Optimización

La optimización es el proceso que hace que la red neuronal se vuelva cada vez mejor en la tarea, o sea, que aprenda. El objetivo de las redes neuronales es simple, minimizar la **función de pérdida**.

Función de pérdida

La función de pérdida nos indica qué tan bien se hizo la tarea, nos dice el error cometido por la red. La función de pérdida más común para las redes neuronales es la **categorical cross-entropy** que para un ejemplo x_i se define como:

$$L_i(y_i, \hat{y}_i) = -\sum_{j=1}^{C} y_i^{(j)} \log(\hat{y}_i^{(j)})$$

donde C es el número de clases, $\hat{y}_i$ es la predicción del modelo para el ejemplo x_i y y_i es la etiqueta asiganada a x_i. La pérdida total es el promedio sobre todos los elementos del conjunto de datos.

$$L = \frac{1}{m} \sum_{i=1}^{m} L_i(\hat{y}_i, y_i)$$

donde m es el número de ejemplos.

```
from tensorflow import keras
import numpy as np

y_true = np.array([[0, 1, 0], [1, 0, 0], [0, 0, 1], [0, 0, 1]])
y_pred = np.array([[0.1, 0.8, 0.1], [0.7, 0.2, 0.1], [.8, .1, .1], [0.01, 0.01, 0.98]])

loss = keras.losses.categorical_crossentropy(y_true, y_pred).numpy()
print(f"Pérdidas individuales: {loss},\nPromedio: {np.mean(loss)}")
```

```
Pérdidas individuales: [0.22314355 0.35667494 2.30258509 0.02020271],
Promedio: 0.7256515738911267
```

Entonces, el objetivo es obtener las predicciones que hagan que L sea lo más pequeña posible. Las predicciones son una función del ejemplo x_i y de los pesos del modelo W. Esto es

$$\hat{y}_i = f(x_i; W)$$

Para minimizar L, se usa la técnica de optimización **Descenso de gradiente** que consiste en sacar el gradiente de L y modificar los pesos en la dirección opuesta. El siguiente código representa a la variante **Descenso de gradiente estocástico** (SGD) donde se parte el conjunto en lotes de datos y actualiza los pesos una vez por cada lote.

```
for batch in data:
    g = grad(L, batch, W)
    W += -lr * g
```

En este código aparece el parámetro `lr` que se llama **learning rate** que nos dice el tamaño del vector con que actualizaremos los pesos. El learning rate es un parámetro importante que debemos elegir antes de entrenar los modelos.

- Un learning rate muy pequeño hará que el entrenamiento tarde en converger.
- Uno muy grande puede hacer que la pérdida nunca disminuya de manera consistente.
- Un learning rate propiado disminuirá la pérdida de manera constante.

Veamos un ejemplo de entrenamiento con varios learning rates.

```
from tensorflow import keras
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical

from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import activations
from tensorflow.keras import optimizers

import numpy as np
from sklearn.metrics import classification_report
```

```
(train, train_labels), (test, test_labels) = mnist.load_data()
train, val, train_labels, val_labels = train_test_split(train,
                                                        train_labels,
                                                        test_size=0.1,
                                                        random_state=42)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434             0s 0us/step
```

```
train = train.reshape((54000, 28 * 28))
train = train.astype('float32') / 255
val = val.reshape((6000, 28 * 28))
val = val.astype('float32') / 255
test = test.reshape((10000, 28 * 28))
test = test.astype('float32') / 255
```

```
train_labels = to_categorical(train_labels)
val_labels = to_categorical(val_labels)
```

```
def create_network():
    network = models.Sequential()
    network.add(layers.Dense(128, activation='sigmoid', input_shape=(28 * 28,)))
    network.add(layers.Dense(128, activation='sigmoid'))
    network.add(layers.Dense(10, activation='softmax'))
    return network
```

```
def plot_results(complete_history, legend):
    for opt in complete_history:
        plt.plot(complete_history[opt].history['val_accuracy'])
        plt.ylabel('accuracy')
        plt.xlabel('epoch')
        plt.legend(legend, loc='upper left')
    plt.show()
    for opt in complete_history:
        plt.plot(complete_history[opt].history['val_loss'])
        plt.yscale('log')
        plt.ylabel('loss')
        plt.xlabel('epoch')
        plt.legend(legend, loc='upper left')
    plt.show()
```

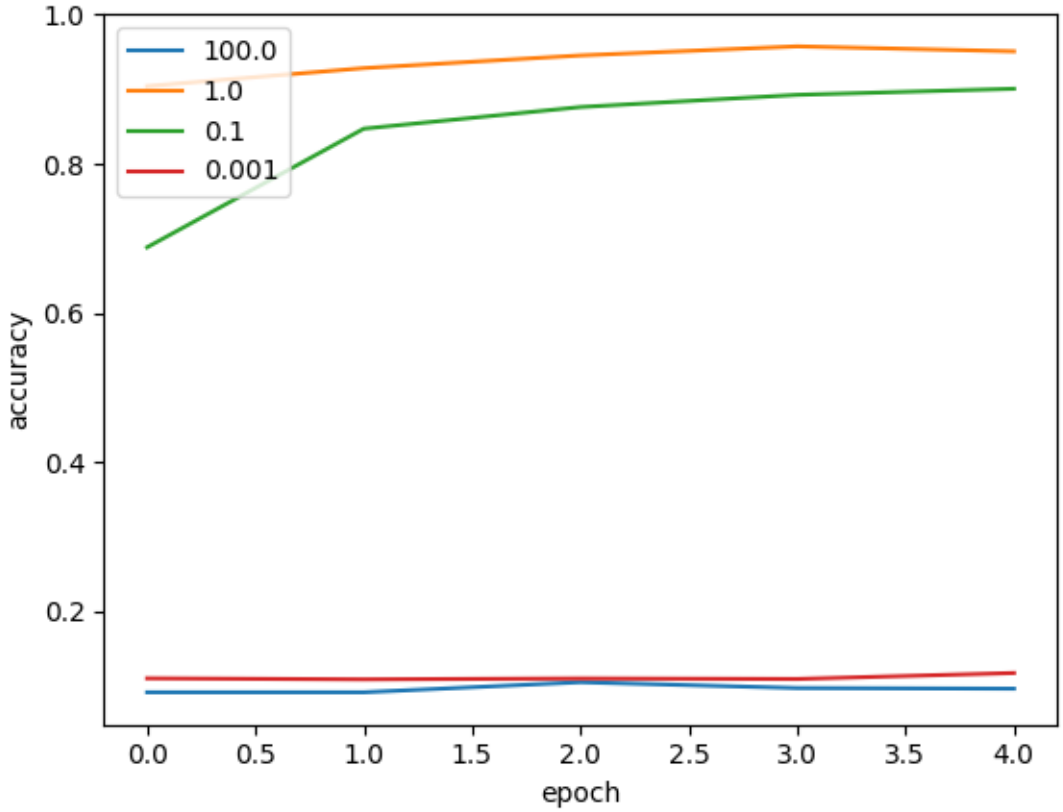
Variar el Learning Rate

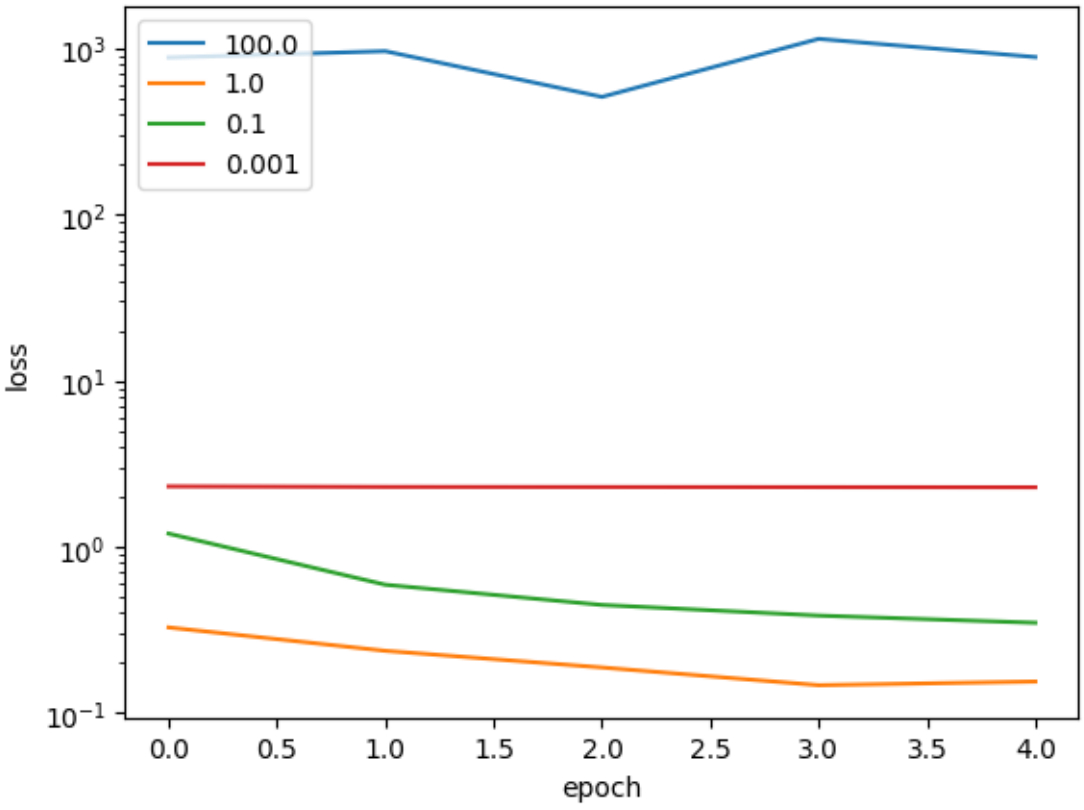
```
lrs = [100., 1., .1, .001]
complete_history = {}
for lr in lrs:
    print("Entrenando con learning rate:", lr)

    network = create_network()
    network.compile(optimizer=optimizers.SGD(learning_rate=lr),
                loss='categorical_crossentropy',
                metrics=['accuracy'])
    history = network.fit(train,
                          train_labels,
                          validation_data=(val, val_labels),
                          epochs=5,
                          batch_size=128)
    complete_history[lr] = history
```

```
Entrenando con learning rate: 100.0
Epoch 1/5
422/422            3s 6ms/step - accuracy: 0.1022 - loss: 880.9357 - val_accuracy: 0.0918 -
Epoch 2/5
422/422            2s 5ms/step - accuracy: 0.0988 - loss: 814.0910 - val_accuracy: 0.0918 -
Epoch 3/5
422/422            2s 5ms/step - accuracy: 0.0977 - loss: 818.9393 - val_accuracy: 0.1053 -
Epoch 4/5
422/422            4s 10ms/step - accuracy: 0.1001 - loss: 810.5908 - val_accuracy: 0.0975 -
Epoch 5/5
422/422            4s 8ms/step - accuracy: 0.0996 - loss: 794.7347 - val_accuracy: 0.0967 -
Entrenando con learning rate: 1.0
Epoch 1/5
422/422            3s 7ms/step - accuracy: 0.5664 - loss: 1.3595 - val_accuracy: 0.9043 - va
Epoch 2/5
422/422            3s 7ms/step - accuracy: 0.9108 - loss: 0.2992 - val_accuracy: 0.9285 - va
Epoch 3/5
422/422            2s 6ms/step - accuracy: 0.9342 - loss: 0.2184 - val_accuracy: 0.9455 - va
Epoch 4/5
422/422            2s 6ms/step - accuracy: 0.9510 - loss: 0.1663 - val_accuracy: 0.9577 - va
Epoch 5/5
422/422            3s 6ms/step - accuracy: 0.9583 - loss: 0.1418 - val_accuracy: 0.9510 - va
Entrenando con learning rate: 0.1
```

```
Epoch 1/5
422/422            4s 9ms/step - accuracy: 0.2813 - loss: 2.1669 - val_accuracy: 0.6883 - va
Epoch 2/5
422/422            4s 6ms/step - accuracy: 0.7668 - loss: 0.9670 - val_accuracy: 0.8473 - va
Epoch 3/5
422/422            2s 6ms/step - accuracy: 0.8573 - loss: 0.5470 - val_accuracy: 0.8763 - va
Epoch 4/5
422/422            3s 6ms/step - accuracy: 0.8815 - loss: 0.4329 - val_accuracy: 0.8927 - va
Epoch 5/5
422/422            3s 7ms/step - accuracy: 0.8943 - loss: 0.3773 - val_accuracy: 0.9008 - va
Entrenando con learning rate: 0.001
Epoch 1/5
422/422            3s 6ms/step - accuracy: 0.1067 - loss: 2.4684 - val_accuracy: 0.1105 - va
Epoch 2/5
422/422            2s 5ms/step - accuracy: 0.1142 - loss: 2.3019 - val_accuracy: 0.1092 - va
Epoch 3/5
422/422            2s 6ms/step - accuracy: 0.1145 - loss: 2.2918 - val_accuracy: 0.1103 - va
Epoch 4/5
422/422            3s 6ms/step - accuracy: 0.1159 - loss: 2.2880 - val_accuracy: 0.1097 - va
Epoch 5/5
422/422            3s 8ms/step - accuracy: 0.1148 - loss: 2.2840 - val_accuracy: 0.1177 - va
```

```
plot_results(complete_history, legend=lrs)
```


100.0
1.0
0.1
0.001
accuracy
epoch
1.0
0.8
0.6
0.4
0.2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



Diferentes Optimizadores

Además del SGD, existen variantes donde se toma en cuenta el **momentum** del descenso para encontrar un mejor mínimo.

```
lr = 0.001

sgd = [optimizers.SGD(learning_rate=lr), "SGD"]
momentum = [optimizers.SGD(learning_rate=lr, momentum=0.9), "Momentum"]
rmsprop = [optimizers.RMSprop(learning_rate=lr), "RMSProp"]
adam = [optimizers.Adam(learning_rate=lr), "Adam"]
OPTIMIZERS = [sgd, momentum, rmsprop, adam]

complete_history = {}
predictions = []
for opt in OPTIMIZERS:
    print("Training:", opt[1])
```

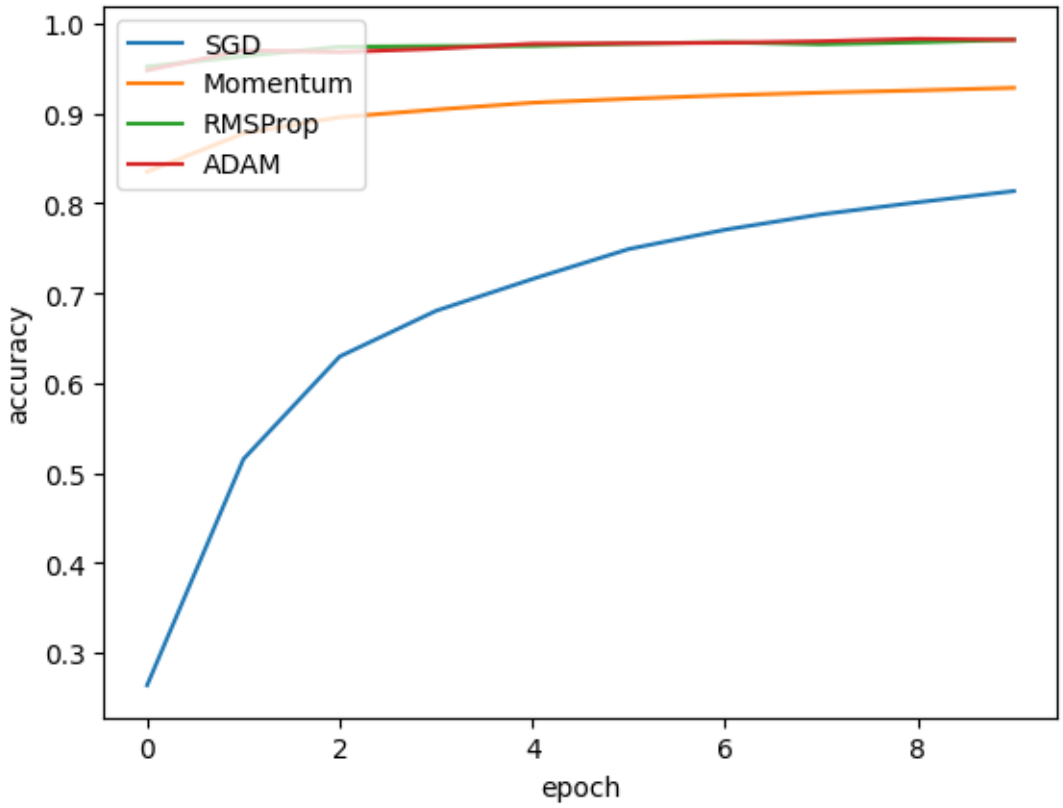
```
    network = create_network()
    network.compile(optimizer=opt[0],
                loss='categorical_crossentropy',
                metrics=['accuracy'])
    history = network.fit(train,
                          train_labels,
                          validation_data=(val, val_labels),
                          epochs=10,
                          batch_size=128)
    complete_history[opt[1]] = history
    pred = network.predict(test)
    predictions.append(pred)
```

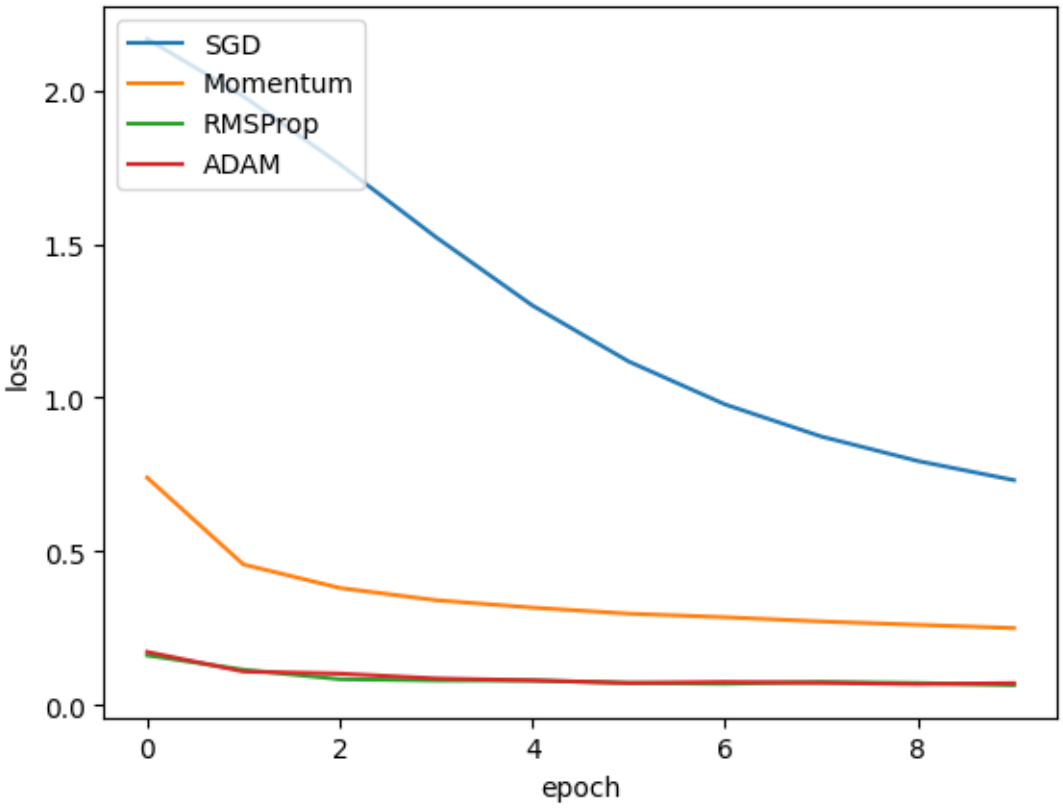
```
Training: SGD

/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do no
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/10
422/422            3s 6ms/step - accuracy: 0.1030 - loss: 2.3043 - val_accuracy: 0.2640 - va
Epoch 2/10
422/422            6s 8ms/step - accuracy: 0.3406 - loss: 2.1194 - val_accuracy: 0.5155 - va
Epoch 3/10
422/422            4s 5ms/step - accuracy: 0.5528 - loss: 1.9224 - val_accuracy: 0.6297 - va
Epoch 4/10
422/422            2s 6ms/step - accuracy: 0.6511 - loss: 1.6934 - val_accuracy: 0.6805 - va
Epoch 5/10
422/422            3s 7ms/step - accuracy: 0.6982 - loss: 1.4573 - val_accuracy: 0.7158 - va
Epoch 6/10
422/422            5s 6ms/step - accuracy: 0.7378 - loss: 1.2407 - val_accuracy: 0.7492 - va
Epoch 7/10
422/422            5s 6ms/step - accuracy: 0.7587 - loss: 1.0760 - val_accuracy: 0.7707 - va
Epoch 8/10
422/422            3s 6ms/step - accuracy: 0.7817 - loss: 0.9429 - val_accuracy: 0.7878 - va
Epoch 9/10
422/422            4s 10ms/step - accuracy: 0.7983 - loss: 0.8457 - val_accuracy: 0.8012 - v
Epoch 10/10
422/422            3s 7ms/step - accuracy: 0.8118 - loss: 0.7712 - val_accuracy: 0.8137 - va
313/313            1s 3ms/step
Training: Momentum
```

```
Epoch 1/10
422/422            3s 6ms/step - accuracy: 0.4416 - loss: 1.8910 - val_accuracy: 0.8352 - va
Epoch 2/10
422/422            5s 7ms/step - accuracy: 0.8498 - loss: 0.6355 - val_accuracy: 0.8778 - va
Epoch 3/10
422/422            5s 7ms/step - accuracy: 0.8814 - loss: 0.4415 - val_accuracy: 0.8958 - va
Epoch 4/10
422/422            3s 6ms/step - accuracy: 0.8962 - loss: 0.3787 - val_accuracy: 0.9043 - va
Epoch 5/10
422/422            4s 9ms/step - accuracy: 0.9043 - loss: 0.3389 - val_accuracy: 0.9120 - va
Epoch 6/10
422/422            3s 6ms/step - accuracy: 0.9121 - loss: 0.3117 - val_accuracy: 0.9163 - va
Epoch 7/10
422/422            5s 7ms/step - accuracy: 0.9161 - loss: 0.2987 - val_accuracy: 0.9202 - va
Epoch 8/10
422/422            5s 7ms/step - accuracy: 0.9168 - loss: 0.2896 - val_accuracy: 0.9232 - va
Epoch 9/10
422/422            5s 6ms/step - accuracy: 0.9236 - loss: 0.2726 - val_accuracy: 0.9257 - va
Epoch 10/10
422/422            3s 6ms/step - accuracy: 0.9254 - loss: 0.2632 - val_accuracy: 0.9285 - va
313/313            1s 2ms/step
Training: RMSProp
Epoch 1/10
422/422            5s 8ms/step - accuracy: 0.8418 - loss: 0.5590 - val_accuracy: 0.9522 - va
Epoch 2/10
422/422            4s 6ms/step - accuracy: 0.9565 - loss: 0.1448 - val_accuracy: 0.9637 - va
Epoch 3/10
422/422            3s 6ms/step - accuracy: 0.9725 - loss: 0.0923 - val_accuracy: 0.9742 - va
Epoch 4/10
422/422            3s 8ms/step - accuracy: 0.9789 - loss: 0.0702 - val_accuracy: 0.9748 - va
Epoch 5/10
422/422            3s 6ms/step - accuracy: 0.9830 - loss: 0.0554 - val_accuracy: 0.9752 - va
Epoch 6/10
422/422            5s 6ms/step - accuracy: 0.9871 - loss: 0.0427 - val_accuracy: 0.9777 - va
Epoch 7/10
422/422            3s 6ms/step - accuracy: 0.9895 - loss: 0.0352 - val_accuracy: 0.9792 - va
Epoch 8/10
422/422            4s 9ms/step - accuracy: 0.9918 - loss: 0.0274 - val_accuracy: 0.9770 - va
Epoch 9/10
422/422            4s 6ms/step - accuracy: 0.9937 - loss: 0.0219 - val_accuracy: 0.9792 - va
Epoch 10/10
422/422            5s 7ms/step - accuracy: 0.9938 - loss: 0.0198 - val_accuracy: 0.9820 - va
313/313            1s 2ms/step
```

```
Training: Adam
Epoch 1/10
422/422            4s 7ms/step - accuracy: 0.8171 - loss: 0.6298 - val_accuracy: 0.9480 - va
Epoch 2/10
422/422            6s 8ms/step - accuracy: 0.9535 - loss: 0.1555 - val_accuracy: 0.9703 - va
Epoch 3/10
422/422            4s 6ms/step - accuracy: 0.9706 - loss: 0.0978 - val_accuracy: 0.9683 - va
Epoch 4/10
422/422            3s 7ms/step - accuracy: 0.9765 - loss: 0.0752 - val_accuracy: 0.9720 - va
Epoch 5/10
422/422            6s 9ms/step - accuracy: 0.9826 - loss: 0.0587 - val_accuracy: 0.9775 - va
Epoch 6/10
422/422            4s 7ms/step - accuracy: 0.9878 - loss: 0.0431 - val_accuracy: 0.9780 - va
Epoch 7/10
422/422            5s 6ms/step - accuracy: 0.9895 - loss: 0.0354 - val_accuracy: 0.9787 - va
Epoch 8/10
422/422            4s 9ms/step - accuracy: 0.9909 - loss: 0.0291 - val_accuracy: 0.9803 - va
Epoch 9/10
422/422            5s 8ms/step - accuracy: 0.9941 - loss: 0.0213 - val_accuracy: 0.9830 - va
Epoch 10/10
422/422            3s 6ms/step - accuracy: 0.9941 - loss: 0.0204 - val_accuracy: 0.9820 - va
313/313            1s 2ms/step
```

```
plot_results(complete_history, legend=['SGD', 'Momentum', 'RMSProp', 'ADAM'])
```


SGD
Momentum
RMSProp
ADAM
accuracy
epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
2
4
6
8




```
for p, opt in zip(predictions, OPTIMIZERS):
    print("=" * 50)
    print("Resultados de", opt[1])
    print(classification_report(test_labels, np.argmax(p, axis=1), digits=4))
```

```
==================================================
Resultados de SGD
              precision    recall  f1-score   support

           0     0.8829    0.9388    0.9100       980
           1     0.8877    0.9753    0.9295      1135
           2     0.8615    0.8014    0.8303      1032
           3     0.7987    0.8446    0.8210      1010
           4     0.7339    0.7413    0.7376       982
           5     0.8265    0.6570    0.7320       892
           6     0.8772    0.9019    0.8893       958
           7     0.8675    0.8599    0.8637      1028
           8     0.7981    0.7752    0.7865       974
```

```
           9     0.6986    0.7146    0.7065      1009

    accuracy                         0.8245     10000
   macro avg     0.8233    0.8210    0.8206     10000
weighted avg     0.8241    0.8245    0.8229     10000

==================================================
Resultados de Momentum
              precision    recall  f1-score   support

           0     0.9368    0.9827    0.9592       980
           1     0.9737    0.9797    0.9767      1135
           2     0.9345    0.9128    0.9235      1032
           3     0.9174    0.9238    0.9206      1010
           4     0.9178    0.9440    0.9307       982
           5     0.9255    0.8778    0.9010       892
           6     0.9297    0.9530    0.9412       958
           7     0.9545    0.9173    0.9355      1028
           8     0.9117    0.9014    0.9066       974
           9     0.9094    0.9158    0.9126      1009

    accuracy                         0.9318     10000
   macro avg     0.9311    0.9308    0.9308     10000
weighted avg     0.9319    0.9318    0.9316     10000

==================================================
Resultados de RMSProp
              precision    recall  f1-score   support

           0     0.9898    0.9888    0.9893       980
           1     0.9929    0.9921    0.9925      1135
           2     0.9834    0.9777    0.9806      1032
           3     0.9889    0.9723    0.9805      1010
           4     0.9689    0.9837    0.9763       982
           5     0.9820    0.9776    0.9798       892
           6     0.9843    0.9802    0.9822       958
           7     0.9657    0.9864    0.9759      1028
           8     0.9713    0.9743    0.9728       974
           9     0.9750    0.9683    0.9717      1009

    accuracy                         0.9803     10000
   macro avg     0.9802    0.9801    0.9802     10000
weighted avg     0.9804    0.9803    0.9803     10000
```

```
==================================================
Resultados de Adam
              precision    recall  f1-score   support

           0     0.9907    0.9827    0.9867       980
           1     0.9929    0.9868    0.9898      1135
           2     0.9613    0.9864    0.9737      1032
           3     0.9839    0.9653    0.9745      1010
           4     0.9825    0.9745    0.9785       982
           5     0.9659    0.9832    0.9744       892
           6     0.9822    0.9812    0.9817       958
           7     0.9880    0.9601    0.9739      1028
           8     0.9607    0.9784    0.9695       974
           9     0.9657    0.9762    0.9709      1009

    accuracy                         0.9775     10000
   macro avg     0.9774    0.9775    0.9774     10000
weighted avg     0.9777    0.9775    0.9775     10000
```