Introduction

In the digital age, the proliferation of AI-generated content has become increasingly prevalent, blurring the lines between authentic and synthetic visuals. From lifelike images to computer-generated artworks, artificial intelligence has demonstrated remarkable capabilities in creating content that mimics reality. However, this technological advancement also raises concerns regarding the authenticity and trustworthiness of digital images.

Project Overview

This project centers around the development of an innovative solution for distinguishing between AI-generated images and real-world photographs using state-of-the-art deep learning techniques. Leveraging the ResNet101 architecture, a powerful convolutional neural network renowned for its effectiveness in image recognition tasks, we aim to build a robust image classifier capable of accurately differentiating between AI-generated and real images.

Significance and Impact

The ability to discern between AI-generated and real images holds significant implications across numerous domains, including content moderation, digital art authentication, and forensic analysis. By providing a reliable and accessible tool for image classification, this project seeks to address the growing need for authenticity verification and content moderation in the realm of digital imagery.

Key Features and Objectives

1. **Accurate Classification:** Develop a high-performance image classifier capable of accurately distinguishing between AI-generated and real images with high precision and recall.
2. **User-Friendly Interface:** Create an intuitive web application interface that allows users to easily upload images and receive real-time classification results, catering to a broad audience with varying levels of technical expertise.
3. **Scalability and Performance:** Ensure the scalability and performance of the classification system to handle large volumes of image data efficiently, accommodating diverse use cases and user demands.
4. **Interpretability:** Provide insights into the decision-making process of the classifier, enabling users to understand the factors influencing classification outcomes and fostering trust in the system's reliability.

System Requirements

Hardware Requirements

- Adequate computational resources, including a CPU or GPU capable of running deep learning tasks efficiently.
- Sufficient RAM to accommodate data processing and model training requirements.

Software Dependencies

- TensorFlow library for deep learning tasks.
- Keras library for building and training deep learning models.
- NumPy for numerical computations.
- Matplotlib and Seaborn for data visualization.
- Streamlit for creating the application interface.

Environment Setup

- Install the required Python libraries using pip or conda package manager.
- Ensure compatibility with the TensorFlow and Keras versions specified in the project.
- Set up a development environment with the necessary dependencies for model training and application deployment.

System Architecture

The system architecture for the image classification project revolves around the utilization of a deep learning model based on the ResNet101 architecture. The architecture encompasses various components and processes, each serving a specific purpose in the classification pipeline. Below is an overview of the system architecture:

Model Architecture

The core of the system is the ResNet101-based deep learning model, which serves as the image classifier. ResNet101 is a deep convolutional neural network architecture known for its effectiveness in image recognition tasks. The model comprises multiple layers of convolutional, pooling, and fully connected layers, allowing it to learn intricate patterns and features from input images.

Data Preprocessing and Augmentation

Before feeding images into the model for training, they undergo preprocessing and augmentation steps. Preprocessing involves resizing, normalization, and other transformations to ensure consistency and compatibility with the model's input requirements. Augmentation techniques such as rotation, shifting, and flipping are applied to increase the diversity and robustness of the training dataset, thereby enhancing the model's ability to generalize to unseen data.

Training Procedure

The training procedure involves feeding the preprocessed and augmented images into the ResNet101 model for training. The model learns to classify images into predefined categories based on the features extracted from the training data. During training, the model's parameters are iteratively adjusted using optimization algorithms such as Adam, with the goal of minimizing the loss function and improving classification accuracy.

Evaluation Metrics and Results

After training, the model's performance is evaluated using various metrics such as accuracy, precision, recall, and F1 score. These metrics provide insights into the model's ability to correctly classify images across different categories. Additionally, graphical representations of the training and validation metrics are generated to visualize the training progress and performance.

Application Interface

The trained model is deployed in a Streamlit web application interface, allowing users to upload images and receive real-time predictions on whether they are AI-generated or real. The interface provides a user-friendly experience, enabling seamless interaction with the classification system.

Deployment

The trained model and web application are deployed on a server infrastructure capable of handling inference requests from multiple users simultaneously. The deployment setup ensures the availability and scalability of the classification system to accommodate varying workloads and user demands.

Usage Guide

A comprehensive usage guide is provided to assist users in navigating the web application and understanding how to upload images for classification. The guide includes step-by-step instructions, example use cases, and troubleshooting tips for common issues.

Model Details

Description of the ResNet-101 Model

ResNet101 is a deep convolutional neural network architecture proposed by Kaiming He et al. It consists of 101 layers, including residual blocks that facilitate training of deeper networks by addressing the vanishing gradient problem.

Customizations and Modifications

- Added additional layers on top of the ResNet101 model, including Gaussian noise, global average pooling, dense, and dropout layers.
- Customized the learning rate schedule using polynomial decay for optimization.
- Implemented custom callbacks for early stopping based on validation loss.

Data Preprocessing and Augmentation

Data preprocessing and augmentation are essential steps in training deep learning models, particularly for image classification tasks. Augmentation techniques help increase the diversity and robustness of the training dataset, leading to improved model generalization and performance. In this project, image data augmentation is applied to the training dataset using various transformations such as rotation, shifting, shearing, zooming, and flipping. These transformations are implemented using the **ImageDataGenerator** class from the Keras library. The augmented images are then saved to an output directory for subsequent use during model training.

Data Augmentation Process

The data augmentation process involves the following steps:

1. **Initialization**: An **ImageDataGenerator** object is initialized with parameters specifying the augmentation techniques to be applied, such as rotation range, width and height shifts, shear range, zoom range, and horizontal flipping.
2. **Directory Setup**: The input directory containing the original training images and an output directory to store the augmented images are defined. If the output directory does not exist, it is created.
3. **Augmentation Loop**: The script iterates over the images in the input directory. For each image, it loads the image, applies the augmentation techniques using the **ImageDataGenerator**, and generates multiple augmented versions of the image. These augmented images are then saved to the output directory.
4. **Augmentation Parameters**: The number of augmented images generated per input image is specified, typically set to ensure a balance between increasing dataset size and avoiding overfitting.

Training and Evaluation

Dataset Overview

The dataset comprises real-world images categorized into classes for training, validation, and test sets.

Training Procedure

The model is trained using the Adam optimizer with a polynomial learning rate decay schedule. Custom callbacks monitor validation loss and implement early stopping to prevent overfitting. Below is an explanation of the training code:

Explanation of Training Code

The training code follows a structured approach to train the ResNet101-based image classifier. Here's a breakdown of the key components:

1. **Importing the required Libraries:**

```
import tensorflow as tf  # TensorFlow library for deep learning
import numpy as np  # NumPy library for numerical computations
import os  # OS module for interacting with the operating system
import keras  # Keras library for building deep learning models
from keras.applications.resnet import ResNet101  # Import ResNet50 architecture
from keras.layers import Dense, GlobalAveragePooling2D, Dropout, GaussianNoise
# Different layers for model architecture
from keras.models import Model  # Model class for defining neural network
architectures
from keras.regularizers import l1_l2  # Regularization for preventing overfitting
from keras.src.legacy.preprocessing.image import ImageDataGenerator  # Image data
preprocessing
from keras.callbacks import EarlyStopping, Callback  # Callbacks for custom
actions during training
from keras.initializers import GlorotUniform #initializes the weights using
Glorot (Xavier) initialization
```

2. **Add GPU Acceleration:**

```
# GPU configuration (if applicable)
gpus = tf.config.list_physical_devices('GPU')  # List available GPUs
if gpus:
    try:
        # Memory allocation for GPU
        for gpu in gpus:
            tf.config.set_logical_device_configuration(gpus[0],
[tf.config.LogicalDeviceConfiguration(memory_limit=5292)])
```

```
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')  # List
logical GPUs
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Error handling for GPU configuration
        print(e)
```

3. **Model Initialization**:

```
# Load the pre-trained ResNet50 model without the top classification layer
base_model = ResNet101(weights=None, include_top=False)  # Changed to ResNet101

# Add new layers on top of the model
x = base_model.output  # Output of the base model
x = GaussianNoise(0.1)(x)  # Add Gaussian noise with a standard deviation of 0.1
x = GlobalAveragePooling2D()(x)  # Global average pooling layer
x = Dense(1024, activation='relu', kernel_initializer=GlorotUniform(),
kernel_regularizer=l1_l2(l1=0.02, l2=0.04))(x)  # Dense layer with ReLU
activation and L2 regularization
x = Dropout(0.4)(x)  # Dropout layer for regularization
predictions = Dense(2, activation='softmax')(x)  # Output layer with softmax
activation

# Define the model
model = Model(inputs=base_model.input, outputs=predictions)  # Combined model

# Freeze base layers
for layer in base_model.layers:
    layer.trainable = True  # Freeze base layers for training

# Define a custom learning rate schedule
train_steps = 7125
lr_schedule = tf.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1e-4,
    decay_steps=train_steps,
    end_learning_rate=1e-5,
    power=2
)
# Compile model with Adam optimizer and binary crossentropy loss
model.compile(optimizer=keras.optimizers.Adam(learning_rate=lr_schedule),
loss='binary_crossentropy', metrics=['accuracy', keras.metrics.Precision(),
keras.metrics.Recall(), keras.metrics.AUC(), keras.metrics.F1Score(average=None,
threshold=None, name="f1_score", dtype=None)])
```

- The pre-trained ResNet101 model is loaded without the top classification layer, allowing for further customization.
- Additional layers are added on top of the base model to adapt it to the specific classification task.
- The model is compiled with appropriate optimizer, loss function, and evaluation metrics.

4. **Data Preprocessing**:

```
# Define the data directories
data_dir = 'data'  # Directory containing data
train_dir = os.path.join(data_dir, 'train')  # Training data directory
validation_dir = os.path.join(data_dir, 'validation')  # Validation data
directory
test_dir = os.path.join(data_dir, 'test')  # Test data directory

# Data augmentation for training images
train_datagen = ImageDataGenerator(rescale=1./255)

# Image data augmentation for validation and test sets
validation_datagen = ImageDataGenerator(rescale=1./255)  # Validation data
generator
test_datagen = ImageDataGenerator(rescale=1./255)  # Test data generator

# Data generators for training, validation, and test sets
train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(224, 224),
        batch_size=16,
        class_mode='categorical')  # Training data generator

validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(224, 224),
        batch_size=16,
        class_mode='categorical')  # Validation data generator

test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(224, 224),
        batch_size=16,
        class_mode='categorical')  # Test data generator
```

- Image data generators are set up to preprocess and augment the training, validation, and test images in real-time.
- Data augmentation techniques such as rescaling are applied to enhance the diversity and robustness of the training data.

5. **Custom Callbacks**:

```
# Custom Callback to stop training if validation loss exceeds training loss
class StopTrainingOnValidationLoss(Callback):
    def __init__(self, filepath):
        super(StopTrainingOnValidationLoss, self).__init__()
        self.filepath = filepath
        self.best_weights = None
        self.best_val_loss = float('inf')

    def on_epoch_end(self, epoch, logs=None):
        val_loss = logs.get('val_loss')
        train_loss = logs.get('loss')
        if val_loss is not None and train_loss is not None:
            if val_loss > train_loss:
                print("\nValidation loss is higher than training loss. Stopping 
training.")
                self.model.stop_training = True
            else:
                if val_loss < self.best_val_loss:
                    print("\nValidation loss is lower than previous best. Saving 
weights.")
                    self.best_val_loss = val_loss
                    self.best_weights = self.model.get_weights()
                    self.model.save(self.filepath)

    def on_train_end(self, logs=None):
        if self.best_weights is not None:
            print("\nLoading weights from the epoch with lowest validation loss.")
            self.model.set_weights(self.best_weights)
```

- Custom callbacks are defined to monitor the training process and perform specific actions based on certain conditions.
- In this case, the **StopTrainingOnValidationLoss** callback stops training if the validation loss exceeds the training loss and saves the best model weights.

6. **Model Training**:

```
# Early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    min_delta=0.001,    # Minimum change to qualify as an improvement
    patience=4,         # Number of epochs with no improvement after which 
training will be stopped
    verbose=1,          # Verbosity mode
    mode='min',         # 'min' mode because lower validation loss is better
    restore_best_weights=True  # Restore model weights from the epoch with the 
best value of the monitored quantity
)

# Define the custom callback
save_path = 'best_model_weights.h5'
stop_on_val_loss = StopTrainingOnValidationLoss(filepath=save_path)

# Train the model
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=60,
    callbacks=[early_stopping, stop_on_val_loss],  # List of callbacks
)
```

- The model is trained using the **fit** method, which iterates over the training data in batches and updates the model parameters accordingly.
- Training progresses over a specified number of epochs, with early stopping applied to prevent overfitting.
- The training progress is monitored, and metrics such as loss, accuracy, precision, recall, AUC, and F1 score are computed and logged.

7. **Evaluation**:

```python
# Evaluate the model on the test set
eval_results = model.evaluate(test_generator)
test_loss, test_accuracy, test_precision, test_recall, test_auc, test_f1_score =
eval_results

# Print test metrics
print(f"Test loss: {test_loss:.4f}")
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
print(f"Test precision: {test_precision:.4f}")
print(f"Test recall: {test_recall:.4f}")
print(f"Test AUC: {test_auc:.4f}")
print(f"Test F1 Score: {test_f1_score[0]:.4f}")  # Extract scalar value for F1
score
```

- After training, the model is evaluated on the test set to assess its generalization performance.
- Evaluation metrics are computed, including loss, accuracy, precision, recall, AUC, and F1 score.
- These metrics provide insights into the model's performance in classifying unseen data.

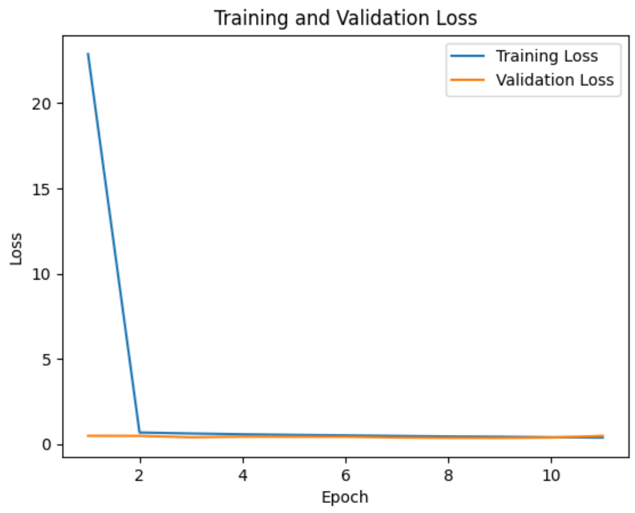
Evaluation Metrics and Results

Once the model is trained, various evaluation metrics are computed on the test set to assess its performance. These metrics include accuracy, precision, recall, AUC, and F1 score. Additionally, graphical representations of these metrics are provided for visualization.

Graphical Representation of Evaluation Metrics

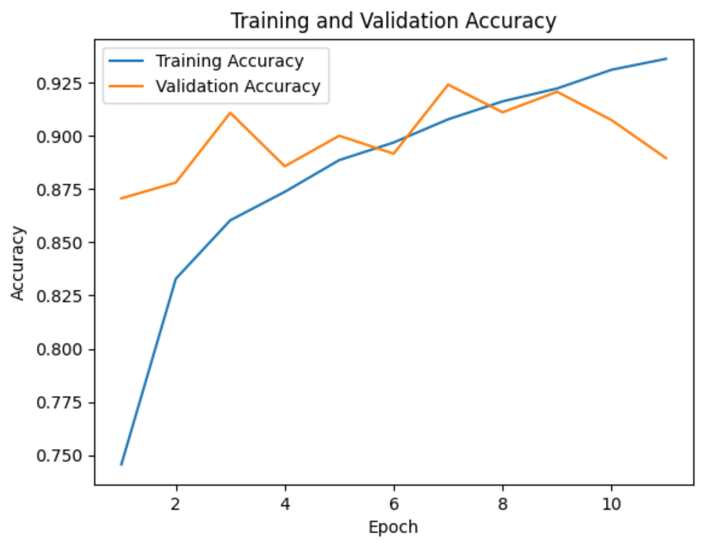
The following graphs illustrate the training and validation performance of the model across different epochs:

Training and Validation Loss



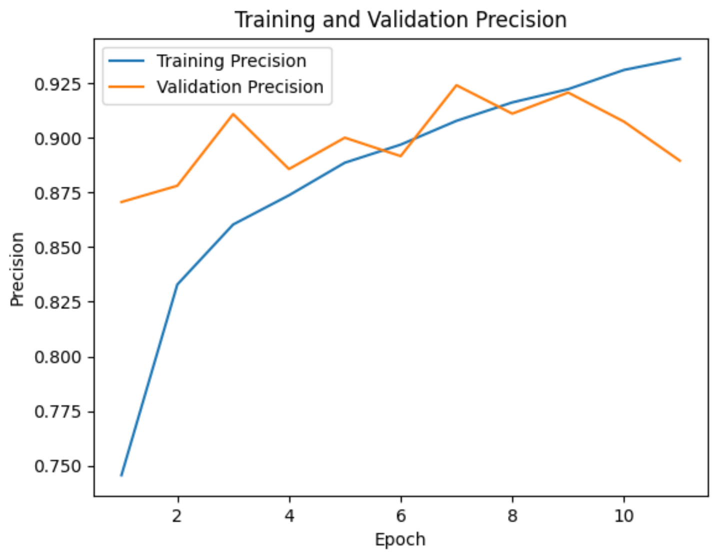
The graph above shows the training and validation loss over epochs. Lower values indicate better performance, with the goal of minimizing loss during training while maintaining low validation loss to prevent overfitting.

Training and Validation Accuracy



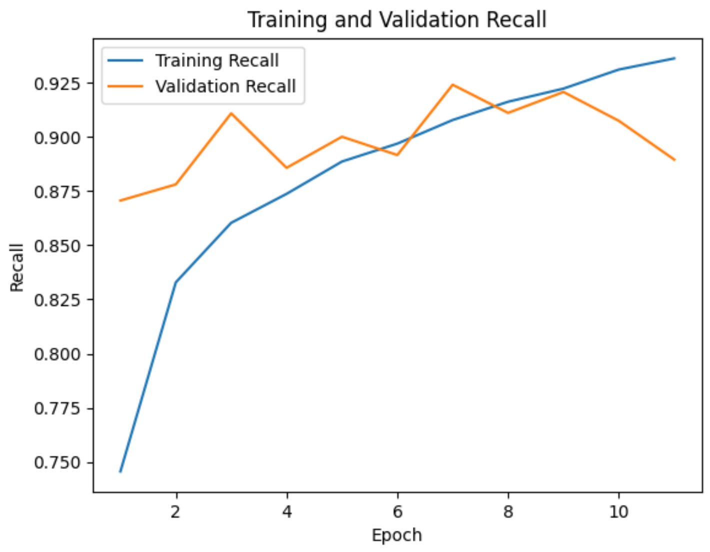
This graph depicts the training and validation accuracy over epochs. Higher accuracy values indicate better model performance in correctly classifying images in both training and validation datasets.

Training and Validation Precision



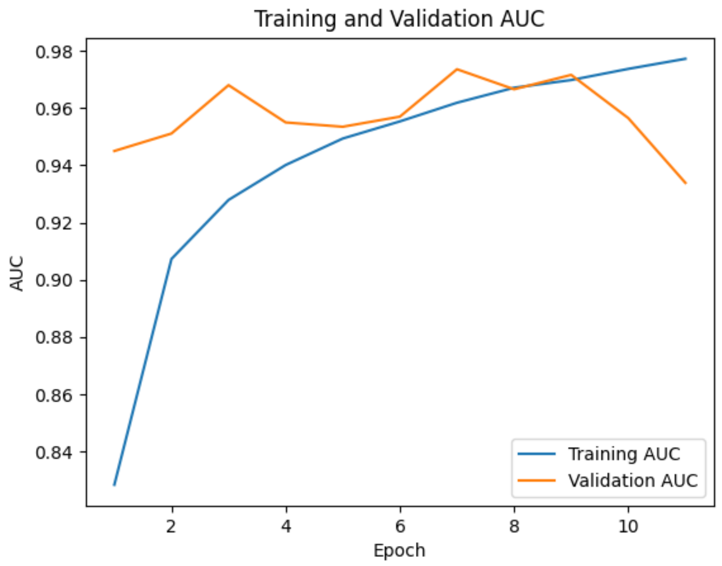
Precision measures the proportion of true positive predictions among all positive predictions. Higher precision values indicate fewer false positives in the predictions made by the model.

Training and Validation Recall



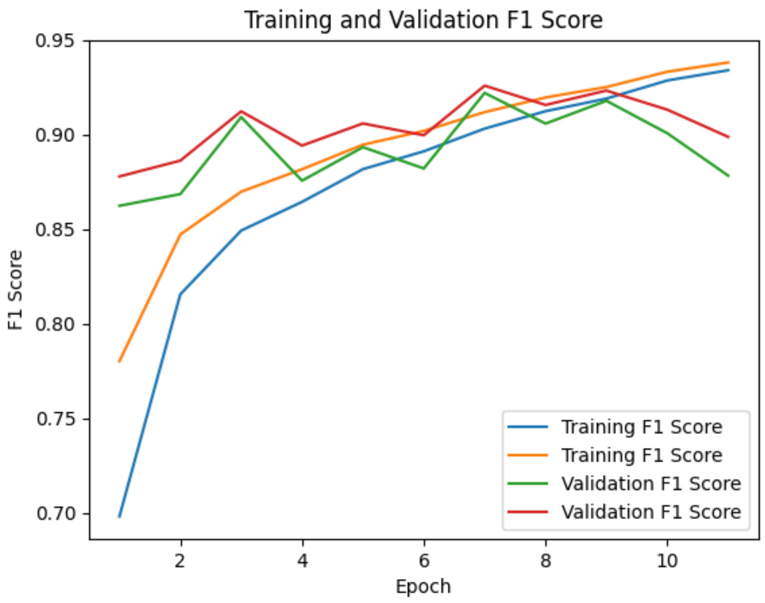
Recall, also known as sensitivity or true positive rate, measures the proportion of true positives correctly identified by the model out of all actual positives. Higher recall values indicate better coverage of positive instances by the model.

Training and Validation AUC (Area Under the ROC Curve)



The AUC metric represents the area under the receiver operating characteristic (ROC) curve. Higher AUC values indicate better discrimination performance of the model across different threshold settings.

Training and Validation F1 Score



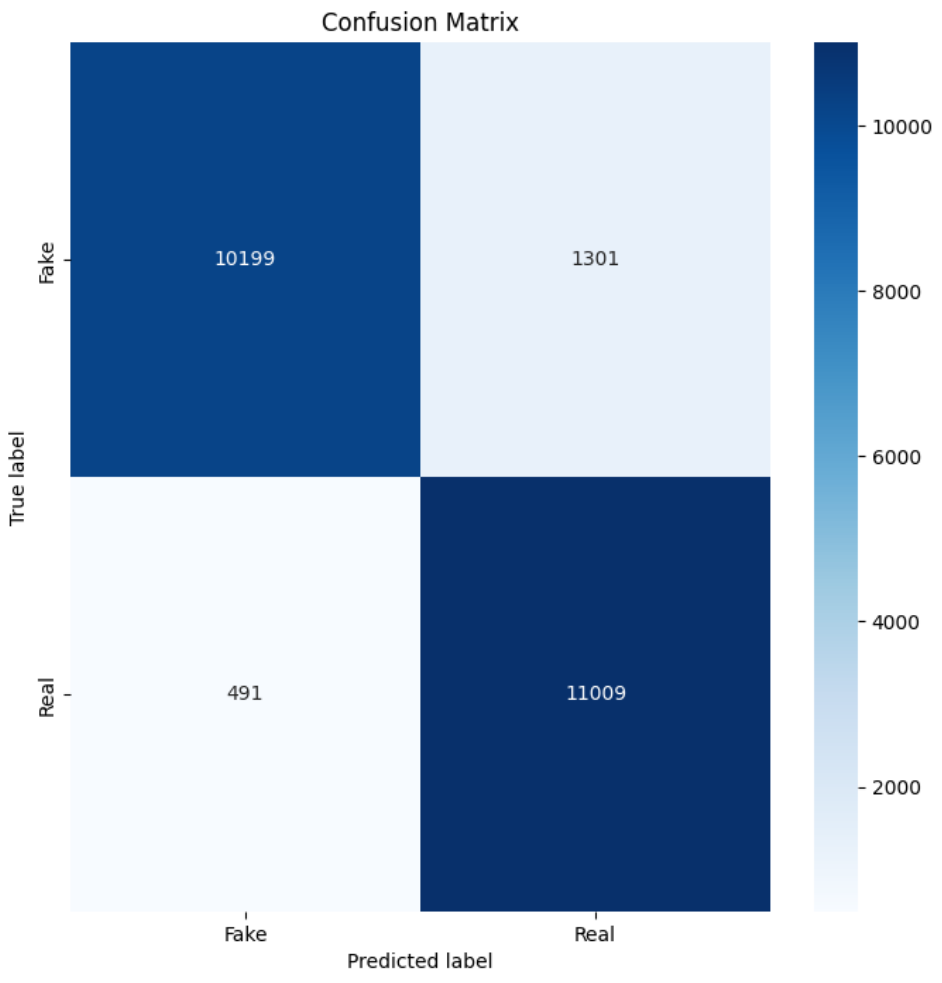
The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both precision and recall. Higher F1 score values indicate better overall performance of the model in terms of both precision and recall.

Model Performance Summary

After evaluating the model on the test set, the following summary of performance metrics is obtained:

- **Test Loss**: `0.3700`
- **Test Accuracy**: `92.21%`
- **Test Precision**: `0.9221`
- **Test Recall**: `0.9221`
- **Test AUC**: `0.9713`
- **Test F1 Score**: `0.9192`

Model Confusion Matrix



Based on the Confusion Matrix:

True Positive (TP): 11009 **True Negative (TN):** 10199

False Positive (FP): 1301 **False Negative (FN):** 491

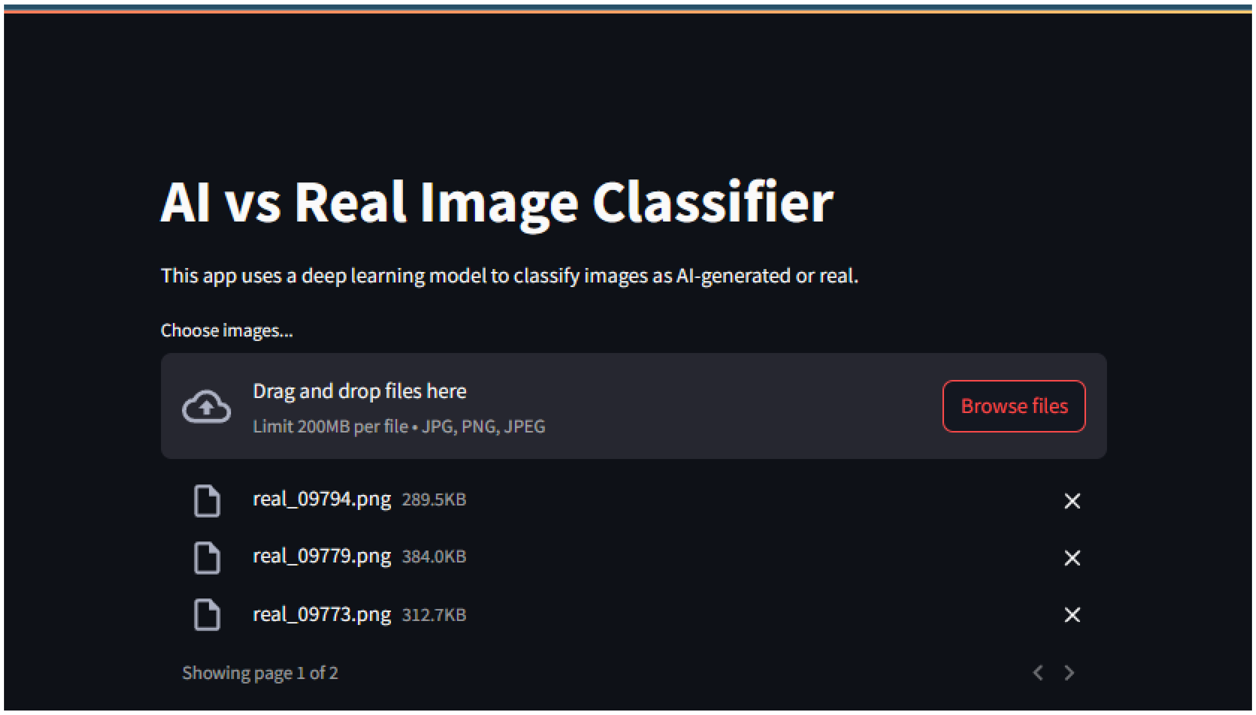
Calculated Accuracy: (TP + TN)/(TP + TN + FP + FN) = 0.9225 = 92.25%

Calculated Precision: (TP)/(TP + FP) = 0.8943 = 89.43%

Calculated Recall: (TP)/(TP + FN) = 0.9573 = 95.73%

Calculated F1 Score: (2 * (Precision * Recall)/(Precision + Recall) = 0.9245 = 92.45%

Application Interface



AI vs Real Image Classifier

This app uses a deep learning model to classify images as AI-generated or real.

Choose images...

Drag and drop files here

Limit 200MB per file • JPG, PNG, JPEG

real_09794.png 289.5KB

real_09779.png 384.0KB

real_09773.png 312.7KB

Showing page 1 of 2

Streamlit Interface Design

The web application interface is designed using Streamlit, a popular Python library for building interactive web applications. The interface allows users to upload images and receive classification results from the trained model.

User Interaction Flow

1. **Upload Images**: Users can upload images from their local device by clicking the "Choose images..." button. Supported image formats include JPG, PNG, and JPEG.
2. **Image Classification**: Upon uploading an image, the application preprocesses it and passes it through the trained ResNet101-based model for classification.
3. **Display Results**: The application displays the uploaded image along with the classification result, indicating whether the image is classified as AI-generated or real. Additionally, the confidence score corresponding to the classification result is provided.

Interface Components

- **File Uploader**: Allows users to upload images for classification. Supports multiple file uploads and restricts accepted file types to JPG, PNG, and JPEG formats.
- **Image Display**: Displays the uploaded image for user reference.

- **Classification Result**: Presents the classification result (AI-generated or real) along with a confidence score indicating the model's confidence in its prediction.

Deployment

- **Deployment Strategy**

  The web application is deployed using a Streamlit server, which hosts the application and serves it to users via a web browser. The trained ResNet101 model is loaded within the application for real-time image classification.

- **Server Configuration**
  The Streamlit server is configured to handle incoming requests from users, process uploaded images, and execute inference using the loaded model. Memory allocation may be optimized based on server resources and application requirements.
- **Monitoring and Scaling**
  Monitoring tools can be employed to track server performance, user interactions, and application metrics. Scalability can be achieved by deploying the application on cloud infrastructure capable of handling increased traffic and resource demands.

Usage Guide

Step-by-Step User Guide

1. **Upload Images**: Click the "Choose images..." button to select images for classification. Multiple images can be uploaded simultaneously.

2. **View Classification Results**: After uploading an image, the application displays the uploaded image and provides the classification result (AI-generated or real) along with a confidence score.

Example Use Cases

- **Image Verification**: Users can verify the authenticity of images by classifying them as AI-generated or real.

- **Content Moderation**: Content platforms can use the application to detect and filter out AI-generated content from real images.

Troubleshooting Common Issues

- **Unsupported File Formats**: Ensure that uploaded images are in the supported formats (JPG, PNG, JPEG).

- **Model Loading**: If the application fails to load the model, check the file path and ensure that the model file (**ai_real_image_classifier_resnet101.keras**) is accessible.

Development

Source Code Structure

The source code for the project is structured as follows:

- **model_training.py**: Contains the code for training the ResNet101-based image classifier model.
- **web_app.py**: Includes the code for the Streamlit web application interface for image classification.
- **data/**: Directory containing the dataset for model training and evaluation.
- **metrics_graphs/**: Directory for storing the graphical representations of training and validation metrics.
- **best_model_weights.h5**: File containing the weights of the best-performing model during training.

Build Instructions

To build and run the project locally, follow these steps:

1. Install the required Python dependencies using pip:

   ```
   pip install tensorflow keras streamlit numpy matplotlib seaborn
   ```

2. Ensure compatibility with the specified versions of TensorFlow and Keras libraries.
3. Navigate to the project directory containing the source code.
4. Run the model training script to train the image classifier:

   ```
   Python3 ./model_training.py
   ```

5. Once the model is trained, launch the Streamlit web application:

   ```
   streamlit run web_app.py
   ```

6. Access the web application interface in your web browser and follow the provided instructions for image classification.

Testing and Quality Assurance

Testing methodologies include unit testing, integration testing, and validation testing to ensure the correctness and robustness of the implemented functionalities. Quality assurance measures involve thorough validation of model predictions, performance evaluation, and user feedback integration.

Third-Party Resources and Libraries

The project makes use of several third-party resources and libraries, including:

- TensorFlow and Keras: Deep learning frameworks for building and training neural network models.
- Streamlit: Python library for creating interactive web applications.
- NumPy: Library for numerical computations and array manipulations.
- Matplotlib and Seaborn: Tools for data visualization and plotting.
- ImageDataGenerator (from Keras): Utility for real-time data augmentation and preprocessing during model training.

Appendix

- **Glossary of Terms**
    - ResNet101: A deep convolutional neural network architecture consisting of 101 layers, known for its effectiveness in image classification tasks.
    - Streamlit: A Python library used for creating interactive web applications with minimal code.
    - ImageDataGenerator: A Keras utility for generating batches of augmented image data during model training.
    - Polynomial Decay: A learning rate scheduling technique where the learning rate decreases over time following a polynomial function.

- **References:**
    - **Data Citation:**
        - https://www.kaggle.com/datasets/birdy654/cifake-real-and-ai-generated-synthetic-images
        - https://www.kaggle.com/datasets/kidonpark1023/fake-or-real-dataset/data
    - **Documentation References:**
        - TensorFlow Documentation. (n.d.). "Getting Started with TensorFlow." [Online]. Available: https://www.tensorflow.org/guide/getting_started
        - Keras Documentation. (n.d.). "Keras API Reference." [Online]. Available: https://keras.io/api/

    - Streamlit Documentation. (n.d.). "Streamlit Documentation." [Online]. Available: https://docs.streamlit.io/en/stable/

- **Code References:**
    - Chollet, F., et al. (2015). "Keras: Deep Learning library for Theano and TensorFlow." [Online]. Available: https://keras.io/
    - Abadi, M., et al. (2015). "TensorFlow: Large-scale machine learning on heterogeneous systems." [Online]. Available: https://www.tensorflow.org/