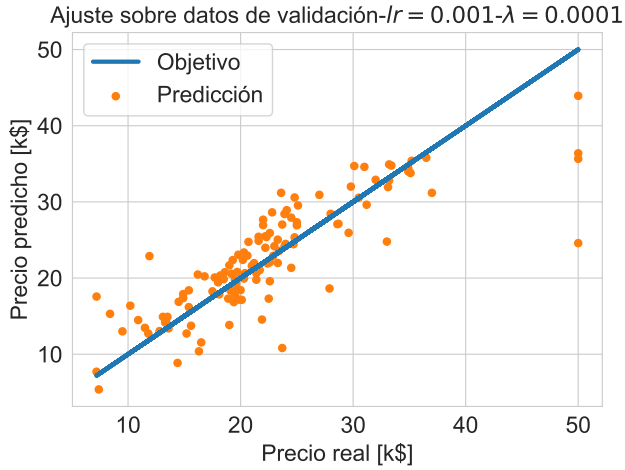
TP4

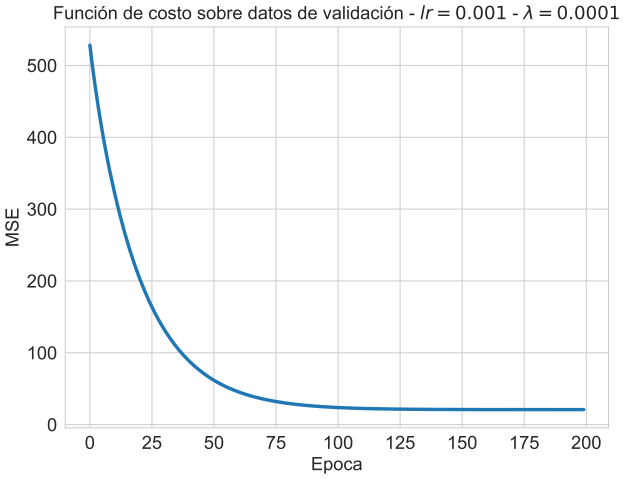
Tomás Hüttebräucker
Aprendizaje Profundo y Redes neuronales artificiales
(26 de octubre de 2020)

EJ1

Se entreno un clasificador lineal que utilice la base de datos sobre inmuebles de Boston de la libreria *sklearn* para predecir el precio de una casa en función de las caracteristicas de la misma.



(a) Predicción de precios de las casas comparados con los precios reales de las mismas



(b) Función de costo del modelo en función de la época de entrnamiento.

Figura 1: Predicción y función de costo para el clasificador lineal implementado para predecir el precio de alojamientos en la ciudad de Boston.

En la figura 1 se muestra la predicción final del clasificador lineal y la evolución de la función de costo. Se ve que aunque el modelo aprende, finalmente no termina realizando una muy buena predicción sobre los datos.

EJ2

Se implementaron los ejercicios 3, 4 y 6 de la práctica 2 utilizando la libreria *keras*. En general la implementación fue mucho mas simple y, además, los algoritmos se ejecutaron mucho mas rápido. Esto muestra que, claramente, la implementación de *keras* es mucho mejor que la implementada por el alumno.

3

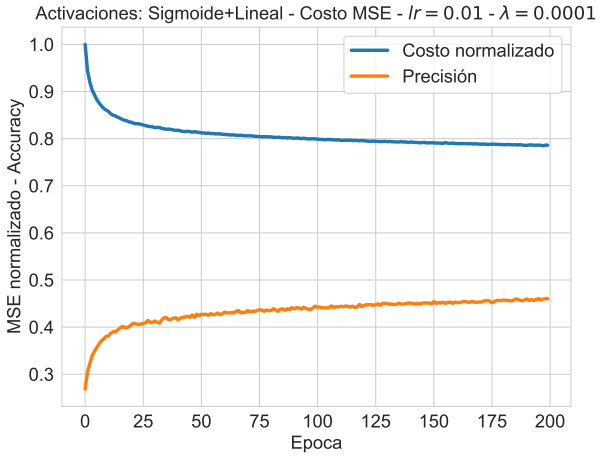


Figura 2: Función de costo normalizada y precisión del modelo del ejercicio 3 del TP2, implementado con *keras*.

4

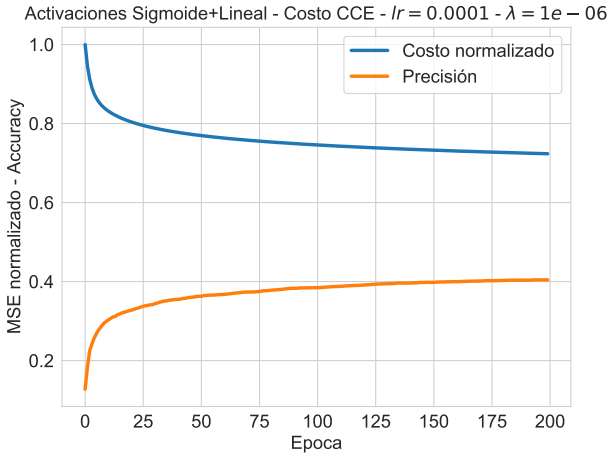
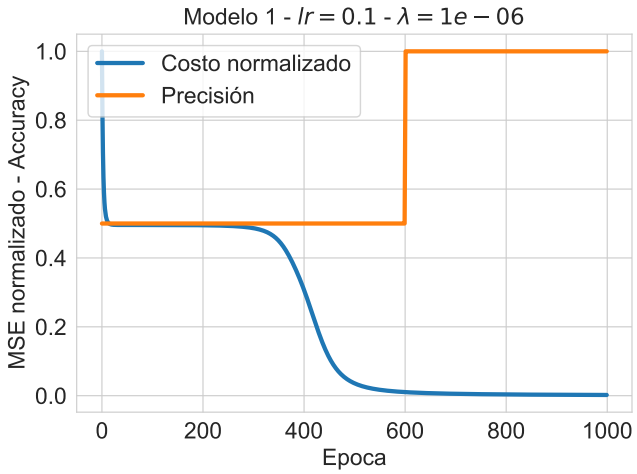
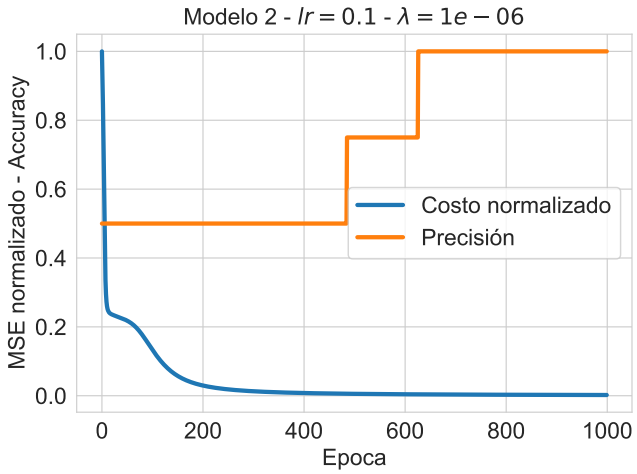


Figura 3: Función de costo normalizada y precisión del modelo del ejercicio 4 del TP2, implementado con *keras*.

6



(a) Modelo secuencial



(b) Modelo con capa de concatenación.

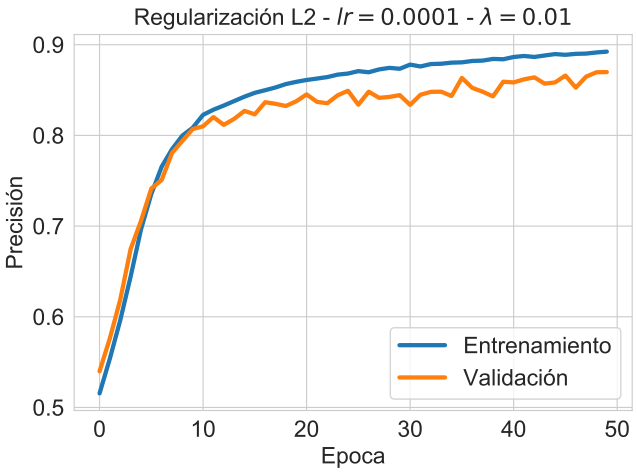
Figura 4: Función de costo normalizada y precisión del modelo del ejercicio 6 del TP2, implementado con *keras*.

EJ3

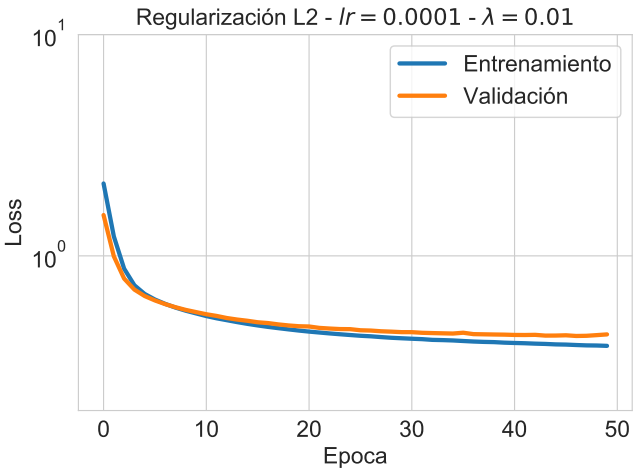
Se busco diseñar una red neuronal capaz de detectar si una pelicula recibió una mala o buena reseña en base a las palabras que contiene la misma. Se utilizó la base de datos de *IMDB* provista por keras. Los datos se preprocesaron llevandolos a una represenatación vectorial donde cada elemento del vector es una palabra y el valor de ese elemento es cuantas veces aparece la palabra en la reseña. Se diseño un modelo compuesto por tres capas ocultas, la primera de 100 neuronas con activación *ReLU*, la segunda de 10 neuronas con activación *ReLU* y la de salida de 1 neurona con activación lineal. Se uso una capa que concatene las entradas de la red con la salida de la capa de 10 neuronas. La función de costo elegida fue *BinaryCrossentropy* y el optimizador tipo *Adam*. Se usaron combinaciones de tres estrategias de regularización de los datos, regularización L2, *Batch Normalization* y *Drop Out*.

En la figura 5 se muestra la evolución de la función de costo y la precicisión del modelo utilizando regularización L2 a medida que pasan las épocas. Se ve que a pesar de utilizar la regularización, hay overfitting, esto puede deberse que el factor de regularización no es lo suficientemente grande.

En la figura 6 se muetra la evolución del modelo al usar *Drop Out* junto con regularización L2. Se ve que en en este caso hay una leve disminución del overfitting, pero en general el modelo tiene un desempeño peor que al utilizar solo regularización L2. Esto puede deberse a que el *Drop Out* introduce mas restricciónes al aumentar la regularización.



(a) Precisión

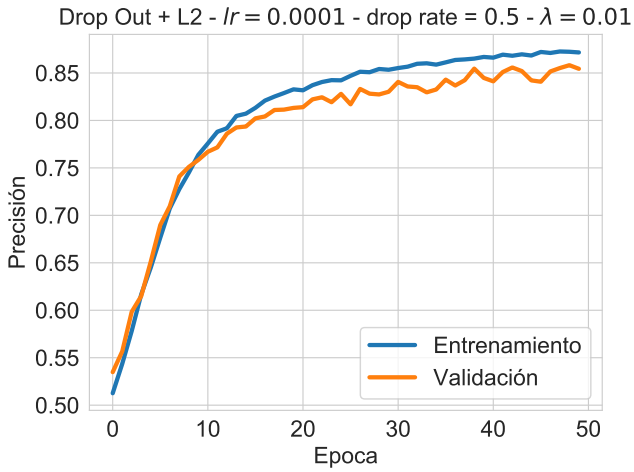


(b) Loss normalizada

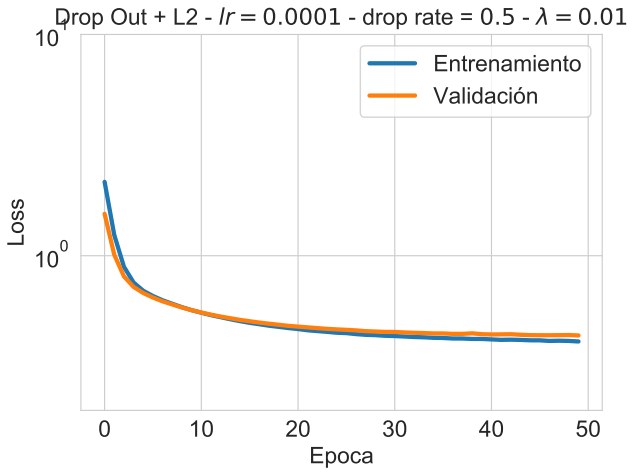
Figura 5: Precisión y loss normalizada del modelo al utilizar regularización L2.

En la figura 7 se muestra la evolución del modelo usando *Batch Normalization* junto con regularización L2. En este caso el rendimiento del modelo es mejor a los dos casos anteriores, llegando a los valores de precisión para los datos de entrenamiento mas altos, sin embargo presenta valores de precisión para los datos de validación similares al utilizar solo regularización L2. Es evidente que de esta forma se observa el overfitting mas alto, lo que puede deberse a que al utilizar el método *Batch Normalization*, el modelo converge mas rápido y comienza a ajustar cosas propias de los datos de entrenamiento y no del problema general. También se ve que al usar *Batch Normalization*

se puede usar un parámetro de regularización mayor sin arruinar el rendimiento de la red.



(a) Precisión



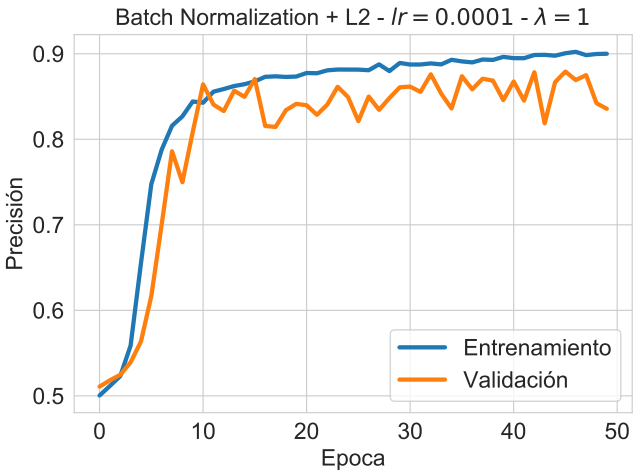
(b) Loss normalizada

Figura 6: Precisión y loss normalizada del modelo al utilizar *Drop Out* junto con regularización L2.

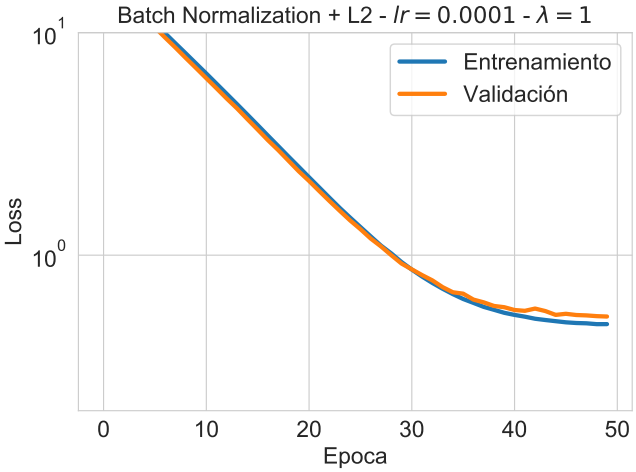
En la tabla 1 se muestran los valores de precisión obtenidos sobre los datos de testing para la arquitectura con las diferentes estrategias de normalización usadas. Se ve que las respuestas son

Método	**Acc. Test**
L2	0.87
DO+L2	0.85
BN+L2	0.82

Tabla 1: Valores de precisión sobre los datos de test para los diferentes modelos usados.



(a) Precisión



(b) Loss normalizada

Figura 7: Precisión y loss normalizada del modelo al utilizar *Batch Normalization* junto con regularización L2.

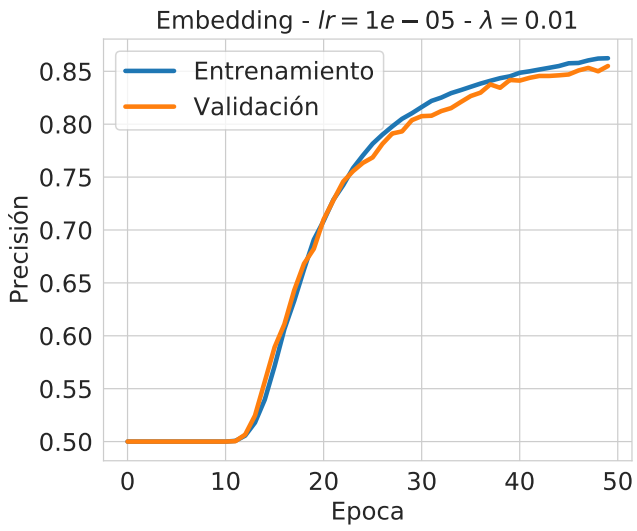
EJ4

Se busco resolver el problema del ejercicio 3 (clasificar reseñas de peliculas) usando ahora *embedding* sobre los datos. El *embedding* consiste en representar las palabras como vectores dentro de un espacio continuo de dimensión elegida. Los parámetros del *embedding* son *n_words* que es el numero de palabras totales usadas por la base de datos, *review_length* que es la longitud a las cuales se fijaron las reseñas (con *padding*) y *emb_dim* que es la dimensión del espacio donde se representaron las palabras.

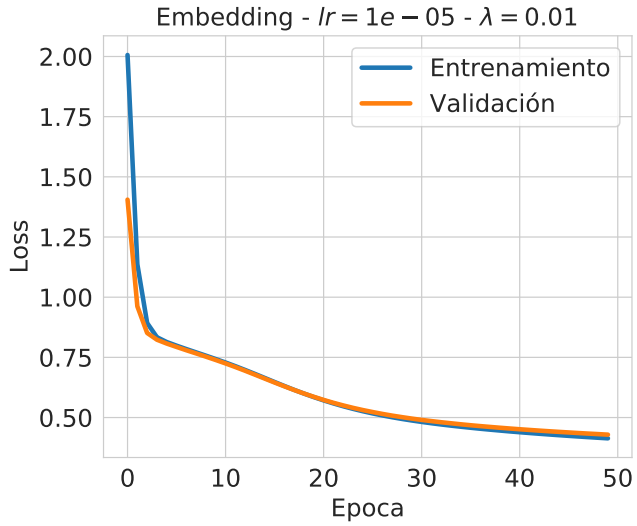
Embedding red densa

Los parámetros usados para el *embedding* fueron:

Parámetro	**Valor**
n_words	10000
review_leght	500
emb_dim	32



(a) Precisión



(b) Loss normalizada

Figura 8: Precisión y loss del modelo al utilizar *embedding*.

La arquitectura del modelo implementado es la siguiente:

```
_________________________________________________
Layer (type)                 Output Shape
=================================================
input (InputLayer)           [(None, 500)]
_________________________________________________
embedding (Embedding)        (None, 500, 32)
_________________________________________________
flatt (Flatten)              (None, 16000)
_________________________________________________
dense_1 (Dense)              (None, 100)
_________________________________________________
dropout_1 (Dropout)          (None, 100)
_________________________________________________
dense_2 (Dense)              (None, 10)
_________________________________________________
dropout_2 (Dropout)          (None, 10)
_________________________________________________
concatenate (Concatenate)    (None, 16010)
(input,dropout_2)
_________________________________________________
dense_9 (Dense)              (None, 1)
=================================================
Total params: 1,937,121
Trainable params: 1,937,121
Non-trainable params: 0
```

donde cada una de las capas densas tenía regularización L2. Esta arquitectura es igual a a la usada en el ejercicio 3, incluyendo como novedad la capa de *embedding*.

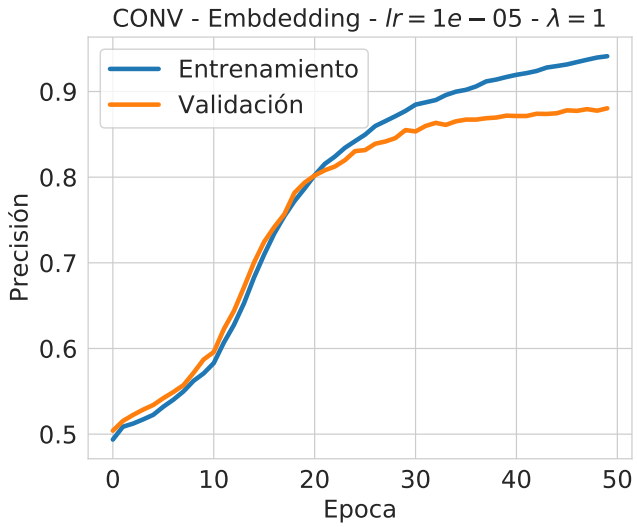
En la figura 8 se muestra la evolución del modelo. Se ve que el modelo tarda mas en aprender, su aprendizaje en las primeras epocas es nulo. El modelo crece rapidamente y en más épocas podria llegar a valores de precisión más altos y parece aprender mas rápido que al mismo modelo sin la presencia de la capa de *embedding*. Tambien se ve que el overfitting del modelo es menor al obtenido sin *embedding* (comparando con el modelo de Drop Out del ejercicio 3).

Embedding + red convolucional

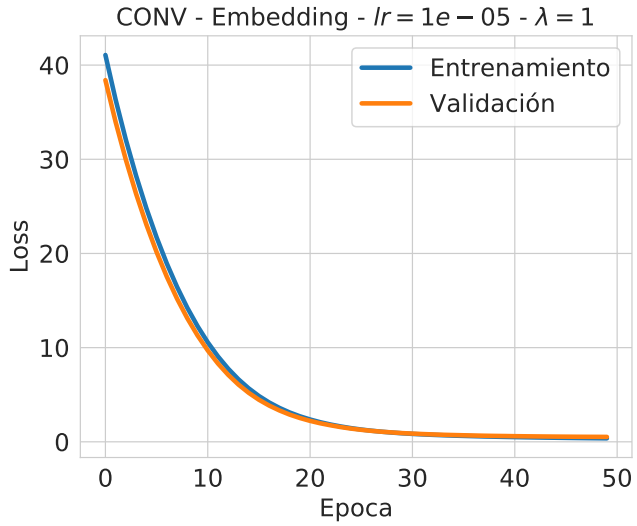
```
_________________________________________________
Layer (type)                 Output Shape
=================================================
embedding_1 (Embedding)      (None, 500, 16)
_________________________________________________
batch_normalization (BatchNo (None, 500, 16)
_________________________________________________
conv1d_1 (Conv1D)            (None, 500, 16)
_________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 250, 16)
_________________________________________________
batch_normalization_1 (Batch (None, 250, 16)
_________________________________________________
conv1d_2 (Conv1D)            (None, 250, 64)
_________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 125, 64)
_________________________________________________
batch_normalization_2 (Batch (None, 125, 64)
_________________________________________________
flatten_1 (Flatten)          (None, 8000)
_________________________________________________
dropout (Dropout)          (None, 8000)
_________________________________________________
dense_1 (Dense)              (None, 1)
=================================================
Total params: 172,305
Trainable params: 172,113
Non-trainable params: 192
```

Se entreno una red con la arquitectura mostrada arriba para resolver la clasificación de reseñas.

En la figura 9 se muestra la evolución de la red en el entrenamiento. Se ve que la red crece mas rápidamente y llega a valores de precisión más altos que la red densa implementada previamente. Sin embargo, también presenta mas *overfitting*. Esto puede deberse a la rápida convergencia de la red.

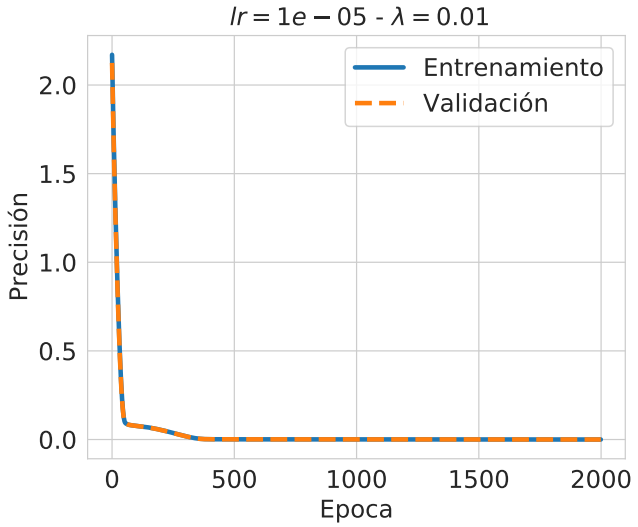


(a) Precisión

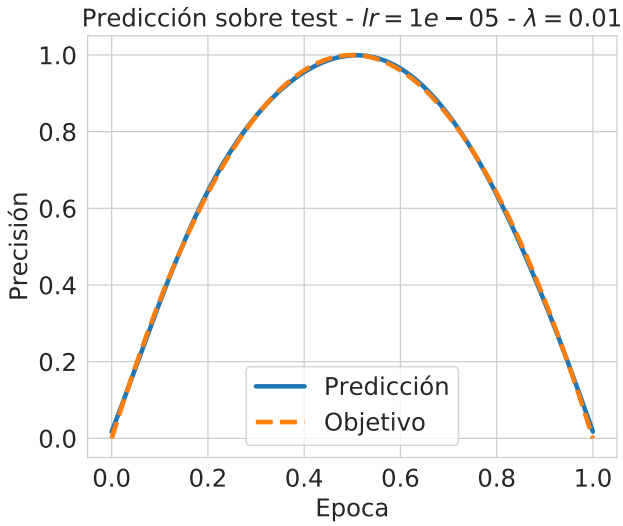


(b) Loss normalizada

Figura 9: Precisión y loss del modelo al utilizar *embedding*.



(a) MSE



(b) Predicción sobre datos de test.

Figura 10: MSE y predicción sobre los datos de entrenamiento del modelo.

EJ5

Se entreno una red con la arquitectura propuesta para resolver el mapeo $x(t) = 4x(t-1)(1-x(t-1))$.

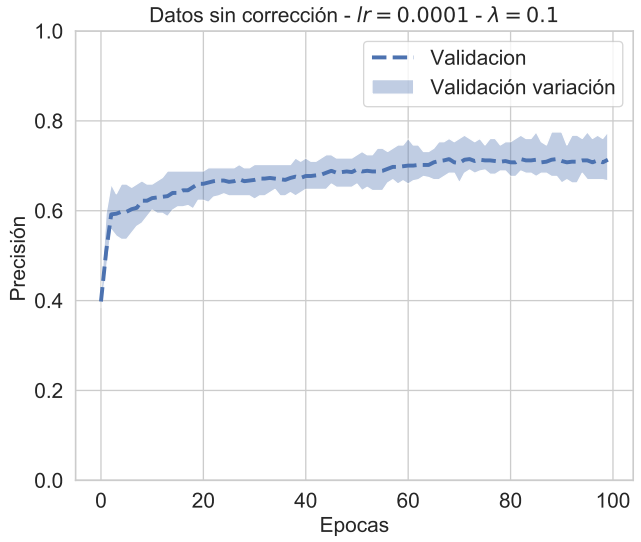
En la figura 10 se muestra el MSE a lo largo de las épocas de entrenamiento y la predicción sobre los datos de testing. Como se ve, el modelo funciona muy bien, prediciendo casi a la perfección.

EJ6

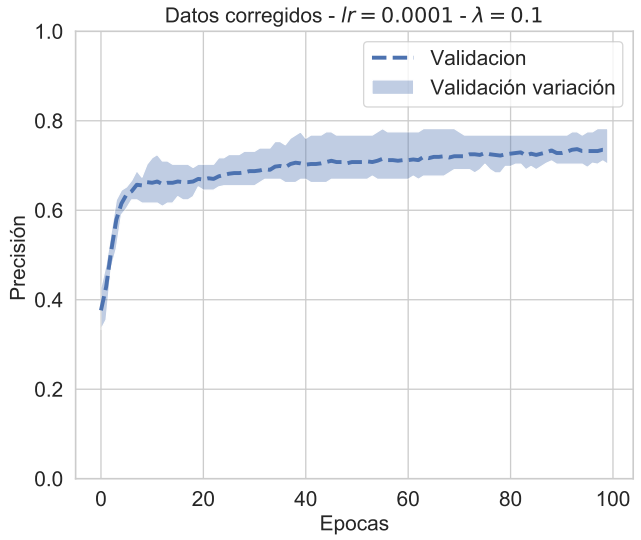
Se diseño una red con el objetivo de predecir, dada la información sobre un paciente, si este tiene o no diabetes. La arquitectura del modelo fue:

```
_________________________________________
Layer (type)                 Output Shape
=========================================
dense (Dense)                (None, 24)
_________________________________________
batch_normalization (BatchNo (None, 24)
_________________________________________
dense_1 (Dense)              (None, 12)
_________________________________________
batch_normalization_1 (Batch (None, 12)
_________________________________________
dense_2 (Dense)              (None, 1)
=========================================
Total params: 673
```

```
Trainable params: 601
Non-trainable params: 72
```



(a) Sin corrección



(b) Con corrección.

Figura 11: Precisión sobre datos de validación al usar datos con y sin correción.

Debido a que los datos presentaban ciertos parámetros erroneos (como nivel de insulina nulo), se realizó una corrección sobre los mismos, cambiando los valores nulos por la media de los datos (no se modificó el numero de embarazos del paciente ya que cero es un numero posible).

Se uso una estrategia de K-Folding, promediando sobre los diferentes modelos entrenados.

En la figura 11 se muestra la precisión sobre datos de validación media y su variación entre modelos. Se ve que al usar corrección el modelo parece ser menor ruidoso pero tiene un rendimiento muy similar.

EJ7

Se creo un *autoencoder* que permita filtrar ruido presente en imagenes de la base de datos de MNIST. La arquitectura de la red fue:

```
_________________________________________________
Layer (type)                 Output Shape
=================================================
input (InputLayer)          [(None, 28, 28, 1)]
_________________________________________________
conv2d (Conv2D)             (None, 28, 28, 32)
_________________________________________________
max_pooling2d (MaxPooling (None, 14, 14, 32)
_________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 32)
_________________________________________________
max_pooling2d_1 (MaxPooling (None, 7, 7, 32)
_________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 8)
_________________________________________________
up_sampling2d (UpSampling (None, 14, 14, 8)
_________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 8)
_________________________________________________
up_sampling2d_1 (UpSampling (None, 28, 28, 8)
_________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 1)
=================================================
Total params: 12,537
Trainable params: 12,537
Non-trainable params: 0
```

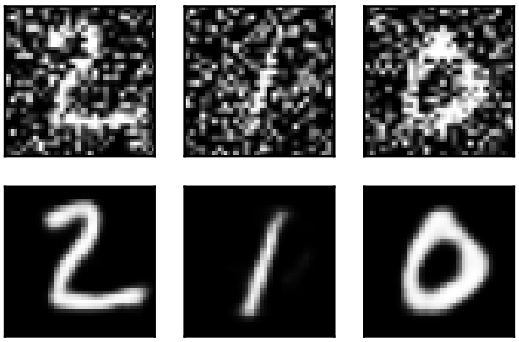


Figura 12: Tres ejemplos de imagenes ruidosas (arriba) y luego del *autoencoder* (abajo) de los datos de testing.

Como se ve en la figura 12, el filtro parece funcionar muy bien.

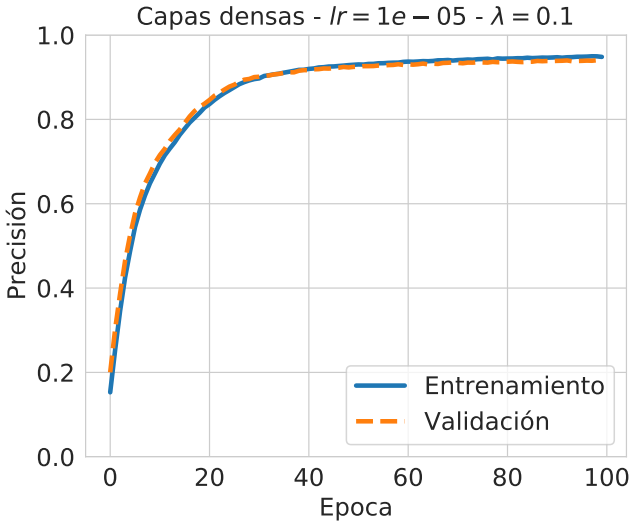
EJ8

Se entrenaron una red de capas densas y una red convolucional para resolver la clasificación de la base de datos MNIST. Las arquitecturas implementadas fueron:

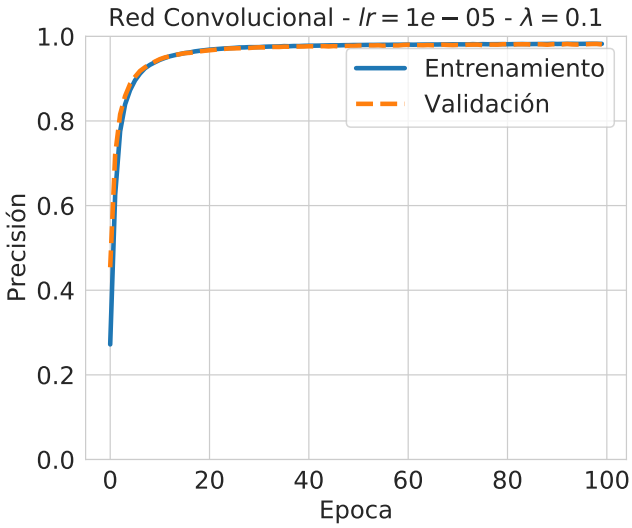
```
Model: "EJ8_Dense"
_________________________________________________
Layer (type)                 Output Shape
=================================================
dense_1 (Dense)              (None, 20)
_________________________________________________
batch_norm_1 (BatchNormaliza (None, 20)
_________________________________________________
dense_2 (Dense)              (None, 10)
_________________________________________________
batch_norm_2 (BatchNormaliza (None, 10)
_________________________________________________
dense_3 (Dense)              (None, 10)
=================================================
Total params: 16,140
Trainable params: 16,080
Non-trainable params: 60
```

```
Model: "EJ8_Conv"
_________________________________________________
Layer (type)                 Output Shape
=================================================
conv_1 (Conv2D)              (None, 28, 28, 32)
_________________________________________________
max_pool_1 (MaxPooling2D)    (None, 14, 14, 32)
_________________________________________________
batch_norm_1 (BatchNormaliza (None, 14, 14, 32)
_________________________________________________
conv_2 (Conv2D)              (None, 14, 14, 16)
_________________________________________________
max_pool_2 (MaxPooling2D)    (None, 7, 7, 16)
_________________________________________________
batch_norm_2 (BatchNormaliza (None, 7, 7, 16)
_________________________________________________
flatten_4 (Flatten)          (None, 784)
_________________________________________________
dense (Dense)                (None, 10)
=================================================
Total params: 12,986
Trainable params: 12,890
Non-trainable params: 96
```

En la figura 13 y 14 se muestra la precisión y la función de costo sobre los datos de entrenamiento y validación para las dos arquitecturas usadas. Se ve que la red convolucional tiene una convergencia mucho mas rápida que la red de capas densas y llega a un valor de precisión mas alto, además, la red de capas densa presenta mayor overfitting, ya que la precisión sobre los datos de validación es menor y crece menos lento que la precisión sobre los datos de entrenamiento. Esto tiene sentido ya que los datos son imagenes.



(a) Red de capas densas.



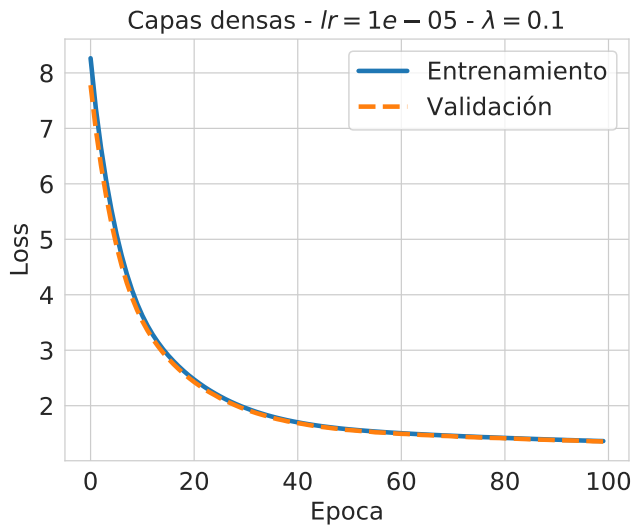
(b) Red convolucional.

Figura 13: Precisión sobre datos de entrenamiento y validación para las dos arquitecturas propuestas.

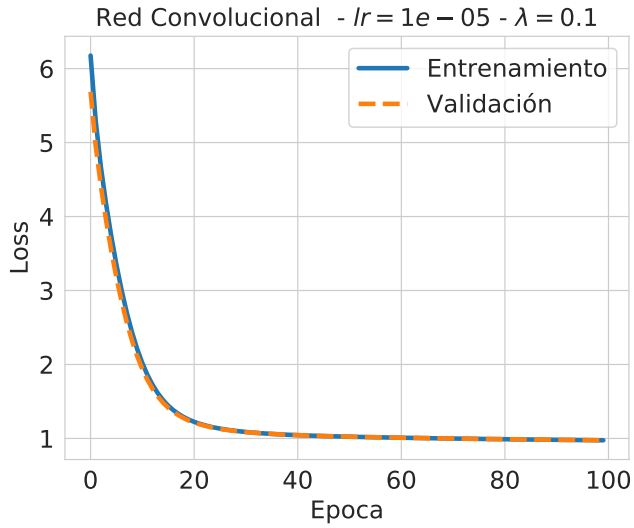
En la tabla 2 se muestran las precisiones de los modelos sobre los datos de test. Como se ve, la red convolucional tiene mejores resultados.

Arquitectura	**Acc. Test**
Capas densas	0.95
Convolucional	0.98

Tabla 2: Valores de precisión sobre los datos de test para los diferentes modelos usados.



(a) Red de capas densas.



(b) Red convolucional.

Figura 14: Precisión sobre datos de entrenamiento y validación para las dos arquitecturas propuestas.

EJ9

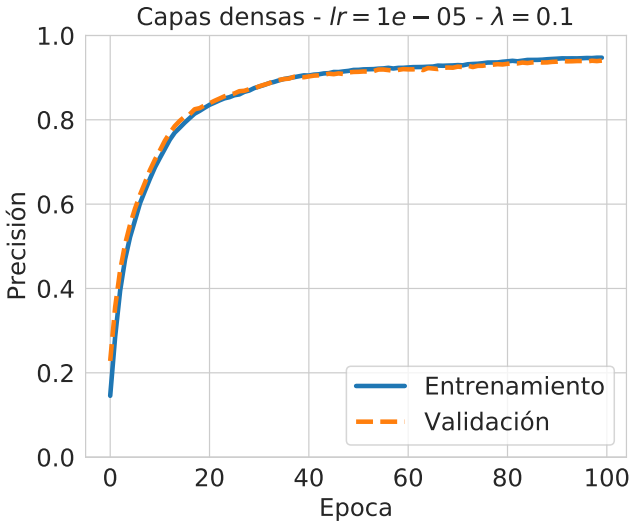
Con las mismas arquitecturas usadas en el ejercicio 8, se resolvió la clasificación de MNIST, pero introduciendo una permutación aleatoria de los datos.

En las figuras 15 y 16 se muestra la evolución del modelo. Se ve que el rendimiento de la red de capas densas no se ve notablemente alterado. En cambio, la red convolucional no llega al mismo nivel de precisión, comparada con la red usando datos no permutados. Esto tiene sentido ya que las redes convolucionales detectan ciertas características espaciales de varios pixeles de las imágenes, y al, realizar una permutación, esas características se pierden.

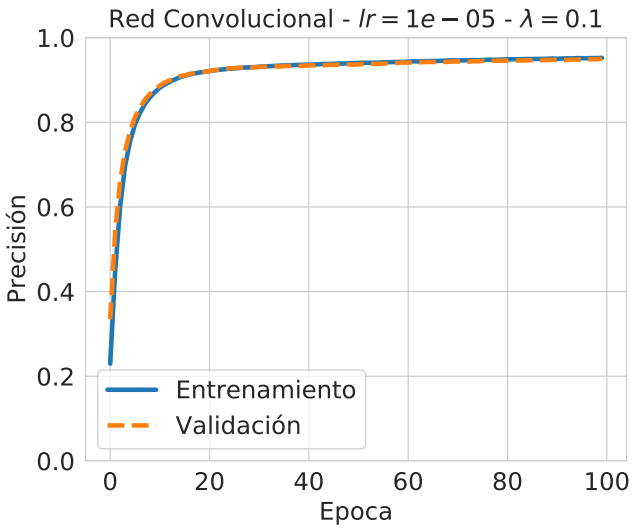
En la tabla 3 se muestran las precisiones de los modelos sobre los datos de test. Como se ve, la red convolucional tiene nuevamente mejores resultados, pero notablemente peor que el obtenido en el ejercicio 8.

Arquitectura	**Acc. Test**
Capas densas	0.94
Convolucional	0.95

Tabla 3: Valores de precisión sobre los datos de test para los diferentes modelos usados con los datos permutados.

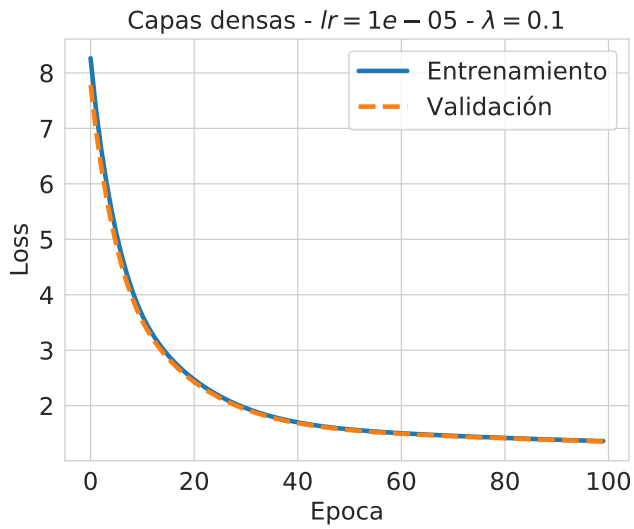


(a) Red de capas densas.

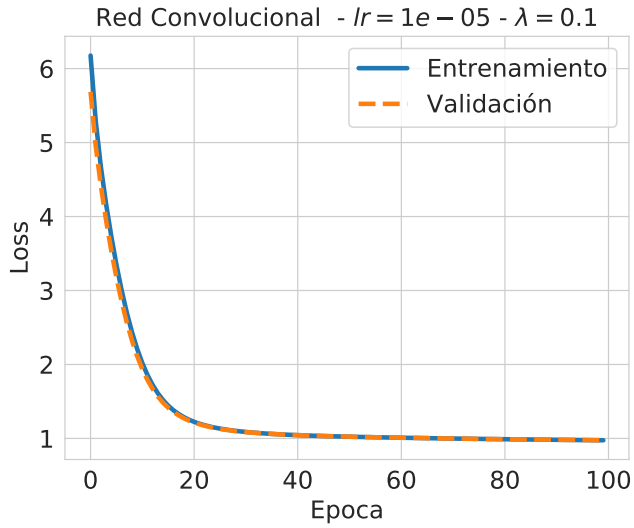


(b) Red convolucional.

Figura 15: Precisión sobre datos de entrenamiento y validación para las dos arquitecturas propuestas con los datos permutados.



(a) Red de capas densas.



(b) Red convolucional.

Figura 16: Precisión sobre datos de entrenamiento y validación para las dos arquitecturas propuestas con los datos permutados.

EJ10

Se entrenaron dos redes inspiradas en las arquitecturas AlexNet y VGG16 para resolver la clasificación CIFAR100. Se uso aumento de datos para mejorar el rendimiento de los modelos.

AlexNet

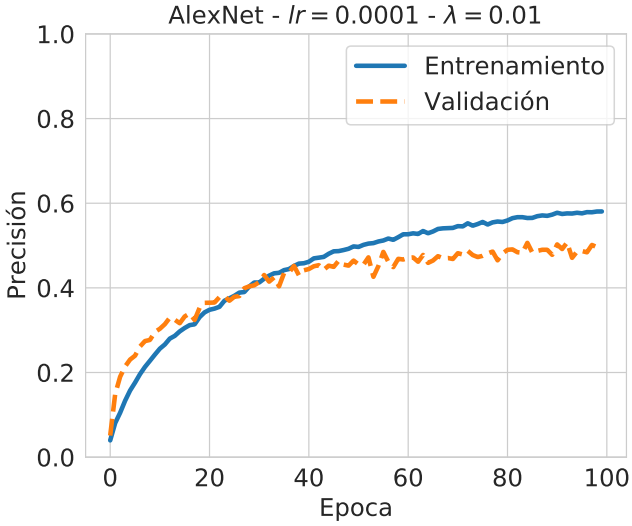
La arquitectura implementada fue:

```
Model: "AlexNet"
_________________________________________________
Layer (type)                 Output Shape
=================================================
conv2d (Conv2D)              (None, 15, 15, 96)
_________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 96)
_________________________________________________
batch_normalization (BatchNo (None, 13, 13, 96)
_________________________________________________
conv2d_1 (Conv2D)            (None, 6, 6, 256)
_________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 256)
_________________________________________________
batch_normalization_1 (Batch (None, 5, 5, 256)
_________________________________________________
conv2d_2 (Conv2D)            (None, 5, 5, 384)
_________________________________________________
batch_normalization_2 (Batch (None, 5, 5, 384)
_________________________________________________
conv2d_3 (Conv2D)            (None, 5, 5, 384)
_________________________________________________
batch_normalization_3 (Batch (None, 5, 5, 384)
_________________________________________________
conv2d_4 (Conv2D)            (None, 5, 5, 256)
_________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 256)
_________________________________________________
batch_normalization_4 (Batch (None, 4, 4, 256)
_________________________________________________
flatten (Flatten)            (None, 4096)
_________________________________________________
dense (Dense)                (None, 512)
_________________________________________________
dropout (Dropout)            (None, 512)
_________________________________________________
batch_normalization_5 (Batch (None, 512)
_________________________________________________
dense_1 (Dense)              (None, 512)
_________________________________________________
dropout_1 (Dropout)          (None, 512)
_________________________________________________
batch_normalization_6 (Batch (None, 512)
_________________________________________________
dense_2 (Dense)              (None, 100)
=================================================
Total params: 5,744,964
Trainable params: 5,740,164
Non-trainable params: 4,800
```

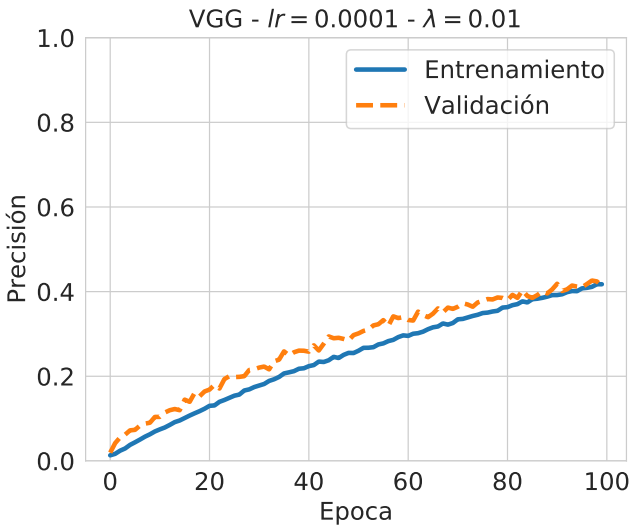
VGG-16

```
Model: "VGG16"
_________________________________________________
Layer (type)                 Output Shape
=================================================
batch_normalization (BatchNo (None, 32, 32, 3)
_________________________________________________
conv2d (Conv2D)              (None, 32, 32, 32)
_________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)
_________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)
_________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)
_________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 32)
_________________________________________________
```

```
conv2d_2 (Conv2D)            (None, 16, 16, 64)
_________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)
_________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)
_________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 64)
_________________________________________________
batch_normalization_4 (Batch (None, 15, 15, 64)
_________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)
_________________________________________________
batch_normalization_5 (Batch (None, 15, 15, 128)
_________________________________________________
conv2d_5 (Conv2D)            (None, 15, 15, 128)
_________________________________________________
batch_normalization_6 (Batch (None, 15, 15, 128)
_________________________________________________
conv2d_6 (Conv2D)            (None, 15, 15, 128)
_________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 128)
_________________________________________________
batch_normalization_7 (Batch (None, 14, 14, 128)
_________________________________________________
conv2d_7 (Conv2D)            (None, 14, 14, 256)
_________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 256)
_________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 256)
_________________________________________________
batch_normalization_9 (Batch (None, 14, 14, 256)
_________________________________________________
conv2d_9 (Conv2D)            (None, 14, 14, 256)
_________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 256)
_________________________________________________
batch_normalization_10 (Batc (None, 13, 13, 256)
_________________________________________________
conv2d_10 (Conv2D)           (None, 13, 13, 256)
_________________________________________________
batch_normalization_11 (Batc (None, 13, 13, 256)
_________________________________________________
conv2d_11 (Conv2D)           (None, 13, 13, 256)
_________________________________________________
batch_normalization_12 (Batc (None, 13, 13, 256)
_________________________________________________
conv2d_12 (Conv2D)           (None, 13, 13, 128)
_________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 12, 12, 128)
_________________________________________________
flatten (Flatten)            (None, 18432)
_________________________________________________
dense (Dense)                (None, 128)
_________________________________________________
dropout (Dropout)            (None, 128)
_________________________________________________
batch_normalization_13 (Batc (None, 128)
_________________________________________________
dense_1 (Dense)              (None, 128)
_________________________________________________
dropout_1 (Dropout)          (None, 128)
_________________________________________________
batch_normalization_14 (Batc (None, 128)
_________________________________________________
dense_2 (Dense)              (None, 128)
_________________________________________________
dropout_2 (Dropout)          (None, 128)
_________________________________________________
batch_normalization_15 (Batc (None, 128)
_________________________________________________
dense_3 (Dense)              (None, 100)
=================================================
Total params: 5,799,440
Trainable params: 5,794,954
Non-trainable params: 4,486
```

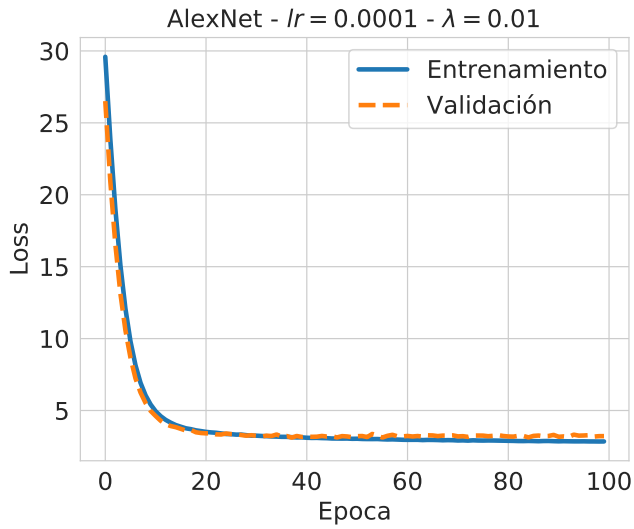


(a) AlexNet

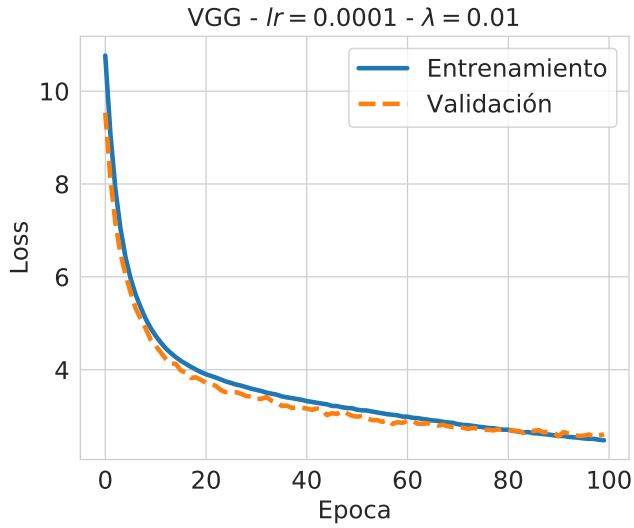


(b) VGG16.

Figura 17: Precisión sobre datos de entrenamiento y validación de CIFAR100 para las dos arquitecturas propuestas.



(a) AlexNet



(b) VGG16.

Figura 18: Función de costo sobre datos de entrenamiento y validación de CIFAR100 para las dos arquitecturas propuestas.

En las figuras 17 y 18 se muestran la precisión y la función de costo del modelo sobre los datos de entrenamiento y validación a lo largo de las épocas. A pesar de conseguir mejores resultados más rápido, la red AlexNet presenta mayor overfitting. Mientras tanto, la red VGG aprende mas lento, pero presenta menor overfitting y parecería que tiene mas potencial de aprendizaje, si se la deja un mayor tiempo entrenando, es posible que supere a la arquitectura AlexNet.

En la tabla 4 se muestra la precisión del modelo sobre los datos de test de CIFAR100

Arquitectura	**Acc. Test**
AlexNet	0.50
VGG16	0.42

Tabla 4: Valores de precisión sobre los datos de test de CIFAR100 para los diferentes modelos usados.