Universidad de O'Higgins

Tarea N° #3:
Informe #3

Sebastián Garrido

COM4402 – Introducción a Inteligencia Artificial
Escuela de Ingeniería, Universidad de O'Higgins
10, Noviembre, 2023

I. Resumen

En este trabajo, se busca mediante la implementación de una red neuronal convolucional, resolver un problema de clasificación de labels o etiquetas a 2 bases de datos propuestas en el ejercicio respectivo.

II. Introduction

En este informe, se hará presente la ejecución, procedimiento, análisis de resultados, estudios, definiciones y conclusiones relacionados al trabajo de clasificación de una red neuronal convolucional que fue realizado. Su lectura se basará en brindar un marco teórico que defina conceptos claves para esta temática, la metodología que explique lo realizado de forma programática, la revisión de resultados, el análisis del procedimiento y lo que se observó, las conclusiones generales de lo que se extrae a partir de lo realizado, su respectivo resumen, y finalmente la exposición de la bibliografía utilizada para hacer este informe.

III. *Marco teórico*

A. *Redes neuronales: "Una **red neuronal** es un modelo simplificado que emula el modo en que el cerebro humano procesa la información: Funciona simultaneando un número elevado de unidades de procesamiento interconectadas que parecen versiones abstractas de neuronas", esto también se puede apreciar cuando las redes neuronales son usadas para generar respuestas, cálculos, entre otras actividades que simulen el pensamiento humano mediante un proceso de aprendizaje profundo.*

B. *Overfiting: Un modelo neuronal en el que exista la presencia de Overfitting, será aquel donde se obtiene un error de entrenamiento relativamente bajo, y un error de validación relativamente alto.*

C. *Underfitting: Un modelo neuronal que tenga Underfitting como característica de sus datos, será aquel cuyos "errores tanto de entrenamiento como de validación son similares y relativamente altos"*

D. *Deep Learning: También conocido como aprendizaje profundo, corresponde a una forma de aprendizaje automático, "donde una máquina intenta imitar al cerebro humano utilizando redes neuronales artificiales con más de tres capas que le permiten hacer predicciones con una gran precision"*

E. *Redes neuronales convolucionales: Es un tipo regularizado de red neuronal de retroalimentación que procesa características mediante la optimización de filtros. " se distinguen de otras redes neuronales por su rendimiento superior con entradas de imagen, voz o señales de audio"*

F. *Función de activación: Tiene la labor de imponer un límite o modificar el valor resultado para poder proseguir a otra neurona. En otras palabras," es una función que transmite la información generada por la combinación lineal de los pesos y las entradas , es decir son la manera de transmitir la información por las conexiones de salida"*

G. *Normalización: "es la organización de datos de manera coherente para reducir la redundancia y mejorar la integridad de los datos", ayudando así a evitar errores y mejorar la eficiencia.*

H. *Pérdida (loss): La función de pérdida evalúa la desviación entre las predicciones y cálculos realizados por la red neuronal, y los valores reales de las observaciones utilizadas durante el aprendizaje." Cuanto menor es el resultado de esta función, más eficiente es la red neuronal."*

I. *Pooling: Es una operación que generalmente se aplica entre dos capas de deconvolución, y tiene como objetivo reducir el tamaño de las imágenes, y a la vez, preservar sus características más esenciales.*

J. *Hiperparámetro: es un parámetro cuyo valor se utiliza para controlar el proceso de aprendizaje.*

K. *Matriz de confusion: Tiene la función de "valorar cómo de bueno es un modelo de clasificación basado en aprendizaje automático", caracterizándose principalmente en mostrar explícitamente cuando una clase se confunde con otra, permitiendo trabajar de forma separada con diferentes tipos de errores.*

L. *Accuracy: corresponde al porcentaje de clasificaciones correctas que un modelo de aprendizaje entrenado logra, dentro de todas las que ejecuta.*

M. *Epocs (épocas): "Es el número total de iteraciones de todos los datos de entrenamiento en un ciclo para entrenar el modelo de aprendizaje automático".*

N. *Optimizadores: Son algoritmos o métodos que se usan para ajustar los parámetros de una red neuronal, como los pesos y el sesgo, para minimizar la función de pérdida*

O. *Strides: es el desplazamiento de un pixel hacia un sentido durante cada iteración.*

P. *Padding: consiste en agregar píxeles de valor cero alrededor de la imagen original, garantizando que ante la convolución, la imagen conserve su tamaño original*

Q. *Pooling: Es una operación que permite analizar el contenido de una imagen por bloques para extraer la información más representativa de las mismas. En el caso de max pooling, permite reducir la cantidad de datos entre una capa y otra, facilitando así el procesamiento de imágenes y entrenamiento de red, pero preservando la información más relevante*

IV. METODOLOGÍA

Considerando el Desarrollo de la tarea, cabe detallar en el sentido y explicación del código.

A. Parte 0

En esta parte del código (cedido con anterioridad para permitir el desarrollo de la tarea), se ejecutan todas las librerías cedidas y añadidas para realizar lo pedido, además de la lectura de datasets, dígitos, definición de funciones, preprocesamiento de los datos y la próxima implementación de las funciones.

```
[ ] """Esta area configura el entorno Jupyter.
    Por favor, no modifique nada en esta celda.
    """
    import os
    import sys
    import time

    # Importar módulos diversos
    from IPython.core.display import display, HTML
```

```
[ ] import os
    import numpy as np
    import tensorflow as tf
    import random
    from unittest.mock import MagicMock
    # TensorFlow y Keras
    import tensorflow as tf
    from tensorflow import keras

    # Librerias Auxiliares
    import numpy as np
    import matplotlib.pyplot as plt

    print("Versión de TensorFlow: ", tf.__version__)

    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()

    Versión de TensorFlow:  2.14.0
```

B. Parte 1

La parte 1 de esta tarea consta simplemente de corregir el código cedido para que funcione, implementando así una función de normalización de imágenes con entrada en el intervalo [0,255] y con salida en el intervalo [0,1]

```
] def normalize_images(images):
      """Normalizar las imágenes de entrada.
      """
      # Normalizar la imagen aquí

      return images/255     #se define 255 como el número más alto a alcanzar en el intervalo [0,255]

  ### *No* modificar las siguientes líneas ###
  test_normalize_images(normalize_images)

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

  Pruebas superadas.
  Puede pasar a la siguiente tarea.
```

para garantizar su funcionamiento, se le añadió /255 al comando que retorna images.

C. *Parte 2*

En esta parte, se busca transformar el tamaño de la matriz que representa el conjunto de datos MNIST ya cargado, de (28,28) a (28,28,1). Esto para que se pueda usar el concepto de canales de color y mapas de características, incluyendo las imágenes en escala de grises. Para esto, se modifican las variables de x_train y x_test para ajustar su dimensionalidad, aplicando numpy y definiendo como axis=-1.

```
[ ] # Escribe aquí tu código
    x_train = np.expand_dims(x_train, axis=-1)
    x_test = np.expand_dims(x_test, axis=-1)   #se definen los parámetros de x train y x text para visualizar su dimensionalidad.

    ### *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))

    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,)
```

D. *Parte 3*

Una vez detallado el preprocesamiento de los labels, y que se requiere del uso de One-hot encoding para representar las salidas de los objetivos, se tiene que implementar una función que lo aplique al ejercicio. Por ende, además del código base se crea dentro de la función códigos que devuelvan una matriz codificada one-hot dado el vector argumento.

```
import numpy as np

def one_hot(vector, number_classes):
    """Devuelve una matriz codificada one-hot dado el vector argumento.
    """
    # se invoca una lista vacía para almacenar los one-hots
    one_hot = []

    # se crea un for para cada elemento en el vector
    for sample in vector:
        # se define array de ceros con la longitud de number_classes
        one_hot_sample = np.zeros(number_classes)
        # se define índice correspondiente a la muestra en 1
        one_hot_sample[sample] = 1
        one_hot.append(one_hot_sample)

    # se transforma la lista en una matriz numpy y luego se retorna.
    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)
```

```
Pruebas superadas.
Puede pasar a la siguiente tarea.
```

Después, se visualiza las formas de las matrices de datos, o en otras palabras, las dimensiones de las variables x_train, y_train, x_text e y_test, tal cual como se define de base en el código dado.

```
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))
```

```
Shape of x_train (60000, 28, 28, 1)
Shape of y_train (60000, 10)
Shape of x_test (60000, 28, 28, 1)
Shape of y_test (60000, 10)
```

E. *Parte 4*

Luego de recibir una extensa orientación sobre definiciónes e implementaciones de una red convolucional y los conceptos de pooling, stride y padding, llega la hora de implementar la primera red neuronal convolucional, esto creando una función net_1, la cual basándose en el código base quedaría de la siguiente manera:

```
[8] # Importar la librería Keras
    import keras
    from keras.models import Model
    from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

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

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

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

        # se reducen los feature maps utilizando max-pooling
        x = MaxPooling2D(pool_size=(2, 2))(x)

        # se aplana el feature map
        x = Flatten()(x)

        # Dense `nb_classes`
        x = Dense(128, activation='relu')(x)
        probabilities = Dense(nb_classes, activation='softmax')(x)

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

        return model
```

En el siguiente fragmento de código

- Crearemos una red utilizando la función que acabas de implementar
- Mostrar un resumen de la red

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

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

Esto además incluye la muestra de su dimensión, y la construcción de una red.

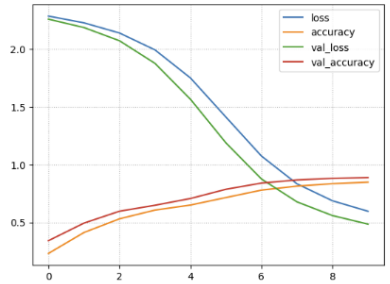
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 28, 28, 32)        320

 conv2d_1 (Conv2D)           (None, 28, 28, 64)        18496

 max_pooling2d (MaxPooling2  (None, 14, 14, 64)        0
 D)

 flatten (Flatten)           (None, 12544)             0

 dense (Dense)               (None, 128)               1605760

 dense_1 (Dense)             (None, 10)                1290

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

F. *Parte 5*

Luego de definer la red anterior, ahora se busca definir los hiperparámetros por los cuales la red podrá aprender. Para esto, se define un tamaño de batch de 128, con 10 épocas, lo cual provee una convergencia a números bajos, que es lo que se buscaba al afinar los hiperparámetros.

```
# Defina los hiperparámetros
batch_size = 128
epochs = 10

### *No* modifique las siguientes líneas ###

# No hay tasa de aprendizaje porque estamos usando los valores recomendados
# para el optimizador Adadelta. Más información aquí:
# 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])
```

```
Epoch 1/10
422/422 - 14s - loss: 2.2859 - accuracy: 0.2311 - val_loss: 2.2594 - val_accuracy: 0.3415 - 14s/epoch - 33ms/step
Epoch 2/10
422/422 - 6s - loss: 2.2277 - accuracy: 0.4121 - val_loss: 2.1870 - val_accuracy: 0.4935 - 6s/epoch - 14ms/step
Epoch 3/10
422/422 - 3s - loss: 2.1398 - accuracy: 0.5306 - val_loss: 2.0727 - val_accuracy: 0.5965 - 3s/epoch - 8ms/step
Epoch 4/10
422/422 - 4s - loss: 1.9935 - accuracy: 0.6063 - val_loss: 1.8769 - val_accuracy: 0.6480 - 4s/epoch - 9ms/step
Epoch 5/10
422/422 - 4s - loss: 1.7503 - accuracy: 0.6498 - val_loss: 1.5657 - val_accuracy: 0.7065 - 4s/epoch - 9ms/step
Epoch 6/10
422/422 - 4s - loss: 1.4113 - accuracy: 0.7149 - val_loss: 1.1880 - val_accuracy: 0.7865 - 4s/epoch - 9ms/step
Epoch 7/10
422/422 - 3s - loss: 1.0742 - accuracy: 0.7793 - val_loss: 0.8756 - val_accuracy: 0.8402 - 3s/epoch - 8ms/step
Epoch 8/10
422/422 - 3s - loss: 0.8338 - accuracy: 0.8138 - val_loss: 0.6776 - val_accuracy: 0.8673 - 3s/epoch - 8ms/step
Epoch 9/10
422/422 - 4s - loss: 0.6868 - accuracy: 0.8349 - val_loss: 0.5589 - val_accuracy: 0.8813 - 4s/epoch - 9ms/step
Epoch 10/10
422/422 - 3s - loss: 0.5963 - accuracy: 0.8481 - val_loss: 0.4847 - val_accuracy: 0.8875 - 3s/epoch - 8ms/step
```




```
================================================================================
Assesing Test dataset...
================================================================================
Test loss: 0.5312371850013733
Test accuracy: 0.8716999888420105
```

Cabe resaltar que los resultados son los siguientes:

Tiempo: 50 segundos

Test loss: 0.5312371850013733

Test accuracy: 0.8716999888420105

G. *Parte 6*

En la parte 6, después de exponer acerca de la factibilidad de usar Max Pooling para reducir la cantidad de datos entre capas de una red, se implementa una red neuronal convolucional sin capas de pooling, haciendo lo mismo que en la parte 4, pero removiendo la capa de max pooling y añadiendo un stride = 2 al bloque de convolución.

```
from keras.models import Model
from keras.layers import Input, Conv2D, Dense, Flatten

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

    # Primer bloque de convolución
    x = Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same')(input_x)

    # Segundo bloque de convolución con stride=2
    x = Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same', strides=(2, 2))(x)

    # Se aplana el feature map
    x = Flatten()(x)

    # Dense number_classes
    x = Dense(128, activation='relu')(x)
    probabilities = Dense(nb_classes, activation='softmax')(x)

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

    return model
```

En el siguiente fragmento de código:

- Crearemos una red utilizando la función que acabamos de implementar
- Mostraremos un resumen de la red

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

     # Construir una red
     model = net_2(sample_shape, 10)
     model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d_2 (Conv2D)           (None, 28, 28, 32)        320

 conv2d_3 (Conv2D)           (None, 14, 14, 64)        18496

 flatten_1 (Flatten)         (None, 12544)             0

 dense_2 (Dense)             (None, 128)               1605760

 dense_3 (Dense)             (None, 10)                1290

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

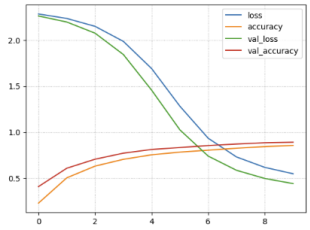
H. *Parte 7*

En este punto se repite el punto 5, para así definir los hiperplanos por los cuales este modelo (sin max pooling)

pueda aprender y entrenar.

```
# Defina los hiperparámetros
batch_size = 128
epochs = 10

### *No* modifique las siguientes líneas ###

# 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()
fig.canvas.draw()


# 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])
```

```
Epoch 1/10
422/422 - 4s - loss: 2.2827 - accuracy: 0.2312 - val_loss: 2.2608 - val_accuracy: 0.4103 - 4s/epoch - 9ms/step
Epoch 2/10
422/422 - 3s - loss: 2.2340 - accuracy: 0.5052 - val_loss: 2.1954 - val_accuracy: 0.6093 - 3s/epoch - 6ms/step
Epoch 3/10
422/422 - 3s - loss: 2.1485 - accuracy: 0.6322 - val_loss: 2.0743 - val_accuracy: 0.7070 - 3s/epoch - 6ms/step
Epoch 4/10
422/422 - 3s - loss: 1.9854 - accuracy: 0.7056 - val_loss: 1.8428 - val_accuracy: 0.7725 - 3s/epoch - 6ms/step
Epoch 5/10
422/422 - 2s - loss: 1.6902 - accuracy: 0.7549 - val_loss: 1.4558 - val_accuracy: 0.8128 - 2s/epoch - 6ms/step
Epoch 6/10
422/422 - 2s - loss: 1.2816 - accuracy: 0.7840 - val_loss: 1.0251 - val_accuracy: 0.8338 - 2s/epoch - 6ms/step
Epoch 7/10
422/422 - 2s - loss: 0.9345 - accuracy: 0.8049 - val_loss: 0.7408 - val_accuracy: 0.8550 - 2s/epoch - 5ms/step
Epoch 8/10
422/422 - 3s - loss: 0.7316 - accuracy: 0.8263 - val_loss: 0.5868 - val_accuracy: 0.8717 - 3s/epoch - 6ms/step
Epoch 9/10
422/422 - 2s - loss: 0.6184 - accuracy: 0.8440 - val_loss: 0.4981 - val_accuracy: 0.8855 - 2s/epoch - 6ms/step
Epoch 10/10
422/422 - 2s - loss: 0.5494 - accuracy: 0.8563 - val_loss: 0.4426 - val_accuracy: 0.8915 - 2s/epoch - 6ms/step
================================================================================
Assesing Test dataset...
================================================================================
Test loss: 0.48820650577545166
Test accuracy: 0.878000020980835
```



Se resalta que sin max pooling los resultados son los siguientes:

Tiempo: 27 segundos

Test loss: 0.48820650577545166

Test accuracy: 0.878000020980835

I. *Parte 8 y 9*

En esta parte se introduce el dataset de cifar10, basado en clases de imágenes y que se pondrá a prueba para desarrollar y evaluar modelos de aprendizaje profundo para la clasificación de imágenes. Para aquello, se aplica el siguiente código base

```
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'}

# Código para graficar la 5ª 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))
```

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
Shape of x_train (50000, 32, 32, 3)
Shape of y_train (50000, 1)
Shape of x_test (50000, 32, 32, 3)
Shape of y_test (50000, 1)
```

Luego, Se repite el código de One-hot para los labels correspondientes y_test e y_train, para luego aplicar la normalización de imágenes

Parte VIII: Codificación One-Hot para los Labels

Tarea: Utilice la función `one_hot()` que creó anteriormente para codificar:

- `y_test`
- `y_train`

```
[15] y_train = one_hot(y_train, 10)
     y_test = one_hot(y_test, 10)

     ### *No* modifique las siguientes líneas ###
     # Imprima los tamaños 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))

     Shape of x_train (50000, 32, 32, 3)
     Shape of y_train (50000, 10)
     Shape of x_test (50000, 32, 32, 3)
     Shape of y_test (50000, 10)
```

Parte IX: Normalizar las imágenes

Tarea: Utilice la función `normalize_images()` que creó anteriormente para normalizar las imágenes en:

- `x_test`
- `x_train`

```
[16] x_train = normalize_images(x_train)
     x_test = normalize_images(x_test)
```

J. *Parte 10 y 11*

Una vez realizado esto, llega el momento de crear una red neuronal para entrenar cifar. Para esto, se decidió usar el modelo net_2 (sin max pooling), debido a que previamente se observó que tiene mejor precisión, menor pérdida y una mayor optimización en su tiempo de ejecución que net_1, por ende y por preferencia se usa net_2. Luego, se procede a usar el código dado de la construcción de la red y su entrenamiento 3 veces, pero usando en el primer caso el optimizador Adadelta, en el segundo Adagrad, y en el tercero Adam.

```
[17] #Se ejecutan en el siguiente orden: Adadelta------------>Adagrad------------>Adam
```

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

     # Construcción de la red
     model = net_2(sample_shape, 10)
     model.summary()

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

```
Model: "model_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_9 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_16 (Conv2D)          (None, 32, 32, 32)        896

 conv2d_17 (Conv2D)          (None, 16, 16, 64)        18496

 flatten_8 (Flatten)         (None, 16384)             0

 dense_16 (Dense)            (None, 128)               2097280

 dense_17 (Dense)            (None, 10)                1290

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

(Cómo se puede observar, se introduce net_2 en model para la construcción de la red, y se escoge adadelta en el parámetro optimizer).

Luego se entrena el modelo, cuya concepción se compone del uso de dataset cifar-10, el uso de conjuntos de entrenamiento y validación, la aplicación de hiperparámetros y sus respectivos gráficos y evaluaciones.

```
[26] # Construya el código dentro de esta celda
     from keras.datasets import cifar10
     from keras.utils import to_categorical
     from sklearn.model_selection import train_test_split

     # Cargar datos CIFAR-10
     (x_train, y_train), (x_test, y_test) = cifar10.load_data()

     # Normalizar y realizar one-hot encoding para las etiquetas
     x_train = normalize_images(x_train)
     x_test = normalize_images(x_test)
     y_train = one_hot(y_train, 10)
     y_test = one_hot(y_test, 10)

     # Dividir conjunto de entrenamiento en entrenamiento y validación
     x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)

     # Construir la red
     sample_shape = x_train[0].shape
     model = net_1(sample_shape, 10)

     # Compilar el modelo
     model.compile(loss='categorical_crossentropy',
                   optimizer='Adadelta',
                   metrics=['accuracy'])

     # Definir hiperparámetros
     batch_size = 128
     epochs = 30

     # Entrenar el modelo
     logs = model.fit(x_train, y_train,
                      batch_size=batch_size,
                      epochs=epochs,
                      verbose=2,
                      validation_data=(x_val, y_val))

     # Graficar pérdidas y precisión
     fig, ax = plt.subplots(1, 2, figsize=(12, 4))

     # Gráfico de pérdidas
     ax[0].plot(logs.history['loss'], label='train_loss')
     ax[0].plot(logs.history['val_loss'], label='val_loss')
     ax[0].set_title('Modelo de pérdidas')
     ax[0].set_xlabel('Épocas')
     ax[0].set_ylabel('Pérdida')
     ax[0].legend()

     # Gráfico de precisión
     ax[1].plot(logs.history['accuracy'], label='train_accuracy')
     ax[1].plot(logs.history['val_accuracy'], label='val_accuracy')
     ax[1].set_title('Modelo de precisión')
     ax[1].set_xlabel('Épocas')
     ax[1].set_ylabel('Precisión')
     ax[1].legend()

     plt.show()

     # Evaluar el rendimiento en el conjunto de pruebas
     print('='*80)
     print('Assessing Test dataset...')
     print('='*80)

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

Este proceso se repite 3 veces, tanto para Adadelta, Adagrad y Adam.

V. *Resultados*

Considerando la metodología anteriormente explicada, ahora consta exhibir los resultados:

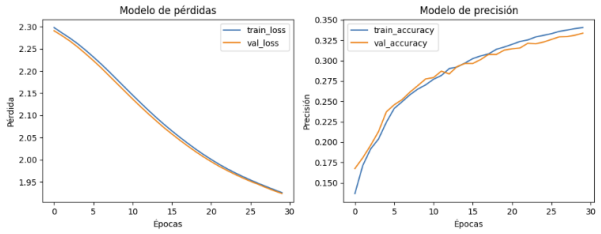
Considerando exclusivamente los resultados exhibidos y plasmados en este informe, respecto a la ejecución de código de ese momento, podemos definir los siguientes resultados:

A la hora de revisar las partes 5 y 7 relacionadas a la construcción de redes neuronales artificiales con convoluciones, toca acentuar que en el caso de ***Net_1*** *(max pooling), toma más tiempo en su ejecución (generalmente sobre 50 segundos), tiene una pérdida generalmente superior a 0.47 en los casos revisados, y una precisión de 0.870 a 0.875 en la mayoría de los casos. Por otro lado,* ***Net_2*** *al no usar Max-Pooling, pero al implementar más strides que 1, brinda un tiempo de ejecución relativamente bajo, (generalmente bajo los 30 segundos), además, posee una precisión robusta entre 0.870 y 0.878 (Siendo en este caso una precisión mayor a la vista en Net_1), y una perdida generalmente menor a 0.50. Dicho esto y absteniéndome a la situación de este informe en particular, la ausencia de Max pooling y el*

aumento de strides brinda resultados mayoritariamente similares a un proceso más complejo, más en este caso donde resulta ser mejor.

A partir de este resultado, se decidió por mera preferencia y eficiencia en el caso anterior, aplicar tal modelo a la resolución del punto 10 y 11 con los optimizadores ADADELTA, ADAGRAD, y ADAM.

En el caso de Adadelta, se puede observar lo siguiente:



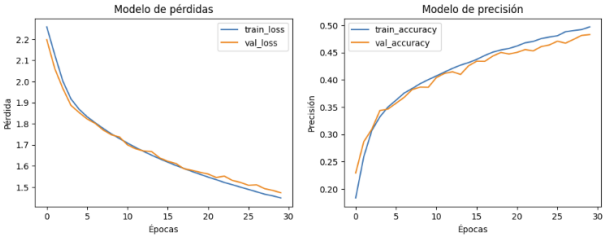

```
==========================================================================
Assessing Test dataset...
==========================================================================
Test loss: 1.9238064289093018
Test accuracy: 0.33889999985694885
```

Test loss: 1.9238064289093018

Test accuracy: 0.33889999985694885

Tiempo: 2 minutos

En el caso de Adagrad, se puede observar lo siguiente:



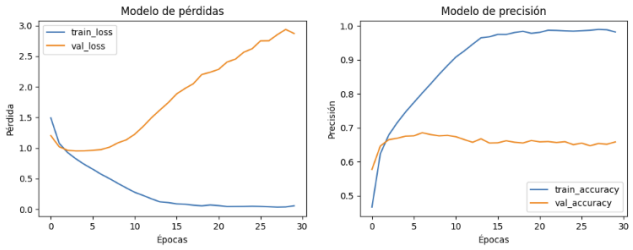

```
==========================================================================
Assessing Test dataset...
==========================================================================
Test loss: 1.4654375314712524
Test accuracy: 0.47760000824928284
```

Test loss: 1.4654375314712524

Test accuracy: 0.47760000824928284

Tiempo: 1 minuto

En el caso de Adam, se puede observar lo siguiente:




```
==========================================================================
Assessing Test dataset...
==========================================================================
Test loss: 3.0132527351379395
Test accuracy: 0.6531999707221985
```

Test loss: 3.0132527351379395

Test accuracy: 0.6531999707221985

Tiempo: 2 minutos

Visualizados estos datos, sus resultados, y considerando las diversas iteraciones hechas para responder de manera más robusta en relación a los datos arrojados, se puede determinar que el optimizador Adam es el mejor en términos de precisión. Si bien muestra fuertes casos de Overfitting, y con ello una alta cifra de pérdida, no deja de ser el optimizador de mejor desempeño.

En segundo lugar, está el optimizador Adagrad. En este proceso se observa una curva de pérdida bastante pareja (aunque con acentuaciones de overfitting en pérdidas y precisión), y por lo demás, arroja la menor cifra de pérdida entre los tres optimizadores. En adición, es el segundo con mejor precisión de los datos, y el proceso de menor demora en su ejecución.

Finalmente está el optimizador Adadelta, el cual posee el lugar medio de pérdida entre los 3 métodos con una cifra cercana a 2.0 generalmente, se observa además el menor caso de overfitting de los 3 escenarios. No obstante, es el optimizador de peor rendimiento ya que siempre queda marginado a la última posición una vez se comparan los datos. En temas de tiempo, tarda lo mismo que ADAM.

En resumen, en este caso en particular, el modelo Net_2 resulta ser el de mejor desempeño comparándose a Net_1, y ADAM es el mejor optimizador visualizado en la realización de este informe.

VI. *Análisis*

K. *En primer lugar, se responderá al punto 12 de preguntas teóricas:*

¿Qué modelo funciona mejor?:

El modelo que funciona mejor es el segundo modelo net_2, esto debido a que resultó tener una mejor precisión, menor pérdida y por mucho, menor tiempo de ejecución considerando la diferencia mínima entre modelos.

¿Qué optimizador funciona mejor?:

Considerando las iteraciones realizadas, el mejor optimizador resulta ser Adam si se considera netamente su precisión como factor determinante. Hay que añadir que adam se enfrenta a ejecuciones de mayor carga, por ende, se arriesga a más error.

¿Existe alguna evidencia de overfitting?:

Así es, siendo el caso más prominente los gráficos presenciados en la aplicación del optimizador Adam

¿Cómo podemos mejorar aún más el rendimiento?

El rendimiento podría mejorar si se realizan ajustes más refinados de los hiperparámetros, o que se exploren otras arquitecturas de red como bien puede ser net_1.

Dicho esto, y teniendo en cuenta los resultados expresados en el item anterior, se puede determinar que la aplicación de Max-Pooling en las estructuras de redes puede ser un elemento crucial para reducir la cantidad de datos entre capas de una red, aunque alternativas tales como el aumento de strides en convoluciones puede contribuir al mismo resultado, con mayor eficiencia en su ejecución.

Así también y como se pudo observar en los resultados, se puede observar que un optimizador complejo como Adam, puede arrojar resultados bastante precisos, a costa de una demora mayor y un profundo aumento de overfitting. Por otro lado, Adagrad si bien ofrece una buena precisión y rapidez en su procesamiento, presenta cuotas de overfitting. Mientras que un método libre de ruido, pero poco efectivo será aquel que aplique Adadelta como optimizador.

VII. *Conclusiones Generales*

En conclusion, es posible concluir que para la mayoría de los casos, un aumento en la complejidad de los procesos auguran un resultado más robusto, preciso, pero ruidoso, en general para el ítem de optimizadores respecto a Adam, y en la mayoría de los casos para la estructura de modelos (con el primer dataset de esta tarea) que incluya max pooling, Stride y Padding en la influencia de la cantidad de datos, y por consecuencia, un resultado positivo a costa de una gran demora.

VIII. *Resumen*

En resumen, se puede observar en el trabajo realizado una lenta, compleja y pesada ejecución del modelamiento, para cualquiera de las redes neuronales, yendo de menos a más a medida que se iba ejecutando el código. Aún así, se observa una ligera demora mayor en aquellos que tengan que lidiar con la aplicación de bases de datos complejas como CIFAR10, y optimizadores complejos como ADAM o ADADELTA. Posteriormente, se requirió hacer todos los cálculos posibles respecto a las 5 redes neuronales convolucionadas existentes para tener los resultados más fieles a lo solicitado, incluyendo ambos escenarios con los 2 modelos previamente guardados (Net_1 y Net_2) para la aplicación de optimizadores en relación del segundo dataset. Mientras que en los cálculos para el primer dataset dió como resultado general a Net_1 como el de mejor desempeño en la mayoría parcial de las situaciones estudiadas, y con un porcentaje cercano al 40% de las situaciones donde fue al revés. Por otro lado, en la totalidad de los casos para el tópico de optimizadores con el segundo dataset, se vió que Adam tuvo el mejor desempeño, seguido de Adagrad, y finalmente por Adadelta.

IX. *Bibliografía*

1. ***[1]*** *IBM. "Redes neuronales (SPSS Modeler)". [En línea]. Disponible en: https://www.ibm.com/docs/es/spss-modeler/saas?topic=networks-neural-model. Accedido el 23 de octubre de 2023.*
2. ***[2]*** *CodificandoBits. "Underfitting y Overfitting: Conceptos Esenciales". [En línea]. Disponible en: https://www.codificandobits.com/blog/underfitting-y-overfitting/#:~:text=Un%20modelo%20con%20underfitting%20es,uno%20de%20validaci%C3%B3n%20relativamente%20alto.. Accedido el 23 de octubre de 2023.*
3. ***[3]*** *Datademia. "¿Qué es el Deep Learning y qué es una Red Neuronal?". [En línea]. Disponible en: https://datademia.es/blog/que-es-deep-learning-y-que-es-una-red-neuronal. Accedido el 23 de octubre de 2023.*

4. ***[4]*** *DataScientest. "Perceptrón: qué es y para qué sirve". [En línea]. Disponible en: https://datascientest.com/es/perceptron-que-es-y-para-que-sirve. Accedido el 23 de octubre de 2023.*
5. ***[5]*** *Telefonica Tech. "Función de Activación en Redes Neuronales". [En línea]. Disponible en: https://aiofthings.telefonicatech.com/recursos/datapedia/funcion-activacion#:~:text=Una%20funci%C3%B3n%20de%20activaci%C3%B3n%20es,por%20las%20conexiones%20de%20salida.. Accedido el 23 de octubre de 2023.*
6. ***[6]*** *Universidad Internacional de la Rioja. "Backpropagation". [En línea]. Disponible en: https://www.unir.net/ingenieria/revista/backpropagation/. Accedido el 23 de octubre de 2023.*
7. ***[7]*** *Telefonica Tech. "Cómo interpretar la matriz de confusión: Ejemplo Práctico". [En línea]. Disponible en: https://telefonicatech.com/blog/como-interpretar-la-matriz-de-confusion-ejemplo-practico. Accedido el 23 de octubre de 2023.*
8. ***[8]*** *Ediciones ENI. "Inteligencia Artificial Fácil: Machine Learning y Deep Learning Prácticos". [En línea]. Disponible en: https://www.ediciones-eni.com/libro/inteligencia-artificial-facil-machine-learning-y-deep-learning-practicos-9782409025327/la-prediccion-con-neuronas. Accedido el 23 de octubre de 2023.*
9. ***[9]*** *Iguazio. "Model Accuracy in Machine Learning". [En línea]. Disponible en: https://www.iguazio.com/glossary/model-accuracy-in-ml/#:~:text=AI%20accuracy%20is%20the%20percentage,is%20often%20abbreviated%20as%20ACC.. Accedido el 23 de octubre de 2023.*
10. ***[10]*** *Huawei Enterprise. "¿Qué es Epoch en Machine Learning?". [En línea]. Disponible en: https://forum.huawei.com/enterprise/es/%C2%BFQu%C3%A9-es-Epoch-en-Machine-Learning/thread/667232453749784577-667212895009779712. Accedido el 23 de octubre de 2023.*

11. [1] IBM, "Convolutional Neural Networks," IBM, [En línea]. Disponible: https://www.ibm.com/es-es/topics/convolutional-neural-networks. [Accedido: 11 Nov 2023].

12. [2] J. I. Blanco, "Por qué la normalización es clave e importante en machine learning y ciencia de datos," Medium, [En línea]. Disponible: https://jorgeiblanco.medium.com/por-qu%C3%A9-la-normalizaci%C3%B3n-es-clave-e-importante-en-machine-learning-y-ciencia-de-datos-4595f15d5be0#:~:text=La%20normalizaci%C3%B3n%20se%20refiere%20a,errores%20y%20mejorar%20la%20eficiencia. [Accedido: 11 Nov 2023].

13. [3] Amazon Web Services, "Hyperparameter Tuning," AWS, [En línea]. Disponible: https://aws.amazon.com/es/what-is/hyperparameter-tuning/. [Accedido: 11 Nov 2023].

14. [4] DataScientest, "Convolutional Neural Network (CNN)," DataScientest, [En línea]. Disponible: https://datascientest.com/es/convolutional-neural-network-es#:~:text=Capa%20de%20Pooling%20(POOL)%3A,preservar%20sus%20caracter%C3%ADsticas%20m%C3%A1s%20esenciales. [Accedido: 11 Nov 2023].

15. Jordi Torres, "Técnicas de regulación en redes neuronales: un vistazo al aprendizaje profundo," LinkedIn, [En línea]. Disponible en: https://es.linkedin.com/pulse/t%C3%A9cnicas-de-regulaci%C3%B3n-redes-neuronales-un-vistazo-al-jordi#:~:text=Optimizadores%3A%20Los%20optimizadores%20son%20algoritmos,SGD)%2C%20Adam%20y%20RMSprop. [Accedido el 11 de noviembre de 2023].
16. CodificandoBits, "Padding, Strides, MaxPooling: Stacking en Redes Convolucionales," CodificandoBits, [En línea]. Disponible en: https://www.codificandobits.com/blog/padding-strides-maxpooling-stacking-redes-convolucionales/. [Accedido el 11 de noviembre de 2023].
17. Bootcamp AI, "Redes Neuronales Convolucionales," Bootcamp AI, [En línea]. Disponible en: https://bootcampai.medium.com/redes-neuronales-convolucionales-5e0ce960caf8#:~:text=Es%20una%20operaci%C3%B3n%20que%20se,tama%C3%B1o%20que%20la%20imagen%20original. [Accedido el 11 de noviembre de 2023]