendizaje, sin embargo, al utilizar los hiperparámetros anteriores, se produce un rápido overfitting, como se observa en la primer figura. Esto puede deberse a que al levantar las activaciones de salida de alguna etapa los parametros de la red son excesivamente altos, por lo que es necesario aumentar el factor de regularización como se observa en la segunda figura. Luego, al encontrar los hiperparámetros adecuados, el gap entre el loss disminuyó sustancialmente, aunque los resultados en el accuracy resultaron mas ruidosos.

A continuación se presenta un resumen de la arquitectura utilizada:

```
Model: "Ejercicio 3 batch normalization"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #
   Connected to
==================================================================================================
input (InputLayer)           [(None, 10000)]           0


_________________________________________________________________________________________________
dense (Dense)                (None, 100)               1000100
   input[0][0]
_________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 100)            400
   dense[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 10)                1010
   batch_normalization[0][0]
_________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 10)             40
   dense_1[0][0]
_________________________________________________________________________________________________
concatenate (Concatenate)    (None, 10010)             0
   input[0][0]

   batch_normalization_1[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)              (None, 1)                 10011
   concatenate[0][0]
==================================================================================================
Total params: 1,011,561
Trainable params: 1,011,341
Non-trainable params: 220
_________________________________________________________________________________________________
```

- Para dropout se obtuvieron los resultados que se observan en la **Figura 12**, en donde se obtuvo una precisión para los datos de test de 0.88. Para este caso los hiperparámetros originales resultaron adecuados.

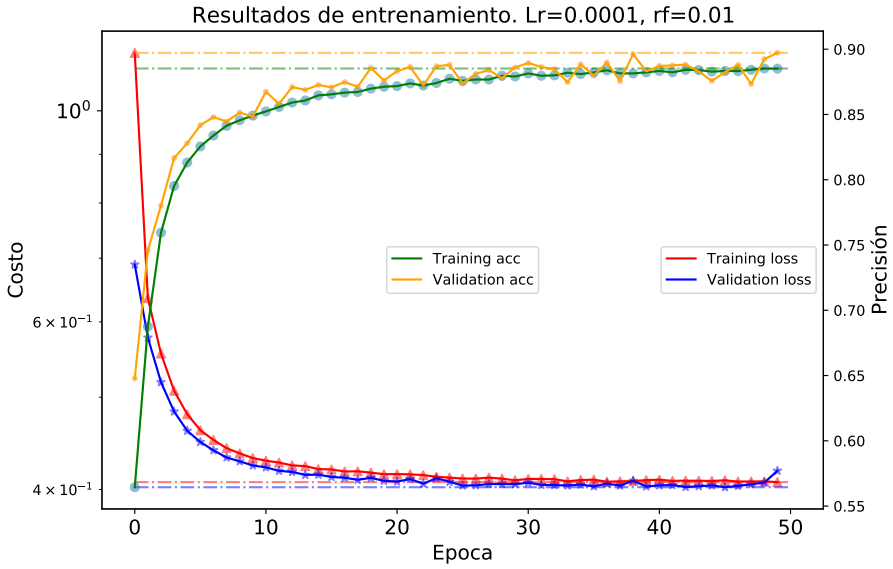

Figure 12. Resultados para el entrenamiento utilizando regularización L2+dropout, se muestran los datos de validacion y de test.

Se observa una reducción del ruido del accuracy con respecto al batch normalization y que además el gap entre el loss de validación y de training disminuyó en gran medida.

La arquitectura utilizada fue:

```
Model: "Ejercicio 3 dropout"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #
   Connected to
==================================================================================================
input (InputLayer)           [(None, 10000)]      0


_________________________________________________________________
dense (Dense)                (None, 100)          1000100
   input[0][0]
_________________________________________________________________
dropout (Dropout)            (None, 100)          0
   dense[0][0]
_________________________________________________________________
dense_1 (Dense)              (None, 10)           1010
   dropout[0][0]
_________________________________________________________________
dropout_1 (Dropout)          (None, 10)           0
   dense_1[0][0]
_________________________________________________________________
concatenate (Concatenate)    (None, 10010)        0
   input[0][0]

   dropout_1[0][0]
_________________________________________________________________
dense_2 (Dense)              (None, 1)            10011
   concatenate[0][0]
```

```
=================================================================================================
Total params: 1,011,121
Trainable params: 1,011,121
Non-trainable params: 0
_________________________________________________________________
```

Sobre el costo computacional, la regularización más veloz resultó la L2 sola, demorando aproximadamente 3 segundos por época. L2+batch normalization demoró 4 segundos por época y L2+dropout demoró entre 3 y 4 segundos dependiendo la época.

Ejercicio 4

Se implemento una red neuronal para clasificar reseñas de IMDB como positivas o negativas. En esta ocasión se realizó un padding a cada reseña para que cada una de ellas tenga 500 palabras, truncando o rellenando con ceros. Luego, se utilizó la layer Embedding de Keras, luego de la de Input, para transformar cada palabra a un espacio de vectores de 32 dimensiones. A la salida del Embedding se realizó un Flatten para ajustar la dimensión de entrada de la capa densa siguiente. El resto de la arquitectura se mantuvo igual a la del ejercicio anterior en donde se concatenó la salida de la capa Flatten con la de la última capa oculta. Se realizó regularización L2 en todas las capas y dropout a la salida de ambas capas ocultas. La arquitectura de dicha red se la resume a continuación:

```
    Model: "Ejercicio 4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
   Connected to
=================================================================================================
input (InputLayer)           [(None, 500)]             0


_________________________________________________________________
embedding (Embedding)         (None, 500, 32)           320000
   input[0][0]
_________________________________________________________________
flatten (Flatten)             (None, 16000)             0
   embedding[0][0]
_________________________________________________________________
dense (Dense)                (None, 100)               1600100
   flatten[0][0]
_________________________________________________________________
dropout (Dropout)             (None, 100)               0
   dense[0][0]
_________________________________________________________________
dense_1 (Dense)               (None, 10)                1010
   dropout[0][0]
_________________________________________________________________
dropout_1(Dropout)            (None, 10)                0
   dense_1[0][0]
_________________________________________________________________
concatenate (Concatenate)     (None, 16010)             0
   flatten[0][0]

   dropout_1[0][0]
_________________________________________________________________
dense_2 (Dense)               (None, 1)                 16011
   concatenate[0][0]
=================================================================================================
Total params: 1,937,121
Trainable params: 1,937,121
Non-trainable params: 0
_________________________________________________________________
```

Se utilizó Adam como regularizador, BinaryCrossEntropy como función costo y BinaryAccuracy como metrica de precisión. Así se obtuvieron los resultados que se muestran en la **Figura 13** en donde se obtuvo una precisión de 0.89 para los datos de test.

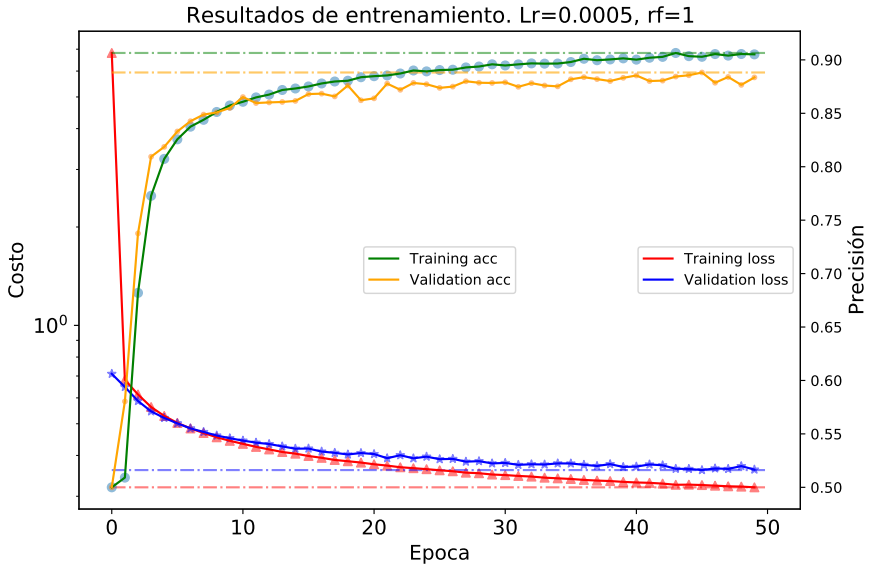

Figure 13. Resultados para el entrenamiento utilizando regularización L2+dropout y la utilización de Embeddings, se muestran los datos de validación y de test.

El Embeddings permite relacionar palabras como por ejemplo 'rico' y 'sabroso' al llevarlas al espacio de vectores con distancias cercanas, por lo que a priori el rendimiento podría ser mayor dado que de la forma anterior, con one hot encoding, estas palabras palabras podrían estar lejos sin poder hacer mucho con ello. Sin embargo, en este caso, se obtuvo un rendimiento apenas mayor sobre los datos de test (0.88 de precisión) en donde aumento el overfitting en general.

A continuación, se propuso una red convolucional que constó de la misma capa de Embedding al comienzo para luego utilizar dos capas convolucionales de 32 y 64 filtros con un kernel de tamaño 5 en ambas direcciones. Se realizó un batch normalization previo a cada capa convolucional y un batch normalization y dropout previo a la capa densa de salida. La arquitectura resulto la siguiente:

```
Model: "Ejercicio 4 convolucional"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(None, 500)]             0

_________________________________________________________________
embedding (Embedding)        (None, 500, 32)           320000

_________________________________________________________________
batch_normalization (Batch   (None, 500, 32)           128

_________________________________________________________________
conv1d (Conv1D)              (None, 500, 16)           2576

_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 250, 16)           0

_________________________________________________________________
batch_normalization_1 (Batch (None, 250, 16)           64
```

```
conv1d_1 (Conv1D)              (None, 250, 32)           2592

max_pooling1d_1 (MaxPooling1 (None, 125, 32)           0

batch_normalization_2 (Batch (None, 125, 32)           128

dropout (Dropout)            (None, 125, 32)           0

flatten (Flatten)            (None, 4000)              0

dense (Dense)                (None, 1)                 4001

=================================================================
Total params: 329,489
Trainable params: 329,329
Non-trainable params: 160
```

Así se obtuvieron los resultados que se muestran en la **Figura 14**

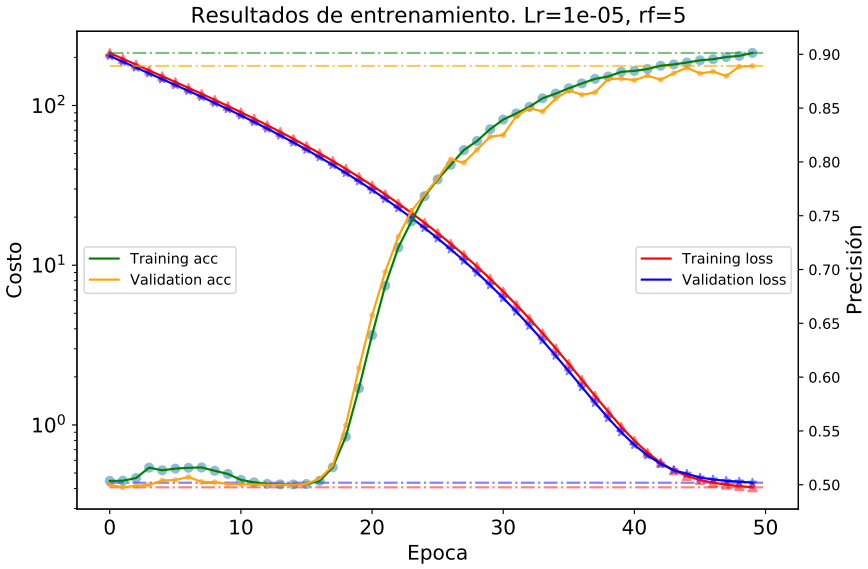

Figure 14. Resultados para el entrenamiento utilizando regularización L2+dropout y la utilización de Embeddings, se muestran los datos de validación y de test para la red convolucional.

En donde los parámetros a entrenar fueron menores que en el caso anterior y se obtuvo un rendimiento similar.

Ejercicio 5

Se utilizó una red neuronal para resolver la evolución de la ecuación caótica y=4x(1-x) en donde la arquitectura implementada se muestra a continuación:

```
Model: "Ejercicio 5"
```

```
Layer (type)                    Output Shape         Param #
   Connected to
==================================================================================================
input (InputLayer)              [(None, 1)]          0

--------------------------------------------------------------------------------------------------
dense (Dense)                   (None, 5)            10
   input[0][0]
--------------------------------------------------------------------------------------------------
concatenate (Concatenate)       (None, 6)            0
   input[0][0]

   dense[0][0]
--------------------------------------------------------------------------------------------------
dense_1 (Dense)                 (None, 1)            7
   concatenate[0][0]
==================================================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
```

Se utilizó la funcion MeanAbsoluteError de Keras como función de costo y la función de activacion tanh para la capa densa dada por el ejercicio. El optimizador fue Adam y para la neurona de salida se utilizó una activación lineal. No se utilizó ninguna regularización dado que el vector y fue calculado análiticamente a partir de la ecuación planteada, por lo que los datos no presentan ruido y no hay riesgo de que la red neuronal aprenda el mismo. Así se obtuvieron los resultados que se muestran en las **Figuras 15** y **16**.

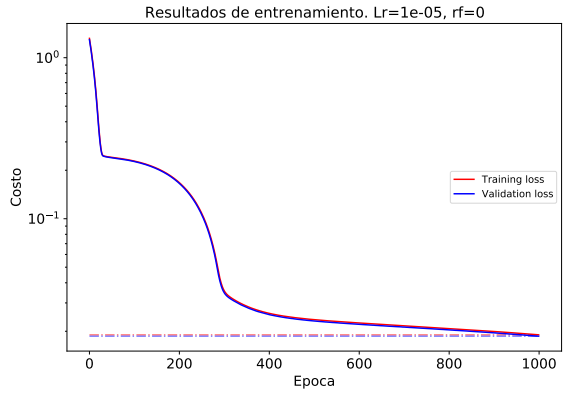

Figure 15. Resultados de la función costo para los datos de test y training

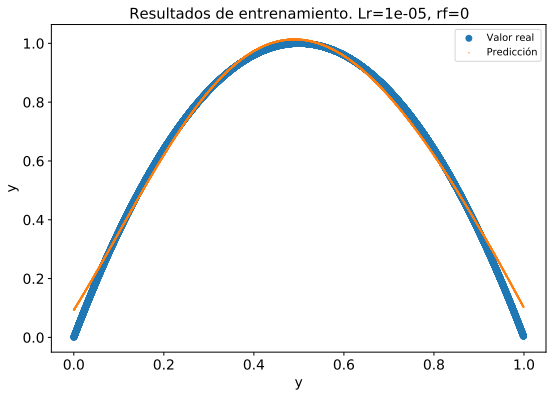

Figure 16. Valor real de salida de la ecuación y=4x(1-x) y valores de predicción de la red neuronal sobre los datos de test.

Como se observa, al no haber ruido, el costo de los datos de training y de test resultan identicos.

Ejercicio 6

Se utilizó el un esquema 5-folding para resolver el problema de clasificacion de 'pima-indians-diabetes.csv'. Para esto se utilizó el siguiente esquema de red neuronal:

```
Model: "Ejercicio 6"
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 8)]               0
```

```
dense (Dense)                (None, 24)                216
batch_normalization (BatchNo (None, 24)                96
dropout (Dropout)            (None, 24)                0
dense_1 (Dense)              (None, 12)                300
batch_normalization_1 (Batch (None, 12)                48
dropout_1 (Dropout)          (None, 12)                0
dense_2 (Dense)              (None, 1)                 13
=================================================================
Total params: 673
Trainable params: 601
Non-trainable params: 72
_________________________________________________________________
```

En donde se utilizó la función de activación relu para las dos capas densas y lineal para la capa de salida. La función de costo fue BinaryCrossEntropy y se utilizó BinaryAccuracy como métrica. El optimizador fue Adam y se utilizó regularizacion L2 en todas las capas. Se realizó un primer entrenamiento en donde solo se normalizaron los datos, obteniendo el resultado que se muestra en la **Figura 17**.

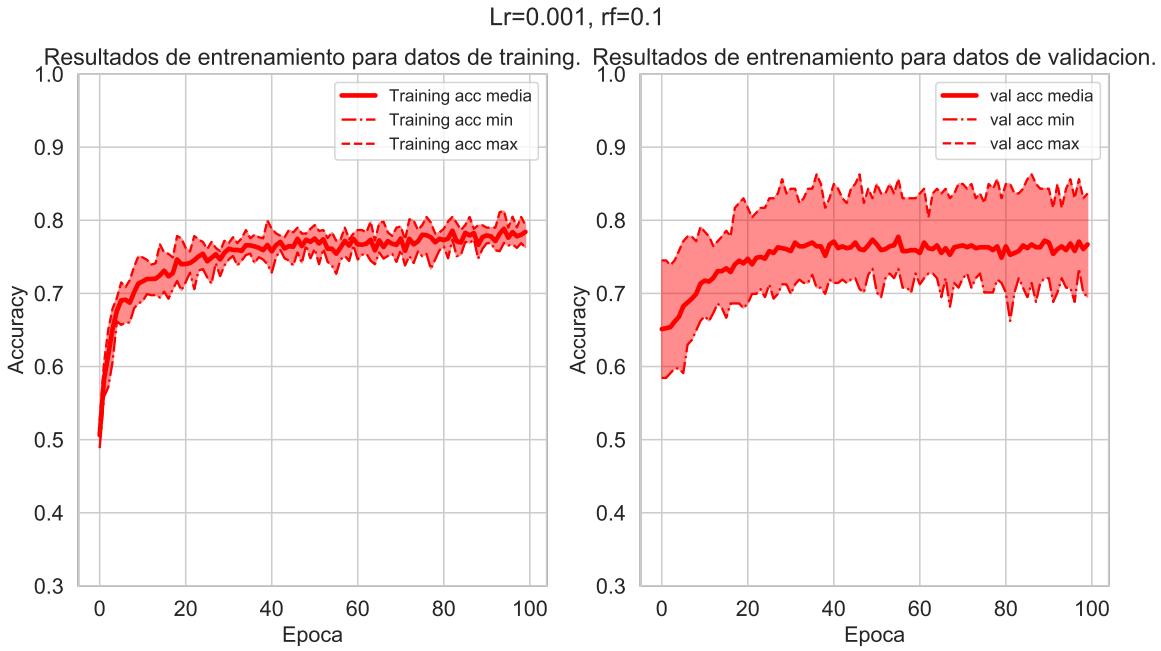

Figure 17. Resultados de entrenamiento para los datos de training y de validación con los datos solamente normalizados.

Luego se remplazó los valores faltantes en los distintos atributos por la media de los mismos salvo en los atributos 'embarazos' y 'edad' donde los ceros se dejaron como tales. Así se obtuvo un resultado como se muestra en la **Figura 18**.

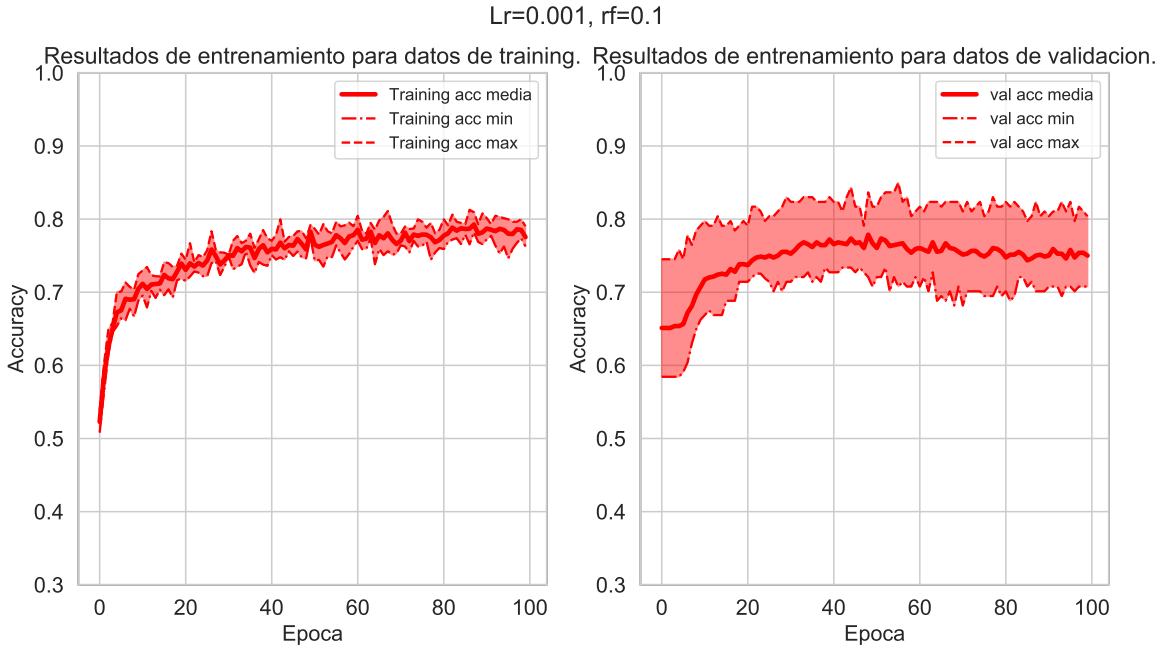

Figure 18. Resultados de entrenamiento para los datos de training y de validación con los datos corregidos.

Se observa un resultado similar con una disminución en el ruido de los resultados.

Ejercicio 7

Se implementó un autoencoder para eliminar ruido del dataset de MNIST. Para esto se normalizaron todas las entradas y se agrego un ruido con distribución normal centrado en cero y con std=0.5. Se utilizó la función clip de numpy para mantener cada pixel de la imagen entre 0 y 1. Así se obtuvieron números manuscritos con ruido como se muestra en la **Figura 19**.

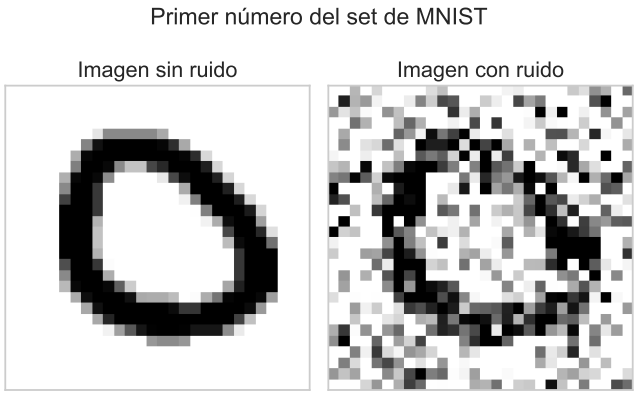

Figure 19. Número manuscrito original y con ruido agregado.

Luego, se implemento la siguiente red neuronal de autoencoder prupuesta por keras (https://blog.keras.io/building-autoencoders-in-keras.html) con las mismas funciones de activación, optimizador, funcion costo y métricas. Se realizó una pequeña variación en la capa de encoder para poder una mejor visualización:

```
Model: "Ejercicio 7"
_________________________________________________________________
```

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 16)	4624
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 16)	0
conv2d_2 (Conv2D)	(None, 7, 7, 1)	145
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 1)	0
conv2d_3 (Conv2D)	(None, 4, 4, 1)	10
up_sampling2d (UpSampling2D)	(None, 8, 8, 1)	0
conv2d_4 (Conv2D)	(None, 8, 8, 16)	160
up_sampling2d_1 (UpSampling2	(None, 16, 16, 16)	0
conv2d_5 (Conv2D)	(None, 14, 14, 32)	4640
up_sampling2d_2 (UpSampling2	(None, 28, 28, 32)	0
conv2d_6 (Conv2D)	(None, 28, 28, 1)	289

Total params: 10,188
Trainable params: 10,188
Non-trainable params: 0

En primer lugar se entrenó la red para solamente para realizar una codificacion y decodificiación de los datos originales sin ruido para observar como se veía la codificación. Así se obtuvieron los resultados que se muestran en las **Figuras** 20 y 21.

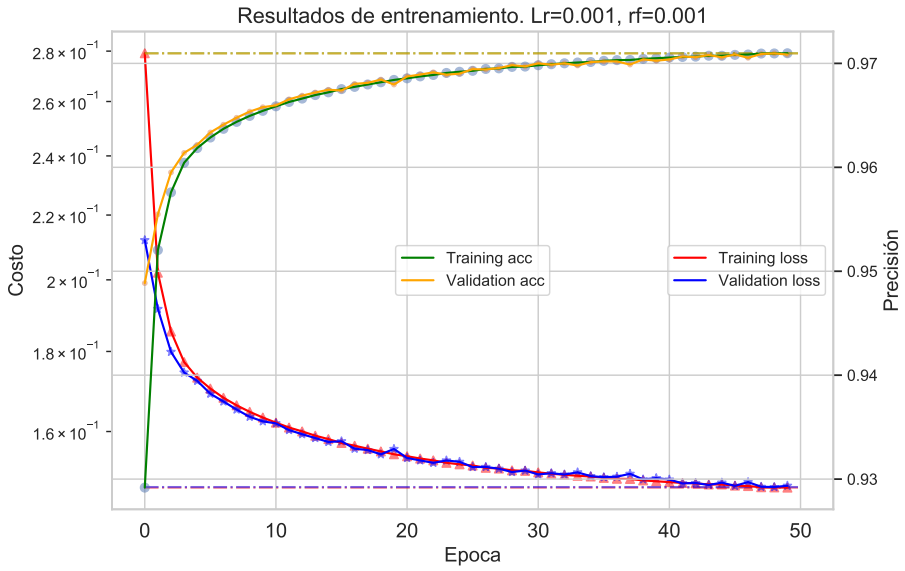

Figure 20. Resultados de entrenamiento para los datos originales.

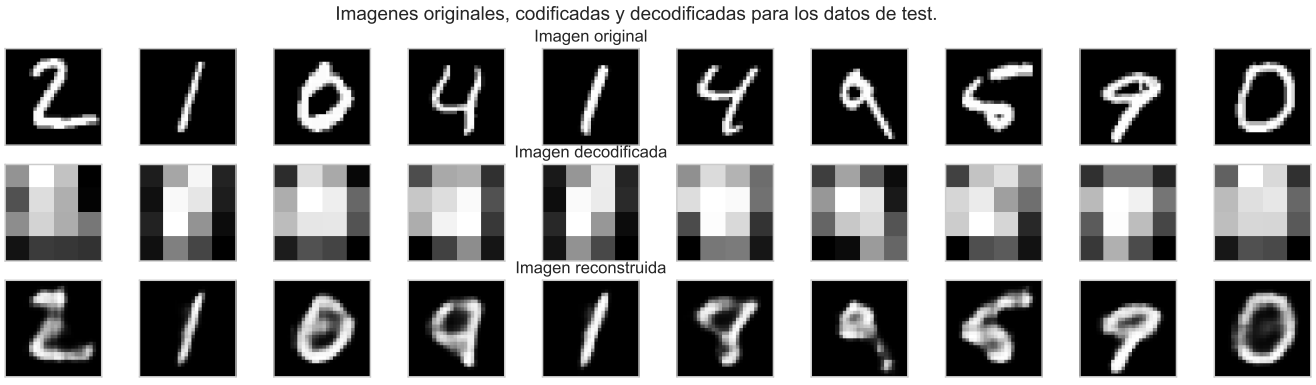

Figure 21. Visualización sobre los datos de test.

Luego, se realizó el mismo entrenamiento pero con los datos con ruido como entrada y los datos originales como salida, así se obtuvieron los resultados que se muestran en las **Figuras** 22 y 23.

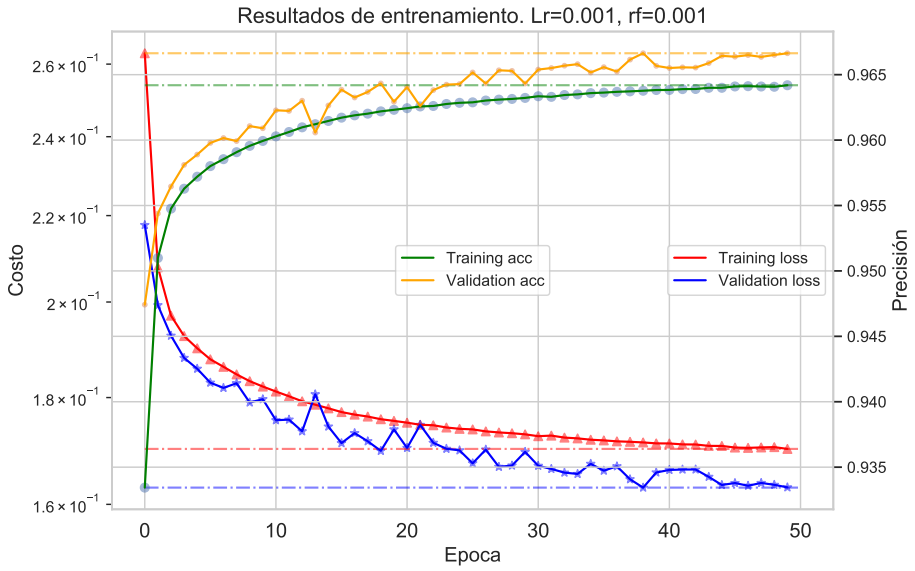

Figure 22. Resultados de entrenamiento para los datos con ruido.

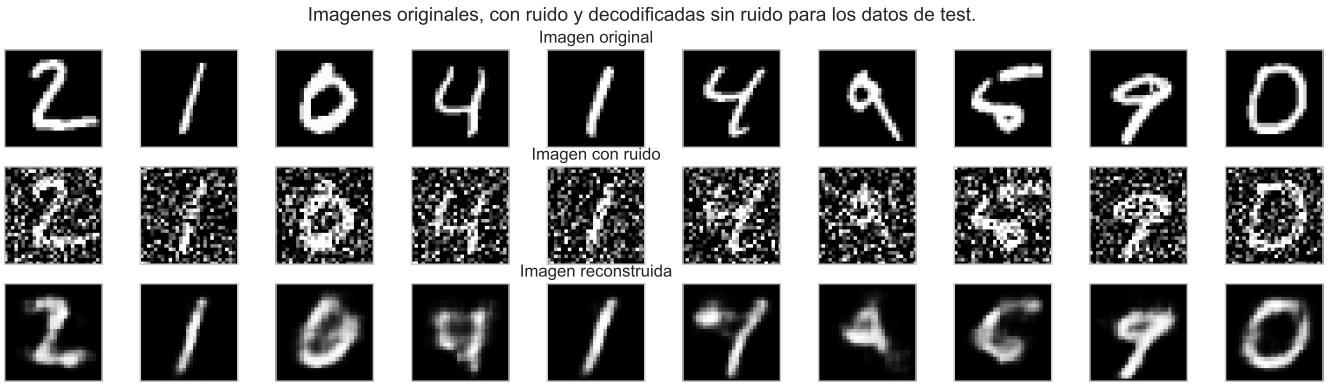

Figure 23. Visualización sobre los datos de test.

Ejercicio 8

Se propuso una arquitectura basada en capas densas y otra que utiliza capas convolucionales para clasificar los dígitos de la base de datos MNISTs. Basado en el libro de Michael Nielsen (http://neuralnetworksanddeeplearning.com/) se propusieron las siguientes arquitecturas:

- Red de capas densas:

```
Model: "Ejercicio_8_densa"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 784)               615440
_________________________________________________________________
batch_normalization (Batc (None, 784)               3136
_________________________________________________________________
dense_1 (Dense)              (None, 100)               78500
_________________________________________________________________
batch_normalization_1 (Batc (None, 100)               400
```

```
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 698,486
Trainable params: 696,718
Non-trainable params: 1,768
_________________________________________________________________
```

- Red de capas convolucionales:

```
Model: "Ejercicio_8_conv"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 10)        5770
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 10)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 10)        14410
_________________________________________________________________
flatten (Flatten)            (None, 1960)              0
_________________________________________________________________
dense (Dense)                (None, 10)                19610
=================================================================
Total params: 39,790
Trainable params: 39,790
Non-trainable params: 0
_________________________________________________________________
```

En donde se utilizó relu como función de activación de las capas ocultas y una activación lineal para la salida. La función costo fue BinaryCrossEntropy y la metrica de precisión BinaryAccuracy. Finalmente, se utilizó Adam como optimizador y L2 como regularizador en todas las capas. Así se obtuvieron los resultados que se muestran en las **Figuras 24** y **25**

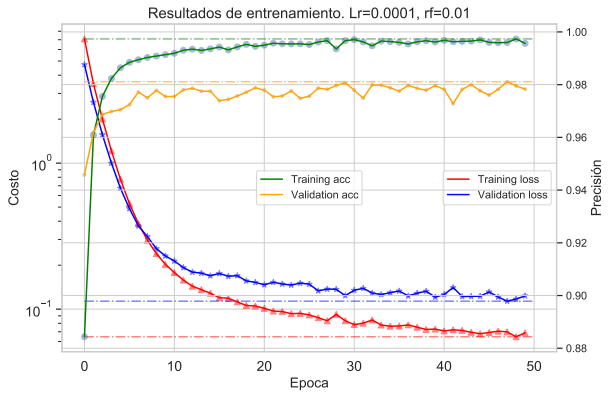

Figure 24. Resultados de entrenamiento para los datos de training y validación para la red densa

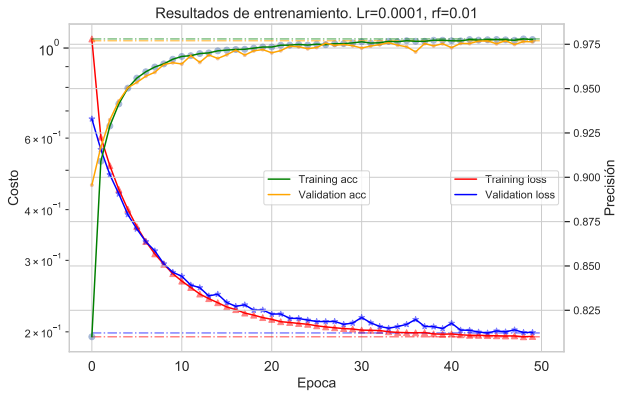

Figure 25. Resultados de entrenamiento y para los datos de training y validación para la red convolucional.

En donde se obtuvo un accuracy para los datos de test de 0.98 y 0.98 respectivamente, siendo la red convolucional la que tiene una mejor perfomance en cuanto a overfitting a pesar de tener menores estrategias de regularización y menores datos a entrenar. Luego se utilizó dropout como regularizador extra para la red densa, obtienendo la siguiente arquitectura:

```
Model: "Ejercicio_8_densa"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
dense (Dense)                (None, 784)               615440
_________________________________________________________________
batch_normalization (Batc (None, 784)               3136
_________________________________________________________________
dropout (Dropout)            (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               78500
_________________________________________________________________
batch_normalization_1 (Batc (None, 100)               400
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 698,486
Trainable params: 696,718
Non-trainable params: 1,768
_________________________________________________________________
```

En donde se obtuvieron la **Figura 26**.

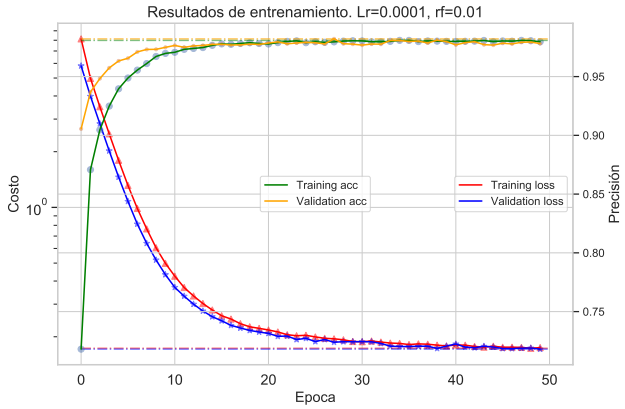

Figure 26. Resultados de entrenamiento para los datos de training y validación para la red densa con dropout.

En donde se obtuvo una precisión para los datos de test de 0.98 pero logrando reducir el overfitting que presentaba la anterior estrategia de regularización para la red densa.

Como se observa, no se utilizaron capas de dropout para las capas convolucionales La idea básica del mismo es remover activaciones individuales de forma azarosa mientras se entrena a la red para hacerla más robusta a los cambios de distintas piezas de entrenamiento y aprender menos ruido de los mismos. Las capas convolucionales tienen una robustez intrínseca al overfitting dado que los pesos compartidos implican que los filtros convolucionales están forzados a aprender a través de toda la imagen haciendolo menos propenso a aprender ruido local por lo que no necesitan demasiadas estrategias de regularización como dropout.

Finalmente, aunque la red neuronal densa no realiza ninguna asunción sobre los datos de entrada, tiende a tener una performance mas pobre o necesitar mayor cantidad de neuronas, esto conlleva a que en general los parámetros a entrenar son mucho mayores con respecto a las redes convolucionales. En particular para la clasificación de imágenes las redes neuronales convolucionales suelen tener una muy buena performance en donde la red se entrena para extraer las mejores características de cada imagen y por lo tanto requiere menos datos de entrenamiento aún así logrando una pérformance incluso superior.

Ejercicio 9

Con la misma implementación del ejercicio anterior se volvió a resolver el problema de clasificación de números manuscritos de MNIST pero con una misma permutación espacial en los pixeles para todas las imagenes como se muestra en la **Figura 27**.

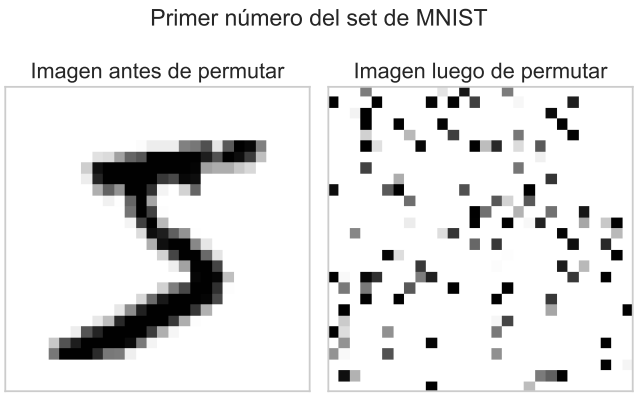

Figure 27. Imagen original y permutada para un número de mnist.

Para el caso de la red densa se utilizo la estrategia de regularización de L2+dropout+batch normalization. Así se obtuvieron los resultados que se muestran en la **Figura 28** y **29**.

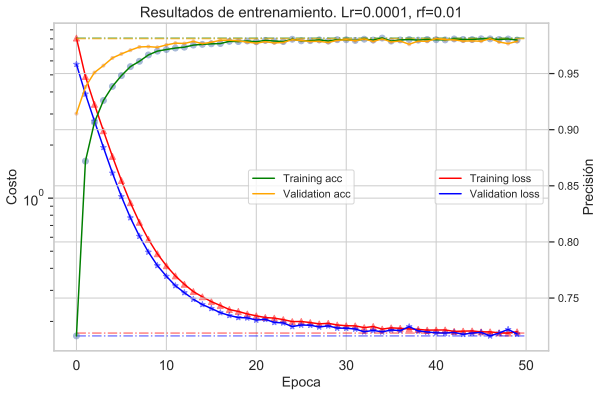

Figure 28. Resultados de entrenamiento para los datos de training y validación para la red densa

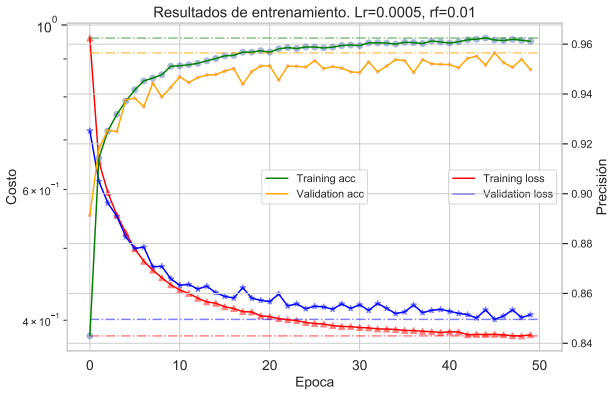

Figure 29. Resultados de entrenamiento y para los datos de training y validación para la red convolucional.

Se obtuvieron una precisión para los datos de test de 98% y 95% para la red densa y convolucional, respectivamente. Como se observa, el resultado para la red densa se vio practicamente inalterado por la permutación mientras que el accuracy para la red convolucional bajo en un 3% aproximadamente. Esto puede deberse a que el uso de filtros es menos efectivas por la perdida de información espacial de zonas locales que si estan presentes en las imagenes originales.

Ejercicio 10

Se implementó una red neuronal inspirada en AlexNet en donde se respetaró dicha arquitectura salvo por el tamaño del kernel de la primer capa convolucional:

```
    Model: "Ejercicio_9_alexnet"
_________________________________________________________________
Layer (type)                 Output Shape
    Param #
=================================================================
conv2d (Conv2D)              (None, 8, 8, 96)
    2688
_________________________________________________________________
```

```
batch_normalization (Batc (None, 8, 8, 96)
    384
_________________________________________________________________
max_pooling2d (MaxPooling (None, 4, 4, 96)
    0
_________________________________________________________________
conv2d_1 (Conv2D)           (None, 4, 4, 256)
    614656
_________________________________________________________________
batch_normalization_1 (Batc (None, 4, 4, 256)
    1024
_________________________________________________________________
max_pooling2d_1 (MaxPooling (None, 2, 2, 256)
    0
_________________________________________________________________
conv2d_2 (Conv2D)           (None, 2, 2, 384)
    885120
_________________________________________________________________
batch_normalization_2 (Batc (None, 2, 2, 384)
    1536
_________________________________________________________________
conv2d_3 (Conv2D)           (None, 2, 2, 184)
    636088
_________________________________________________________________
batch_normalization_3 (Batc (None, 2, 2, 184)
    736
_________________________________________________________________
conv2d_4 (Conv2D)           (None, 2, 2, 256)
    424192
_________________________________________________________________
batch_normalization_3 (Batc (None, 2, 2, 256)
    1024
_________________________________________________________________
max_pooling2d_2 (MaxPooling (None, 1, 1, 256)
    0
_________________________________________________________________
flatten (Flatten)          (None, 256)
    0
_________________________________________________________________
dropout (Dropout)          (None, 256)
    0
_________________________________________________________________
batch_normalization_4 (Batc (None, 256)
    1024
_________________________________________________________________
dense (Dense)              (None, 1024)
    263168
_________________________________________________________________
dropout_1 (Dropout)         (None, 1024)
    0
_________________________________________________________________
batch_normalization_5 (Batc (None, 1024)
    4096
_________________________________________________________________
dense_1 (Dense)             (None, 1024)
    1049600
_________________________________________________________________
dense_2 (Dense)             (None, 100)
    102500
=================================================================
```

```
Total params: 3,987,836
Trainable params: 3,982,924
Non-trainable params: 4,912
```

En donde se utilizó Adam como optimizador, CategoricalCrossEntropy como función de costo y CategoricalAccuracy como métrica de precisión para resolver el problema de clasificacion de CIFAR100. Se utilizó la función de tensorflow ImageDataGenerator para aumentarlos datos. Así se obtuvo el resultado que se muestra en la **Figura 30**.

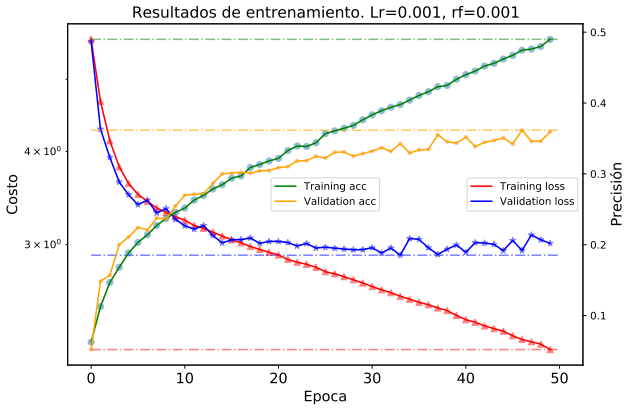

Figure 30. Resultados de entrenamiento para la adaptacion de AlexNet.

En donde se observa un gran overfitting todavia no se estabilizó el loss ni el accuracy en los datos de entrenamiento. Es porible que deba incrementarse el factor de regularización.

Luego se utilizó la red neuronal VGG16 en donde solo se modifico el numero de neuronas de las capas densas a 1/4 de las mismas para resolver el mismo problema. Entonces, la arquitectura fue:

```
Model: "Ejercicio_9_VGG16"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNo	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
batch_normalization_1 (Batch	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
batch_normalization_2 (Batch	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
batch_normalization_3 (Batch	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
batch_normalization_4 (Batch	(None, 8, 8, 128)	512

conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
batch_normalization_5 (Batch	(None, 8, 8, 128)	512
conv2d_6 (Conv2D)	(None, 8, 8, 128)	147584
batch_normalization_6 (Batch	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 128)	0
conv2d_7 (Conv2D)	(None, 4, 4, 256)	295168
batch_normalization_7 (Batch	(None, 4, 4, 256)	1024
conv2d_8 (Conv2D)	(None, 4, 4, 256)	590080
batch_normalization_8 (Batch	(None, 4, 4, 256)	1024
conv2d_9 (Conv2D)	(None, 4, 4, 256)	590080
batch_normalization_9 (Batch	(None, 4, 4, 256)	1024
max_pooling2d_3 (MaxPooling2	(None, 2, 2, 256)	0
conv2d_10 (Conv2D)	(None, 2, 2, 256)	590080
batch_normalization_10 (Batc	(None, 2, 2, 256)	1024
conv2d_11 (Conv2D)	(None, 2, 2, 256)	590080
batch_normalization_11 (Batc	(None, 2, 2, 256)	1024
conv2d_12 (Conv2D)	(None, 2, 2, 256)	590080
batch_normalization_12 (Batc	(None, 2, 2, 256)	1024
max_pooling2d_4 (MaxPooling2	(None, 1, 1, 256)	0
flatten (Flatten)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
batch_normalization_13 (Batc	(None, 256)	1024
dense (Dense)	(None, 6272)	1611904
dropout_1 (Dropout)	(None, 6272)	0
batch_normalization_14 (Batc	(None, 6272)	25088
dense_1 (Dense)	(None, 1024)	6423552
dropout_2 (Dropout)	(None, 1024)	0
batch_normalization_15 (Batc	(None, 1024)	4096
dense_2 (Dense)	(None, 1024)	1049600
dense_3 (Dense)	(None, 100)	102500

==========

Así se obtuvieron los resultados que se muestran en la **Figura 2**.

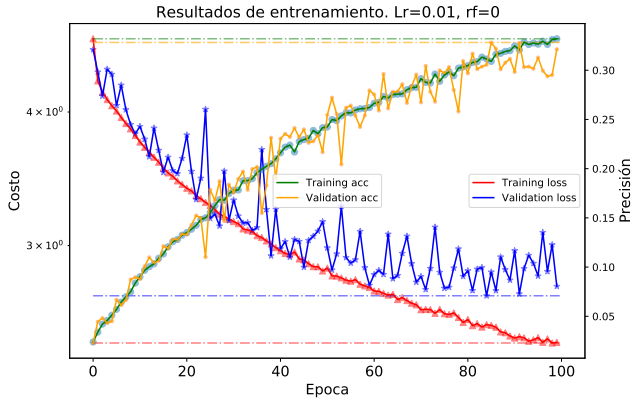

Figure 31. Resultados de entrenamiento para la adaptacion de VGG16.

En donde es posible que falten épocas para la convergencia