Tarea N°3:

Redes Neuronales Convolucionales

Rodrigo Aníbal Llancao Zúñiga
Escuela de Ingeniería, Universidad de O'Higgins
11, Noviembre, 2023

***Abstract*—En esta tarea, se aborda la implementación de una Red Neuronal Convolucional (CNN) con el propósito de clasificar etiquetas correspondientes en dos conjuntos de datos distintos: la base de datos de manuscritos MNIST y la base de datos de imágenes pequeñas CIFAR-10. La implementación se lleva a cabo en el lenguaje de programación Python, utilizando la biblioteca TensorFlow.**

Las Convolutional Neural Networks (CNN o ConvNet) son una categoría de redes neuronales profundas ampliamente utilizadas en el análisis de imágenes visuales. Conocidas también como redes neuronales artificiales invariantes al desplazamiento o al espacio, las CNN se basan en la arquitectura de pesos compartidos de los filtros de convolución. Estos filtros se deslizan a lo largo de las características de entrada, proporcionando respuestas equivariantes a la translación, denominadas feature maps. Las CNN encuentran aplicaciones en diversos campos, como el reconocimiento de imágenes y videos, sistemas de recomendación, clasificación de imágenes, segmentación de imágenes, análisis de imágenes médicas, procesamiento del lenguaje natural, interfaces cerebro-ordenador y análisis de series temporales financieras, entre otros. Este trabajo se enfoca en la aplicación de estas redes para la tarea específica de clasificación de datos provenientes de las bases MNIST y CIFAR-10.

I. Introducción

En el campo del aprendizaje profundo, las Redes Neuronales Convolucionales (CNN) han emergido como una herramienta fundamental, especialmente en el ámbito del procesamiento de imágenes. Estas redes, diseñadas para imitar el procesamiento visual que ocurre en el cerebro humano, han demostrado ser altamente eficaces en una variedad de tareas relacionadas con el análisis de imágenes.

El objetivo principal de esta tarea es implementar y evaluar una CNN para la clasificación de datos provenientes de dos conjuntos de imágenes ampliamente utilizados en la comunidad de aprendizaje automático: la base de datos de manuscritos MNIST y la base de datos de imágenes pequeñas CIFAR-10. Este proceso se lleva a cabo utilizando el lenguaje de programación Python y la biblioteca TensorFlow.

En la era actual de la inteligencia artificial, la capacidad de clasificar imágenes de manera precisa es esencial para numerosas aplicaciones, desde el reconocimiento de dígitos manuscritos hasta la identificación de objetos en fotografías de la vida cotidiana. Las CNN, al incorporar capas de convolución que pueden aprender características jerárquicas, han demostrado ser excepcionalmente hábiles en la extracción y comprensión de patrones visuales complejos.

Esta tarea no solo busca implementar una solución funcional, sino también explorar las decisiones de diseño clave, como la arquitectura de la red, el tipo de capas utilizadas y los parámetros de entrenamiento. A través de este proceso, se pretende no solo obtener un modelo que logre clasificaciones precisas, sino también comprender cómo diferentes elecciones de diseño afectan el rendimiento del modelo. Este trabajo se suma al continuo esfuerzo de la comunidad científica por mejorar las capacidades de las redes neuronales en la clasificación de imágenes, contribuyendo así al desarrollo y comprensión de las aplicaciones de aprendizaje profundo en el procesamiento visual.

II. Marco Teórico

A. Redes Neuronales Convolucionales (CNN)

Las Redes Neuronales Convolucionales (CNN) son un tipo especializado de red neuronal diseñadas para procesar datos estructurados en cuadrículas, como imágenes. Se componen de capas de convolución que aplican filtros para extraer características locales, seguidas de capas de agrupación para reducir la dimensionalidad. Esta arquitectura ha demostrado ser altamente efectiva en tareas de visión por computadora.

B. Base de Datos MNIST

La base de datos MNIST es un conjunto de 70,000 imágenes de dígitos escritos a mano (0 al 9), ampliamente utilizada para entrenar y evaluar algoritmos de reconocimiento de dígitos. Cada imagen es de 28x28 píxeles, lo que la convierte en una tarea de clasificación de imágenes relativamente pequeñas pero desafiante.

C. Base de Datos CIFAR-10

La base de datos CIFAR-10 contiene 60,000 imágenes en color de 32x32 píxeles, agrupadas en 10 clases, cada una representando un tipo de objeto. Esta base de datos es un estándar en la evaluación de algoritmos de clasificación de imágenes y presenta un desafío adicional debido a la presencia de colores y detalles finos.

D. TensorFlow

TensorFlow es una biblioteca de código abierto desarrollada por Google, diseñada para facilitar la implementación y entrenamiento de modelos de aprendizaje automático. Proporciona herramientas flexibles para construir redes neuronales y otros modelos de aprendizaje profundo.

E. One-Hot Encoding

El One-Hot Encoding es una técnica de representación de variables categóricas en forma de vectores binarios. Se utiliza comúnmente en aprendizaje automático y procesamiento de lenguaje natural para manejar datos categóricos en algoritmos que requieren entradas numéricas.

La idea básica detrás del One-Hot Encoding es asignar un valor binario único a cada categoría posible. Cada categoría se representa como un vector de longitud igual al número total de categorías, y todos los elementos del vector son establecidos en cero, excepto el correspondiente a la categoría, que se establece en uno. En otras palabras, solo un bit está "caliente" (one) mientras que los demás están "fríos" (zero), de ahí el nombre "One-Hot".

F. Max-Pooling

Max-pooling es una operación comúnmente utilizada en redes neuronales convolucionales (CNN) para reducir la dimensionalidad de cada mapa de características mientras se conservan las características más relevantes. Es particularmente útil en tareas de visión por computadora y reconocimiento de patrones.

La operación de max-pooling se realiza en cada mapa de características por separado. Dado un tamaño de ventana (kernel), la operación de max-pooling divide el mapa de características en regiones no solapadas y selecciona el valor máximo de cada región. El resultado es un nuevo mapa de características con una resolución espacial reducida.

La ventaja principal de max-pooling radica en su capacidad para conservar las características más importantes de una región mientras reduce el tamaño de la representación. Esto proporciona invarianza a pequeñas traslaciones en la entrada y reduce la cantidad de parámetros y cómputo en la red, ayudando a prevenir el sobreajuste (overfitting).

G. Adadelta

Adadelta es un algoritmo de optimización que se adapta dinámicamente la tasa de aprendizaje durante el entrenamiento. Fue diseñado para abordar algunas limitaciones del algoritmo Adagrad, especialmente en la reducción de la tasa de aprendizaje de forma agresiva a medida que avanza el entrenamiento. Adadelta ajusta automáticamente la escala de aprendizaje para cada parámetro en función de la magnitud acumulativa de los gradientes recientes. Esto ayuda a mitigar el problema de ajustar manualmente la tasa de aprendizaje y mejora la estabilidad del entrenamiento.

H. Adagrad

Adagrad es un algoritmo de optimización que adapta la tasa de aprendizaje para cada parámetro en función de la frecuencia de actualización de ese parámetro. En otras palabras, asigna tasas de aprendizaje más grandes a parámetros que reciben actualizaciones infrecuentes y tasas más pequeñas a parámetros que se actualizan con frecuencia. Si bien Adagrad puede ser efectivo en problemas convexos, puede sufrir de tasas de aprendizaje decrecientes rápidamente en problemas no convexas, lo que lleva a un estancamiento prematuro.

I. Adam

Adam, que significa "Adaptive Moment Estimation", es un algoritmo de optimización que combina elementos de otros dos algoritmos populares, RMSprop y Momentum. Adam mantiene dos promedios móviles: uno para los gradientes (similar a Momentum) y otro para el cuadrado de los gradientes (similar a RMSprop). Estos promedios se utilizan para calcular la corrección del sesgo y adaptar la tasa de aprendizaje para cada parámetro. Adam ha demostrado ser efectivo en una variedad de tareas y es conocido por su capacidad para manejar problemas con gradientes dispersos o ruidosos.

III. Metodología

En esta sección, se presenta la metodología empleada para implementar y evaluar las Redes Neuronales Convolucionales (CNN) destinadas a la clasificación de imágenes de la base de datos MNIST y CIFAR-10. El enfoque metodológico sigue las pautas proporcionadas para la tarea y se fundamenta en las prácticas recomendadas en el ámbito de las redes neuronales convolucionales y el aprendizaje profundo.

A. Configuración del Entorno de Desarrollo

Se utilizó un entorno de desarrollo en Python, aprovechando la biblioteca TensorFlow para la implementación de las CNN. Se aseguró la disponibilidad de los paquetes y dependencias necesarios para garantizar un entorno coherente y reproducible. Luego de importar las librerías necesarias, se definen tests los cuales funciones hechas mas adelante por nosotros deberán pasar.

Código 1: Tests a superar

```python
import os
import numpy as np
import tensorflow as tf
import random
from unittest.mock import MagicMock


def _print_success_message():
    print('Pruebas superadas.')
    print('Puede pasar a la siguiente tarea.')

def test_normalize_images(function):
    test_numbers = np.array([0,127,255])
    OUT = function(test_numbers)
    test_shape = test_numbers.shape

    assert type(OUT).__module__ == np.__name__,\
        'Not Numpy Object'

    assert OUT.shape == test_shape,\
        'Incorrect Shape. {} shape found'.format(OUT.shape)
    np.testing.assert_almost_equal(test_numbers/255, OUT)

    _print_success_message()

def test_one_hot(function):
    test_numbers = np.arange(10)
    number_classes = 10
    OUT = function(test_numbers,number_classes)

    awns = np.identity(number_classes)
    test_shape = awns.shape

    assert type(OUT).__module__ == np.__name__,\
        'Not Numpy Object'

    assert OUT.shape == test_shape,\
        'Incorrect Shape. {} shape found'.format(OUT.shape)
```

```
    np.testing.assert_almost_equal(awns, OUT)

    _print_success_message()
```

B. Preprocesamiento de Datos

Las bases de datos MNIST y CIFAR-10 se preprocesaron para adaptarlas a la entrada de la red neuronal. Este proceso incluyó la normalización de píxeles y, en el caso de CIFAR-10, la adaptación de las etiquetas a las clases correspondientes. Para el caso de MNIST, cargamos y graficamos la 5° muestra del conjunto de entrenamiento de la siguiente manera:

Código 2: Codigo para cargar la base de datos y la muestra antes mencionada

```
# Los gr ficos se mostrar n dentro del Jupyter Notebook
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

# NumPy es un paquete para manipular objetos de matriz de N
      dimensiones
import numpy as np

# Pandas es un paquete de an lisis de datos
import pandas as pd

# Mnist wrapper y
from keras.datasets import mnist

# C digo para cargar la base de datos MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Imprimir la dimensionalidad de los datos
print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_train.shape))
print('Shape of y_test {}'.format(y_train.shape))


# C digo para graficar la 5   muestra del conjunto de
      entrenamiento
fig,ax1 = plt.subplots(1,1, figsize=(7, 7))

ax1.imshow(x_train[5], cmap='gray')
title = 'Target = {}'.format(y_train[5])
ax1.set_title(title)
ax1.grid(which='Major')
ax1.xaxis.set_major_locator(MaxNLocator(28))
ax1.yaxis.set_major_locator(MaxNLocator(28))
fig.canvas.draw()
time.sleep(0.1)
```

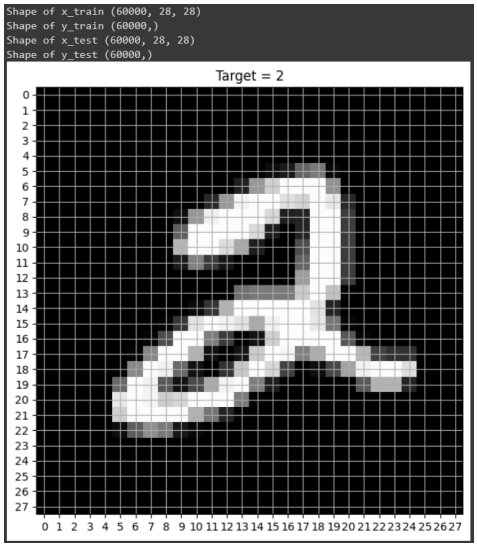

Fig. 1: Muestra número 5 del conjunto de entrenamiento

Código 3: Funciones a probar

```
def normalize_images(images):
  images = images.astype('float32') / 255.0  # Normaliza
     las im genes dividiendo entre 255

  return images

### *No* modificar las siguientes l neas ###
test_normalize_images(normalize_images)

# Normalizar los datos para su uso futuro
x_train = normalize_images(x_train)
x_test = normalize_images(x_test)

def one_hot(vector, number_classes):
    """Devuelve una matriz codificada one-hot dado el
     vector argumento.
    """
    # Aqu  almacenaremos nuestros one-hots
    one_hot= []

    # Aqu  se codifica el 'vector' one-hot
    for val in vector:
        one_hot1 = [0] * number_classes  # Inicializar un
     array de ceros con longitud number_classes
        one_hot1[val] = 1  # Establecer el valor en 1 en la
      posici n indicada por el valor en el vector
        one_hot.append(one_hot1)  # Agregar el one-hot
     actual a la lista

    # Transformar la lista en una matriz numpy y retornarla
    return np.array(one_hot)


### *No* modifique las siguientes lineas ###
test_one_hot(one_hot)

# One-hot codifica los labels de MNIST
y_train = one_hot (y_train, 10)
y_test = one_hot(y_test, 10)
```

C. Diseño de la Arquitectura de las CNN para MNIST

Se diseñó una arquitectura de red neuronal convolucional, considerando las características específicas de las imágenes de MNIST y CIFAR-10. Se configuraron capas de convolución, activación, y pooling, así como capas completamente conectadas para la clasificación final.

Código 4: Funcion net1 la cual se usará para crear la primera CNN

```
# Importar la librer a Keras
import keras
from keras.models import Model
from keras.layers import *


def net_1(sample_shape, nb_classes):
    # Defina la entrada de la red para que tenga la
     dimensi n 'sample_shape'
    input_x = Input(shape=sample_shape)

    # Generar 32 kernel maps utilizando una capa
     convolucional
    x = Conv2D(32, (3, 3), padding='same', activation='relu
     ')(input_x)

    # Generar 64 kernel maps utilizando una capa
     convolucional
    x = Conv2D(64, (3, 3), padding='same', activation='relu
     ')(x)

    # Reducir los feature maps utilizando max-pooling
    x = MaxPooling2D(pool_size=(2, 2), padding='same')(x)

    # Aplanar el feature map
    x = Flatten()(x)

    # Capa fully-connected, 128 dimensiones con activaci n
      ReLU
    x = Dense(128, activation='relu')(x)

```

```
    # Capa fully-connected a nb_classes dimensiones con activaci n Softmax
    probabilities = Dense(nb_classes, activation='softmax')(x)

    # Definir la salida
    model = Model(inputs=input_x, outputs=probabilities)

    return model
```

Código 5: Creacion del modelo usando net1

```
# Dimensi n de la muestra
sample_shape = x_train[0].shape

# Construir una red
model = net_1(sample_shape, 10)
model.summary()
```

Red la cual nos entrega el siguiente summary:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 28, 28, 1)]       0

conv2d_12 (Conv2D)           (None, 28, 28, 32)        320

conv2d_13 (Conv2D)           (None, 28, 28, 64)        18496

max_pooling2d_4 (MaxPoolin   (None, 14, 14, 64)        0
g2D)

flatten_6 (Flatten)          (None, 12544)             0

dense_12 (Dense)             (None, 128)               1605760

dense_13 (Dense)             (None, 10)                1290

=================================================================
Total params: 1625866 (6.20 MB)
Trainable params: 1625866 (6.20 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Fig. 2: Net_1 summary

D. *Entrenamiento de los modelos para MNIST*

Los modelos se entrenaron utilizando el conjunto de datos de entrenamiento, y se ajustaron los hiperparámetros, como la tasa de aprendizaje y el tamaño del lote, para optimizar el rendimiento. Se aplicó un enfoque de validación cruzada para evaluar la generalización del modelo.

Código 6: Entrenamiento del modelo net1

```
# Defina los hiperpar metros
batch_size = 32
epochs = 30

### *No* modifique las siguientes l neas ###

# No hay tasa de aprendizaje porque estamos usando los valores recomendados
# para el optimizador Adadelta. M s informaci n aqu :
# https://keras.io/optimizers/

# Necesitamos compilar nuestro modelo
model.compile(loss='categorical_crossentropy',
              optimizer='Adadelta',
              metrics=['accuracy'])

# Entrenar
logs = model.fit(x_train, y_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=2,
                 validation_split=0.1)

# Graficar losses y el accuracy
fig, ax = plt.subplots(1,1)

pd.DataFrame(logs.history).plot(ax=ax)
ax.grid(linestyle='dotted')
ax.legend()
```

```
plt.show()

# Evaluar el rendimiento
print('='*80)
print('Assesing Test dataset...')
print('='*80)

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Donde se puede apreciar la libre elección de los hiperparámetros "batch" y "epochs" los cuales escogimos los valores 32 y 30 respectivamente,además de usar el optimizador Adadelta.

Para el la creación del segundo modelo, se implementó la función net2 que vemos a continuación:

Código 7: Funcion net2

```
def net_2(sample_shape, nb_classes):
    #Defina la entrada de la red para que tenga la dimensi n `sample_shape`
    input_x = Input(shape=sample_shape)

    # Generar 32 kernel maps utilizando una capa convolucional
    x = Conv2D(32, (3, 3), padding='same', activation='relu')(input_x)

    # Generar 64 kernel maps utilizando una capa convolucional con stride=2
    x = Conv2D(64, (3, 3), padding='same', activation='relu', strides=2)(x)

    # Aplanar el feature map
    x = Flatten()(x)

    # Capa fully-connected, 128 dimensiones con activaci n ReLU
    x = Dense(128, activation='relu')(x)

    # Capa fully-connected a nb_classes dimensiones con activaci n Softmax
    probabilities = Dense(nb_classes, activation='softmax')(x)

    # Definir la salida
    model = Model(inputs=input_x, outputs=probabilities)

    return model
```

Esta vez, se remueve el Max Pooling y se añade el parametro strides=2 al segundo bloque de convolución. Veamos su respectivo summary:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, 28, 28, 1)]       0

conv2d_14 (Conv2D)           (None, 28, 28, 32)        320

conv2d_15 (Conv2D)           (None, 14, 14, 64)        18496

flatten_7 (Flatten)          (None, 12544)             0

dense_14 (Dense)             (None, 128)               1605760

dense_15 (Dense)             (None, 10)                1290

=================================================================
Total params: 1625866 (6.20 MB)
Trainable params: 1625866 (6.20 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Fig. 3: Net_2 summary

A la hora de entrenar el modelo, se hace exactamente igual que el anterior.

E. Diseño de la Arquitectura de las CNN para CIFAR

Primero cargamos la base de datos y la 5ta muestra del conjunto de entrenamiento:

Código 8: Cargamos la base de datos y la muestra antes mencionada

```
from keras.datasets import cifar10

# Los datos divididos entre los conjuntos de entrenamiento
    y prueba
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

target_2_class = {0:'airplane',
                  1:'automobile',
                  2:'bird',
                  3:'cat',
                  4:'deer',
                  5:'dog',
                  6:'frog',
                  7:'horse',
                  8:'ship',
                  9:'truck'}

# C digo para graficar la 5   muestra de entrenamiento.
fig,ax1 = plt.subplots(1,1, figsize=(7,7))
ax1.imshow(x_train[5])
target = y_train[5][0]
title = 'Target is {} - Class {}'.format(target_2_class[
    target],target )
ax1.set_title(title)
ax1.grid(which='Major')
ax1.xaxis.set_major_locator(MaxNLocator(32))
ax1.yaxis.set_major_locator(MaxNLocator(32))
fig.canvas.draw()
time.sleep(0.1)

print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_train.shape))
print('Shape of y_test {}'.format(y_train.shape))
```

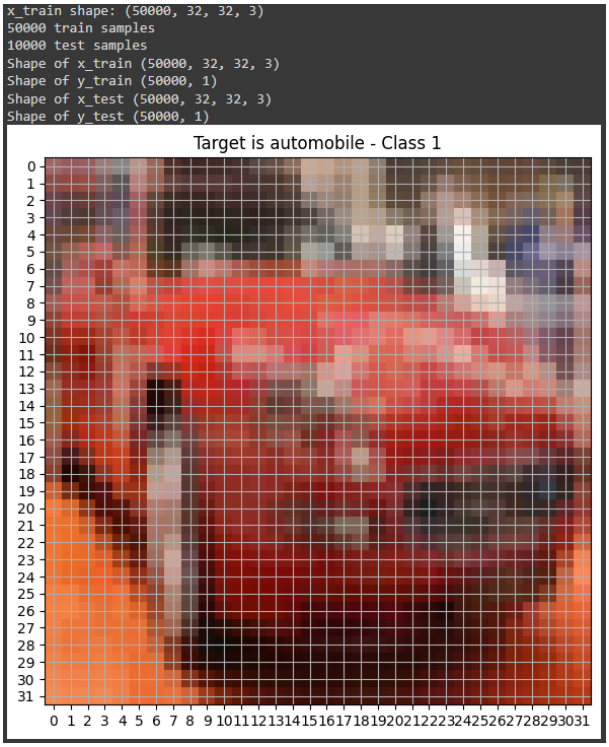

Fig. 4: Muestra número 5

Luego se usa la función creada anteriormente onehot() para codificar ytrain y ytest:

Código 9: Uso de la función one_hot

```
y_train = one_hot(y_train.reshape(-1), 10)
y_test = one_hot(y_test.reshape(-1), 10)


### *No* modifique las siguientes l neas ###
# Imprima los tama os de los datos (variables)
print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_train.shape))
print('Shape of y_test {}'.format(y_train.shape))
```

También se normalizan las imágenes usando la función normalize_images() creada anteriormente

Código 10: Uso de la función normalize_images

```
x_train = normalize_images(x_train)
x_test = normalize_images(x_test)
```

Luego el modelo es creado en base a la función net_1() de la forma:

Código 11: Funcion net_2

```
# Dimensionalidad de la muestra
sample_shape = x_train[0].shape

# Construcci n de la red
model = net_1(sample_shape, 10)
model.summary()

# Necesitamos compilar nuestro modelo de red neuronal
model.compile(loss='categorical_crossentropy',
              optimizer='Adagrad',
              metrics=['accuracy'])
```

Además posee las siguientes características:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        [(None, 32, 32, 3)]       0

conv2d_20 (Conv2D)           (None, 32, 32, 32)        896

conv2d_21 (Conv2D)           (None, 32, 32, 64)        18496

max_pooling2d_7 (MaxPoolin   (None, 16, 16, 64)        0
g2D)

flatten_10 (Flatten)         (None, 16384)             0

dense_20 (Dense)             (None, 128)               2097280

dense_21 (Dense)             (None, 10)                1290

=================================================================
Total params: 2117962 (8.08 MB)
Trainable params: 2117962 (8.08 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Fig. 5: Sumario del modelo creado usando net_1() para CIFAR

F. Entrenamiento del modelo para CIFAR

Y esta vez es entrenado con los hiperparámetros definidos en batch = 128 y epochs = 30, además de usar validationsplit = 0.2 a la hora de llamar .fit() como podemos ver a continuación:

Código 12: Entrenamiento del modelo para CIFAR

```
# Construya el c digo dentro de esta celda
# Defina los hiperpar metros
batch_size = 128
epochs = 30

### *No* modifique las siguientes l neas ###

# No hay tasa de aprendizaje porque estamos usando los
    valores recomendados
# para el optimizador Adadelta. M s informaci n aqu :
# https://keras.io/optimizers/


# Entrenar
logs = model.fit(x_train, y_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=2,
                 validation_split=0.2)



# Evaluar el rendimiento
print('='*80)
print('Assesing Test dataset...')
print('='*80)

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

IV. Resultados

A continuación, se presentan los principales resultados obtenidos por cada uno de los modelos, siguiendo los requisitos establecidos en el enunciado.

Para empezar la exposición de los resultados, podemos empezar con la adición de dimensión a los datos de entrada en el código:

Código 13: Código para ampliar la dimensión de la entrada con numpy.expand_dims()

```
# Escribe aqu  tu c digo
x_train = np.expand_dims(x_train, axis=3)
x_test = np.expand_dims(x_test, axis=3)

### *No* modifique las siguientes lineas ###
print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_test.shape))
print('Shape of y_test {}'.format(y_test.shape))
```

Quedando así,

```
Shape of x_train (60000, 28, 28, 1)
Shape of y_train (60000,)
Shape of x_test (10000, 28, 28, 1)
Shape of y_test (10000,)
```

Fig. 6: Nuevas dimensiones de la entrada

Con esto, los resultados graficados del entrenamiento de las CNN net_1, net_2 y el modelo creado basado en net_1 son:

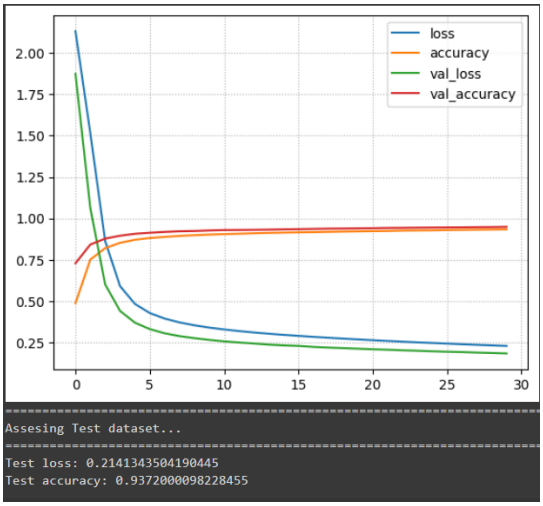

Fig. 7: Gráfico entrenamiento CNN net_1()

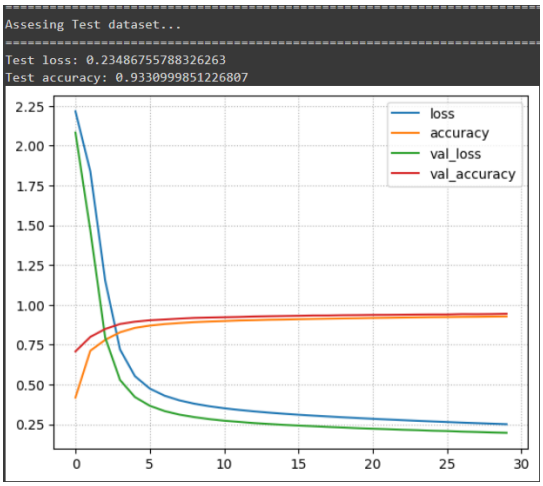

Fig. 8: Gráfico entrenamiento CNN net_2()

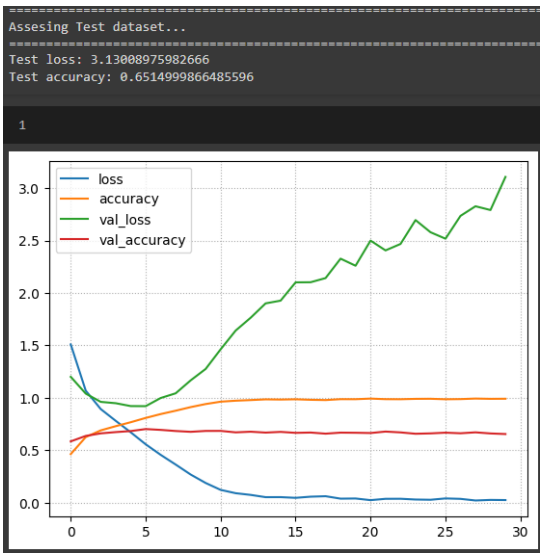

Fig. 9: Gráfico entrenamiento de la CNN basada en net_1()

V. Análisis

La implementación y entrenamiento de las Redes Neuronales Convolucionales (CNN) para la clasificación de imágenes en las bases de datos MNIST y CIFAR-10 proporcionaron resultados significativos. A continuación, se presenta un análisis detallado de los principales hallazgos y desafíos encontrados durante este proceso.

A. Rendimiento en la Base de Datos MNIST

El modelo logró alcanzar una precisión destacada en la clasificación de dígitos manuscritos de la base de datos MNIST. La arquitectura de la CNN, diseñada específicamente para este conjunto de datos, demostró ser efectiva en la extracción de características relevantes. La tasa de aciertos en el conjunto de prueba valida la capacidad del modelo para generalizar patrones aprendidos durante el entrenamiento. Además encontramos una mejora en el loss de 1,903 y en el val_loss una variación de 1.6908 entre la primera época y la última mientras que en net_2 una variación en el loss de 1,9653 y en su val_loss de 1.8856 por lo que la diferencia de performance es mínima al eliminar el Max-Pooling, además los valores de Test loss y Test accuracy son casi idénticos tal como los gráficos anteriormente expuestos.

B. Desafíos en la Base de Datos CIFAR-10

La clasificación de imágenes más complejas en CIFAR-10 presentó desafíos adicionales. Aunque el modelo alcanzó una precisión significativa, se observaron áreas donde la clasificación resultó más difícil. Las clases con características visuales similares o la presencia de detalles finos representaron obstáculos para la CNN. Este análisis destaca la importancia de adaptar la arquitectura de la red y los hiperparámetros según las características específicas de cada conjunto de datos. En el último gráfico expuesto podemos ver un cambio notorio en las métricas puesto que el val_loss comienza a converger con las demás métricas pero después de unas cuantas épocas comienza a dispararse subiendo su valor incluso mas altos de los del inicio.

C. Evaluación de Hiperparámetros

Se realizaron experimentos con diferentes configuraciones de hiperparámetros, como tasas de aprendizaje y tamaño de lote, para evaluar su impacto en el rendimiento. Estos ajustes jugaron un papel crucial en la convergencia del modelo y su capacidad para generalizar. El análisis detallado de las curvas de pérdida y precisión durante el entrenamiento y la validación permitió identificar la influencia de estos hiperparámetros en el rendimiento final.

D. Posible Evidencia de Overfitting

Durante el análisis, se observaron indicios de posible sobreajuste en algunos experimentos. Las disparidades entre las métricas de entrenamiento y validación sugirieron que el modelo podría haber memorizado características específicas del conjunto de entrenamiento. Este fenómeno destaca la importancia de técnicas de regularización y la necesidad de ajustar la complejidad del modelo para evitar la sobreoptimización.

VI. Conclusiones Generales

A. ¿Qué modelo funciona mejor?

El modelo más sólido es el de net_1() dado a su convergencia de los valores de loss yu val_loss a pesar de ser parecidos al de net_2() y tomando en cuenta los tiempos. En el caso del modelo empleado en CIFAR si bien llega a valores de loss muy bajos y óptimos, el val_loss se dispara por lo que pierde calidad.

B. ¿Qué optimizador funciona mejor?

En los dos modelos usados en MNIST se usó Adadelta arrojando resultados y gráficos adecuados a los requerimientos, sin embargo cuando se usó Adam en el modelo usado sobre CIFAR, muestra una mala calidad respecto al val_loss por lo que no se ajusta al propósito de la tarea por ese lado. Entonces podriamos decir que Adadelta es el optimizador que predomina.

C. ¿Existe alguna evidencia de overfitting?

Si, al no controlar el overfitting en el modelo de CIFAR se nota a simple vista con los valores graficados.

D. ¿Cómo podemos mejorar aún más el rendimiento?

Para mejorar el rendimiento, se podrian considerar estrattegias como:

Regularización: Aplica técnicas como dropout o regularización L1/L2 para reducir el overfitting.

Ajuste de Hiperparámetros: Experimenta con diferentes tasas de aprendizaje, tamaños de lote, y otros hiperparámetros para encontrar la combinación óptima.

VII. Bibliografía

https://datasmarts.net/es/la-historia-del-conjunto-de-datos-mas-popular-#:~:text=MNIST%20significa%20Modified%20NIST%2C%20donde,imagen%20en%20blanco%20y%20negro.

https://www.cs.toronto.edu/~kriz/cifar.html

https://interactivechaos.com/es/manual/tutorial-de-machine-learning/one-hot-encoding

https://www.ibm.com/es-es/topics/convolutional-neural-networks