Report on AI Image Detector

LAM, Tsz Kit
SID: 1155194085
The Chinese University of Hong Kong
1155194085@link.cuhk.edu.hk

CHOI Ho Yan
SID: 1155194468
The Chinese University of Hong Kong
1155194468@link.cuhk.edu.hk

Abstract

This paper targets the problem of high similarity between AI-generated images and human-crafted images, proposes an AI Image Detector using three image classification models, namely custom Convolutional Neural Network, ResNet and Vision Transformer, and analyzes their performance and effectiveness.

1 Introduction

Recently, AI-generated images adapted to the style of Ghibli, a Japanese animation studio well-known for its unique anime style behind My Neighbor Totoro, Spirited Away and other classical films, have sparked heated discussions about the role of AI in artwork. The latest GPT-4o model from OpenAI has incorporated this artistic style into its image generator, resulting in an 11% increase in global ChatGPT app downloads and a 5% rise in weekly active users [1].

The underlying problem behind this discussion is the decrease in disparity between human-crafted images and AI-generated images, leading to many potential threats of misleading and misuses of information. People are more sensitive to images when receiving information, and there is an average of 2 million images generated daily [2], highlighting the severe consequences of AI images. For example, forgery AI photos and information in celebrity reputations or political campaigns could frame public opinion, with more than 80% of Americans expressing concerns about AI misuse in the U.S. presidential election [3].

To mitigate the potential concerns about AI images, this project presents a binary classification model that classifies between AI-generated and human-designed images. We hope that this AI image detector can effectively recognize AI photos so that people can be more cautious about the image sources when receiving new information.

2 Problem Statements

Given a dataset divided into "AI-generated image" and "human-generated image" two classes, for each image in the dataset, the goal of our classifier is to predict which class it belongs to. We implement the classifier by designing a custom CNN architecture and fine-tuning an existing ViT architecture. In each case, the model is trained with labelled images and tested with unseen images. Each model performance is improved by means of the accuracy and loss for each epoch, and analyzed by its confusion matrix. We further discuss the comparison and improvements of these two models.

3 Data

3.1 Data Source

We would use a dataset named "AI vs. Human-Generated Images" published in Kaggle [4]. The image data are sampled from the Shutterstock platform across different categories and labelled either "human-created images" or "AI-generated images", in which the images are pairings of an authentic image and its equivalent generated image using AI generative models. We perform the model training and testing based on the provided dataset, assuming this data source is reliable and general.

Figure 1: An example of an authentic image paired with its AI-generated equivalent [4]

3.2 Data Extraction

The original dataset contains approximately 80,000 training data and 5,500 testing data, with CSV files indicating the class labels for each image filename. However, we found that the labeling for the test set is not clear and the folder structures are not well organized.

Therefore, we randomly extracted 25,000 training data and 5,000 testing data from the training dataset for each "AI-generated image" class and "human-generated image" class.

3.3 Data Preprocessing

After extracting the necessary dataset, the images are resized to 224x224 pixels to reduce training computations and unify the input size [5]. We apply ImageNet normalization to the pixel values. The ImageNet normalization involves subtracting the mean and dividing by the standard deviation of the ImageNet dataset RGB channels (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]), which is a common practice and provide a better model training convergence [5].

We also perform data argumentation on training data. The random transformations, such as horizontal flips, rotations and color jitter, are applied to increase the dataset diversity and prevent overfitting. A sample of code snippet can be found in Appendix A.

4 Model Architectures

Image classification is a fundamental task in computer vision which assigns a specific category to an image based on its visual content. Early image classification methods flatten images into pixels and use traditional feedforward neural networks or heuristic algorithms like SIFT and SURF [6]. With a successful breakthrough in image classification accuracy by AlexNet using the deep layers, ReLU activation and dropout regularization, convolutional neural networks (CNNs) begin to be widely adopted in image classification [7]. Later with an introduction of the attention mechanism and Vision Transformer (ViT), the performance of classification is further improved for large-scale tasks [8].

Following the history and literature reviews, we build our AI image classifier by implementing a custom CNN and a transfer learning of ResNet and ViT architecture. We present the architecture overviews, training and testing results in the following three subsections, and analyze the performance of three models in the next sections.

4.1 Custom Convolutional Neural Network

A Convolutional Neural Network is a special type of feedforward neural network designed to learn spatial features adaptively through convolutional filters or kernels [9]. Similar to many neural networks, neurons in CNN layers, however, are organized into three dimensions (height, width and depth) representing the spatial information. Therefore, CNN is suitable for performing image-related machine learning (color channels for depth).

A CNN is mainly composed of convolutional layers, pooling layers and fully-connected layers [9]. The convolutional layer is used to extract spatial features, such as edges and textures of an image, by applying learnable filters to the input; the pooling layer is responsible for reducing spatial dimensions to prevent overfitting while preserving important features; the fully-connected layer is used to combine features from previous layers for classification and output the final category class information of the image.

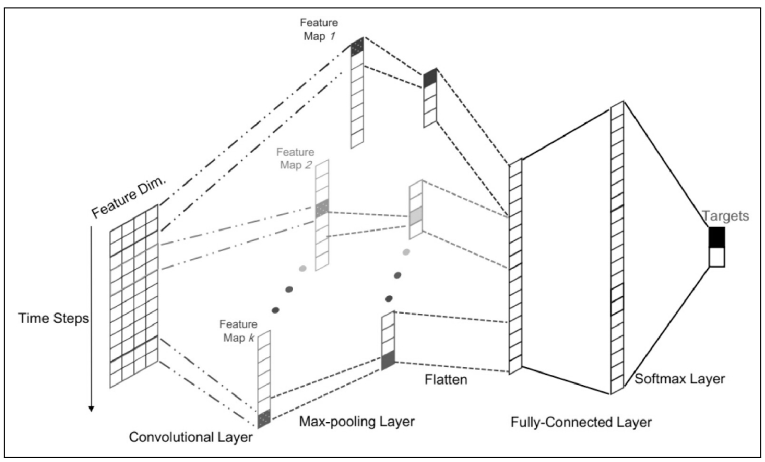

Figure 2: An example of CNN layer structure [10]

4.1.1 Architecture Design

Following the introduction, we design our CNN architecture for AI image classifier. See Figure 3.

In convolutional layers, the spatial features are extract from low level (edges, shapes) to high level by layers. Each output size could be calculated by $(\lfloor\frac{W-K+2P}{S}\rfloor+1)\times(\lfloor\frac{H-K+2P}{S}\rfloor+1)$, where W is input width, H is input width, K is kernel size, P is padding, S is stride.

After each convolutional layer, we apply Batch Normalization and ReLU activation function. Batch Normalization helps maintain a stable input distribution and activation function helps the network learn more effectively by avoiding dead neurons.

In pooling layers, the dimensions are reduced by $(\lfloor\frac{W-K}{S}\rfloor+1)\times(\lfloor\frac{H-K}{S}\rfloor+1)$, where W is input width, H is input width, K is kernel size, S is stride.

After three blocks of convolutional layer and the pooling layer, the outputs are flatten from $28\times 28\times 64$ to 50176 neutrons in the input layer of fully-connected layer, and dropoutted by 40% probability to prevent overfitting. The output layer consists of 2 neutrons for binary classification.

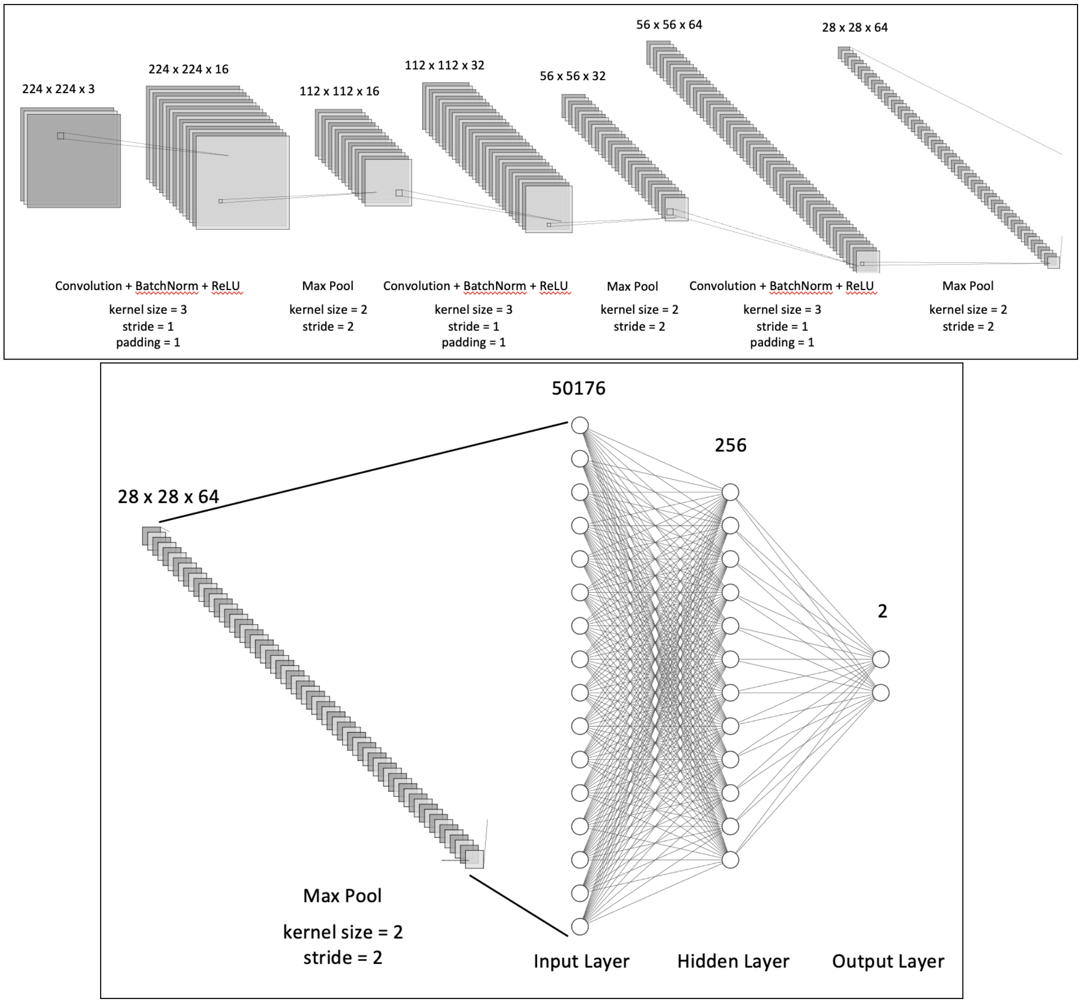

Figure 3: CNN architecture of AI Image Classifier

4.1.2 Training Process

The dataset is first loaded into two lists, the train and test datasets, where each image is labelled 0 for the AI-generated image, 1 for the human-crafted image. The dataset is fixed throughout the training.

During the model training, each image in the train dataset is passed through the model architecture and we obtain the two neutron logit values in the output layer. The logits are directly converted to class predictions by choosing the label with the highest logit. For example, if the output logits are [2.2, -1.3], the predicted class is 0 (AI Image) as 2.2 > -1.3. We do not apply softmax function for prediction probabilities as the loss function we use is Cross Entropy Loss expecting logits as inputs. We also use Adaptive Moment Estimation (Adam) as an optimizer for adaptive learning rate and faster convergence.

After the classification of one image, the loss is calculated by Cross Entropy Loss and propagated back to the architecture layers. The accuracy is calculated by comparing the predicted class with the ground-truth class. All the loss and accuracy are cumulated for data visualization and analysis.

The training is repeated for all the images in train dataset in one epoch. The model is trained for 10 epochs and the following shows a training log. A complete Python program is attached in the Appendix B.

```
===== Start training =====
Epoch [1/10], Loss: 0.4317, Train Acc: 82.40%
Test Acc: 83.46%
Epoch [2/10], Loss: 0.3268, Train Acc: 86.66%
Test Acc: 88.50%
Epoch [3/10], Loss: 0.2971, Train Acc: 87.93%
Test Acc: 87.73%
Epoch [4/10], Loss: 0.2719, Train Acc: 88.91%
Test Acc: 91.11%
Epoch [5/10], Loss: 0.2498, Train Acc: 89.97%
Test Acc: 89.16%
Epoch [6/10], Loss: 0.2338, Train Acc: 90.69%
Test Acc: 90.25%
Epoch [7/10], Loss: 0.2177, Train Acc: 91.40%
Test Acc: 90.51%
Epoch [8/10], Loss: 0.2025, Train Acc: 92.09%
Test Acc: 93.12%
Epoch [9/10], Loss: 0.1887, Train Acc: 92.56%
Test Acc: 91.15%
Epoch [10/10], Loss: 0.1749, Train Acc: 93.12%
Test Acc: 93.46%
===== Finish training =====
```

Listing 1: CNN Model Training Log

4.1.3 Results and Analysis

Figure 4 shows two data visualizations of training loss and test accuracy over epochs.

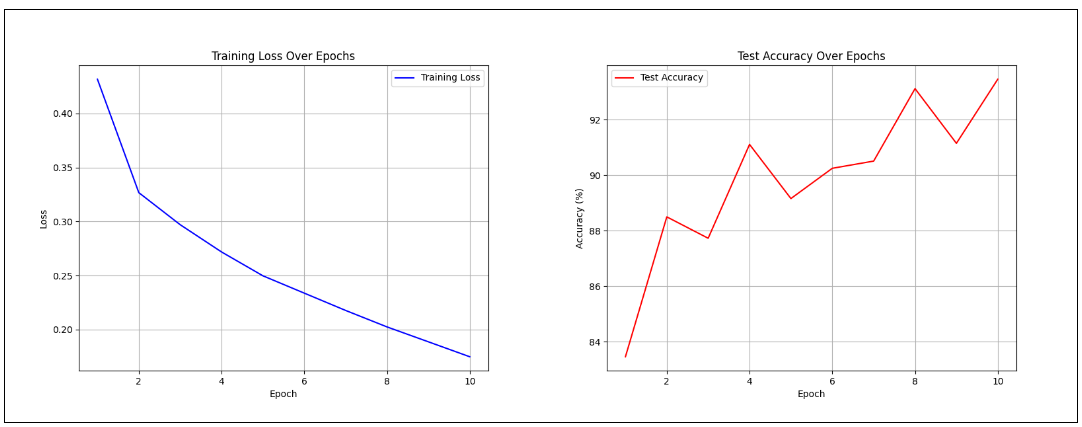

Figure 4: CNN Training Loss (left) and Test Accuracy (right)

In the training loss, the loss starts at 0.43 and constantly decreases to about 0.17 after 10 epochs. The steady downward trend indicates that the model learns and optimizes well on the training data. The loss has not fully flattened out by epoch 10, suggesting that the model can be improved with more training epochs.

In the test accuracy, the accuracy starts at 0.82 and increases to approximately 0.93 after 10 epochs, with some fluctuations along the trend. The overall upward trend is positive, showing that the model generalizes well to the test set as training progresses. The fluctuations suggest some instability in the learning process. This could be due to the reason of the model overshooting optimal weights with a high learning rate, or the model struggles with some variability in test dataset.

Figure 5 shows a confusion matrix of the classification model after 10 epochs. We calculate the accuracy, precision, recall, and F1 score based on the confusion matrix.

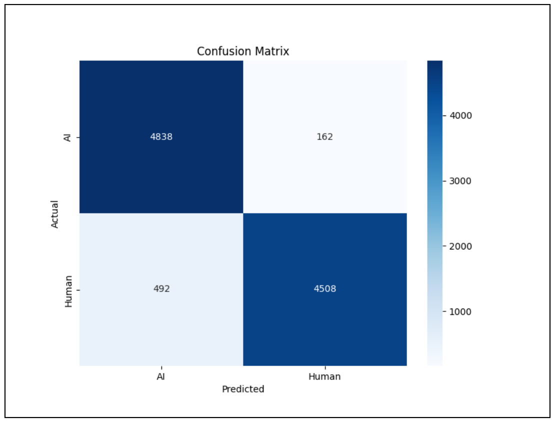

Figure 5: CNN Confusion Matrix

Accuracy	Precision	Recall	F1-Score
93.46%	90.77%	96.76%	93.67%

Table 1: Accuracy, Precision, Recall, and F1 Score

The model performs better at identifying AI images (higher recall) but has more false positives (492 human images misclassified as AI images) than false negatives (162 AI images misclassified as human images). This suggests the model might be slightly biased toward predicting "AI images". The model in general achieves a satisfactory accuracy of 93%.

4.2 ResNet

ResNet, or Residual Neural Network, is a pioneering deep learning architecture introduced in 2015 by researchers at Microsoft Research [11]. It tackles the challenge of training very deep neural networks by incorporating residual connections, which enable layers to learn residual functions relative to their inputs. This approach facilitates the training of networks with hundreds of layers, significantly enhancing performance in image recognition tasks. ResNet achieved remarkable success by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2015, demonstrating its robustness and effectiveness in computer vision applications. Therefore, we applied ResNet for the classification task.

4.2.1 Architecture Design

ResNet50 is a 50-layer deep convolutional neural network designed to address vanishing gradients in deep networks through residual connections. The architecture begins with an input layer processing $224 \times 224 \times 3$ images, followed by an initial 7×7 convolution (stride 2) and 3×3 max pooling (stride 2), reducing spatial dimensions to $56 \times 56 \times 64$. These layers extract low-level features while halving resolution for computational efficiency[12].

The core of ResNet50 consists of four stages of bottleneck residual blocks. Each block compresses channels with a 1×1 convolution, processes features via a 3×3 convolution, and restores dimensions with another 1×1 convolution. A residual connection skips these operations, adding the input directly to the output to preserve gradient flow. **Stage 2** (3 blocks) maintains 56×56 resolution but increases depth to 256 channels, balancing detail retention and feature complexity[12].

Stage 3 (4 blocks) downsamples to 28×28 resolution using strided convolutions in the first block, expanding to 512 channels. This stage captures mid-level features like textures and patterns. **Stage 4** (6 blocks) further reduces resolution to 14×14 while increasing channels to 1024, enabling the network to learn hierarchical representations of objects[12].

Layer Name	Output Size	Configuration	Repetitions
Input	$224 \times 224 \times 3$	-	-
Conv1	$112 \times 112 \times 64$	7×7, 64, stride 2	1
Pool1	$56 \times 56 \times 64$	3×3 max pool, stride 2	1
Stage 2 (Conv2_x)	$56 \times 56 \times 256$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix}$ Bottleneck block, Residual connection	3
Stage 3 (Conv3_x)	$28 \times 28 \times 512$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix}$ Bottleneck block, Residual connection	4
Stage 4 (Conv4_x)	$14 \times 14 \times 1024$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix}$ Bottleneck block, Residual connection	6
Stage 5 (Conv5_x)	$7 \times 7 \times 2048$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix}$ Bottleneck block, Residual connection	3
Global Avg Pool	$1 \times 1 \times 2048$	7×7 global average pool	1
FC	1000	Fully connected, softmax	1

Table 2: ResNet50 Architecture

The final residual stage (**Stage 5**, 3 blocks) operates at 7×7 resolution with 2048 channels, focusing on high-level semantic features. The network concludes with global average pooling, collapsing spatial dimensions to $1 \times 1 \times 2048$, and a fully connected layer for classification. This design ensures efficient training while achieving state-of-the-art accuracy on tackling tasks like image classification[12].

4.2.2 Training Process

The implementation uses a transfer learning approach with a pre-trained ResNet50 model to classify between human and AI-generated images. The base ResNet architecture, pre-trained on ImageNet, serves as a feature extractor with its weights frozen to preserve learned representations. Only the final classifier has been customized with a two-layer neural network (512 hidden units with ReLU activation and dropout regularization) that outputs two classes. This design allows the model to leverage general visual features while specializing in detecting subtle differences between human and AI-generated content.

During training, the model processes images in batches of 16 through 10 epochs, using Adam optimizer with a learning rate of 0.001 and Cross Entropy Loss. Performance is tracked through training losses and test accuracies, with the best-performing epoch's model weights saved for inference. The validation process produces a confusion matrix showing the model's discrimination capability between human and AI-generated content. This efficient approach enables accurate classification while requiring minimal computational resources by training only the task-specific layers rather than the entire network, complete Python program is attached in the Appendix C.

```
===== Start training =====
Epoch [1/10], Loss: 0.3948, Train Acc: 81.87%
Test Acc: 88.98%
Epoch [2/10], Loss: 0.3383, Train Acc: 85.35%
Test Acc: 89.36%
Epoch [3/10], Loss: 0.3226, Train Acc: 86.22%
Test Acc: 90.44%
Epoch [4/10], Loss: 0.3118, Train Acc: 86.65%
Test Acc: 91.37%
Epoch [5/10], Loss: 0.3061, Train Acc: 87.02%
Test Acc: 90.33%
```

```
Epoch [6/10], Loss: 0.2957, Train Acc: 87.58%
Test Acc: 92.03%
Epoch [7/10], Loss: 0.2915, Train Acc: 87.65%
Test Acc: 91.61%
Epoch [8/10], Loss: 0.2889, Train Acc: 87.94%
Test Acc: 92.23%
Epoch [9/10], Loss: 0.2841, Train Acc: 88.46%
Test Acc: 92.52%
Epoch [10/10], Loss: 0.2802, Train Acc: 89.93%
Test Acc: 92.91%
===== Finish training =====
```

Listing 2: ResNet Model Training Log

4.2.3 Results and Analysis

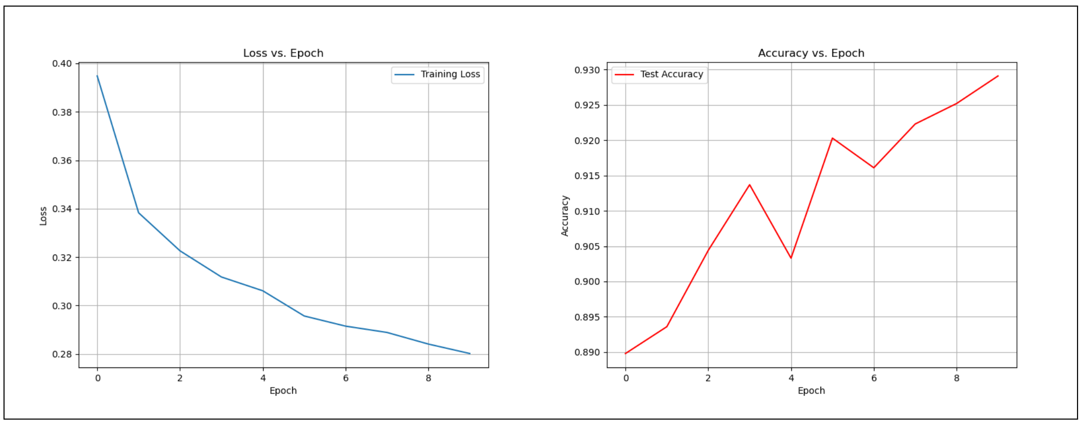

Figure 6: ResNet50 Training Loss (left) and Test Accuracy (right)

In the training loss, the loss starts at 0.39 and constantly decreases to about 0.12 after 10 epochs. The steady downward trend indicates that the model learns and optimizes moderately on the training data. The loss tends to be flattened out by epoch 10, indicating that the model can be improved slightly with more training epochs.

In the test accuracy, the accuracy starts at 0.82 and increases to approximately 0.93 after 10 epochs, with some fluctuations along the trend. The overall upward trend is positive, showing that the model performs good generalization to the test set as training progresses. The fluctuations suggest some instability in the learning process. This could be due to the reason of the batch training process, causing variation among the data samples in the batch.

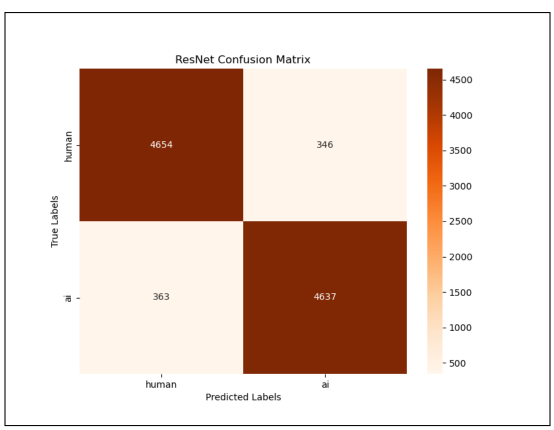

Figure 7: ResNet50 Confusion Matrix of Best Epoch

Accuracy	Precision	Recall	F1-Score
92.91%	93.05%	92.74%	92.89%

Table 3: Accuracy, Precision, Recall, and F1 Score

Figure 7 shows a confusion matrix of the classification model after 10 epochs. We can observe that the model is well balanced on the Accuracy, Precision, Recall, F1-Score as the True Negative equally high as True Positive, and False Negative equally low as False positive indicates the maturity of ResNet50 model on performing classification task among images.

4.3 Vision Transformer

The Vision Transformer (ViT) is a deep learning model introduced in 2020 by Dosovitskiy et al. for image classification, adapting the transformer architecture from natural language processing to computer vision [1]. Unlike convolutional neural networks (CNNs), ViT processes images by dividing them into fixed-size patches (e.g., 16x16 pixels), flattening each patch into a vector, and embedding it into a sequence of tokens. These tokens, along with a special classification (CLS) token, are fed into a series of transformer encoder layers that use self-attention to capture global relationships across the image. Positional embeddings are added to retain spatial information. ViT's output, typically the CLS token's embedding, is passed through a linear layer for classification. ViT excels in tasks like ImageNet classification when pre-trained on large datasets, offering advantages in capturing long-range dependencies compared to CNNs, though it requires substantial computational resources [13].

4.3.1 Architecture Design

Layer Name	**Output Size**	**Configuration**	**Repetitions**
Input	$224 \times 224 \times 3$	-	-
Patch Embedding	197×768	Patch size 16×16, 768 dim, flatten and linear projection	1
CLS Token	1×768	Prepend learnable CLS token	1
Positional Embedding	197×768	Add learnable 1D positional embeddings	1
Transformer Encoder	197×768	Multi-Head Self-Attention (12 heads, 768 dim) LayerNorm, MLP (3072 dim, GELU) Residual connections, LayerNorm Dropout (0.1)	12
LayerNorm	197×768	Final LayerNorm	1
CLS Token Output	1×768	Extract CLS token embedding	1
MLP Head	1000	Linear layer, softmax	1

Table 4: ViT-B/16 Architecture

ViT-B/16 processes images as sequences of 16×16 patches, linearly projecting them to 768D embeddings. A learnable `[CLS]` token and positional embeddings are added to the 197 resulting tokens (14×14 patches + `[CLS]`). The core architecture consists of 12 identical transformer layers, each featuring:

- Multi-head self-attention (12 heads)
- MLP (3072 hidden units)
- Pre-layer normalization
- Residual connections

The model maintains a 197×768 representation throughout, applying dropout ($p = 0.1$) for regularization. After the final layer norm, the [CLS] token's 768D embedding is extracted for classification via a linear head. This pure transformer approach demonstrates that global self-attention can effectively replace convolutions for vision tasks.

Key features: Patch embeddings, learned positional encoding, [CLS] token aggregation, and standard transformer components adapted for visual data.

4.3.2 Training Process

During training, the model processes images in batches of 16 through 10 epochs, using Adam optimizer with a slightly lower learning rate of 0.0005 to avoid destabilizing the sensitive transformer weights, and Cross Entropy Loss for classification. Performance metrics including training loss and test accuracy are tracked throughout the process, with the best-performing model weights saved for inference. The validation phase generates a confusion matrix that visualizes the model's effectiveness in discriminating between human and AI-generated content, complete Python program is attached in the Appendix D.

```
===== Start training =====
Epoch [1/10], Loss: 0.1763, Train Acc: 93.42%
Test Acc: 95.24%
Epoch [2/10], Loss: 0.1683, Train Acc: 94.54%
Test Acc: 96.36%
Epoch [3/10], Loss: 0.1526, Train Acc: 94.63%
Test Acc: 96.44%
Epoch [4/10], Loss: 0.1398, Train Acc: 93.56%
Test Acc: 95.37%
Epoch [5/10], Loss: 0.1312, Train Acc: 94.52%
Test Acc: 96.33%
Epoch [6/10], Loss: 0.1301, Train Acc: 95.10%
Test Acc: 96.91%
Epoch [7/10], Loss: 0.1247, Train Acc: 95.01%
Test Acc: 96.82%
Epoch [8/10], Loss: 0.1178, Train Acc: 95.23%
Test Acc: 97.04%
Epoch [9/10], Loss: 0.0941, Train Acc: 95.08%
Test Acc: 96.89%
Epoch [10/10], Loss: 0.0902, Train Acc: 95.30%
Test Acc: 97.11%
===== Finish training =====
```

Listing 3: ViT Model Training Log

4.3.3 Results and Analysis

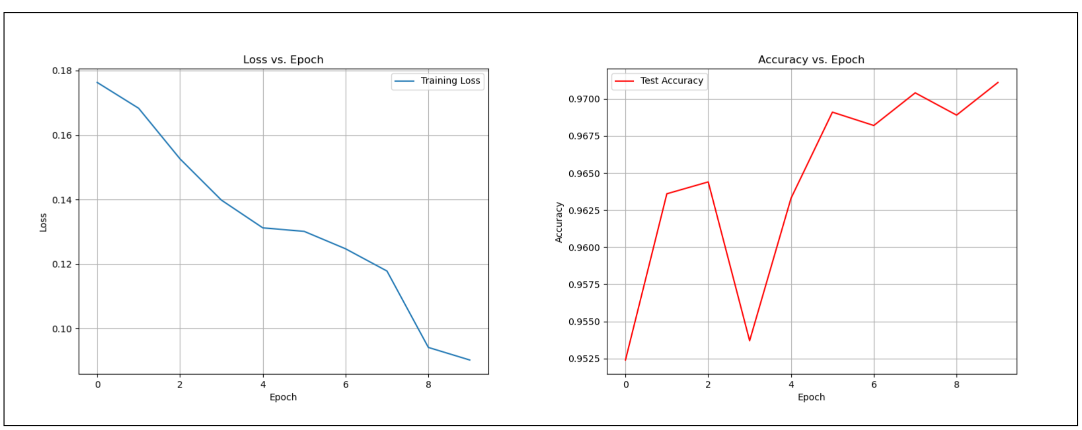

Figure 8: ViT Training Loss (left) and Test Accuracy (right)

In the training loss, the loss starts at 0.18 and constantly decreases to less than 0.1 after 10 epochs. The training loss is satisfactory that indicates the incredibility of ViT on handling image classification tasks. The loss is still having a linear trend after 10 epochs, indicating that the model can be improved with more epochs.

In the test accuracy, the accuracy starts at 0.95 and increases to over 0.97 after 10 epochs, with some fluctuations along the trend. The overall upward trend is positive, showing that the model performs good generalization to the test set as training progresses. The fluctuations suggest some instability in the learning process. This could be due to the reason of the batch training process, causing variation among the data samples in the batch.

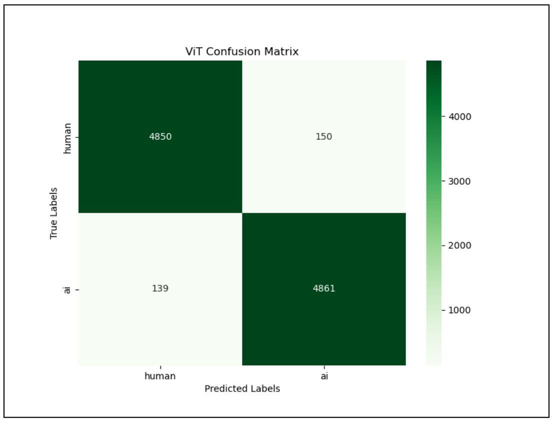

Figure 9: ViT Confusion Matrix of Best Epoch

Accuracy	Precision	Recall	F1-Score
97.11%	97.01%	97.22%	97.11%

Table 5: Accuracy, Precision, Recall, and F1 Score

Figure 9 shows a confusion matrix of the classification model after 10 epochs. We can observe that the model is well balanced on the Accuracy, Precision, Recall, F1-Score of roughly 97% as the True Negative equally high as True Positive, and False Negative equally low as False positive indicates the maturity of ViT model on performing classification task among images.

5 Discussion

Among all 3 models implemented, ViT successfully achieved high accuracy with 97%, which is higher than our expectation of 95% , which provide a good evidence on its ability to perform classification tasks. While the traditional hierarchical processing like out custom CNN model and ResNet50 performs slightly lower than our expectaion, yet still achive a considerable accuracy of around 93%.

We have discussed regarding the reason, and we believe that Vision Transformer (ViT) outperforms ResNet50 primarily because its self-attention mechanism captures global relationships between image patches simultaneously, unlike ResNet's hierarchical local processing through convolutional layers. ViT's ability to model long-range dependencies across the entire image is particularly advantageous for detecting subtle artifacts and unnatural patterns characteristic of AI-generated content. Additionally, ViT's patch-based representation and fewer inductive biases about image structure allow it to better identify the statistical irregularities and perceptual inconsistencies that distinguish AI-generated images from human-created ones [14].

Also, the accuracy of our custom CNN and ResNet50 performs similarly, and we believe the CNN's tailored architecture is optimized specifically for this binary classification task, while we have used the pretained ResNet50 model that may not specifically suitable for such task, with a too small learning rate that hindered the learning process.

Due to the limited project time and report space, we could not do further studies on our AI image detector. Therefore, we suggest one possible further action is to deploy our models to applications such as websites, so that our detector is practical to use, and we can collect feedback and more test datasets from users to improve our model. Another possible further study is to investigate the datasets and the potential reasons for misclassified images. There could be some false data or findings of AI-generated images perfectly imitating human images, which enhances our understanding towards this topic.

6 Conclusion

This paper targets the problem of high similarity between AI-generated images and human-crafted images, proposes three classification models, namely custom Convolutional Neural Network, ResNet and Vision Transformer, and analyzes their performance and effectiveness.

As the generative models progress, AI-generated images, which forge the human-crafted images, are widely spread on the Internet, posing artwork copyright infringement and information security challenges. An effective solution would be to develop a highly accurate AI image detector that classifies the AI-generated images and human-crafted images. The architecture behind the detector is a binary classification model, and three different classification models, namely custom Convolutional Neural Network, ResNet and Vision Transformer are presented.

Data has been first collected, extracted, and pre-processed by normalization and augmentation methods. Custom Convolutional Neural Network, mainly composed of 3 blocks of convolutional layers and max-pooling layers, and a 3-fully-connected layers, provide a good accuracy of 93%. For ResNet, it achieves a considerable accuracy of 93%, which is in line with the model performance. For Visual Transformer, it achieves a satisfactory accuracy of 97%, which we believe is capable of providing a reliable suggestion on distinguishing AI-generated images and human-crafted images.

The result shows that all three approaches are robust models for AI-human image classification. It is suggested that deployment to the application and in-depth studies on the dataset could be further explored.

References

[1] “South China Morning Post,” *South China Morning Post*, Apr. 02, 2025. https://www.scmp.com/tech/big-tech/article/3304828/chatgpt-usage-hits-record-studio-ghibli-style-ai-images-go-viral?module=perpetual_scroll_0&pgtype=article (accessed Apr. 04, 2025).

[2] N. Tiku, “AI can now create any image in seconds, bringing wonder and danger,” *Washington Post*, Sep. 28, 2022. https://www.washingtonpost.com/technology/interactive/2022/artificial-intelligence-images-dall-e/

[3] H. Y. Yan, G. Morrow, K.-C. Yang, and J. Wihbey, “The origin of public concerns over AI supercharging misinformation in the 2024 U.S. presidential election,” *The origin of public concerns over AI supercharging misinformation in the 2024 U.S. presidential election*, Jan. 2025, doi: https://doi.org/10.37016/mr-2020-171.

[4] A. Sala, “AI vs. Human-Generated Images,” *Kaggle.com*, 2025. https://www.kaggle.com/datasets/alessandrasala79/ai-vs-human-generated-dataset (accessed May 05, 2025).

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2012.

[6] “Understanding CNN for Image Processing | Svitla Systems,” *Svitla Systems*, Jun. 21, 2024. https://svitla.com/blog/cnn-for-image-processing/

[7] “The evolution of image classification explained,” *Stanford.edu*, 2012. https://stanford.edu/ shervine/blog/evolution-image-classification-explained

[8] L. Chen, S. Li, Q. Bai, J. Yang, S. Jiang, and Y. Miao, “Review of Image Classification Algorithms Based on Convolutional Neural Networks,” *Remote Sensing*, vol. 13, no. 22, p. 4712, Nov. 2021, doi: https://doi.org/10.3390/rs13224712.

[9] Keiron O’Shea and R. R. Nash, “An Introduction to Convolutional Neural Networks,” *arXiv* (Cornell University), Nov. 2015, doi: https://doi.org/10.48550/arxiv.1511.08458.

[10] “Convolutional Neural Network - an overview | ScienceDirect Topics,” *www.sciencedirect.com.* https://www.sciencedirect.com/topics/engineering/convolutional-neural-network

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *www.cv-foundation.org* Dec. 2015. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

[12] B. Koonce, "ResNet 50," Apress eBooks, pp. 63–72, Jan. 2021, doi:https://doi.org/10.1007/978-1-4842-6168-2_6

[13] S.-H. Tsang, “Review: Vision Transformer (ViT),” Medium, Feb. 05, 2022. https://sh-tsang.medium.com/review-vision-transformer-vit-406568603de0

[14] Nouar AlDahoul and Y. Zaki, “Detecting AI-Generated Images Using Vision Transformers: A Robust Approach for Safeguarding Visual Media Integrity,” Jan. 2025, doi: https://doi.org/10.2139/ssrn.5029893

Appendices

A Data Transformation Python Code Snippet

```
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])
```

B CNN Image Classifier Python Program

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import os
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix

train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])

def load_dataset(data_dir):

    train_dataset = ImageFolder(root=os.path.join(data_dir, 'train'),
    transform=train_transform)
    test_dataset = ImageFolder(root=os.path.join(data_dir, 'test'),
    transform=test_transform)

```

```python
    train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

    return train_loader, test_loader

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.relu = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Convolutional Layer 1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)

        # Convolutional Layer 2
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(32)

        # Convolutional Layer 3
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(64)

        # Fully Connected Layers
        self.dropout = nn.Dropout(0.4)
        self.fc1 = nn.Linear(28 * 28 * 64, 256)
        self.fc2 = nn.Linear(256, 2)

    def forward(self, x):

        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.pool(self.relu(self.bn2(self.conv2(x))))
        x = self.pool(self.relu(self.bn3(self.conv3(x))))

        x = x.view(x.size(0), -1)

        x = self.dropout(x)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def train_model(model, train_loader, test_loader, criterion, optimizer, num_epochs, device):

    best_val_acc = 0.0
    train_losses = []
    test_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_loss = running_loss / len(train_loader)
        train_losses.append(train_loss)
        train_acc = 100 * correct / total
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/
    len(train_loader):.4f}, Train Acc: {train_acc:.2f}%")

        # Testing
        model.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                test_total += labels.size(0)
                test_correct += (predicted == labels).sum().item()

        val_acc = 100 * test_correct / test_total
        test_accuracies.append(val_acc)
        print(f"Test Acc: {val_acc:.2f}%")

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), 'cnn_model.pth')

    return train_losses, test_accuracies

# Plot Graphs
def plot_training_loss(losses):
    save_path='training_loss.png'
    plt.figure(figsize=(8, 6))
    plt.plot(range(1, len(losses) + 1), losses, 'b-', label='Training
    Loss')
    plt.title('Training Loss Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.legend()
    plt.savefig(save_path)
    plt.close()

def plot_test_accuracy(accuracies):
    save_path='test_accuracy.png'
    plt.figure(figsize=(8, 6))
    plt.plot(range(1, len(accuracies) + 1), accuracies, 'r-', label='
    Test Accuracy')
    plt.title('Test Accuracy Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.grid(True)
    plt.legend()
    plt.savefig(save_path)
    plt.close()

def plot_confusion_matrix(true_labels, pred_labels):
```

```python
    save_path='confusion_matrix.png'
    classes = ['AI', 'Human']
    cm = confusion_matrix(true_labels, pred_labels)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.savefig(save_path)
    plt.close()

def compute_confusion_matrix(model, val_loader, device):
    model.eval()
    true_labels = []
    pred_labels = []

    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            true_labels.extend(labels.cpu().numpy())
            pred_labels.extend(predicted.cpu().numpy())

    return true_labels, pred_labels

if __name__ == "__main__":

    data_dir = 'dataset'
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    num_epochs = 10

    try:
        train_loader, test_loader = load_dataset(data_dir)

        # Verify data loading
        for images, labels in train_loader:
            print(f"Training batch shape: {images.shape}, Labels: {labels}")
            break

        for images, labels in test_loader:
            print(f"Testing batch shape: {images.shape}, Labels: {labels}")
            break

        print(f"Class mapping: {train_loader.dataset.class_to_idx}")

        # Initialize model, loss, and optimizer
        model = CNN().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        # Train the model
        print("===== Start training =====")
        train_losses, test_accuracies = train_model(model, train_loader, test_loader, criterion, optimizer, num_epochs, device)
        print("===== Finish training =====")

        # Plot graphs
        plot_training_loss(train_losses)
        plot_test_accuracy(test_accuracies)
```

```
        true_labels, pred_labels = compute_confusion_matrix(model, test_loader, device)
        plot_confusion_matrix(true_labels, pred_labels)

    except Exception as e:
        print(f"Error: {str(e)}")
```

C ResNet Image Classifier Python Program

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset
from torchvision import models
from torchvision.models import ResNet50_Weights  # Import weights
from PIL import Image
import os
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Import transforms and loader from transform.py
from transform import test_transform, load_dataset

# Configuration
class Config:
    IMAGE_SIZE = 224
    BATCH_SIZE = 16
    NUM_EPOCHS = 10
    LEARNING_RATE = 0.001
    NUM_CLASSES = 2
    DATA_DIR = r"\dataset" # Directory to load data
    DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    FIGURE_DIR = r"\ResNet_Result"  # Directory to save figures

os.makedirs(Config.FIGURE_DIR, exist_ok=True)

# Create datasets and dataloaders using transforms from transform.py
def create_dataloaders():
    print("Loading datasets...")
    train_loader, test_loader = load_dataset(Config.DATA_DIR, Config.BATCH_SIZE)
    print(f"Using device: {Config.DEVICE}")
    return train_loader, test_loader

# ResNet-based Model
class ResNet(nn.Module):
    def __init__(self, num_classes=Config.NUM_CLASSES):
        super(ResNet, self).__init__()
        # Use a pretrained ResNet model
        self.resnet = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)

        for param in self.resnet.parameters():
            param.requires_grad = False

        num_ftrs = self.resnet.fc.in_features
        self.resnet.fc = nn.Sequential(
            nn.Linear(num_ftrs, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
```

```python
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        return self.resnet(x)

# Plot training and validation metrics
def plot_metrics(train_losses, train_accuracies, test_losses, test_accuracies):
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss vs. Epoch')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'train_loss_plot.png'), dpi=300)

    plt.figure(figsize=(10, 5))
    plt.plot(test_accuracies, label='Test Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs. Epoch')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'test_accuracy_plot.png'), dpi=300)

    plt.show()

# Plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, class_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Oranges', xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('ResNet Confusion Matrix')
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'confusion_matrix.png'), dpi=300)
    plt.show()

def train_model(model, dataloaders, criterion, optimizer, num_epochs=Config.NUM_EPOCHS):
    best_acc = 0.0
    best_epoch = 0

    # Lists to store metrics for plotting
    train_losses = []
    train_accuracies = []
    test_losses = []
    test_accuracies = []
    best_epoch_labels = []
    best_epoch_predictions = []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 20)

        current_epoch_labels = []
        current_epoch_predictions = []

        for phase in ['train', 'test']:
```

```
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            total = len(dataloaders[phase])

            #progress bar
            with tqdm(total=total, desc=f'{phase}') as pbar:
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(Config.DEVICE)
                    labels = labels.to(Config.DEVICE)

                    optimizer.zero_grad()

                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)

                    # Store test predictions for current epoch
                    if phase == 'test':
                        current_epoch_labels.extend(labels.cpu().numpy())
                        current_epoch_predictions.extend(preds.cpu().numpy())

                    pbar.update(1)
                    pbar.set_postfix({'loss': f'{loss.item():.4f}'})

            # Calculate epoch statistics
            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # Store metrics for plotting
            if phase == 'train':
                train_losses.append(epoch_loss)
                train_accuracies.append(epoch_acc.cpu().numpy())
            else:
                test_losses.append(epoch_loss)
                test_accuracies.append(epoch_acc.cpu().numpy())

                if epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_epoch = epoch
                    torch.save(model.state_dict(), 'best_ResNet_model.pth')

                    best_epoch_labels = current_epoch_labels.copy()
```

```
                    best_epoch_predictions = current_epoch_predictions.copy()

        print()

    print(f'Best test accuracy: {best_acc:.4f} at epoch {best_epoch+1}')

    plot_metrics(train_losses, train_accuracies, test_losses, test_accuracies)

    print(f"Creating confusion matrix from epoch {best_epoch+1} (best accuracy)")
    plot_confusion_matrix(best_epoch_labels, best_epoch_predictions, class_names=['human', 'ai'])

    return model

def main():
    print(f"PyTorch version: {torch.__version__}")

    model = ResNet().to(Config.DEVICE)

    criterion = nn.CrossEntropyLoss()

    optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),
                           lr=Config.LEARNING_RATE)

    train_loader, test_loader = create_dataloaders()
    dataloaders = {'train': train_loader, 'test': test_loader}

    print(model)

    model = train_model(model, dataloaders, criterion, optimizer, Config.NUM_EPOCHS)

    print('Training complete. Models saved.')

if __name__ == '__main__':
    main()

# Interface for classification
def predict_image(image_path, model_path='best_Resnet_model.pth'):
    # Load the model
    model = ResNet().to(Config.DEVICE)
    model.load_state_dict(torch.load(model_path, map_location=Config.DEVICE))
    model.eval()

    image = Image.open(image_path).convert('RGB')
    image_tensor = test_transform(image).unsqueeze(0).to(Config.DEVICE)

    with torch.no_grad():
        outputs = model(image_tensor)
        _, preds = torch.max(outputs, 1)
        probabilities = torch.nn.functional.softmax(outputs, dim=1)

    class_names = ['human', 'ai']
    return {
        'class': class_names[preds.item()],
        'confidence': probabilities[0][preds.item()].item(),
        'human_prob': probabilities[0][0].item(),
        'ai_prob': probabilities[0][1].item(),
```

```
        'is_human': preds.item() == 0
    }
```

D Visual Transformer Image Classifier Python Program

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset
from torchvision import models
from torchvision.models import ViT_B_16_Weights  # Import weights for ViT
from PIL import Image
import os
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.metrics import confusion_matrix
import seaborn as sns
from transform import test_transform, load_dataset

# Configuration
class Config:
    IMAGE_SIZE = 224
    BATCH_SIZE = 16
    NUM_EPOCHS = 10
    LEARNING_RATE = 0.0005
    NUM_CLASSES = 2
    DATA_DIR = r"\dataset"
    DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    FIGURE_DIR = r"\ViT_Result"

os.makedirs(Config.FIGURE_DIR, exist_ok=True)

# Create datasets and dataloaders
def create_dataloaders():
    print("Loading datasets...")
    train_loader, test_loader = load_dataset(Config.DATA_DIR, Config.BATCH_SIZE)
    print(f"Using device: {Config.DEVICE}")
    return train_loader, test_loader

class ViTClassifier(nn.Module):
    def __init__(self, num_classes=Config.NUM_CLASSES):
        super(ViTClassifier, self).__init__()
        self.vit = models.vit_b_16(weights=ViT_B_16_Weights.IMAGENET1K_V1)

        for param in self.vit.parameters():
            param.requires_grad = False

        # Replace the head (classification layer)
        self.vit.heads = nn.Sequential(
            nn.Linear(768, 512),
            nn.LayerNorm(512),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(512, num_classes)
        )

        for param in self.vit.heads.parameters():
            param.requires_grad = True

```

```python
        self.vit.encoder.pos_embedding.requires_grad = True

    def forward(self, x):
        return self.vit(x)

# Plot training and validation metrics
def plot_metrics(train_losses, train_accuracies, test_losses, test_accuracies):
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss vs. Epoch')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'train_loss_plot.png'), dpi=300)

    plt.figure(figsize=(10, 5))
    plt.plot(test_accuracies, label='Test Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs. Epoch')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'test_accuracy_plot.png'), dpi=300)

    plt.show()

# Plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, class_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Greens', xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('ViT Confusion Matrix')
    plt.savefig(os.path.join(Config.FIGURE_DIR, 'confusion_matrix.png'), dpi=300)
    plt.show()

def train_model(model, dataloaders, criterion, optimizer, num_epochs=Config.NUM_EPOCHS):
    best_acc = 0.0
    best_epoch = 0

    # Lists to store metrics for plotting
    train_losses = []
    train_accuracies = []
    test_losses = []
    test_accuracies = []
    best_epoch_labels = []
    best_epoch_predictions = []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 20)

        current_epoch_labels = []
        current_epoch_predictions = []

        for phase in ['train', 'test']:
            if phase == 'train':
```

```
                model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        total = len(dataloaders[phase])

        #progress bar
        with tqdm(total=total, desc=f'{phase}') as pbar:
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(Config.DEVICE)
                labels = labels.to(Config.DEVICE)

                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

                if phase == 'test':
                    current_epoch_labels.extend(labels.cpu().numpy())
                    current_epoch_predictions.extend(preds.cpu().numpy())

                pbar.update(1)
                pbar.set_postfix({'loss': f'{loss.item():.4f}'})

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

        print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

        # Store metrics for plotting
        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc.cpu().numpy())
        else:
            test_losses.append(epoch_loss)
            test_accuracies.append(epoch_acc.cpu().numpy())

            if epoch_acc > best_acc:
                best_acc = epoch_acc
                best_epoch = epoch
                torch.save(model.state_dict(), 'best_ResNet_model.pth')

                best_epoch_labels = current_epoch_labels.copy()
                best_epoch_predictions = current_epoch_predictions.copy()

```

```python
        print()

    print(f'Best test accuracy: {best_acc:.4f} at epoch {best_epoch+1}')

    plot_metrics(train_losses, train_accuracies, test_losses, test_accuracies)

    print(f"Creating confusion matrix from epoch {best_epoch+1} (best accuracy)")
    plot_confusion_matrix(best_epoch_labels, best_epoch_predictions, class_names=['human', 'ai'])

    return model

def main():
    print(f"PyTorch version: {torch.__version__}")

    model = ViTClassifier().to(Config.DEVICE)

    criterion = nn.CrossEntropyLoss()

    optimizer = optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()),
                            lr=Config.LEARNING_RATE, weight_decay=0.01)

    # Create dataloaders
    train_loader, test_loader = create_dataloaders()
    dataloaders = {'train': train_loader, 'test': test_loader}

    print(model)

    model = train_model(model, dataloaders, criterion, optimizer, Config.NUM_EPOCHS)

    torch.save(model.state_dict(), 'final_ViT_model.pth')
    print('Training complete. Models saved.')

if __name__ == '__main__':
    main()

# Interface for classification using ViT model
def predict_image(image_path, model_path='best_ViT_model.pth'):
    # Load the model
    model = ViTClassifier().to(Config.DEVICE)
    model.load_state_dict(torch.load(model_path, map_location=Config.DEVICE))
    model.eval()

    # Load and transform the image
    image = Image.open(image_path).convert('RGB')
    image_tensor = test_transform(image).unsqueeze(0).to(Config.DEVICE)

    with torch.no_grad():
        outputs = model(image_tensor)
        _, preds = torch.max(outputs, 1)
        probabilities = torch.nn.functional.softmax(outputs, dim=1)

    class_names = ['human', 'ai']
    return {
        'class': class_names[preds.item()],
        'confidence': probabilities[0][preds.item()].item(),
        'human_prob': probabilities[0][0].item(),
        'ai_prob': probabilities[0][1].item(),
```

```
            'is_human': preds.item() == 0
    }
```