Deepfake Image Detection

Shree Thavarekere
Department of Computer Science
Columbia University
New York, NY
srt2165@columbia.edu

Divya Sathiyamoorthy
Department of Computer Science
Columbia University
New York, NY
ds4221@columbia.edu

Abstract

Recently, apprehension towards the developments in Artificial Intelligence and Machine Learning has been rising rapidly. AI-generated content is becoming more and more sophisticated, which has led to several concerns being raised, such as plagiarism, privacy, loss of jobs, and malicious use. One specific category of AI-generated content is images - more specifically, Deepfakes. Deepfakes make use of deep learning techniques (especially generative adversarial networks) to generate or manipulate facial appearances. The detection of such Deepfakes, or equivalently, the classification of an image as real or AI-generated is, thus, a necessary and relevant problem in today's environment. This work aims to compare and contrast two different approaches to the aforementioned problem and perhaps suggest improvements.

1 Introduction

The word 'Deepfake' comes from 'deep learning' and 'fake.' These are images, video, or audio generated by Artificial Intelligence (AI) that appear real to the human eye or ear. The most common form of Deepfakes are those of human faces. The most widespread approach to create Deepfakes is to use generative adversarial networks (GANs) [1].

While Deepfakes are an interesting, and even useful area of research, unless we are able to detect them accurately, they can be dangerous. [2] lists several examples of illegal usage of Deepfakes in many ways such as crimes against individual people, crimes against organization (fraud, forgery, falsification of data, identity theft, etc.), and so-called 'reputational' attacks. [3] tests human ability to detect deepfakes, and reports that humans were able to correctly detect Deepfakes only $60 - 64\%$ of the time. Moreover, they state that familiarization exercises and explanations on detecting Deepfakes did not significantly improve people's ability to correctly detect the fake images.

Given that humans aren't great at detecting fake images, training AI systems to detect Deepfakes is the need of the hour. Deepfake-related research became prominent in 2018. Since then, the research in this field has increased exponentially, and several studies focus on detecting fake images [4]. There are several different methods and approaches used to detect Deepfakes. [5] outlines certain methods used to achieve Deepfake detection by machines, and classifies them broadly into two categories - feature-based and machine learning-based. They further claim that deep-learning based approaches achieve the highest accuracy.

In this work, we will focus on AI-generated images. We will compare two recent approaches to Deepfake image detection - [6] and [7], both of which utilize deep learning techniques to classify images (these two studies are summarized in the following section).

[8] reports that most models decrease in performance when trained and tested on different datasets (i.e, datasets other than the one they were originally designed for). So, one aspect of comparison that we will focus on is how well the models perform on different datasets.

Preprint. Under review.

2 Related Work

In [6], Rafique et. al. introduce an automated approach for classifying deep fake images by harnessing the potential of deep learning and machine learning techniques. They claim that traditional machine learning methods that are reliant on manually crafted features prove insufficient in capturing the intricate patterns inherent in deep fake content. Their proposed framework combines error-level analysis with convolutional neural networks (CNNs) and employs support vector machines (SVM) and k-nearest neighbors (KNN) for classification, achieving 89.5% accuracy with the Residual Network and KNN combination.
The methodology proposed in this paper for deep fake detection consists of three pivotal stages: Error Level Analysis (ELA) for preliminary image analysis, Feature Extraction via Convolutional Neural Networks (CNN) for in-depth feature capture, and Classification employing Support Vector Machines (SVM) and K-Nearest Neighbors (KNN) to form an integrated approach for detecting deep fake content. The ultimate goal is to proficiently discriminate between genuine and manipulated images, capitalizing on the capabilities of deep learning and machine learning techniques.
The dataset used in this work is the Real and Fake Face Database from Yonsei University's Computational Intelligence and Photography Lab [9].

Abir et. al. utilize deep learning techniques and Convolutional Neural Networks (CNN) for the purpose of identifying deepfake content in [7]. Emphasizing the importance of explainable artificial intelligence (XAI) for transparency, the study employs various CNN models, such as InceptionV3, ResNet152V2, DenseNet201, and InceptionResNetV2, achieving a remarkable accuracy of 99.87% with InceptionResNetV2.
To validate the models, the study incorporates the Local Interpretable Model-Agnostic Explanations (LIME) algorithm, explaining the models' classifications. Furthermore, the research suggests future enhancements, such as exploring transfer learning, extending XAI to video samples, and refining the model's focus on specific facial features for better deepfake detection.
The dataset used in this study is the 140k Real and Fake Faces from Kaggle [10].

3 Methodology

In this work, we implement two studies - [6] and [7]. We then compare the two works on the following criteria: (here, "own" dataset refers to the dataset that was used to train and test the model in the original study, and "different" dataset refers to a dataset that the model has not been designed for and has thus, never seen)

1. Complexity of implementation
2. Training time
3. Performance on own dataset
4. Performance on different dataset

The comparison is summarized in 1.

Note: If certain aspects of [6] and [7] were not relevant to our study, they were not included in our implementation.

3.1 Implementation of Paper 1 [6]

3.1.1 Data Preparation

The dataset used in this paper is [9]. The data consists of $600 \times 600 \times 3$ images. The dataset contains 1080 real and 960 fake images. For our analysis, we manually split the dataset into train and test datasets. The dataset comprises images in JPEG format, and the complete collection is accessible for download from Kaggle as a compressed zip file. The following steps were taken to preprocess the data before inputting it into the models.

1. **Pre-processing**:
   We performed error level analysis on the images to identify portions of image with varying compression levels.

(a) The image is resized according to the input layer specifications of a Convolutional Neural Network (CNN) model.

(b) The resized image is then recompressed with an error rate of around 95%. This involves saving the image in a compressed format, such as JPEG, with a noticeable loss of quality.

(c) The difference between the original resized image and the recompressed image is computed. This results in an Error Level Analysis (ELA) image, where areas with different compression levels are highlighted.

The following pseudocode illustrates the computation of the Error Level Analysis (ELA):

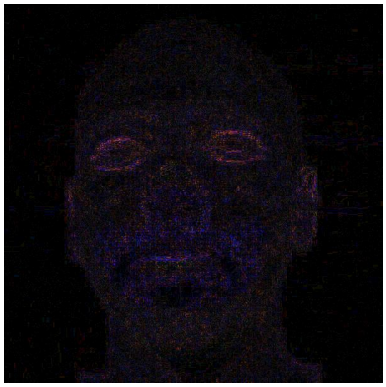
Algorithm 1 Error Level Analysis

for *each image in dataset* **do**
- resize image to (224, 224)
- compress the resized image with a 95% error rate
- compute the difference between the original resized and recompressed image
- enhance brightness of the ELA image and save it

end

The images generated by the algorithm are displayed below:

(a) ELA of Real Image



(b) ELA of Fake Image

3.1.2 Model Implementation

We chose to implement the three pre-trained models that were mentioned in [6] - ResNet18, GoogLeNet, and SqueezeNet for deep feature extraction.

1. The ResNet18 architecture is part of the Residual Network (ResNet) family, featuring several convolution layers with a 3x3 kernel size. The unique aspect of ResNets lies in their use of residual learning, where layers fit a residual mapping, and resulting outputs are added to those of stacked layers. This implementation of skip connections significantly reduces the loss of information during training, allowing for faster and more efficient training compared to conventional convolutional neural networks (CNNs).

2. Developed by Google researchers, GoogLeNet, also known as Inception v1, is a deep convolutional neural network renowned for its innovative inception modules and efficient architecture. With 22 layers, the network employs a 1×1 convolution filter size extensively to increase depth while minimizing computational complexity. GoogLeNet adopts a unique approach by incorporating global average pooling towards the end of the architecture. This involves inputting a 7×7 feature map and averaging it down to a compact 1×1 feature map. Such global average pooling reduces the number of trainable parameters, enhancing the network's overall performance.

3. Developed by researchers at UC Berkeley and Stanford University, SqueezeNet stands out as a highly efficient and lightweight convolutional neural network (CNN) architecture. Acknowledging the advantages of smaller CNNs, the design prioritizes reduced communication in distributed training, faster training times, and lower memory requirements, making it computationally economical. Additionally, the architecture employs "squeeze layers" to reduce the number of input channels processed by 3×3 filters, further diminishing the overall parameter count.

We used PyTorch to load these pre-trained models. The following pseudocode illustrates the training process of the model.

Algorithm 2 Deep Feature Extraction

```
torch.hub.load('pytorch/vision:v0.10.0', modelName, pretrained=True)
for each ELA image do
    - pre-process the ELA image by resizing and converting it to a tensor
    - add an extra dimension to the tensor to create a batch
    - pass the input batch through the model to obtain deep features
    return features
end
- store deep features in an array
- store corresponding real or fake label in an array
```

Following the extraction of deep features, we partitioned the dataset into training and testing sets. Subsequently, we employed Support Vector Machine (SVM) and k-nearest neighbors (KNN) classifiers, utilizing the optimal hyperparameter configurations detailed in the paper, to classify the deep features. The model's performance was assessed through rigorous evaluation. This entire procedure was iteratively applied to each of the three pre-trained models.

3.2 Implementation of Paper 2 [7]

3.2.1 Data Preparation

The dataset used in this paper is [10]. The data consists of $256 \times 256 \times 3$ images, split into train, test, and validation datasets. The training dataset contains 1 million such images, roughly split into half fake and half real images. The data is available in the form of CSVs with paths to the images in the directory. The entire dataset is available for download from Kaggle as a zip file.
The following steps were used to prepare the data for training (and testing):

1. **Dataset Creation:**
   (a) First, using the paths given in the CSV, the pixels of each image had to be extracted and stored in the CSV. This proved to be a very challenging task because, in this form, the dataset would occupy about 75 GB of space, which cannot fit into the RAM of even the GCE that we were using.
   (b) So, in order to process the dataset without storing the entire thing in memory, we made use of a library called `dask`. This enabled us to process the dataset one chunk at a time (called *partitions*) and then append to a CSV as the processing completed.
   (c) We then ran the processing for the test and validation datasets in parallel, but the train dataset had to be processed alone afterwards due to its enormous size. We stored the newly created datasets in a Google Cloud Storage bucket.
   (d) Even with the above technique, creating the datasets took a very long time (close to 36 hours for the training dataset alone).

   The train-test split was approximately $5 : 1$ (this is what was done in the paper).

The following pseudocode demonstrates how dataset creation was done:

Algorithm 3 Dataset Creation

repartition dataset into 20 partitions
for *each partition of the dataset* **do**
  - read path to image
  - convert image to array of pixels
  - add array of pixels to dataframe
  - append dataframe to final CSV
end

2. **Pre-processing:**

   (a) After the datasets were prepared, they had to be used to create arrays of data that could be fed into models. Again, `dask` was used to circumvent the problem of processing a dataset that can't fit into memory. We took advantage of the lazy execution of `dask` to run all the preprocessing steps on the arrays (converting strings to arrays of floating point numbers, reshaping arrays, etc.). The preprocessing was done block-wise on the arrays.

   (b) Once the arrays were ready to be given to the model, a generator function was used to generate one block of the array at a time.

The following pseudocode was followed to preprocess the dataset:

Algorithm 4 Data Preprocessing

- convert `dask` dataframe to `dask` array
for *each block in array* **do**
  - convert string of array to array
  - reshape array to $256 \times 256 \times 3$
  - add pre-processed block to new array
end

3.2.2 Model Implementation

Of the pre-trained models used in [7], we chose to use Inception Resnet V2 in our analysis because this model performed very well.
Inception Resnet V2 is a CNN that enhances the Inception family of Convolutional Neural Network architectures by including residual connections between layers. It is a pre-trained model that was trained on over a million images.
We made use of `tensorflow` and `keras` to load the pre-trained model. As per the details of the paper, the final layer was a custom-layer consisting of an Average Pooling 2D layer, a Dense layer with 512 hidden units and the Relu activation function, and an output layer. The Adam optimizer was used, the loss was binary cross-entropy, and the training was run for 5 epochs.
As mentioned above, the training data was too big to fit into the memory. So, the training was done in batches, with each bacth of data corresponding to one block of the training array. The training time was very high (about 20 hours for 5 epochs).
After training, the model was evaluated using the test data prepared earlier.

The pseudo-code used to train the model is presented below (note: `dask` uses lazy execution, so by calling `compute()` we force `dask` to actually execute all the operations that have been defined on the array so far; this is the reason `compute()` can only be called on one block at a time - the entire array does not fit in memory):

Algorithm 5 Training the Model in Batches

```
for epochs := 1...5 do
    /* A generator was used to generate the blocks of data */
    for block of training data do
        run block.compute()
        run model.train_on_batch() to update model
    end
end
```

4 Experimental Results

4.1 Paper 1

4.1.1 Complexity of Implementation

Implementing the model proved straightforward as we leveraged pre-trained models from PyTorch. Additionally, the manageable size of the dataset referenced in [6] facilitated a streamlined training process, avoiding excessive time or memory consumption.

4.1.2 Training Time

The training duration was reasonable, taking approximately 5 minutes to execute Error Level Analysis, extract deep features, and perform classification using SVM and KNN across all three pre-trained models - ResNet18, GoogleNet, and SqueezeNet.

4.1.3 Performance on own dataset

1. **ResNet18**: The model achieved an accuracy of 62.8% with the SVM classifier and 53.8% with the KNN classifier.
2. **GoogleNet**: The model achieved an accuracy of 61.4% with the SVM classifier and 58.2% with the KNN classifier.
3. **SqueezeNet**: The model achieved an accuracy of 56.2% with the SVM classifier and 57.2% with the KNN classifier.

There are a couple of reasons that contribute to the disparity between our model's accuracy and the accuracies reported in the paper:

1. The dataset comprised 2000 images, approximately evenly divided between real and fake images. To facilitate training and testing, we had to partition this dataset into two sets, resulting in a substantially smaller training dataset. This reduction in training data size could potentially contribute to the observed lower accuracy. Besides, the authors have not provided information regarding the ratio used for the train-test split.
2. The hyperparameters employed for normalizing the image tensors during the preprocessing step in the deep feature extraction process have not been specified by the authors.

4.1.4 Performance on different dataset

The model performed better on the other dataset achieving the following accuracies.

1. **ResNet18**: The model achieved an accuracy of 89.1% with the SVM classifier and 63.7% with the KNN classifier.
2. **GoogleNet**: The model achieved an accuracy of 72.5% with the SVM classifier and 64.7% with the KNN classifier.
3. **SqueezeNet**: The model achieved an accuracy of 70.0% with the SVM classifier and 64.5% with the KNN classifier.

The original training dataset comprised approximately 1 million images in just the training dataset. The test dataset consisted of 20,000 images, with an equal distribution of real and fake images. Due to

the substantial size of the dataset and lack of compute resources, we utilized only a subset of images for model training. Specifically, we selected 1000 real and 1000 fake images from both the training and test datasets. This subset was considerably more extensive than the original dataset referenced in the research paper. Therefore, the model demonstrated strong performance when trained on this significant volume of data. This finding seemingly contradicts what [8] reports. But, since the dataset sizes were different, we cannot conclude that this apparent contradiction is accurate. We can, however, conclude that the amount of training data has a substantial influence on the performance of the model.

4.2 Paper 2

4.2.1 Complexity of Implementation

Implementing the model itself was not very complex, especially since it used a pre-trained model as the base with a custom top-layer. However, the extremely large size of the dataset presented a lot of difficulty, and thus most of the complexity was in effectively using the entire dataset with the limited compute and memory resources. This was achieved through efficiently processing the data and feeding it to the model in batches.

4.2.2 Training Time

The training time was very high. A single epoch took close to 6 hours to run (we used a GCE VM with 26 GB RAM, 15 GB GPU RAM, and 300 GB Disk storage). The overall training time for 5 epochs was over 30 hours. This was due to the enormous amount of training data. The authors do not give any information about how they handled the large dataset, nor do they mention what their experimental set-up was.

4.2.3 Performance on own dataset

The model achieved an accuracy of $\approx 81\%$ on the test dataset.
There are a couple of reasons why our accuracy did not exactly match that presented in the paper:

1. We were not able to run 20 epochs (which is what the authors did) due to lack of compute resources and time. As explained above, just getting the model to 5 epochs took well over 15 hours, and this was just running time. Preparing the data, and evaluating the model took a lot of additional time. The GPU RAM limit of 15GB was also a constraint.

2. We were unable to validate the model during training (we were only able to test the model afterwards using the test data). This was because, as explained above, we used the Keras API `train_on_batch`, which does not have a provision for validation data. In addition, we were not able to store both the train and validation data in memory simultaneously. It is unclear how the authors implemented model validation with the data.

In the future, if we continue working on this project, we will find a way to increase the number of epochs and validate the model during training. Once those two steps are implemented, our implementation should match that of the paper.
Observation: While training, we made an interesting observation. Because the dataset is sorted (the first half of the data is labelled 1 (real) and the second half is labelled 0 (fake)), after the first epcoh, the model just labelled all examples as 0 (thus, achieving a 50% accuracy. This was, of course, rectified in the future (by shuffling the data).

4.2.4 Performance on different dataset

As expected, the model performed very poorly on the different dataset, achieving an accuracy of only about 53% when run for 5 epochs. Interestingly, on increasing the number of epochs to 20 (as recommended by the authors), the test accuracy dropped to $\approx 46\%$ (here, the dataset has a total of only 2000 images, so there was no issue in running 20 epochs).
An accuracy of $\approx 53\%$ means that the model is only very slightly better than random guessing. So, the findings in [8] are consistent with our experimental results.

5 Conclusion and Future Scope

The following table summarizes the comparison of [6] and [7]:

Category	Paper 1	Paper 2
Implementation Complexity	Medium	High
Training Time	5 minutes	$\approx$ 30 hours
Performance on Own Dataset	$\approx 63\%$	$\approx 81\%$
Performance on Different Dataset	$\approx 90\%$	$\approx 53\%$

Table 1: Comparison of Papers

From our experiments, the model from paper 1 seems to perform the best. Its accuracy is not excellent on its original dataset, but this is because the dataset is too small. When run on even a small subset of the larger dataset, it outperforms model 2 (caveat: since we were unable to run model 2 for the prescribed number of epochs, this analysis might not be entirely accurate).
If this work is continued (perhaps with more compute power), the second model can be trained for much longer, and maybe even tweaked to improve accuracy.

6 Team Member Contributions

These were our contributions:

- Divya Sathiyamoorthy (ds4221): full implementation of paper 1
- Shree Thavarekere (srt2165): full implementation of paper 2

In addition, we both worked on analysis, cross-dataset evaluation, writing the paper, and fixing bugs in the project overall.

References

[1] Yisroel Mirsky and Wenke Lee. The creation and detection of deepfakes: A survey. 54(1), jan 2021.

[2] Angela Busacca and Melchiorre Alberto Monaca. Deepfake: Creation, purpose, risks. In *Innovations and Economic and Social Changes due to Artificial Intelligence: The State of the Art*, pages 55–68. Springer, 2023.

[3] Sergi D Bray, Shane D Johnson, and Bennett Kleinberg. Testing human ability to detect 'deepfake'images of human faces. *Journal of Cybersecurity*, 9(1):tyad011, 2023.

[4] Md Shohel Rana, Mohammad Nur Nobi, Beddhu Murali, and Andrew H Sung. Deepfake detection: A systematic literature review. *IEEE access*, 10:25494–25513, 2022.

[5] Tao Zhang. Deepfake generation and detection, a survey. *Multimedia Tools and Applications*, 81(5):6259–6276, 2022.

[6] Rimsha Rafique, Rahma Gantassi, Rashid Amin, Jaroslav Frnda, Aida Mustapha, and Asma Hassan Alshehri. Deep fake detection and classification using error-level analysis and deep learning. *Scientific Reports*, 13(1):7422, May 2023.

[7] Wahidul Hasan Abir, Faria Rahman Khanam, Kazi Nabiul Alam, Myriam Hadjouni, Hela Elmannai, Sami Bourouis, Rajesh Dey, and Mohammad Monirujjaman Khan. Detecting deepfake images using deep learning techniques and explainable ai methods. *Intelligent Automation & Soft Computing*, 35(2):2151–2169, 2023.

[8] Leandro A. Passos, Danilo Jodas, Kelton A. P. da Costa, Luis A. Souza Júnior, Douglas Rodrigues, Javier Del Ser, David Camacho, and João Paulo Papa. A review of deep learning-based approaches for deepfake content detection, 2023.

[9] Yonsei University Computational Intelligence and Photography Lab. Real and fake face database. https://www.kaggle.com/datasets/ciplab/real-and-fake-face-detection.

[10] Kaggle. 140k real and fake faces. https://www.kaggle.com/datasets/xhlulu/140k-real-and-fake-faces.