12MBID10_YépezCallo_JuanCarlos

July 16, 2023

1 PROYECTO DE PROGRAMACION "Deep Vision in classification tasks"

- Estrategia 1: Entrenar desde cero

Comando !pip freeze para cotillear a Google Colab

[ ]:
```
%%capture
!pip freeze
```

[1]:
```
import os
import shutil
import zipfile
import matplotlib.pyplot as plt
import numpy as np
import cv2
import tensorflow as tf
tf.__version__
```

[1]:
```
'2.12.0'
```

2 CARGANDO EL CONJUNTO DE DATOS DE LA PLATAFORMA DE KAGGLE

[2]:
```
# Nos aseguramos que tenemos instalada la última versión de la API de Kaggle en Colab
!pip install --upgrade --force-reinstall --no-deps kaggle
```

```
Collecting kaggle
  Downloading kaggle-1.5.15.tar.gz (77 kB)
                             77.9/77.9 kB
3.4 MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Building wheels for collected packages: kaggle
  Building wheel for kaggle (setup.py) … done
  Created wheel for kaggle: filename=kaggle-1.5.15-py3-none-any.whl size=99605
sha256=add768dadf18d2c7c042f1d819e8404fd7fc159205cdee9c1eac3b1b51bba4ee
```

```
  Stored in directory: /root/.cache/pip/wheels/46/0f/33/40c049c224ee941c2b3a7abb
858fc34d93e827f9a833d40f09
Successfully built kaggle
Installing collected packages: kaggle
  Attempting uninstall: kaggle
    Found existing installation: kaggle 1.5.15
    Uninstalling kaggle-1.5.15:
      Successfully uninstalled kaggle-1.5.15
Successfully installed kaggle-1.5.15
```

[3]:
```
# Seleccionar el API Token personal previamente descargado (fichero kaggle.json)
from google.colab import files
files.upload()
```

```
<IPython.core.display.HTML object>

Saving kaggle.json to kaggle.json
```

[3]:
```
{'kaggle.json':
 b'{"username":"juankyep","key":"6a740588de92c2c3e1bcc7ae29da3c96"}'}
```

[4]:
```
!ls
```

```
kaggle.json  sample_data
```

[5]:
```
# Creamos un directorio en el que copiamos el fichero kaggle.json
!mkdir ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

[6]:
```
# Ya podemos listar los datasets disponibles en kaggle para su descarga
%%capture
!kaggle datasets list
```

[7]:
```
!kaggle datasets download -d tr1gg3rtrash/yoga-posture-dataset
```

```
Downloading yoga-posture-dataset.zip to /content
 99% 444M/447M [00:03<00:00, 127MB/s]
100% 447M/447M [00:03<00:00, 147MB/s]
```

[7]:
```
from google.colab import drive


# Almaceno el modelo en Drive
# Montamos la unidad de Drive
drive.mount('/content/drive') #(X)
# Establezco una ruta absoluta a un directorio existente de mi Google Drive
```

```
BASE_FOLDER = "/content/drive/MyDrive/
 ↪12MBID_Proyecto_Programacion_Colab_Segunda_Convocatoria"
# Creamos un directorio en el que copiaremors el datasets
nombre_directorio = os.path.join(BASE_FOLDER,'my_dataset')

# Ruta absoluta del directorio que quieres crear

BASE_FOLDER_DATA = os.path.abspath(nombre_directorio)

# Verificar si el directorio ya existe
if not os.path.exists(BASE_FOLDER_DATA):
     #Crear el directorio si no existe
    os.mkdir(BASE_FOLDER_DATA)
    print(f"El directorio {nombre_directorio} ha sido creado en␣
 ↪{BASE_FOLDER_DATA}.")
else:
    print(f"El directorio {nombre_directorio} ya existe en {BASE_FOLDER_DATA}.")

    # Creamos un directorio en el que guardaremos todos los modelos entrenados
nombre_directorio_modelo = os.path.join(BASE_FOLDER,'Models')

# Ruta absoluta del directorio que quieres crear
MODELS_DIR = os.path.abspath(os.path.join(BASE_FOLDER,'Models'))

# Verificar si el directorio ya existe
if not os.path.exists(MODELS_DIR):
    # Crear el directorio si no existe
    os.mkdir(MODELS_DIR)
    print(f"El directorio {nombre_directorio_modelo} ha sido creado en␣
 ↪{MODELS_DIR}.")
else:
    print(f"El directorio {nombre_directorio_modelo} ya existe en {MODELS_DIR}.
 ↪")
```

```
Mounted at /content/drive
El directorio /content/drive/MyDrive/12MBID_Proyecto_Programacion_Colab_Segunda_
Convocatoria/my_dataset ya existe en /content/drive/MyDrive/12MBID_Proyecto_Prog
ramacion_Colab_Segunda_Convocatoria/my_dataset.
El directorio /content/drive/MyDrive/12MBID_Proyecto_Programacion_Colab_Segunda_
Convocatoria/Models ya existe en /content/drive/MyDrive/12MBID_Proyecto_Programa
cion_Colab_Segunda_Convocatoria/Models.
```

[8]:
```
# Descomprimimos los datos y los dejamos listos para trabajar
%%capture
!unzip yoga-posture-dataset.zip -d "/content/drive/MyDrive/
 ↪12MBID_Proyecto_Programacion_Colab_Segunda_Convocatoria/my_dataset"
```

3 INSPECCION DEL CONJUNTO DE DATOS

3.1 Lectura de datos tomando como referencia un BASE_FOLDER_DATA

[9]:
```
# Escogiendo y mostrando una imagen al azar del conjunto de my_datatest
idx = np.random.randint(70, 74)
print(idx)
print(BASE_FOLDER_DATA +'/Adho Mukha Svanasana/File' + str(idx) + '.jpeg')
img = cv2.imread(BASE_FOLDER_DATA +'/Adho Mukha Svanasana/File' + str(idx) + '.
 ↪jpeg', cv2.COLOR_BGR2RGB)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
```

```
71
/content/drive/MyDrive/12MBID_Proyecto_Programacion_Colab_Segunda_Convocatoria/m
y_dataset/Adho Mukha Svanasana/File71.jpeg
```

[9]:
```
<matplotlib.image.AxesImage at 0x7d97260c4790>
```

3.2 Exploracion de la estructura del dataset y verificacion de las carpetas y archivos disponibles.

[10]:
```
# The folder names are our Classes
class_names = sorted(os.listdir(BASE_FOLDER_DATA))
class_names.remove('Poses.json')
n_classes = len(class_names)
```

```
print(f"Total Number of Classes : {n_classes}")
print(f"Classes : \n{class_names}")
```

```
Total Number of Classes : 47
Classes :
['Adho Mukha Svanasana', 'Adho Mukha Vrksasana', 'Alanasana', 'Anjaneyasana',
'Ardha Chandrasana', 'Ardha Matsyendrasana', 'Ardha Navasana', 'Ardha Pincha
Mayurasana', 'Ashta Chandrasana', 'Baddha Konasana', 'Bakasana', 'Balasana',
'Bitilasana', 'Camatkarasana', 'Dhanurasana', 'Eka Pada Rajakapotasana',
'Garudasana', 'Halasana', 'Hanumanasana', 'Malasana', 'Marjaryasana',
'Navasana', 'Padmasana', 'Parsva Virabhadrasana', 'Parsvottanasana',
'Paschimottanasana', 'Phalakasana', 'Pincha Mayurasana', 'Salamba Bhujangasana',
'Salamba Sarvangasana', 'Setu Bandha Sarvangasana', 'Sivasana', 'Supta
Kapotasana', 'Trikonasana', 'Upavistha Konasana', 'Urdhva Dhanurasana', 'Urdhva
Mukha Svsnssana', 'Ustrasana', 'Utkatasana', 'Uttanasana', 'Utthita Hasta
Padangusthasana', 'Utthita Parsvakonasana', 'Vasisthasana', 'Virabhadrasana
One', 'Virabhadrasana Three', 'Virabhadrasana Two', 'Vrksasana']
```

[11]:
```
# Before loading the images, we need to check the class distribution.
# Antes de cargar las imagenes, necesitamos revisar la cantidad de imagenes␣
 ↪distribuida por clase.
class_dis = [len(os.listdir(BASE_FOLDER_DATA + f"/{name}")) for name in␣
 ↪class_names]
print(class_dis)
```

```
[74, 65, 18, 71, 59, 96, 13, 54, 12, 81, 84, 79, 94, 62, 54, 53, 85, 71, 41, 73,
56, 18, 77, 14, 43, 62, 66, 43, 62, 73, 66, 20, 13, 23, 17, 74, 69, 96, 81, 71,
64, 69, 80, 64, 69, 61, 68]
```

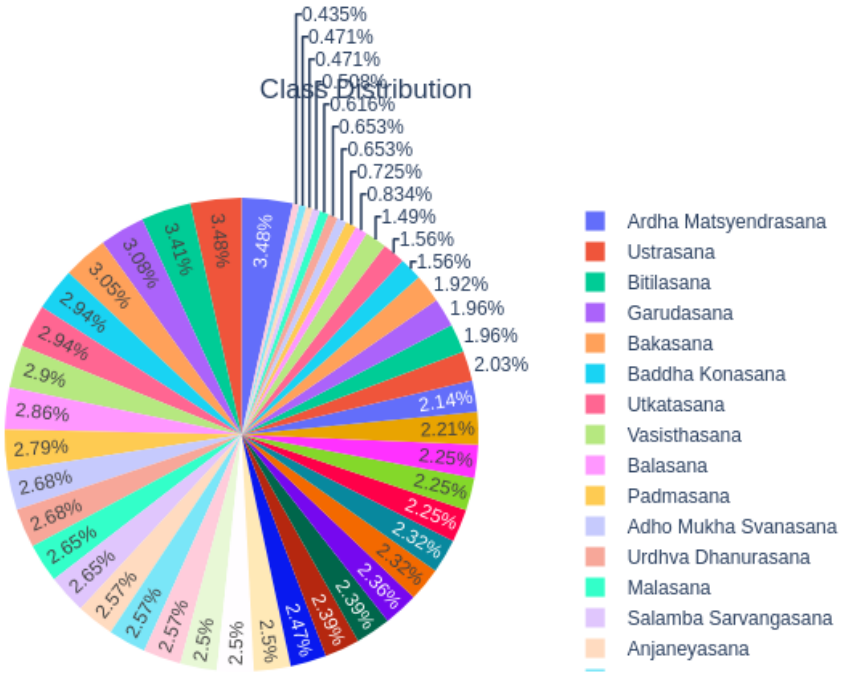
3.3 Observacion de la distribución de imágenes por categoría para comprender el desequilibrio en los datos.

[12]:
```
# Let's understand more by visualizing this class distribution.
# Entenderemos mejor por visualilzacion de las clases distribuidas
!pip install -U kaleido
import plotly.express as px
import plotly.io as pio
from IPython.display import Image

# Generar el gráfico
fig = px.pie(
    names=class_names,
    values=class_dis,
    title="Class Distribution"
)
fig.update_layout({'title': {'x': 0.5}})
```

```
# Guardar el gráfico como una imagen
pio.orca.config.default_format = "png"
pio.write_image(fig, 'grafico.png')

# Mostrar la imagen en el notebook
Image('grafico.png')
```

```
Collecting kaleido
  Downloading kaleido-0.2.1-py2.py3-none-manylinux1_x86_64.whl (79.9 MB)
                                    79.9/79.9 MB
20.0 MB/s eta 0:00:00
Installing collected packages: kaleido
Successfully installed kaleido-0.2.1
```

[12]:

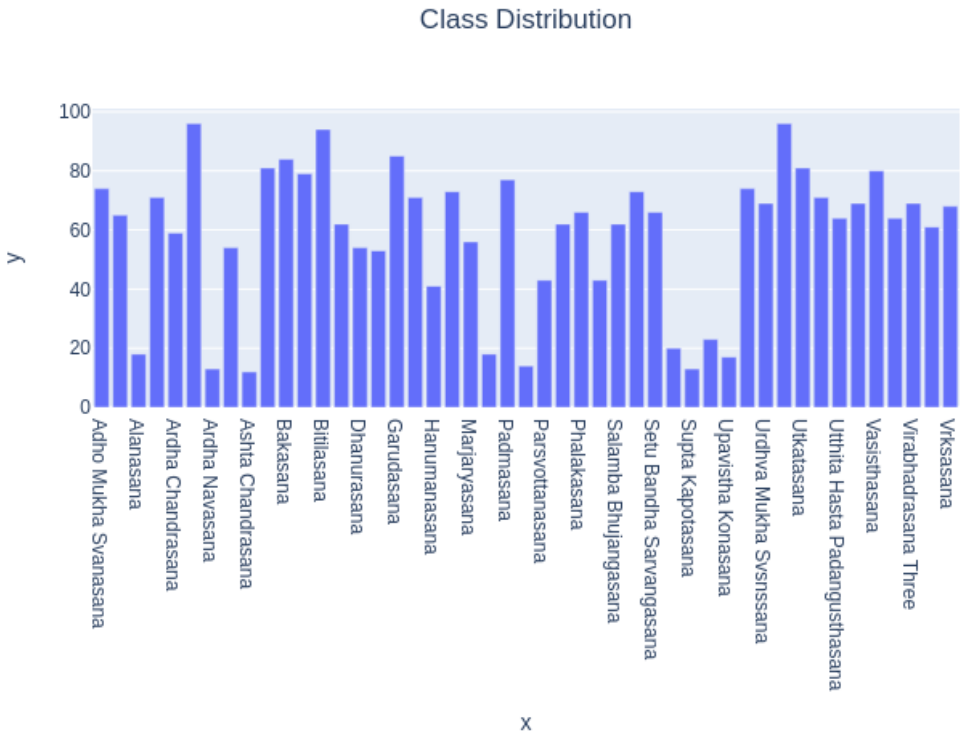


[13]:

```
# gráfico de barras de distribucion de imagenes por clases
fig = px.bar(
    x=class_names,
    y=class_dis,
    title="Class Distribution",
)
fig.update_layout({'title':{
```

```
    'x':0.5
}})
# Guardar el gráfico como una imagen
pio.orca.config.default_format = "png"
pio.write_image(fig, 'grafico1.png')

# Mostrar la imagen en el notebook
Image('grafico1.png')
```

[13]:



4 ACONDICIONAMIENTO DEL CONJUNTO DE DATOS

4.1 Equilibrado de Imagenes por Clase

La distribución de clases es buena, pero algunas clases tienen un número muy bajo de imágenes. Esto afectará nuestro modelo final porque las predicciones de estas clases no serán tan precisas.

4.1.1 Generacion de Dataframe

[14]:
```
from pathlib import Path
import pandas as pd

image_dir = Path(BASE_FOLDER_DATA)
```

```
# visualizamos el dataframe con los campos filepaths y labels
filepaths = list(image_dir.glob(r'**/*.jpg')) + list(image_dir.glob(r'**/*.
 ↪png')) \
+ list(image_dir.glob(r'**/*.jpeg')) + list(image_dir.glob(r'**/*.gif'))

labels = list(map(lambda x: os.path.split(os.path.split(x)[0])[1], filepaths))

filepaths = pd.Series(filepaths, name='Filepath').astype(str)
labels = pd.Series(labels, name='Label')

# Concatenate filepaths and labels
image_df = pd.concat([filepaths, labels], axis=1)

print(image_df.shape)
image_df
```

```
(2758, 2)
```

[14]:

	Filepath	Label
0	/content/drive/MyDrive/12MBID_Proyecto_Program…	Adho Mukha Svanasana
1	/content/drive/MyDrive/12MBID_Proyecto_Program…	Ardha Navasana
2	/content/drive/MyDrive/12MBID_Proyecto_Program…	Baddha Konasana
3	/content/drive/MyDrive/12MBID_Proyecto_Program…	Baddha Konasana
4	/content/drive/MyDrive/12MBID_Proyecto_Program…	Baddha Konasana
…	…	…
2753	/content/drive/MyDrive/12MBID_Proyecto_Program…	Vrksasana
2754	/content/drive/MyDrive/12MBID_Proyecto_Program…	Vrksasana
2755	/content/drive/MyDrive/12MBID_Proyecto_Program…	Vrksasana
2756	/content/drive/MyDrive/12MBID_Proyecto_Program…	Virabhadrasana One
2757	/content/drive/MyDrive/12MBID_Proyecto_Program…	Virabhadrasana Two

[2758 rows x 2 columns]

Verificamos la existencia de archivos gif y convertirlo en jpeg

[32]:
```
img_serie = pd.Series(image_df['Filepath'])
img_gif = img_serie[img_serie.str.contains('.gif')]
print(img_gif)
```

```
2756    /content/drive/MyDrive/12MBID_Proyecto_Program…
2757    /content/drive/MyDrive/12MBID_Proyecto_Program…
Name: Filepath, dtype: object
```

[38]:
```
import os
from PIL import Image

def convert_gif_to_jpg(df):
```

```
    for i, row in df.iterrows():
        filepath = row['Filepath']
        _, ext = os.path.splitext(filepath)
        if ext == ".gif" and os.path.exists(filepath):
            im = Image.open(filepath)
            im = im.convert("RGBA")
            im = im.convert("RGB")
            img_path = os.path.splitext(filepath)[0] + '.jpg'
            im.save(img_path, 'JPEG')
            img = cv2.imread(img_path)

            # Eliminar el archivo GIF original
            os.remove(filepath)
            print(f"El archivo {filepath} ha sido eliminado después de ser␣
 ↪reemplazado por {img_path}")
            #plt.imshow(img)

# Llamar a la función pasando el DataFrame con los Filepath
convert_gif_to_jpg(image_df)
```

Verificamos que cargue correctamente la imagen y su etiqueta del dataframe

[18]:
```
# Verificamos que cargue imagen aleatoriamente
random_index = np.random.randint(0, len(image_df), 16)
fig, axes = plt.subplots(nrows = 4, ncols = 4, figsize = (10,10), subplot_kw =␣
 ↪{'xticks':[], 'yticks':[]})

for i, ax in enumerate(axes.flat):
    #print(i, ax)
    ax.imshow(plt.imread(image_df.Filepath[random_index[i]]))
    ax.set_title(image_df.Label[random_index[i]])
plt.tight_layout()
plt.show()
```

4.1.2 Aumento de Imagenes Equilibrado por Clase

El dataframe `image_df` presenta un desequilibrio significativo, ya que algunas clases contienen hasta 98 imágenes, mientras que una sola clase tiene solo 12 imágenes. Con el objetivo de abordar este desequilibrio, se procederá a aumentar el número de filas en `image_df` mediante la generación de imágenes adicionales. El objetivo es alcanzar un total de 100 muestras en cada clase.

Creacion de directorios de train, valid, y test

[19]:
```
# Creamos un directorio en el que copiaremors el datasets
nombre_directorio_clases = os.path.join(BASE_FOLDER, "dataset")

# Ruta absoluta del directorio que quieres crear
DATASET_DIR = os.path.abspath(nombre_directorio_clases)
```

```
# Verificar si el directorio ya existe
if not os.path.exists(DATASET_DIR):
    # Crear el directorio si no existe
    os.mkdir(DATASET_DIR)
    print(f"El directorio {nombre_directorio_clases} ha sido creado en␣
 ↪{DATASET_DIR}.")
else:
    print(f"El directorio {nombre_directorio_clases} ya existe en {DATASET_DIR}.
 ↪")

test_dir = os.path.join(DATASET_DIR,'test')
train_dir = os.path.join(DATASET_DIR,'train')
valid_dir = os.path.join(DATASET_DIR,'valid')

# Verificar si el directorio de test y train ya existe
if not os.path.exists(train_dir):
    # Crear el directorio si no existe
    os.mkdir(train_dir)
    print(f"El directorio 'train' ha sido creado en {DATASET_DIR}.")
else:
    print(f"El directorio 'train' ya existe en {DATASET_DIR}.")


# Verificar si el directorio de test y train ya existe
if not os.path.exists(test_dir):
    # Crear el directorio si no existe
    os.mkdir(test_dir)
    print(f"El directorio 'test' ha sido creado en {DATASET_DIR}.")
else:
    print(f"El directorio 'test' ya existe en {DATASET_DIR}.")

# Verificar si el directorio de test y train ya existe
if not os.path.exists(valid_dir):
    # Crear el directorio si no existe
    os.mkdir(valid_dir)
    print(f"El directorio 'valid' ha sido creado en {DATASET_DIR}.")
else:
    print(f"El directorio 'valid' ya existe en {DATASET_DIR}.")
```

```
El directorio /content/drive/MyDrive/12MBID_Proyecto_Programacion_Colab_Segunda_
Convocatoria/dataset ha sido creado en /content/drive/MyDrive/12MBID_Proyecto_Pr
ogramacion_Colab_Segunda_Convocatoria/dataset.
El directorio 'train' ha sido creado en /content/drive/MyDrive/12MBID_Proyecto_P
rogramacion_Colab_Segunda_Convocatoria/dataset.
El directorio 'test' ha sido creado en /content/drive/MyDrive/12MBID_Proyecto_Pr
ogramacion_Colab_Segunda_Convocatoria/dataset.
```

```
El directorio 'valid' ha sido creado en /content/drive/MyDrive/12MBID_Proyecto_P
rogramacion_Colab_Segunda_Convocatoria/dataset.
```

Division de dataframe

[20]:
```python
from sklearn.model_selection import train_test_split


# Realizar una separación estratificada de los datos en train y test
train_df, test_df = train_test_split(image_df, test_size=0.15,␣
 ↪stratify=image_df["Label"], random_state=42)

train_df, valid_df  = train_test_split(train_df, test_size=0.15,␣
 ↪stratify=train_df["Label"], random_state=42)

#print(y_test_df)
# Mostrar la distribución de clases en el conjunto de datos completo
print("Distribución de clases en el conjunto de datos completo:")
print("Total: ", len(image_df))
print(image_df["Label"].value_counts())

unique_train, counts_train = np.unique(train_df["Label"], return_counts=True)
# Mostrar la distribución de clases en el conjunto de datos de entrenamiento
print("\nDistribución de clases en el conjunto de datos de entrenamiento con:
 ↪",len(unique_train), "clases")
print("Total de muestras:", len(train_df["Label"]), "imagenes")
print(train_df["Label"].value_counts())

unique_test, counts_test = np.unique(test_df["Label"], return_counts=True)
# Mostrar la distribución de clases en el conjunto de datos de prueba
print("\nDistribución de clases en el conjunto de datos de prueba con:
 ↪",len(unique_test), "clases")
print("Total de muestras:", len(test_df["Label"]), "imagenes")
print(test_df["Label"].value_counts())

unique_valid, counts_valid = np.unique(valid_df["Label"], return_counts=True)
print("\nDistribución de clases en el conjunto de datos de validacion:",␣
 ↪len(unique_valid), "clases")
print("Total de muestras:", len(valid_df["Label"]), "imagenes")
print(valid_df["Label"].value_counts())
```

```
Distribución de clases en el conjunto de datos completo:
Total:  2758
Ustrasana                    96
Ardha Matsyendrasana         96
Bitilasana                   94
Garudasana                   85
Bakasana                     84
```

```
Baddha Konasana                 81
Utkatasana                      81
Vasisthasana                    80
Balasana                        79
Padmasana                       77
Adho Mukha Svanasana            74
Urdhva Dhanurasana              74
Salamba Sarvangasana            73
Malasana                        73
Halasana                        71
Uttanasana                      71
Anjaneyasana                    71
Urdhva Mukha Svsnssana          69
Virabhadrasana Three            69
Utthita Parsvakonasana          69
Vrksasana                       68
Phalakasana                     66
Setu Bandha Sarvangasana        66
Adho Mukha Vrksasana            65
Utthita Hasta Padangusthasana   64
Virabhadrasana One              64
Camatkarasana                   62
Salamba Bhujangasana            62
Paschimottanasana               62
Virabhadrasana Two              61
Ardha Chandrasana               59
Marjaryasana                    56
Ardha Pincha Mayurasana         54
Dhanurasana                     54
Eka Pada Rajakapotasana         53
Pincha Mayurasana               43
Parsvottanasana                 43
Hanumanasana                    41
Trikonasana                     23
Sivasana                        20
Alanasana                       18
Navasana                        18
Upavistha Konasana              17
Parsva Virabhadrasana           14
Supta Kapotasana                13
Ardha Navasana                  13
Ashta Chandrasana               12
Name: Label, dtype: int64

Distribución de clases en el conjunto de datos de entrenamiento con: 47 clases
Total de muestras: 1992 imagenes
Ardha Matsyendrasana            70
Ustrasana                       70
```

```
Bitilasana                      68
Garudasana                      61
Bakasana                        60
Baddha Konasana                 59
Utkatasana                      59
Vasisthasana                    58
Balasana                        57
Padmasana                       55
Urdhva Dhanurasana              54
Salamba Sarvangasana            53
Malasana                        53
Adho Mukha Svanasana            53
Anjaneyasana                    51
Uttanasana                      51
Halasana                        51
Virabhadrasana Three            50
Utthita Parsvakonasana          50
Urdhva Mukha Svsnssana          50
Vrksasana                       49
Setu Bandha Sarvangasana        48
Phalakasana                     48
Adho Mukha Vrksasana            47
Utthita Hasta Padangusthasana   46
Virabhadrasana One              46
Salamba Bhujangasana            45
Paschimottanasana               45
Camatkarasana                   45
Virabhadrasana Two              44
Ardha Chandrasana               42
Marjaryasana                    41
Dhanurasana                     39
Ardha Pincha Mayurasana         39
Eka Pada Rajakapotasana         38
Pincha Mayurasana               31
Parsvottanasana                 31
Hanumanasana                    30
Trikonasana                     17
Sivasana                        14
Alanasana                       13
Navasana                        13
Upavistha Konasana              12
Parsva Virabhadrasana           10
Supta Kapotasana                 9
Ardha Navasana                   9
Ashta Chandrasana                8
Name: Label, dtype: int64

Distribución de clases en el conjunto de datos de prueba con: 47 clases
```

Total de muestras: 414 imagenes

Ardha Matsyendrasana	14
Ustrasana	14
Bitilasana	14
Garudasana	13
Bakasana	13
Balasana	12
Utkatasana	12
Baddha Konasana	12
Padmasana	12
Vasisthasana	12
Adho Mukha Svanasana	11
Uttanasana	11
Urdhva Dhanurasana	11
Halasana	11
Anjaneyasana	11
Salamba Sarvangasana	11
Malasana	11
Virabhadrasana One	10
Vrksasana	10
Phalakasana	10
Urdhva Mukha Svsnssana	10
Utthita Parsvakonasana	10
Virabhadrasana Three	10
Setu Bandha Sarvangasana	10
Adho Mukha Vrksasana	10
Utthita Hasta Padangusthasana	10
Virabhadrasana Two	9
Camatkarasana	9
Ardha Chandrasana	9
Salamba Bhujangasana	9
Paschimottanasana	9
Marjaryasana	8
Dhanurasana	8
Eka Pada Rajakapotasana	8
Ardha Pincha Mayurasana	8
Pincha Mayurasana	7
Parsvottanasana	6
Hanumanasana	6
Navasana	3
Upavistha Konasana	3
Sivasana	3
Trikonasana	3
Alanasana	3
Ashta Chandrasana	2
Ardha Navasana	2
Supta Kapotasana	2
Parsva Virabhadrasana	2

```
Name: Label, dtype: int64

Distribución de clases en el conjunto de datos de validacion: 47 clases
Total de muestras: 352 imagenes
Ardha Matsyendrasana             12
Ustrasana                        12
Bitilasana                       12
Bakasana                         11
Garudasana                       11
Baddha Konasana                  10
Balasana                         10
Adho Mukha Svanasana             10
Padmasana                        10
Utkatasana                       10
Vasisthasana                     10
Malasana                          9
Salamba Sarvangasana              9
Vrksasana                         9
Utthita Parsvakonasana            9
Virabhadrasana Three              9
Halasana                          9
Anjaneyasana                      9
Urdhva Mukha Svsnssana            9
Uttanasana                        9
Urdhva Dhanurasana                9
Camatkarasana                     8
Virabhadrasana One                8
Salamba Bhujangasana              8
Setu Bandha Sarvangasana          8
Paschimottanasana                 8
Virabhadrasana Two                8
Utthita Hasta Padangusthasana     8
Adho Mukha Vrksasana              8
Ardha Chandrasana                 8
Phalakasana                       8
Ardha Pincha Mayurasana           7
Marjaryasana                      7
Dhanurasana                       7
Eka Pada Rajakapotasana           7
Parsvottanasana                   6
Hanumanasana                      5
Pincha Mayurasana                 5
Trikonasana                       3
Sivasana                          3
Supta Kapotasana                  2
Upavistha Konasana                2
Ardha Navasana                    2
Ashta Chandrasana                 2
```

```
Parsva Virabhadrasana           2
Alanasana                       2
Navasana                        2
Name: Label, dtype: int64
```

Funcion de Balance de imagenes por clase con aumento de datos

[92]:
```
%%capture
# Funcion de balance de imagenes por clase
# Esta funcion creará un directorio aug donde se almacenara las imagenes␣
 ↪aumentadas por clase


from tensorflow.keras.preprocessing.image import ImageDataGenerator,␣
 ↪array_to_img, img_to_array, load_img

def balance(df, n, working_dir, img_size, target_dir):
    df=df.copy()
    print('Initial length of dataframe is ', len(df))
    aug_dir=os.path.join(working_dir, target_dir)# directorio 'directorio␣
 ↪destino' para almacenar las imagenes aumentadas
    if os.path.isdir(aug_dir):# Empieza con un directorio vacio
        shutil.rmtree(aug_dir)
    os.mkdir(aug_dir)
    for label in df['Label'].unique():
        dir_path=os.path.join(aug_dir,label)
        os.mkdir(dir_path) # Crea directorios de clases dentro del directorio␣
 ↪"aug"
                           #(es un directorio para almacenar imágenes␣
 ↪aumentadas).
    # crea y almacena las imagenes aumentadas
    total=0
    if os.path.basename(target_dir) == 'train_df':
        gen=ImageDataGenerator(horizontal_flip=True,  rotation_range=(-10,10),␣
 ↪width_shift_range=.1,
                                    height_shift_range=.1, zoom_range=(0.1,1.
 ↪2),fill_mode='constant')
    else:
        gen=ImageDataGenerator(horizontal_flip=True, zoom_range=(1.1,1.5))

    groups=df.groupby('Label') # Agrupar por clase
    for label in df['Label'].unique():  # para cada clase
        group=groups.get_group(label)  # Un DataFrame que contiene solo las␣
 ↪filas con la etiqueta especificada.
        sample_count=len(group)   # Determinar cuántas muestras hay en esta␣
 ↪clase.
```

```
        if sample_count< n: # si la clase tiene menos del número objetivo de␣
↪imágenes.
            aug_img_count=0
            delta=n - sample_count  # número de imágenes aumentadas a crear.
            target_dir_label=os.path.join(aug_dir, label)  # definir dónde␣
↪escribir las imágenes
            msg='{0:40s} for class {1:^30s} creating {2:^5s} augmented images'.
↪format(' ', label, str(delta))
            print(msg, '\r', end='') # imprimir sobre la misma linea
            aug_gen=gen.flow_from_dataframe( group,  x_col='Filepath',␣
↪y_col=None, target_size=img_size,
                                            class_mode=None, batch_size=1,␣
↪shuffle=False,
                                            save_to_dir=target_dir_label,␣
↪save_prefix='aug-', color_mode='rgb',
                                            save_format='jpg')
            while aug_img_count<delta:
                images=next(aug_gen)
                aug_img_count += len(images)
            total +=aug_img_count
    print('Total Augmented images created= ', total)
    # Crear aug_df y fusionarla con train_df para crear un conjunto de␣
↪entrenamiento compuesto llamado ndf.
    aug_fpaths=[]
    aug_labels=[]
    classlist=os.listdir(aug_dir)
    for klass in classlist:
        classpath=os.path.join(aug_dir, klass)
        flist=os.listdir(classpath)
        for f in flist:
            fpath=os.path.join(classpath,f)
            aug_fpaths.append(fpath)
            aug_labels.append(klass)
    Fseries=pd.Series(aug_fpaths, name='Filepath')
    Lseries=pd.Series(aug_labels, name='Label')
    aug_df=pd.concat([Fseries, Lseries], axis=1)
    df=pd.concat([df,aug_df], axis=0).reset_index(drop=True)
    print('Length of augmented dataframe is now ', len(df))
    return df

img_size=(224,224) # tamaño de imagenes aumentadas
working_dir=DATASET_DIR # directorio a almacenar las imagenes aumentadas

n=70 # numero de muestras maxima en cada clase
target_dir = train_dir
df = train_df
```

```
train_aug_df=balance(df, n, working_dir, img_size, target_dir)
n=15
target_dir = valid_dir
df = valid_df
valid_aug_df=balance(df, n, working_dir, img_size, target_dir)
n=15
target_dir = test_dir
df = test_df
test_aug_df=balance(df, n, working_dir, img_size, target_dir)
```

Mostrar la distribución de clases en el conjunto de datos completo original mas el aumento

[22]:
```
# Mostrar la distribución de clases en el conjunto de datos completo
print("Distribución de clases en el conjunto de datos completo:")
print("Total: ", len(image_df))
print(image_df["Label"].value_counts())

unique_train, counts_train = np.unique(train_aug_df["Label"],␣
 ↪return_counts=True)
# Mostrar la distribución de clases en el conjunto de datos de entrenamiento
print("\nDistribución de clases en el conjunto de datos de entrenamiento con:
 ↪",len(unique_train), "clases")
print("Total de muestras:", len(train_aug_df["Label"]), "imagenes")
print(train_aug_df["Label"].value_counts())

unique_test, counts_test = np.unique(test_aug_df["Label"], return_counts=True)
# Mostrar la distribución de clases en el conjunto de datos de prueba
print("\nDistribución de clases en el conjunto de datos de prueba con:
 ↪",len(unique_test), "clases")
print("Total de muestras:", len(test_aug_df["Label"]), "imagenes")
print(test_aug_df["Label"].value_counts())

unique_valid, counts_valid = np.unique(valid_aug_df["Label"],␣
 ↪return_counts=True)
print("\nDistribución de clases en el conjunto de datos de validacion:",␣
 ↪len(unique_valid), "clases")
print("Total de muestras:", len(valid_aug_df["Label"]), "imagenes")
print(valid_aug_df["Label"].value_counts())
```

```
Distribución de clases en el conjunto de datos completo:
Total:  2758
Ustrasana                       96
Ardha Matsyendrasana            96
Bitilasana                      94
Garudasana                      85
Bakasana                        84
Baddha Konasana                 81
```

```
Utkatasana                          81
Vasisthasana                        80
Balasana                            79
Padmasana                           77
Adho Mukha Svanasana                74
Urdhva Dhanurasana                  74
Salamba Sarvangasana                73
Malasana                            73
Halasana                            71
Uttanasana                          71
Anjaneyasana                        71
Urdhva Mukha Svsnssana              69
Virabhadrasana Three                69
Utthita Parsvakonasana              69
Vrksasana                           68
Phalakasana                         66
Setu Bandha Sarvangasana            66
Adho Mukha Vrksasana                65
Utthita Hasta Padangusthasana       64
Virabhadrasana One                  64
Camatkarasana                       62
Salamba Bhujangasana                62
Paschimottanasana                   62
Virabhadrasana Two                  61
Ardha Chandrasana                   59
Marjaryasana                        56
Ardha Pincha Mayurasana             54
Dhanurasana                         54
Eka Pada Rajakapotasana             53
Pincha Mayurasana                   43
Parsvottanasana                     43
Hanumanasana                        41
Trikonasana                         23
Sivasana                            20
Alanasana                           18
Navasana                            18
Upavistha Konasana                  17
Parsva Virabhadrasana               14
Supta Kapotasana                    13
Ardha Navasana                      13
Ashta Chandrasana                   12
Name: Label, dtype: int64

Distribución de clases en el conjunto de datos de entrenamiento con: 47 clases
Total de muestras: 3290 imagenes
Camatkarasana                       70
Sivasana                            70
Virabhadrasana Two                  70
```

```
Adho Mukha Svanasana             70
Adho Mukha Vrksasana             70
Vasisthasana                     70
Eka Pada Rajakapotasana          70
Salamba Sarvangasana             70
Virabhadrasana Three             70
Supta Kapotasana                 70
Dhanurasana                      70
Trikonasana                      70
Malasana                         70
Setu Bandha Sarvangasana         70
Balasana                         70
Ardha Navasana                   70
Vrksasana                        70
Anjaneyasana                     70
Parsvottanasana                  70
Ashta Chandrasana                70
Uttanasana                       70
Navasana                         70
Bitilasana                       70
Marjaryasana                     70
Padmasana                        70
Bakasana                         70
Hanumanasana                     70
Urdhva Mukha Svsnssana           70
Alanasana                        70
Baddha Konasana                  70
Virabhadrasana One               70
Ardha Pincha Mayurasana          70
Utthita Parsvakonasana           70
Ardha Chandrasana                70
Phalakasana                      70
Urdhva Dhanurasana               70
Paschimottanasana                70
Utthita Hasta Padangusthasana    70
Ardha Matsyendrasana             70
Halasana                         70
Salamba Bhujangasana             70
Upavistha Konasana               70
Pincha Mayurasana                70
Garudasana                       70
Ustrasana                        70
Utkatasana                       70
Parsva Virabhadrasana            70
Name: Label, dtype: int64

Distribución de clases en el conjunto de datos de prueba con: 47 clases
Total de muestras: 705 imagenes
```

```
Padmasana                       15
Vrksasana                       15
Upavistha Konasana              15
Trikonasana                     15
Marjaryasana                    15
Parsvottanasana                 15
Camatkarasana                   15
Alanasana                       15
Hanumanasana                    15
Ardha Matsyendrasana            15
Sivasana                        15
Urdhva Dhanurasana              15
Salamba Bhujangasana            15
Phalakasana                     15
Urdhva Mukha Svsnssana          15
Virabhadrasana Three            15
Ardha Pincha Mayurasana         15
Virabhadrasana Two              15
Eka Pada Rajakapotasana         15
Ardha Navasana                  15
Utthita Parsvakonasana          15
Supta Kapotasana                15
Adho Mukha Svanasana            15
Ardha Chandrasana               15
Paschimottanasana               15
Bitilasana                      15
Virabhadrasana One              15
Vasisthasana                    15
Setu Bandha Sarvangasana        15
Adho Mukha Vrksasana            15
Bakasana                        15
Dhanurasana                     15
Garudasana                      15
Baddha Konasana                 15
Pincha Mayurasana               15
Utkatasana                      15
Navasana                        15
Uttanasana                      15
Utthita Hasta Padangusthasana   15
Halasana                        15
Anjaneyasana                    15
Salamba Sarvangasana            15
Balasana                        15
Ashta Chandrasana               15
Malasana                        15
Ustrasana                       15
Parsva Virabhadrasana           15
Name: Label, dtype: int64
```

Distribución de clases en el conjunto de datos de validacion: 47 clases
Total de muestras: 705 imagenes

Dhanurasana	15
Virabhadrasana Three	15
Ardha Matsyendrasana	15
Parsvottanasana	15
Vrksasana	15
Salamba Sarvangasana	15
Adho Mukha Svanasana	15
Balasana	15
Trikonasana	15
Hanumanasana	15
Baddha Konasana	15
Salamba Bhujangasana	15
Utthita Parsvakonasana	15
Ardha Pincha Mayurasana	15
Phalakasana	15
Vasisthasana	15
Utkatasana	15
Adho Mukha Vrksasana	15
Marjaryasana	15
Ashta Chandrasana	15
Parsva Virabhadrasana	15
Alanasana	15
Camatkarasana	15
Paschimottanasana	15
Supta Kapotasana	15
Sivasana	15
Anjaneyasana	15
Urdhva Mukha Svsnssana	15
Virabhadrasana One	15
Uttanasana	15
Bakasana	15
Setu Bandha Sarvangasana	15
Pincha Mayurasana	15
Ardha Navasana	15
Padmasana	15
Urdhva Dhanurasana	15
Virabhadrasana Two	15
Bitilasana	15
Garudasana	15
Halasana	15
Malasana	15
Upavistha Konasana	15
Eka Pada Rajakapotasana	15
Ardha Chandrasana	15
Ustrasana	15

```
Utthita Hasta Padangusthasana    15
Navasana                         15
Name: Label, dtype: int64
```

4.2 Juntando imagenes divididas y aumentadas en directorios clases

[93]:
```
%%capture
import cv2
import os

def df_to_dir(df, target_dir):
    sum_img = 0
    for klass in sorted(df['Label'].unique()):
        classpath = os.path.join(target_dir, klass)
        j = 0
        for i, f in enumerate(df['Filepath']):
            if klass == list(df['Label'])[i] and os.path.exists(f):
                img = cv2.imread(f)
                cv2.imwrite(os.path.join(classpath, os.path.basename(f)), img)
                j += 1
        sum_img += j
        print(f'{j} imágenes en {klass}')

    print(f'Total Copiados: {sum_img} en␣
 ↪{target_dir}============================== ')

df = train_df
target_dir = train_dir
df_to_dir(df, target_dir)

df = valid_df
target_dir = valid_dir
df_to_dir(df, target_dir)

df = test_df
target_dir = test_dir
df_to_dir(df, target_dir)
```

5 Visualizacion de las imagenes equilibradas estratificado por clase

Dividiremos en 15% para validación estratificado por clase y 15% para test estratificado por clase. La presencia de un desequilibrio en la cantidad de datos por clase o categoría puede tener un impacto negativo en el rendimiento del modelo y en la precisión de las predicciones. Para abordar esta situación, se realiza una división estratificada de los datos en los conjuntos de test y validación. Esta división asegura que cada clase tenga una representación adecuada en ambos conjuntos, evitando así posibles sesgos y permitiendo una evaluación más justa y precisa del modelo. Al aplicar esta estrategia, se busca mitigar los efectos negativos que podrían surgir debido a la falta de datos en

algunas clases, promoviendo un rendimiento más equilibrado y confiable del modelo en general.

[45]:
```
# gráfico de barras de distribucion de imagenes por clases de entrenamiento
unique_train, counts_train = np.unique(train_aug_df["Label"],␣
 ↪return_counts=True)

fig = px.bar(
    x=unique_train,
    y=counts_train,
    labels={"x": "Clases", "y": "Frecuencia"},
    title=f"Class Distribution - Train (Total Images:␣
 ↪{len(train_aug_df['Label'])})",
)
fig.update_layout({'title':{
    'x':0.5
}})
fig.show()

# gráfico de barras de distribucion de imagenes por clases de test
fig = px.bar(
    x=unique_test,
    y=counts_test,
    labels={"x": "Clases", "y": "Frecuencia"},
    title=f"Class Distribution - Test (Total Images:␣
 ↪{len(valid_aug_df['Label'])})",
)
fig.update_layout({'title':{
    'x':0.5
}})
fig.show()

# gráfico de barras de distribucion de imagenes por clases de validacion
fig = px.bar(
    x=unique_valid,
    y=counts_valid,
    labels={"x": "Clases", "y": "Frecuencia"},
    title=f"Class Distribution - Validation (Total Images:␣
 ↪{len(test_aug_df['Label'])})",
)
fig.update_layout({'title':{
    'x':0.5
}})
fig.show()
```

5.1 Preprocesamiento de imágenes:

5.2 Generacion de datos en tensores para el entrenamiento

Para procesar las imágenes en nuestro modelo, primero debemos leer y cargarlas como tensores. Para este propósito, creamos una función especializada que se encarga de este proceso. Esta función no solo carga las imágenes, sino que también realiza varias transformaciones importantes.

En primer lugar, normaliza los valores de los píxeles para que estén en el rango de 0 a 1, lo cual es fundamental para el entrenamiento efectivo del modelo. Esto asegura que los datos estén en una escala consistente y facilita el cálculo de gradientes durante el proceso de optimización.

Además, la función redimensiona las imágenes al tamaño de 224x224 píxeles. Esta dimensión es comúnmente utilizada en muchos modelos de redes neuronales convolucionales y nos permite asegurar que todas las imágenes tengan la misma dimensión. Esto es crucial para garantizar que el modelo pueda procesar eficientemente todas las muestras de imágenes sin problemas de dimensiones inconsistentes.

Otro aspecto importante a tener en cuenta es la conversión de las imágenes en arrays, lo cual es necesario para representarlas como tensores en el contexto del modelo de aprendizaje profundo. Al convertir las imágenes en arrays, podemos manipular y operar con ellas de manera más eficiente utilizando las capacidades de los frameworks de aprendizaje automático.

Funcion Generadora de imagenes en tensores:

Esta función que genera dataset de imágenes en tensores desempeña un papel crucial en la preparación de los datos para el entrenamiento de nuestro modelo. Realiza la normalización de valores de píxeles, redimensiona las imágenes y las convierte en arrays, nos proporciona los tensores de entrada (X) y las etiquetas de salida (y) para nuestro modelo de aprendizaje profundo. lo que nos permite llevar a cabo un procesamiento y entrenamiento efectivos en nuestra red neuronal convolucional.

[46]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Parámetros de preprocesamiento
input_shape = (224, 224, 3)  # Tamaño de entrada de las imágenes
batch_size = 32  # Reemplaza 32 con el valor adecuado para tu caso

# Definir generadores de datos para aumentación y preprocesamiento
train_datagen = ImageDataGenerator(rescale=1.0 / 255 ) # Reescalar los valores␣
 ↪de píxeles al rango [0, 1]
valid_datagen = ImageDataGenerator(rescale=1.0 / 255)
test_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Cargar imágenes de entrenamiento, validación y prueba utilizando generadores␣
 ↪de datos
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical'
```

```
)

valid_generator = valid_datagen.flow_from_directory(
    valid_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
```

```
Found 3289 images belonging to 47 classes.
Found 704 images belonging to 47 classes.
Found 705 images belonging to 47 classes.
```

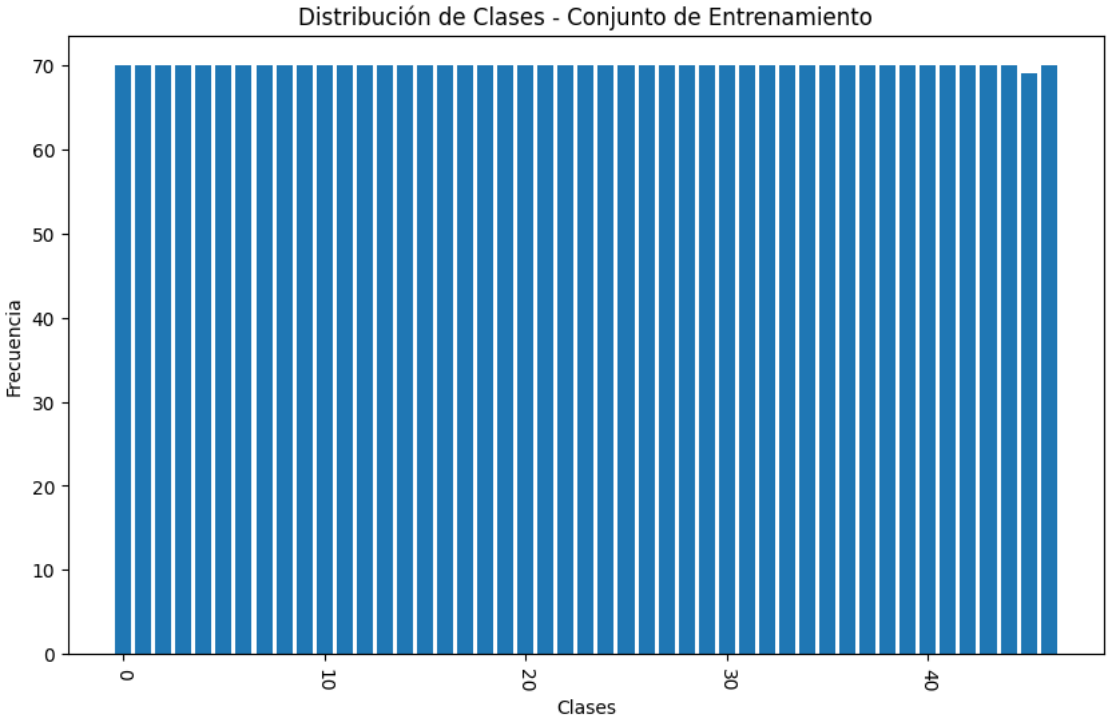
Veficando los tensores generados de entrada (X) y las etiquetas de salida (y) para datos de train, test y valid

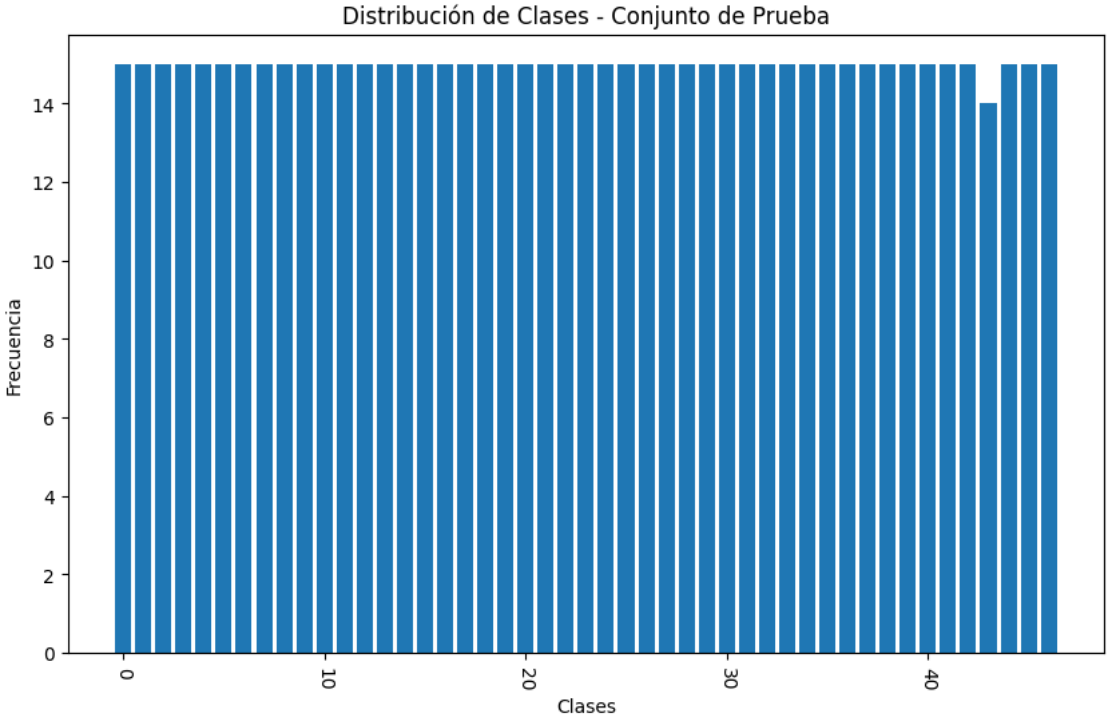
[47]:
```
# Generacion de datos de train
print(f"x_train, y_train: {train_generator.image_shape, train_generator.labels.
 ↪shape}")
# Generacion de datos de test

print(f"x_test, y_test: {valid_generator.image_shape, valid_generator.labels.
 ↪shape}")
# Generacion de datos de valid

print(f"x_valid, y_valid: {test_generator.image_shape, test_generator.labels.
 ↪shape}")

total_images = train_generator.labels.shape[0] + valid_generator.labels.
 ↪shape[0]\
+ test_generator.labels.shape[0]
print(f"Total images: {total_images}")
```

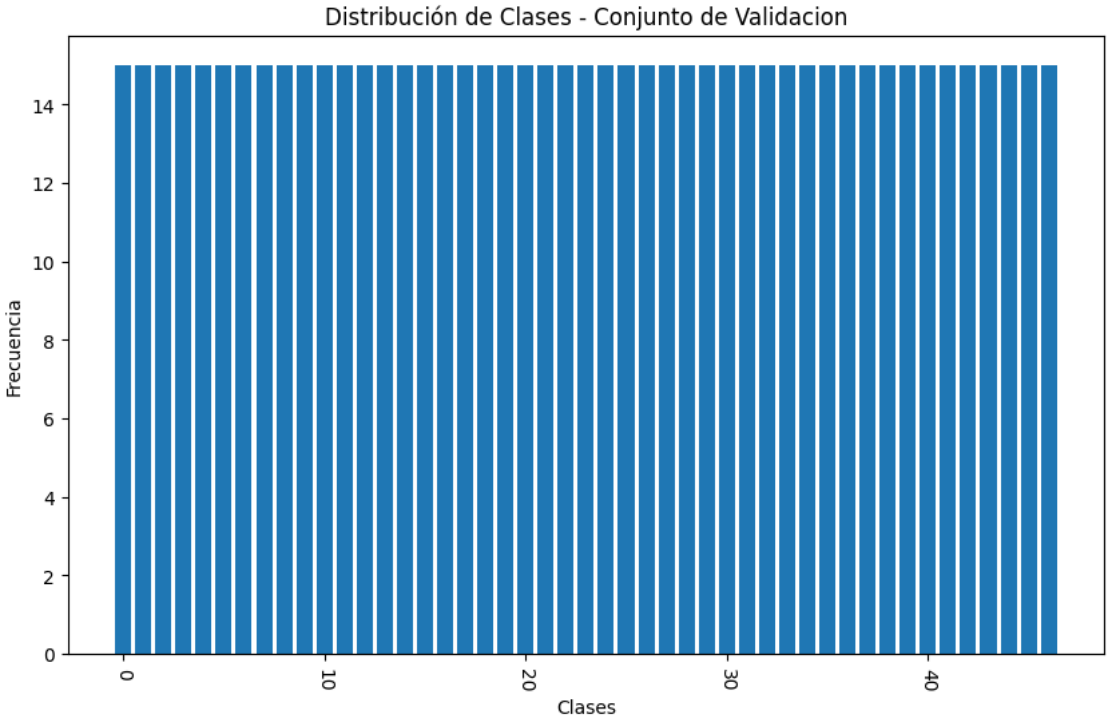
```
x_train, y_train: ((224, 224, 3), (3289,))
x_test, y_test: ((224, 224, 3), (704,))
x_valid, y_valid: ((224, 224, 3), (705,))
Total images: 4698
```

5.3 Revisando el equilibrado de datos (tensores) gráficamente

[48]:
```
import numpy as np
import matplotlib.pyplot as plt

# Obtener la distribución de clases en el conjunto de datos de entrenamiento
unique_train, counts_train = np.unique(train_generator.labels,␣
 ↪return_counts=True)

# Crear el gráfico de barras para el conjunto de entrenamiento
plt.figure(figsize=(10, 6))
plt.bar(unique_train, counts_train)
plt.xlabel('Clases')
plt.ylabel('Frecuencia')
plt.title('Distribución de Clases - Conjunto de Entrenamiento')
plt.xticks(rotation=-90)  # Rotar el texto del eje x 90 grados
plt.show()

# Obtener la distribución de clases en el conjunto de datos de prueba
unique_test, counts_test = np.unique(valid_generator.labels, return_counts=True)

# Crear el gráfico de barras para el conjunto de prueba
plt.figure(figsize=(10, 6))
plt.bar(unique_test, counts_test)
plt.xlabel('Clases')
plt.ylabel('Frecuencia')
plt.title('Distribución de Clases - Conjunto de Prueba')
plt.xticks(rotation=-90)  # Rotar el texto del eje x 90 grados
plt.show()

# Obtener la distribución de clases en el conjunto de datos de validacion
unique_valid, counts_valid = np.unique(test_generator.labels,␣
 ↪return_counts=True)

# Crear el gráfico de barras para el conjunto de validacion
plt.figure(figsize=(10, 6))
plt.bar(unique_valid, counts_valid)
plt.xlabel('Clases')
plt.ylabel('Frecuencia')
plt.title('Distribución de Clases - Conjunto de Validacion')
plt.xticks(rotation=-90)  # Rotar el texto del eje x 90 grados
plt.show()
```


Distribución de Clases - Conjunto de Entrenamiento
Frecuencia
Clases
0
10
20
30
40
50
60
70
0
10
20
30
40


Distribución de Clases - Conjunto de Prueba
Frecuencia
Clases
0
2
4
6
8
10
12
14
0
10
20
30
40



Vista rapida de las etiquetas de salida generadas

[49]:
```
print(f"y_train: {train_generator.labels.shape}")
print(f"Las 5 primeras etiquetas categoricas del dataset train:␣
 ↪{train_generator.labels[:5]}")
print(f"Cuarta etiqueta categorica del dataset train : {train_generator.
 ↪labels[3]}") # categoría de la cuarta etiqueta del dataset train
```

```
y_train: (3289,)
Las 5 primeras etiquetas categoricas del dataset train: [0 0 0 0 0]
Cuarta etiqueta categorica del dataset train : 0
```

6 CONSTRUCCION DE LA TOPOLOGIA DE RED NEURONAL (CNN) Y ENTRENANDOLA

6.1 Aplicacion API funcional con capas mas densas y REGULARIZACION

[52]:
```
import numpy as np
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense,␣
 ↪Dropout, BatchNormalization, MaxPooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

```
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import warnings
from sklearn.exceptions import UndefinedMetricWarning
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import regularizers

tf.keras.backend.clear_session()


########################################
###### Definimos la arquitectura ########
########################################
# BASE MODEL
# Definimos entradas
inputs = Input(shape=input_shape)
epochs = 20
batch_size = 32

train_generator.batch_size = batch_size
train_generator.reset()  # Reiniciar el generador de datos de entrenamiento␣
 ↪antes de la evaluación

valid_generator.batch_size = batch_size
valid_generator.reset()  # Reiniciar el generador de datos de validacion antes␣
 ↪de la evaluación

test_generator.batch_size = batch_size
test_generator.reset()   # Reiniciar el generador de datos de prueba antes de␣
 ↪la evaluación

print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de entrenamiento = {train_generator.batch_size}
Batch size del generador de validacion = {valid_generator.batch_size}
Batch size del generador de prueba = {test_generator.batch_size}''')


# Primer set de capas CONV => RELU => CONV => RELU => POOL
x1 = Conv2D(32, (3, 3), padding="same")(inputs)
x1 = BatchNormalization()(x1)
x1 = Activation("relu")(x1)
#x1 = Conv2D(32, (3, 3), padding="same")(x1)
#x1 = BatchNormalization()(x1)
#x1 = Activation("relu")(x1)
x1 = MaxPooling2D(pool_size=(2, 2))(x1)
#x1 = Dropout(0.05)(x1)
```

```
# Segundo set de capas CONV => RELU => CONV => RELU => POOL
x2 = Conv2D(64, (3, 3), padding="same")(x1)
x2 = BatchNormalization()(x2)
x2 = Activation("relu")(x2)
#x2 = Conv2D(64, (3, 3), padding="same")(x2)
#x2 = BatchNormalization()(x2)
#x2 = Activation("relu")(x2)
x2 = MaxPooling2D(pool_size=(2, 2))(x2)
#x2 = Dropout(0.1)(x2)

# Tercer set de capas CONV => RELU => CONV => RELU => POOL
x3 = Conv2D(256, (3, 3), padding="same")(x2)
x3 = BatchNormalization()(x3)
x3 = Activation("relu")(x3)
#x3 = Conv2D(256, (3, 3), padding="same", activation="relu",␣
 ↪kernel_regularizer=regularizers.l2(0.0001))(x3)
#x3 = BatchNormalization()(x3)
#x3 = Activation("relu")(x3)
x3 = MaxPooling2D(pool_size=(2, 2))(x3)
#x3 = Dropout(0.2)(x3)

# Cuarto bloque de convolución
x4 = Conv2D(256, (3, 3), padding="same", kernel_regularizer=regularizers.l2(0.
 ↪0001))(x3)
x4 = BatchNormalization()(x4)
x4 = Activation("relu")(x4)
x4 = MaxPooling2D(pool_size=(2, 2))(x4)

# TOP MODEL
# Primer (y único) set de capas FC => RELU
xfc = Flatten()(x4)
xfc = Dense(512)(xfc)
xfc = BatchNormalization()(xfc)
xfc = Activation("relu")(xfc)
xfc = Dropout(0.5)(xfc)

xfc = Dense(256)(xfc)
xfc = BatchNormalization()(xfc)
xfc = Activation("relu")(xfc)
xfc = Dropout(0.5)(xfc)

# Clasificador softmax
predictions = Dense(train_generator.num_classes, activation="softmax")(xfc)

# Unimos las entradas y el modelo mediante la función Model con parámetros␣
 ↪inputs y ouputs
```

```
model_cnn = Model(inputs=inputs, outputs=predictions)
model_cnn.summary()

# Compilar modelo
model_cnn.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,␣
 ↪epsilon=1e-08),
    metrics=['accuracy']
)

H = model_cnn.fit(
    train_generator,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=valid_generator,
    steps_per_epoch=len(train_generator),
    validation_steps=len(valid_generator)
)


# Almacenamos el modelo empleando la función mdoel.save de Keras
model_cnn.save(MODELS_DIR+"deepCNN_CIFAR10.h5") #(X)

# Evaluando el modelo de predicción con las imágenes de prueba
print("[INFO]: Evaluando red neuronal...")
predictions = model_cnn.predict(test_generator, steps=len(test_generator))
y_pred = np.argmax(predictions, axis=1)
y_true = test_generator.classes
class_names = list(test_generator.class_indices.keys())
print(classification_report(y_true, y_pred, target_names=class_names,␣
 ↪zero_division=1))

# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, epochs), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, epochs), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, epochs), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, epochs), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show
```

```
Batch size = 32  Epochs = 20
```

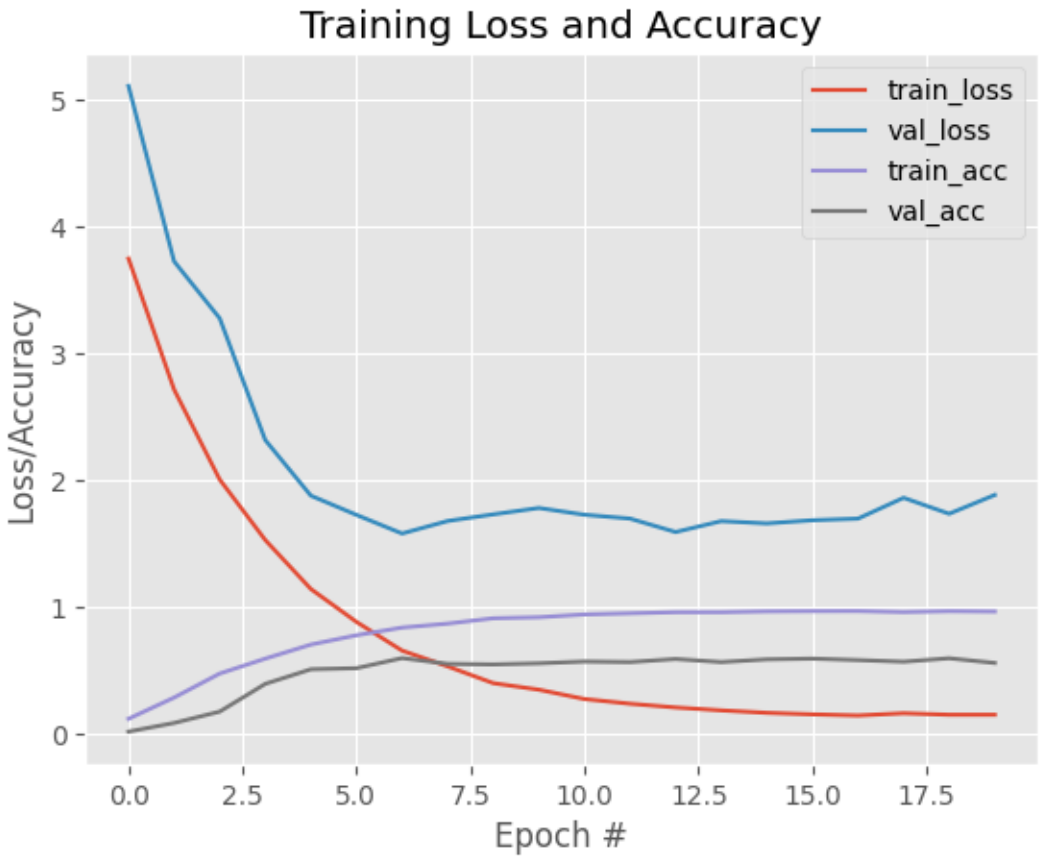
```
Batch size del generador de entrenamiento = 32
Batch size del generador de validacion = 32
Batch size del generador de prueba = 32
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 conv2d (Conv2D)             (None, 224, 224, 32)      896

 batch_normalization (BatchN  (None, 224, 224, 32)     128
 ormalization)

 activation (Activation)     (None, 224, 224, 32)      0

 max_pooling2d (MaxPooling2D  (None, 112, 112, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 112, 112, 64)      18496

 batch_normalization_1 (Batc  (None, 112, 112, 64)     256
 hNormalization)

 activation_1 (Activation)   (None, 112, 112, 64)      0

 max_pooling2d_1 (MaxPooling  (None, 56, 56, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 56, 56, 256)       147712

 batch_normalization_2 (Batc  (None, 56, 56, 256)      1024
 hNormalization)

 activation_2 (Activation)   (None, 56, 56, 256)       0

 max_pooling2d_2 (MaxPooling  (None, 28, 28, 256)      0
 2D)

 conv2d_3 (Conv2D)           (None, 28, 28, 256)       590080

 batch_normalization_3 (Batc  (None, 28, 28, 256)      1024
 hNormalization)

 activation_3 (Activation)   (None, 28, 28, 256)       0

 max_pooling2d_3 (MaxPooling  (None, 14, 14, 256)      0
 2D)
```

```
 flatten (Flatten)           (None, 50176)             0

 dense (Dense)               (None, 512)               25690624

 batch_normalization_4 (Batc  (None, 512)              2048
 hNormalization)

 activation_4 (Activation)   (None, 512)               0

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 256)               131328

 batch_normalization_5 (Batc  (None, 256)              1024
 hNormalization)

 activation_5 (Activation)   (None, 256)               0

 dropout_1 (Dropout)         (None, 256)               0

 dense_2 (Dense)             (None, 47)                12079

=================================================================
Total params: 26,596,719
Trainable params: 26,593,967
Non-trainable params: 2,752
_________________________________________________________________
Epoch 1/20
103/103 [==============================] - 28s 211ms/step - loss: 3.7500 -
accuracy: 0.1222 - val_loss: 5.1104 - val_accuracy: 0.0213
Epoch 2/20
103/103 [==============================] - 22s 211ms/step - loss: 2.7173 -
accuracy: 0.2898 - val_loss: 3.7278 - val_accuracy: 0.0895
Epoch 3/20
103/103 [==============================] - 25s 240ms/step - loss: 2.0085 -
accuracy: 0.4789 - val_loss: 3.2792 - val_accuracy: 0.1790
Epoch 4/20
103/103 [==============================] - 24s 231ms/step - loss: 1.5315 -
accuracy: 0.5965 - val_loss: 2.3192 - val_accuracy: 0.3977
Epoch 5/20
103/103 [==============================] - 22s 213ms/step - loss: 1.1455 -
accuracy: 0.7075 - val_loss: 1.8811 - val_accuracy: 0.5128
Epoch 6/20
103/103 [==============================] - 24s 231ms/step - loss: 0.8861 -
accuracy: 0.7808 - val_loss: 1.7295 - val_accuracy: 0.5213
Epoch 7/20
103/103 [==============================] - 22s 213ms/step - loss: 0.6599 -
```

```
accuracy: 0.8413 - val_loss: 1.5827 - val_accuracy: 0.6009
Epoch 8/20
103/103 [==============================] - 22s 209ms/step - loss: 0.5357 -
accuracy: 0.8726 - val_loss: 1.6814 - val_accuracy: 0.5554
Epoch 9/20
103/103 [==============================] - 24s 236ms/step - loss: 0.4033 -
accuracy: 0.9149 - val_loss: 1.7336 - val_accuracy: 0.5511
Epoch 10/20
103/103 [==============================] - 22s 213ms/step - loss: 0.3519 -
accuracy: 0.9228 - val_loss: 1.7839 - val_accuracy: 0.5597
Epoch 11/20
103/103 [==============================] - 22s 209ms/step - loss: 0.2783 -
accuracy: 0.9450 - val_loss: 1.7307 - val_accuracy: 0.5739
Epoch 12/20
103/103 [==============================] - 22s 210ms/step - loss: 0.2412 -
accuracy: 0.9541 - val_loss: 1.7004 - val_accuracy: 0.5696
Epoch 13/20
103/103 [==============================] - 26s 253ms/step - loss: 0.2118 -
accuracy: 0.9626 - val_loss: 1.5943 - val_accuracy: 0.5938
Epoch 14/20
103/103 [==============================] - 22s 216ms/step - loss: 0.1887 -
accuracy: 0.9629 - val_loss: 1.6803 - val_accuracy: 0.5696
Epoch 15/20
103/103 [==============================] - 22s 214ms/step - loss: 0.1694 -
accuracy: 0.9693 - val_loss: 1.6624 - val_accuracy: 0.5909
Epoch 16/20
103/103 [==============================] - 22s 214ms/step - loss: 0.1575 -
accuracy: 0.9711 - val_loss: 1.6874 - val_accuracy: 0.5966
Epoch 17/20
103/103 [==============================] - 22s 212ms/step - loss: 0.1486 -
accuracy: 0.9717 - val_loss: 1.6998 - val_accuracy: 0.5852
Epoch 18/20
103/103 [==============================] - 23s 217ms/step - loss: 0.1661 -
accuracy: 0.9641 - val_loss: 1.8643 - val_accuracy: 0.5724
Epoch 19/20
103/103 [==============================] - 22s 208ms/step - loss: 0.1544 -
accuracy: 0.9708 - val_loss: 1.7384 - val_accuracy: 0.5994
Epoch 20/20
103/103 [==============================] - 25s 241ms/step - loss: 0.1545 -
accuracy: 0.9681 - val_loss: 1.8868 - val_accuracy: 0.5625
[INFO]: Evaluando red neuronal…
23/23 [==============================] - 4s 181ms/step
                           precision    recall  f1-score   support

         Adho Mukha Svanasana       0.67      0.53      0.59        15
         Adho Mukha Vrksasana       0.22      0.27      0.24        15
                    Alanasana       1.00      0.47      0.64        15
                Anjaneyasana       0.50      0.47      0.48        15
```

Ardha Chandrasana	0.57	0.87	0.68	15
Ardha Matsyendrasana	1.00	0.27	0.42	15
Ardha Navasana	0.36	0.33	0.34	15
Ardha Pincha Mayurasana	0.67	0.53	0.59	15
Ashta Chandrasana	0.00	0.00	0.00	15
Baddha Konasana	0.27	0.40	0.32	15
Bakasana	0.50	0.73	0.59	15
Balasana	1.00	0.33	0.50	15
Bitilasana	0.60	0.80	0.69	15
Camatkarasana	0.82	0.60	0.69	15
Dhanurasana	0.73	0.53	0.62	15
Eka Pada Rajakapotasana	0.53	0.53	0.53	15
Garudasana	0.69	0.73	0.71	15
Halasana	0.73	0.73	0.73	15
Hanumanasana	0.36	0.93	0.52	15
Malasana	0.67	0.53	0.59	15
Marjaryasana	0.53	0.60	0.56	15
Navasana	0.73	0.53	0.62	15
Padmasana	0.50	0.27	0.35	15
Parsva Virabhadrasana	0.00	0.00	0.00	15
Parsvottanasana	1.00	0.20	0.33	15
Paschimottanasana	0.59	0.67	0.62	15
Phalakasana	0.50	0.33	0.40	15
Pincha Mayurasana	1.00	0.27	0.42	15
Salamba Bhujangasana	0.45	0.60	0.51	15
Salamba Sarvangasana	0.23	0.80	0.36	15
Setu Bandha Sarvangasana	0.80	0.80	0.80	15
Sivasana	0.29	1.00	0.45	15
Supta Kapotasana	1.00	0.07	0.12	15
Trikonasana	0.71	0.80	0.75	15
Upavistha Konasana	0.83	0.33	0.48	15
Urdhva Dhanurasana	0.62	0.87	0.72	15
Urdhva Mukha Svsnssana	0.93	0.87	0.90	15
Ustrasana	0.63	0.80	0.71	15
Utkatasana	0.50	0.87	0.63	15
Uttanasana	0.75	0.60	0.67	15
Utthita Hasta Padangusthasana	0.90	0.60	0.72	15
Utthita Parsvakonasana	1.00	0.87	0.93	15
Vasisthasana	0.82	0.60	0.69	15
Virabhadrasana One	0.86	0.40	0.55	15
Virabhadrasana Three	0.91	0.67	0.77	15
Virabhadrasana Two	1.00	0.73	0.85	15
Vrksasana	0.85	0.73	0.79	15
accuracy			0.56	705
macro avg	0.66	0.56	0.56	705
weighted avg	0.66	0.56	0.56	705

[52]: <function matplotlib.pyplot.show(close=None, block=None)>



7 MONITOREO Y EVALUACIÓN DEL MODELO PREDICTIVO POR CADA EXPERIMENTO PARA TOMAR DECISIONES

7.1 80 Registros de metricas e hiperparametros de entrenamientos del Modelo

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

El dataframe muestra los primeros 5 registros, los 5 ultimos y los 10 valores mas altos del entrenamiento, total mostrara 15 registros con los hiperparametros de entrenamiento, modelo con mejor accuracy sera considerado el mejor modelo.

[ ]:
```
import pandas as pd
```

```
# Lee el archivo CSV y crea un DataFrame
df = pd.read_csv('/content/drive/MyDrive/
 ↪12MBID_Proyecto_Programacion_Colab_Segunda_Convocatoria/hiperparametros de␣
 ↪entrenamiento.csv', sep=";")

# Muestra las primeras 5 filas
head_df = df.head(5)

# Muestra las últimas 5 filas
tail_df = df.tail(5)

# Obtiene los 10 valores más altos de la primera columna
top_10_values = df.nlargest(5, df.columns[0])

# Combina las primeras 5 filas y las últimas 5 filas en un solo DataFrame
combined_df = pd.concat([head_df, top_10_values, tail_df])

# Muestra el DataFrame combinado
combined_df.head(15)
```

[ ]:

	precision	recall	f1-core	dropout capa 1	dropout capa 2	\
0	0.48	0.46	0.46	0.10	0.1	
1	0.41	0.38	0.37	0.10	0.1	
2	0.49	0.47	0.46	0.10	0.1	
3	0.47	0.45	0.45	0.10	0.1	
4	0.40	0.36	0.35	0.10	0.1	
53	0.85	0.03	0.00	0.05	0.1	
52	0.81	0.03	0.01	0.05	0.1	
28	0.61	0.12	0.12	0.10	0.1	
27	0.58	0.22	0.22	0.10	0.1	
46	0.54	0.50	0.51	0.05	0.1	
74	0.51	0.48	0.48	0.05	0.1	
75	0.52	0.50	0.50	0.05	0.1	
76	0.51	0.48	0.48	0.05	0.1	
77	0.49	0.48	0.47	0.05	0.1	
78	0.52	0.49	0.49	0.05	0.1	

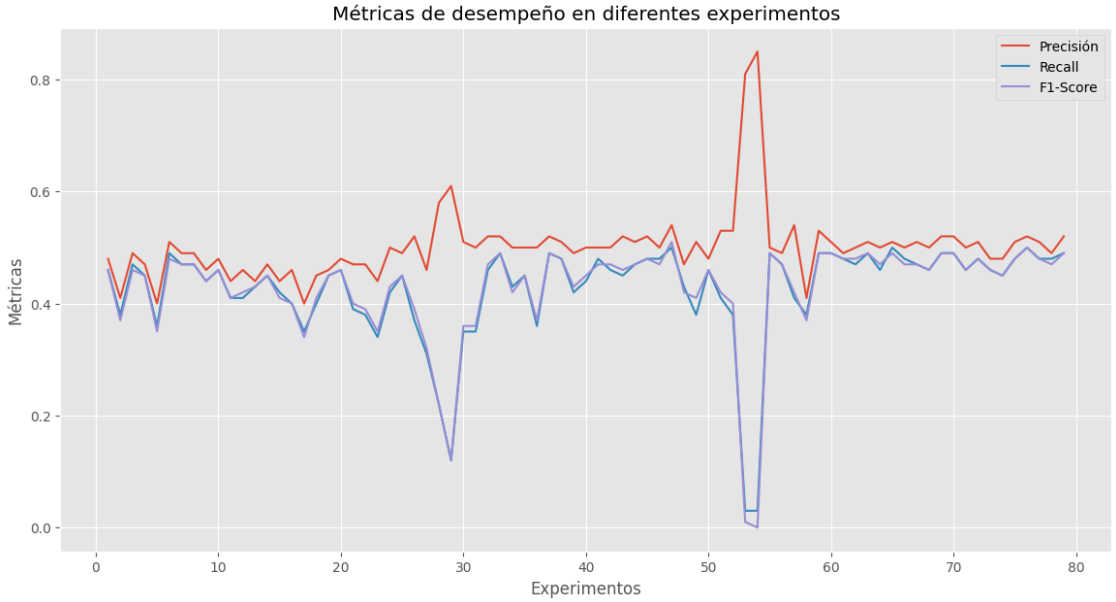
	dropout capa 3	dropout capa 4	batch size	epoch	Regularizacion	\
0	0.20	0.50	64	20	0.0000	
1	0.20	0.30	32	20	0.0000	
2	0.20	0.50	32	20	0.0000	
3	0.20	0.50	16	20	0.0000	
4	0.20	0.50	8	25	0.0000	
53	0.25	0.55	256	30	0.0001	
52	0.15	0.55	256	30	0.0001	
28	0.20	0.50	128	20	0.0010	
27	0.20	0.50	128	20	0.0100	

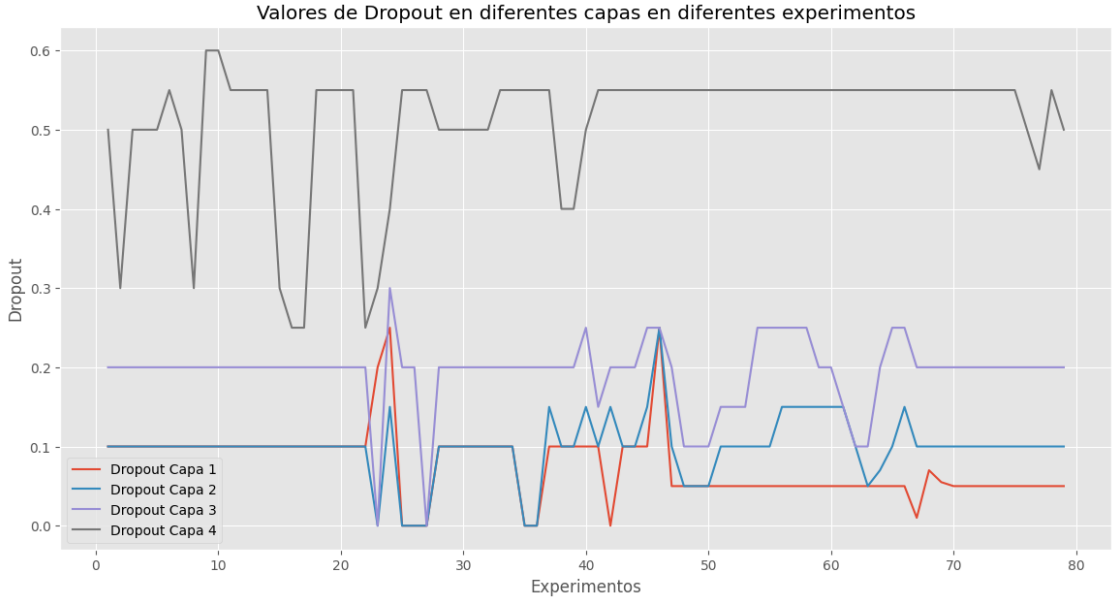
```
46              0.20            0.55          128      30          0.0001
74              0.20            0.55          128      30          0.0001
75              0.20            0.50          128      30          0.0001
76              0.20            0.45          128      30          0.0001
77              0.20            0.55          128      30          0.0001
78              0.20            0.50          128      30          0.0001

    learning rate
0           0.001
1           0.001
2           0.001
3           0.001
4           0.001
53          0.001
52          0.001
28          0.001
27          0.001
46          0.001
74          0.001
75          0.001
76          0.001
77          0.001
78          0.001
```

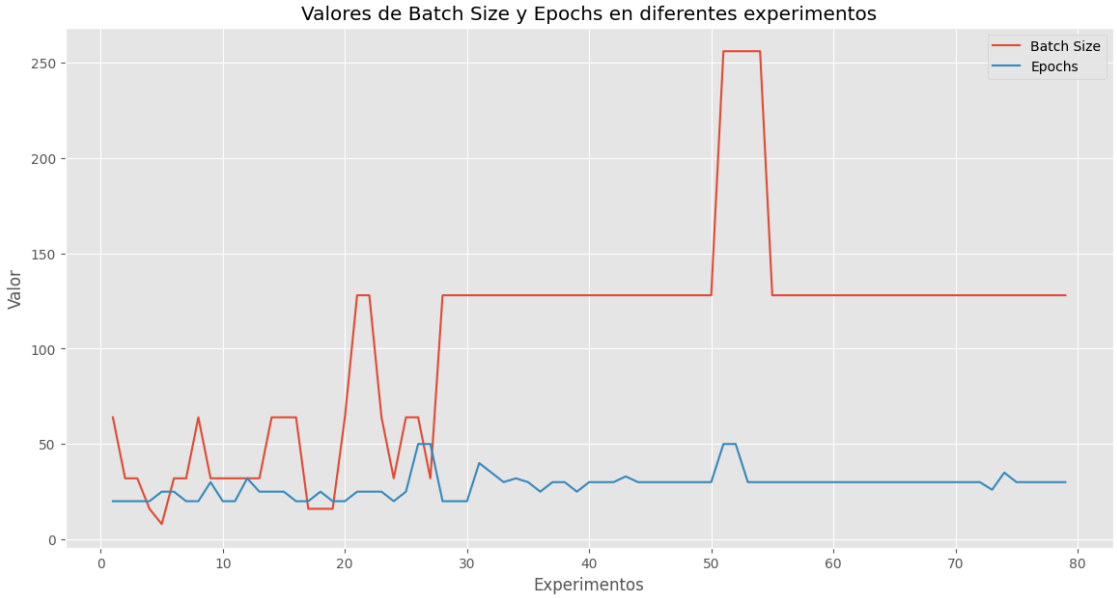
7.2 Inspección gráfica de las métricas e hiperparámetros de los registros de entrenamientos

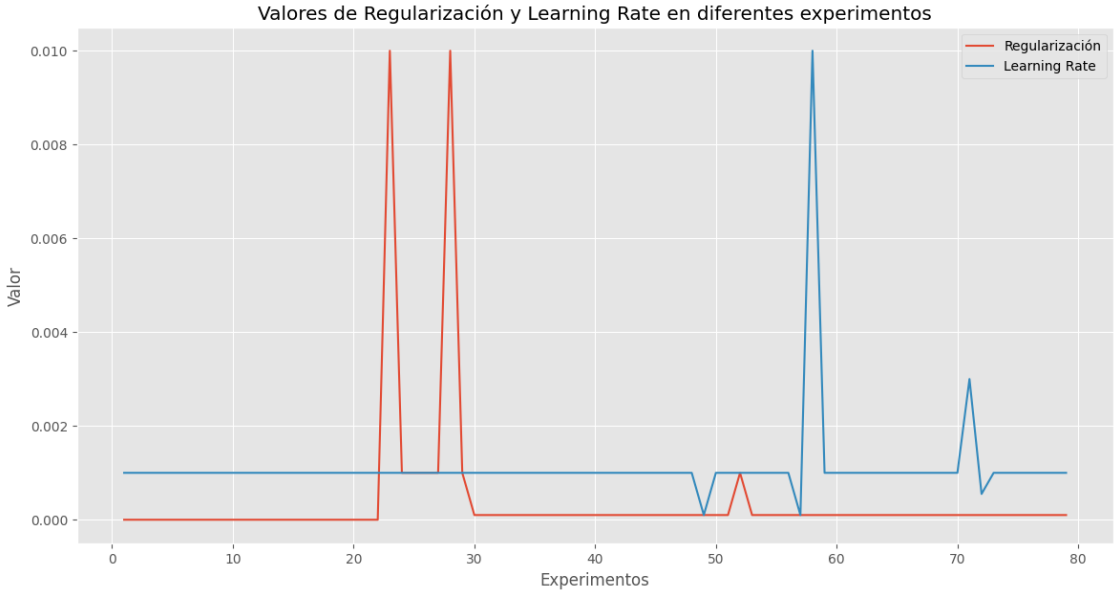
[ ]:
```python
import matplotlib.pyplot as plt
# Leer el archivo CSV especificando el motor de lectura como 'python'
df = pd.read_csv('/content/drive/MyDrive/
 ↪12MBID_Proyecto_Programacion_Colab_Segunda_Convocatoria/hiperparametros de␣
 ↪entrenamiento.csv', sep=';', engine='python')

# Extraer los datos de cada columna como listas
precision = df['precision'].tolist()
recall = df['recall'].tolist()
f1_score = df['f1-core'].tolist()
dropout_layer1 = df['dropout capa 1'].tolist()
dropout_layer2 = df['dropout capa 2'].tolist()
dropout_layer3 = df['dropout capa 3'].tolist()
dropout_layer4 = df['dropout capa 4'].tolist()
batch_size = df['batch size'].tolist()
epochs = df['epoch'].tolist()
regularization = df['Regularizacion'].tolist()
learnig_rate = df['learning rate'].tolist()
```

```
x = list(range(1, 80))  # Número de experimentos

# Gráfico de Precisión, Recall y F1-Score
plt.figure(figsize=(14, 7))
plt.plot(x, precision, label='Precisión')
plt.plot(x, recall, label='Recall')
plt.plot(x, f1_score, label='F1-Score')
plt.xlabel('Experimentos')
plt.ylabel('Métricas')
plt.title('Métricas de desempeño en diferentes experimentos')
plt.legend()
plt.show()

# Gráfico de Dropout en las capas
plt.figure(figsize=(14, 7))
plt.plot(x, dropout_layer1, label='Dropout Capa 1')
plt.plot(x, dropout_layer2, label='Dropout Capa 2')
plt.plot(x, dropout_layer3, label='Dropout Capa 3')
plt.plot(x, dropout_layer4, label='Dropout Capa 4')
plt.xlabel('Experimentos')
plt.ylabel('Dropout')
plt.title('Valores de Dropout en diferentes capas en diferentes experimentos')
plt.legend()
plt.show()

# Gráfico de Batch Size y Epochs
plt.figure(figsize=(14, 7))
plt.plot(x, batch_size, label='Batch Size')
plt.plot(x, epochs, label='Epochs')
plt.xlabel('Experimentos')
plt.ylabel('Valor')
plt.title('Valores de Batch Size y Epochs en diferentes experimentos')
plt.legend()
plt.show()

# Gráfico de Regularización
plt.figure(figsize=(14, 7))
plt.plot(x, regularization, label='Regularización')
plt.plot(x, learnig_rate, label='Learning Rate')
plt.xlabel('Experimentos')
plt.ylabel('Valor')
plt.title('Valores de Regularización y Learning Rate en diferentes␣
 ↪experimentos')
plt.legend()
plt.show()
```


Métricas de desempeño en diferentes experimentos
Precisión
Recall
F1-Score
Métricas
Experimentos
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50
60
70
80


Valores de Dropout en diferentes capas en diferentes experimentos
Dropout Capa 1
Dropout Capa 2
Dropout Capa 3
Dropout Capa 4
Dropout
Experimentos
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
10
20
30
40
50
60
70
80





8 EVALUACION

Con el objetivo de seleccionar la mejor red neuronal más prometedora, se probaron diferentes configuraciones. En un primer enfoque, se probó una topología sin regularización, mientras que en el segundo enfoque se aplicó dropout en la segunda capa densa y en el flatten. Sin embargo la mas prometedora, fue en la tercera topología donde se implementaron capas más densas y se utilizaron más hiperparámetros, como la regularización en la tercera capa junto con dropout en todas las

capas. Se empleó el método API funcional para su construcción.

Para evaluar y concluir sobre los resultados del entrenamiento, se analizaron las métricas de precisión, recall y F1-score, para determinar el mejor modelo. Además.

1. Tamaño de los datos: Inicialmente con 2,760 imágenes, y después de aplicar aumentación sintética equilibrada por clase, aumentamos a 4,698 imágenes. Esto ayudó a abordar el desequilibrio de clases y obtener un conjunto de datos balanceado.

2. Tamaño de imagenes: Se usa unas dimensiones de imagen de entrada de 224x224 píxeles debido a su compatibilidad con modelos preentrenados y la técnica de transfer learning. Muchos modelos de redes neuronales preentrenados, como VGG16, ResNet, Inception, entre otros, están diseñados para aceptar imágenes en un tamaño específico, y 224x224 es uno de los tamaños comúnmente utilizados.

3. Distribución de clases: Los datos se dividen en 47 clases, con diferentes cantidades de imágenes por clase. El equilibrio se logró aumentando el número de imágenes a 100 por clase, lo cual resultó en un total de 4,700 imágenes. Esta estrategia de equilibrio de clases es importante para evitar sesgos hacia las clases dominantes y mejorar la capacidad del modelo para aprender patrones en las clases minoritarias durante el entrenamiento.

4. Preprocesamiento de datos: Aplicamos técnicas de normalización y codificación numerica para las clases. También redimensionamos las etiquetas de salida para que tuvieran la forma adecuada. Estos pasos son esenciales para garantizar que los datos estén en el formato correcto y sean adecuados para el entrenamiento de la red neuronal.

5. Hiperparámetros: Al analizar los hiperparámetros utilizados en cada entrenamiento, se observa que las métricas de precisión, recall y F1-score varían significativamente. Esto nos lleva a experimentar con diferentes valores de hiperparámetros, como el tamaño de lote (batch size), el número de épocas, la regularización y la tasa de aprendizaje (learning rate). Estos parámetros desempeñan un papel fundamental en el rendimiento del modelo y su capacidad para generalizar.

Ahora, al analizar las métricas de entrenamiento obtenidas:

- Precisión: En general, las precisiones obtenidas oscilan entre 0.4 y 0.6. Esto indica que el modelo tiene dificultades para clasificar correctamente algunas de las clases. Una precisión más alta indicaría una mayor capacidad del modelo para clasificar correctamente las imágenes.

- Recall: Los valores de recall también varían entre 0.03 y 0.5. Un recall más alto indicaría que el modelo puede identificar correctamente un mayor número de muestras positivas para cada clase.

- F1-score: Los valores de F1-score están en el rango de 0.01 a 0.51. El F1-score es una métrica que combina precisión y recall, y un valor más alto indica un mejor rendimiento general del modelo.

- Parece haber una tendencia hacia mejores resultados cuando se utiliza un dropout mayor en las capas 3 y 4. Los dropout de las capas 1 y 2 se mantienen constantes en 0.05 y 0.1 respectivamente, mientras que los de las capas 3 y 4 varían principalmente entre 0.2 y 0.55. Esta observación sugiere que un dropout más agresivo en las capas más profundas de la red puede contribuir a mejorar la capacidad de generalización y reducir el sobreajuste.

- El tamaño del lote (batch size) utilizado en los entrenamientos varía principalmente entre 16, 32, 64 y 128. Sin embargo, el que ofrece mejores métricas es el de 128.
- La cantidad de épocas también varía, aunque la mayoría de los entrenamientos se realizan durante 20 a 30 épocas. Las épocas muy altas tienden a dar resultados muy bajos en las métricas, llegando al overfitting. No obstante, el mejor resultado se obtuvo con 30 épocas, donde el modelo logró converger adecuadamente.
- La regularización y la tasa de aprendizaje (learning rate) se mantienen constantes en la mayoría de los entrenamientos, con valores de 0.0001 para la regularización y 0.001 para la tasa de aprendizaje. Al variar estos valores, las métricas disminuyeron significativamente.

En general, Los hiperparámetros utilizados en el entrenamiento del mejor modelo son los siguientes:

-Epochs: 20 -Batch size: 32 -Learning rate: 0.001 -Optimizador: Adam -Función de pérdida: categorical_crossentropy -Métricas de evaluación: precisión (accuracy)

La arquitectura del modelo consiste en una red convolucional con varias capas de convolución, activaciones ReLU, normalización por lotes y capas de agrupación máxima (max pooling). La arquitectura incluye también capas totalmente conectadas, aplicando regularización mediante dropout, y una capa de salida con activación softmax.

El modelo se entrenó utilizando un conjunto de entrenamiento, validación y prueba. Durante el entrenamiento, se realizaron 20 épocas con un tamaño de lote de 32. Se evaluó el rendimiento del modelo en el conjunto de prueba y se generó un informe de clasificación que muestra las métricas de precisión, recall y puntuación F1 para cada clase.

9 CONCLUSION

Las conclusiones del resultado son las siguientes:

El modelo alcanzó una precisión promedio de aproximadamente 0.56 en el conjunto de prueba. Algunas clases obtuvieron resultados destacados, con altos valores de precisión y recall, mientras que otras clases obtuvieron resultados más bajos. La clase “Setu Bandha Sarvangasana” obtuvo la mejor puntuación F1 (0.80), mientras que las clases “Ashta Chandrasana” y “Parsva Virabhadrasana” obtuvieron la puntuación F1 más baja (0.00). Las gráficas de pérdida y precisión durante el entrenamiento muestran una disminución de la pérdida y un aumento de la precisión tanto en el conjunto de entrenamiento como en el de validación, lo que indica que el modelo está aprendiendo adecuadamente. En general, el modelo muestra un rendimiento promedio en la clasificación de las posturas de yoga. Algunas clases son más difíciles de clasificar que otras, lo que podría requerir ajustes en el modelo o una mayor cantidad de datos de entrenamiento para mejorar el rendimiento.

10 TRANSFER LEARNING Y FINE TUNING

10.1 Generador de datos del modelo VGG16

[56]:
```
from tensorflow.keras.applications.vgg16 import preprocess_input
input_shape = (224, 224, 3)

# Definir generadores de datos para aumentación y preprocesamiento
```

```
train_datagen_tl = ImageDataGenerator( preprocessing_function=preprocess_input)
valid_datagen_tl = ImageDataGenerator(preprocessing_function=preprocess_input)
test_datagen_tl = ImageDataGenerator(preprocessing_function=preprocess_input)

# Cargar imágenes de entrenamiento, validación y prueba utilizando generadores␣
 ↪de datos
train_generator_tl = train_datagen_tl.flow_from_directory(
    train_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical'
)

valid_generator_tl = valid_datagen_tl.flow_from_directory(
    valid_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator_tl = test_datagen_tl.flow_from_directory(
    test_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False,
)
```

```
Found 3289 images belonging to 47 classes.
Found 704 images belonging to 47 classes.
Found 705 images belonging to 47 classes.
```

10.2 Entrenando el Modelo VGG16

[57]:
```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D,  Dense,␣
 ↪Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import regularizers


num_classes = 47
epochs = 20
batch_size = 32
```

```
train_generator_tl.batch_size = batch_size
train_generator_tl.reset()  # Reiniciar el generador de datos de entrenamiento␣
 ↪antes de la evaluación

valid_generator_tl.batch_size = batch_size
valid_generator_tl.reset()  # Reiniciar el generador de datos de validacion␣
 ↪antes de la evaluación

print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de entrenamiento = {train_generator_tl.batch_size}
Batch size del generador de validacion = {valid_generator_tl.batch_size}''')

# Cargar la arquitectura VGG16 sin las capas densas (fully connected)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224,␣
 ↪224, 3))

# Congelar las capas convolucionales para que no se actualicen durante el␣
 ↪entrenamiento
for layer in base_model.layers:
    layer.trainable = False

# Agregar nuevas capas densas para la clasificación
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu',kernel_regularizer=regularizers.l2(0.0001))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)

x = Dense(256)(x)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Dropout(0.5)(x)

predictions = Dense(num_classes, activation='softmax')(x)

# Crear el modelo final que incluye tanto las capas de VGG16 como las nuevas␣
 ↪capas densas
model = Model(inputs=base_model.input, outputs=predictions)

# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])

# Entrenar el modelo con tus datos
```

```
history = model.fit(train_generator_tl, epochs=epochs,␣
 ↪validation_data=valid_generator_tl)

# Almacenamos el modelo empleando la función model.save de Keras
model.save(MODELS_DIR + "TransferLearning_Yoga_VGG.h5") #(X)

# Evaluando el modelo de predicción con las imágenes de prueba
print("[INFO]: Evaluando red neuronal...")
test_generator_tl.batch_size = batch_size
test_generator_tl.reset()   # Reiniciar el generador de datos de prueba antes␣
 ↪de la evaluación
print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de prueba = {test_generator_tl.batch_size}''')

# Evaluar el modelo en el conjunto de test
test_loss, test_acc = model.evaluate(test_generator_tl, verbose=2)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_acc:.4f}')

predictions = model.predict(test_generator_tl, steps=len(test_generator_tl))
y_pred = np.argmax(predictions, axis=1)
y_true = test_generator_tl.classes
class_names = list(test_generator_tl.class_indices.keys())
print(classification_report(y_true, y_pred, target_names=class_names,␣
 ↪zero_division=1))

# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure(figsize=(14, 7))
plt.plot(np.arange(0, len(history.history["loss"])), history.history["loss"],␣
 ↪label="train_loss")
plt.plot(np.arange(0, len(history.history["val_loss"])), history.
 ↪history["val_loss"], label="val_loss")
plt.plot(np.arange(0, len(history.history["accuracy"])), history.
 ↪history["accuracy"], label="train_acc")
plt.plot(np.arange(0, len(history.history["val_accuracy"])), history.
 ↪history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
#plt.legend(loc="lower left")
plt.savefig("training_plot.png")
plt.show()
```

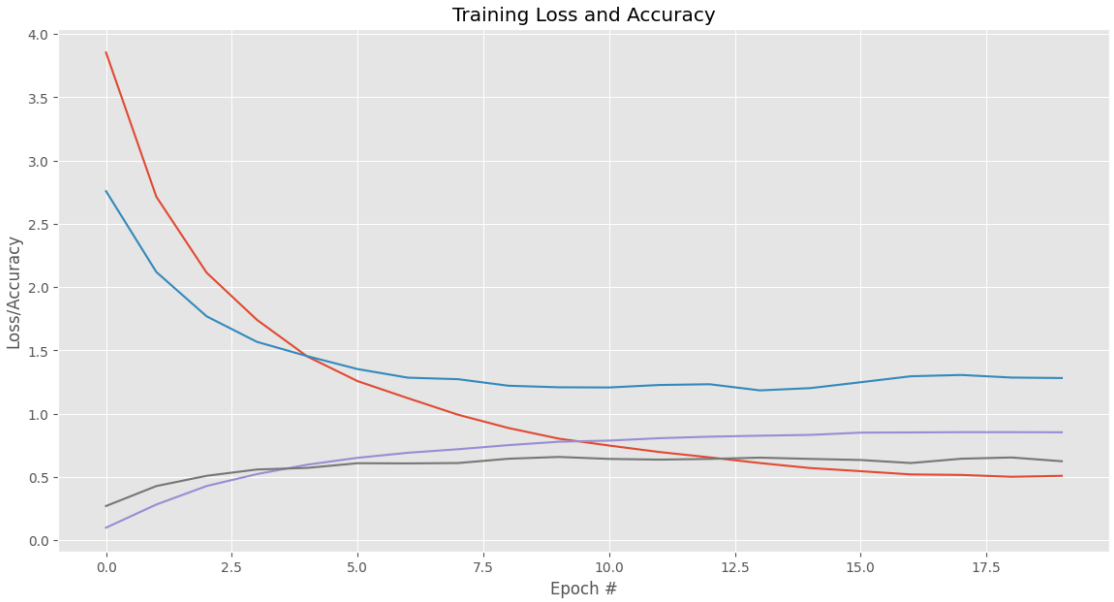
```
Batch size = 32  Epochs = 20
Batch size del generador de entrenamiento = 32
Batch size del generador de validacion = 32
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
Epoch 1/20
103/103 [==============================] - 39s 305ms/step - loss: 3.8541 -
accuracy: 0.1000 - val_loss: 2.7585 - val_accuracy: 0.2713
Epoch 2/20
103/103 [==============================] - 25s 241ms/step - loss: 2.7145 -
accuracy: 0.2831 - val_loss: 2.1200 - val_accuracy: 0.4290
Epoch 3/20
103/103 [==============================] - 25s 237ms/step - loss: 2.1144 -
accuracy: 0.4293 - val_loss: 1.7697 - val_accuracy: 0.5099
Epoch 4/20
103/103 [==============================] - 25s 239ms/step - loss: 1.7420 -
accuracy: 0.5236 - val_loss: 1.5681 - val_accuracy: 0.5597
Epoch 5/20
103/103 [==============================] - 25s 246ms/step - loss: 1.4527 -
accuracy: 0.5978 - val_loss: 1.4552 - val_accuracy: 0.5724
Epoch 6/20
103/103 [==============================] - 25s 243ms/step - loss: 1.2580 -
accuracy: 0.6522 - val_loss: 1.3532 - val_accuracy: 0.6094
Epoch 7/20
103/103 [==============================] - 25s 242ms/step - loss: 1.1233 -
accuracy: 0.6920 - val_loss: 1.2852 - val_accuracy: 0.6080
Epoch 8/20
103/103 [==============================] - 25s 239ms/step - loss: 0.9922 -
accuracy: 0.7200 - val_loss: 1.2731 - val_accuracy: 0.6108
Epoch 9/20
103/103 [==============================] - 25s 237ms/step - loss: 0.8875 -
accuracy: 0.7522 - val_loss: 1.2216 - val_accuracy: 0.6449
Epoch 10/20
103/103 [==============================] - 25s 242ms/step - loss: 0.8034 -
accuracy: 0.7790 - val_loss: 1.2093 - val_accuracy: 0.6591
Epoch 11/20
103/103 [==============================] - 25s 240ms/step - loss: 0.7491 -
accuracy: 0.7881 - val_loss: 1.2079 - val_accuracy: 0.6435
Epoch 12/20
103/103 [==============================] - 25s 242ms/step - loss: 0.6979 -
accuracy: 0.8069 - val_loss: 1.2274 - val_accuracy: 0.6378
Epoch 13/20
103/103 [==============================] - 25s 241ms/step - loss: 0.6556 -
accuracy: 0.8191 - val_loss: 1.2333 - val_accuracy: 0.6435
Epoch 14/20
103/103 [==============================] - 24s 237ms/step - loss: 0.6107 -
accuracy: 0.8267 - val_loss: 1.1848 - val_accuracy: 0.6534
Epoch 15/20
103/103 [==============================] - 25s 239ms/step - loss: 0.5711 -
accuracy: 0.8331 - val_loss: 1.2028 - val_accuracy: 0.6435
```

```
Epoch 16/20
103/103 [==============================] - 25s 239ms/step - loss: 0.5464 -
accuracy: 0.8510 - val_loss: 1.2490 - val_accuracy: 0.6349
Epoch 17/20
103/103 [==============================] - 25s 236ms/step - loss: 0.5208 -
accuracy: 0.8525 - val_loss: 1.2963 - val_accuracy: 0.6108
Epoch 18/20
103/103 [==============================] - 25s 236ms/step - loss: 0.5168 -
accuracy: 0.8547 - val_loss: 1.3065 - val_accuracy: 0.6449
Epoch 19/20
103/103 [==============================] - 25s 239ms/step - loss: 0.5023 -
accuracy: 0.8547 - val_loss: 1.2861 - val_accuracy: 0.6548
Epoch 20/20
103/103 [==============================] - 24s 233ms/step - loss: 0.5097 -
accuracy: 0.8538 - val_loss: 1.2823 - val_accuracy: 0.6250
[INFO]: Evaluando red neuronal…
Batch size = 32  Epochs = 20
Batch size del generador de prueba = 32
23/23 - 5s - loss: 1.1613 - accuracy: 0.6936 - 5s/epoch - 206ms/step
Test Loss: 1.1613
Test Accuracy: 0.6936
23/23 [==============================] - 5s 201ms/step
```

	precision	recall	f1-score	support
Adho Mukha Svanasana	0.62	0.67	0.65	15
Adho Mukha Vrksasana	0.50	0.67	0.57	15
Alanasana	0.50	0.27	0.35	15
Anjaneyasana	0.40	0.53	0.46	15
Ardha Chandrasana	0.72	0.87	0.79	15
Ardha Matsyendrasana	0.93	0.87	0.90	15
Ardha Navasana	0.60	0.80	0.69	15
Ardha Pincha Mayurasana	0.62	0.67	0.65	15
Ashta Chandrasana	0.00	0.00	0.00	15
Baddha Konasana	0.70	0.47	0.56	15
Bakasana	0.71	0.80	0.75	15
Balasana	1.00	0.33	0.50	15
Bitilasana	0.53	0.67	0.59	15
Camatkarasana	0.73	0.73	0.73	15
Dhanurasana	0.87	0.87	0.87	15
Eka Pada Rajakapotasana	0.58	0.73	0.65	15
Garudasana	0.78	0.93	0.85	15
Halasana	0.77	0.67	0.71	15
Hanumanasana	0.58	0.73	0.65	15
Malasana	0.71	0.67	0.69	15
Marjaryasana	0.75	0.60	0.67	15
Navasana	0.69	0.60	0.64	15
Padmasana	0.85	0.73	0.79	15
Parsva Virabhadrasana	1.00	0.13	0.24	15

Parsvottanasana	0.50	0.53	0.52	15
Paschimottanasana	1.00	0.87	0.93	15
Phalakasana	0.56	0.67	0.61	15
Pincha Mayurasana	0.53	0.53	0.53	15
Salamba Bhujangasana	0.75	0.60	0.67	15
Salamba Sarvangasana	0.76	0.87	0.81	15
Setu Bandha Sarvangasana	0.83	0.67	0.74	15
Sivasana	0.94	1.00	0.97	15
Supta Kapotasana	0.72	0.87	0.79	15
Trikonasana	0.79	0.73	0.76	15
Upavistha Konasana	0.67	0.80	0.73	15
Urdhva Dhanurasana	0.65	0.87	0.74	15
Urdhva Mukha Svsnssana	0.82	0.93	0.87	15
Ustrasana	0.76	0.87	0.81	15
Utkatasana	0.54	0.87	0.67	15
Uttanasana	0.69	0.73	0.71	15
Utthita Hasta Padangusthasana	0.83	0.67	0.74	15
Utthita Parsvakonasana	0.86	0.80	0.83	15
Vasisthasana	0.73	0.73	0.73	15
Virabhadrasana One	0.30	0.47	0.37	15
Virabhadrasana Three	0.92	0.73	0.81	15
Virabhadrasana Two	1.00	0.93	0.97	15
Vrksasana	0.93	0.87	0.90	15
accuracy			0.69	705
macro avg	0.71	0.69	0.68	705
weighted avg	0.71	0.69	0.68	705



10.3 Evaluacion de las metricas y toma de decisiones

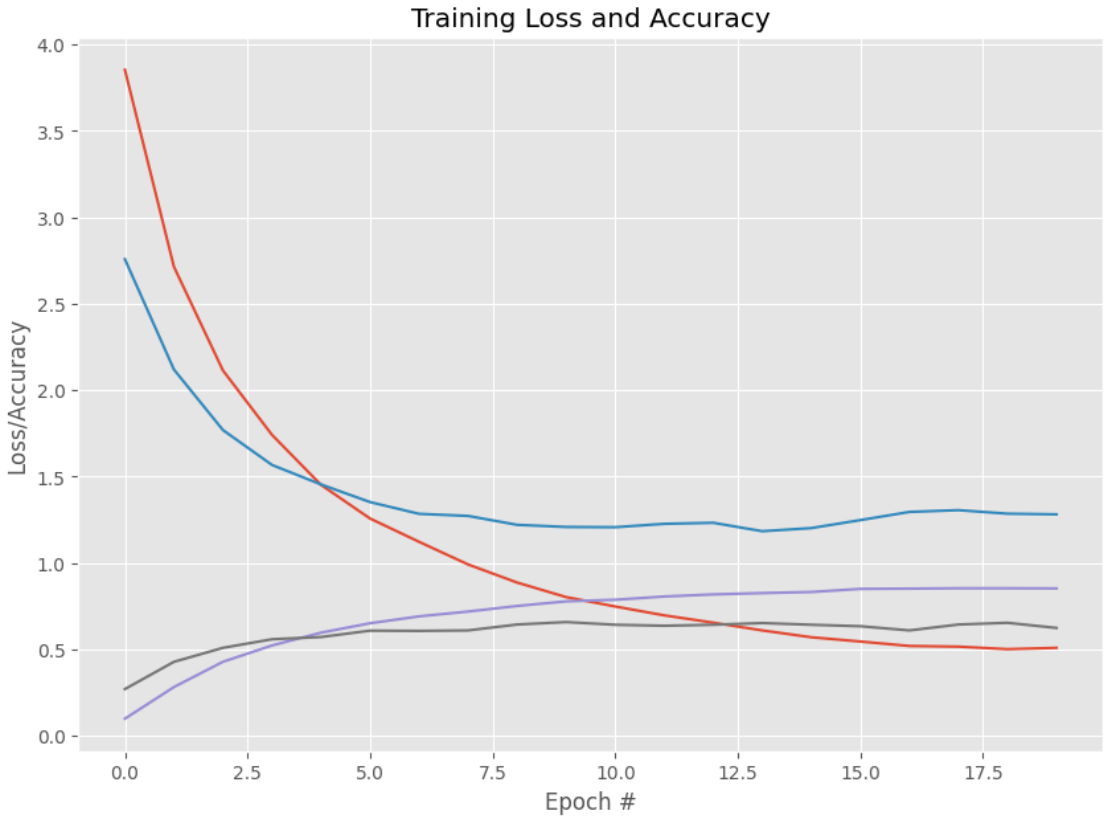
[60]:
```
test_generator_tl.batch_size = batch_size
test_generator_tl.reset()   # Reiniciar el generador de datos de prueba antes␣
 ↪de la evaluación
print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de prueba = {test_generator_tl.batch_size}''')

# Evaluar el modelo en el conjunto de test
test_loss, test_acc = model.evaluate(test_generator_tl, verbose=2)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_acc:.4f}')

# Obtener las predicciones en el conjunto de test
predictions = model.predict(test_generator_tl)
y_pred = np.argmax(predictions, axis=1)
y_true = test_generator_tl.classes
class_labels = list(test_generator_tl.class_indices.keys())

# Generar el reporte de clasificación
print(classification_report(y_true, y_pred,␣
 ↪target_names=class_labels,zero_division=1))

# Graficar las curvas de entrenamiento

plt.style.use("ggplot")
plt.figure(figsize=(10, 7))
plt.plot(np.arange(0, len(history.history["loss"])), history.history["loss"],␣
 ↪label="train_loss")
plt.plot(np.arange(0, len(history.history["val_loss"])), history.
 ↪history["val_loss"], label="val_loss")
plt.plot(np.arange(0, len(history.history["accuracy"])), history.
 ↪history["accuracy"], label="train_acc")
plt.plot(np.arange(0, len(history.history["val_accuracy"])), history.
 ↪history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
#plt.legend(loc="lower left")
plt.savefig("training_plot.png")
plt.show()
```

```
Batch size = 32  Epochs = 20
Batch size del generador de prueba = 32
23/23 - 5s - loss: 1.1613 - accuracy: 0.6936 - 5s/epoch - 213ms/step
```

```
Test Loss: 1.1613
Test Accuracy: 0.6936
23/23 [==============================] - 4s 189ms/step
```

	precision	recall	f1-score	support
Adho Mukha Svanasana	0.62	0.67	0.65	15
Adho Mukha Vrksasana	0.50	0.67	0.57	15
Alanasana	0.50	0.27	0.35	15
Anjaneyasana	0.40	0.53	0.46	15
Ardha Chandrasana	0.72	0.87	0.79	15
Ardha Matsyendrasana	0.93	0.87	0.90	15
Ardha Navasana	0.60	0.80	0.69	15
Ardha Pincha Mayurasana	0.62	0.67	0.65	15
Ashta Chandrasana	0.00	0.00	0.00	15
Baddha Konasana	0.70	0.47	0.56	15
Bakasana	0.71	0.80	0.75	15
Balasana	1.00	0.33	0.50	15
Bitilasana	0.53	0.67	0.59	15
Camatkarasana	0.73	0.73	0.73	15
Dhanurasana	0.87	0.87	0.87	15
Eka Pada Rajakapotasana	0.58	0.73	0.65	15
Garudasana	0.78	0.93	0.85	15
Halasana	0.77	0.67	0.71	15
Hanumanasana	0.58	0.73	0.65	15
Malasana	0.71	0.67	0.69	15
Marjaryasana	0.75	0.60	0.67	15
Navasana	0.69	0.60	0.64	15
Padmasana	0.85	0.73	0.79	15
Parsva Virabhadrasana	1.00	0.13	0.24	15
Parsvottanasana	0.50	0.53	0.52	15
Paschimottanasana	1.00	0.87	0.93	15
Phalakasana	0.56	0.67	0.61	15
Pincha Mayurasana	0.53	0.53	0.53	15
Salamba Bhujangasana	0.75	0.60	0.67	15
Salamba Sarvangasana	0.76	0.87	0.81	15
Setu Bandha Sarvangasana	0.83	0.67	0.74	15
Sivasana	0.94	1.00	0.97	15
Supta Kapotasana	0.72	0.87	0.79	15
Trikonasana	0.79	0.73	0.76	15
Upavistha Konasana	0.67	0.80	0.73	15
Urdhva Dhanurasana	0.65	0.87	0.74	15
Urdhva Mukha Svsnssana	0.82	0.93	0.87	15
Ustrasana	0.76	0.87	0.81	15
Utkatasana	0.54	0.87	0.67	15
Uttanasana	0.69	0.73	0.71	15
Utthita Hasta Padangusthasana	0.83	0.67	0.74	15
Utthita Parsvakonasana	0.86	0.80	0.83	15
Vasisthasana	0.73	0.73	0.73	15

Virabhadrasana One	0.30	0.47	0.37	15
Virabhadrasana Three	0.92	0.73	0.81	15
Virabhadrasana Two	1.00	0.93	0.97	15
Vrksasana	0.93	0.87	0.90	15
accuracy			0.69	705
macro avg	0.71	0.69	0.68	705
weighted avg	0.71	0.69	0.68	705

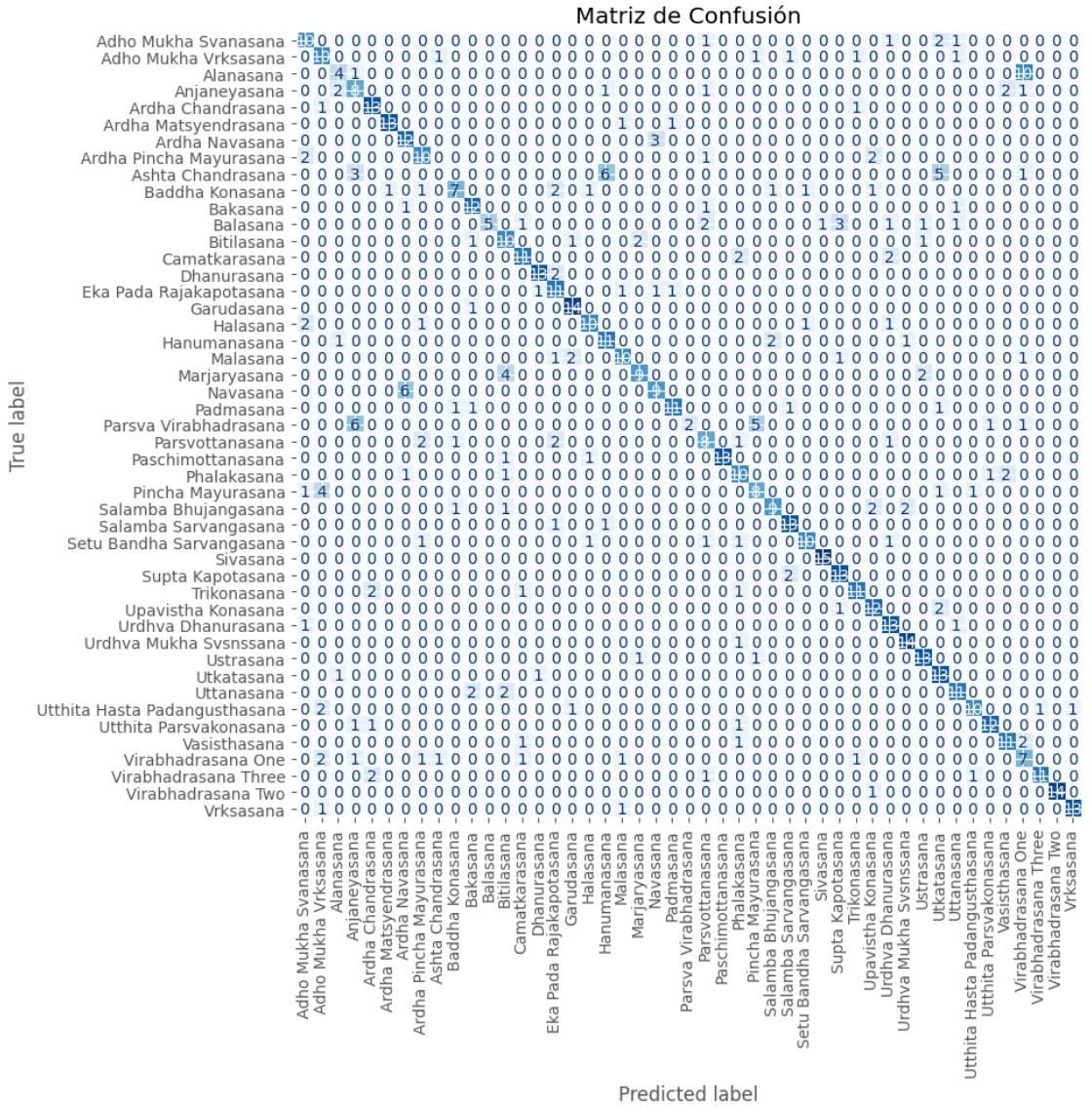


10.4 Matriz de confusion

[61]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪ConfusionMatrixDisplay

# Obtén las etiquetas predichas y las verdaderas etiquetas de tus datos de␣
 ↪prueba
# Calcula la matriz de confusión
predictions = model.predict(test_generator_tl)
```

```
cm = confusion_matrix(test_generator_tl.classes, np.argmax(predictions, axis=1))

# Grafica la matriz de confusión
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(cmap=plt.cm.Blues, ax=ax, xticks_rotation=90)

plt.title('Matriz de Confusión')
# Elimina la barra de color
ax.get_images()[0].colorbar.remove()
plt.tight_layout()
plt.show()
```

```
23/23 [==============================] - 5s 198ms/step
```



10.5 Evaluacion modelo VGG16

El metodo aplicado es transfer learning utilizando el modelo VGG16 pre-entrenado en el conjunto de datos de imágenes de yoga.

Hiperparámetros utilizados:

- `num_classes`: El número de clases del conjunto de datos (en este caso, 47 clases de poses de yoga).

El modelo se compila utilizando el optimizador Adam con una tasa de aprendizaje de 0.001 y la función de pérdida "categorical_crossentropy". La métrica de evaluación utilizada es la precisión ("accuracy").

El modelo se entrena utilizando un generador de datos (`train_generator_tl`) durante el número especificado de épocas.

Luego, se evalúa el modelo utilizando un generador de datos de prueba (`test_generator_tl`). Se calcula la pérdida y la precisión en el conjunto de prueba.

Finalmente, se muestra un informe de clasificación que incluye la precisión, el recall y el F1-score para cada clase.

Concluciones: de los resultado del transfer learning, algunos puntos importantes son:

- El modelo alcanza una precisión de prueba de aproximadamente 0.6936, lo que indica un rendimiento decente en la clasificación de las poses de yoga.
- El modelo muestra cierto grado de sobreajuste, ya que la precisión en el conjunto de entrenamiento es más alta que en el conjunto de validación.
- El gráfico de pérdida y precisión muestra una mejora gradual durante las primeras épocas, pero luego se estabiliza.
- Al observar el informe de clasificación, se puede ver que algunas poses de yoga tienen una precisión y recall más altos que otras, lo que puede indicar desafíos específicos para clasificar ciertas poses.

En general, el modelo de transfer learning utilizando VGG16 muestra un rendimiento prometedor en la clasificación de poses de yoga, pero podría haber margen para mejorar ajustando los hiperparámetros, realizando un ajuste fino de la red o utilizando otros modelos pre-entrenados.

10.6 APLICACION DE FINE TUNING VGG16 PARA MEJORAR LA CLASIFICACION

[62]:
```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout,␣
↪BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import regularizers
```

```
batch_size = 32
epochs = 20

train_generator_tl.batch_size = batch_size
train_generator_tl.reset()  # Reiniciar el generador de datos de entrenamiento␣
 ↪antes de la evaluación

valid_generator_tl.batch_size = batch_size
valid_generator_tl.reset()  # Reiniciar el generador de datos de validacion␣
 ↪antes de la evaluación

test_generator_tl.batch_size = batch_size
test_generator_tl.reset()   # Reiniciar el generador de datos de prueba antes␣
 ↪de la evaluación

print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de entrenamiento = {train_generator_tl.batch_size}
Batch size del generador de validacion = {valid_generator_tl.batch_size}
Batch size del generador de prueba = {test_generator_tl.batch_size}''')

# Cargar la arquitectura VGG16 sin las capas densas (fully connected)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224,␣
 ↪224, 3))

# Congelar las capas convolucionales inferiores
for layer in base_model.layers[:-1]:
    layer.trainable = False
base_model.layers[-2].trainable = True

# Agregar nuevas capas densas para la clasificación
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.0001))(x)
x = BatchNormalization()(x)
x = Dropout(0.55)(x)

x = Dense(256)(x)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Dropout(0.4)(x)

predictions = Dense(num_classes, activation='softmax')(x)

# Crear el modelo final que incluye tanto las capas de VGG16 como las nuevas␣
 ↪capas densas
model = Model(inputs=base_model.input, outputs=predictions)
```

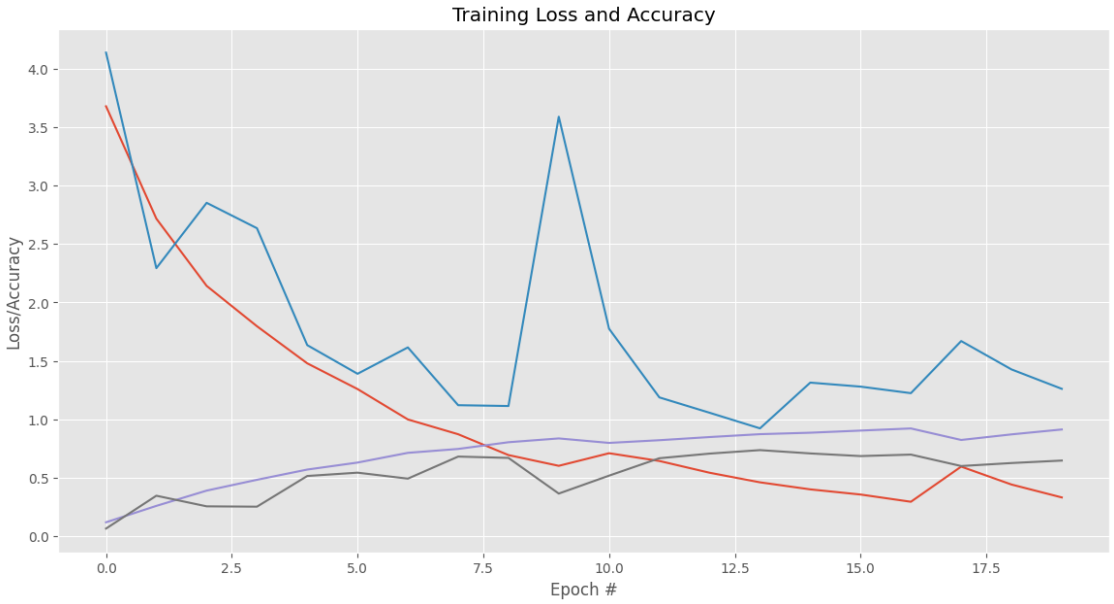
```
# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])

# Entrenar el modelo con tus datos
history = model.fit(train_generator_tl, epochs=epochs,␣
 ↪validation_data=valid_generator_tl)

# Almacenamos el modelo empleando la función model.save de Keras
model.save(r'./'+"FineTuning_Yoga_VGG.h5") #(X)

# Evaluando el modelo de predicción con las imágenes de prueba
print("[INFO]: Evaluando red neuronal...")
test_generator_tl.batch_size = batch_size
test_generator_tl.reset()   # Reiniciar el generador de datos de prueba antes␣
 ↪de la evaluación
print(f'''Batch size = {batch_size}  Epochs = {epochs}
Batch size del generador de prueba = {test_generator_tl.batch_size}''')

# Evaluar el modelo en el conjunto de test
test_loss, test_acc = model.evaluate(test_generator_tl, verbose=2)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_acc:.4f}')

predictions = model.predict(test_generator_tl, steps=len(test_generator_tl))
y_pred = np.argmax(predictions, axis=1)
y_true = test_generator_tl.classes
class_names = list(test_generator_tl.class_indices.keys())
print(classification_report(y_true, y_pred, target_names=class_names,␣
 ↪zero_division=1))

# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure(figsize=(14, 7))
plt.plot(np.arange(0, len(history.history["loss"])), history.history["loss"],␣
 ↪label="train_loss")
plt.plot(np.arange(0, len(history.history["val_loss"])), history.
 ↪history["val_loss"], label="val_loss")
plt.plot(np.arange(0, len(history.history["accuracy"])), history.
 ↪history["accuracy"], label="train_acc")
plt.plot(np.arange(0, len(history.history["val_accuracy"])), history.
 ↪history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
```

```
#plt.legend(loc="lower left")
plt.savefig("training_plot.png")
plt.show()
```

```
Batch size = 32  Epochs = 20
Batch size del generador de entrenamiento = 32
Batch size del generador de validacion = 32
Batch size del generador de prueba = 32
Epoch 1/20
103/103 [==============================] - 29s 248ms/step - loss: 3.6804 -
accuracy: 0.1177 - val_loss: 4.1401 - val_accuracy: 0.0639
Epoch 2/20
103/103 [==============================] - 25s 243ms/step - loss: 2.7179 -
accuracy: 0.2581 - val_loss: 2.2938 - val_accuracy: 0.3452
Epoch 3/20
103/103 [==============================] - 25s 243ms/step - loss: 2.1422 -
accuracy: 0.3889 - val_loss: 2.8535 - val_accuracy: 0.2543
Epoch 4/20
103/103 [==============================] - 26s 250ms/step - loss: 1.7967 -
accuracy: 0.4816 - val_loss: 2.6361 - val_accuracy: 0.2514
Epoch 5/20
103/103 [==============================] - 25s 242ms/step - loss: 1.4789 -
accuracy: 0.5692 - val_loss: 1.6348 - val_accuracy: 0.5142
Epoch 6/20
103/103 [==============================] - 25s 240ms/step - loss: 1.2586 -
accuracy: 0.6291 - val_loss: 1.3894 - val_accuracy: 0.5426
Epoch 7/20
103/103 [==============================] - 26s 251ms/step - loss: 0.9984 -
accuracy: 0.7124 - val_loss: 1.6153 - val_accuracy: 0.4915
Epoch 8/20
103/103 [==============================] - 26s 255ms/step - loss: 0.8717 -
accuracy: 0.7458 - val_loss: 1.1204 - val_accuracy: 0.6804
Epoch 9/20
103/103 [==============================] - 25s 244ms/step - loss: 0.6932 -
accuracy: 0.8033 - val_loss: 1.1134 - val_accuracy: 0.6690
Epoch 10/20
103/103 [==============================] - 27s 258ms/step - loss: 0.6012 -
accuracy: 0.8358 - val_loss: 3.5909 - val_accuracy: 0.3636
Epoch 11/20
103/103 [==============================] - 25s 243ms/step - loss: 0.7091 -
accuracy: 0.7975 - val_loss: 1.7751 - val_accuracy: 0.5170
Epoch 12/20
103/103 [==============================] - 26s 249ms/step - loss: 0.6428 -
accuracy: 0.8206 - val_loss: 1.1871 - val_accuracy: 0.6662
Epoch 13/20
103/103 [==============================] - 27s 259ms/step - loss: 0.5418 -
accuracy: 0.8477 - val_loss: 1.0555 - val_accuracy: 0.7060
```

```
Epoch 14/20
103/103 [==============================] - 25s 241ms/step - loss: 0.4599 -
accuracy: 0.8723 - val_loss: 0.9220 - val_accuracy: 0.7358
Epoch 15/20
103/103 [==============================] - 25s 243ms/step - loss: 0.3995 -
accuracy: 0.8848 - val_loss: 1.3138 - val_accuracy: 0.7074
Epoch 16/20
103/103 [==============================] - 25s 246ms/step - loss: 0.3553 -
accuracy: 0.9030 - val_loss: 1.2794 - val_accuracy: 0.6847
Epoch 17/20
103/103 [==============================] - 25s 245ms/step - loss: 0.2934 -
accuracy: 0.9213 - val_loss: 1.2234 - val_accuracy: 0.6974
Epoch 18/20
103/103 [==============================] - 25s 242ms/step - loss: 0.5945 -
accuracy: 0.8227 - val_loss: 1.6697 - val_accuracy: 0.5994
Epoch 19/20
103/103 [==============================] - 25s 246ms/step - loss: 0.4410 -
accuracy: 0.8705 - val_loss: 1.4268 - val_accuracy: 0.6250
Epoch 20/20
103/103 [==============================] - 25s 243ms/step - loss: 0.3305 -
accuracy: 0.9124 - val_loss: 1.2606 - val_accuracy: 0.6463
[INFO]: Evaluando red neuronal…
Batch size = 32  Epochs = 20
Batch size del generador de prueba = 32
23/23 - 4s - loss: 1.2985 - accuracy: 0.6652 - 4s/epoch - 183ms/step
Test Loss: 1.2985
Test Accuracy: 0.6652
23/23 [==============================] - 5s 197ms/step
                            precision    recall  f1-score   support

         Adho Mukha Svanasana       0.89      0.53      0.67        15
         Adho Mukha Vrksasana       0.23      0.60      0.33        15
                   Alanasana       0.00      0.00      0.00        15
                Anjaneyasana       0.86      0.40      0.55        15
           Ardha Chandrasana       0.65      0.73      0.69        15
         Ardha Matsyendrasana       1.00      0.27      0.42        15
               Ardha Navasana       0.58      1.00      0.73        15
      Ardha Pincha Mayurasana       0.65      0.73      0.69        15
            Ashta Chandrasana       0.29      0.47      0.36        15
              Baddha Konasana       0.86      0.40      0.55        15
                     Bakasana       1.00      0.67      0.80        15
                     Balasana       0.86      0.40      0.55        15
                  Bitilasana       0.80      0.53      0.64        15
               Camatkarasana       1.00      0.73      0.85        15
                 Dhanurasana       1.00      0.60      0.75        15
      Eka Pada Rajakapotasana       0.93      0.93      0.93        15
                  Garudasana       0.78      0.93      0.85        15
                    Halasana       0.55      0.73      0.63        15
```

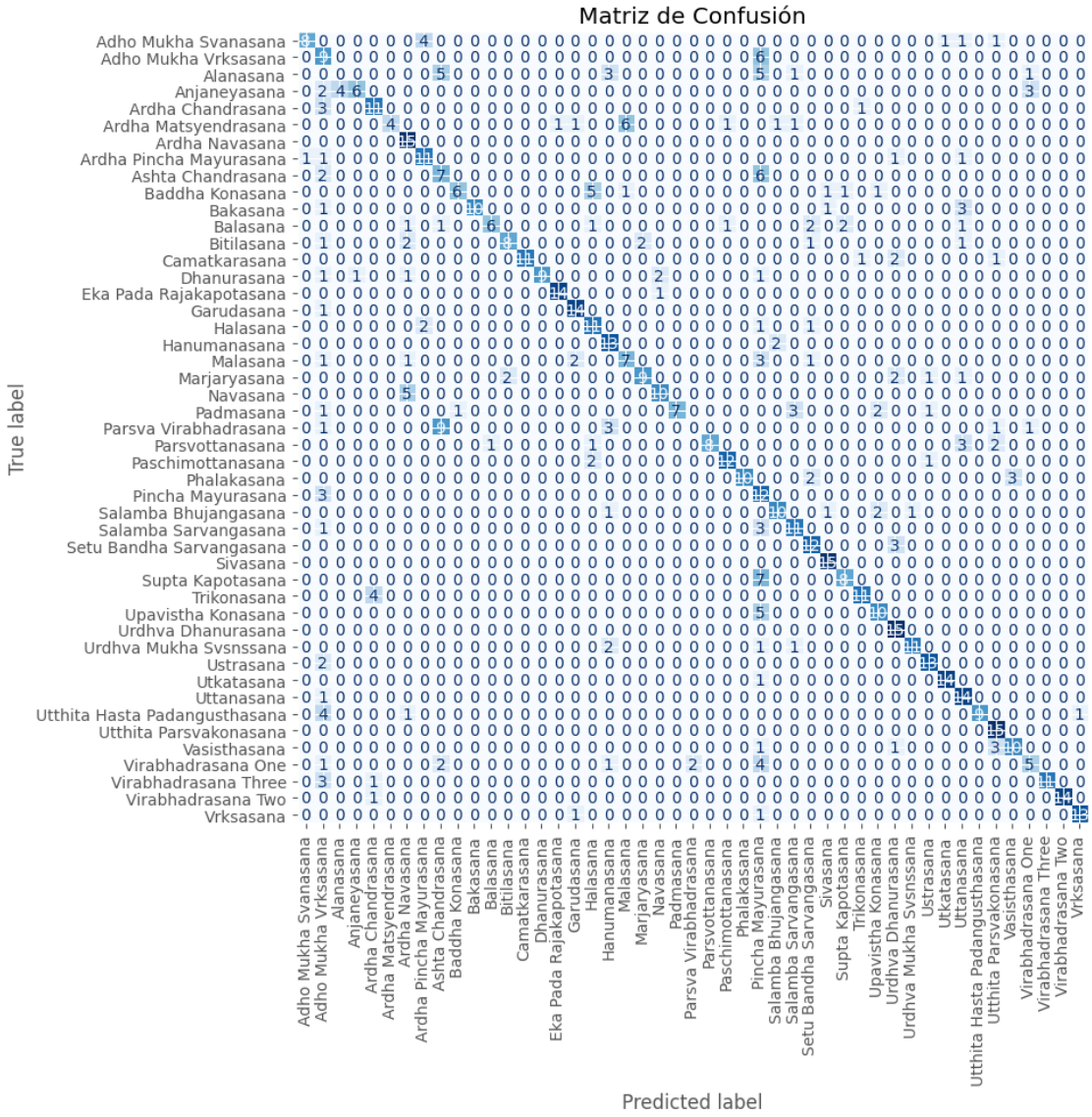
Hanumanasana	0.57	0.87	0.68	15
Malasana	0.50	0.47	0.48	15
Marjaryasana	0.82	0.60	0.69	15
Navasana	0.77	0.67	0.71	15
Padmasana	1.00	0.47	0.64	15
Parsva Virabhadrasana	0.00	0.00	0.00	15
Parsvottanasana	1.00	0.53	0.70	15
Paschimottanasana	0.86	0.80	0.83	15
Phalakasana	1.00	0.67	0.80	15
Pincha Mayurasana	0.21	0.80	0.33	15
Salamba Bhujangasana	0.77	0.67	0.71	15
Salamba Sarvangasana	0.65	0.73	0.69	15
Setu Bandha Sarvangasana	0.63	0.80	0.71	15
Sivasana	0.83	1.00	0.91	15
Supta Kapotasana	0.73	0.53	0.62	15
Trikonasana	0.85	0.73	0.79	15
Upavistha Konasana	0.67	0.67	0.67	15
Urdhva Dhanurasana	0.62	1.00	0.77	15
Urdhva Mukha Svsnssana	0.92	0.73	0.81	15
Ustrasana	0.81	0.87	0.84	15
Utkatasana	0.93	0.93	0.93	15
Uttanasana	0.56	0.93	0.70	15
Utthita Hasta Padangusthasana	1.00	0.60	0.75	15
Utthita Parsvakonasana	0.65	1.00	0.79	15
Vasisthasana	0.77	0.67	0.71	15
Virabhadrasana One	0.50	0.33	0.40	15
Virabhadrasana Three	1.00	0.73	0.85	15
Virabhadrasana Two	1.00	0.93	0.97	15
Vrksasana	0.93	0.87	0.90	15
accuracy			0.67	705
macro avg	0.73	0.67	0.67	705
weighted avg	0.73	0.67	0.67	705



10.7 Matriz de Confusion

[64]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪ConfusionMatrixDisplay

#predictions.argmax(axis=1) es igual a np.argmax(predictions, axis=1)
# Calcula la matriz de confusión
predictions = model.predict(test_generator_tl)
cm = confusion_matrix(test_generator_tl.classes, np.argmax(predictions, axis=1))

# Grafica la matriz de confusión
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(cmap=plt.cm.Blues, ax=ax, xticks_rotation=90)

plt.title('Matriz de Confusión')
# Elimina la barra de color
ax.get_images()[0].colorbar.remove()
plt.tight_layout()
plt.show()
```

```
23/23 [==============================] - 5s 204ms/step
```



10.8 CONCLUSION DEL ENTRENAMIENTO CON VGG16

En el primer método de transfer learning, se utilizó la arquitectura VGG16 preentrenada en ImageNet. Se congelaron todas las capas convolucionales y se agregaron nuevas capas densas para la clasificación. Se utilizó una función de activación ReLU, regularización L2, batch normalization y dropout para evitar el sobreajuste.

En cuanto a los hiperparámetros, se entrenó durante 20 épocas con un tamaño de lote de 32. Se utilizaron generadores de datos para el entrenamiento, validación y prueba, con el tamaño de lote configurado en 32 para cada uno.

El modelo logró una precisión de prueba de aproximadamente 0.6936 y un puntaje F1 promedio de 0.68 en la clasificación de las posturas de yoga. La gráfica de precisión y pérdida muestra una

mejora gradual durante el entrenamiento.

En el segundo método de fine-tuning, también se utilizó la arquitectura VGG16 preentrenada en ImageNet. Sin embargo, en este caso, se congelaron todas las capas convolucionales excepto la penúltima capa. Se agregaron nuevas capas densas para la clasificación y se aplicaron técnicas de regularización, batch normalization y dropout.

En cuanto a los hiperparámetros, se utilizaron los mismos valores que en el método de transfer learning anterior, con 20 épocas de entrenamiento y un tamaño de lote de 32.

El modelo obtuvo una precisión de prueba de aproximadamente 0.6652 y un puntaje F1 promedio de 0.67 en la clasificación de las posturas de yoga. La gráfica de precisión y pérdida muestra una convergencia más lenta en comparación con el método de transfer learning anterior.

Comparando ambos métodos, se puede observar que el primer método de transfer learning logró un rendimiento ligeramente mejor en términos de precisión y puntaje F1. Esto puede atribuirse a la capacidad de la red preentrenada para extraer características relevantes de las imágenes y al uso de técnicas de regularización para evitar el sobreajuste.

En cambio, el segundo método de fine-tuning, aunque permite una mayor flexibilidad al entrenar algunas de las capas convolucionales, no logró mejorar significativamente el rendimiento en comparación con el primer método. Esto puede deberse a que las capas convolucionales superiores de la red preentrenada ya han aprendido características relevantes para la clasificación de imágenes y no es necesario ajustarlas demasiado.

En resumen, el primer método de transfer learning congelando todas las capas convolucionales, excepto las capas densas agregadas, mostró mejores resultados en comparación con el segundo método de fine-tuning. Esto destaca la importancia de aprovechar el conocimiento previo de una red preentrenada para tareas de clasificación de imágenes.

11 ENTRENAMIENTO Y COMPARACION DE MODELOS (ResNet50, VGG16, MobileNet)

11.1 Generadores de datos para los diferentes modelos pre-entrenados

[95]:
```
from tensorflow.keras.applications.vgg16 import preprocess_input as␣
 ↪preprocess_input_vgg16
from tensorflow.keras.applications.resnet50 import preprocess_input as␣
 ↪preprocess_input_resnet50
from tensorflow.keras.applications.mobilenet import preprocess_input as␣
 ↪preprocess_input_mobilenet
from tensorflow.keras.preprocessing.image import ImageDataGenerator

input_shape = (224, 224, 3)  # Tamaño de entrada de las imágenes
batch_size = 32


# Definir generadores de datos para aumentación y preprocesamiento
train_datagen_vgg16 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_vgg16)
```

```
valid_datagen_vgg16 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_vgg16)
test_datagen_vgg16 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_vgg16)

train_datagen_resnet50 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_resnet50)
valid_datagen_resnet50 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_resnet50)
test_datagen_resnet50 =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_resnet50)

train_datagen_mobilenet =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_mobilenet)
valid_datagen_mobilenet =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_mobilenet)
test_datagen_mobilenet =␣
 ↪ImageDataGenerator(preprocessing_function=preprocess_input_mobilenet)

# Cargar imágenes de entrenamiento, validación y prueba utilizando generadores␣
 ↪de datos
train_generator_vgg16 = train_datagen_vgg16.flow_from_directory(
    train_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical'
)

valid_generator_vgg16 = valid_datagen_vgg16.flow_from_directory(
    valid_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator_vgg16 = test_datagen_vgg16.flow_from_directory(
    test_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

train_generator_resnet50 = train_datagen_resnet50.flow_from_directory(
    train_dir,
```

```
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical'
)

valid_generator_resnet50 = valid_datagen_resnet50.flow_from_directory(
    valid_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator_resnet50 = test_datagen_resnet50.flow_from_directory(
    test_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

train_generator_mobilenet = train_datagen_mobilenet.flow_from_directory(
    train_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical'
)

valid_generator_mobilenet = valid_datagen_mobilenet.flow_from_directory(
    valid_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator_mobilenet = test_datagen_mobilenet.flow_from_directory(
    test_dir,
    target_size=input_shape[:2],
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
```

```
Found 3289 images belonging to 47 classes.
Found 704 images belonging to 47 classes.
Found 705 images belonging to 47 classes.
```

```
Found 3289 images belonging to 47 classes.
Found 704 images belonging to 47 classes.
Found 705 images belonging to 47 classes.
Found 3289 images belonging to 47 classes.
Found 704 images belonging to 47 classes.
Found 705 images belonging to 47 classes.
```

11.2 Entrenamiento de modelos a comparar (VGG16, ResNet50, MobileNet)

Se realiza comparaciones entre modelos VGG16, ResNet50 y MobileNet utilizando diferentes combinaciones de hiperparámetros. A continuación, se describen las características principales del método de entrenamiento:

1. Selección de modelos: Se definen los modelos a utilizar (VGG16, ResNet50 y MobileNet) y se almacenan en una lista junto con los generadores de datos correspondientes.
2. Definición de hiperparámetros: Se definen diferentes combinaciones de hiperparámetros, como dropout, unidades densas, tasa de aprendizaje, regularización L2, épocas y tamaño de lote.
3. Creación del modelo: Para cada modelo y combinación de hiperparámetros, se crea un modelo específico. Se carga la arquitectura del modelo base preentrenado en ImageNet y se agregan capas densas personalizadas para la clasificación.
4. Compilación y entrenamiento del modelo: Se compila el modelo utilizando el optimizador Adam y se especifica la función de pérdida y las métricas. Luego, se entrena el modelo utilizando los generadores de datos de entrenamiento y validación, con los callbacks de Early Stopping para detener el entrenamiento temprano si no hay mejora en la métrica seleccionada.
5. Evaluación del modelo: Después de entrenar cada modelo, se evalúa su rendimiento en el conjunto de prueba. Se calcula la pérdida y la precisión y se muestra en la consola.
6. Almacenamiento y análisis de resultados: Todos los modelos entrenados se almacenan en una lista, junto con sus nombres y generadores de datos de prueba correspondientes. También se guarda el historial de métricas de entrenamiento para cada modelo. Luego, se crea un DataFrame para almacenar los resultados de precisión, puntaje F1 y exactitud para cada modelo y combinación de hiperparámetros. Los modelos con la máxima exactitud se filtran y se guardan en archivos h5.
7. Presentación de resultados: Finalmente, se muestra el DataFrame con los mejores modelos filtrados, que muestra la precisión (promedio ponderado y promedio macro), el puntaje F1 (promedio ponderado y promedio macro) y la exactitud para cada modelo.

En resumen, el código realiza la comparación de modelos VGG16, ResNet50 y MobileNet con diferentes combinaciones de hiperparámetros y muestra los resultados de precisión, puntaje F1 y exactitud para cada modelo. También guarda los mejores modelos según la exactitud obtenida en archivos h5. Esto permite analizar y seleccionar los modelos que logran el mejor rendimiento en la clasificación de las posturas de yoga.

[ ]:
```
%%capture
import pandas as pd
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.layers import Input, Conv2D, Activation,␣
 ↪GlobalAveragePooling2D, Flatten, Dense, Dropout, BatchNormalization,␣
 ↪MaxPooling2D
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.applications import VGG16, ResNet50, MobileNet
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model, load_model
from tensorflow.keras import regularizers
from tensorflow.keras.models import save_model
from sklearn.metrics import classification_report


num_classes= 47

hyperparameter_combinations = [
    {'dropout': 0.5, 'dense_units': 256, 'learning_rate': 0.001,␣
 ↪'l2_regularization': 0.0001, 'epochs': 50, 'batch_size': 32},
    {'dropout': 0.3, 'dense_units': 128, 'learning_rate': 0.001,␣
 ↪'l2_regularization': 0.0001, 'epochs': 16, 'batch_size': 32},
    {'dropout': 0.3, 'dense_units': 128, 'learning_rate': 0.01,␣
 ↪'l2_regularization': 0.0001, 'epochs': 32, 'batch_size': 64},
    {'dropout': 0.5, 'dense_units': 256, 'learning_rate': 0.01,␣
 ↪'l2_regularization': 0.0001, 'epochs': 32, 'batch_size': 64},
    {'dropout': 0.55, 'dense_units': 526, 'learning_rate': 0.001,␣
 ↪'l2_regularization': 0.0001, 'epochs': 32, 'batch_size': 128},
]

# Definir los callbacks deseados

early_stopping_callback = EarlyStopping(patience=3)

# Lista de modelos
models = [
    (VGG16, train_generator_vgg16, valid_generator_vgg16, test_generator_vgg16),
    (ResNet50, train_generator_resnet50, valid_generator_resnet50,␣
 ↪test_generator_resnet50),
    (MobileNet, train_generator_mobilenet, valid_generator_mobilenet,␣
 ↪test_generator_mobilenet)
]
all_models = []
all_models_names = []  # Lista para almacenar los nombres de los modelos
best_models = [] # Lista para almacenar los mejores modelos
history_list = []
model_names = []
test_generators = []  # Lista para almacenar los generadores de datos de prueba
```

```
for model_class, train_generator, valid_generator, test_generator in models:
    # Obtener el nombre del modelo
    model_name = model_class.__name__
    model_names.append(model_name)

    for hyperparameters in hyperparameter_combinations:
        dropout = hyperparameters['dropout']
        dense_units = hyperparameters['dense_units']
        learning_rate = hyperparameters['learning_rate']
        l2_regularization = hyperparameters['l2_regularization']
        epochs = hyperparameters['epochs']
        batch_size = hyperparameters['batch_size']

        # Crear el modelo
        base_model = model_class(weights='imagenet', include_top=False,␣
↪input_shape=(224, 224, 3))

        # Congelar las capas convolucionales inferiores
        for layer in base_model.layers[:-1]:
            layer.trainable = False
        base_model.layers[-2].trainable = True

        # Agregar capas densas para la clasificación
        x = base_model.output
        x = GlobalAveragePooling2D()(x)
        x = Dense(dense_units, activation='relu',␣
↪kernel_regularizer=regularizers.l2(l2_regularization))(x)
        x = Dropout(dropout)(x)

        x = Dense(dense_units)(x)
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
        x = Dropout(dropout)(x)
        predictions = Dense(num_classes, activation='softmax')(x)

        # Crear el modelo final
        model = Model(inputs=base_model.input, outputs=predictions)

        # Compilar el modelo con el optimizador y learning rate
        optimizer = Adam(learning_rate=learning_rate)
        model.compile(optimizer=optimizer, loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

        # Entrenar el modelo con los datos de entrenamiento y los callbacks
        history = model.fit(
            train_generator,
            epochs=epochs,
```

```
            steps_per_epoch=len(train_generator),
            validation_data=valid_generator,
            validation_steps=len(valid_generator),
            callbacks=[early_stopping_callback],
            batch_size=batch_size
        )

        # Guardando todos los modelos entrenados por cada iteracion
        all_models.append(model)
        # Guardando los nombres de los modelos
        all_models_names.append(model_name)
        # Almacenar el historial de métricas en una lista
        history_list.append(history)
        # Guardando el historial de cada entrenamiento en cada iteracion
        test_generators.append(test_generator)

        # Evaluar el modelo en el conjunto de prueba
        loss, accuracy = model.evaluate(test_generator)
        print(f"{model_class.__name__} - Loss: {loss} - Accuracy: {accuracy}")

  # Crear un DataFrame para almacenar los resultados
results_df = pd.DataFrame(columns=['Model', 'Precision (weighted avg)',␣
 ↪'Precision (macro avg)', 'F1-Score (weighted avg)', 'F1-Score (macro avg)',␣
 ↪'Accuracy'])

# Llenar el DataFrame con los resultados
for model, model_name, test_generator in zip(all_models, all_models_names,␣
 ↪test_generators):

    # Obtener las etiquetas verdaderas
    y_true = test_generator.labels

    # Predecir las etiquetas utilizando el modelo
    y_pred = model.predict(test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Generar el informe de clasificación
    report = classification_report(y_true, y_pred, zero_division=1,␣
 ↪output_dict=True)

    # Obtener los valores de precision, recall, f1-score y support
    precision_weighted_avg = report['weighted avg']['precision']
    precision_macro_avg = report['macro avg']['precision']
    f1_score_weighted_avg = report['weighted avg']['f1-score']
    f1_score_macro_avg = report['macro avg']['f1-score']

    # Calcular el accuracy
```

```
    cm = confusion_matrix(y_true, y_pred)
    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    # Agregar los resultados al DataFrame
    results_df.loc[len(results_df)] = [model_name, precision_weighted_avg,␣
 ↪precision_macro_avg, f1_score_weighted_avg,
                                       f1_score_macro_avg, accuracy]

# Obtener el máximo valor de accuracy para cada modelo
max_accuracy_per_model = results_df.groupby('Model')['Accuracy'].max()

# Filtrar los resultados originales usando los máximos valores de accuracy
filtered_results_df = results_df[results_df.apply(
    lambda row: row['Accuracy'] == max_accuracy_per_model[row['Model']],␣
 ↪axis=1)]

# Guardar los mejores modelos filtrados
for model_name in filtered_results_df['Model']:
    # Obtener el índice del modelo con el nombre correspondiente
    model_index = all_models_names.index(model_name)

    # Obtener el modelo correspondiente
    best_model = all_models[model_index]

    # Crear el directorio para el modelo si no existe
    model_dir = os.path.join(MODELS_DIR, model_name)
    os.makedirs(model_dir, exist_ok=True)

    # Guardar el modelo en el directorio correspondiente
    model_path = os.path.join(model_dir, f'{model_name}.h5')
    save_model(best_model, model_path)

# Mostrar el DataFrame con los resultados filtrados
print(filtered_results_df.to_string(index=False))
```

11.3 Eleccion del mejor modelo por tipo de red

[67]:
```
from sklearn.metrics import classification_report
# Crear un DataFrame para almacenar los resultados
results_df = pd.DataFrame(columns=['Model', 'Precision (weighted avg)',␣
 ↪'Precision (macro avg)', 'F1-Score (weighted avg)', 'F1-Score (macro avg)',␣
 ↪'Accuracy'])

# Llenar el DataFrame con los resultados
for model, model_name, test_generator in zip(all_models, all_models_names,␣
 ↪test_generators):
```

```
    # Obtener las etiquetas verdaderas
    y_true = test_generator.labels

    # Predecir las etiquetas utilizando el modelo
    y_pred = model.predict(test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Generar el informe de clasificación
    report = classification_report(y_true, y_pred, zero_division=1,␣
 ↪output_dict=True)

    # Obtener los valores de precision, recall, f1-score y support
    precision_weighted_avg = report['weighted avg']['precision']
    precision_macro_avg = report['macro avg']['precision']
    f1_score_weighted_avg = report['weighted avg']['f1-score']
    f1_score_macro_avg = report['macro avg']['f1-score']

    # Calcular el accuracy
    cm = confusion_matrix(y_true, y_pred)
    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    # Agregar los resultados al DataFrame
    results_df.loc[len(results_df)] = [model_name, precision_weighted_avg,␣
 ↪precision_macro_avg, f1_score_weighted_avg,
                                       f1_score_macro_avg, accuracy]

# Obtener el máximo valor de accuracy para cada modelo
max_accuracy_per_model = results_df.groupby('Model')['Accuracy'].max()

# Filtrar los resultados originales usando los máximos valores de accuracy
filtered_results_df = results_df[results_df.apply(
    lambda row: row['Accuracy'] == max_accuracy_per_model[row['Model']],␣
 ↪axis=1)]

# Mostrar el DataFrame con los resultados filtrados
print(filtered_results_df.to_string(index=False))
```

```
23/23 [==============================] - 4s 180ms/step
23/23 [==============================] - 4s 192ms/step
23/23 [==============================] - 4s 176ms/step
23/23 [==============================] - 4s 168ms/step
23/23 [==============================] - 4s 190ms/step
23/23 [==============================] - 4s 184ms/step
23/23 [==============================] - 4s 172ms/step
23/23 [==============================] - 4s 193ms/step
23/23 [==============================] - 4s 171ms/step
```

```
23/23 [==============================] - 4s 168ms/step
23/23 [==============================] - 4s 163ms/step
23/23 [==============================] - 3s 150ms/step
23/23 [==============================] - 4s 155ms/step
23/23 [==============================] - 4s 158ms/step
23/23 [==============================] - 4s 157ms/step
    Model  Precision (weighted avg)  Precision (macro avg)  F1-Score (weighted
avg)  F1-Score (macro avg)  Accuracy
    VGG16                  0.813850               0.813850
0.775939              0.775939  0.778723
 ResNet50                  0.736124               0.736124
0.704948              0.704948  0.709220
MobileNet                  0.791342               0.791342
0.750848              0.750848  0.754610
```

12 EVALUACION Y TOMA DE DECISIONES PARA ELEJIR EL MEJOR MODELO

12.1 Revision de Metricas de todas las iteraciones de entrenamiento

[68]:
```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
# Crear un DataFrame para almacenar los resultados
results_df = pd.DataFrame(columns=['Model', 'Precision (weighted avg)',␣
 ↪'Precision (macro avg)', 'F1-Score (weighted avg)', 'F1-Score (macro avg)',␣
 ↪'Accuracy'])

# Llenar el DataFrame con los resultados
for model, model_name, test_generator in zip(all_models, all_models_names,␣
 ↪test_generators):

    # Obtener las etiquetas verdaderas
    y_true = test_generator.labels

    # Predecir las etiquetas utilizando el modelo
    y_pred = model.predict(test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Generar el informe de clasificación
    report = classification_report(y_true, y_pred, zero_division=1,␣
 ↪output_dict=True)

    # Obtener los valores de precision, recall, f1-score y support
    precision_weighted_avg = report['weighted avg']['precision']
    precision_macro_avg = report['macro avg']['precision']
    f1_score_weighted_avg = report['weighted avg']['f1-score']
    f1_score_macro_avg = report['macro avg']['f1-score']
```

```
    # Calcular el accuracy
    cm = confusion_matrix(y_true, y_pred)
    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    # Agregar los resultados al DataFrame
    results_df.loc[len(results_df)] = [model_name, precision_weighted_avg,␣
 ↪precision_macro_avg, f1_score_weighted_avg, f1_score_macro_avg, accuracy]

# Mostrar el DataFrame con los resultados
#print(results_df.to_string(index=False))
results_df
```

```
23/23 [==============================] - 4s 179ms/step
23/23 [==============================] - 4s 190ms/step
23/23 [==============================] - 4s 176ms/step
23/23 [==============================] - 4s 186ms/step
23/23 [==============================] - 4s 174ms/step
23/23 [==============================] - 4s 175ms/step
23/23 [==============================] - 4s 188ms/step
23/23 [==============================] - 4s 180ms/step
23/23 [==============================] - 5s 206ms/step
23/23 [==============================] - 5s 204ms/step
23/23 [==============================] - 3s 149ms/step
23/23 [==============================] - 3s 148ms/step
23/23 [==============================] - 4s 160ms/step
23/23 [==============================] - 4s 152ms/step
23/23 [==============================] - 3s 149ms/step
```

[68]:

	Model	Precision (weighted avg)	Precision (macro avg)	\
0	VGG16	0.767282	0.767282	
1	VGG16	0.780923	0.780923	
2	VGG16	0.737124	0.737124	
3	VGG16	0.775089	0.775089	
4	VGG16	0.813850	0.813850	
5	ResNet50	0.736124	0.736124	
6	ResNet50	0.719373	0.719373	
7	ResNet50	0.679140	0.679140	
8	ResNet50	0.640020	0.640020	
9	ResNet50	0.723267	0.723267	
10	MobileNet	0.791342	0.791342	
11	MobileNet	0.761445	0.761445	
12	MobileNet	0.701015	0.701015	
13	MobileNet	0.708215	0.708215	
14	MobileNet	0.781086	0.781086	

F1-Score (weighted avg)	F1-Score (macro avg)	Accuracy

0	0.747698	0.747698	0.760284
1	0.756022	0.756022	0.764539
2	0.669144	0.669144	0.670922
3	0.705713	0.705713	0.706383
4	0.775939	0.775939	0.778723
5	0.704948	0.704948	0.709220
6	0.682179	0.682179	0.685106
7	0.607183	0.607183	0.619858
8	0.568115	0.568115	0.592908
9	0.686543	0.686543	0.686525
10	0.750848	0.750848	0.754610
11	0.664288	0.664288	0.672340
12	0.610533	0.610533	0.621277
13	0.662019	0.662019	0.673759
14	0.747149	0.747149	0.750355

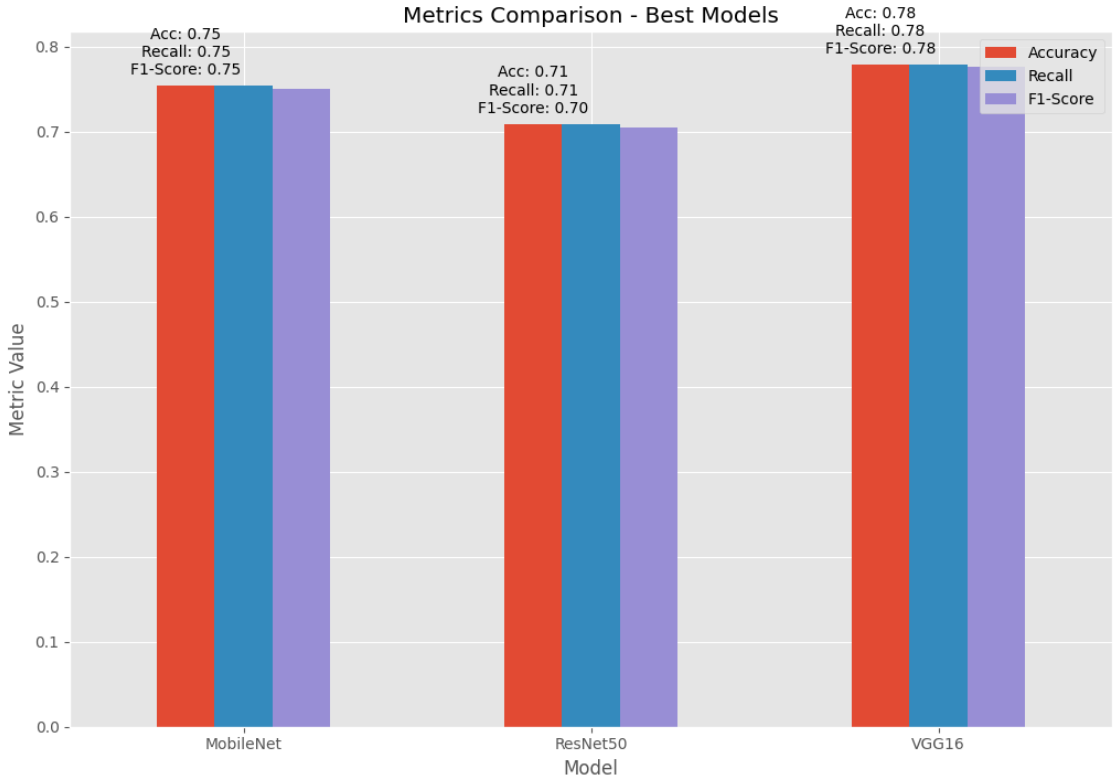
12.2 Grafica de barras de métricas del mejor modelo por tipo

[72]:
```
from sklearn.metrics import f1_score as f1_score_func
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score

# Crear un DataFrame para almacenar las métricas
metrics_df = pd.DataFrame(columns=['Model', 'Accuracy', 'Recall', 'F1-Score'])

# Obtener los mejores modelos por tipo y agregar las métricas al DataFrame
for model_name in set(filtered_results_df['Model']):
    # Obtener el índice del modelo con el mayor accuracy
    best_model_index = filtered_results_df[filtered_results_df['Model'] ==␣
 ↪model_name].index[0]

    # Obtener el mejor modelo
    best_model = all_models[best_model_index]

    # Obtener el generador de prueba para el mejor modelo
    best_test_generator = test_generators[best_model_index]

    # Obtener las etiquetas verdaderas
    y_true = best_test_generator.labels

    # Predecir las etiquetas utilizando el mejor modelo
    y_pred = best_model.predict(best_test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Calcular las métricas de accuracy, recall y f1-score
    accuracy = accuracy_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred, average='weighted')
```

```
    f1_score_value = f1_score_func(y_true, y_pred, average='weighted')

    # Agregar las métricas al DataFrame
    metrics_df.loc[len(metrics_df)] = [model_name, accuracy, recall,␣
 ↪f1_score_value]


# Ordenar el DataFrame por el modelo
metrics_df = metrics_df.sort_values(by='Model')

# Graficar las barras con las métricas
plt.rcParams["figure.figsize"] = (10, 7)
plt.figure(figsize=(7, 7))
ax = metrics_df.plot(x='Model', y=['Accuracy', 'Recall', 'F1-Score'],␣
 ↪kind='bar')
plt.title('Metrics Comparison - Best Models')
plt.xlabel('Model')
plt.ylabel('Metric Value')
plt.legend(loc='upper right')
plt.xticks(rotation=0)
plt.tight_layout(pad=0, w_pad=0, h_pad=0)

# Agregar las etiquetas de accuracy, recall y f1-score sobre cada barra
for p, accuracy, recall, f1_score_value in zip(ax.patches,␣
 ↪metrics_df['Accuracy'], metrics_df['Recall'], metrics_df['F1-Score']):
    ax.annotate(f'Acc: {accuracy:.2f}\nRecall: {recall:.2f}\nF1-Score:␣
 ↪{f1_score_value:.2f}', (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center', va='bottom', xytext=(0, 5), textcoords='offset␣
 ↪points')

plt.show()
```

```
23/23 [==============================] - 5s 191ms/step
23/23 [==============================] - 3s 147ms/step
23/23 [==============================] - 4s 178ms/step
```

```
<Figure size 700x700 with 0 Axes>
```



12.3 Matriz de Confusión

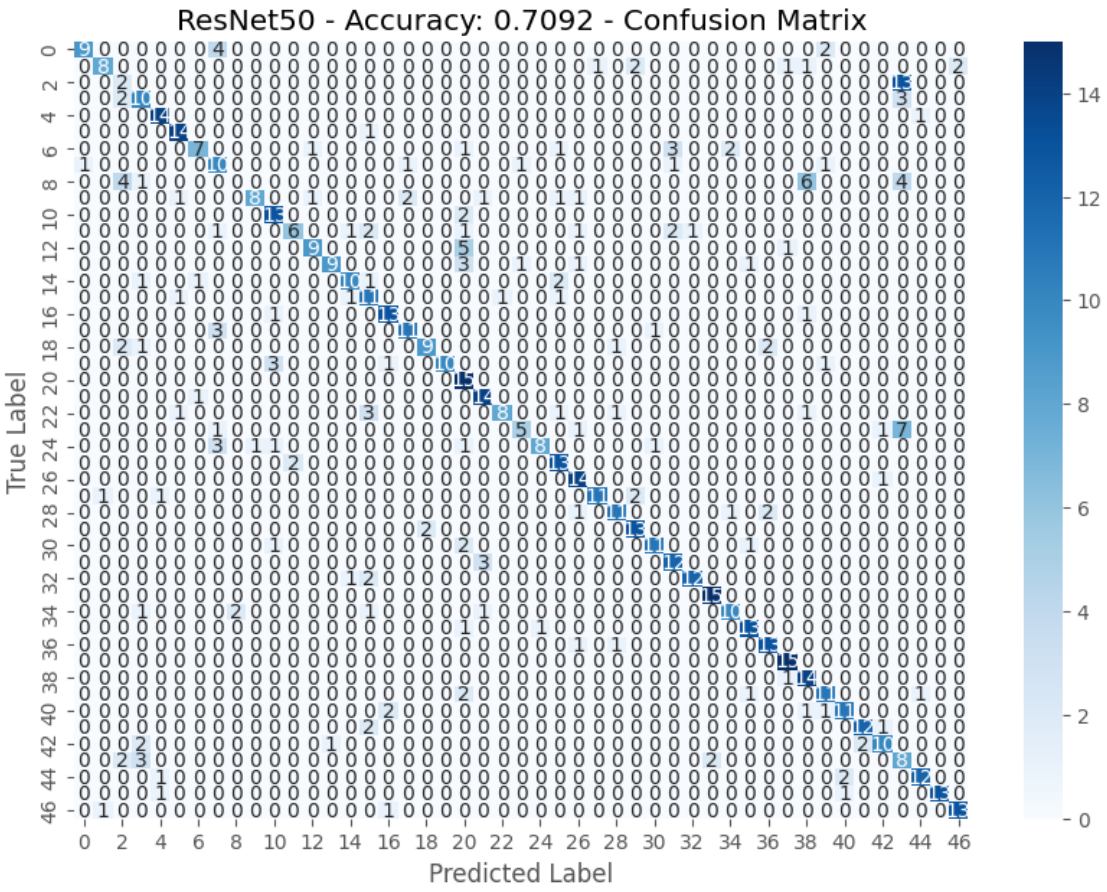
[75]:
```
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix

# Obtener el máximo valor de accuracy para cada modelo
max_accuracy_per_model = results_df.groupby('Model')['Accuracy'].max()

# Filtrar los resultados originales usando los máximos valores de accuracy
filtered_results_df = results_df[results_df.apply(
    lambda row: row['Accuracy'] == max_accuracy_per_model[row['Model']],␣
 ↪axis=1)]

# Mostrar el DataFrame con los resultados filtrados
print(filtered_results_df.to_string(index=False))

# Graficar los resultados del mejor modelo para cada arquitectura
for model_name in set(filtered_results_df['Model']):
    # Obtener el índice del modelo con el mayor accuracy
    best_model_index = filtered_results_df[filtered_results_df['Model'] ==␣
 ↪model_name].index[0]
```

```
    # Obtener el mejor modelo
    best_model = all_models[best_model_index]

    # Obtener el generador de prueba para el mejor modelo
    best_test_generator = test_generators[best_model_index]

    # Obtener las etiquetas verdaderas
    y_true = best_test_generator.labels

    # Predecir las etiquetas utilizando el mejor modelo
    y_pred = best_model.predict(best_test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Generar el informe de clasificación
    report = classification_report(y_true, y_pred, target_names=class_names,␣
↪zero_division=1)

    # Calcular la matriz de confusión
    cm = confusion_matrix(y_true, y_pred)

    # Calcular el accuracy
    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    # Mostrar el reporte de clasificación
    print(f"Classification Report - {model_name}")
    print(report)
    print("-----------------------------------------------------------")

    # Plotear la matriz de confusión
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)
    plt.title(f'{model_name} - Accuracy: {accuracy:.4f} - Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()
```

```
    Model  Precision (weighted avg)  Precision (macro avg)  F1-Score (weighted
avg)  F1-Score (macro avg)  Accuracy
    VGG16                  0.813850               0.813850
0.775939               0.775939  0.778723
 ResNet50                  0.736124               0.736124
0.704948               0.704948  0.709220
MobileNet                  0.791342               0.791342
0.750848               0.750848  0.754610
23/23 [==============================] - 4s 178ms/step
Classification Report - ResNet50
                              precision    recall  f1-score   support
```

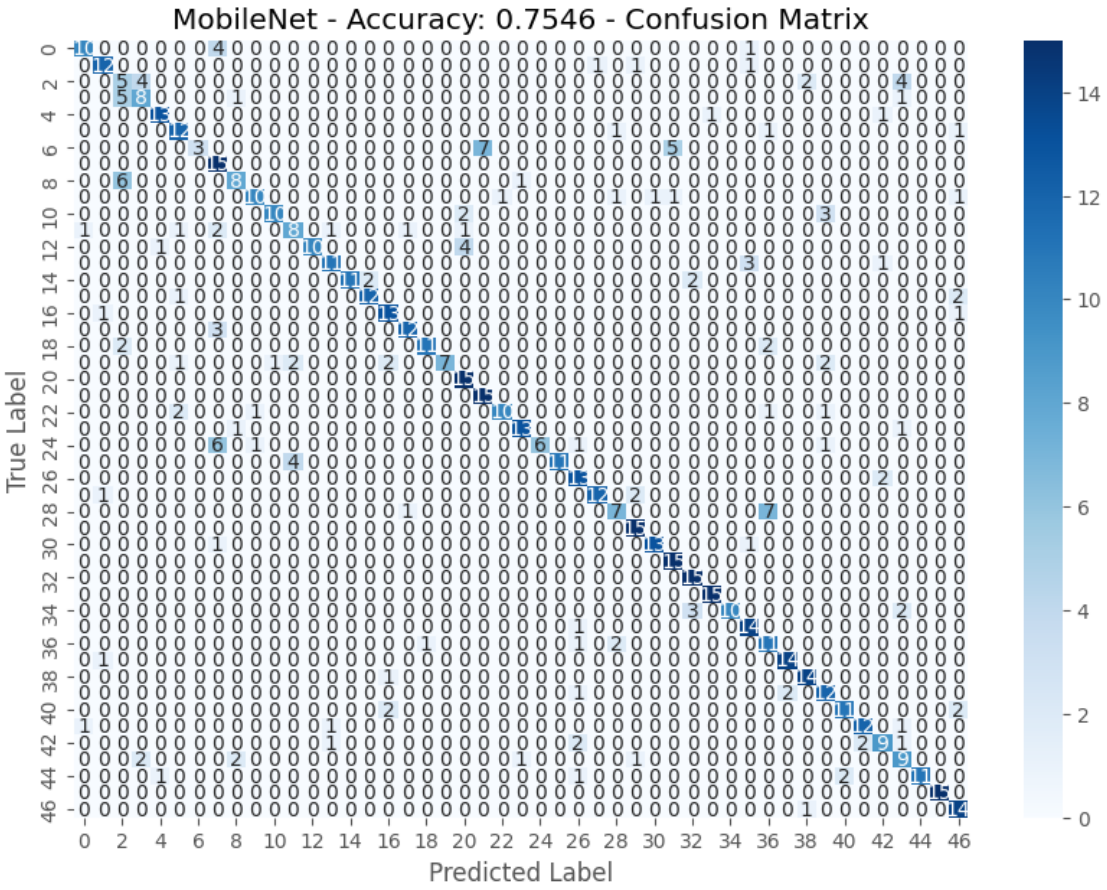
Adho Mukha Svanasana	0.90	0.60	0.72	15
Adho Mukha Vrksasana	0.80	0.53	0.64	15
Alanasana	0.17	0.13	0.15	15
Anjaneyasana	0.53	0.67	0.59	15
Ardha Chandrasana	0.82	0.93	0.87	15
Ardha Matsyendrasana	0.82	0.93	0.87	15
Ardha Navasana	0.78	0.47	0.58	15
Ardha Pincha Mayurasana	0.45	0.67	0.54	15
Ashta Chandrasana	0.00	0.00	0.00	15
Baddha Konasana	0.89	0.53	0.67	15
Bakasana	0.68	0.87	0.76	15
Balasana	0.75	0.40	0.52	15
Bitilasana	0.82	0.60	0.69	15
Camatkarasana	0.90	0.60	0.72	15
Dhanurasana	0.77	0.67	0.71	15
Eka Pada Rajakapotasana	0.48	0.73	0.58	15
Garudasana	0.76	0.87	0.81	15
Halasana	0.79	0.73	0.76	15
Hanumanasana	0.82	0.60	0.69	15
Malasana	1.00	0.67	0.80	15
Marjaryasana	0.45	1.00	0.62	15
Navasana	0.74	0.93	0.82	15
Padmasana	0.89	0.53	0.67	15
Parsva Virabhadrasana	0.71	0.33	0.45	15
Parsvottanasana	0.89	0.53	0.67	15
Paschimottanasana	0.68	0.87	0.76	15
Phalakasana	0.70	0.93	0.80	15
Pincha Mayurasana	0.92	0.73	0.81	15
Salamba Bhujangasana	0.79	0.73	0.76	15
Salamba Sarvangasana	0.76	0.87	0.81	15
Setu Bandha Sarvangasana	0.85	0.73	0.79	15
Sivasana	0.67	0.80	0.73	15
Supta Kapotasana	0.92	0.80	0.86	15
Trikonasana	0.88	1.00	0.94	15
Upavistha Konasana	0.77	0.67	0.71	15
Urdhva Dhanurasana	0.81	0.87	0.84	15
Urdhva Mukha Svsnssana	0.76	0.87	0.81	15
Ustrasana	0.83	1.00	0.91	15
Utkatasana	0.58	0.93	0.72	15
Uttanasana	0.69	0.73	0.71	15
Utthita Hasta Padangusthasana	0.79	0.73	0.76	15
Utthita Parsvakonasana	0.86	0.80	0.83	15
Vasisthasana	0.77	0.67	0.71	15
Virabhadrasana One	0.23	0.53	0.32	15
Virabhadrasana Three	0.86	0.80	0.83	15
Virabhadrasana Two	1.00	0.87	0.93	15
Vrksasana	0.87	0.87	0.87	15

accuracy			0.71	705
macro avg	0.74	0.71	0.70	705
weighted avg	0.74	0.71	0.70	705




```
23/23 [==============================] - 4s 159ms/step
Classification Report - MobileNet
                              precision    recall  f1-score   support

          Adho Mukha Svanasana       0.83      0.67      0.74        15
          Adho Mukha Vrksasana       0.80      0.80      0.80        15
                    Alanasana       0.28      0.33      0.30        15
                 Anjaneyasana       0.57      0.53      0.55        15
            Ardha Chandrasana       0.87      0.87      0.87        15
          Ardha Matsyendrasana       0.71      0.80      0.75        15
                Ardha Navasana       1.00      0.20      0.33        15
       Ardha Pincha Mayurasana       0.48      1.00      0.65        15
            Ashta Chandrasana       0.67      0.53      0.59        15
```

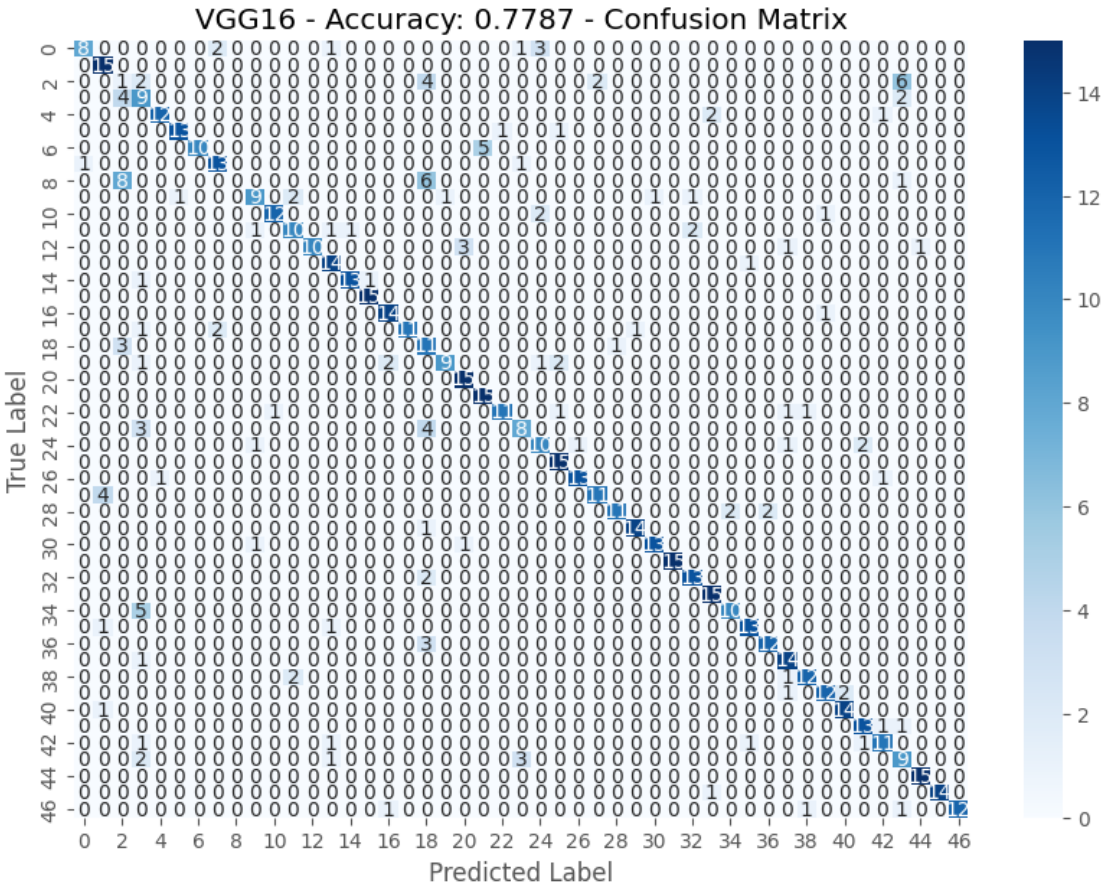
Baddha Konasana	0.83	0.67	0.74	15
Bakasana	0.91	0.67	0.77	15
Balasana	0.57	0.53	0.55	15
Bitilasana	1.00	0.67	0.80	15
Camatkarasana	0.79	0.73	0.76	15
Dhanurasana	1.00	0.73	0.85	15
Eka Pada Rajakapotasana	0.86	0.80	0.83	15
Garudasana	0.72	0.87	0.79	15
Halasana	0.86	0.80	0.83	15
Hanumanasana	0.92	0.73	0.81	15
Malasana	1.00	0.47	0.64	15
Marjaryasana	0.68	1.00	0.81	15
Navasana	0.68	1.00	0.81	15
Padmasana	0.91	0.67	0.77	15
Parsva Virabhadrasana	0.87	0.87	0.87	15
Parsvottanasana	1.00	0.40	0.57	15
Paschimottanasana	1.00	0.73	0.85	15
Phalakasana	0.65	0.87	0.74	15
Pincha Mayurasana	0.92	0.80	0.86	15
Salamba Bhujangasana	0.64	0.47	0.54	15
Salamba Sarvangasana	0.79	1.00	0.88	15
Setu Bandha Sarvangasana	0.93	0.87	0.90	15
Sivasana	0.71	1.00	0.83	15
Supta Kapotasana	0.75	1.00	0.86	15
Trikonasana	0.94	1.00	0.97	15
Upavistha Konasana	1.00	0.67	0.80	15
Urdhva Dhanurasana	0.70	0.93	0.80	15
Urdhva Mukha Svsnssana	0.50	0.73	0.59	15
Ustrasana	0.88	0.93	0.90	15
Utkatasana	0.82	0.93	0.87	15
Uttanasana	0.63	0.80	0.71	15
Utthita Hasta Padangusthasana	0.85	0.73	0.79	15
Utthita Parsvakonasana	0.86	0.80	0.83	15
Vasisthasana	0.69	0.60	0.64	15
Virabhadrasana One	0.47	0.60	0.53	15
Virabhadrasana Three	1.00	0.73	0.85	15
Virabhadrasana Two	1.00	1.00	1.00	15
Vrksasana	0.67	0.93	0.78	15
accuracy			0.75	705
macro avg	0.79	0.75	0.75	705
weighted avg	0.79	0.75	0.75	705




```
23/23 [==============================] - 4s 194ms/step
Classification Report - VGG16
```

	precision	recall	f1-score	support
Adho Mukha Svanasana	0.89	0.53	0.67	15
Adho Mukha Vrksasana	0.71	1.00	0.83	15
Alanasana	0.06	0.07	0.06	15
Anjaneyasana	0.35	0.60	0.44	15
Ardha Chandrasana	0.92	0.80	0.86	15
Ardha Matsyendrasana	0.93	0.87	0.90	15
Ardha Navasana	1.00	0.67	0.80	15
Ardha Pincha Mayurasana	0.76	0.87	0.81	15
Ashta Chandrasana	1.00	0.00	0.00	15
Baddha Konasana	0.75	0.60	0.67	15
Bakasana	0.92	0.80	0.86	15
Balasana	0.71	0.67	0.69	15
Bitilasana	1.00	0.67	0.80	15
Camatkarasana	0.74	0.93	0.82	15
Dhanurasana	0.93	0.87	0.90	15

Eka Pada Rajakapotasana	0.94	1.00	0.97	15
Garudasana	0.82	0.93	0.87	15
Halasana	1.00	0.73	0.85	15
Hanumanasana	0.35	0.73	0.48	15
Malasana	0.90	0.60	0.72	15
Marjaryasana	0.79	1.00	0.88	15
Navasana	0.75	1.00	0.86	15
Padmasana	0.92	0.73	0.81	15
Parsva Virabhadrasana	0.62	0.53	0.57	15
Parsvottanasana	0.62	0.67	0.65	15
Paschimottanasana	0.79	1.00	0.88	15
Phalakasana	0.93	0.87	0.90	15
Pincha Mayurasana	0.85	0.73	0.79	15
Salamba Bhujangasana	0.92	0.73	0.81	15
Salamba Sarvangasana	0.93	0.93	0.93	15
Setu Bandha Sarvangasana	0.93	0.87	0.90	15
Sivasana	1.00	1.00	1.00	15
Supta Kapotasana	0.81	0.87	0.84	15
Trikonasana	0.83	1.00	0.91	15
Upavistha Konasana	0.83	0.67	0.74	15
Urdhva Dhanurasana	0.87	0.87	0.87	15
Urdhva Mukha Svsnssana	0.86	0.80	0.83	15
Ustrasana	0.74	0.93	0.82	15
Utkatasana	0.86	0.80	0.83	15
Uttanasana	0.86	0.80	0.83	15
Utthita Hasta Padangusthasana	0.88	0.93	0.90	15
Utthita Parsvakonasana	0.81	0.87	0.84	15
Vasisthasana	0.79	0.73	0.76	15
Virabhadrasana One	0.45	0.60	0.51	15
Virabhadrasana Three	0.94	1.00	0.97	15
Virabhadrasana Two	1.00	0.93	0.97	15
Vrksasana	1.00	0.80	0.89	15
accuracy			0.78	705
macro avg	0.81	0.78	0.78	705
weighted avg	0.81	0.78	0.78	705



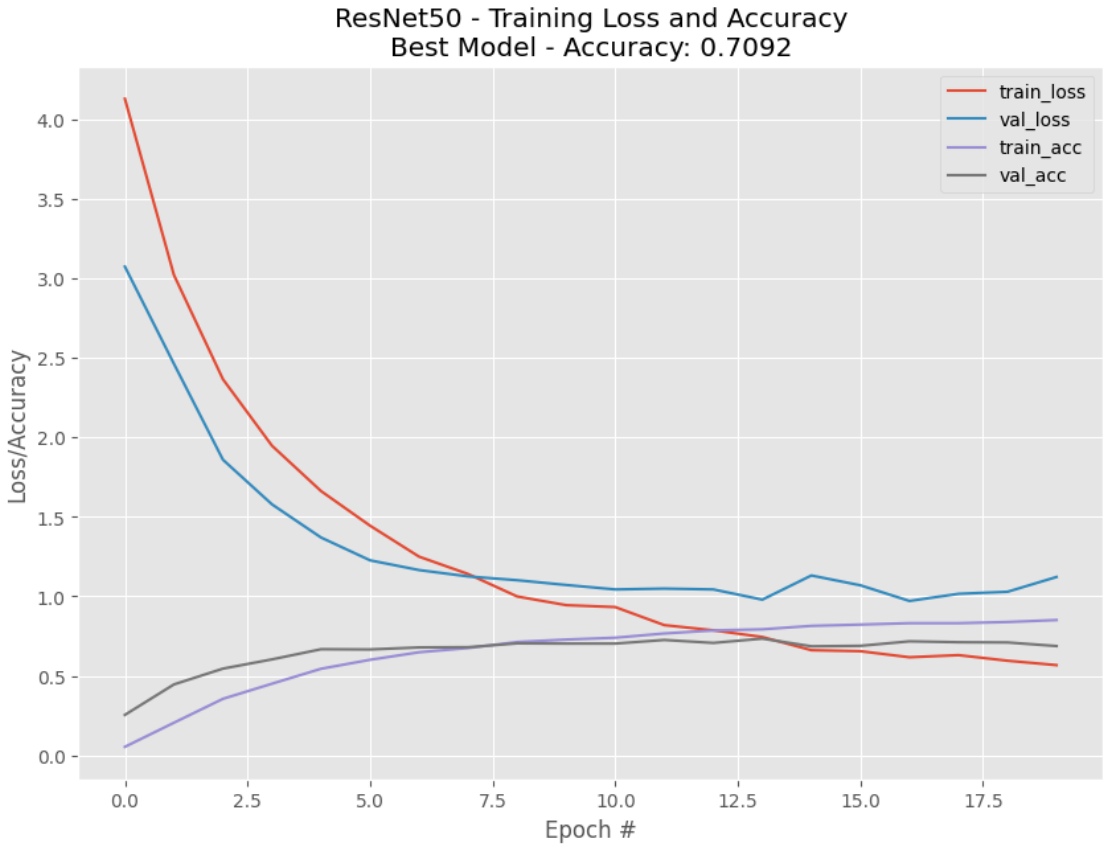
12.4 Gráfica de los mejores modelos

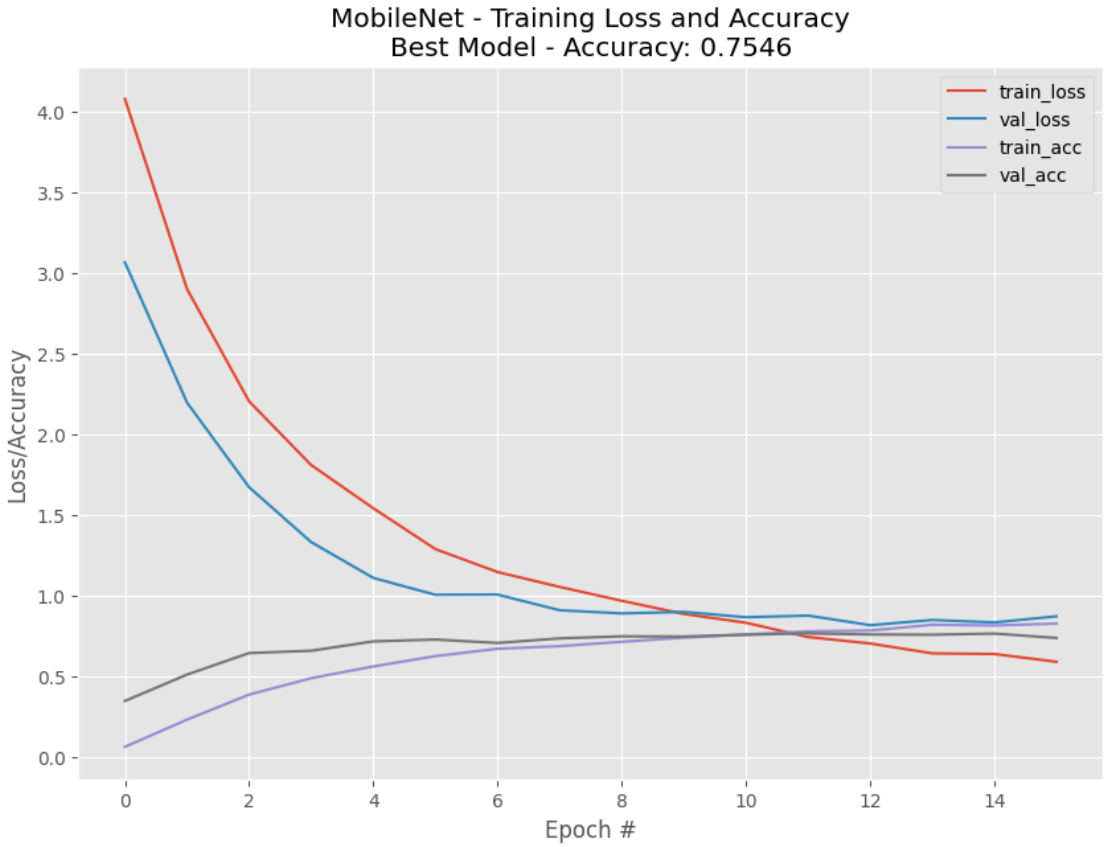
[76]:
```
import numpy as np
import matplotlib.pyplot as plt

plt.style.use("ggplot")

# Crear un DataFrame para almacenar los resultados
results_df = pd.DataFrame(columns=['Model', 'Precision (weighted avg)',␣
 ↪'Precision (macro avg)', 'F1-Score (weighted avg)', 'F1-Score (macro avg)',␣
 ↪'Accuracy'])

# Llenar el DataFrame con los resultados
for model, model_name, test_generator in zip(all_models, all_models_names,␣
 ↪test_generators):
    # Cargar el modelo guardado
    # model = load_model(f'{model_name.lower()}_model.h5')
```

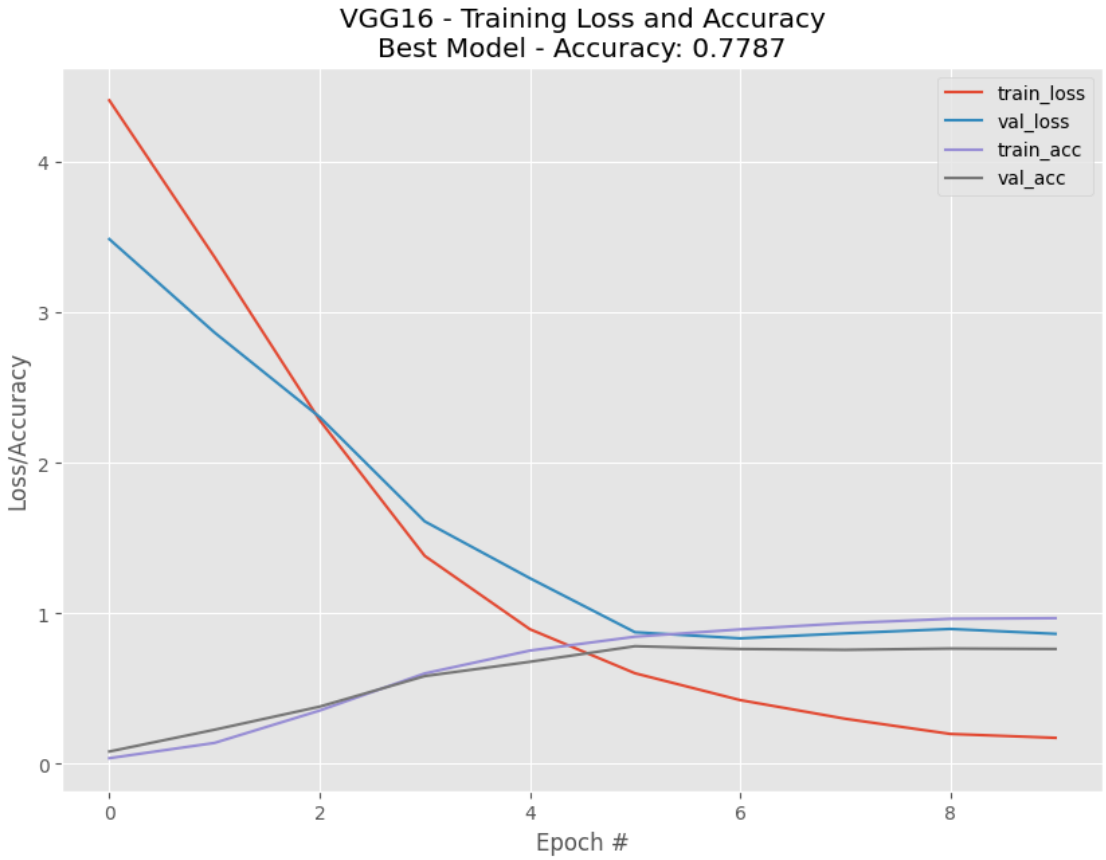
```
    # Obtener las etiquetas verdaderas
    y_true = test_generator.labels

    # Predecir las etiquetas utilizando el modelo
    y_pred = model.predict(test_generator)
    y_pred = np.argmax(y_pred, axis=1)

    # Generar el informe de clasificación
    report = classification_report(y_true, y_pred, zero_division=1,␣
 ↪output_dict=True)

    # Obtener los valores de precision, recall, f1-score y support
    precision_weighted_avg = report['weighted avg']['precision']
    precision_macro_avg = report['macro avg']['precision']
    f1_score_weighted_avg = report['weighted avg']['f1-score']
    f1_score_macro_avg = report['macro avg']['f1-score']

    # Calcular el accuracy
    cm = confusion_matrix(y_true, y_pred)
    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    # Agregar los resultados al DataFrame
    results_df.loc[len(results_df)] = [model_name, precision_weighted_avg,␣
 ↪precision_macro_avg, f1_score_weighted_avg,
                                       f1_score_macro_avg, accuracy]

# Obtener el máximo valor de accuracy para cada modelo
max_accuracy_per_model = results_df.groupby('Model')['Accuracy'].max()

# Filtrar los resultados originales usando los máximos valores de accuracy
filtered_results_df = results_df[results_df.apply(
    lambda row: row['Accuracy'] == max_accuracy_per_model[row['Model']],␣
 ↪axis=1)]

# Mostrar el DataFrame con los resultados filtrados
print(filtered_results_df.to_string(index=False))

# Graficar los resultados del mejor modelo para cada arquitectura
for model_name in set(filtered_results_df['Model']):
    # Obtener el índice del modelo con el mayor accuracy
    best_model_index = filtered_results_df[filtered_results_df['Model'] ==␣
 ↪model_name].index[0]

    plt.figure(figsize=(10, 7))
    plt.plot(history_list[best_model_index].epoch,␣
 ↪history_list[best_model_index].history["loss"], label="train_loss")
```

```
    plt.plot(history_list[best_model_index].epoch,␣
↪history_list[best_model_index].history["val_loss"], label="val_loss")
    plt.plot(history_list[best_model_index].epoch,␣
↪history_list[best_model_index].history["accuracy"], label="train_acc")
    plt.plot(history_list[best_model_index].epoch,␣
↪history_list[best_model_index].history["val_accuracy"], label="val_acc")

    accuracy = filtered_results_df.loc[best_model_index, 'Accuracy']
    plt.title(f"{model_name} - Training Loss and Accuracy\nBest Model -␣
↪Accuracy: {accuracy:.4f}")
    plt.xlabel("Epoch #")
    plt.ylabel("Loss/Accuracy")
    plt.legend()
    plt.show()
```

```
23/23 [==============================] - 4s 179ms/step
23/23 [==============================] - 4s 187ms/step
23/23 [==============================] - 4s 180ms/step
23/23 [==============================] - 4s 176ms/step
23/23 [==============================] - 4s 188ms/step
23/23 [==============================] - 4s 177ms/step
23/23 [==============================] - 4s 177ms/step
23/23 [==============================] - 4s 182ms/step
23/23 [==============================] - 4s 171ms/step
23/23 [==============================] - 4s 185ms/step
23/23 [==============================] - 4s 155ms/step
23/23 [==============================] - 3s 148ms/step
23/23 [==============================] - 4s 152ms/step
23/23 [==============================] - 4s 173ms/step
23/23 [==============================] - 3s 146ms/step
    Model  Precision (weighted avg)  Precision (macro avg)  F1-Score (weighted
avg)  F1-Score (macro avg)  Accuracy
    VGG16                  0.813850               0.813850
0.775939              0.775939  0.778723
 ResNet50                  0.736124               0.736124
0.704948              0.704948  0.709220
MobileNet                  0.791342               0.791342
0.750848              0.750848  0.754610
```


ResNet50 - Training Loss and Accuracy
Best Model - Accuracy: 0.7092
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


MobileNet - Training Loss and Accuracy
Best Model - Accuracy: 0.7546
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
2
4
6
8
10
12
14



13 CONCLUSIONES

Después de evaluar los resultados de los modelos VGG16, ResNet50 y MobileNet, podemos extraer las siguientes conclusiones basadas en los resultados de los tres modelos evaluados (VGG16, ResNet50 y MobileNet):

1. VGG16: El modelo VGG16 alcanza la mayor exactitud de los tres modelos evaluados, con un valor de 0.778723. También muestra un buen desempeño en las métricas de precisión (weighted avg) y puntaje F1 (weighted avg), con valores de 0.813850 y 0.775939 respectivamente. Esto indica que el modelo es capaz de clasificar correctamente la mayoría de las posturas de yoga en el conjunto de prueba.

2. ResNet50: El modelo ResNet50 obtiene una exactitud de 0.709220, que es inferior a la de VGG16 y MobileNet. Sin embargo, aún muestra un rendimiento razonable en las métricas de precisión (weighted avg) y puntaje F1 (weighted avg), con valores de 0.736124 y 0.704948 respectivamente. Esto sugiere que el modelo es capaz de realizar una clasificación aceptable de las posturas de yoga, aunque no alcanza el mismo nivel de precisión que VGG16.

3. MobileNet: El modelo MobileNet obtiene una exactitud de 0.754610, que es ligeramente inferior a la de VGG16 pero mayor que la de ResNet50. Las métricas de precisión (weighted avg) y puntaje F1 (weighted avg) también son relativamente altas, con valores de 0.791342

y 0.750848 respectivamente. Esto indica que el modelo tiene un buen rendimiento en la clasificación de las posturas de yoga, aunque no supera al modelo VGG16 en términos de exactitud.

En conclusión, el modelo VGG16 muestra el mejor rendimiento en la clasificación de posturas de yoga, con una alta exactitud y buenas métricas de precisión y puntaje F1. Sin embargo, tanto ResNet50 como MobileNet también ofrecen un rendimiento razonable, aunque ligeramente inferior en comparación. La elección del mejor modelo dependerá de la importancia relativa que se le dé a la exactitud y a otras métricas de evaluación, así como de las restricciones de recursos computacionales y de memoria disponibles.

[91]:
```
# Mostrar el DataFrame con los resultados filtrados
#print(filtered_results_df.to_string(index=False))
filtered_results_df
```

[91]:

	Model	Precision (weighted avg)	Precision (macro avg)	F1-Score (weighted avg)	F1-Score (macro avg)	Accuracy
4	VGG16	0.813850	0.813850	0.775939	0.775939	0.778723
5	ResNet50	0.736124	0.736124	0.704948	0.704948	0.709220
10	MobileNet	0.791342	0.791342	0.750848	0.750848	0.754610

[87]:
```
%%capture
!sudo apt install texlive-full
```

[98]:
```
%%capture
name_IPYNB_file = '/12MBID10_YépezCallo_JuanCarlos.ipynb'
get_ipython().system(
    "apt update >> /dev/null && apt install texlive-xetex␣
 ↪texlive-fonts-recommended texlive-generic-recommended -y >> /dev/null"
)
get_ipython().system(
    f"jupyter nbconvert --output-dir='{BASE_FOLDER}'␣
 ↪'{BASE_FOLDER}{name_IPYNB_file}' --to pdf"
)
```