Deep Learning in Fake Face & Facial Expression Recognition

Louise Liu, Kira Luo, Sydney Li, Grace Xie

Table of contents

01 **Abstract**

problem, approach, and describe results

02 **Exploratory Data Analysis**

Summarize & Visualize Data

03 **Model Training and Evaluation**

two different cognitive problems with four models

04 **Model Operations**

Model architecture & parameter update

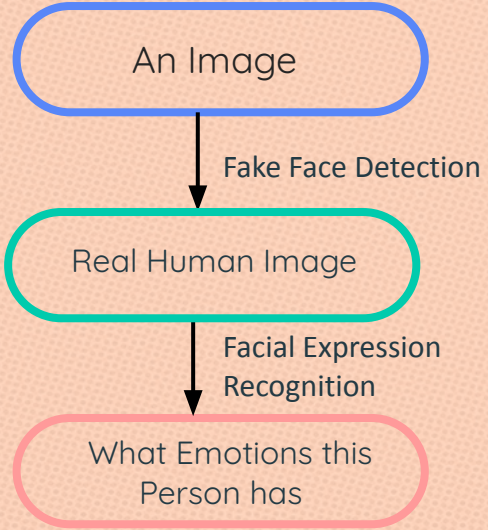
05 **Conclusion**

Introduction

<u>Problem:</u> Detecting fake face images and **facial expression recognition** are significant tasks in applications spanning various domains such as human-computer interaction, surveillance, healthcare, and entertainment.

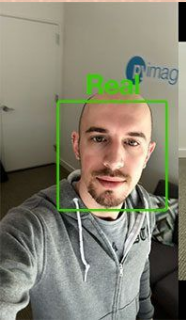
<u>Approach and Desired Results:</u> This project aims to develop a robust combination of fake face detection and facial expression recognition system using deep learning techniques **Custom CNNs and Xception** detect fake faces or put them into distinct categories such as happy, sad, angry, fear, surprise, and neutral.

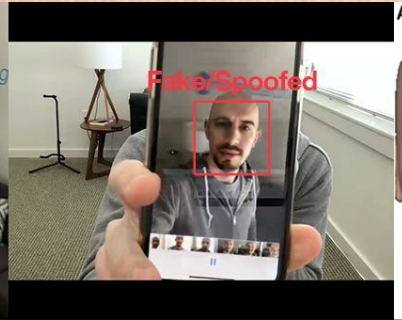
Project Flow

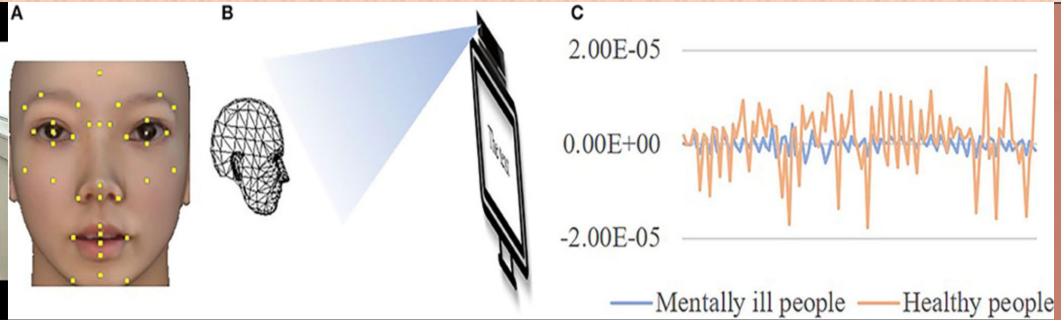


Possible Applications

- **Digital Media and Entertainment:** In the media industry, detecting fake faces can protect against the misuse of deepfake technology, which could be used to create misleading or malicious content.
- **Mental Health Monitoring:** FER (facial expression recognition) can help monitor patients' emotional states, providing valuable insights for mental health professionals. It can aid in detecting depression, anxiety, or stress levels.
- **Behavioral Analysis:** Detecting fake faces and FER can both be used in surveillance systems to detect suspicious behavior or emotional distress, enhancing public safety and security measures.







Dataset Overview

Face Image Data

Composed of 2 datasets coming from kaggle

Deepfake_faces Dataset

Detecting fake vs real face from all images

Metadata.csv

Includes 95634 images' attributes like videoname, orginial_width, original_height, and label (fake vs real)

Faces_224 folder

With 95634 images pictures from videos

FER Dataset

approximately 36K images that are 48x48 pixel grayscale faces. Each image corresponds to a facial expression in one of six categories such as happy, sad, angry, and etc.

Train Images (80%)

Val Images (20%)

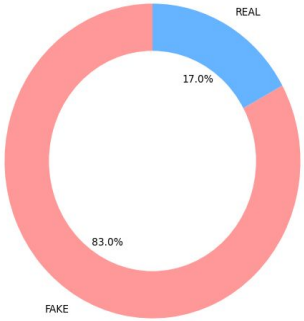
Problem 1:

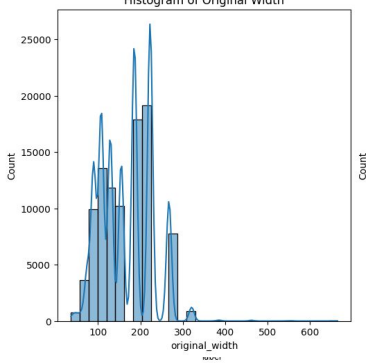
Fake Face Detection

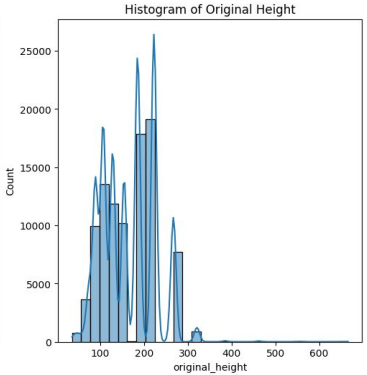
- Exploratory Data Analysis
- Model Training & Evaluation
- Model Operations

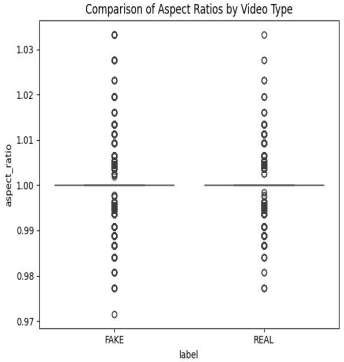
EDA- Fake Face Detection

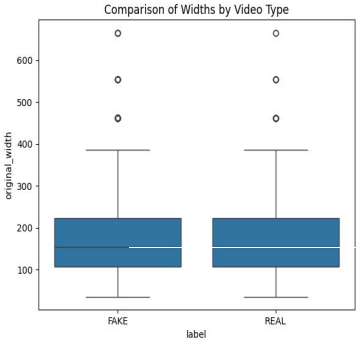
- **Data Source**
  - https://www.kaggle.com/datasets/dagnelies/deepfake-faces/data
- **Variables of Interest:**
  - Response Variable: Image Label (Real vs Fake)
  - “Predictor Variables”: features extracted by the network layers in the models from the 95634 input images
- **Imbalanced Dataset:** Fake Images accounts for 83% of all images. Thus we **randomly sampled 1000 for each label** (REAL v.s. FAKE) to gain a balanced dataset.
- **Histograms of Images Height and Widths** are normally distributed, mostly clustering around certain sizes.
- **Box and whisker plot**: show similar distributions of image widths, heights, and aspect ratios between fake and real images, suggesting little differentiation in these dimensions by image authenticity.

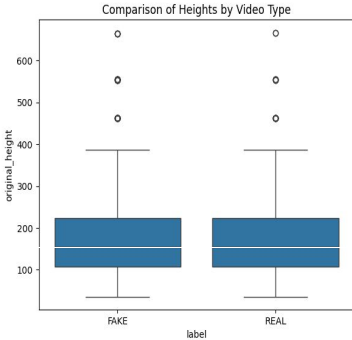






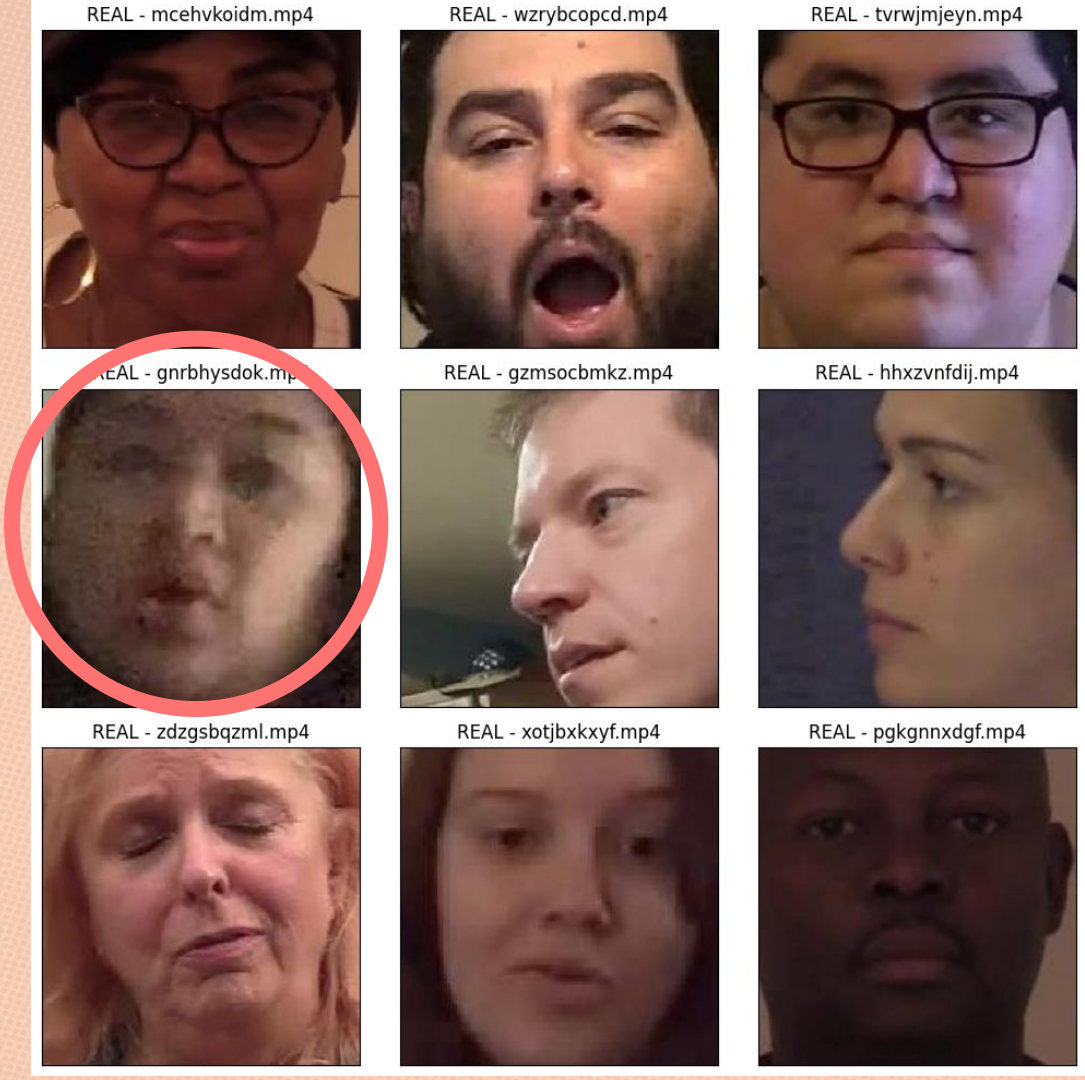






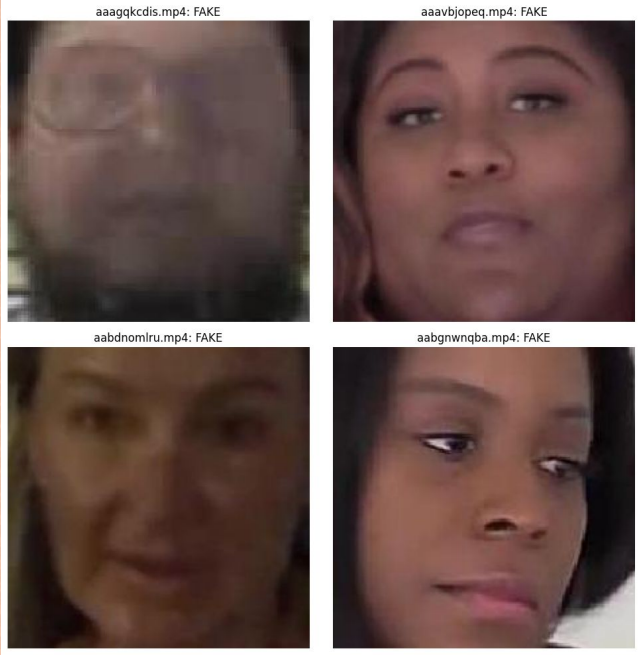
EDA Continued

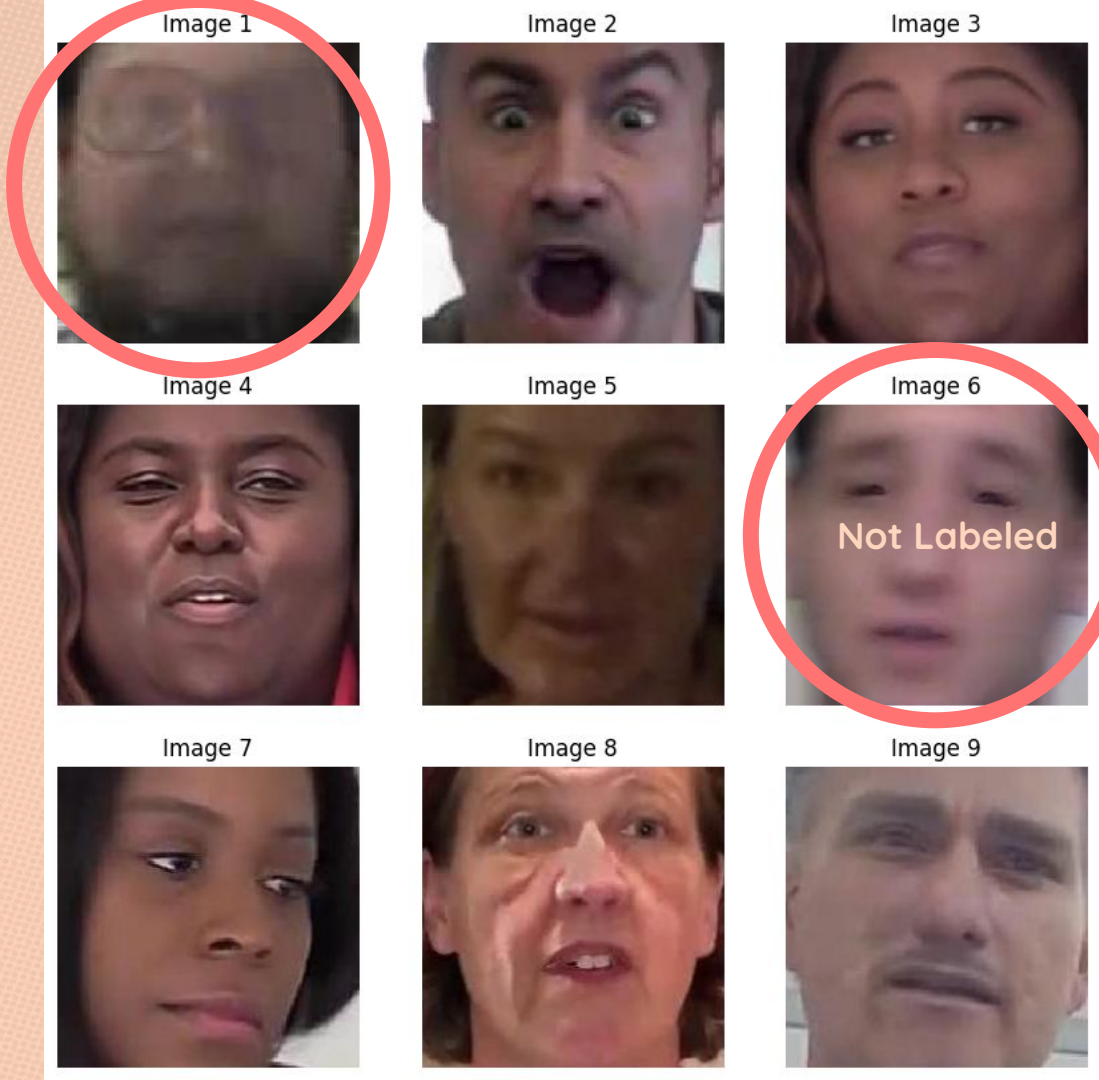
- All these pictures are real
- But one of them looks fake
- **Key Takeaway**: it is hard to tell whether a face is fake when the picture has low resolution



EDA Continued

- Guess which of the 9 images are fake?





Problem 1 - Fake Detection

- **Cognitive Problem:** Classifies faces images into “Real” OR “Fake”
- **Dataset Used:** Deepfake_faces Dataset
- **Models Used:**

1. **Xception Model:**
    - **Why Used:** Xception leverages depth wise separable convolutions, making it highly efficient at capturing details in images
    - **Pre-training on ImageNet** so the robust hierarchy of image features it learned can be effectively generalize to new images and identify inconsistencies typical of fake faces, such as unnatural skin textures, odd facial contours, or inconsistencies in lighting.
    - **No Top Layer:** The top layers (typically fully connected layers) are excluded (include_top=False). This allows for custom layers to be added for specific tasks, in this case, binary classification.
    - **Global Average Pooling Layer**
    - **Dense Layer with Sigmoid Activation**

Problem 1 - Fake Detection

2. <u>Custom CNN Model</u> <u>(compiled with the Adam optimizer)</u>

- **Why Used:** The convolutional layers in CNN are capable of extracting intricate features from images, which is crucial in identifying the often subtle manipulations involved in deepfakes.
- **Input Layer:** takes an input with the shape (224, 224, 3), which corresponds to color images of size 224x224 pixels. The "3" represents the three color channels (RGB).
- **3 Convolutional Layers:**
  - First Convolutional Layer: 32 filters of size (3x3) with ReLU activation.
  - Second Convolutional Layer: 64 filters of size (3x3) with ReLU activation.
  - Third Convolutional Layer: 128 filters of size (3x3) with ReLU activation.
- **Max Pooling Layers & Flattening**
- **Dense Layers:**
  - First Dense Layer: 512 neurons with ReLU activation.
  - Dropout Layer: Dropout at 50% probability
  - Output Layer: A single neuron with a sigmoid activation function.

Model Evaluation

Customized CNN

Accuracy: 50% (random guess)

Class “Fake”: Precision: 0.5, Recall: 1, F1-Score: 0.67

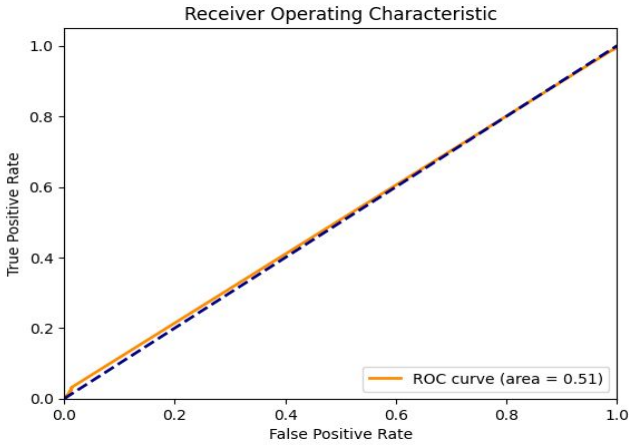
Class “Real”: Precision: 0.33, Recall: 0, F1-Score: 0

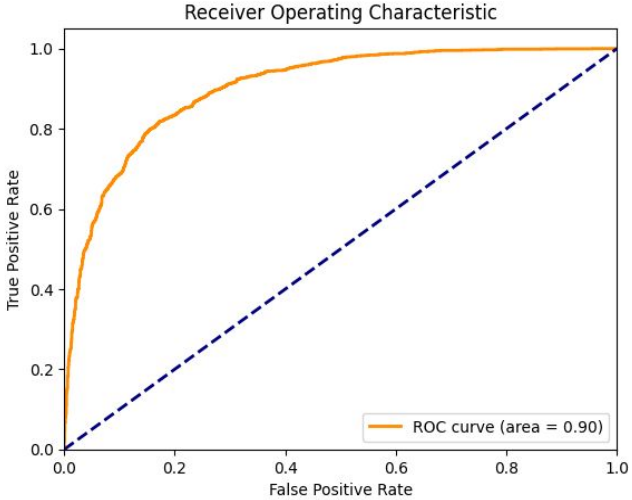
Xception Model

Accuracy: 76% (much better)

Class “Fake”: Precision: 0.93, Recall: 0.57, F1-Score: 0.7

Class “Real”: Precision: 0.69, Recall: 0.96, F1-Score: 0.8





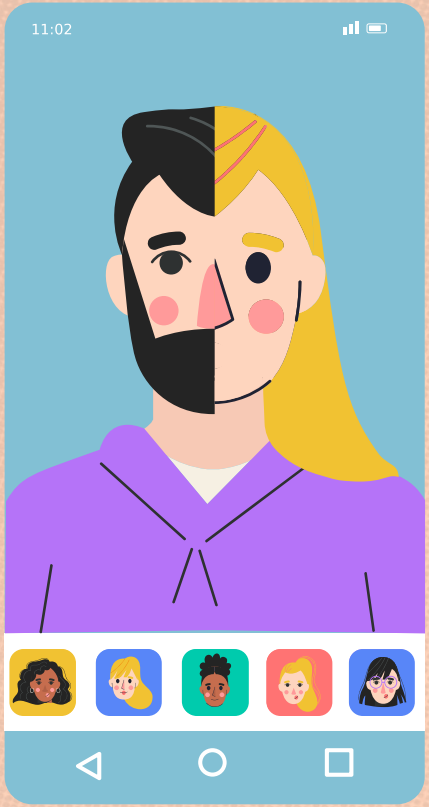
Model Deployment

Deployment Mechanism

1. Prepare the Models and upload to S3
2. Develop the Streamlit App
3. Deploy the Streamlit App via AWS EC2 or ECS
4. Security and Access Management
5. Monitoring and Logging
6. Update and Maintenance

Maintenance Plans

- **Model Maintenance Plan**: Use cloud-specific tools like AWS CloudWatch and Azure Monitor to continuously monitor the model's performance and operational metrics.
- **Parameter Update Process**: Set up an automated pipeline that can trigger retraining of the model based on specific criteria; Incorporate user feedback and real-world outcomes into the training data to continuously improve the model.



Problem 2:

Facial Expression Recognition

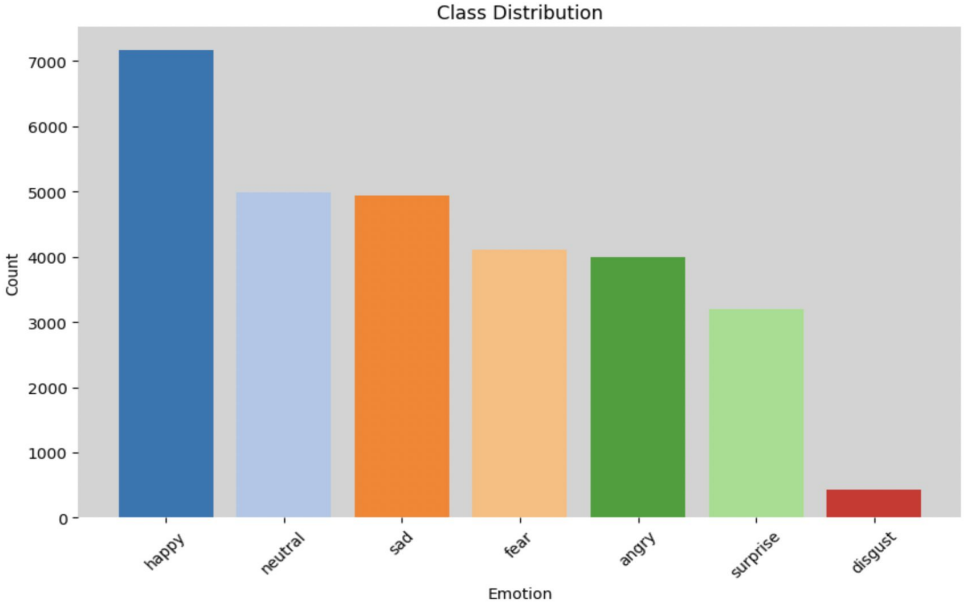
- Exploratory Data Analysis
- Model Training & Evaluation
- Model Operations

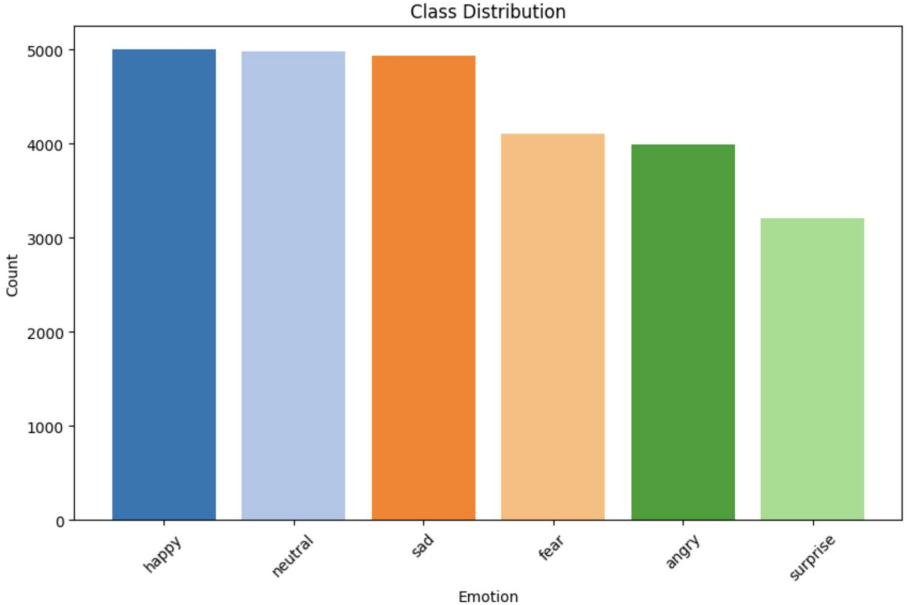
Problem 2 - Facial Expression

- **Problem:** Classify Face Expressions into 6 classes: ***"happy", "sad", "angry", "fear", "neutral"***, *and* ***"surprise"***.
- **Dataset Used:** Kaggle
- **Approach:** Used VGG16 and ResNet50 pre-trained models for transfer learning, and a robust CNN with multiple convolutional layers and dropout.
- **Evaluation**: Plotted training and validation accuracy, loss, precision, recall, and F1-score to monitor model performance.

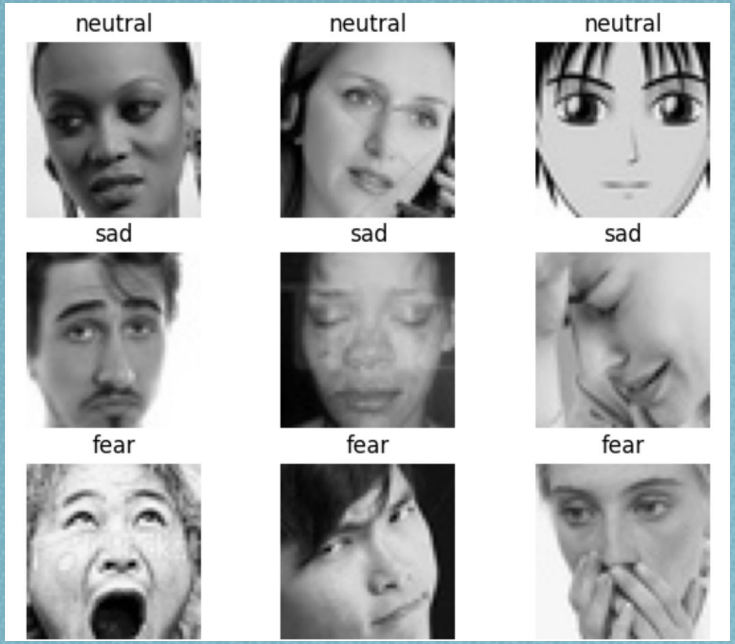
EDA- Facial Expression Recognition

- Numbers of expressions for each class
- Down to 5000 samples of ‘Happy’ to achieve a balanced dataset
- Remove “Disgust”





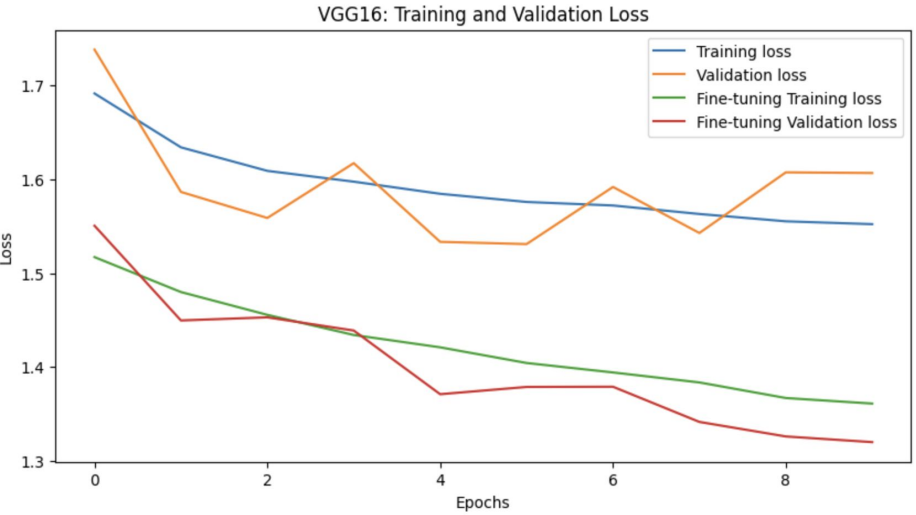
EDA- Facial Expression Recognition

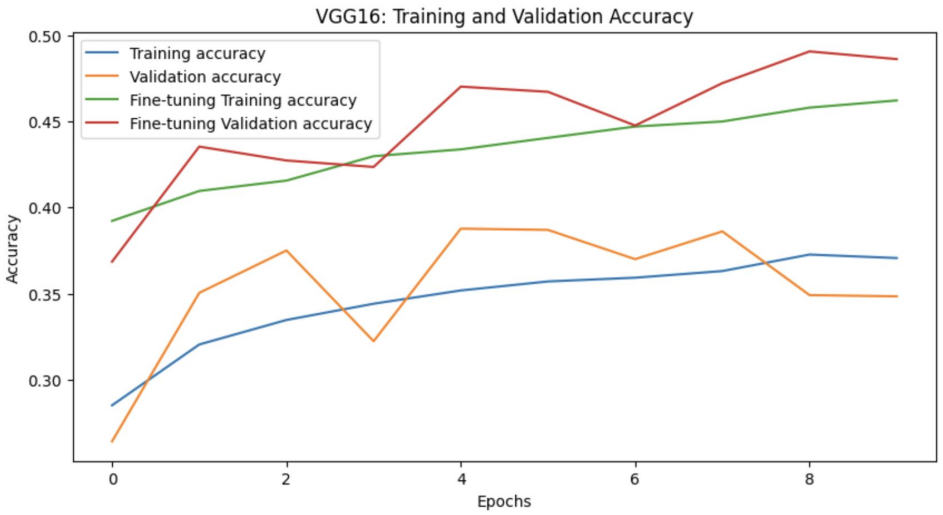


Sample of Images

Outputs - Pretrained VGG16

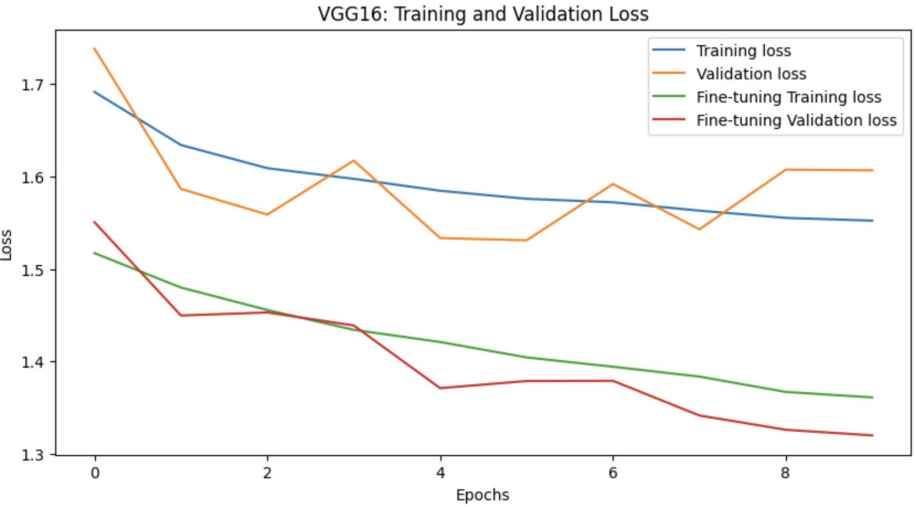
- Accuracy: 0.354
- Fine-Tuning Accuracy: 0.449

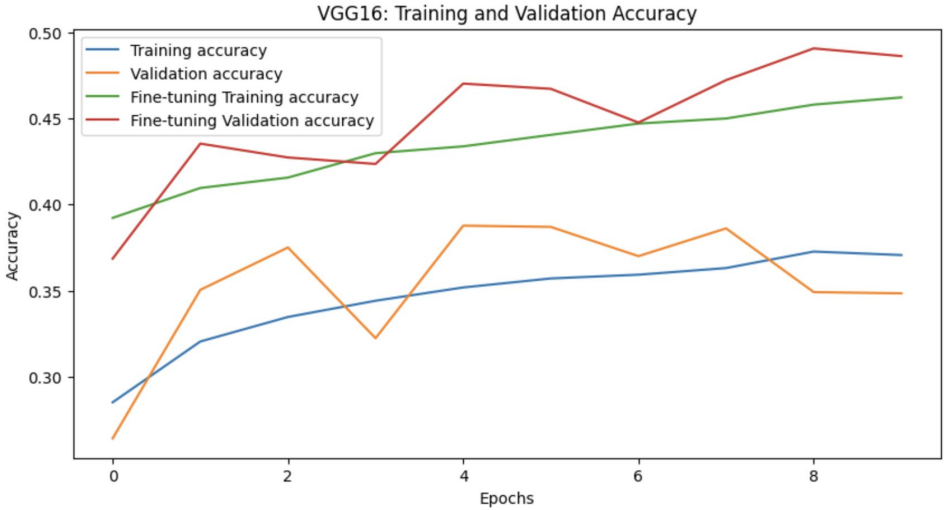




Outputs - Pretrained Resnet50

- Accuracy: 0.195
- Fine-Tuning Accuracy: 0.240



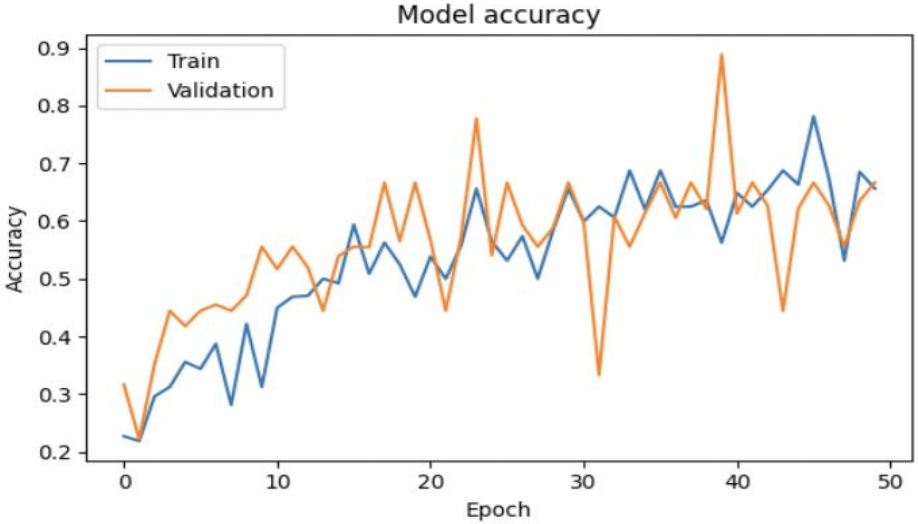


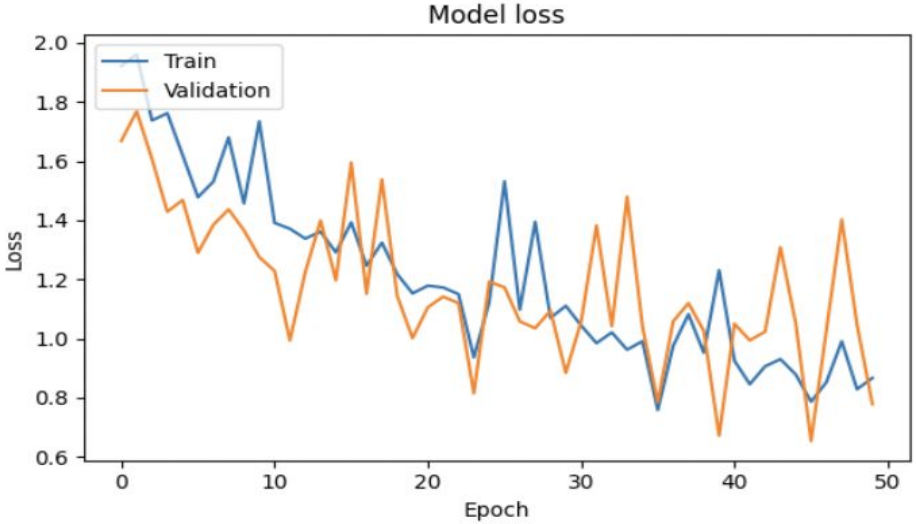
CNN - Model Architecture

- Input layer: Accepts images of size 48 × 48 × 3
- 4 convolutional layers:
    - Batch normalization and ReLU activation
    - Max pooling
    - Dropout with 0.25 rate
- 2 fully connected layers
- Flattening layer: converts the 3D feature maps into a 1D feature vector.

Outputs - CNN

- Accuracy: 0.634
- Loss: 1.022





Demo Time

Model Deployment

- S3 Bucket Storage Image
- AWS CloudWatch Monitor Performance Metrics
- AWS Lambda Train Model
- ECR (Docker Image) → ECS → Web Application

Conclusion & Discussion

For Fake Detection:

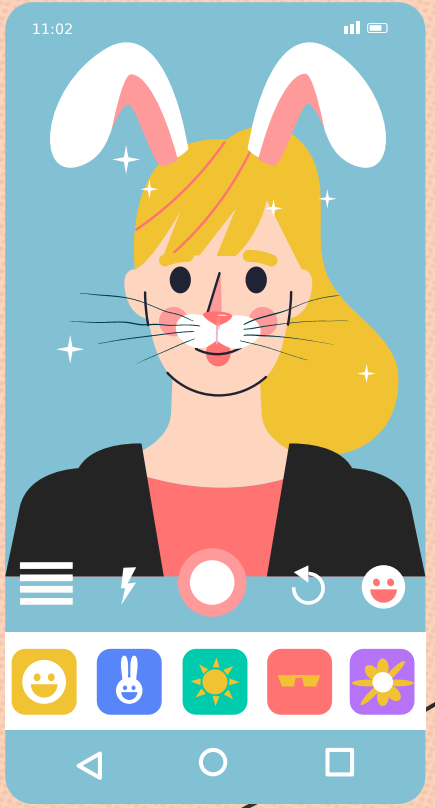
- Both models are affected by the imbalanced dataset even after data processing
- Xception model wins over customized CNN in terms accuracy and also all other metrics

For Facial Expression Recognition:

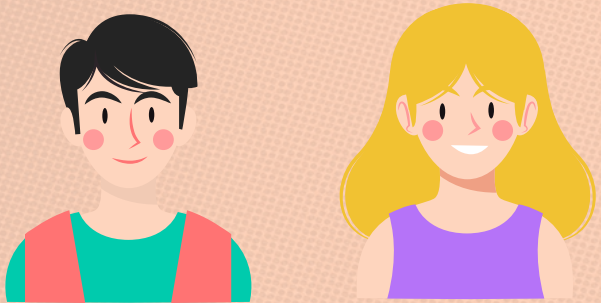
- Out of 3 models, CNN wins over Pretrained VGG16 and Pretrained Resnet 50 in terms of accuracy

For the combined models, it could be possibly used for the following areas:

- Improved User Experience
- Security Enhancements
- Healthcare Applications



Thank you for listening



Appendix

Abstract

Project Objective: This project aims to differentiate between real and fake human images and recognize facial expressions, enhancing security and authenticity in digital interactions. It's crucial for combating deepfake technology and improving human-computer interaction, especially in fields like online communication, security systems, and digital media.

Problems and Approach:

1. Fake Face Detection → Customized CNN v.s Xception Model
2. Facial Expression Recognition → VGG 16 v.s Resnet50 v.s CNN

Desired Result:

1. High Accuracy in Fake Face Detection
2. Robust Facial Expression Recognition
3. Generalization Across Diverse Datasets
4. Real-Time Processing Capability

Problem 1 Appendix

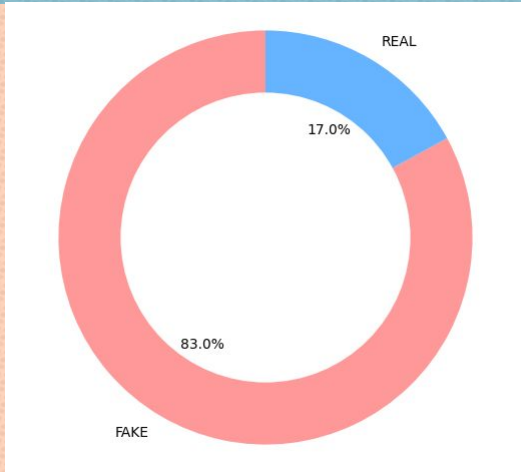
Exploratory Data Analysis

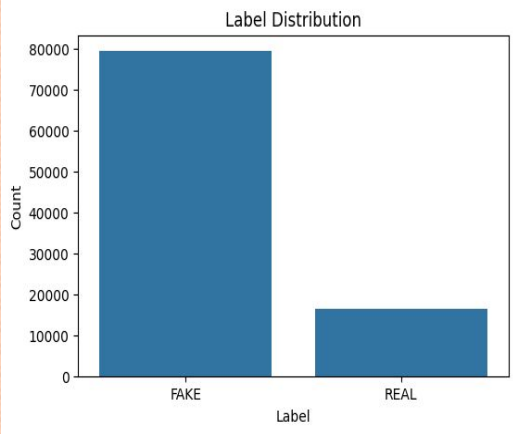
Dataste Composition:

1. Metadata.csv
2. Faces_224 folder (With 95634 images pictures from videos)

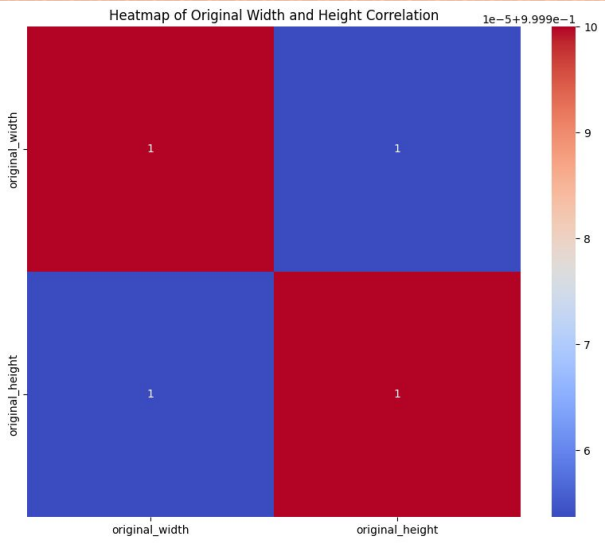
Details of Metadata.csv:

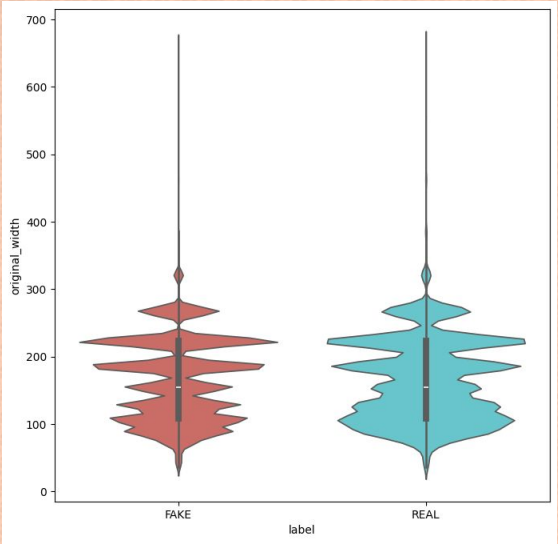
- **Size:** Includes 95634 images' attributes
- **Four Columns:** videoname, original_width, original_height, label, original
- **Shape:** (95634, 5)
- **Data Distribution:**
    - Count of 'FAKE' labels: 79341 (83%)
    - Count of 'REAL' labels: 16293 (17%)
- **Key Takeaway:** the dataset is highly imbalanced so upsampling/ downsampling or SMOTE is needed

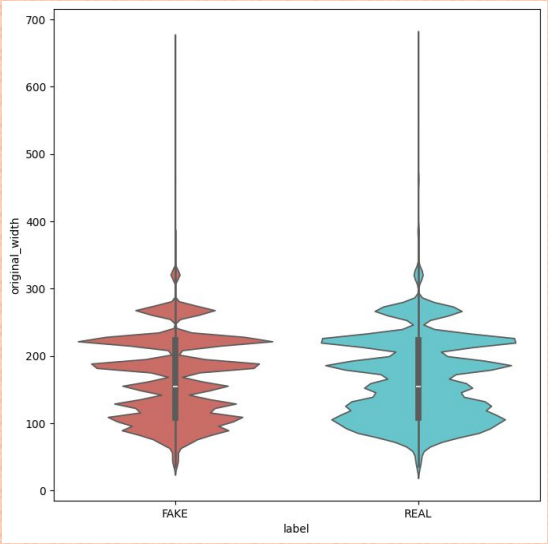




Exploratory Data Analysis



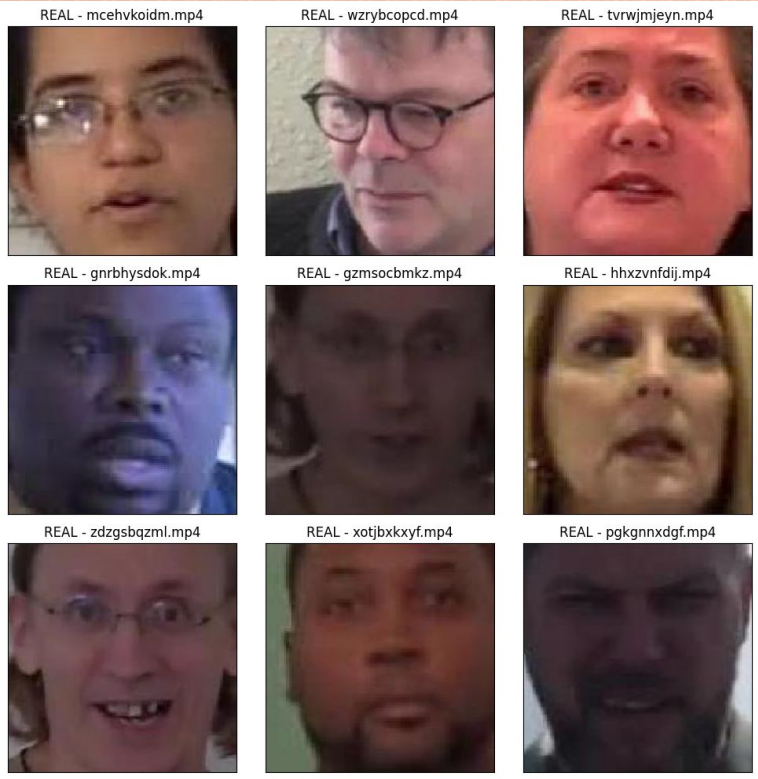




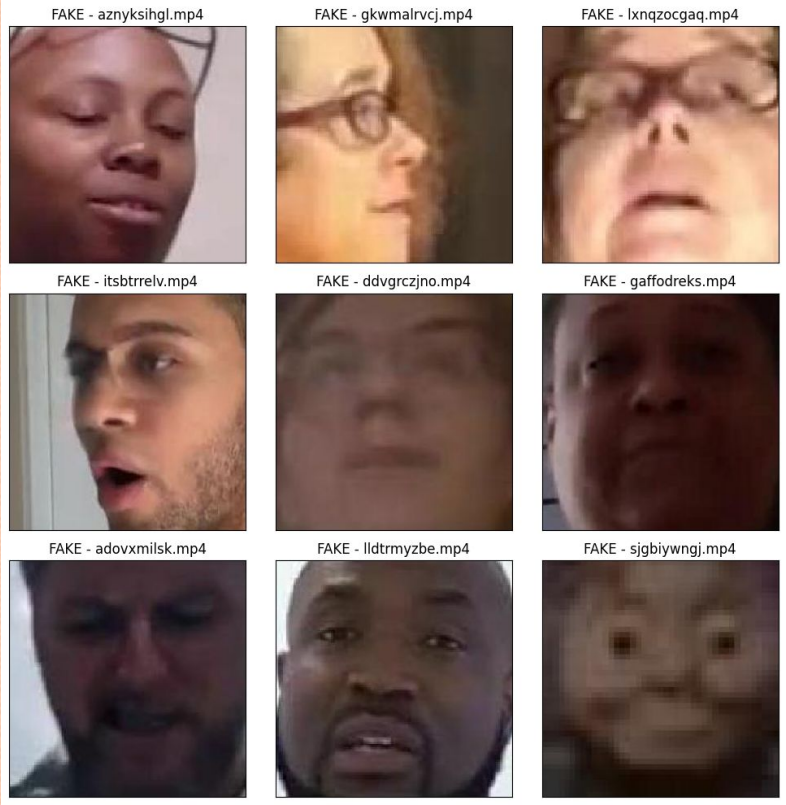
The heatmap shows a perfect positive correlation (value = 1) between original width and original height, indicating identical variance patterns. But this has no real implication in our case

The violin plots show similar distributions of original width and height for both fake and real images, with slight variations.

Exploratory Data Analysis



More examples
on
Real vs
Fake Images



Model Training

Model 1: Custom CNN Model

Description: It is a custom Convolutional Neural Network (CNN) built using TensorFlow's Keras API

Model Architecture: shown by the diagram on the top right

Training: 10 epoch with training accuracy around 0.49 and val accuracy with 0.5 (validation_split=0.2)

```
custom_cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 222, 222, 32)      896

 max_pooling2d (MaxPooling2  (None, 111, 111, 32)      0
 D)

 conv2d_1 (Conv2D)           (None, 109, 109, 64)      18496

 max_pooling2d_1 (MaxPoolin  (None, 54, 54, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 52, 52, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 26, 26, 128)       0
 g2D)

 flatten (Flatten)           (None, 86528)             0

 dense (Dense)               (None, 512)               44302848

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 44396609 (169.36 MB)
Trainable params: 44396609 (169.36 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Train Custom CNN Model
history_custom_cnn = custom_cnn_model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=10
)
```

```
Epoch 1/10
350/350 [==============================] - 159s 453ms/step - loss: 0.6932 - accuracy: 0.4990 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 2/10
350/350 [==============================] - 158s 451ms/step - loss: 0.6932 - accuracy: 0.4985 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 3/10
350/350 [==============================] - 158s 453ms/step - loss: 0.6932 - accuracy: 0.4974 - val_loss: 0.6931 - val_accuracy: 0.5000
Epoch 4/10
350/350 [==============================] - 158s 452ms/step - loss: 0.6939 - accuracy: 0.5030 - val_loss: 0.6933 - val_accuracy: 0.5000
Epoch 5/10
350/350 [==============================] - 159s 454ms/step - loss: 0.6934 - accuracy: 0.4969 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 6/10
350/350 [==============================] - 163s 465ms/step - loss: 0.6935 - accuracy: 0.4937 - val_loss: 0.6931 - val_accuracy: 0.5000
Epoch 7/10
350/350 [==============================] - 159s 456ms/step - loss: 0.6933 - accuracy: 0.4927 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 8/10
350/350 [==============================] - 158s 452ms/step - loss: 0.6932 - accuracy: 0.4988 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 9/10
350/350 [==============================] - 158s 452ms/step - loss: 0.6933 - accuracy: 0.4991 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 10/10
350/350 [==============================] - 162s 462ms/step - loss: 0.6931 - accuracy: 0.5070 - val_loss: 0.6932 - val_accuracy: 0.5000
```

Model Training

Model 2: Transfer Learning/ Xception Model V1

Description: It is a deep convolutional neural network developed by Google. Highly efficient for both computational and model parameter perspectives.

Model Architecture: shown by the diagram on the top right

Training: 10 epoch with training accuracy around 0.55-0.58 and val accuracy with 0.56-0.61(validation_split=0.2)

```
transfer_learning_model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, None, None, 3)]  0         []

 block1_conv1 (Conv2D)          (None, None, None, 32)  864         ['input_2[0][0]']

 block1_conv1_bn (BatchNorm     (None, None, None, 32)  128         ['block1_conv1[0][0]']
 alization)

 block1_conv1_act (Activati     (None, None, None, 32)  0           ['block1_conv1_bn[0][0]']
 on)

 block1_conv2 (Conv2D)          (None, None, None, 64)  18432       ['block1_conv1_act[0][0]']

 block1_conv2_bn (BatchNorm     (None, None, None, 64)  256         ['block1_conv2[0][0]']
 alization)

 block1_conv2_act (Activati     (None, None, None, 64)  0           ['block1_conv2_bn[0][0]']
 on)

 block2_sepconv1 (Separable     (None, None, None, 128) 8768        ['block1_conv2_act[0][0]']
 Conv2D)

 block2_sepconv1_bn (BatchN     (None, None, None, 128) 512         ['block2_sepconv1[0][0]']
 ormalization)

 block2_sepconv2_act (Activ     (None, None, None, 128) 0           ['block2_sepconv1_bn[0][0]']
 ation)

 block2_sepconv2 (Separable     (None, None, None, 128) 17536       ['block2_sepconv2_act[0][0]']
 Conv2D)

 block2_sepconv2_bn (BatchN     (None, None, None, 128) 512         ['block2_sepconv2[0][0]']
 ormalization)

 conv2d_7 (Conv2D)              (None, None, None, 128) 8192        ['block1_conv2_act[0][0]']

 block2_pool (MaxPooling2D)     (None, None, None, 128) 0           ['block2_sepconv2_bn[0][0]']

 batch_normalization_4 (Bat     (None, None, None, 128) 512         ['conv2d_7[0][0]']
 chNormalization)

 add_12 (Add)                   (None, None, None, 128) 0           ['block2_pool[0][0]',
                                                                      'batch_normalization_4[0][0]'
                                                                     ]

 block3_sepconv1_act (Activ     (None, None, None, 128) 0           ['add_12[0][0]']
 ation)

 block3_sepconv1 (Separable     (None, None, None, 256) 33920       ['block3_sepconv1_act[0][0]']
 Conv2D)

 block3_sepconv1_bn (BatchN     (None, None, None, 256) 1024        ['block3_sepconv1[0][0]']
 ormalization)
```

```
# Train Transfer Learning Model
history_transfer_learning_model = transfer_learning_model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=10
)
```

```
Epoch 1/10
350/350 [==============================] - 168s 472ms/step - loss: 1.1410 - accuracy: 0.5513 - val_loss: 0.9132 - val_accuracy: 0.5819
Epoch 2/10
350/350 [==============================] - 165s 472ms/step - loss: 1.0762 - accuracy: 0.5655 - val_loss: 0.9052 - val_accuracy: 0.5910
Epoch 3/10
350/350 [==============================] - 167s 477ms/step - loss: 1.2126 - accuracy: 0.5662 - val_loss: 1.0578 - val_accuracy: 0.5844
Epoch 4/10
350/350 [==============================] - 165s 471ms/step - loss: 1.0274 - accuracy: 0.5833 - val_loss: 1.5720 - val_accuracy: 0.5331
Epoch 5/10
350/350 [==============================] - 165s 472ms/step - loss: 1.1449 - accuracy: 0.5850 - val_loss: 0.7929 - val_accuracy: 0.6112
Epoch 6/10
350/350 [==============================] - 164s 470ms/step - loss: 1.0298 - accuracy: 0.5854 - val_loss: 0.8180 - val_accuracy: 0.5979
Epoch 7/10
350/350 [==============================] - 164s 470ms/step - loss: 1.0407 - accuracy: 0.5886 - val_loss: 0.8778 - val_accuracy: 0.5969
Epoch 8/10
350/350 [==============================] - 164s 469ms/step - loss: 1.0638 - accuracy: 0.5846 - val_loss: 0.9313 - val_accuracy: 0.6131
Epoch 9/10
350/350 [==============================] - 168s 481ms/step - loss: 1.1110 - accuracy: 0.5925 - val_loss: 1.4263 - val_accuracy: 0.5494
Epoch 10/10
350/350 [==============================] - 165s 471ms/step - loss: 1.0269 - accuracy: 0.5847 - val_loss: 1.0970 - val_accuracy: 0.5675
```

Model Training

Model 3: Transfer Learning/ Xception Model V2

Description: This is updated and fine tuned over the Xception Model V1

Model Architecture: shown by the diagram on the right

Training: 10 epoch with training accuracy around 0.63-0.88 and val accuracy with 0.70-0.74 (validation_split=0.2)

```python
def transfer_learning_model_v2():
    # Load the base model from Xception pre-trained on ImageNet, without the top layer
    base_model = tf.keras.applications.Xception(weights="imagenet", include_top=False)

    # Add global average pooling layer as the new top layer of the model
    avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)

    # Add a dense layer with sigmoid activation for binary classification
    output = tf.keras.layers.Dense(1, activation="sigmoid")(avg)

    # Define the model specifying inputs and outputs
    model = tf.keras.Model(inputs=base_model.input, outputs=output)

    # Freeze the layers of the base model initially
    for layer in base_model.layers:
        layer.trainable = False

    # Unfreeze certain layers for more fine-tuning
    for layer in base_model.layers[56:]:
        layer.trainable = True

    # Use a slower learning rate for fine-tuning
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
    model.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=["accuracy"])

    return model

# Create the model by calling the function
transfer_learning_model_v2 = transfer_learning_model_v2()
```

```python
# Train Transfer Learning Model
history_transfer_learning_model_v2 = transfer_learning_model_v2.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=10
)
```

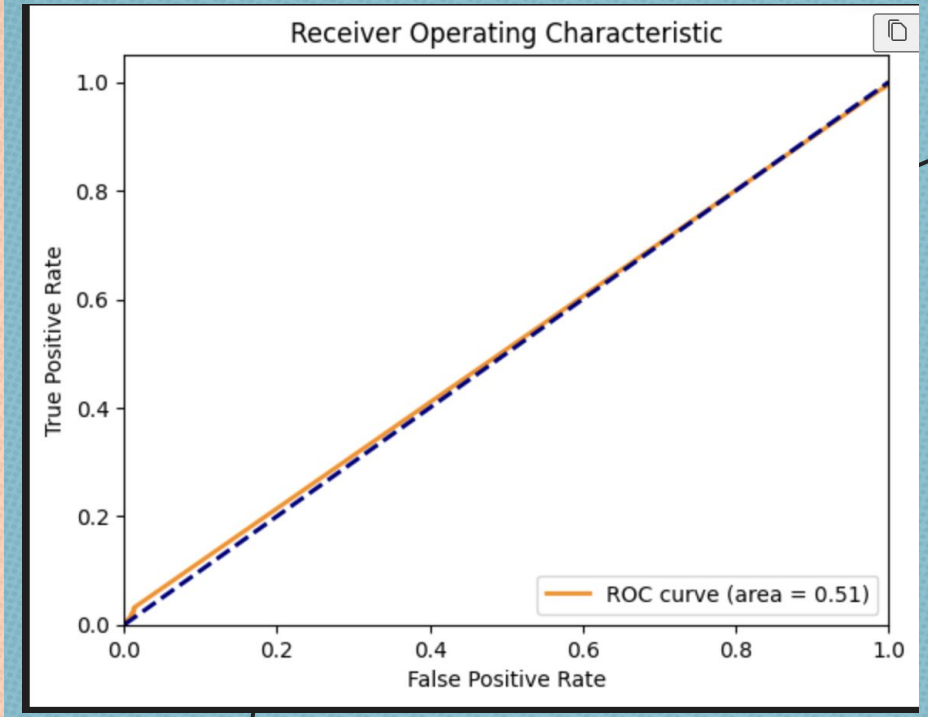
```
Epoch 1/10
350/350 [==============================] - 178s 480ms/step - loss: 0.6365 - accuracy: 0.6323 - val_loss: 0.5633 - val_accuracy: 0.7017
Epoch 2/10
350/350 [==============================] - 168s 479ms/step - loss: 0.5332 - accuracy: 0.7292 - val_loss: 0.5813 - val_accuracy: 0.7000
Epoch 3/10
350/350 [==============================] - 165s 472ms/step - loss: 0.4685 - accuracy: 0.7734 - val_loss: 0.4534 - val_accuracy: 0.7904
Epoch 4/10
350/350 [==============================] - 167s 477ms/step - loss: 0.4201 - accuracy: 0.8052 - val_loss: 0.5495 - val_accuracy: 0.7558
Epoch 5/10
350/350 [==============================] - 166s 474ms/step - loss: 0.3943 - accuracy: 0.8169 - val_loss: 0.4425 - val_accuracy: 0.7933
Epoch 6/10
350/350 [==============================] - 166s 473ms/step - loss: 0.3550 - accuracy: 0.8414 - val_loss: 0.5092 - val_accuracy: 0.7690
Epoch 7/10
350/350 [==============================] - 169s 483ms/step - loss: 0.3288 - accuracy: 0.8542 - val_loss: 0.4291 - val_accuracy: 0.8073
Epoch 8/10
350/350 [==============================] - 168s 480ms/step - loss: 0.3010 - accuracy: 0.8638 - val_loss: 0.4126 - val_accuracy: 0.8227
Epoch 9/10
350/350 [==============================] - 169s 483ms/step - loss: 0.2825 - accuracy: 0.8788 - val_loss: 0.4055 - val_accuracy: 0.8135
Epoch 10/10
350/350 [==============================] - 170s 485ms/step - loss: 0.2662 - accuracy: 0.8873 - val_loss: 0.6619 - val_accuracy: 0.7450
```

Model Evaluation

1. Custom CNN Model:

Classification Report:
Precision: [0] 0.50 [1] 0.33
Recall: [0] 1.00 [1] 0.00
F1-score: [0] 0.67 [1] 0.00
Support: [0]2000 [1]2000

Receiver Operating Characteristic (ROC) Curve:
Area under ROC curve (AUC): 0.51

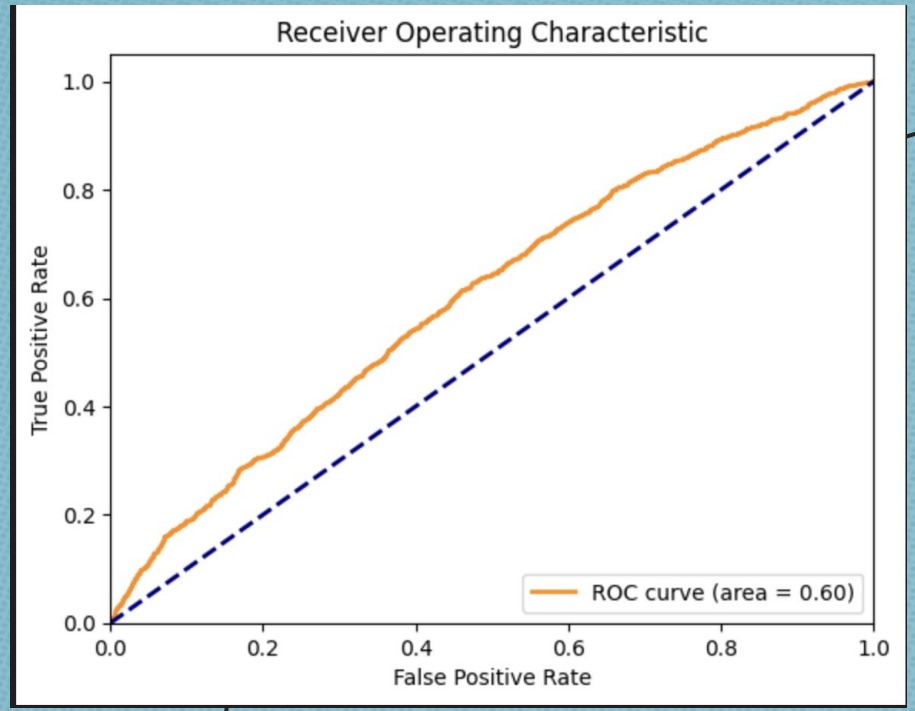


Model Evaluation

2. Transfer Learning Model Transfer Learning/ Xception Model V1:

Classification Report:
Precision: [0] 0.53 [1] 0.61
Recall: [0] 0.85 [1] 0.61
F1-score: [0] 0.65 [1] 0.35
Support: [0]2000 [1]2000

Receiver Operating Characteristic (ROC) Curve:
Area under ROC curve (AUC): 0.6

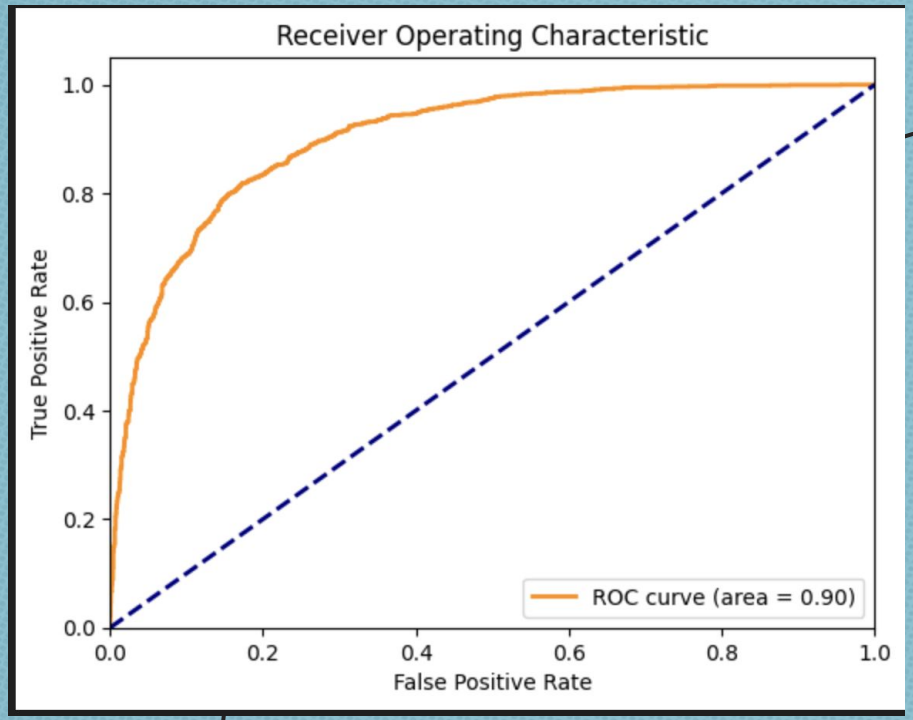


Model Evaluation

3. Transfer Learning Model Transfer Learning/ Xception Model V2:

Classification Report:
Precision: [0] 0.93 [1] 0.69
Recall: [0] 0.57 [1] 0.96
F1-score: [0] 0.70 [1] 0.80
Support: [0]2000 [1]2000

Receiver Operating Characteristic (ROC) Curve:
Area under ROC curve (AUC): 0.90



Model Deployment (Using Streamlit)

- Uploading Images
  - Supported formats: JPG, PNG, JPEG.
  - Maximum file size: 5 MB.
  - Use the 'Choose a file' button to upload images.

- Choosing a Model
  - Select the desired model from the dropdown menu.
  - Model descriptions available via hover tooltips.

- Analyzing Images
  - Click 'Analyze' to start model processing.
  - Processing time varies based on image/model complexity.

- Understanding Results
  - Results show classification: 'Real' or 'Fake'.
  - Confidence score displayed: Above 0.5 indicates 'Real'.

Model Operation - Custom CNN

- Input: 224x224x3 images
- Base Model: Pre-trained ResNet50 (frozen initially)
- Convolutional Layers:
  - Conv2D with 32 filters, kernel size (3,3), ReLU activation
  - MaxPooling2D with pool size (2,2)
  - Conv2D with 64 filters, kernel size (3,3), ReLU activation
  - MaxPooling2D with pool size (2,2)
  - Conv2D with 128 filters, kernel size (3,3), ReLU activation
  - MaxPooling2D with pool size (2,2)

Model Operation CNN Cont.

- Flatten Layer:
  - Flatten the output of the convolutional layers
- Dense Layers:
  - Dense layer with 512 units, ReLU activation
  - Dropout layer with dropout rate of 0.5
  - Dense layer with 1 unit, sigmoid activation for binary classification
- Training :
  - Optimizer: Adam
  - Loss: Binary Cross Entropy
  - Metrics: Accuracy, Precision, Recall, F1-Score
- Fine-Tuning:

Model Operation Xception Model

- Input: Images of size 224x224x3
- Base Model: Pre-trained Xception model (frozen initially)
- Output: 1 unit with sigmoid activation for binary classification
- Training Configuration:
    - Optimizer: SGD
    - Loss: Binary Cross Entropy
    - Momentum: 0.9
    - Metrics: Accuracy, Precision, Recall, F1-Score
- Fine-Tuning Strategy:
    - Unfreezing Layers: Unfreeze the last 5 layers of the base model.
    - Adjust Learning Rate: Lower the learning rate for fine-tuning.

Problem 2 Appendix

VGG16:

Validation accuracy	Fine-tuning Validation accuracy
0.264113	0.368520
0.350374	0.435340
0.375000	0.427275
0.322293	0.423531
0.387673	0.470190
0.386953	0.467166
0.369960	0.447581
0.386089	0.472206
0.349078	0.490639
0.348358	0.486175

Resnet50:

Validation accuracy	Fine-tuning Validation accuracy
0.263105	0.188364
0.183036	0.213854
0.172955	0.231567
0.178139	0.248416
0.187788	0.257200
0.191100	0.250720
0.176843	0.253600
0.205933	0.255616
0.206077	0.254176
0.182460	0.251008

VGG16 Params:

```
=================================================
Total params: 25692038 (98.01 MB)
Trainable params: 2104326 (8.03 MB)
Non-trainable params: 23587712 (89.98 MB)
```

Resnet50 Params:

```
=================================================
Total params: 15246150 (58.16 MB)
Trainable params: 2891270 (11.03 MB)
Non-trainable params: 12354880 (47.13 MB)
```

Model Operation Resnet50

- Input: 224x224x3 images
- Base Model: Pre-trained ResNet50 (frozen initially)
- Dense Layer: 1024 units, ReLU activation
- Output: 6 units, softmax activation
- Training:
  - Optimizer: Adam
  - Loss: Categorical Cross entropy
  - Metrics: Accuracy, Precision, Recall, F1-Score
- Fine-Tuning: Unfreeze last 5 layers, lower learning rate

Model Operation VGG16

- Input: 224x224x3 images
- Base Model: Pre-trained VGG16 (frozen initially)
- Dense Layer: 1024 units, ReLU activation
- Output: 6 units, softmax activation
- Training:
    - Optimizer: Adam
    - Loss: Categorical Cross entropy
    - Metrics: Accuracy, Precision, Recall, F1-Score
- Fine-Tuning: Unfreeze last 5 layers, lower learning rate

Model Operation CNN

- Convolutional Layers:
  - Layer 1: 64 filters, 3×3, kernel, 'same' padding, followed by Batch Normalization, ReLU activation, MaxPooling (2×2), and Dropout (0.25).
  - Layer 2: 128 filters, 5×5 kernel, 'same' padding, followed by Batch Normalization, ReLU activation, MaxPooling (2×2), and Dropout (0.25).
  - Layer 3: 512 filters, 3×3 kernel, 'same' padding, followed by Batch Normalization, ReLU activation, MaxPooling (2×2), and Dropout (0.25).
  - Layer 4: 512 filters, 3×3 kernel, 'same' padding, followed by Batch Normalization, ReLU activation, MaxPooling (2×2), and Dropout (0.25).

Model Operation CNN Continued

- Flatten Layer:
  - Converts the 3D feature maps from the convolutional layers into a 1D feature vector.
- Fully Connected Layers:
  - Layer 1: 256 units, followed by Batch Normalization, ReLU activation, and Dropout (0.25).
  - Layer 2: 512 units, followed by Batch Normalization, ReLU activation, and Dropout (0.25).
- Output Layer:
  - 6 units (one for each facial expression class) with Softmax activation for multi-class classification.
- Optimizer:
  - Adam optimizer with a learning rate of 0.0001.
- Loss Function:
  - Categorical Cross Entropy, suitable for multi-class classification tasks.

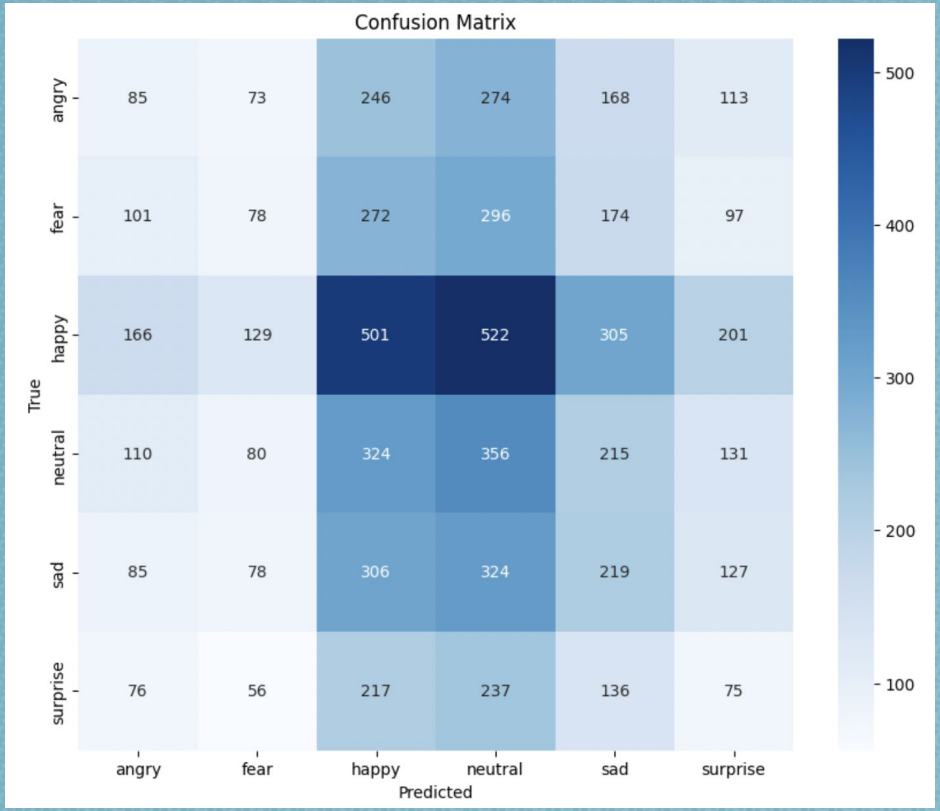
CNN Architecture Summary

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 48, 48, 64)	1,792
batch_normalization (BatchNormalization)	(None, 48, 48, 64)	256
activation (Activation)	(None, 48, 48, 64)	0
max_pooling2d (MaxPooling2D)	(None, 24, 24, 64)	0
dropout (Dropout)	(None, 24, 24, 64)	0
conv2d_1 (Conv2D)	(None, 24, 24, 128)	204,928
batch_normalization_1 (BatchNormalization)	(None, 24, 24, 128)	512
activation_1 (Activation)	(None, 24, 24, 128)	0
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 128)	0
dropout_1 (Dropout)	(None, 12, 12, 128)	0
conv2d_2 (Conv2D)	(None, 12, 12, 512)	590,336
batch_normalization_2 (BatchNormalization)	(None, 12, 12, 512)	2,048
activation_2 (Activation)	(None, 12, 12, 512)	0
max_pooling2d_2 (MaxPooling2D)	(None, 6, 6, 512)	0
dropout_2 (Dropout)	(None, 6, 6, 512)	0
conv2d_3 (Conv2D)	(None, 6, 6, 512)	2,359,808
batch_normalization_3 (BatchNormalization)	(None, 6, 6, 512)	2,048
activation_3 (Activation)	(None, 6, 6, 512)	0
max_pooling2d_3 (MaxPooling2D)	(None, 3, 3, 512)	0
dropout_3 (Dropout)	(None, 3, 3, 512)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 256)	1,179,904
batch_normalization_4 (BatchNormalization)	(None, 256)	1,024
activation_4 (Activation)	(None, 256)	0
dropout_4 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 512)	131,584
batch_normalization_5 (BatchNormalization)	(None, 512)	2,048
activation_5 (Activation)	(None, 512)	0
dropout_5 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 6)	3,078

Total params: 4,479,366 (17.09 MB)

Trainable params: 4,475,398 (17.07 MB)

Non-trainable params: 3,968 (15.50 KB)



CNN Model Evaluation:

Confusion Matrix

References and resources

1. https://www.kaggle.com/datasets/dagnelies/deepfake-faces/data
2. https://www.kaggle.com/datasets/jonathanoheix/face-expression-recognition-dataset/code
3. https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918
4. https://en.wikipedia.org/wiki/Convolutional_neural_network
5. https://arxiv.org/abs/1512.03385
6. https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568