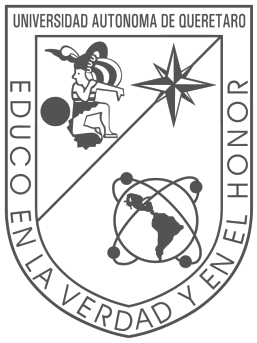
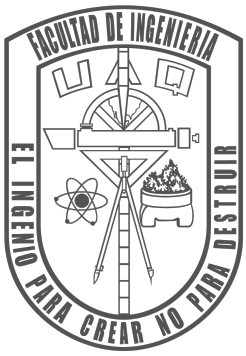
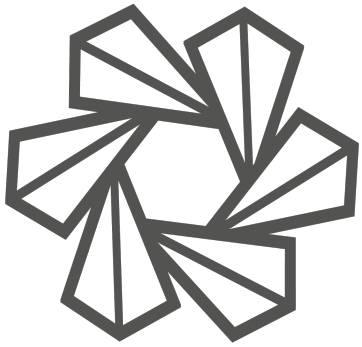
Universidad Autónoma de Querétaro

Facultad de Ingeniería
División de Investigación y Posgrado







Reporte 5

Red neuronal Perceptrón Multicapa

Maestría en Ciencias en Inteligencia Artificial
Optativa de especialidad II - Deep Learning

Aldo Cervantes Marquez
Expediente: 262775
Profesor: Dr. Sebastián Salazar Colores

Santiago de Querétaro, Querétaro, México
Semestre 2022-2
13 de Septiembre de 2022

Índice

1. Introducción

La presente práctica consiste en implementar una red neuronal Perceptrón multicapa, con el fin de poder clasificar un *dataset* que consiste en imágenes de números en escala de grises. Esto significa que se analizará píxel a píxel, su comportamiento de cada número. Dicha base de datos se encuentra embebida en la librería de $tensorflow$. Por lo que la práctica implicará el uso de redes neuronales, así como tambien el uso de conceptos anteriormente vistos.
Todos los elementos teóricos de esta sección fueron obtenidos de [1, 2] y de los apuntes de la clase.

1.1. Red neuronal perceptrón

Las redes neuronales son un método de inteligencia artificial que permite la capacidad de predecir comportamientos [], el cual se basa en el funcionamiento del cerebro humano mediante neuronas.
Las redes neuronales constan básicamente de 3 partes que son (véase Figura 1):

- **Capa de entrada**: consistentes de los valores de entrada del modelo.
- **Capas ocultas**: consisten en los conjuntos de neuronas intermedias.
- **Capa de salida**: consiste de la capa de salidas de la red (valores esperados)

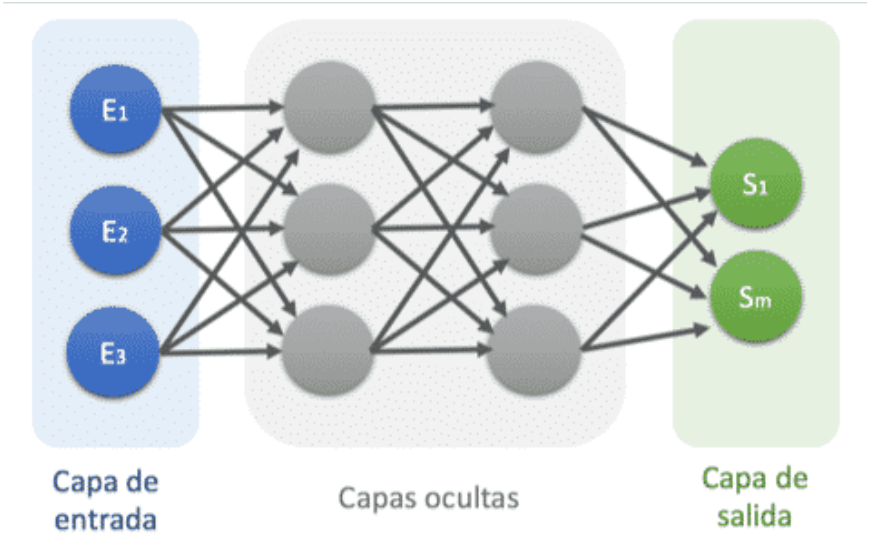


Figura 1: Estructura de red neuronal.

La nomenclatura a utilizar para representar en forma vectorial una red neuronal es la siguiente:

$$r_n = [n_1(f_{a1}), n_2(f_{a2}), n_3(f_{a3}), \ldots, n_p(f_{ap})] \tag{1}$$

en donde la posición de la lista será equivalente a la posición de la capa $pos(p)$ (excluyendo las capas de entrada, por lo que se iniciará por una capa oculta) con la función de activación f_{ap} y siendo la ultima posición $pos(final)$ con el numero de neuronas n de la capa de salida.

Por otro lado, la red neuronal de tipo perceptrón consta de la siguiente arquitectura Obteniendo una ecuación de recta como se muestra a continuación (vease Figura 2).

Figura 2: Perceptrón monocapa.

Por lo que se puede generalizar que la neurona tiene un comportamiento:

$$(\sum_{i=1}^{n}(\omega_i x_i)) + b \tag{2}$$

en donde se puede destacar que se tiene una sumatoria de pesos por los valores de la variable x y finalmente sumándole un bias (b), algo análogo a θ_0 o termino independiente. Esto es algo parecido al valor de z del algoritmo de gradiente descendente.

Posteriormente se tiene la función de activación, la cual es variable según el caso que se requiera. Para este caso, la librería de $tensorflow$ cuenta con las siguientes funciones de activación [3].

1. deserialize(...): Returns activation function given a string identifier.
2. elu(...): Exponential Linear Unit.
3. exponential(...): Exponential activation function.
4. gelu(...): Applies the Gaussian error linear unit (GELU) activation function.
5. get(...): Returns function.
6. hard_sigmoid(...): Hard sigmoid activation function.
7. linear(...): Linear activation function (pass-through).
8. relu(...): Applies the rectified linear unit activation function.
9. selu(...): Scaled Exponential Linear Unit (SELU).
10. serialize(...): Returns the string identifier of an activation function.
11. sigmoid(...): Sigmoid activation function, sigmoid(x) = 1 / (1 + exp(-x)).
12. softmax(...): Softmax converts a vector of values to a probability distribution.
13. softplus(...): Softplus activation function, softplus(x) = log(exp(x) + 1).
14. softsign(...): Softsign activation function, softsign(x) = x / (abs(x) + 1).
15. swish(...): Swish activation function, swish(x) = x * sigmoid(x).
16. tanh(...): Hyperbolic tangent activation function.

1.1.1. Cálculo del exactitud

Existen criterios para observar que tan bueno es el ajuste hecho, y se basa en la medición de la exactitud del sistema en función de la predicción y el valor verdadero.

$$exactitud = \frac{n_{casos favorables}}{n_{totales}} \times 100\,\% \tag{3}$$

Como se observa, es una división de los valores acertados (es decir, valores en los que se obtiene el mismo resultado al esperado) entre la cantida de valores existentes.

1.2. Función de perdida

El uso de entropía cruzada es la más adecuada para este problema pues se adecua mejor para problemas de clasificación como este, esta representada como:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i log(h_\theta(x_i)) + (1 - yî) log(1 - h_\theta(x_i))] \tag{4}$$

1.2.1. Optimizadores

La idea es reducir la función de perdida (4), por lo que la librería de $tensorflow$ cuenta con los siguientes optimizadores.

1. class Adadelta: Optimizer that implements the Adadelta algorithm.
2. class Adagrad: Optimizer that implements the Adagrad algorithm.
3. class Adam: Optimizer that implements the Adam algorithm.
4. class Adamax: Optimizer that implements the Adamax algorithm.
5. class Ftrl: Optimizer that implements the FTRL algorithm.
6. class Nadam: Optimizer that implements the NAdam algorithm.
7. class Optimizer: Base class for Keras optimizers.
8. class RMSprop: Optimizer that implements the RMSprop algorithm.
9. class SGD: Gradient descent (with momentum) optimizer.

1.3. Normalización de los datos

Otra parte importante es la normalización de datos y en este caso, se tenia un conjunto que tenia valores entre 0 y 255, por lo que se realizó la operación siguiente para cada elemento de la base datos.

$$dato_i := \frac{dato_i}{max(datos)} \tag{5}$$

Obteniendo datos en el intervalo $[0, 1]$.

1.4. One Hot

Esta técnico permite categorizar los valores numéricos de la salida (y) en un valor discreto posicionado en un lugar referente a su valor originial.

Para realizar esto, se debe observar el rango de valores en los que se tienen los valores de y. En este caso se tiene un intervalo de $[0, 9]$ lo que implican 10 posibilidades de respuesta. Entonces el *onehot* será de $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

1.5. División de datos

En este caso se utilizará una división de 60 %, 20 %, 20 % para dividir el entrenamiento, validación y prueba. Sin embargo, también se utilizará la técnica de *batch*, en la cual se tomará una muestra del 60 % de los datos como entrenamiento, equivalente a $\frac{1}{x} \times d_{entrenamiento}$, debido a la alta cantidad de datos que tiene la base de datos.

2. Justificación

El uso de redes neuronales para la predicción de problemas multivariables ha sido de mucho interés en el campo de la inteligencia artificial, por lo que su uso e implementación en la clasificación de objetos es altamente requerido [4]. Pudiendo lograr abstracciones que una persona normal o algun algoritmo matematico no pudiera resolver. Se obtiene una respuesta rápida y es fácil de interpretar [5].

3. Resultados

Para esta sección, se mostrará una tabla que pueda mostrar los aspectos más importantes de la red neuronal propuesta.

Tabla 1: Resultados de red neuronal perceptrón multicapa.

No.	Estructura	Optimizador	Tasa de aprendizaje	Épocas	Batch	Exactitud final entrenamiento	Exactitud final validación	Exactitud final prueba	Tiempo de ejecución
1	[25(s),7(s),10(s),10(smax)]	SGD	0.01	50	$\frac{1}{32}$	42.69%	42.78%	42.55%	60.6s
2	[25(s),10(s),10(smax)]	SGD	0.01	50	$\frac{1}{64}$	88.72%	88.22%	88.38%	30.22s
3	[250(s),100(s),10(smax)]	SGD	0.001	50	$\frac{1}{16}$	88.71%	88.50%	88.3%	116.44s
4	[35(s),20(s), 20(s),10(smax)]	Adam	0.01	50	$\frac{1}{32}$	98.70%	95.55%	95.72%	63.92s
5	[7(s),7(s),20(s), 20(s),10(smax)]	Adam	0.01	50	$\frac{1}{32}$	91.88%	88.75%	89.12%	67.06s
6	[250(s),150(s), 100(s),10(smax)]	Adam	0.001	50	$\frac{1}{64}$	100%	97.98%	98.18%	51.63s

Donde:

- **s**: sigmoide
- **smax**: softmax

4. Conclusiones

El uso de neuronas perceptrón multicapa para el desarrollo de esta práctica, han permitido que se logrará entender, la estructura y las configuraciones de las mismas. Se observó que las configuraciones para este problema, tenían mejores resultados conforme más neuronas se ponían en la capa, más no exactamente si se ponían muchas capas. Por lo que considero eso un punto muy importante a considerar según la aplicación. De igual modo, observo que al incrementar las capas, también se debieron incrementar las épocas para poder obtener un mejor resultado pero eso afectaba gravemente el tiempo de ejecución. Cabe resaltar que los tiempos de ejecución se realizaron con la ejecución del código de mi computadora con buenas características (procesador Ryzen 7 5000 y Tarjeta gráfica Nvidia RTX 3050). Cabe mencionar que los mejores resultados se obtuvieron con el algoritmo de optimización de Adam, además de obtener el mejor tiempo de ejecución en relación a los resultados obtenidos en la prueba. Por otro lado el algoritmo de optimización de SGD no obtuvo resultados tan favorables, sin embargo, no descarto que pueda tener resultados mejores, pero se deberán realizar más pruebas.

Referencias

[1] "Perceptrón - red neuronal - diego calvo." https://www.diegocalvo.es/perceptron/. (Accessed on 09/12/2022).

[2] "Perceptrón multicapa - red neuronal - diego calvo." https://www.diegocalvo.es/perceptron-multicapa/. (Accessed on 09/12/2022).

[3] "Module: tf.keras.activations — tensorflow v2.10.0." https://www.tensorflow.org/api_docs/python/tf/keras/activations. (Accessed on 09/12/2022).

[4] "View of statistical control of multivariant processes through the artificial neural network multilayer perceptron and the mewma graphic analysis." https://latamt.ieeer9.org/index.php/transactions/article/view/1284/491. (Accessed on 09/12/2022).

[5] "¿qué es una red neuronal? guía de ia y ml - aws." https://aws.amazon.com/es/what-is/neural-network/#:~:text=Las%20redes%20neuronales%20pueden%20ayudar,lineales%20y%20que%20son%20complejos. (Accessed on 09/12/2022).

practica perceptron multi capa

September 12, 2022

1 Tratamiento de datos

Se utilizará la división de datos de la forma:

Entrenamiento 60%

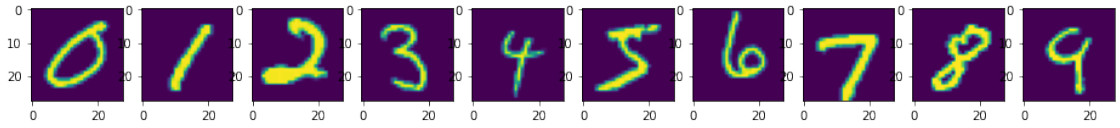
Validación 20%

Prueba 20%

[81]:
```
import  tensorflow as tf

from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
import numpy as np
from sklearn.model_selection import train_test_split#60,20,20
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
(X,Y),(X1,Y1)=tf.keras.datasets.mnist.load_data()

plt.figure(figsize=(16,4))

plt.subplot(1,10,1)
plt.imshow(X[4415,:,:])
plt.subplot(1,10,2)
plt.imshow(X[3,:,:])
plt.subplot(1,10,3)
plt.imshow(X[1010,:,:])
plt.subplot(1,10,4)
plt.imshow(X[12344,:,:])
plt.subplot(1,10,5)
plt.imshow(X[412,:,:])
plt.subplot(1,10,6)
plt.imshow(X[0,:,:])
plt.subplot(1,10,7)
plt.imshow(X[1802,:,:])
plt.subplot(1,10,8)
plt.imshow(X[4545,:,:])
```

```
plt.subplot(1,10,9)
plt.imshow(X[415,:,:])
plt.subplot(1,10,10)
plt.imshow(X[344,:,:])
X=X.reshape(len(X),X.shape[1]*X.shape[2])
#Y=Y.reshape(len(Y),Y.shape[1]*Y.shape[2])
X1=X1.reshape(len(X1),X1.shape[1]*X1.shape[2])
#Y1=Y1.reshape(len(Y1),Y1.shape[1]*Y1.shape[2])
X=np.vstack((X,X1))
Y=np.hstack((Y,Y1))


x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size = 0.8)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, train_size =␣
↪0.75)
print( x_train.shape, x_val.shape, x_test.shape)
print( y_train.shape, y_val.shape, y_test.shape)
```

```
(42000, 784) (14000, 784) (14000, 784)
(42000,) (14000,) (14000,)
```



2 Normalización de los datos

Se realiza una división sobre el valor máximo de los datos

$$dato_i := \frac{dato_i}{max(datos)}$$

Se obtendran datos en el intervalo $[0, 1]$

[82]:
```
x_train_norm=(x_train/x_train.max())
x_val_norm=(x_val/x_val.max())
x_test_norm=(x_test/x_test.max())
```

Esto solo se aplicará para los valores de la variable independiente.

Para los valores de y se hará la configuración de ONE HOT

Para realizar esto, se debe observar el rango de valores en los que se tienen los valores de y. En este caso se tiene un intervalo de $[0, 9]$ lo que implican 10 posibilidades de respuesta. Entonces el *onehot* será de

[83]:
```
from tensorflow.keras.utils import to_categorical
#print(y_train.min(),y_train.max())
print(y_train[8760])
y_train_oh=to_categorical(y_train,y_train.max()+1) ## porque se incluye el 0
y_test_oh=to_categorical(y_test,y_test.max()+1)
y_val_oh=to_categorical(y_val,y_val.max()+1)
print(y_train_oh[8760])
```

```
0
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

3 Propuesta de red neuronal.

Uso de neurona perceptron multicapa

[84]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


model=Sequential()
model.add(Dense(7,input_shape=(784,),activation='sigmoid'))
model.add(Dense(7,activation='sigmoid'))
model.add(Dense(20,activation='sigmoid'))
model.add(Dense(20,activation='sigmoid'))
model.add(Dense(10,activation='softmax'))
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_25 (Dense)             (None, 7)                 5495
_________________________________________________________________
dense_26 (Dense)             (None, 7)                 56
_________________________________________________________________
dense_27 (Dense)             (None, 20)                160
_________________________________________________________________
dense_28 (Dense)             (None, 20)                420
_________________________________________________________________
dense_29 (Dense)             (None, 10)                210
=================================================================
Total params: 6,341
Trainable params: 6,341
Non-trainable params: 0
_________________________________________________________________
```

4 Caracteristicas de la red

[85]:
```
model.compile(loss='categorical_crossentropy',optimizer=tf.keras.optimizers.
↪Adam(learning_rate=0.01),metrics=['accuracy'])
```

5 Aprendizaje de la red

[86]:
```
import time


t1=time.time()
model.fit(x_train_norm, y_train_oh, validation_data = (x_val_norm, y_val_oh),␣
↪epochs = 50 , batch_size = 32, verbose = 1)
t2=time.time()
print('tiempo de entrenamiento y validación',t2-t1)
```

```
Epoch 1/50
1313/1313 [==============================] - 3s 2ms/step - loss: 0.9984 -
accuracy: 0.6353 - val_loss: 0.6494 - val_accuracy: 0.7904
Epoch 2/50
1313/1313 [==============================] - 2s 1ms/step - loss: 0.6022 -
accuracy: 0.7972 - val_loss: 0.5763 - val_accuracy: 0.8353
Epoch 3/50
1313/1313 [==============================] - 2s 1ms/step - loss: 0.5281 -
accuracy: 0.8510 - val_loss: 0.5250 - val_accuracy: 0.8535
Epoch 4/50
1313/1313 [==============================] - 1s 996us/step - loss: 0.4884 -
accuracy: 0.8683 - val_loss: 0.5229 - val_accuracy: 0.8674
Epoch 5/50
1313/1313 [==============================] - 1s 986us/step - loss: 0.4671 -
accuracy: 0.8792 - val_loss: 0.4893 - val_accuracy: 0.8769
Epoch 6/50
1313/1313 [==============================] - 1s 987us/step - loss: 0.4495 -
accuracy: 0.8836 - val_loss: 0.4674 - val_accuracy: 0.8796
Epoch 7/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.4268 -
accuracy: 0.8908 - val_loss: 0.4497 - val_accuracy: 0.8865
Epoch 8/50
1313/1313 [==============================] - 1s 982us/step - loss: 0.4197 -
accuracy: 0.8920 - val_loss: 0.4610 - val_accuracy: 0.8775
Epoch 9/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.4150 -
accuracy: 0.8931 - val_loss: 0.4480 - val_accuracy: 0.8876
Epoch 10/50
1313/1313 [==============================] - 1s 984us/step - loss: 0.4106 -
accuracy: 0.8929 - val_loss: 0.4569 - val_accuracy: 0.8842
Epoch 11/50
```

```
1313/1313 [==============================] - 1s 986us/step - loss: 0.4062 -
accuracy: 0.8964 - val_loss: 0.4485 - val_accuracy: 0.8878
Epoch 12/50
1313/1313 [==============================] - 1s 976us/step - loss: 0.3988 -
accuracy: 0.8969 - val_loss: 0.4734 - val_accuracy: 0.8827
Epoch 13/50
1313/1313 [==============================] - 1s 962us/step - loss: 0.3951 -
accuracy: 0.8989 - val_loss: 0.4634 - val_accuracy: 0.8854
Epoch 14/50
1313/1313 [==============================] - 1s 979us/step - loss: 0.3930 -
accuracy: 0.8990 - val_loss: 0.4875 - val_accuracy: 0.8786
Epoch 15/50
1313/1313 [==============================] - 1s 970us/step - loss: 0.3882 -
accuracy: 0.8999 - val_loss: 0.4559 - val_accuracy: 0.8847
Epoch 16/50
1313/1313 [==============================] - 1s 967us/step - loss: 0.3835 -
accuracy: 0.9011 - val_loss: 0.4532 - val_accuracy: 0.8879
Epoch 17/50
1313/1313 [==============================] - 1s 974us/step - loss: 0.3815 -
accuracy: 0.9021 - val_loss: 0.4551 - val_accuracy: 0.8849
Epoch 18/50
1313/1313 [==============================] - 1s 970us/step - loss: 0.3800 -
accuracy: 0.9025 - val_loss: 0.4695 - val_accuracy: 0.8847
Epoch 19/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3739 -
accuracy: 0.9051 - val_loss: 0.4658 - val_accuracy: 0.8839
Epoch 20/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3723 -
accuracy: 0.9039 - val_loss: 0.4428 - val_accuracy: 0.8916
Epoch 21/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3681 -
accuracy: 0.9051 - val_loss: 0.4370 - val_accuracy: 0.8889
Epoch 22/50
1313/1313 [==============================] - 1s 978us/step - loss: 0.3705 -
accuracy: 0.9056 - val_loss: 0.4471 - val_accuracy: 0.8869
Epoch 23/50
1313/1313 [==============================] - 1s 977us/step - loss: 0.3675 -
accuracy: 0.9061 - val_loss: 0.4476 - val_accuracy: 0.8849
Epoch 24/50
1313/1313 [==============================] - 1s 979us/step - loss: 0.3676 -
accuracy: 0.9060 - val_loss: 0.4687 - val_accuracy: 0.8840
Epoch 25/50
1313/1313 [==============================] - 1s 976us/step - loss: 0.3698 -
accuracy: 0.9047 - val_loss: 0.4560 - val_accuracy: 0.8859
Epoch 26/50
1313/1313 [==============================] - 1s 974us/step - loss: 0.3613 -
accuracy: 0.9077 - val_loss: 0.4506 - val_accuracy: 0.8866
Epoch 27/50
```

```
1313/1313 [==============================] - 1s 969us/step - loss: 0.3634 -
accuracy: 0.9084 - val_loss: 0.4450 - val_accuracy: 0.8871
Epoch 28/50
1313/1313 [==============================] - 1s 971us/step - loss: 0.3572 -
accuracy: 0.9086 - val_loss: 0.4601 - val_accuracy: 0.8834
Epoch 29/50
1313/1313 [==============================] - 1s 973us/step - loss: 0.3577 -
accuracy: 0.9073 - val_loss: 0.4521 - val_accuracy: 0.8861
Epoch 30/50
1313/1313 [==============================] - 1s 975us/step - loss: 0.3534 -
accuracy: 0.9099 - val_loss: 0.4868 - val_accuracy: 0.8781
Epoch 31/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3537 -
accuracy: 0.9089 - val_loss: 0.4670 - val_accuracy: 0.8846
Epoch 32/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3514 -
accuracy: 0.9101 - val_loss: 0.4484 - val_accuracy: 0.8901
Epoch 33/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3535 -
accuracy: 0.9092 - val_loss: 0.4746 - val_accuracy: 0.8826
Epoch 34/50
1313/1313 [==============================] - 1s 971us/step - loss: 0.3515 -
accuracy: 0.9095 - val_loss: 0.4546 - val_accuracy: 0.8838
Epoch 35/50
1313/1313 [==============================] - 1s 972us/step - loss: 0.3521 -
accuracy: 0.9100 - val_loss: 0.4422 - val_accuracy: 0.8893
Epoch 36/50
1313/1313 [==============================] - 1s 976us/step - loss: 0.3482 -
accuracy: 0.9105 - val_loss: 0.4540 - val_accuracy: 0.8881
Epoch 37/50
1313/1313 [==============================] - 1s 969us/step - loss: 0.3463 -
accuracy: 0.9115 - val_loss: 0.4312 - val_accuracy: 0.8906
Epoch 38/50
1313/1313 [==============================] - 1s 970us/step - loss: 0.3455 -
accuracy: 0.9105 - val_loss: 0.4606 - val_accuracy: 0.8840
Epoch 39/50
1313/1313 [==============================] - 1s 967us/step - loss: 0.3455 -
accuracy: 0.9116 - val_loss: 0.4447 - val_accuracy: 0.8862
Epoch 40/50
1313/1313 [==============================] - 1s 972us/step - loss: 0.3454 -
accuracy: 0.9105 - val_loss: 0.4532 - val_accuracy: 0.8850
Epoch 41/50
1313/1313 [==============================] - 1s 972us/step - loss: 0.3479 -
accuracy: 0.9102 - val_loss: 0.4570 - val_accuracy: 0.8849
Epoch 42/50
1313/1313 [==============================] - 1s 972us/step - loss: 0.3477 -
accuracy: 0.9110 - val_loss: 0.4514 - val_accuracy: 0.8874
Epoch 43/50
```

```
1313/1313 [==============================] - 1s 988us/step - loss: 0.3399 -
accuracy: 0.9138 - val_loss: 0.4670 - val_accuracy: 0.8859
Epoch 44/50
1313/1313 [==============================] - 1s 1ms/step - loss: 0.3413 -
accuracy: 0.9122 - val_loss: 0.4651 - val_accuracy: 0.8846
Epoch 45/50
1313/1313 [==============================] - 1s 977us/step - loss: 0.3416 -
accuracy: 0.9129 - val_loss: 0.4479 - val_accuracy: 0.8887
Epoch 46/50
1313/1313 [==============================] - 1s 995us/step - loss: 0.3373 -
accuracy: 0.9130 - val_loss: 0.5011 - val_accuracy: 0.8716
Epoch 47/50
1313/1313 [==============================] - 1s 974us/step - loss: 0.3409 -
accuracy: 0.9120 - val_loss: 0.4549 - val_accuracy: 0.8889
Epoch 48/50
1313/1313 [==============================] - 1s 972us/step - loss: 0.3365 -
accuracy: 0.9141 - val_loss: 0.4556 - val_accuracy: 0.8889
Epoch 49/50
1313/1313 [==============================] - 1s 974us/step - loss: 0.3389 -
accuracy: 0.9122 - val_loss: 0.4592 - val_accuracy: 0.8861
Epoch 50/50
1313/1313 [==============================] - 1s 982us/step - loss: 0.3380 -
accuracy: 0.9128 - val_loss: 0.4526 - val_accuracy: 0.8876
tiempo de entrenamiento y validación 67.06097769737244
```

[87]:
```
pred=model.predict(x_test_norm)
#print(pred)
pred = np.argmax(pred, axis = 1)
#print(pred)
#print(pred.shape)
label = np.argmax(y_test_oh,axis = 1)

exactitud_test=0
for a in range(len(pred)):
    if pred[a]==y_test[a]:
        exactitud_test+=1
print('exactitud de la prueba= ',100*exactitud_test/len(pred),'%')


pred_train=model.predict(x_train_norm)
#print(pred_train)
pred_train = np.argmax(pred_train, axis = 1)
#print(pred_train)
#print(pred_train.shape)
label_train = np.argmax(y_train_oh,axis = 1)
exactitud_train=0
for a in range(len(pred_train)):
```

```
    if pred_train[a]==y_train[a]:
        exactitud_train+=1
print('exactitud del entrenamiento= ',100*exactitud_train/len(pred_train),'%')


pred_val=model.predict(x_val_norm)
#print(pred_train)
pred_val = np.argmax(pred_val, axis = 1)
#print(pred_val)
#print(pred_val.shape)
label_val = np.argmax(y_val_oh,axis = 1)
exactitud_val=0
for a in range(len(pred_val)):
    if pred_val[a]==y_val[a]:
        exactitud_val+=1
print('exactitud de la validación= ',100*exactitud_val/len(pred_val),'%')
```

```
exactitud de la prueba=  89.12857142857143 %
exactitud del entrenamiento=  91.88095238095238 %
exactitud de la validación=  88.75714285714285 %
```

Caracteristicas • Estructura • Optimizador • Tasa de aprendizaje • Exactitud final entrenamiento • Exactitud final validación • Exactitud final test • Épocas • Batch • Tiempo de entrenamiento

¿Con el batch se debe de dividir el 60 20 20?