Importing Necessary Library

```
# Importing Necessary Libraries
import wandb
from tensorflow.keras.preprocessing.image import ImageDataGenerator
#from keras.applications import resnet
from tensorflow.keras.applications import VGG16, ResNet50, ResNet101,
InceptionResNetV2
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import cv2
import os
```

Loading the Dataset

```
config = {
    'path_dir':
"/kaggle/input/monkeypox-detection-dataset/converted_data",
}
config
```

```
{'path_dir':
'/kaggle/input/monkeypox-detection-dataset/converted_data'}
```

```
# Initialize Weights & Biases
wandb.login(key="e8a360829806e69a22f56a7eb4c7b07aab8c6485")
wandb.init(project="final-project-ablations",
name="amuhairw_monkeypox-detection_run1")

print("[INFO] loading images...")
imagePaths = list(paths.list_images(config['path_dir']))
data = []
labels = []

print(f"Number of image paths loaded: {len(imagePaths)}")
```

```
wandb: Using wandb-core as the SDK backend.  Please refer to https://wandb.me/wandb-core for more information.
wandb: Currently logged in as: amuhairw (delta-group-50). Use `wandb login --relogin` to force relogin
wandb: WARNING If you're specifying your api key in code, ensure this code is not shared publicly.
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wandb login` from the command line.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
```

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

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

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

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

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

```
[INFO] loading images...
Number of image paths loaded: 90
```

```python
# Define paths directly
root_dir = config["path_dir"]
monkeypox_label = "Monkeypox_gray"
monkeypox_path = f"{root_dir}/{monkeypox_label}"
# chickenpox_path = f"config["path_dir"]/chicken_pox"
chickenpox_label = "Chickenpox_grayscale" #"chicken_pox"
chickenpox_path = f"{root_dir}/{chickenpox_label}"

# Function to load images from a directory with a specific label
def load_images_from_folder(folder, label):
    images = []
    labels_list = []
    count = 0

    # Check if directory exists
    if not os.path.exists(folder):
        print(f"Warning: {folder} directory does not exist")
        return images, labels_list

    # Load all images from the directory
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        if os.path.isfile(img_path):
            image = cv2.imread(img_path)
            if image is not None:
                image = cv2.resize(image, (224, 224))  # Resize to match VGG16 input
                images.append(image)
```

```python
                labels_list.append(label)
                count += 1

    print(f"Loaded {count} images from {folder}")
    return images, labels_list

# Load Chickenpox images
chickenpox_images, chickenpox_labels =
load_images_from_folder(chickenpox_path, chickenpox_label)

# Load Monkeypox images
monkeypox_images, monkeypox_labels =
load_images_from_folder(monkeypox_path, monkeypox_label)

# Combine the data
data = chickenpox_images + monkeypox_images
labels = chickenpox_labels + monkeypox_labels

# Convert to numpy arrays
data = np.array(data, dtype="float32") / 255.0  # Normalize pixel
values
labels = np.array(labels)

print(f"Final dataset size: {len(data)} images")
print(f"Unique labels: {np.unique(labels)}")
print(f"Label counts: {[(label, (labels == label).sum()) for label in
np.unique(labels)]}")
```

```
Loaded 47 images from
/kaggle/input/monkeypox-detection-dataset/converted_data/Chickenpox_gr
ayscale
Loaded 43 images from
/kaggle/input/monkeypox-detection-dataset/converted_data/Monkeypox_gra
y
Final dataset size: 90 images
Unique labels: ['Chickenpox_grayscale' 'Monkeypox_gray']
Label counts: [('Chickenpox_grayscale', 47), ('Monkeypox_gray', 43)]
```

Image preprocessing and extract the Label

```python
# Create dictionaries to store data by class
class_data = {monkeypox_label: [], chickenpox_label: []}
for img, lbl in zip(data, labels):
    class_data[lbl].append(img)

# Create train and test sets with specific counts
trainX = []
testX = []
trainY = []
testY = []
```

```
# Monkeypox: 34 train, 9 test
monkey_images = class_data[monkeypox_label]
if len(monkey_images) >= 43:
    monkey_train = monkey_images[:34]
    monkey_test = monkey_images[34:43]
    trainX.extend(monkey_train)
    testX.extend(monkey_test)
    trainY.extend([monkeypox_label] * len(monkey_train))
    testY.extend([monkeypox_label] * len(monkey_test))
else:
    print(f"Warning: Not enough Monkeypox images. Found
{len(monkey_images)}, need 43")

# Chickenpox: 38 train, 9 test
chicken_images = class_data[chickenpox_label]
if len(chicken_images) >= 47:
    chicken_train = chicken_images[:38]
    chicken_test = chicken_images[38:47]
    trainX.extend(chicken_train)
    testX.extend(chicken_test)
    trainY.extend([chickenpox_label] * len(chicken_train))
    testY.extend([chickenpox_label] * len(chicken_test))
else:
    print(f"Warning: Not enough Chickenpox images. Found
{len(chicken_images)}, need 47")

# Convert to numpy arrays
trainX = np.array(trainX, dtype="float32")
testX = np.array(testX, dtype="float32")
trainY = np.array(trainY)
testY = np.array(testY)

print(f"Train data shape: {trainX.shape}, Train labels shape:
{trainY.shape}")
print(f"Test data shape: {testX.shape}, Test labels shape:
{testY.shape}")
```

```
Train data shape: (72, 224, 224, 3), Train labels shape: (72,)
Test data shape: (18, 224, 224, 3), Test labels shape: (18,)
```

```
# Convert labels to categorical (if needed)
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
trainY = to_categorical(trainY)
testY = to_categorical(testY)

# # Initialize the training data augmentation object
# trainAug = ImageDataGenerator(
```

```
#     rotation_range=45,
#     width_shift_range=0.02,
#     height_shift_range=0.02,
#     zoom_range=0.02,
#     horizontal_flip=True,
#     fill_mode="nearest"
# )

baseModel =VGG16(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
#baseModel =ResNet50(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
# baseModel =ResNet101(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
#baseModel =InceptionResNetV2(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)
model = Model(inputs=baseModel.input, outputs=headModel)
for layer in baseModel.layers:
     layer.trainable = False
     # compile our model

# model.summary()

INIT_LR = 1e-3
EPOCHS = 100
BS = 30

import tensorflow as tf
opt = Adam(learning_rate=INIT_LR, decay=INIT_LR / EPOCHS)
# #opt = tf.keras.optimizers.SGD(learning_rate=INIT_LR)
# opt = tf.keras.optimizers.RMSprop(learning_rate=INIT_LR)



# opt = Adam(learning_rate=INIT_LR)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/optimizers/
base_optimizer.py:33: UserWarning: Argument `decay` is no longer
supported and will be ignored.
  warnings.warn(
```

```
# # compile our model
# print("[INFO] compiling model...")
# #opt = Adam(learning_rate=INIT_LR, decay=INIT_LR / EPOCHS)
# model.compile(loss="binary_crossentropy", optimizer=opt,
```

```python
#     metrics=["accuracy"])
# #model.compile(loss="hinge", optimizer=opt,
#     #metrics=["accuracy"])
# # train the head of the network
# print("[INFO] training head...")
# wandb_callback = wandb.keras.WandbMetricsLogger(log_freq="epoch")
# import time
# t1=time.process_time()
# H = model.fit(
#     trainAug.flow(trainX, trainY, batch_size=BS),
#     steps_per_epoch=len(trainX) // BS,
#     validation_data=(testX, testY),
#     validation_steps=len(testX) // BS,
#     epochs=EPOCHS,
#       callbacks=[wandb_callback]
# )
# t2 =time.process_time()
# print("process time:", t2-t1)
# #model.save("vgg16.h5")









# Compile the model
print("[INFO] compiling model...")
model.compile(loss="binary_crossentropy", optimizer=opt,
metrics=["accuracy"])

# # Start timing
# t1 = time.process_time()

# Train the model
print("[INFO] training head...")
wandb_callback = wandb.keras.WandbMetricsLogger(log_freq="epoch")
import time
t1=time.process_time()
H = model.fit(
    x=trainX,
    y=trainY,
    batch_size=BS,
    steps_per_epoch=len(trainX) // BS,
    validation_data=(testX, testY),
    validation_steps=len(testX) // BS,
    epochs=EPOCHS,
     callbacks=[wandb_callback]
```

```
)

# End timing
t2 = time.process_time()
print("Process time:", t2 - t1)
```

```
[INFO] compiling model...
[INFO] training head...
Epoch 1/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 32s 14s/step - accuracy: 0.5222 - loss:
0.8342 - val_accuracy: 0.5000 - val_loss: 0.7498
Epoch 2/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 12s 150ms/step - accuracy: 0.6667 - loss:
0.7194 - val_accuracy: 0.5000 - val_loss: 0.7346
Epoch 3/100

/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out
of data; interrupting training. Make sure that your dataset or
generator can generate at least `steps_per_epoch * epochs` batches.
You may need to use the `.repeat()` function when building your
dataset.
  self.gen.throw(typ, value, traceback)

2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 315ms/step - accuracy: 0.4667 - loss:
0.7756 - val_accuracy: 0.4444 - val_loss: 0.7166
Epoch 4/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 143ms/step - accuracy: 0.5000 - loss:
0.7637 - val_accuracy: 0.3889 - val_loss: 0.7130
Epoch 5/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 314ms/step - accuracy: 0.5333 - loss:
0.7050 - val_accuracy: 0.4444 - val_loss: 0.7081
Epoch 6/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 144ms/step - accuracy: 0.7500 - loss:
0.6825 - val_accuracy: 0.4444 - val_loss: 0.7053
Epoch 7/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 316ms/step - accuracy: 0.5444 - loss:
0.7113 - val_accuracy: 0.4444 - val_loss: 0.7007
Epoch 8/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step - accuracy: 0.5000 - loss:
0.7165 - val_accuracy: 0.5000 - val_loss: 0.6984
Epoch 9/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 315ms/step - accuracy: 0.4667 - loss:
0.7088 - val_accuracy: 0.6111 - val_loss: 0.6941
Epoch 10/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.6667 - loss:
0.6966 - val_accuracy: 0.6111 - val_loss: 0.6929
Epoch 11/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 315ms/step - accuracy: 0.6111 - loss:
0.6609 - val_accuracy: 0.6667 - val_loss: 0.6897
Epoch 12/100
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.6667 - loss:
0.6158 - val_accuracy: 0.6667 - val_loss: 0.6878
Epoch 13/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 312ms/step - accuracy: 0.6111 - loss:
0.6640 - val_accuracy: 0.7222 - val_loss: 0.6849
Epoch 14/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.5000 - loss:
0.7130 - val_accuracy: 0.7222 - val_loss: 0.6833
Epoch 15/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 312ms/step - accuracy: 0.6333 - loss:
0.6726 - val_accuracy: 0.7222 - val_loss: 0.6796
Epoch 16/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 136ms/step - accuracy: 0.5000 - loss:
0.6913 - val_accuracy: 0.7222 - val_loss: 0.6778
Epoch 17/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 310ms/step - accuracy: 0.6556 - loss:
0.6550 - val_accuracy: 0.6667 - val_loss: 0.6741
Epoch 18/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 137ms/step - accuracy: 0.5000 - loss:
0.6628 - val_accuracy: 0.6667 - val_loss: 0.6730
Epoch 19/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 307ms/step - accuracy: 0.5111 - loss:
0.6974 - val_accuracy: 0.6667 - val_loss: 0.6715
Epoch 20/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.8333 - loss:
0.6237 - val_accuracy: 0.7222 - val_loss: 0.6703
Epoch 21/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 309ms/step - accuracy: 0.6333 - loss:
0.6472 - val_accuracy: 0.7222 - val_loss: 0.6672
Epoch 22/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - accuracy: 0.6667 - loss:
0.6768 - val_accuracy: 0.7222 - val_loss: 0.6654
Epoch 23/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 311ms/step - accuracy: 0.5222 - loss:
0.6799 - val_accuracy: 0.7222 - val_loss: 0.6611
Epoch 24/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - accuracy: 0.6667 - loss:
0.6189 - val_accuracy: 0.7222 - val_loss: 0.6586
Epoch 25/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 304ms/step - accuracy: 0.5778 - loss:
0.6582 - val_accuracy: 0.7778 - val_loss: 0.6538
Epoch 26/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step - accuracy: 0.7500 - loss:
0.6065 - val_accuracy: 0.8333 - val_loss: 0.6514
Epoch 27/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 304ms/step - accuracy: 0.6556 - loss:
0.6307 - val_accuracy: 0.7222 - val_loss: 0.6477
Epoch 28/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - accuracy: 0.6667 - loss:
```

```
0.6196 - val_accuracy: 0.7222 - val_loss: 0.6459
Epoch 29/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 307ms/step - accuracy: 0.6889 - loss:
0.6191 - val_accuracy: 0.7778 - val_loss: 0.6426
Epoch 30/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - accuracy: 0.7500 - loss:
0.6429 - val_accuracy: 0.8333 - val_loss: 0.6411
Epoch 31/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 300ms/step - accuracy: 0.6444 - loss:
0.6449 - val_accuracy: 0.8333 - val_loss: 0.6384
Epoch 32/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 137ms/step - accuracy: 1.0000 - loss:
0.5639 - val_accuracy: 0.8333 - val_loss: 0.6361
Epoch 33/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 299ms/step - accuracy: 0.7111 - loss:
0.6160 - val_accuracy: 0.7778 - val_loss: 0.6319
Epoch 34/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 137ms/step - accuracy: 0.9167 - loss:
0.5142 - val_accuracy: 0.7778 - val_loss: 0.6297
Epoch 35/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 301ms/step - accuracy: 0.6778 - loss:
0.6063 - val_accuracy: 0.8333 - val_loss: 0.6260
Epoch 36/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - accuracy: 0.8333 - loss:
0.5787 - val_accuracy: 0.8333 - val_loss: 0.6244
Epoch 37/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 303ms/step - accuracy: 0.6889 - loss:
0.6094 - val_accuracy: 0.8333 - val_loss: 0.6217
Epoch 38/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 137ms/step - accuracy: 0.6667 - loss:
0.5859 - val_accuracy: 0.8333 - val_loss: 0.6205
Epoch 39/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 299ms/step - accuracy: 0.7333 - loss:
0.5824 - val_accuracy: 0.8333 - val_loss: 0.6194
Epoch 40/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.5000 - loss:
0.6837 - val_accuracy: 0.8333 - val_loss: 0.6183
Epoch 41/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 299ms/step - accuracy: 0.6333 - loss:
0.6274 - val_accuracy: 0.7778 - val_loss: 0.6165
Epoch 42/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - accuracy: 0.7500 - loss:
0.6058 - val_accuracy: 0.7778 - val_loss: 0.6158
Epoch 43/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 301ms/step - accuracy: 0.7222 - loss:
0.5917 - val_accuracy: 0.7778 - val_loss: 0.6146
Epoch 44/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.8333 - loss:
0.4968 - val_accuracy: 0.8333 - val_loss: 0.6140
```

```
Epoch 45/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 297ms/step - accuracy: 0.7667 - loss:
0.5907 - val_accuracy: 0.7778 - val_loss: 0.6146
Epoch 46/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.7500 - loss:
0.6068 - val_accuracy: 0.7778 - val_loss: 0.6155
Epoch 47/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 292ms/step - accuracy: 0.7111 - loss:
0.5861 - val_accuracy: 0.7778 - val_loss: 0.6177
Epoch 48/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 130ms/step - accuracy: 0.8333 - loss:
0.5848 - val_accuracy: 0.7778 - val_loss: 0.6180
Epoch 49/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 298ms/step - accuracy: 0.6778 - loss:
0.6139 - val_accuracy: 0.7778 - val_loss: 0.6168
Epoch 50/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 130ms/step - accuracy: 0.8333 - loss:
0.5066 - val_accuracy: 0.7778 - val_loss: 0.6160
Epoch 51/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 295ms/step - accuracy: 0.8333 - loss:
0.5644 - val_accuracy: 0.7222 - val_loss: 0.6130
Epoch 52/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.7500 - loss:
0.6055 - val_accuracy: 0.7778 - val_loss: 0.6117
Epoch 53/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 295ms/step - accuracy: 0.7333 - loss:
0.6029 - val_accuracy: 0.8333 - val_loss: 0.6094
Epoch 54/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - accuracy: 0.8333 - loss:
0.5212 - val_accuracy: 0.8333 - val_loss: 0.6083
Epoch 55/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 295ms/step - accuracy: 0.7667 - loss:
0.5439 - val_accuracy: 0.7778 - val_loss: 0.6055
Epoch 56/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.7500 - loss:
0.5221 - val_accuracy: 0.8333 - val_loss: 0.6051
Epoch 57/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 294ms/step - accuracy: 0.8333 - loss:
0.5287 - val_accuracy: 0.7778 - val_loss: 0.6033
Epoch 58/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 130ms/step - accuracy: 0.8333 - loss:
0.5953 - val_accuracy: 0.7778 - val_loss: 0.6017
Epoch 59/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 291ms/step - accuracy: 0.7444 - loss:
0.5694 - val_accuracy: 0.7778 - val_loss: 0.5981
Epoch 60/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.8333 - loss:
0.5319 - val_accuracy: 0.7778 - val_loss: 0.5959
Epoch 61/100
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 290ms/step - accuracy: 0.6667 - loss:
0.5868 - val_accuracy: 0.7778 - val_loss: 0.5925
Epoch 62/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 1.0000 - loss:
0.3589 - val_accuracy: 0.7778 - val_loss: 0.5909
Epoch 63/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 291ms/step - accuracy: 0.7333 - loss:
0.5504 - val_accuracy: 0.7778 - val_loss: 0.5878
Epoch 64/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.8333 - loss:
0.5171 - val_accuracy: 0.7778 - val_loss: 0.5863
Epoch 65/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 292ms/step - accuracy: 0.6556 - loss:
0.5698 - val_accuracy: 0.8333 - val_loss: 0.5834
Epoch 66/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.7500 - loss:
0.5349 - val_accuracy: 0.8333 - val_loss: 0.5825
Epoch 67/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 290ms/step - accuracy: 0.8000 - loss:
0.4947 - val_accuracy: 0.7778 - val_loss: 0.5816
Epoch 68/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.4167 - loss:
0.6712 - val_accuracy: 0.7778 - val_loss: 0.5813
Epoch 69/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 292ms/step - accuracy: 0.8222 - loss:
0.4926 - val_accuracy: 0.7778 - val_loss: 0.5800
Epoch 70/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - accuracy: 0.6667 - loss:
0.5999 - val_accuracy: 0.7778 - val_loss: 0.5803
Epoch 71/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 289ms/step - accuracy: 0.8000 - loss:
0.4883 - val_accuracy: 0.7778 - val_loss: 0.5801
Epoch 72/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.8333 - loss:
0.5464 - val_accuracy: 0.7778 - val_loss: 0.5794
Epoch 73/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 288ms/step - accuracy: 0.7222 - loss:
0.5413 - val_accuracy: 0.7778 - val_loss: 0.5776
Epoch 74/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.7500 - loss:
0.6149 - val_accuracy: 0.7778 - val_loss: 0.5770
Epoch 75/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 288ms/step - accuracy: 0.8333 - loss:
0.4688 - val_accuracy: 0.7778 - val_loss: 0.5770
Epoch 76/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - accuracy: 0.6667 - loss:
0.6027 - val_accuracy: 0.7778 - val_loss: 0.5760
Epoch 77/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 287ms/step - accuracy: 0.8556 - loss:
```

```
0.4828 - val_accuracy: 0.7778 - val_loss: 0.5743
Epoch 78/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 131ms/step - accuracy: 0.5833 - loss:
0.6933 - val_accuracy: 0.7778 - val_loss: 0.5738
Epoch 79/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 286ms/step - accuracy: 0.7778 - loss:
0.5042 - val_accuracy: 0.7778 - val_loss: 0.5747
Epoch 80/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.5833 - loss:
0.5442 - val_accuracy: 0.7222 - val_loss: 0.5766
Epoch 81/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 284ms/step - accuracy: 0.6889 - loss:
0.5600 - val_accuracy: 0.7222 - val_loss: 0.5783
Epoch 82/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.6667 - loss:
0.5332 - val_accuracy: 0.7222 - val_loss: 0.5790
Epoch 83/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 284ms/step - accuracy: 0.7444 - loss:
0.5096 - val_accuracy: 0.7222 - val_loss: 0.5779
Epoch 84/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - accuracy: 0.6667 - loss:
0.5752 - val_accuracy: 0.7778 - val_loss: 0.5756
Epoch 85/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 285ms/step - accuracy: 0.8111 - loss:
0.4990 - val_accuracy: 0.7778 - val_loss: 0.5710
Epoch 86/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - accuracy: 0.9167 - loss:
0.4465 - val_accuracy: 0.7222 - val_loss: 0.5688
Epoch 87/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 292ms/step - accuracy: 0.8444 - loss:
0.4969 - val_accuracy: 0.7778 - val_loss: 0.5671
Epoch 88/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 131ms/step - accuracy: 0.8333 - loss:
0.5251 - val_accuracy: 0.7778 - val_loss: 0.5666
Epoch 89/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 281ms/step - accuracy: 0.8000 - loss:
0.5044 - val_accuracy: 0.7778 - val_loss: 0.5680
Epoch 90/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - accuracy: 0.7500 - loss:
0.4863 - val_accuracy: 0.8333 - val_loss: 0.5677
Epoch 91/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - accuracy: 0.8333 - loss:
0.4770 - val_accuracy: 0.7778 - val_loss: 0.5640
Epoch 92/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - accuracy: 0.7500 - loss:
0.5125 - val_accuracy: 0.7778 - val_loss: 0.5625
Epoch 93/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 282ms/step - accuracy: 0.7556 - loss:
0.4958 - val_accuracy: 0.7778 - val_loss: 0.5581
```

```
Epoch 94/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.9167 - loss:
0.4622 - val_accuracy: 0.7778 - val_loss: 0.5583
Epoch 95/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 283ms/step - accuracy: 0.8111 - loss:
0.4746 - val_accuracy: 0.7778 - val_loss: 0.5588
Epoch 96/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 127ms/step - accuracy: 0.8333 - loss:
0.4081 - val_accuracy: 0.7778 - val_loss: 0.5594
Epoch 97/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 284ms/step - accuracy: 0.7444 - loss:
0.5301 - val_accuracy: 0.7778 - val_loss: 0.5589
Epoch 98/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.9167 - loss:
0.4358 - val_accuracy: 0.7778 - val_loss: 0.5577
Epoch 99/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 282ms/step - accuracy: 0.7889 - loss:
0.4566 - val_accuracy: 0.7778 - val_loss: 0.5554
Epoch 100/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - accuracy: 0.8333 - loss:
0.4224 - val_accuracy: 0.7778 - val_loss: 0.5538
Process time: 77.96895648899999
```

```
print(len(H.history['loss']))
```

```
100
```

```
model.save_weights("my_model_weights.weights.h5")

# Log model weights to W&B as an artifact
artifact = wandb.Artifact("monkeypox-model", type="model")
artifact.add_file("my_model_weights.weights.h5")
wandb.log_artifact(artifact)

#classification report on training
predIdxs = model.predict(trainX)
# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
trainpredict = np.argmax(predIdxs, axis=1)
print(classification_report(trainY.argmax(axis=1), trainpredict,
      target_names=lb.classes_))
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 23s 3s/step
                      precision    recall  f1-score   support

Chickenpox_grayscale       0.79      0.89      0.84        38
      Monkeypox_gray       0.86      0.74      0.79        34

            accuracy                           0.82        72
           macro avg       0.83      0.82      0.82        72
```

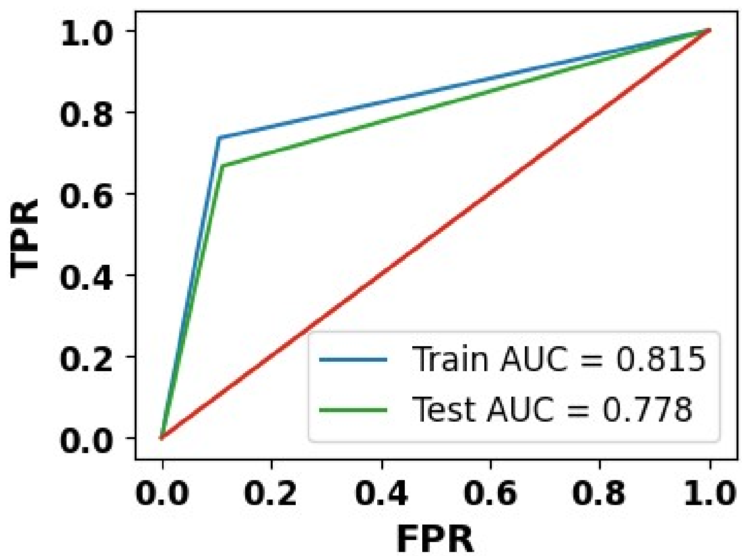
```
        weighted avg       0.82      0.82      0.82        72


# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(trainY.argmax(axis=1), trainpredict)
total = sum(sum(cm))
print(cm)
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])
# show the confusion matrix, accuracy, sensitivity, and specificity
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))

# Log metrics to W&B
wandb.log({
    "train_accuracy": acc,
    "train_sensitivity": sensitivity,
    "train_specificity": specificity
})

[[34  4]
 [ 9 25]]
acc: 0.8194
sensitivity: 0.8947
specificity: 0.7353

# make predictions on the testing set
print("[INFO] evaluating network...")
predIdys = model.predict(testX, batch_size=BS)
testpredict = np.argmax(predIdys, axis=1)
print(classification_report(testY.argmax(axis=1), testpredict,
     target_names=lb.classes_))

[INFO] evaluating network...
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 319ms/step
                      precision    recall  f1-score   support

Chickenpox_grayscale       0.73      0.89      0.80         9
      Monkeypox_gray       0.86      0.67      0.75         9

            accuracy                           0.78        18
           macro avg       0.79      0.78      0.77        18
        weighted avg       0.79      0.78      0.77        18


# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(testY.argmax(axis=1), testpredict)
```

```python
total = sum(sum(cm))
print(cm)
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])
# show the confusion matrix, accuracy, sensitivity, and specificity
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))

# Log test metrics to W&B
wandb.log({
    "test_accuracy": acc,
    "test_sensitivity": sensitivity,
    "test_specificity": specificity
})
```

```
[[8 1]
 [3 6]]
acc: 0.7778
sensitivity: 0.8889
specificity: 0.6667
```

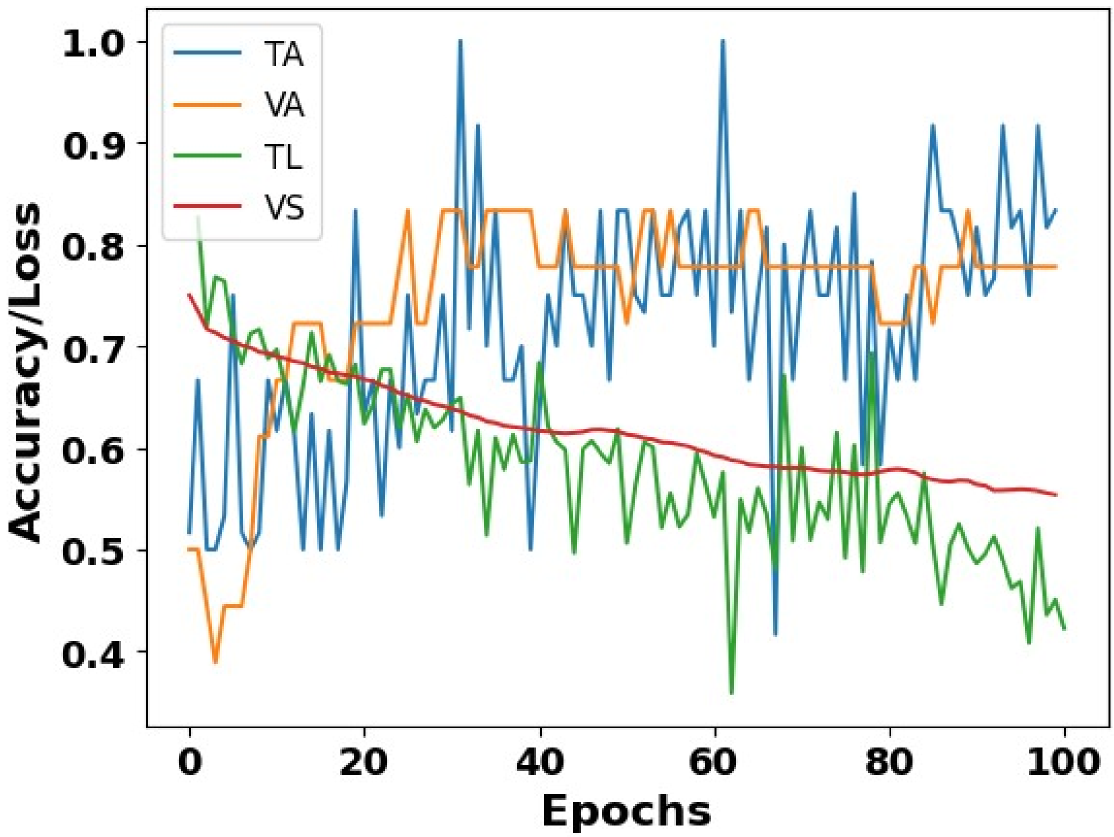
```python
from sklearn import metrics
from sklearn.metrics import roc_auc_score, auc
from sklearn.metrics import roc_curve
fig = plt.figure(figsize = (4, 3))
fpr1,tpr1,_=roc_curve(np.argmax(trainY, axis=1),np.argmax(predIdxs,
axis=1))
fpr2,tpr2,_=roc_curve(np.argmax(testY, axis=1),np.argmax(predIdys,
axis=1))
area_under_curve1=auc(fpr1,tpr1)
random_probs=[0 for i in range(len(trainY.ravel()))]
p_fpr1,p_tpr1,threshold=roc_curve(trainY.ravel(),random_probs,
pos_label=1)
plt.plot(fpr1,tpr1, label='Train AUC =
{:.3f}'.format(area_under_curve1))
plt.plot(p_fpr1, p_tpr1)
area_under_curve2=auc(fpr2,tpr2)
random_probs2=[0 for i in range(len(testY.ravel()))]
p_fpr2,p_tpr2,threshold=roc_curve(testY.ravel(),random_probs2,
pos_label=1)
plt.plot(fpr2,tpr2, label='Test AUC =
{:.3f}'.format(area_under_curve2))
plt.plot(p_fpr2, p_tpr2)

    # x label
plt.xlabel('FPR',fontsize=14, fontdict=dict(weight='bold'))
    # y label
plt.ylabel('TPR', fontsize=14, fontdict=dict(weight='bold'))
```

```
plt.xticks( rotation=0, weight = 'bold', )
plt.yticks( rotation=0, weight = 'bold')
plt.tick_params(rotation=0,axis='y', labelsize=12)
plt.tick_params(rotation=0,axis='x', labelsize=12)
plt.legend()
plt.legend(prop={'size':12})
plt.savefig('ROC',dpi=200, bbox_inches='tight')
plt.show();

# Log ROC curve to W&B
wandb.log({"ROC_curve": wandb.Image(plt)})
```




```
<Figure size 640x480 with 0 Axes>


from matplotlib.ticker import FormatStrFormatter

import matplotlib
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

N=len(H.history['loss'])
fig, ax = plt.subplots()
#plt.rcParams["font.family"] = "serif"
acc = H.history['accuracy']
val_acc = H.history['val_accuracy']
#font={'size':10}
#matplotlib.rc('font',**font)
loss = H.history['loss']
val_loss = H.history['val_loss']
epochs = range(1, len(acc) + 1)
```

```
plt.plot(np.arange(0, N), H.history["accuracy"], label="TA")
#plt.plot(epochs, loss, label='Training loss')
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="VA")
plt.plot(epochs, loss, label='TL')
#plt.plot(np.arange(0, N), H.history["val_accuracy"],
label="validation accuracy")
plt.plot(np.arange(0, N), val_loss, label='VS')
#plt.plot(epochs, acc, 'bo', label='Training acc')
#plt.plot(1, val_acc, 'b', label='Validation acc')
#plt.title('Training and validation accuracy')
plt.xlabel('Epochs',fontsize=16, fontdict=dict(weight='bold'))
    # y label
plt.ylabel('Accuracy/Loss', fontsize=16, fontdict=dict(weight='bold'))
#plt.ylabel("Accuracy")
#plt.xlabel("Epochs")
plt.xticks( rotation=0, weight = 'bold' )
plt.yticks( rotation=0, weight = 'bold')
plt.tick_params(rotation=0,axis='y', labelsize=14)
plt.tick_params(rotation=0,axis='x', labelsize=14)
#plt.grid('white')
#plt.grid(axis='x', color='0.95')
plt.legend(loc='best')
plt.legend(prop={'size':12})
#fig = plt.figure(figsize = (4, 3))
plt.savefig('ACC',dpi=200, bbox_inches='tight')
plt.show()

# Log accuracy/loss plot to W&B
wandb.log({"Accuracy_Loss_Plot": wandb.Image(plt)})

# Finish W&B run
wandb.finish()
```




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

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<Figure size 640x480 with 0 Axes>
```