dmlmjo

April 27, 2025

0.1 CNN classifier

This project focuses on classifying brain tumours from MRI scans using deep learning models. A custom CNN and three fine-tuned pre-trained architectures (VGG16, ResNet50, and EfficientNetB0) were evaluated based on accuracy, precision, recall, F1-score, and confusion matrices. The study compares model performance, efficiency, and clinical applicability, highlighting the balance between accuracy and resource demands. Data augmentation was used to improve generalisation, and results aim to guide the selection of suitable models for real-world diagnostic use.

Dataset Source The dataset used in this project was obtained from Kaggle: Brain Tumor Dataset (Praneet Pawar, 2023). The dataset contains MRI images categorised into tumour and non-tumour classes and is publicly available for academic and research purposes. https://www.kaggle.com/datasets/praneet0327/brain-tumor-dataset

0.1.1 Import the necessary libraries.

[4]:
```
import os
import numpy as np
import cv2
import glob
import random
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten,
↪Dropout, BatchNormalization, GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16, ResNet50, EfficientNetB0
from tensorflow.keras.applications.vgg16 import preprocess_input as
↪vgg_preprocess
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
```

0.1.2 Preview the images in our dataset.

[6]:
```python
# Set the path to the images.

data_path = os.listdir('/Users/mj/Downloads/Brain_Tumor_Dataset')
```

[7]:
```python
# Identify the classes in the dataset.
# There are two classes, 'Positive' and 'Negative'
# Positive: Tumour is present in the image
# Negative: No tumour present

classes = os.listdir('/Users/mj/Downloads/Brain_Tumor_Dataset')
classes
```

[7]:
```
['Positive', 'Negative']
```

[108]:
```python
# Reanme the files in the negative directory to ensure consistency and␣
↪readability

folder = '/Users/mj/Downloads/Brain_Tumor_Dataset/Negative/'
count = 1

for filename in os.listdir(folder):
    source = folder + filename
    destination = folder + "N_" +str(count)+".jpg"
    os.rename(source, destination)
    count+=1
print("All files are in the Negative directory have been renamed.")
```

```
All files are in the Negative directory have been renamed.
```

[8]:
```python
# Rename the files in the 'Positive' directory to ensure consistency and␣
↪readability

folder = '/Users/mj/Downloads/Brain_Tumor_Dataset/Positive/'
count = 1

for filename in os.listdir(folder):
    source = folder + filename
    destination = folder + "Y_" +str(count)+".jpg"
    os.rename(source, destination)
    count+=1
print("All files are in the Positive directory have been renamed.")
```

```
All files are in the Positive directory have been renamed.
```

0.1.3 Data Preprocessing

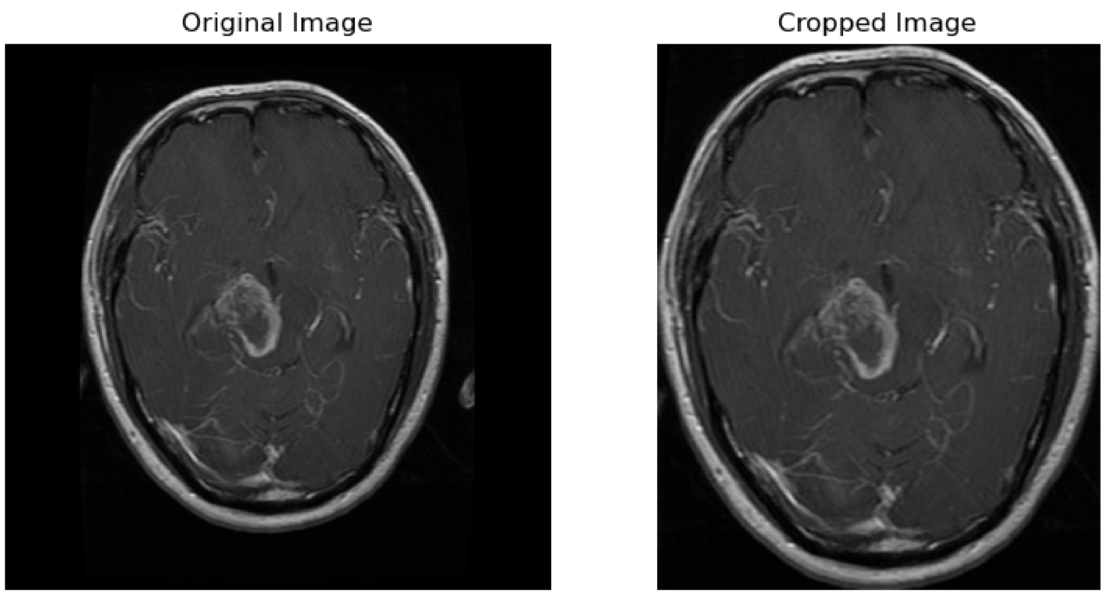
Before model training, the MRI images underwent several preprocessing steps to improve quality and ensure consistency. First, images were cropped to focus on the brain region and remove irrelevant background noise. Normalisation was then applied, scaling pixel values to the [0, 1] range to aid model convergence. Image enhancement techniques were used to improve contrast and highlight tumour regions more clearly. Finally, an ImageDataGenerator was employed to perform real-time data augmentation, introducing variations such as rotation, zoom, and flipping, which helped improve model generalisation and reduce overfitting.

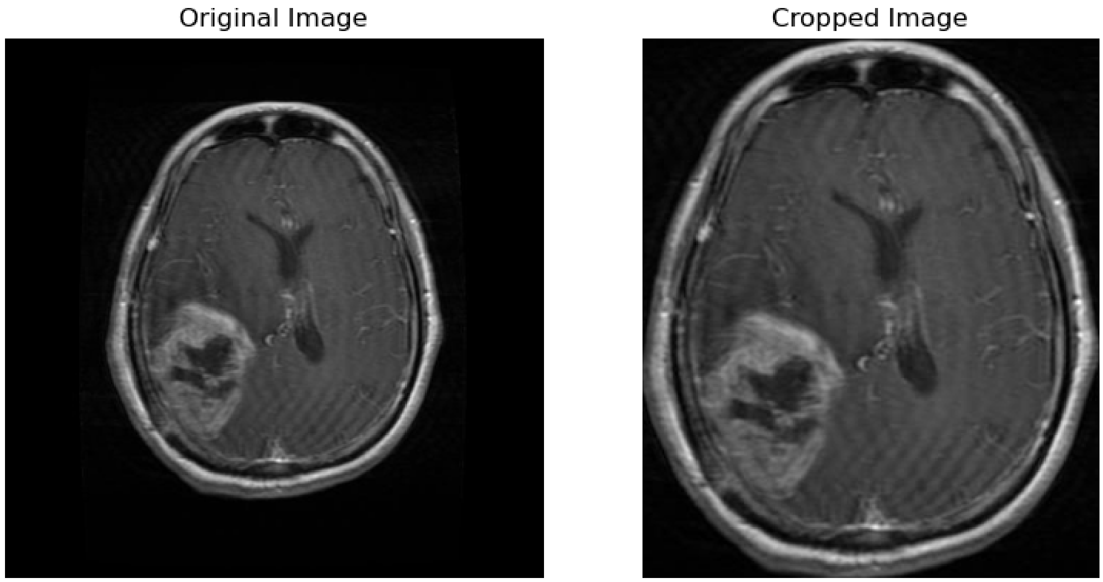
[12]:
```
def enhance_image(image):
    lab = cv2.cvtColor(image, cv2.COLOR_RGB2LAB)
    l, a, b = cv2.split(lab)
    clahe = cv2.createCLAHE(clipLimit=2.0)
    cl = clahe.apply(l)
    limg = cv2.merge((cl, a, b))
    enhanced = cv2.cvtColor(limg, cv2.COLOR_LAB2RGB)
    return enhanced
```

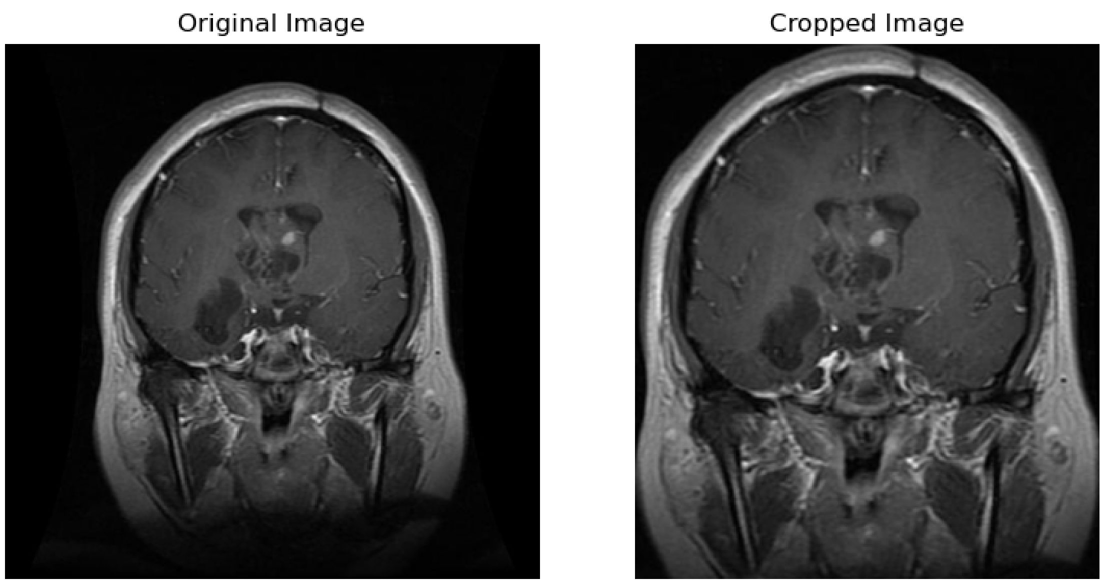
[14]:
```
# Cropping images helps the CNN model to focus on relevant regions of interest␣
↪and remove irrelevant areas.
# Cropping reduces noise, improves model training and increases accuracy.

def crop_brain_contour(image):
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    _, thresh = cv2.threshold(blurred, 45, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.
↪CHAIN_APPROX_SIMPLE)
    if contours:
        c = max(contours, key=cv2.contourArea)
        x, y, w, h = cv2.boundingRect(c)
        return image[y:y+h, x:x+w]
    return image
```

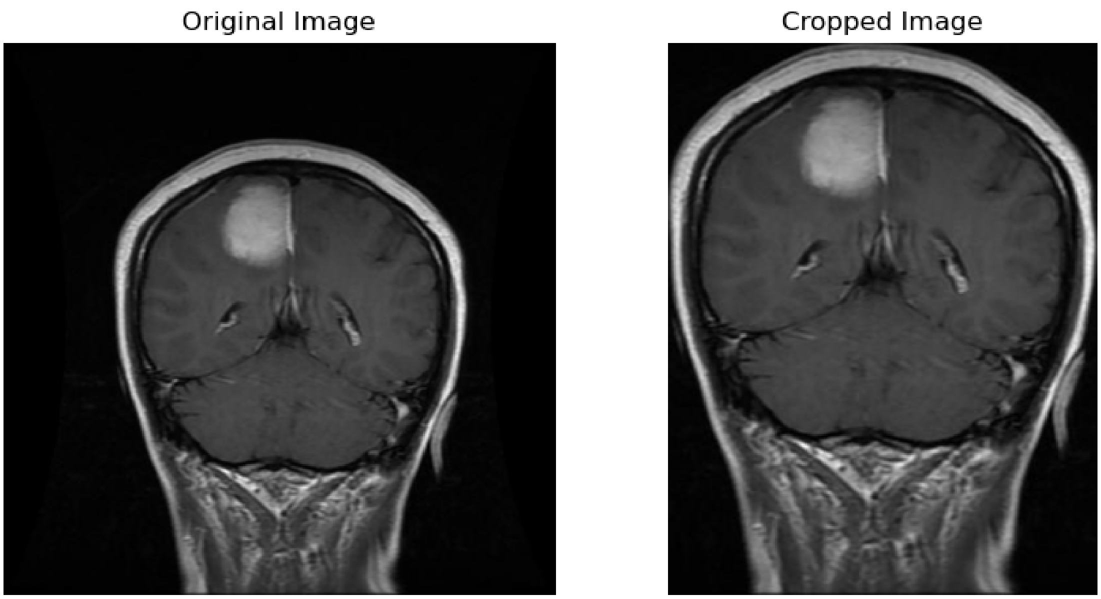
[15]:
```
# Visualise random images before and after cropping

def show_before_after_crop(data_dir, sample_size=5):
    all_images = []
    for root, dirs, files in os.walk(data_dir):
        for file in files:
            if file.lower().endswith(('.png', '.jpg', '.jpeg')):
                all_images.append(os.path.join(root, file))

    selected_images = random.sample(all_images, sample_size)

    for img_path in selected_images:
        image = cv2.imread(img_path)
```

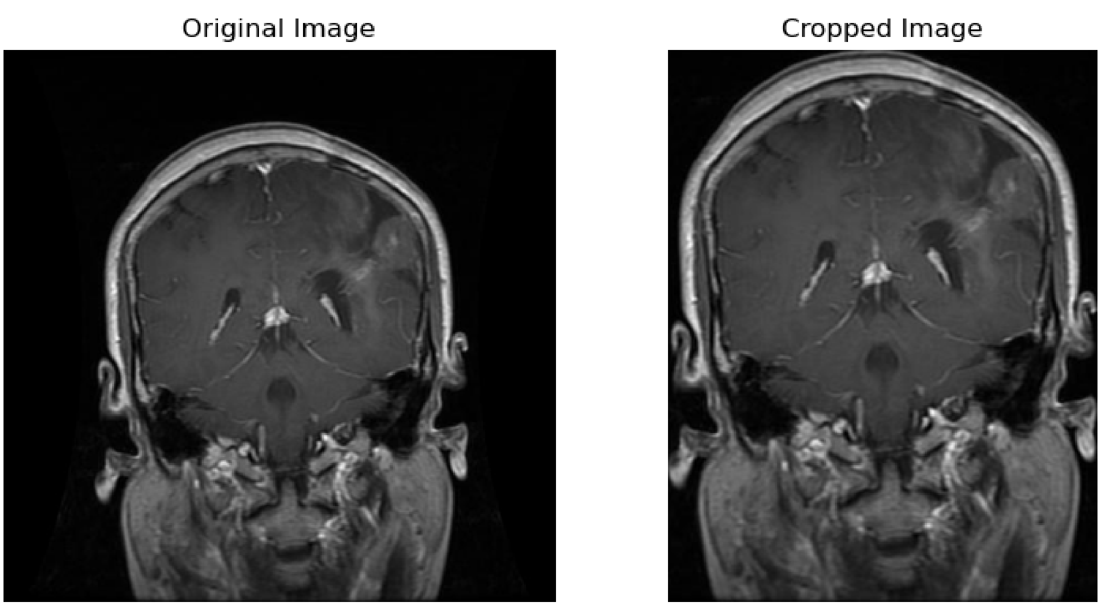
```
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        cropped = crop_brain_contour(image)


        plt.figure(figsize=(8, 4))
        plt.subplot(1, 2, 1)
        plt.imshow(image)
        plt.title('Original Image')
        plt.axis('off')

        plt.subplot(1, 2, 2)
        plt.imshow(cropped)
        plt.title('Cropped Image')
        plt.axis('off')

        plt.tight_layout()
        plt.show()
data_directory = "/Users/mj/Downloads/Brain_Tumor_Dataset"
show_before_after_crop(data_directory, sample_size=5)
```




Original Image
Cropped Image


Original Image
Cropped Image





[17]:
```
##### Iterates through folders labelled Positive and Negative to assign class labels.
##### Loads each image and resizes it to 224x224 pixels.
##### Converts the image from BGR (OpenCV default) to RGB format.
##### Enhances the image using CLAHE and crops it to isolate the brain region.
##### Ensures the final image is resized to a consistent shape after cropping.
##### Appends the processed image and its label to their respective lists.
```

```
##### The images and labels are returned as NumPy arrays, suitable for feeding␣
↪into machine learning models.



def load_images(data_dir, img_size=(224, 224)):
    images = []
    labels = []
    for label in ['Positive', 'Negative']:
        path = os.path.join(data_dir, label)
        for img in os.listdir(path):
            img_path = os.path.join(path, img)
            image = cv2.imread(img_path)
            if image is not None:
                image = cv2.resize(image, img_size)
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                image = enhance_image(image)
                image = crop_brain_contour(image)
                if image is not None:
                    image = cv2.resize(image, img_size)
                    images.append(image)
                    labels.append(1 if label == 'Positive' else 0)
    return np.array(images), np.array(labels)
```

The next part of the code Loads the dataset from the specified directory, normalises the image pixel values to a 0–1 range to improve neural network convergence and splits the dataset into training and testing sets using stratified sampling to preserve class balance.

[19]:
```
# The dataset was divided into training and testing sets using an 80:20 ratio.
# Stratified sampling was used to ensure that the class distribution remained␣
↪consistent across both sets.
# A random state of 42 was set to guarantee reproducibility of results.


data_path = "/Users/mj/Downloads/Brain_Tumor_Dataset"
X, y = load_images(data_path)
X = X / 255.0
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
↪test_size=0.2, random_state=42)
```

[21]:
```
# ImageDataGenerator is used to apply random transformations to the training␣
↪images
# Prevents overfitting

##### Small rotations (up to 15 degrees)
##### Random zooming (up to 10%)
```

```
##### Horizontal flipping
##### This technique helps create a more diverse training set without requiring␣
↪additional labelled data.



datagen = ImageDataGenerator(rotation_range=15, zoom_range=0.1,␣
↪horizontal_flip=True)
datagen.fit(X_train)
```

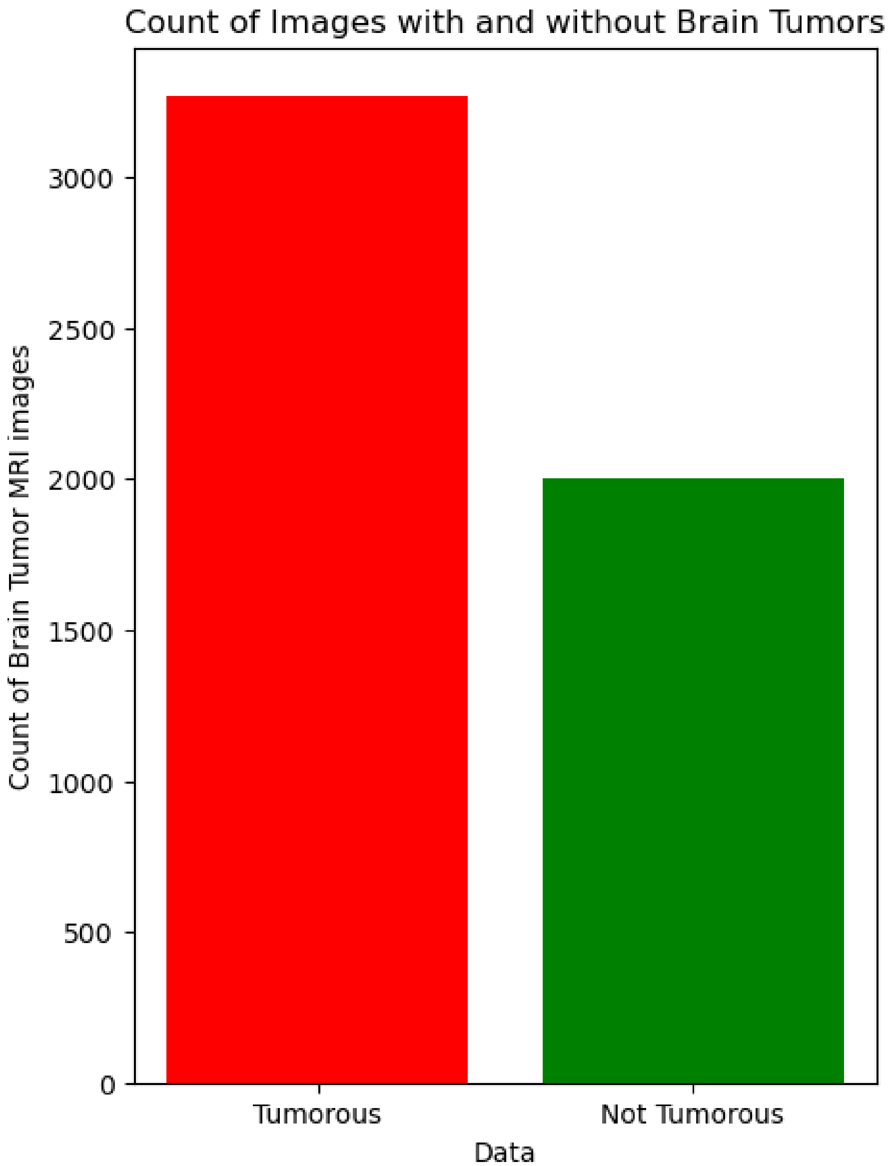
0.1.4 Exploratory Data Analysis

Identify and visualise trends in the dataset

[23]:
```
# The number of images with and without brain tumours

list_positive = os.listdir('/Users/mj/Downloads/Brain_Tumor_Dataset/Positive/')
tumorous_images = len(list_positive)
print(f'There are {tumorous_images} images with brain tumours')

list_negative = os.listdir('/Users/mj/Downloads/Brain_Tumor_Dataset/Negative/')
non_tumorous = len(list_negative)
print(f'There are {non_tumorous} images without brain tumours')
```

```
There are 3266 images with brain tumours
There are 2000 images without brain tumours
```

[24]:
```
data = {'Tumorous': tumorous_images, 'Not Tumorous': non_tumorous}

typex = list(data.keys())
values = list(data.values())
colours = ['red' if label == 'Tumorous' else 'green' for label in typex]

fig = plt.figure(figsize=(5, 7))
plt.bar(typex, values, color=colours)

plt.xlabel('Data')
plt.ylabel('Count of Brain Tumor MRI images')
plt.title('Count of Images with and without Brain Tumors')
plt.show()
```

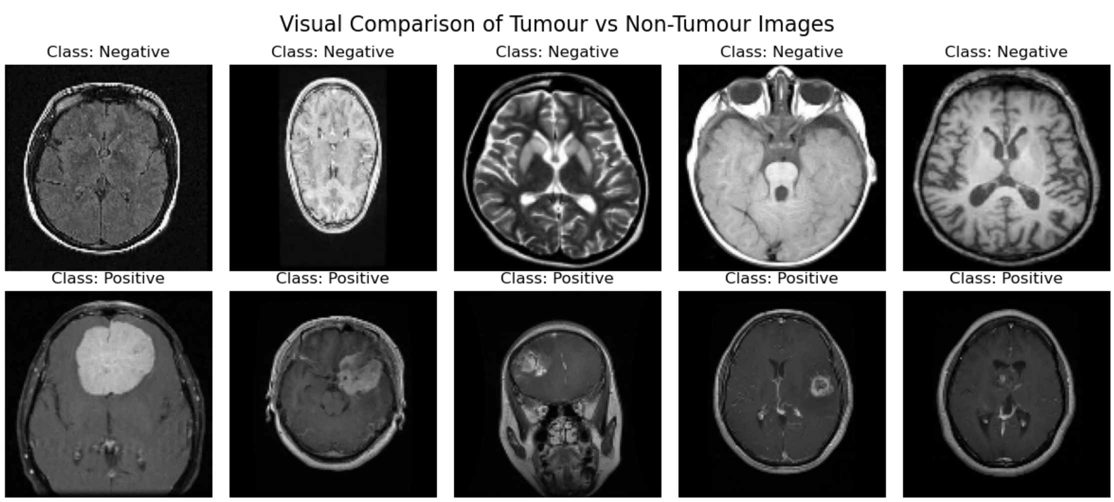


There is and imbalance where there are significantly more images in the 'Positive' (Tumorous) directory than the 'Negative' (Not Tumorous)ndirectory. This can affect how the model predits and it's accuracy. To manage this class weighting will be used.

Let us visualise some images in the dataset and their classes

[27]:
```
tumour = []
no_tumour = []

for image in glob.iglob('/Users/mj/Downloads/Brain_Tumor_Dataset/Positive/*.
↪jpg'):
```

```python
    img = cv2.imread(image)
    img = cv2.resize(img, (120, 120))
    tumour.append(img)
    if len(tumour) == 5:
        break

for image in glob.iglob('/Users/mj/Downloads/Brain_Tumor_Dataset/Negative/*.jpg'):
    img = cv2.imread(image)
    img = cv2.resize(img, (120, 120))
    no_tumour.append(img)
    if len(no_tumour) == 5:
        break

plt.figure(figsize=(12, 5))
for i in range(5):
    plt.subplot(2, 5, i + 1)
    plt.imshow(cv2.cvtColor(no_tumour[i], cv2.COLOR_BGR2RGB))
    plt.title("Class: Negative")
    plt.axis('off')

    plt.subplot(2, 5, i + 6)
    plt.imshow(cv2.cvtColor(tumour[i], cv2.COLOR_BGR2RGB))
    plt.title("Class: Positive")
    plt.axis('off')

plt.tight_layout()
plt.suptitle("Visual Comparison of Tumour vs Non-Tumour Images", fontsize=16, y=1.05)
plt.show()
```



0.1.5 CNN Model Building, Training, and Evaluation

This section covers the implementation of both custom and transfer learning-based CNN models for brain tumour classification, along with their training, fine-tuning, and evaluation. The create_custom_cnn function defines a custom Convolutional Neural Network (CNN) architecture from scratch using the Keras `Sequential` model.

- **Conv2D + MaxPooling2D**: These layers extract spatial features from the image. Filters of increasing size (32 → 64 → 128) allow the model to learn progressively more complex patterns.
- **BatchNormalization**: Normalises the outputs of convolution layers, stabilising learning and speeding up convergence.
- **Flatten + Dense**: Converts the 2D features into a 1D vector to pass into dense (fully connected) layers for classification.
- **Dropout**: Regularises the model by randomly disabling neurons, helping to reduce overfitting.
- **Sigmoid Output**: Used for binary classification (tumour vs no tumour).
- **Compilation**: The model is compiled with the Adam optimiser, binary cross-entropy loss (suitable for binary problems), and accuracy as the metric.

---

[30]:
```python
def create_custom_cnn(input_shape=(224, 224, 3)):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=input_shape),
        MaxPooling2D(pool_size=(2,2)),
        BatchNormalization(),

        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D(pool_size=(2,2)),
        BatchNormalization(),

        Conv2D(128, (3,3), activation='relu'),
        MaxPooling2D(pool_size=(2,2)),
        BatchNormalization(),

        Flatten(),
        Dense(256, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=Adam(1e-4), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

0.1.6 Transfer Model Building

The `build_transfer_model()` function wraps a **pre-trained model** with additional layers to adapt it for a **binary classification task**.

- **`base_model.trainable = False`**: **Freezes** the pre-trained layers, preventing them from being updated during training, allowing the model to retain the valuable features learned from the pre-trained data.
- **`GlobalAveragePooling2D`**: Reduces the output dimensions from the base model, summarising the spatial information while maintaining important features.
- **`Dense + Dropout + Output`**: Adds new **trainable layers** on top of the base model, specifically designed for the binary classification task. The dropout layer helps to reduce overfitting.
- **Compilation**: The model is compiled using the same **loss function**, **optimizer**, and **metrics** as the custom model, ensuring consistency in the training process.

Transfer learning is particularly useful when **training data is limited**, as it leverages features learned from large datasets (such as ImageNet), improving the model's ability to generalise with fewer data points.

[32]:
```
def build_transfer_model(base_model):
    base_model.trainable = False
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.5)(x)
    output = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=base_model.input, outputs=output)
    model.compile(optimizer=Adam(1e-4), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

0.1.7 Model Training Overview

The model is trained using **data augmentation** and **early stopping** to improve generalisation and prevent overfitting.

- **`datagen.flow(...)`**: Applies **real-time data augmentation** to increase dataset variability and expose the model to diverse examples.
- **`EarlyStopping`**: Stops training if the **validation loss** does not improve for 5 epochs, ensuring that the best-performing model is retained and preventing overfitting.
- **Validation Data**: Used to monitor **model generalisation** during training by evaluating the model on unseen data after each epoch.

These strategies help to **prevent overfitting** and ensure that the model performs well on new, unseen data.

0.1.8 Fine-Tuning Process

The `fine_tune_model()` function unfreezes the top layers of the pre-trained base model, allowing it to adapt to the new dataset while retaining general features learned from previous training.

- **`base_model.trainable = True`**: Unlocks the pre-trained base model for further training, enabling weight updates.

- **Selective Layer Freezing**: Freezes most layers except for the top `unfreeze_layers` to avoid **catastrophic forgetting** of previously learned features.
- **Lower Learning Rate**: Fine-tuning uses a smaller learning rate (1e-5) to make **subtle updates** to the pre-trained weights without disturbing important low-level features.
- `EarlyStopping`: Still used to monitor **validation performance** and stop training early if there's no improvement.

Fine-tuning allows the model to **adjust high-level features** specific to your task, without losing the generalised knowledge from the pre-trained model.

[34]:
```
def train_model(model, X_train, y_train, X_test, y_test, name="model"):
    early_stop = EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)
    return model.fit(datagen.flow(X_train, y_train, batch_size=32),
                     validation_data=(X_test, y_test),
                     epochs=25,
                     callbacks=[early_stop],
                     verbose=1)

def fine_tune_model(model, base_model, X_train, y_train, X_test, y_test, name,␣
 ↪unfreeze_layers=20):
    base_model.trainable = True
    for layer in base_model.layers[:-unfreeze_layers]:
        layer.trainable = False

    model.compile(optimizer=Adam(1e-5), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    early_stop = EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)
    return model.fit(datagen.flow(X_train, y_train, batch_size=32),
                     validation_data=(X_test, y_test),
                     epochs=15,
                     callbacks=[early_stop],
                     verbose=1)
```

[35]:
```
def evaluate_model(model, X_test, y_test, name):
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    print(f"\n  Evaluation for {name}")
    print(classification_report(y_test, y_pred))
    print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

def plot_training(history, title):
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs_range = range(len(acc))
```

```python
    plt.figure(figsize=(14, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, acc, label='Train Accuracy')
    plt.plot(epochs_range, val_acc, label='Val Accuracy')
    plt.title(f'{title} Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, loss, label='Train Loss')
    plt.plot(epochs_range, val_loss, label='Val Loss')
    plt.title(f'{title} Loss')
    plt.legend()
    plt.show()
```

[36]:

```python
# Custom model

custom_model = create_custom_cnn()
custom_history = train_model(custom_model, X_train, y_train, X_test, y_test,␣
 ↪"Custom CNN")
evaluate_model(custom_model, X_test, y_test, "Custom CNN")
```

```
/opt/anaconda3/lib/python3.12/site-
packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/25

/opt/anaconda3/lib/python3.12/site-
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()

132/132              60s 451ms/step -
accuracy: 0.8729 - loss: 0.5429 - val_accuracy: 0.6205 - val_loss: 3.4834
Epoch 2/25
132/132              57s 428ms/step -
accuracy: 0.9391 - loss: 0.1851 - val_accuracy: 0.6205 - val_loss: 4.0163
Epoch 3/25
132/132              58s 438ms/step -
accuracy: 0.9587 - loss: 0.1193 - val_accuracy: 0.7799 - val_loss: 0.9282
Epoch 4/25
132/132              58s 439ms/step -
accuracy: 0.9617 - loss: 0.1125 - val_accuracy: 0.9231 - val_loss: 0.2842
```

```
Epoch 5/25
132/132             1553s 12s/step -
accuracy: 0.9681 - loss: 0.0913 - val_accuracy: 0.9848 - val_loss: 0.0715
Epoch 6/25
132/132             55s 415ms/step -
accuracy: 0.9694 - loss: 0.0918 - val_accuracy: 0.9877 - val_loss: 0.0635
Epoch 7/25
132/132             57s 430ms/step -
accuracy: 0.9777 - loss: 0.0734 - val_accuracy: 0.9896 - val_loss: 0.0582
Epoch 8/25
132/132             57s 434ms/step -
accuracy: 0.9753 - loss: 0.0637 - val_accuracy: 0.9791 - val_loss: 0.0948
Epoch 9/25
132/132             58s 440ms/step -
accuracy: 0.9762 - loss: 0.0633 - val_accuracy: 0.9194 - val_loss: 0.3169
Epoch 10/25
132/132             59s 445ms/step -
accuracy: 0.9786 - loss: 0.0736 - val_accuracy: 0.9905 - val_loss: 0.0664
Epoch 11/25
132/132             1116s 9s/step -
accuracy: 0.9765 - loss: 0.0711 - val_accuracy: 0.9867 - val_loss: 0.0648
Epoch 12/25
132/132             3749s 29s/step -
accuracy: 0.9781 - loss: 0.0659 - val_accuracy: 0.9915 - val_loss: 0.0577
Epoch 13/25
132/132             3715s 28s/step -
accuracy: 0.9880 - loss: 0.0465 - val_accuracy: 0.9924 - val_loss: 0.0547
Epoch 14/25
132/132             56s 421ms/step -
accuracy: 0.9819 - loss: 0.0490 - val_accuracy: 0.9905 - val_loss: 0.0622
Epoch 15/25
132/132             57s 430ms/step -
accuracy: 0.9858 - loss: 0.0446 - val_accuracy: 0.9905 - val_loss: 0.0764
Epoch 16/25
132/132             58s 439ms/step -
accuracy: 0.9900 - loss: 0.0318 - val_accuracy: 0.9905 - val_loss: 0.1000
Epoch 17/25
132/132             59s 447ms/step -
accuracy: 0.9876 - loss: 0.0401 - val_accuracy: 0.9924 - val_loss: 0.0653
Epoch 18/25
132/132             59s 450ms/step -
accuracy: 0.9921 - loss: 0.0224 - val_accuracy: 0.9896 - val_loss: 0.0560
33/33             3s 100ms/step

 Evaluation for Custom CNN
             precision    recall  f1-score   support

           0       0.99      0.99      0.99       400
```

```
           1       0.99      1.00      0.99       654

    accuracy                           0.99      1054
   macro avg       0.99      0.99      0.99      1054
weighted avg       0.99      0.99      0.99      1054

Confusion Matrix:
 [[395   5]
 [  3 651]]
```

[50]:

```
# VGG16 Transfer Learning

vgg_base = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224,␣
 ↪3))
vgg_model = build_transfer_model(vgg_base)
vgg_history = train_model(vgg_model, X_train, y_train, X_test, y_test, "VGG16")
vgg_fine = fine_tune_model(vgg_model, vgg_base, X_train, y_train, X_test,␣
 ↪y_test, "VGG16", 8)
evaluate_model(vgg_model, X_test, y_test, "VGG16 (Fine-Tuned)")
```

```
Epoch 1/25
132/132              358s 3s/step -
accuracy: 0.5721 - loss: 0.7012 - val_accuracy: 0.7865 - val_loss: 0.5300
Epoch 2/25
132/132              398s 3s/step -
accuracy: 0.7137 - loss: 0.5564 - val_accuracy: 0.8795 - val_loss: 0.4304
Epoch 3/25
132/132              434s 3s/step -
accuracy: 0.8287 - loss: 0.4518 - val_accuracy: 0.9042 - val_loss: 0.3617
Epoch 4/25
132/132              431s 3s/step -
accuracy: 0.8629 - loss: 0.3849 - val_accuracy: 0.9146 - val_loss: 0.3157
Epoch 5/25
132/132              401s 3s/step -
accuracy: 0.8940 - loss: 0.3303 - val_accuracy: 0.9127 - val_loss: 0.2730
Epoch 6/25
132/132              404s 3s/step -
accuracy: 0.9100 - loss: 0.2954 - val_accuracy: 0.9250 - val_loss: 0.2478
Epoch 7/25
132/132              415s 3s/step -
accuracy: 0.9141 - loss: 0.2683 - val_accuracy: 0.9345 - val_loss: 0.2301
Epoch 8/25
132/132              418s 3s/step -
accuracy: 0.9189 - loss: 0.2372 - val_accuracy: 0.9383 - val_loss: 0.2207
Epoch 9/25
132/132              416s 3s/step -
accuracy: 0.9225 - loss: 0.2348 - val_accuracy: 0.9421 - val_loss: 0.2027
Epoch 10/25
```

```
132/132              413s 3s/step -
accuracy: 0.9174 - loss: 0.2236 - val_accuracy: 0.9459 - val_loss: 0.1937
Epoch 11/25
132/132              431s 3s/step -
accuracy: 0.9247 - loss: 0.2125 - val_accuracy: 0.9488 - val_loss: 0.1807
Epoch 12/25
132/132              435s 3s/step -
accuracy: 0.9368 - loss: 0.1928 - val_accuracy: 0.9497 - val_loss: 0.1787
Epoch 13/25
132/132              427s 3s/step -
accuracy: 0.9397 - loss: 0.1865 - val_accuracy: 0.9497 - val_loss: 0.1749
Epoch 14/25
132/132              419s 3s/step -
accuracy: 0.9459 - loss: 0.1757 - val_accuracy: 0.9516 - val_loss: 0.1680
Epoch 15/25
132/132              435s 3s/step -
accuracy: 0.9425 - loss: 0.1829 - val_accuracy: 0.9516 - val_loss: 0.1631
Epoch 16/25
132/132              423s 3s/step -
accuracy: 0.9406 - loss: 0.1717 - val_accuracy: 0.9535 - val_loss: 0.1557
Epoch 17/25
132/132              411s 3s/step -
accuracy: 0.9496 - loss: 0.1661 - val_accuracy: 0.9526 - val_loss: 0.1627
Epoch 18/25
132/132              432s 3s/step -
accuracy: 0.9471 - loss: 0.1646 - val_accuracy: 0.9554 - val_loss: 0.1494
Epoch 19/25
132/132              435s 3s/step -
accuracy: 0.9445 - loss: 0.1707 - val_accuracy: 0.9545 - val_loss: 0.1482
Epoch 20/25
132/132              421s 3s/step -
accuracy: 0.9395 - loss: 0.1669 - val_accuracy: 0.9554 - val_loss: 0.1466
Epoch 21/25
132/132              422s 3s/step -
accuracy: 0.9435 - loss: 0.1550 - val_accuracy: 0.9564 - val_loss: 0.1429
Epoch 22/25
132/132              404s 3s/step -
accuracy: 0.9496 - loss: 0.1500 - val_accuracy: 0.9564 - val_loss: 0.1438
Epoch 23/25
132/132              405s 3s/step -
accuracy: 0.9390 - loss: 0.1610 - val_accuracy: 0.9554 - val_loss: 0.1455
Epoch 24/25
132/132              407s 3s/step -
accuracy: 0.9516 - loss: 0.1458 - val_accuracy: 0.9497 - val_loss: 0.1496
Epoch 25/25
132/132              405s 3s/step -
accuracy: 0.9472 - loss: 0.1461 - val_accuracy: 0.9564 - val_loss: 0.1383
Epoch 1/15
```

```
132/132              663s 5s/step -
accuracy: 0.9346 - loss: 0.1640 - val_accuracy: 0.9782 - val_loss: 0.0703
Epoch 2/15
132/132              657s 5s/step -
accuracy: 0.9707 - loss: 0.0867 - val_accuracy: 0.9810 - val_loss: 0.0514
Epoch 3/15
132/132              656s 5s/step -
accuracy: 0.9824 - loss: 0.0489 - val_accuracy: 0.9867 - val_loss: 0.0459
Epoch 4/15
132/132              649s 5s/step -
accuracy: 0.9897 - loss: 0.0364 - val_accuracy: 0.9905 - val_loss: 0.0347
Epoch 5/15
132/132              646s 5s/step -
accuracy: 0.9955 - loss: 0.0190 - val_accuracy: 0.9943 - val_loss: 0.0218
Epoch 6/15
132/132              649s 5s/step -
accuracy: 0.9955 - loss: 0.0140 - val_accuracy: 0.9953 - val_loss: 0.0196
Epoch 7/15
132/132              647s 5s/step -
accuracy: 0.9980 - loss: 0.0101 - val_accuracy: 0.9934 - val_loss: 0.0292
Epoch 8/15
132/132              646s 5s/step -
accuracy: 0.9987 - loss: 0.0072 - val_accuracy: 0.9953 - val_loss: 0.0182
Epoch 9/15
132/132              651s 5s/step -
accuracy: 0.9954 - loss: 0.0143 - val_accuracy: 0.9953 - val_loss: 0.0237
Epoch 10/15
132/132              682s 5s/step -
accuracy: 0.9973 - loss: 0.0086 - val_accuracy: 0.9953 - val_loss: 0.0195
Epoch 11/15
132/132              646s 5s/step -
accuracy: 0.9989 - loss: 0.0025 - val_accuracy: 0.9953 - val_loss: 0.0201
Epoch 12/15
132/132              648s 5s/step -
accuracy: 0.9942 - loss: 0.0163 - val_accuracy: 0.9924 - val_loss: 0.0270
Epoch 13/15
132/132              650s 5s/step -
accuracy: 0.9985 - loss: 0.0085 - val_accuracy: 0.9953 - val_loss: 0.0282
33/33              81s 2s/step

 Evaluation for VGG16 (Fine-Tuned)
              precision    recall  f1-score   support

           0       1.00      0.99      0.99       400
           1       0.99      1.00      1.00       654

    accuracy                           1.00      1054
   macro avg       1.00      0.99      0.99      1054
```

```
weighted avg       1.00      1.00      1.00      1054

Confusion Matrix:
 [[395   5]
 [  0 654]]
```

[52]:

```python
# ResNet50 Transfer Learning

resnet_base = ResNet50(weights='imagenet', include_top=False, input_shape=(224,
    224, 3))
resnet_model = build_transfer_model(resnet_base)
resnet_history = train_model(resnet_model, X_train, y_train, X_test, y_test,
    "ResNet50")
resnet_fine = fine_tune_model(resnet_model, resnet_base, X_train, y_train,
    X_test, y_test, "ResNet50", 10)
evaluate_model(resnet_model, X_test, y_test, "ResNet50 (Fine-Tuned)")
```

```
Epoch 1/25

/opt/anaconda3/lib/python3.12/site-
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()

132/132             106s 791ms/step -
accuracy: 0.5977 - loss: 0.7056 - val_accuracy: 0.7761 - val_loss: 0.5162
Epoch 2/25
132/132             102s 772ms/step -
accuracy: 0.7727 - loss: 0.5210 - val_accuracy: 0.8406 - val_loss: 0.4499
Epoch 3/25
132/132             104s 790ms/step -
accuracy: 0.8030 - loss: 0.4703 - val_accuracy: 0.8681 - val_loss: 0.4269
Epoch 4/25
132/132             107s 812ms/step -
accuracy: 0.8323 - loss: 0.4353 - val_accuracy: 0.8719 - val_loss: 0.4004
Epoch 5/25
132/132             108s 815ms/step -
accuracy: 0.8393 - loss: 0.4106 - val_accuracy: 0.8748 - val_loss: 0.3874
Epoch 6/25
132/132             109s 823ms/step -
accuracy: 0.8557 - loss: 0.3895 - val_accuracy: 0.8786 - val_loss: 0.3679
Epoch 7/25
132/132             109s 827ms/step -
accuracy: 0.8421 - loss: 0.3928 - val_accuracy: 0.8786 - val_loss: 0.3639
Epoch 8/25
```

```
132/132              114s 861ms/step -
accuracy: 0.8630 - loss: 0.3665 - val_accuracy: 0.8795 - val_loss: 0.3573
Epoch 9/25
132/132              170s 1s/step -
accuracy: 0.8604 - loss: 0.3469 - val_accuracy: 0.8767 - val_loss: 0.3624
Epoch 10/25
132/132              106s 801ms/step -
accuracy: 0.8535 - loss: 0.3578 - val_accuracy: 0.8833 - val_loss: 0.3376
Epoch 11/25
132/132              108s 816ms/step -
accuracy: 0.8632 - loss: 0.3450 - val_accuracy: 0.8795 - val_loss: 0.3259
Epoch 12/25
132/132              307s 2s/step -
accuracy: 0.8652 - loss: 0.3404 - val_accuracy: 0.8767 - val_loss: 0.3579
Epoch 13/25
132/132              102s 775ms/step -
accuracy: 0.8662 - loss: 0.3347 - val_accuracy: 0.8852 - val_loss: 0.3261
Epoch 14/25
132/132              106s 803ms/step -
accuracy: 0.8717 - loss: 0.3311 - val_accuracy: 0.8852 - val_loss: 0.3281
Epoch 15/25
132/132              108s 820ms/step -
accuracy: 0.8691 - loss: 0.3276 - val_accuracy: 0.8871 - val_loss: 0.3115
Epoch 16/25
132/132              908s 7s/step -
accuracy: 0.8767 - loss: 0.3162 - val_accuracy: 0.8861 - val_loss: 0.3031
Epoch 17/25
132/132              102s 777ms/step -
accuracy: 0.8674 - loss: 0.3303 - val_accuracy: 0.8880 - val_loss: 0.3048
Epoch 18/25
132/132              105s 798ms/step -
accuracy: 0.8826 - loss: 0.3039 - val_accuracy: 0.8909 - val_loss: 0.3060
Epoch 19/25
132/132              1473s 11s/step -
accuracy: 0.8762 - loss: 0.3097 - val_accuracy: 0.8928 - val_loss: 0.3050
Epoch 20/25
132/132              101s 762ms/step -
accuracy: 0.8732 - loss: 0.3112 - val_accuracy: 0.8947 - val_loss: 0.2998
Epoch 21/25
132/132              104s 791ms/step -
accuracy: 0.8744 - loss: 0.2985 - val_accuracy: 0.8937 - val_loss: 0.2991
Epoch 22/25
132/132              107s 809ms/step -
accuracy: 0.8856 - loss: 0.2930 - val_accuracy: 0.8956 - val_loss: 0.2868
Epoch 23/25
132/132              108s 820ms/step -
accuracy: 0.8923 - loss: 0.2875 - val_accuracy: 0.8956 - val_loss: 0.2847
Epoch 24/25
```

```
132/132              108s 820ms/step -
accuracy: 0.8825 - loss: 0.2850 - val_accuracy: 0.8947 - val_loss: 0.2893
Epoch 25/25
132/132              114s 866ms/step -
accuracy: 0.8859 - loss: 0.2840 - val_accuracy: 0.8947 - val_loss: 0.2857
Epoch 1/15
132/132              131s 974ms/step -
accuracy: 0.8218 - loss: 0.5007 - val_accuracy: 0.5275 - val_loss: 0.9426
Epoch 2/15
132/132              130s 985ms/step -
accuracy: 0.9282 - loss: 0.1933 - val_accuracy: 0.7021 - val_loss: 0.5829
Epoch 3/15
132/132              131s 991ms/step -
accuracy: 0.9460 - loss: 0.1552 - val_accuracy: 0.9032 - val_loss: 0.2524
Epoch 4/15
132/132              132s 999ms/step -
accuracy: 0.9428 - loss: 0.1386 - val_accuracy: 0.9080 - val_loss: 0.2334
Epoch 5/15
132/132              1647s 13s/step -
accuracy: 0.9480 - loss: 0.1343 - val_accuracy: 0.9564 - val_loss: 0.1239
Epoch 6/15
132/132              113s 854ms/step -
accuracy: 0.9596 - loss: 0.1209 - val_accuracy: 0.9526 - val_loss: 0.1280
Epoch 7/15
132/132              117s 890ms/step -
accuracy: 0.9504 - loss: 0.1300 - val_accuracy: 0.9620 - val_loss: 0.1015
Epoch 8/15
132/132              1100s 8s/step -
accuracy: 0.9548 - loss: 0.1203 - val_accuracy: 0.9658 - val_loss: 0.0970
Epoch 9/15
132/132              114s 862ms/step -
accuracy: 0.9594 - loss: 0.0972 - val_accuracy: 0.9564 - val_loss: 0.1191
Epoch 10/15
132/132              119s 900ms/step -
accuracy: 0.9645 - loss: 0.0959 - val_accuracy: 0.9725 - val_loss: 0.0972
Epoch 11/15
132/132              251s 2s/step -
accuracy: 0.9673 - loss: 0.1026 - val_accuracy: 0.9658 - val_loss: 0.0928
Epoch 12/15
132/132              118s 897ms/step -
accuracy: 0.9677 - loss: 0.0870 - val_accuracy: 0.9573 - val_loss: 0.0999
Epoch 13/15
132/132              120s 913ms/step -
accuracy: 0.9616 - loss: 0.1046 - val_accuracy: 0.9583 - val_loss: 0.1045
Epoch 14/15
132/132              3120s 24s/step -
accuracy: 0.9623 - loss: 0.1023 - val_accuracy: 0.9753 - val_loss: 0.0783
Epoch 15/15
```

```
132/132              113s 855ms/step -
accuracy: 0.9686 - loss: 0.0886 - val_accuracy: 0.9734 - val_loss: 0.0789
33/33              21s 618ms/step

 Evaluation for ResNet50 (Fine-Tuned)
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       400
           1       0.99      0.97      0.98       654

    accuracy                           0.98      1054
   macro avg       0.97      0.98      0.97      1054
weighted avg       0.98      0.98      0.98      1054

Confusion Matrix:
 [[392   8]
 [ 18 636]]
```

[53]:

```
# EfficientNetB0 Transfer Learning

efficient_base = EfficientNetB0(weights='imagenet', include_top=False,␣
↪input_shape=(224, 224, 3))
efficient_model = build_transfer_model(efficient_base)
efficient_history = train_model(efficient_model, X_train, y_train, X_test,␣
↪y_test, "EfficientNetB0")
efficient_fine = fine_tune_model(efficient_model, efficient_base, X_train,␣
↪y_train, X_test, y_test, "EfficientNetB0", 12)
evaluate_model(efficient_model, X_test, y_test, "EfficientNetB0 (Fine-Tuned)")
```

```
/opt/anaconda3/lib/python3.12/site-
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 1/25
132/132              47s 335ms/step -
accuracy: 0.5855 - loss: 0.6794 - val_accuracy: 0.6205 - val_loss: 0.6761
Epoch 2/25
132/132              43s 326ms/step -
accuracy: 0.6187 - loss: 0.6663 - val_accuracy: 0.6205 - val_loss: 0.6636
Epoch 3/25
132/132              44s 336ms/step -
accuracy: 0.6241 - loss: 0.6619 - val_accuracy: 0.6205 - val_loss: 0.6617
Epoch 4/25
132/132              1111s 8s/step -
```

```
accuracy: 0.6220 - loss: 0.6631 - val_accuracy: 0.6205 - val_loss: 0.6597
Epoch 5/25
132/132            42s 315ms/step -
accuracy: 0.6185 - loss: 0.6640 - val_accuracy: 0.6205 - val_loss: 0.6630
Epoch 6/25
132/132            43s 324ms/step -
accuracy: 0.6235 - loss: 0.6623 - val_accuracy: 0.6205 - val_loss: 0.6578
Epoch 7/25
132/132            43s 325ms/step -
accuracy: 0.6169 - loss: 0.6599 - val_accuracy: 0.6205 - val_loss: 0.6578
Epoch 8/25
132/132            44s 329ms/step -
accuracy: 0.6054 - loss: 0.6667 - val_accuracy: 0.6205 - val_loss: 0.6569
Epoch 9/25
132/132            44s 335ms/step -
accuracy: 0.6221 - loss: 0.6603 - val_accuracy: 0.6205 - val_loss: 0.6559
Epoch 10/25
132/132            44s 335ms/step -
accuracy: 0.6263 - loss: 0.6549 - val_accuracy: 0.6205 - val_loss: 0.6559
Epoch 11/25
132/132            44s 334ms/step -
accuracy: 0.6166 - loss: 0.6630 - val_accuracy: 0.6205 - val_loss: 0.6557
Epoch 12/25
132/132            1868s 14s/step -
accuracy: 0.6170 - loss: 0.6612 - val_accuracy: 0.6205 - val_loss: 0.6544
Epoch 13/25
132/132            453s 3s/step -
accuracy: 0.6035 - loss: 0.6683 - val_accuracy: 0.6205 - val_loss: 0.6536
Epoch 14/25
132/132            42s 316ms/step -
accuracy: 0.6197 - loss: 0.6574 - val_accuracy: 0.6205 - val_loss: 0.6530
Epoch 15/25
132/132            43s 324ms/step -
accuracy: 0.6189 - loss: 0.6586 - val_accuracy: 0.6205 - val_loss: 0.6519
Epoch 16/25
132/132            43s 327ms/step -
accuracy: 0.6103 - loss: 0.6581 - val_accuracy: 0.6205 - val_loss: 0.6541
Epoch 17/25
132/132            43s 328ms/step -
accuracy: 0.6169 - loss: 0.6561 - val_accuracy: 0.6205 - val_loss: 0.6507
Epoch 18/25
132/132            44s 332ms/step -
accuracy: 0.6242 - loss: 0.6535 - val_accuracy: 0.6205 - val_loss: 0.6505
Epoch 19/25
132/132            44s 337ms/step -
accuracy: 0.6217 - loss: 0.6551 - val_accuracy: 0.6205 - val_loss: 0.6491
Epoch 20/25
132/132            575s 4s/step -
```

```
accuracy: 0.6121 - loss: 0.6579 - val_accuracy: 0.6205 - val_loss: 0.6478
Epoch 21/25
132/132            42s 318ms/step -
accuracy: 0.6304 - loss: 0.6499 - val_accuracy: 0.6205 - val_loss: 0.6469
Epoch 22/25
132/132            42s 320ms/step -
accuracy: 0.6155 - loss: 0.6575 - val_accuracy: 0.6205 - val_loss: 0.6465
Epoch 23/25
132/132            43s 328ms/step -
accuracy: 0.6143 - loss: 0.6562 - val_accuracy: 0.6205 - val_loss: 0.6452
Epoch 24/25
132/132            44s 332ms/step -
accuracy: 0.6213 - loss: 0.6503 - val_accuracy: 0.6205 - val_loss: 0.6451
Epoch 25/25
132/132            44s 335ms/step -
accuracy: 0.6089 - loss: 0.6577 - val_accuracy: 0.6205 - val_loss: 0.6503
Epoch 1/15
132/132            52s 373ms/step -
accuracy: 0.5199 - loss: 0.7086 - val_accuracy: 0.6205 - val_loss: 0.6496
Epoch 2/15
132/132            49s 369ms/step -
accuracy: 0.6219 - loss: 0.6537 - val_accuracy: 0.6205 - val_loss: 0.6459
Epoch 3/15
132/132            4125s 31s/step -
accuracy: 0.6141 - loss: 0.6487 - val_accuracy: 0.6224 - val_loss: 0.6293
Epoch 4/15
132/132            44s 332ms/step -
accuracy: 0.6468 - loss: 0.6344 - val_accuracy: 0.6461 - val_loss: 0.6070
Epoch 5/15
132/132            246s 2s/step -
accuracy: 0.6628 - loss: 0.6196 - val_accuracy: 0.6755 - val_loss: 0.5905
Epoch 6/15
132/132            91s 687ms/step -
accuracy: 0.6707 - loss: 0.6126 - val_accuracy: 0.7059 - val_loss: 0.5750
Epoch 7/15
132/132            45s 342ms/step -
accuracy: 0.6828 - loss: 0.6020 - val_accuracy: 0.7125 - val_loss: 0.5659
Epoch 8/15
132/132            46s 349ms/step -
accuracy: 0.6863 - loss: 0.5957 - val_accuracy: 0.7334 - val_loss: 0.5512
Epoch 9/15
132/132            47s 352ms/step -
accuracy: 0.7152 - loss: 0.5779 - val_accuracy: 0.7296 - val_loss: 0.5452
Epoch 10/15
132/132            47s 355ms/step -
accuracy: 0.7162 - loss: 0.5751 - val_accuracy: 0.7571 - val_loss: 0.5364
Epoch 11/15
132/132            47s 358ms/step -
```

```
accuracy: 0.7288 - loss: 0.5619 - val_accuracy: 0.7457 - val_loss: 0.5279
Epoch 12/15
132/132            48s 365ms/step -
accuracy: 0.7402 - loss: 0.5604 - val_accuracy: 0.7628 - val_loss: 0.5183
Epoch 13/15
132/132            547s 4s/step -
accuracy: 0.7549 - loss: 0.5505 - val_accuracy: 0.7704 - val_loss: 0.5090
Epoch 14/15
132/132            45s 341ms/step -
accuracy: 0.7548 - loss: 0.5435 - val_accuracy: 0.7751 - val_loss: 0.5053
Epoch 15/15
132/132            46s 350ms/step -
accuracy: 0.7657 - loss: 0.5307 - val_accuracy: 0.7884 - val_loss: 0.4974
33/33            9s 259ms/step

 Evaluation for EfficientNetB0 (Fine-Tuned)
              precision    recall  f1-score   support

           0       0.91      0.49      0.64       400
           1       0.76      0.97      0.85       654

    accuracy                           0.79      1054
   macro avg       0.83      0.73      0.74      1054
weighted avg       0.81      0.79      0.77      1054

Confusion Matrix:
 [[197 203]
 [ 20 634]]
```

[ ]:

```

```

0.1.9 Custom CNN vs Transfer Learning Comparison

Aspect	Custom CNN	Transfer Learning (e.g. VGG16, ResNet50, EfficientNetB0)
Base Model	Built from scratch (`create_custom_cnn()`)	Pre-trained on ImageNet (`weights='imagenet'`)
Weights	Randomly initialised	Loaded from pre-trained model
Feature Extraction	Learns all features from your dataset	Uses existing rich features from ImageNet
Training Strategy	Entire model trained from the start	First train custom top layers, then fine-tune deeper layers
Fine-Tuning	Not applicable unless implemented manually	Optional second phase: unfreeze top layers and retrain
Training Time	Typically longer, depending on architecture	Faster initially; may increase during fine-tuning

Aspect	Custom CNN	Transfer Learning (e.g. VGG16, ResNet50, EfficientNetB0)
Performance	Varies based on design and dataset	Often higher, especially on small or mid-sized datasets
Flexibility	Fully customisable design	Limited by structure of the base model
Typical Use Case	When experimenting or needing full control	When seeking strong performance with minimal tuning

[54]:
```python
def evaluate_model(model, X_test, y_test, name):
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    print(f"\n=Evaluation for {name}")
    print(classification_report(y_test, y_pred))
    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:\n", cm)
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[102]:
```python
# Evaluate all fine-tuned models

evaluate_model(vgg_model, X_test, y_test, "VGG16 (Fine-Tuned)")
evaluate_model(resnet_model, X_test, y_test, "ResNet50 (Fine-Tuned)")
evaluate_model(efficient_model, X_test, y_test, "EfficientNetB0 (Fine-Tuned)")
```

```
33/33                 69s 2s/step

=Evaluation for VGG16 (Fine-Tuned)
              precision    recall  f1-score   support

           0       1.00      0.99      0.99       400
           1       0.99      1.00      1.00       654

    accuracy                           1.00      1054
   macro avg       1.00      0.99      0.99      1054
weighted avg       1.00      1.00      1.00      1054

Confusion Matrix:
 [[395   5]
```

```
 [  0 654]]
33/33               21s 632ms/step

=Evaluation for ResNet50 (Fine-Tuned)
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       400
           1       0.99      0.97      0.98       654

    accuracy                           0.98      1054
   macro avg       0.97      0.98      0.97      1054
weighted avg       0.98      0.98      0.98      1054

Confusion Matrix:
 [[392   8]
 [ 18 636]]
33/33               9s 261ms/step

=Evaluation for EfficientNetB0 (Fine-Tuned)
              precision    recall  f1-score   support

           0       0.91      0.49      0.64       400
           1       0.76      0.97      0.85       654

    accuracy                           0.79      1054
   macro avg       0.83      0.73      0.74      1054
weighted avg       0.81      0.79      0.77      1054

Confusion Matrix:
 [[197 203]
 [ 20 634]]
```

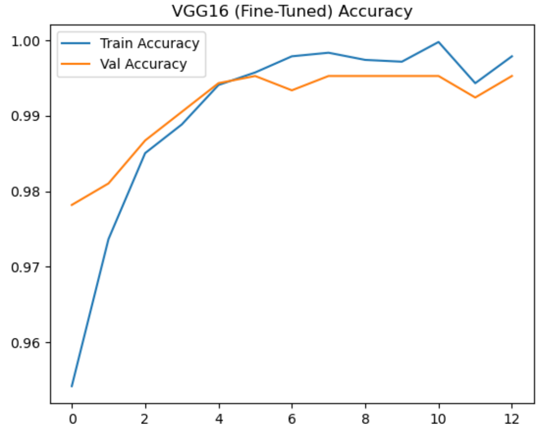
[ ]:

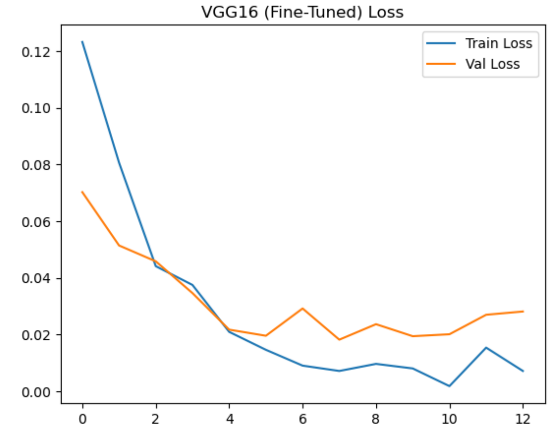
[ ]:

0.1.10 Visualise Training Histories.

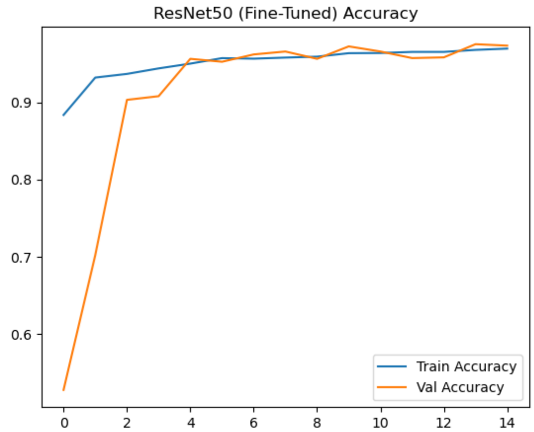
[64]:

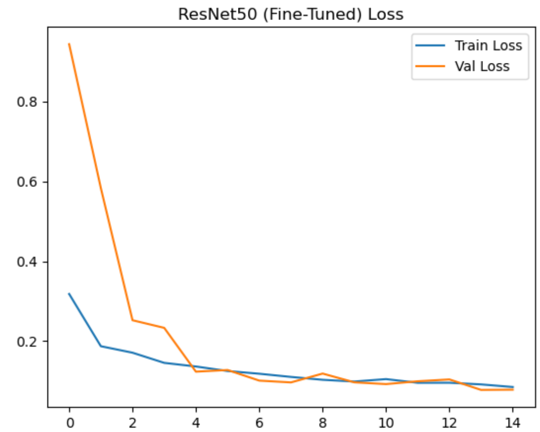
```
plot_training(vgg_fine, "VGG16 (Fine-Tuned)")
```



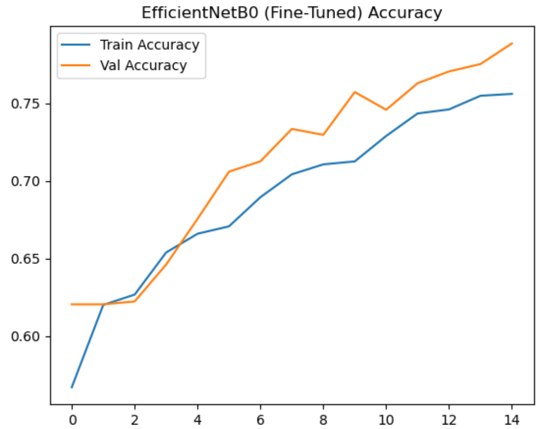


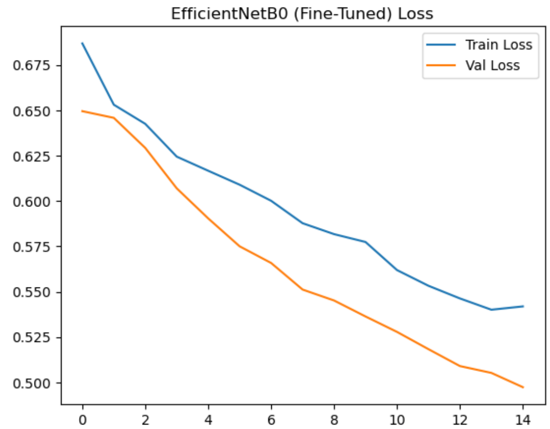
[65]:
```
plot_training(resnet_fine, "ResNet50 (Fine-Tuned)")
```





[67]:
```
plot_training(efficient_fine, "EfficientNetB0 (Fine-Tuned)")
```



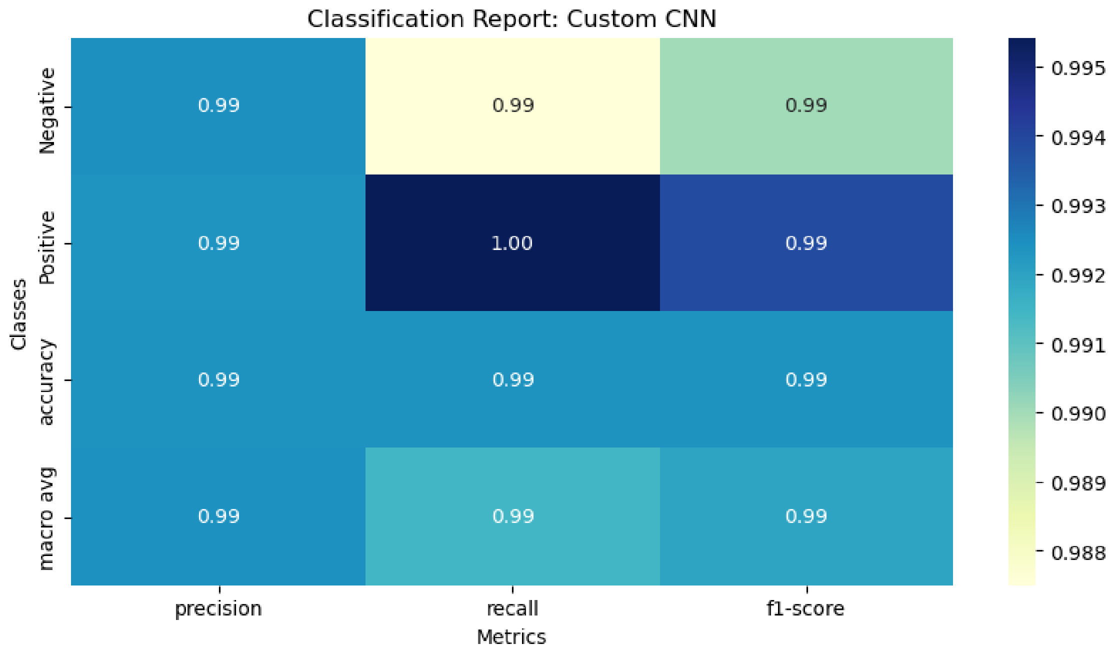


[70]:
```
def plot_classification_report(model, X_test, y_test, class_names, title):
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    report = classification_report(y_test, y_pred, target_names=class_names,␣
 ↪output_dict=True)
    df = pd.DataFrame(report).transpose()

    plt.figure(figsize=(10, 5))
    sns.heatmap(df.iloc[:-1, :-1], annot=True, cmap="YlGnBu", fmt=".2f")
    plt.title(f'Classification Report: {title}')
    plt.ylabel('Classes')
    plt.xlabel('Metrics')
    plt.show()
```

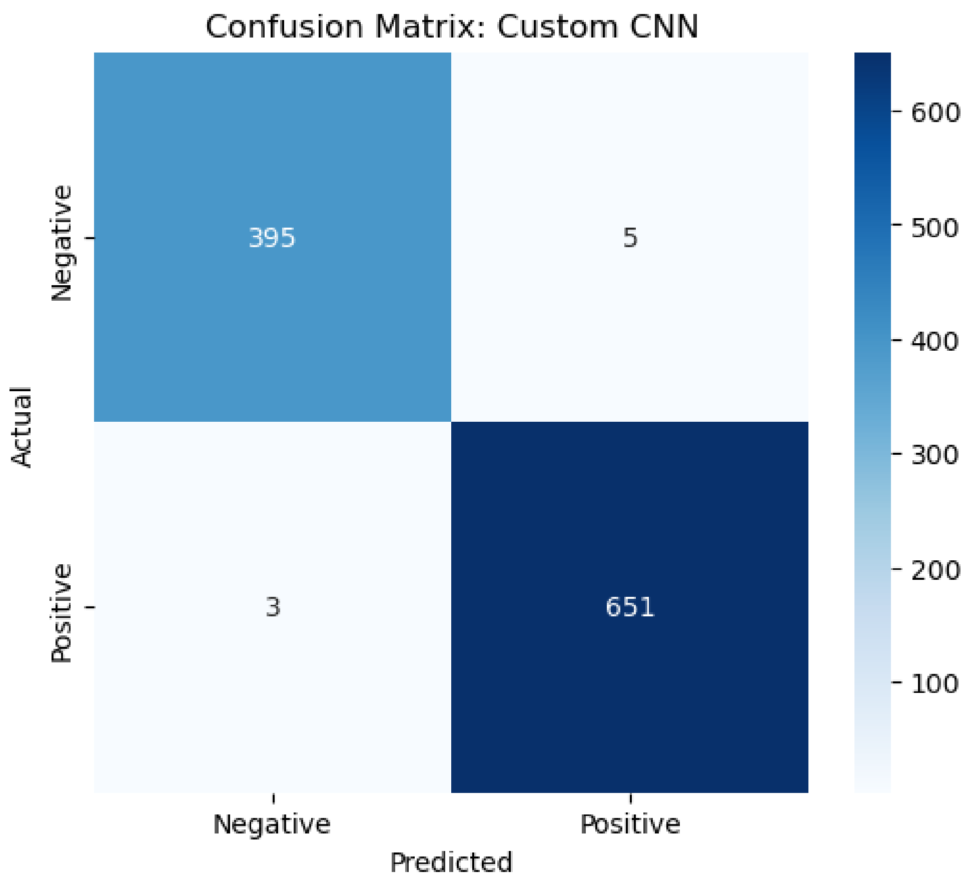
[95]:
```
def plot_confusion(model, X_test, y_test, class_names, title):
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,␣
 ↪yticklabels=class_names)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Confusion Matrix: {title}')
    plt.show()
```

[92]:
```
class_names = ['Negative', 'Positive']

plot_classification_report(custom_model, X_test, y_test, class_names, "Custom␣
 ↪CNN")
plot_confusion(custom_model, X_test, y_test, class_names, "Custom CNN")
```

```
33/33              3s 88ms/step
```

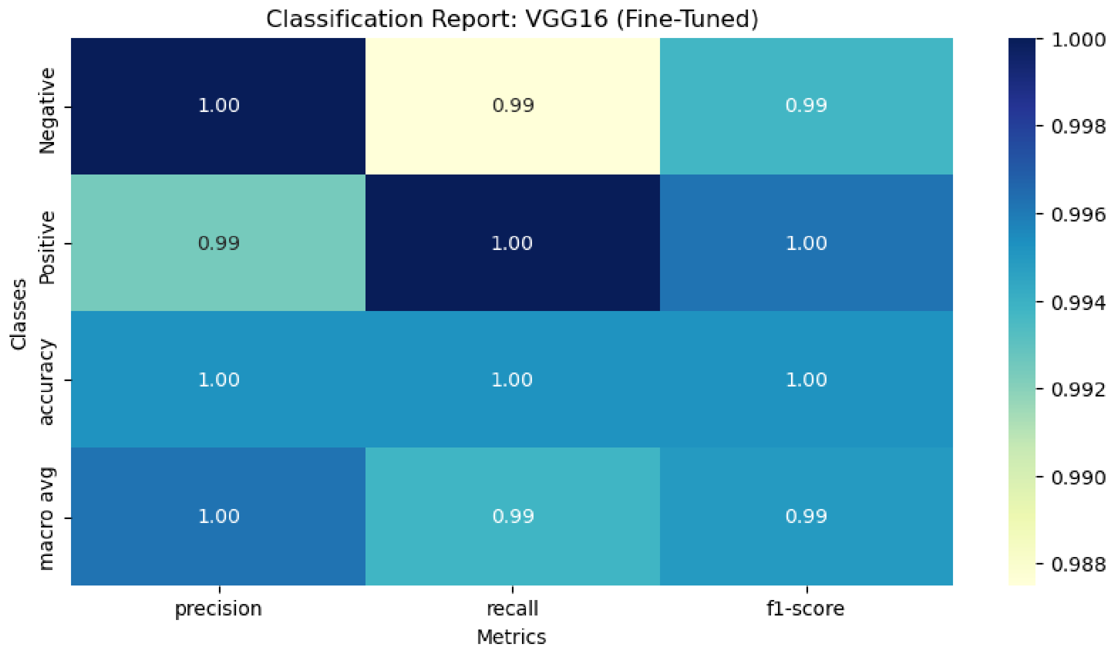



```
33/33                 3s 92ms/step
```

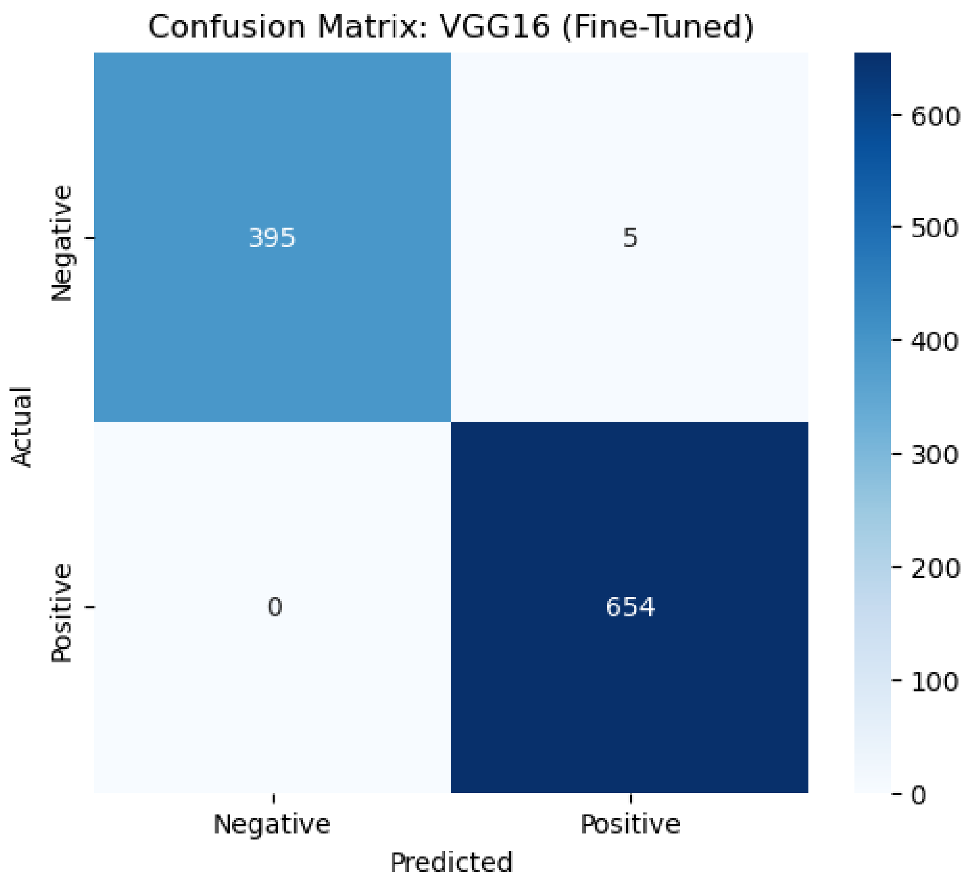


[94]:
```
plot_classification_report(vgg_model, X_test, y_test, class_names, "VGG16␣
↪(Fine-Tuned)")
plot_confusion(vgg_model, X_test, y_test, class_names, "VGG16 (Fine-Tuned)")
```

```
33/33                69s 2s/step
```

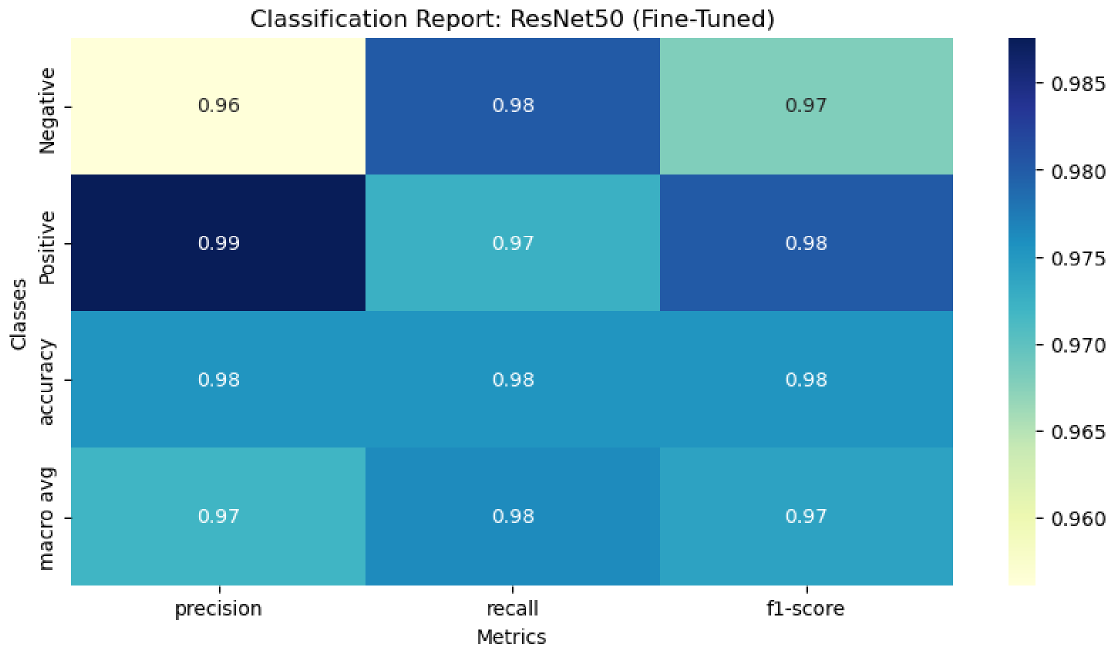



```
33/33                 71s 2s/step
```

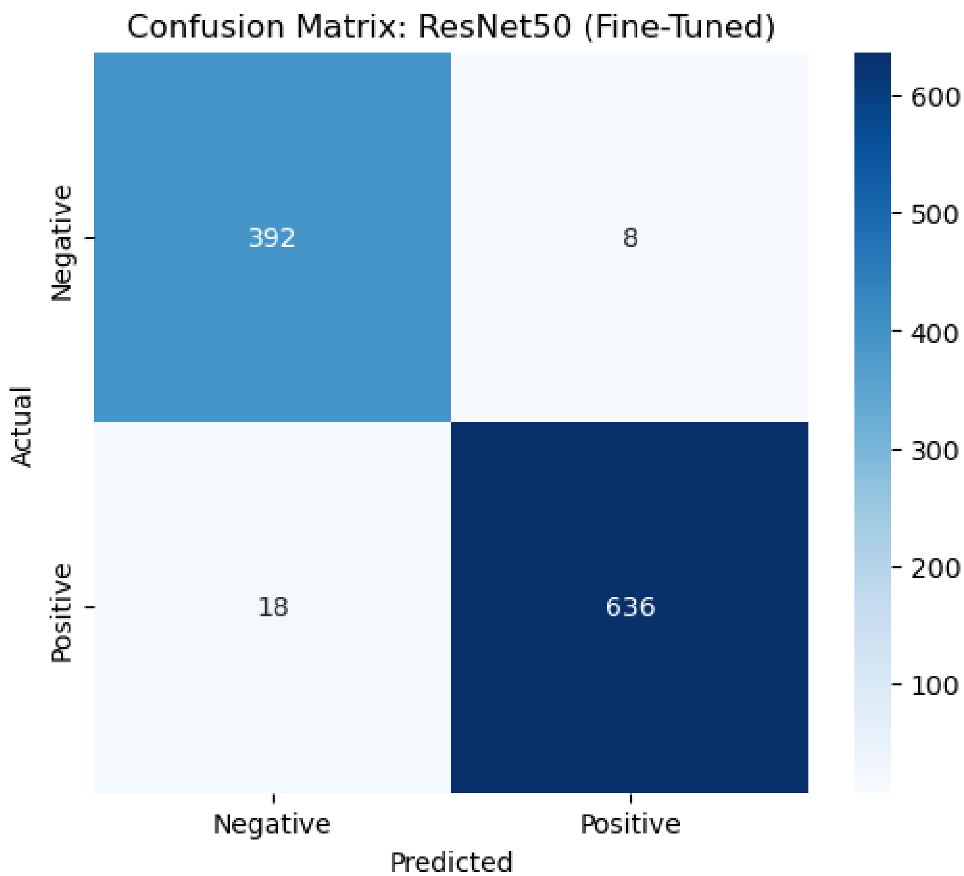


[78]:
```
plot_classification_report(resnet_model, X_test, y_test, class_names, "ResNet50␣
↪(Fine-Tuned)")
plot_confusion(resnet_model, X_test, y_test, class_names, "ResNet50␣
↪(Fine-Tuned)")
```

```
33/33                21s 638ms/step
```

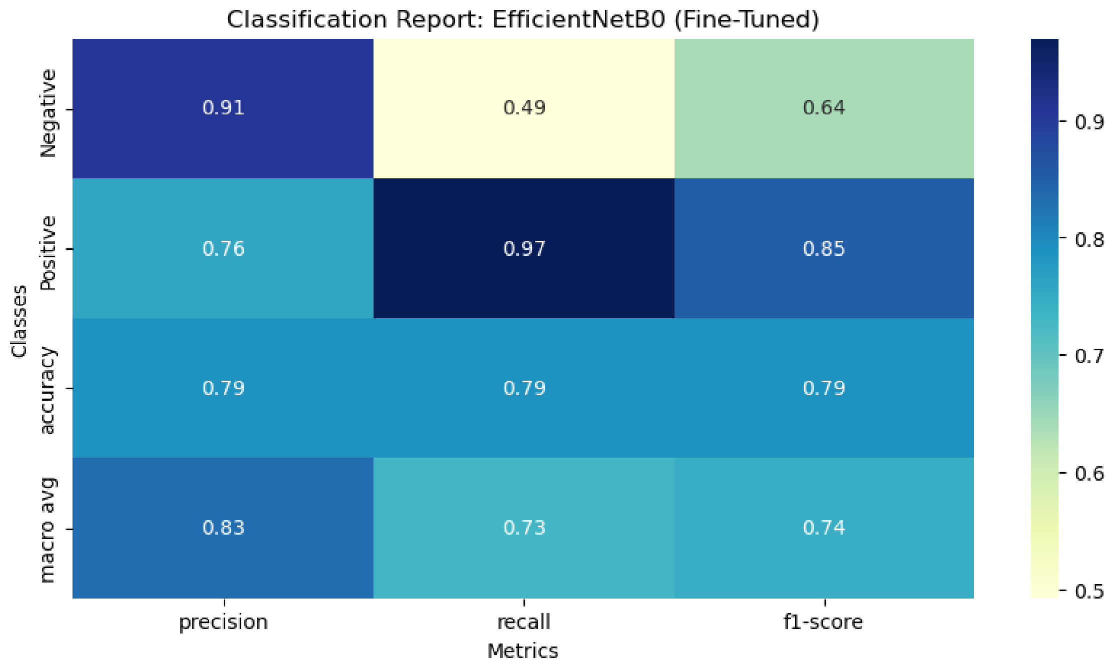



```
33/33                 21s 630ms/step
```

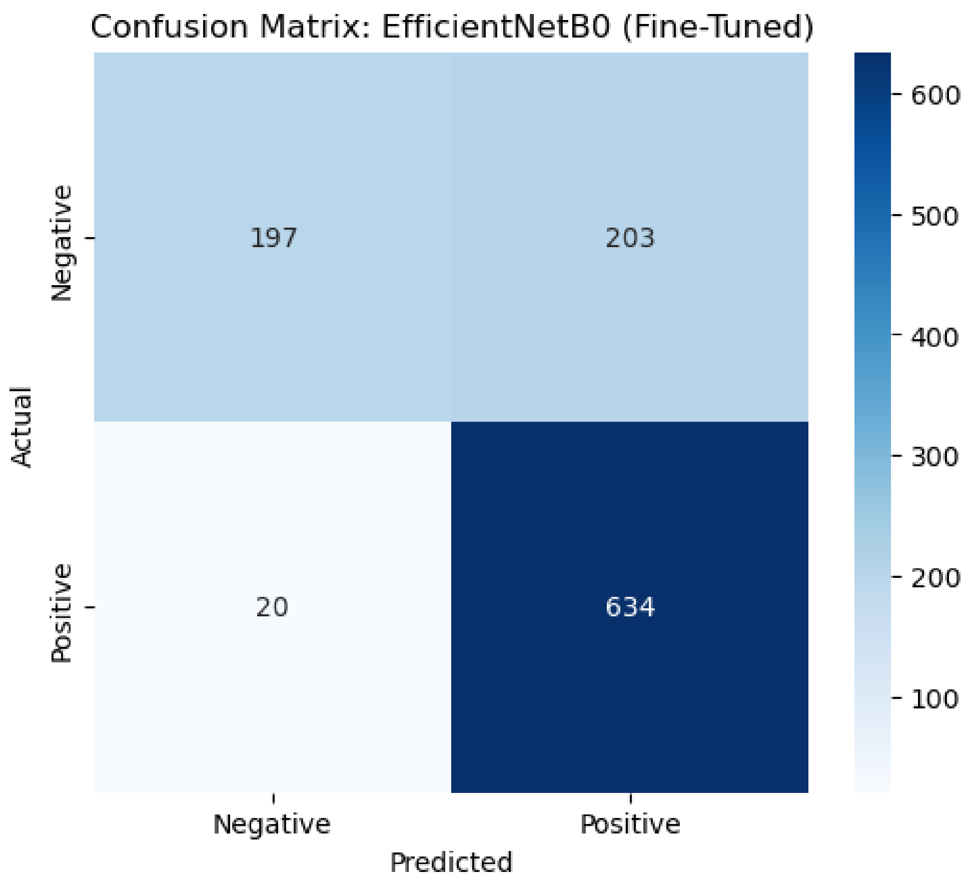


[79]:
```
plot_classification_report(efficient_model, X_test, y_test, class_names,␣
  ↪"EfficientNetB0 (Fine-Tuned)")
plot_confusion(efficient_model, X_test, y_test, class_names, "EfficientNetB0␣
  ↪(Fine-Tuned)")
```

```
33/33                9s 258ms/step
```




```
33/33                 8s 255ms/step
```


Confusion Matrix: EfficientNetB0 (Fine-Tuned)
Actual
Negative
Positive
197
203
20
634
Negative
Positive
Predicted
600
500
400
300
200
100