**Advanced AI Real Image Classifier using ResNet101-based Deep Learning**

**A Project Report**

Submitted in partial fulfilment of the

Requirements for the award of the degree of

**MASTER OF SCIENCE (INFORMATION TECHNOLOGY)**

**By**

**Joehan Misquitta**

2324-MSC12-047

**Under the esteemed guidance of**

**Ahtesham Shaikh**

**Professor**

A blue and white emblem

Description automatically generated

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**WILSON COLLEGE**

**(Affiliated to University of Mumbai)**

**MUMBAI, 400007**

**MAHARASHTRA 2023-2024**

**WILSON COLLEGE**

**(Affiliated to University of Mumbai)**

**MUMBAI-MAHARASHTRA-400007**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

A blue and white emblem

Description automatically generated

This is to certify that the project entitled, “Advanced AI Real Image Classifier using ResNet101-based Deep Learning”, is bonafied work of **Joehan Misquitta** bearing Roll.No**: 2324-MSC12-047** submitted in partial fulfillment of the requirements for the award of degree of MASTER OF SCIENCE in INFORMATION TECHNOLOGY from University of Mumbai.

**Subject Incharge: IT Incharge:**

**External Examiner**

**Date: College Seal**

**Index**

|  |
| --- |
| 1. **Introduction** |
| * + Project Overview |
| * + Significance and Impact |
| * + Key Features and Objectives |
| 1. **System Requirements** |
| * + Hardware Requirements |
| * + Software Dependencies |
| * + Environment Setup |
| 1. **System Architecture** |
| * + Overview |
| * + Model Architecture |
| * + Data Preprocessing and Augmentation |
| * + Training Procedure |
| 1. **Model Details** |
| 1. **Training and Evaluation** |
| 1. **Data Preprocessing and Augmentation** |
| * + Data Augmentation Techniques |
| * + Implementation Details |
| 1. **Model Training and Evaluation** |
| * + Dataset Overview |
| * + Training Procedure |
| * + Evaluation Metrics and Results |
| 1. **Application Interface** |
| * + Streamlit Interface Design |
| * + User Interaction Flow |
| * + Interface Components |
| 1. **Deployment** |
| * + Deployment Strategy |
| * + Server Configuration |
| * + Monitoring and Scaling |
| 1. **Usage Guide** |
| * + Step-by-Step User Guide |
| * + Example Use Cases |
| * + Troubleshooting Common Issues |
| 1. **Development** |
| * + Source Code Structure |
| * + Build Instructions |
| * + Testing and Quality Assurance |
| 1. **Conclusions** |
| * Significance of the System |
| * Limitations of the System |
| * Future Scope of the System |
| 1. **Third-Party Resources and Libraries** |

**Abstract**

In today's digital age, the proliferation of AI-generated content poses significant challenges for content moderation and verification. To address this issue, this project introduces an advanced AI real image classifier based on the ResNet101 deep learning architecture. The classifier aims to distinguish between AI-generated and real images, thereby enhancing trust and authenticity in digital content.

The project encompasses the development of a deep learning model using TensorFlow and Keras, along with the implementation of a web application interface using Streamlit. The model is trained on a dataset of labeled images, employing techniques such as data augmentation, model customization, and performance optimization to achieve accurate classification results.

Key features of the system include single image processing, support for .jpg and .png file formats, and the provision of future scope for batch processing and download options. Through rigorous testing and quality assurance practices, including unit testing, integration testing, and performance evaluation, the classifier demonstrates robustness and reliability in real-world scenarios.

In conclusion, the AI real image classifier holds significant implications for content moderation and verification, offering a reliable solution to combat the proliferation of AI-generated content and promote trustworthiness in online environments. Through continuous development and refinement, the classifier aims to address emerging challenges and adapt to evolving needs in the digital landscape.

**ACKNOWLEDGEMENT**

I would like to express my deepest gratitude to Mr. Ahtesham Shaikh for his invaluable guidance and mentorship throughout the duration of this project. His expertise, encouragement, and dedication have been instrumental in shaping the development and success of the AI real image classifier. I am deeply grateful for his unwavering support, patience, and insightful feedback, which have inspired me to overcome challenges and strive for excellence in every aspect of this project.

Additionally, I extend my sincere appreciation to Prof. Mrs. Srilatha Ratnam for providing me with the opportunity to undertake this project. Her trust, encouragement, and support have been invaluable, and her belief in my abilities has empowered me to explore new horizons and strive for excellence. I am grateful for Prof. Mrs. Srilatha Ratnam's mentorship, guidance, and dedication to fostering academic growth and innovation, which have played a pivotal role in shaping the direction and outcomes of this project.

Both Mr. Ahtesham Shaikh and Prof. Mrs. Srilatha Ratnam have been instrumental in my academic and professional development, and I am truly indebted to them for their contributions to this project.

**DECLARATION**

I hereby declare that the project entitled, "AI Real Image Classifier using ResNet101-based Deep Learning," conducted at Wilson College Mumbai, has not been duplicated or submitted to any other institution for the purpose of obtaining any academic degree. To the best of my knowledge, no other individual besides myself has submitted a similar project to any other institution.

**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. Optimally you need a gpu with atleast 6GB of VRAM.
* 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

1. **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)

1. **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

# UnFreeze base layers

for layer in base\_model.layers:

    layer.trainable = True  # UnFreeze 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.

1. **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.

1. **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.

1. **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.

1. **Evaluation**:

# 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**

A graph of training and validation loss

Description automatically generated

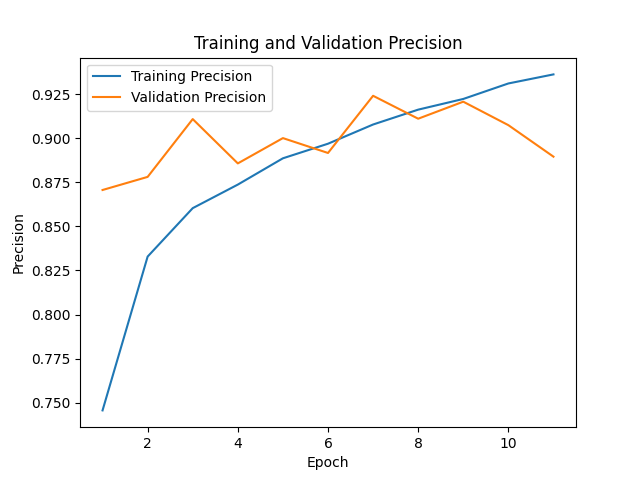
The graph above shows the training and validation loss over epochs. Lower values indicate better performance, with the goal of minimizing loss during training while maintaining low validation loss to prevent overfitting.

**Training and Validation Accuracy**

**A graph with blue and orange lines

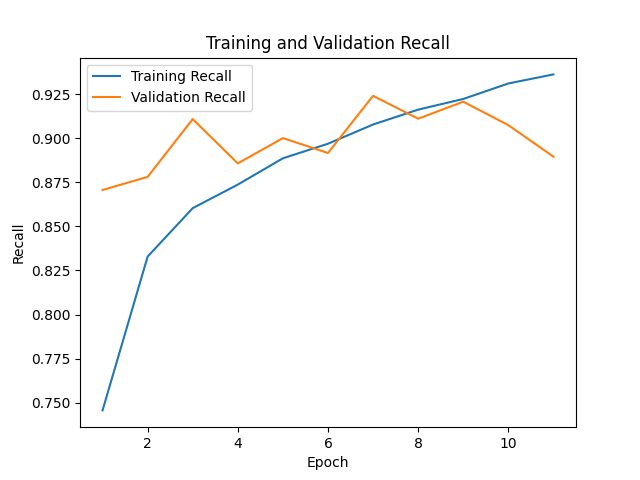
Description automatically generated**

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)**

**A graph of a line graph

Description automatically generated with medium confidence**

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**

**A graph of training and validation

Description automatically generated**

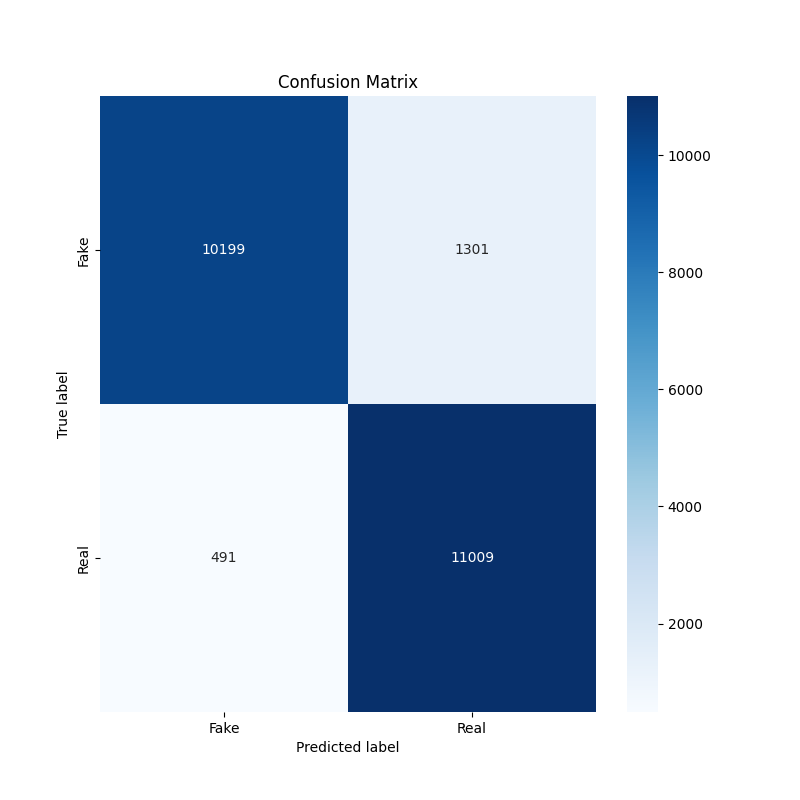
The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both precision and recall. Higher F1 score values indicate better overall performance of the model in terms of both precision and recall.

**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**

**A screenshot of a computer

Description automatically generated**

**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.

**Explanation of Web Application Code**

**Main Components**

1. **Importing Libraries:** Various libraries such as TensorFlow, Streamlit, and Keras are imported for deep learning, web application development, and image processing.

import tensorflow as tf

import streamlit as st

from keras.models import load\_model

from keras.preprocessing import image

from keras.preprocessing.image import img\_to\_array

from keras.applications.resnet50 import preprocess\_input

import numpy as np

1. **GPU Configuration:** The code configures GPU memory allocation if GPUs are available.

# 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)

1. **Loading Trained Model:** The pre-trained ResNet101 model is loaded using the load\_model function from Keras.

# Load your trained model

model = load\_model('ai\_real\_image\_classifier\_resnet101.keras')

1. **Preprocessing Images:** The preprocess\_image function preprocesses the uploaded image before making predictions.

# Define a function to preprocess the images to the correct format

def preprocess\_image(*uploaded\_file*):

    img = image.load\_img(*uploaded\_file*, *target\_size*=(224, 224))

    img\_array = img\_to\_array(img)

    img\_array = np.expand\_dims(img\_array, *axis*=0)

    img\_array = img\_array / 255.0  # Rescale the image

    return img\_array

1. **Streamlit Interface:** Streamlit is used to create the user interface for uploading images and displaying classification results.

# Set up the Streamlit interface

st.title('AI vs Real Image Classifier')

st.write("This app uses a deep learning model to classify images as AI-generated or real.")

1. **File Upload Interface:** Streamlit's file uploader component allows users to upload images for classification.

# Upload file interface for multiple files

uploaded\_files = st.file\_uploader("Choose images...", *type*=["jpg", "png", "jpeg"], *accept\_multiple\_files*=True)

if uploaded\_files is not None:

    for uploaded\_file in uploaded\_files:

        # Display the uploaded image

        st.image(uploaded\_file, *caption*='Uploaded Image', *use\_column\_width*=True)

        st.write("Classifying...")

        # Preprocess the uploaded image

        preprocessed\_image = preprocess\_image(uploaded\_file)

1. **Making Predictions:** The uploaded images are preprocessed, and predictions are made using the loaded model.

        # Make a prediction

        predictions = model.predict(preprocessed\_image)

        confidence\_score = np.max(predictions)  # Get the highest probability value as the confidence score

1. **Displaying Results:** The classification results, including the predicted class and confidence score, are displayed to the user.

        # Display the results

        class\_names = ['AI-generated', 'Real']

        string\_result = class\_names[np.argmax(predictions)]

        st.success(f'The image is classified as: {string\_result}')

        st.write(f'Confidence Score: {confidence\_score:.5f}')  # Display the confidence score rounded to five decimal places

**Implementation Details**

* **GPU Configuration**:
  + The code checks for the availability of GPUs using TensorFlow's **list\_physical\_devices** function.
  + If GPUs are available, it configures GPU memory allocation to limit memory usage during model training and inference.
* **Error Handling**:
  + Error handling mechanisms are implemented to ensure robustness and smooth execution of the web application.
  + Potential errors, such as file upload failures or model loading errors, are anticipated and handled gracefully to provide a seamless user experience.
* **Debugging Output**:
  + Debugging statements, such as print statements, are included throughout the code to aid in diagnosing issues during development and testing.
  + These statements provide valuable insights into the execution flow and help identify any potential errors or unexpected behavior.
* **Deployment Considerations**:
  + When deploying the web application in a production environment, considerations such as server resources, scalability, and security need to be addressed.
  + Streamlit's built-in deployment capabilities or other hosting platforms can be utilized to deploy the web application and make it accessible to users over the internet.
  + Security measures, such as authentication and encryption, may be implemented to protect user data and ensure the integrity of the application.

**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

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

Python3 ./model\_training.py

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

streamlit run web\_app.py

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

**Testing and Quality Assurance**

Ensuring the robustness and reliability of the AI real image classifier is paramount to its effectiveness in real-world applications. To achieve this, rigorous testing and quality assurance practices are implemented throughout the development lifecycle. The testing process encompasses various stages, including unit testing, integration testing, and performance evaluation, aimed at validating the functionality, accuracy, and efficiency of the classifier.

**Unit Testing**

Unit testing involves testing individual components or modules of the classifier in isolation to verify their correctness and functionality. Each module, such as data preprocessing functions, model architecture, and web application interface, is subjected to unit tests to ensure it behaves as expected under different conditions. Tools like PyTest and TensorFlow's built-in testing utilities are employed to automate the testing process and validate the behavior of each component.

**Integration Testing**

Integration testing focuses on testing the interactions and integration of different components within the classifier. This includes testing the end-to-end workflow of data preprocessing, model training, and inference, as well as the interaction between the web application interface and the underlying model. Integration tests validate the system's overall functionality and identify any potential issues or inconsistencies that may arise from component interactions.

**Performance Evaluation**

Performance evaluation is conducted to assess the classifier's performance in terms of accuracy, speed, and resource utilization. This involves benchmarking the classifier against various metrics, including classification accuracy, inference time, and memory consumption, under different scenarios and workloads. Performance profiling tools like TensorFlow Profiler and system monitoring utilities are utilized to analyze and optimize the classifier's performance, ensuring it meets the desired performance targets and scalability requirements.

**Cross-Validation**

Cross-validation techniques, such as k-fold cross-validation, are employed to assess the classifier's generalization performance and mitigate overfitting. By partitioning the dataset into multiple subsets and training the classifier on different combinations of training and validation data, cross-validation helps validate the classifier's robustness and reliability across diverse datasets and conditions.

**Quality Assurance**

Quality assurance measures, including code reviews, documentation audits, and continuous integration (CI) pipelines, are integrated into the development process to maintain code quality and consistency. Code reviews facilitate collaboration and ensure adherence to coding standards and best practices, while documentation audits ensure that project documentation remains up-to-date and comprehensive. CI pipelines automate the build, test, and deployment process, enabling early detection of issues and ensuring the stability and reliability of the classifier throughout its lifecycle.

**Conclusions**

**Significance of the System**

The AI real image classifier developed in this project holds significant implications for content moderation and verification in the digital realm. By accurately distinguishing between AI-generated and real images, the classifier contributes to enhancing trust and authenticity in online content. For instance, when integrated into social media platforms or digital content repositories, it empowers users to discern the origin and authenticity of images, thus mitigating the spread of misinformation and fake content.

Furthermore, the classifier's capability to differentiate between AI-generated and real images facilitates targeted content filtering and moderation, enabling platforms to enforce community guidelines effectively. This enhances user experience by promoting a safer and more trustworthy online environment.

**Limitations of the System**

Despite its advancements, the current implementation of the AI real image classifier has certain limitations:

1. Single Image Processing: The system is currently limited to processing one image at a time, restricting its scalability and throughput for batch processing scenarios.
2. Supported File Formats: The system can only process images in .jpg and .png file formats, limiting its compatibility with other image formats commonly used on the web.

**Future Scope of the System**

To address the limitations and enhance the functionality of the system, the following future scope is identified:

1. Batch Processing: Implement batch processing capabilities to enable the classifier to process multiple images simultaneously. This would improve efficiency and scalability, particularly in scenarios where large volumes of images need to be classified.
2. Download Options: Introduce options or capabilities to download processed images in PDF format or as a compressed ZIP file. This feature would enhance user convenience by facilitating the export and sharing of classified images for further analysis or archival purposes.

**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/>