```python
import cv2
import os
import numpy as np
from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten,
Dense, LSTM, SimpleRNN
from keras.optimizers import Adam
import tensorflow as tf
from sklearn.metrics import accuracy_score, classification_report
from tensorflow.keras import layers, models, optimizers
import keras
# from tensorflow.keras.applications import EfficientNetB0, VGG16
from tensorflow.keras import *
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from skimage import io
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
from tensorflow.keras.applications import VGG16, ResNet50,
InceptionV3, Xception, EfficientNetB0, EfficientNetB7
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
load_img, img_to_array
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Precision, Recall
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os
# from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
import tensorflow as tf
# from tensorflow.keras.applications import VGG16, ResNet50,
InceptionV3, Xception
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
load_img, img_to_array
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
import sklearn
from tensorflow.keras.metrics import Precision, Recall
# from sklearn.model_selection import GridSearchCV
from tensorflow.keras.callbacks import ModelCheckpoint,
ReduceLROnPlateau

from sklearn.metrics import confusion_matrix, classification_report
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os
from sklearn.utils import class_weight
from tensorflow.keras.callbacks import ReduceLROnPlateau,
EarlyStopping
from tensorflow.keras import regularizers
from tensorflow.keras.applications.efficientnet import
preprocess_input as effnet_preprocess
```

```
2025-06-08 00:05:40.658694: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to
register cuFFT factory: Attempting to register factory for plugin
cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
E0000 00:00:1749341140.863863      35 cuda_dnn.cc:8310] Unable to
register cuDNN factory: Attempting to register factory for plugin
cuDNN when one has already been registered
E0000 00:00:1749341140.929232      35 cuda_blas.cc:1418] Unable to
register cuBLAS factory: Attempting to register factory for plugin
cuBLAS when one has already been registered
```

```
train_data_dir =
'/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train'
test_data_dir = '/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/test'
image_size = (224, 224)
image_size2 = (600, 600) # EfficientNetB7
batch_size = 16

train_datagen = ImageDataGenerator(
        preprocessing_function=effnet_preprocess,
        rotation_range=20,     #Random rotation between 0 and 45
        width_shift_range=0.2,   #% shift
        brightness_range=[0.8, 1.2],      # New: Random brightness
        channel_shift_range=10.,          # New: Channel shift
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='constant',
        cval=125,
        # rescale=1.0/255.0,
        validation_split=0.25
    # 20% of the ENTIRE dataset will be used for validation (80%
train, 20% test)
    # => (.75*80, 0.25*80, 20 )=> (60 train, 20  validation, 20 test)
    )
```

```
val_datagen = ImageDataGenerator(
    preprocessing_function=effnet_preprocess,
    validation_split=0.25
)
test_datagen =
ImageDataGenerator(preprocessing_function=effnet_preprocess,)

# Create ImageDataGenerators for training and validation with data
augmentation
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='training'  # Use the 'training' subset
)

# Load the validation data
validation_generator = val_datagen.flow_from_directory(
    train_data_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation'  # Use the 'validation' subset
)


# Load the data
test_generator = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical'
)
###################### EfficientNetB7
batch_size = 8
# Create ImageDataGenerators for training and validation with data
augmentation
train_generator2 = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=image_size2,
    batch_size=batch_size,
    class_mode='categorical',
    subset='training'  # Use the 'training' subset
)

# Load the validation data
validation_generator2 = val_datagen.flow_from_directory(
    train_data_dir,
    target_size=image_size2,
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation'  # Use the 'validation' subset
)


# Load the data
test_generator2 = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=image_size2,
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 1214 images belonging to 3 classes.
Found 404 images belonging to 3 classes.
Found 403 images belonging to 3 classes.
Found 1214 images belonging to 3 classes.
Found 404 images belonging to 3 classes.
Found 403 images belonging to 3 classes.
```

```
X_train, y_train = next(train_generator)
X_val, y_val = next(validation_generator)
X_test, y_test = next(test_generator)

total = 0
for class_name in os.listdir(train_data_dir):
    class_path = os.path.join(train_data_dir, class_name)
    if os.path.isdir(class_path):
        n = len([fname for fname in os.listdir(class_path)
                 if fname.lower().endswith(('.jpg', '.jpeg', '.png',
'.bmp'))])
        print(f"{class_name}: {n} images")
        total += n
print(f"Total images before augmentation: {total}")
```

```
CONTROL: 614 images
PD: 204 images
AD: 800 images
Total images before augmentation: 1618
```

```
print(f"Train images used (after augmentation):
{train_generator.samples}")
```

```
Train images used (after augmentation): 1214
```

```
def unfreeze_top_blocks(base_model, model_name):
    for layer in base_model.layers:
        layer.trainable = False  # Freeze all first

    if model_name == 'VGG16':
        for layer in base_model.layers:
            if layer.name.startswith('block5_') or
layer.name.startswith('block4_'):
                layer.trainable = True
    elif model_name == 'ResNet50':
        for layer in base_model.layers:
            if layer.name.startswith('conv5_') or
layer.name.startswith('conv4_'):
                layer.trainable = True
    elif model_name == 'InceptionV3':
        for layer in base_model.layers:
            if 'mixed7' in layer.name or 'mixed8' in layer.name or
'mixed9' in layer.name:
                layer.trainable = True
    elif model_name == 'Xception':
        for layer in base_model.layers:
            if 'block13' in layer.name or 'block14' in layer.name:
                layer.trainable = True
    elif model_name == 'EfficientNetB7' or model_name ==
'EfficientNetB0':
        for layer in base_model.layers:
            if 'block6' in layer.name or 'block7' in layer.name:
                layer.trainable = True
```

Note: new and old functions below

I created a new **"build_and_train_model()"** function but also left the old one (just extended the name with "_old") below. The new function does the 2 step process of training and also freezes 80% of the layers of each model - allowing you to fine-tune 20% of the layers. You could always change this to get more specific with each model, but this is a solid generalized approach that I think works.

```
initial_epochs= 10
def build_and_train_model_two_phase(
    model_name,
    base_model,
    train_generator,
    validation_generator,
    initial_epochs=initial_epochs,      # train head for 5–10 epochs
    fine_tune_epochs= 80,    # then fine-tune for approx 10–15 more (or
even more than that?)
    initial_lr=1e-4,
    fine_tune_lr=3e-5,
    dropout_rate=0.7
):
    """
    Two-phase training:
      Phase 1: Freeze ~80% of base_model, train only head for
```

```
initial_epochs.
      Phase 2: Unfreeze last blocks of base_model, recompile with
lower LR, continue training.
    Returns: (fine-tuned_model, combined_history)
    """

    # -------------------------------------------------------------
    # STEP A: Freeze most of the base_model for Phase 1
    # -------------------------------------------------------------
    # 1) freeze the earliest 60 % of layers in base_model.
    print("here1")
    for layer in base_model.layers:
        layer.trainable = False

    # now - base_model has 80% of its layers frozen, 20% trainable!

    # -------------------------------------------------------------
    # STEP B: Build & compile the full model (head + base)
    # -------------------------------------------------------------
    print("here2")
    x = Flatten()(base_model.output)                        # Flatten
the feature map
    x = Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.001))(x)                    # One
hidden Dense layer
    x = Dropout(dropout_rate)(x)                            # Dropout
for regularization
    output = Dense(3, activation='softmax')(x)              # 3-way
softmax for AD/PD/Control

    model = Model(inputs=base_model.input, outputs=output)

    # Compile with a “standard” learning rate for the head (Phase 1)
    print("here3")
    model.compile(
        optimizer=Adam(learning_rate=initial_lr),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(), Recall()]
    )

    # -------------------------------------------------------------
    # STEP C: Compute class_weights
    # -------------------------------------------------------------
    print("here4")
    classes = train_generator.classes
    unique_classes = np.unique(classes)  # e.g. array([0,1,2])

    weights = class_weight.compute_class_weight(
        class_weight='balanced',
        classes=unique_classes,
        y=classes
    )
    class_weights = dict(zip(unique_classes, weights))

    # -----------------------------------------------------------
    # STEP D: Phase 1 – Train only the head (base_model is mostly
frozen)
    # -----------------------------------------------------------
   # Learning rate reducer
    print("here5")
    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.3,
        patience=2,
        min_lr=1e-7,
        verbose=2
    )

    # Early stopping
    early_stop = EarlyStopping(
        monitor='val_loss',
        patience=12,
        restore_best_weights=True,
        verbose=2
    )
    print(">>> PHASE 1: Training head (with ~80% of base frozen) <<<")
    print("here6")
    history_phase1 = model.fit(
        train_generator,
        steps_per_epoch = train_generator.samples //
train_generator.batch_size,
        validation_data   = validation_generator,
        validation_steps  = validation_generator.samples //
validation_generator.batch_size,
        epochs            = initial_epochs,
        callbacks         = [reduce_lr, early_stop],
        class_weight      = class_weights,
        verbose           = 2
    )


    # -----------------------------------------------------------
    # STEP E: Phase 2 – Unfreeze the final blocks of base_model
    # -----------------------------------------------------------
    print("\n>>> PHASE 2: Fine-tuning top layers <<<")

    # (a) Freeze everything again first, then unfreeze only the final
blocks by name/index.
    for layer in base_model.layers:
        layer.trainable = False
    unfreeze_top_blocks(base_model, model_name)
    # ** Here is where you unfreeze only the “last blocks” (approx 20%
of layers) **
    # You can either do:
    #   (1) Replace the condition below with the exact block-name rule
for your architecture.
    #   For example, if this is ResNet50, you might do:
    #       if layer.name.startswith('conv5_') or
layer.name.startswith('conv4_'): layer.trainable=True
    #
    #   -or-
    #   (2) for simplicity - we’ll unfreeze everything from index N
onward again:

    # (b) recompile with a lower learning rate (to avoid wrecking the
pretrained weights)
    model.compile(
        optimizer=Adam(learning_rate=fine_tune_lr),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(), Recall()]
    )

    # (c) continue training (starting from where Phase 1 left off) -
with same class_weight
    history_phase2 = model.fit(
        train_generator,
        steps_per_epoch = train_generator.samples //
train_generator.batch_size,
        validation_data = validation_generator,
        validation_steps = validation_generator.samples //
validation_generator.batch_size,
        epochs = initial_epochs + fine_tune_epochs,
        initial_epoch = initial_epochs,  # resume from Phase 1
        callbacks = [reduce_lr, early_stop],
        class_weight = class_weights,
        verbose = 2
    )

    # ----------------------------------------------------------
    # STEP F: Combine the histories so we can plot later if desired
    # ----------------------------------------------------------
    combined_history = {
        'loss': history_phase1.history['loss'] +
history_phase2.history['loss'],
        'val_loss': history_phase1.history['val_loss'] +
history_phase2.history['val_loss'],
        'accuracy': history_phase1.history.get('accuracy', []) +
history_phase2.history.get('accuracy', []),
        'val_accuracy': history_phase1.history.get('val_accuracy', [])
```

```
+ history_phase2.history.get('val_accuracy', []),
        'precision': history_phase1.history.get('precision', []) +
history_phase2.history.get('precision', []),
        'val_precision': history_phase1.history.get('val_precision',
[]) + history_phase2.history.get('val_precision', []),
        'recall': history_phase1.history.get('recall', []) +
history_phase2.history.get('recall', []),
        'val_recall': history_phase1.history.get('val_recall', []) +
history_phase2.history.get('val_recall', []),
    }

    return model, combined_history

def build_and_train_model_old(base_model, train_generator,
validation_generator, epochs=50,learning_rate=0.0001,
dropout_rate=0.5):
    # Freeze the convolutional base
    # base_model.trainable = False
    # for layer in base_model.layers:
    #     layer.trainable = False
    N = 3
    for layer in base_model.layers[:N]:
        layer.trainable = False
    for layer in base_model.layers[N:]:
        layer.trainable = True

    # Add custom classification layers
    x = Flatten()(base_model.output)
    x = Dense(256, activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    output = Dense(3, activation='softmax')(x)  # 3 classes:
Alzheimer's, Parkinson's, Control

    # Create the model
    model = Model(inputs=base_model.input, outputs=output)

    # Compile the model
    learning_rate1 = float(1e-4)

    model.compile(
        optimizer=Adam(learning_rate=learning_rate1),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(), Recall()]
    )


    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',    # or 'val_accuracy'
        factor=0.5,            # reduce by half
        patience=3,            # wait this many epochs with no
```

```
improvement
        min_lr=1e-7,           # never lower than this
        verbose=1
    )

    from sklearn.utils import class_weight
    import numpy as np


    classes = train_generator.classes
    class_labels = np.unique(classes)
    weights = class_weight.compute_class_weight(
        class_weight='balanced',
        classes=class_labels,
        y=classes
    )
    class_weights = dict(zip(class_labels, weights))

    history = model.fit(
        train_generator,
        steps_per_epoch=train_generator.samples // batch_size,
        validation_steps=validation_generator.samples // batch_size,
        validation_data=validation_generator,
        epochs=epochs,
        callbacks=[reduce_lr],
        class_weight=class_weights   # <-- singular
    )
    return model, history

from tensorflow.keras.callbacks import Callback

class TerminateOnMetric(Callback):
    def __init__(self, monitor='accuracy', threshold=0.80):
        super().__init__()
        self.monitor = monitor
        self.threshold = threshold

    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        metric_value = logs.get(self.monitor)
        if metric_value is not None:
            if metric_value >= self.threshold:
                print(f"\nReached {self.monitor} = {metric_value:.4f},
stopping training!")
                self.model.stop_training = True

# Example usage:
callback = TerminateOnMetric(monitor='accuracy', threshold=0.80)
```

```
 pretrained_models = {
    'VGG16': VGG16(include_top=False, input_shape=(224, 224, 3),
weights='imagenet', pooling='avg'),
    'ResNet50': ResNet50(include_top=False, input_shape=(224, 224, 3),
weights='imagenet', pooling='avg'),
    'InceptionV3': InceptionV3(include_top=False, input_shape=(224,
224, 3), weights='imagenet', pooling='avg'),
    'Xception': Xception(include_top=False, input_shape=(224, 224, 3),
weights='imagenet', pooling='avg'),
    'EfficientNetB0': EfficientNetB0(include_top=False,
input_shape=(224, 224, 3), weights='imagenet', pooling='avg'),
    'EfficientNetB7': EfficientNetB7(include_top=False,
input_shape=(600, 600, 3), weights='imagenet', pooling='avg'),

}

# Store the trained models and their histories
trained_models = {}
histories = {}

for model_name, base_model in pretrained_models.items():
    print(f"Training model: {model_name}")
    if (model_name == 'EfficientNetB7'): model, history =
build_and_train_model_two_phase(model_name, base_model,
train_generator2, validation_generator2)
    if (model_name != 'EfficientNetB7'): model, history =
build_and_train_model_two_phase(model_name, base_model,
train_generator, validation_generator)
    trained_models[model_name] = model
    histories[model_name] = history
```

```
I0000 00:00:1749341155.532996      35 gpu_device.cc:2022] Created
device /job:localhost/replica:0/task:0/device:GPU:0 with 15513 MB
memory:  -> device: 0, name: Tesla P100-PCIE-16GB, pci bus id:
0000:00:04.0, compute capability: 6.0

Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/inception_v3/
inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
Training model: InceptionV3
here1
here2
here3
here4
here5
>>> PHASE 1: Training head (with ~80% of base frozen) <<<
here6
```

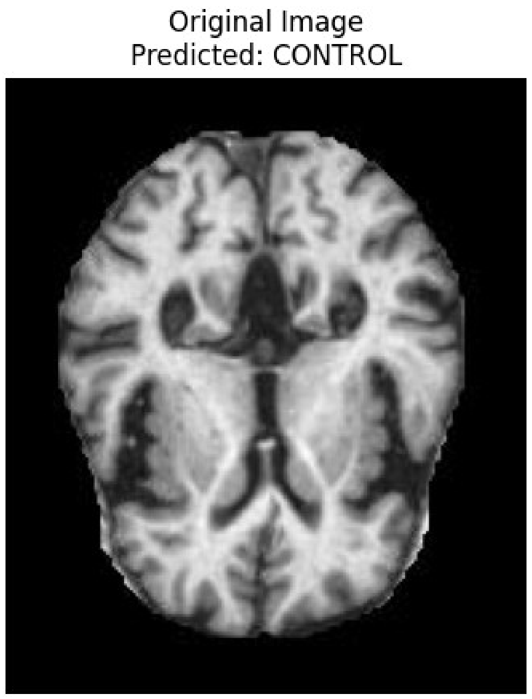
```
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/
data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset`
class should call `super().__init__(**kwargs)` in its constructor.
`**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()

Epoch 1/2

WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1749341175.609458     136 service.cc:148] XLA service
0x78ae00002220 initialized for platform CUDA (this does not guarantee
that XLA will be used). Devices:
I0000 00:00:1749341175.610469     136 service.cc:156]   StreamExecutor
device (0): Tesla P100-PCIE-16GB, Compute Capability 6.0
I0000 00:00:1749341177.506546     136 cuda_dnn.cc:529] Loaded cuDNN
version 90300
I0000 00:00:1749341183.262067     136 device_compiler.h:188] Compiled
cluster using XLA!  This line is logged at most once for the lifetime
of the process.

75/75 - 52s - 687ms/step - accuracy: 0.4950 - loss: 18.0879 -
precision: 0.4950 - recall: 0.4950 - val_accuracy: 0.7675 - val_loss:
3.5330 - val_precision: 0.7675 - val_recall: 0.7675 - learning_rate:
1.0000e-04
Epoch 2/2

/usr/local/lib/python3.11/dist-packages/keras/src/trainers/
epoch_iterator.py:107: 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._interrupted_warning()

75/75 - 1s - 11ms/step - accuracy: 0.6875 - loss: 7.8686 - precision:
0.6875 - recall: 0.6875 - val_accuracy: 0.7450 - val_loss: 4.0609 -
val_precision: 0.7450 - val_recall: 0.7450 - learning_rate: 1.0000e-04
Restoring model weights from the end of the best epoch: 1.

>>> PHASE 2: Fine-tuning top layers <<<
Epoch 3/4
75/75 - 44s - 585ms/step - accuracy: 0.6352 - loss: 6.1902 -
precision_1: 0.6363 - recall_1: 0.6352 - val_accuracy: 0.8500 -
val_loss: 1.5917 - val_precision_1: 0.8500 - val_recall_1: 0.8500 -
learning_rate: 3.0000e-05
Epoch 4/4
75/75 - 1s - 13ms/step - accuracy: 0.5000 - loss: 7.0118 -
precision_1: 0.5000 - recall_1: 0.5000 - val_accuracy: 0.8525 -
```

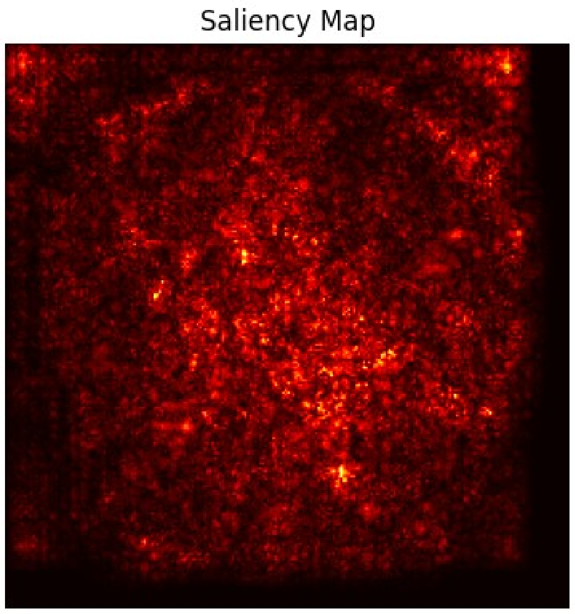
```
val_loss: 1.5239 - val_precision_1: 0.8525 - val_recall_1: 0.8525 -
learning_rate: 3.0000e-05
Restoring model weights from the end of the best epoch: 4.
```

```
 def evaluate_model(model, test_generator):
    # Evaluate the model
    test_loss, test_accuracy, test_precision, test_recall =
model.evaluate(test_generator)
    test_f1 = 2 * (test_precision * test_recall) / (test_precision +
test_recall + 1e-2)

    # Predict the classes for validation data
    test_preds = model.predict(test_generator)
    test_preds = np.argmax(test_preds, axis=1)
    test_true = test_generator.classes

    # Generate confusion matrix
    cm = confusion_matrix(test_true, test_preds)

    return test_accuracy, test_precision, test_recall, test_f1, cm

results = []
test_accuracies = []
for model_name, model in trained_models.items():
    print(f"Evaluating model: {model_name}")
    if (model_name != 'EfficientNetB7'): test_accuracy,
test_precision, test_recall, test_f1, cm = evaluate_model(model,
test_generator)
    if (model_name == 'EfficientNetB7'): test_accuracy,
test_precision, test_recall, test_f1, cm = evaluate_model(model,
test_generator2)
    results.append({
        'Model': model_name,
        'Accuracy': test_accuracy,
        'Precision': test_precision,
        'Recall': test_recall,
        'F1 Score': test_f1,
        'Confusion Matrix': cm
    })
    test_accuracies.append(test_accuracy)

print(results)
```

```
Evaluating model: InceptionV3
26/26 ━━━━━━━━━━━━━━━━━━━━ 6s 238ms/step - accuracy: 0.6521 - loss:
4.2733 - precision_1: 0.6521 - recall_1: 0.6521
26/26 ━━━━━━━━━━━━━━━━━━━━ 13s 268ms/step
[{'Model': 'InceptionV3', 'Accuracy': 0.6848635077476501, 'Precision':
0.6848635077476501, 'Recall': 0.6848635077476501, 'F1 Score':
0.6798997468003312, 'Confusion Matrix': array([[113,  62,  25],
       [ 87,  50,  16],
       [ 25,  16,   9]])}]

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import load_img,
img_to_array

# Example: replace with your actual class order
class_names = ['AD', 'CONTROL', 'PD']

def generate_saliency_map(model, img_path, class_idx, image_size=(224,
224)):
    img = load_img(img_path, target_size=image_size)
    img_array = img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = img_array / 255.0  # Normalize the image

    import tensorflow as tf
    with tf.GradientTape() as tape:
        inputs = tf.cast(img_array, tf.float32)
        tape.watch(inputs)
        predictions = model(inputs)
        loss = predictions[:, class_idx]
    grads = tape.gradient(loss, inputs)
    abs_grads = tf.abs(grads)
    saliency = tf.reduce_max(abs_grads, axis=-1)
    saliency = saliency[0]
    saliency = (saliency - tf.reduce_min(saliency)) /
(tf.reduce_max(saliency) - tf.reduce_min(saliency) + 1e-10)
    return saliency

def plot_saliency_map(img_path, saliency, predicted_class_name):
    img = load_img(img_path)
    img = np.array(img)
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(img)
    plt.title(f'Original Image\nPredicted: {predicted_class_name}')
    plt.axis('off')
    plt.subplot(1, 2, 2)
    plt.imshow(saliency, cmap='hot')
    plt.title('Saliency Map')
    plt.axis('off')
    plt.show()

# Example test images (replace with your own paths)
test_images = [

'/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/AD/AD_10.png',
'/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/CONTROL/CONTROLA
D_10.png',

'/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/PD/PD_10.png'
]

image_size = (224, 224)  # or your input size

# for model_name, model in trained_models.items():
#     print(f"\n=== Model: {model_name} ===")
#     for img_path in test_images:
#         if (model_name != 'EfficientNetB7'): img_arr =
img_to_array(load_img(img_path, target_size=image_size)).reshape(1,
224, 224, 3) / 255.0
#         if (model_name == 'EfficientNetB7'): img_arr =
img_to_array(load_img(img_path, target_size=image_size2)).reshape(1,
600, 600, 3) / 255.0
#         preds = model.predict(img_arr)
#         predicted_class_idx = np.argmax(preds[0])
#         predicted_class_name = class_names[predicted_class_idx]
#         print(f"Image: {img_path}")
#         print(f"Predicted class: {predicted_class_name} (index:
{predicted_class_idx}), Probabilities: {preds[0]}")
#         saliency = generate_saliency_map(model, img_path,
predicted_class_idx, image_size=image_size)
#         plot_saliency_map(img_path, saliency, predicted_class_name)

for model_name, model in trained_models.items():
    print(f"\n=== Model: {model_name} ===")
    for img_path in test_images:
        if model_name == 'EfficientNetB7':
            curr_size = image_size2  # (600, 600)
        else:
            curr_size = image_size   # (224, 224)
        img_arr = img_to_array(load_img(img_path,
target_size=curr_size))
        img_arr = np.expand_dims(img_arr, axis=0) / 255.0

        preds = model.predict(img_arr)
        predicted_class_idx = np.argmax(preds[0])
        predicted_class_name = class_names[predicted_class_idx]
        print(f"Image: {img_path}")
        print(f"Predicted class: {predicted_class_name} (index:
{predicted_class_idx}), Probabilities: {preds[0]}")

        saliency = generate_saliency_map(model, img_path,
predicted_class_idx, image_size=curr_size)
        plot_saliency_map(img_path, saliency, predicted_class_name)
```

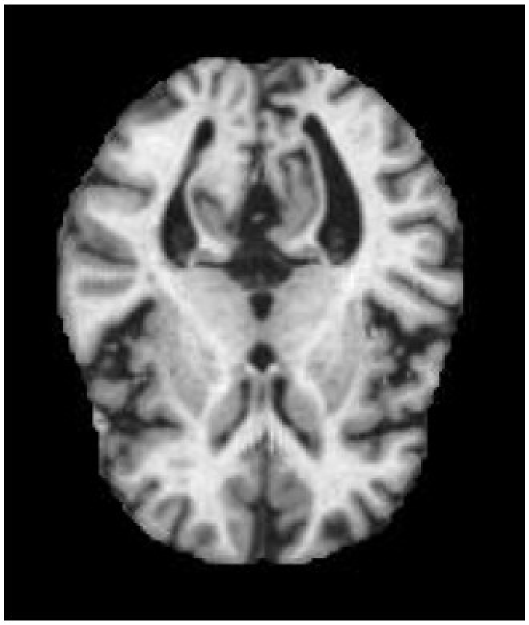
```
=== Model: InceptionV3 ===
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Image:
/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/AD/AD_10.png
Predicted class: CONTROL (index: 1), Probabilities: [0.29125798
0.5280658  0.18067624]
```



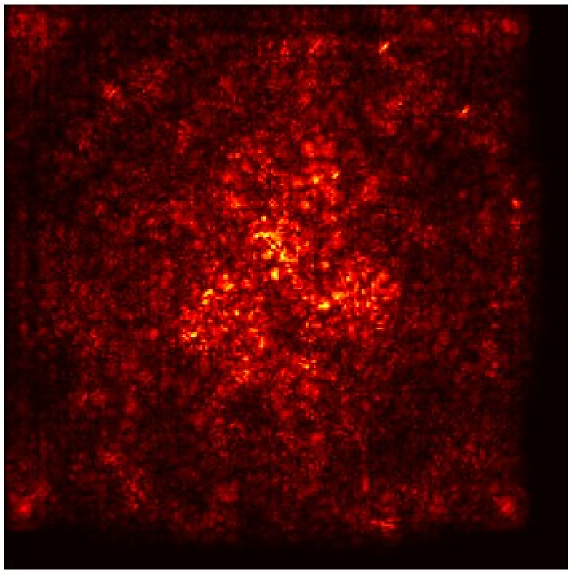


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Image:
/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/CONTROL/CONTROLAD
_10.png
Predicted class: CONTROL (index: 1), Probabilities: [0.18877828
0.55403847 0.25718325]
```

Original Image
Predicted: CONTROL

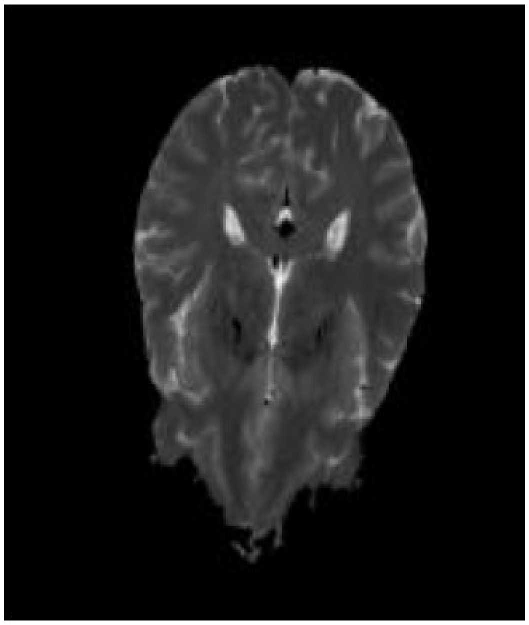


Saliency Map

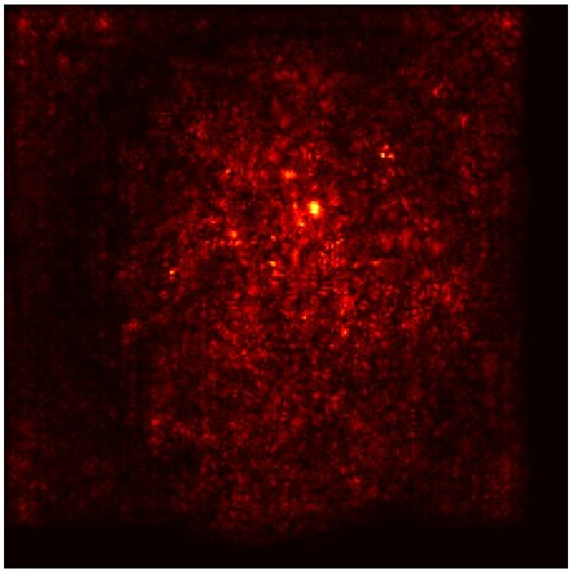


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Image:
/kaggle/input/classifying-ad-and-pd-v3/3_cls_2/train/PD/PD_10.png
Predicted class: CONTROL (index: 1), Probabilities: [0.1849218
0.46847308 0.34660518]
```

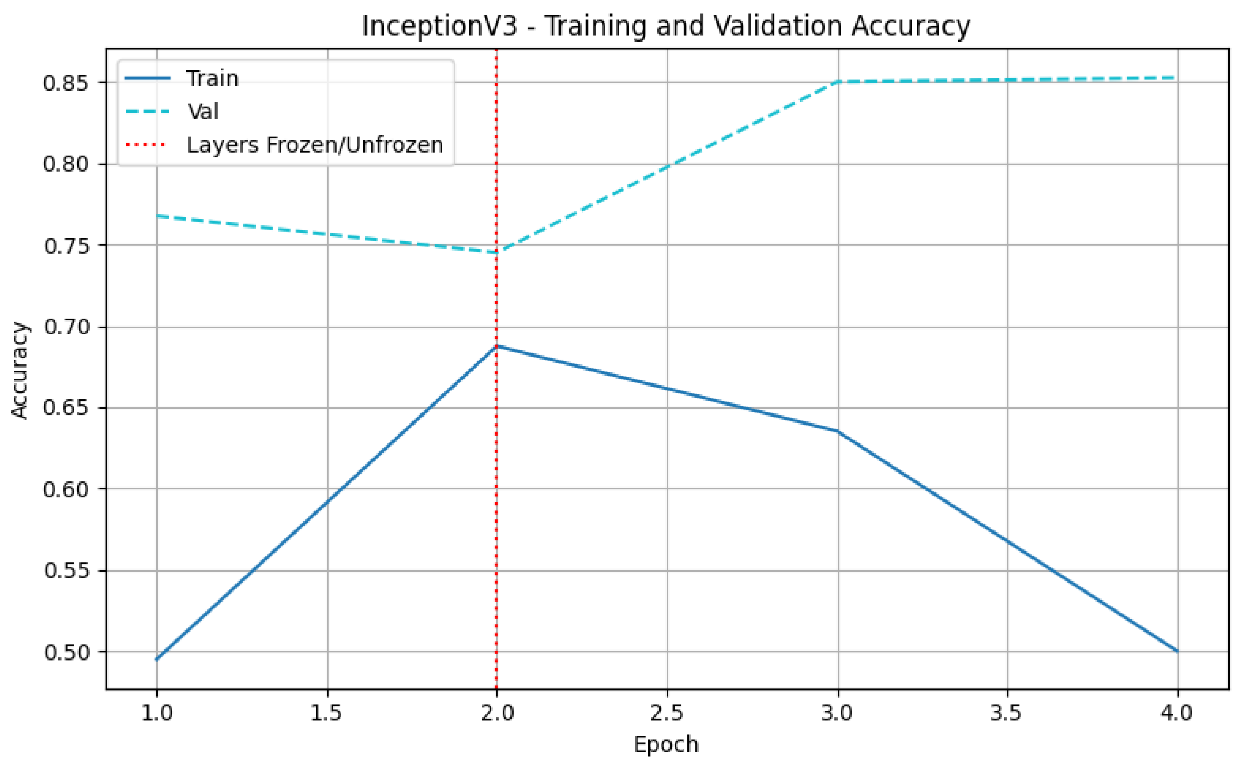
Original Image
Predicted: CONTROL



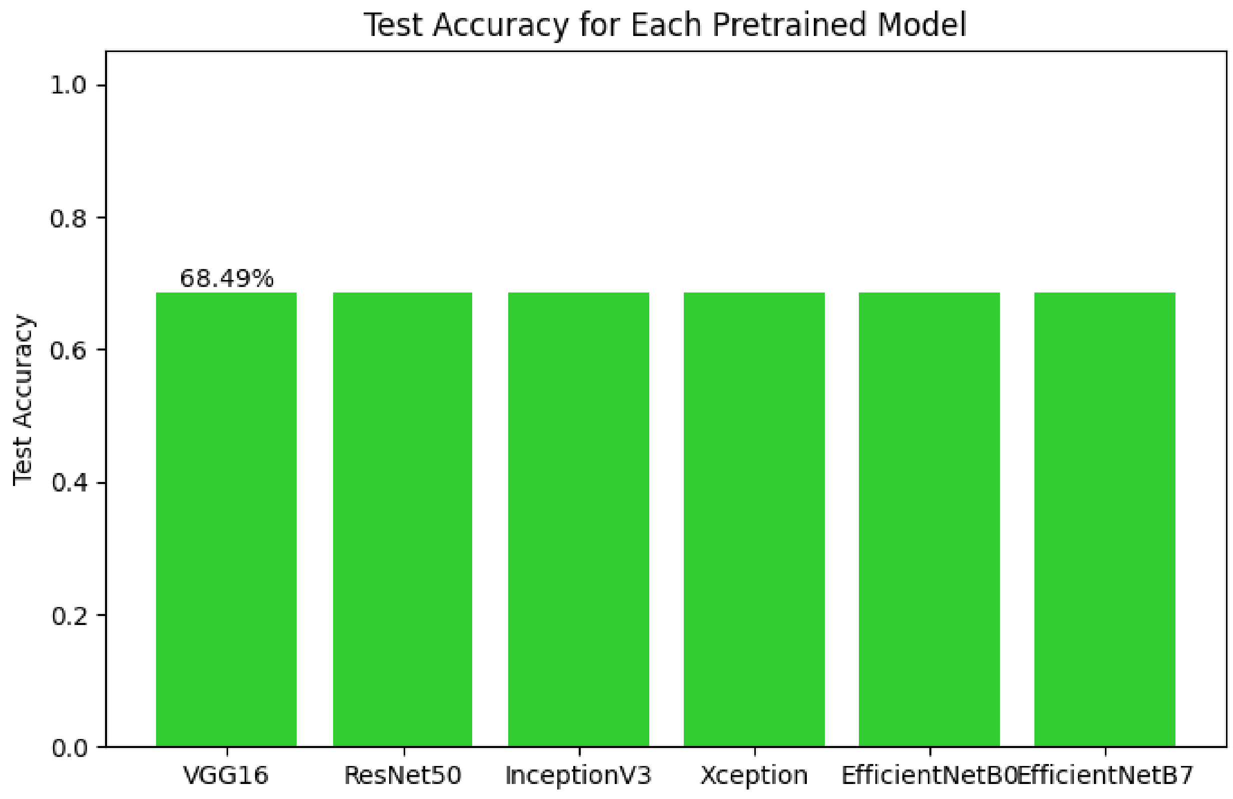
Saliency Map



```
import matplotlib.pyplot as plt

def plot_each_model_accuracy(histories, initial_epochs=None):
    """
    Plots training and validation accuracy for each model in separate
figures.

    Args:
        histories (dict): Keys are model names, values are Keras
History.history dicts.
        initial_epochs (int or None): If provided, draws a vertical
line at this epoch.
    """
    for model_name, history in histories.items():
        acc = history.get('accuracy') or history.get('acc')
        val_acc = history.get('val_accuracy') or
history.get('val_acc')
        epochs = range(1, len(acc) + 1)

        plt.figure(figsize=(8, 5))
        plt.plot(epochs, acc, label="Train", color='tab:blue',
linestyle='-')
        plt.plot(epochs, val_acc, label="Val", color='tab:cyan',
linestyle='--')
        if initial_epochs is not None:
            plt.axvline(x=initial_epochs, linestyle=':', color='red',
label='Layers Frozen/Unfrozen')
        plt.title(f"{model_name} - Training and Validation Accuracy")
        plt.xlabel("Epoch")
        plt.ylabel("Accuracy")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()

# Usage:
plot_each_model_accuracy(histories, initial_epochs=initial_epochs)
# If you don't have initial_epochs, just call:
plot_each_model_accuracy(histories)
```



```
import matplotlib.pyplot as plt

# Example model names in order
model_names = ['VGG16', 'ResNet50', 'InceptionV3', 'Xception',
'EfficientNetB0','EfficientNetB7']

# Your test_accuracies list
# test_accuracies = [0.81, 0.85, 0.83, 0.80, 0.88]  # Example

# Find the index (and name) of the model with the highest accuracy
best_index = max(range(len(test_accuracies)), key=lambda i:
test_accuracies[i])
best_accuracy = test_accuracies[best_index]
best_model = model_names[best_index]
print(f"Highest test accuracy: {best_accuracy:.4f} ({best_model})")

# Plot
plt.figure(figsize=(8, 5))
bars = plt.bar(model_names, test_accuracies, color='limegreen')
plt.ylabel("Test Accuracy")
plt.ylim(0, 1.05)
plt.title("Test Accuracy for Each Pretrained Model")
for bar, acc in zip(bars, test_accuracies):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(),
f"{acc:.2%}",
                ha='center', va='bottom', fontsize=10)
plt.show()

Highest test accuracy: 0.6849 (VGG16)
```


Test Accuracy for Each Pretrained Model
Test Accuracy
VGG16
ResNet50
InceptionV3
Xception
EfficientNetB0
EfficientNetB7