COMP-SCI 5530

Principles of Data Science

Final Project Report

DeepFake Classification System

Sai Teja Uppuluri

Nikhil Kollipara

Srikanth Baddam

Afrid Shaik

02-28-2025

Introduction

DeepFakes are fake images or videos created by artificial intelligence. They look real to the human eye but are completely generated by machines. While this technology can be used for fun or art, it can also be harmful—especially when used to spread fake news or deceive people.

In this project, our team worked on building a system that can look at a facial image and predict whether it is Real or Fake (DeepFake). The goal was to train a machine learning model to make this prediction accurately.

We used a large dataset of face images, divided into Real and Fake categories. Our main model was a Convolutional Neural Network (CNN) that we trained from scratch. We focused on this model and improved its accuracy with proper training and tuning.

Apart from the main model, we also did some experiments. One of those experiments was using MobileNetV2, a lightweight deep learning model. In this test, we froze the main layers of MobileNetV2 and trained only the final layers on our dataset. This helped us understand how transfer learning works, but it did not perform better than our CNN.

We also added a few enhancements to make our system better:

- Data Augmentation: to make the model more general and improve performance on new images
- Grad-CAM: to visualize which parts of the image the model is focusing on while predicting
- Hard Example Mining: to find images that the model got wrong and retrain it on those tricky examples

Finally, we created a simple web application using Flask. This allows users to upload an image and get the prediction with a confidence score, showing whether the image is real or fake.

This report explains each part of the project clearly—from dataset preparation and model training to testing, evaluation, and deployment.

Dataset Overview

To train and test our DeepFake detection model, we used a large dataset of face images. These images were divided into two classes:

- Real – genuine facial images
- Fake – AI-generated (DeepFake) images

We made sure the data was well-organized and balanced so that the model could learn to tell the difference clearly.

Folder Structure

Our dataset is arranged in the following way:

```
Dataset/
├── Test/
│   ├── Real/     # 5413 images
│   └── Fake/     # 5492 images
├── Train/
│   ├── Real/     # 70,001 images
│   └── Fake/     # 70,001 images
├── Validation/
│   ├── Real/     # 19,787 images
│   └── Fake/     # 19,641 images
```

Each image was placed in the correct folder depending on whether it was real or fake. We used the folder names to assign labels: 0 for real and 1 for fake.

Preprocessing Steps

Before training the model, we had to clean and prepare the data. Here's what we did:

- Resized all images to 128x128 pixels (to reduce training time and memory usage)
- Converted images to RGB format (3 channels)
- Normalized pixel values to be between 0 and 1 (better for model performance)
- Included check conditions for file compatibility, image size and dimensions
- Saved the cleaned images and their labels into .npy files (for faster loading during training)

Why Preprocessing Was Important

Since the dataset was large (140,002 training images), reading and processing images every time would slow things down. By preprocessing and saving them ahead of time, we reduced load times and made our training pipeline more efficient.

CNN Model Development

For this project, we built and trained a Convolutional Neural Network (CNN) from scratch as our main model. This model takes face images and learns to classify them as either Real (0) or Fake (1). We used TensorFlow and Keras to define the model, and OpenCV-based custom data generators for loading images during training.

CNN Architecture

Our CNN model includes the following layers:

- Conv2D(32) + ReLU + MaxPooling2D
- Conv2D(64) + ReLU + MaxPooling2D
- Conv2D(128) + ReLU + MaxPooling2D
- Flatten layer to convert feature maps into a 1D vector
- Dense(128) with ReLU for learning complex features
- Dropout(0.5) to avoid overfitting
- Dense(1) with sigmoid to output a binary class (real or fake)

Data Loading

Instead of loading all images into memory, we used a custom class ImageDataGeneratorCV (from data_generator.py). It loads and preprocesses images in real-time using OpenCV.

```
train_gen = ImageDataGeneratorCV('Dataset/Train', batch_size=32)
val_gen = ImageDataGeneratorCV('Dataset/Validation', batch_size=32)
```

This made training faster and memory-efficient, especially since our dataset is large.

Training Details

- Loss Function: Binary Cross-Entropy
- Optimizer: Adam
- Batch Size: 32
- Epochs: 5 (can be increased for better results)

We trained the model like this:

```
history = model.fit(train_gen, validation_data=val_gen, epochs=5)
```

After training, we saved the model using:

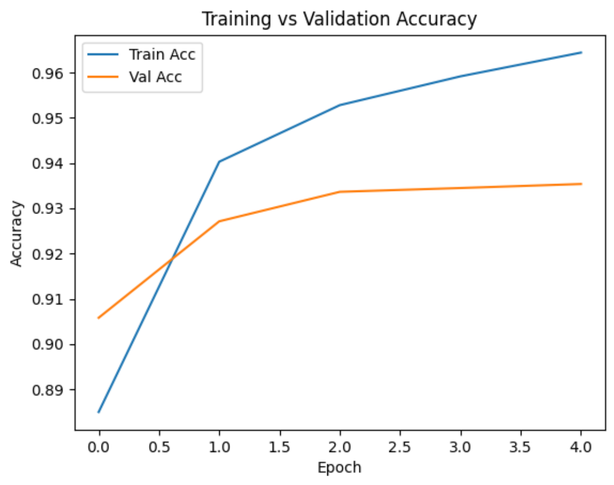
```
model.save('models/cnn_model.h5')
```

Accuracy Plot

To track model performance, we plotted both training and validation accuracy:

plt.plot(history.history['accuracy'], label='Train Acc')

plt.plot(history.history['val_accuracy'], label='Val Acc')



Accuracy Plot

This chart was saved to results/cnn_accuracy.png. From the plot, we noticed:

- Training accuracy improved steadily and reached about 96.7%.
- Validation accuracy also improved and stayed close to 93.5%.
- The gap between the two curves suggests the model is learning well but may slightly overfit if we keep training for too long.

Overall, the model performed strongly on both training and unseen validation data.

Experimental Enhancements

Apart from our main CNN model, we tested a few additional ideas in one combined script to explore if they could improve model performance or training speed. These included transfer learning, augmentation, Grad-CAM, and hard example mining.

MobileNetV2 with Data Augmentation

We used a pre-trained MobileNetV2 model as the feature extractor and froze all of its base layers. On top, we added a custom classifier with a global pooling layer, dense layer, dropout, and sigmoid output. We also applied light augmentation to the training images to make the model more general.

Here is the exact code we used:

```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(128,
128, 3))
base_model.trainable = False

model = models.Sequential([
   base_model,
   layers.GlobalAveragePooling2D(),
   layers.Dense(64, activation='relu'),
   layers.Dropout(0.4),
   layers.Dense(1, activation='sigmoid')
])
```

We compiled the model using:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

We used ImageDataGenerator for rescaling and augmenting the training images:

```
train_aug = ImageDataGenerator(
   rescale=1./255,
   rotation_range=10,
   width_shift_range=0.05,
   height_shift_range=0.05,
   shear_range=0.05,
   zoom_range=0.05,
   horizontal_flip=True
)
```

And we trained the model with:

```
history = model.fit(train_gen, validation_data=val_gen, epochs=5)
```

Result

Even with augmentation, this model did not perform better than our custom CNN. Freezing the base layers meant the model couldn’t fully adapt to our dataset, and training only a small classifier on top limited its learning capacity.

Model Evaluation

After training our CNN model, we evaluated it using a separate test dataset. The goal was to see how well the model performs on unseen images and understand its strengths and weaknesses.

Evaluation Metrics

We used standard classification metrics:

- Precision: how many predicted positives were correct
- Recall: how many actual positives were captured
- F1-score: balance between precision and recall
- Accuracy: overall correct predictions

We ran the evaluation using evaluate.py, where we:

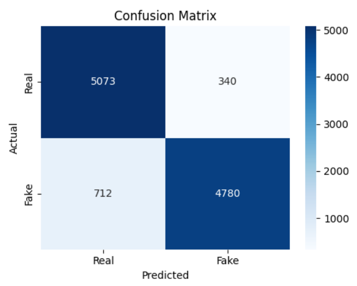
- Loaded the trained CNN model (cnn_model.h5)
- Ran predictions on the test set using our custom ImageDataGeneratorCV
- Plotted the confusion matrix
- Printed classification metrics

Results

Here is the classification report:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Real	0.88	0.94	0.91	5413
Fake	0.93	0.87	0.90	5492
Overall Accuracy	—	—	**0.90**	10905

We also generated this confusion matrix:



Confusion Matrix

This shows:

- The model correctly predicted 5073 real and 4780 fake images.
- It made 340 false positives and 712 false negatives.
- The slightly higher recall for real images suggests the model is more confident in identifying real images than fake ones.

Visual Evaluation

To get a better sense of model behavior, we visualized:

- 5 Correct Predictions
- 5 Incorrect Predictions

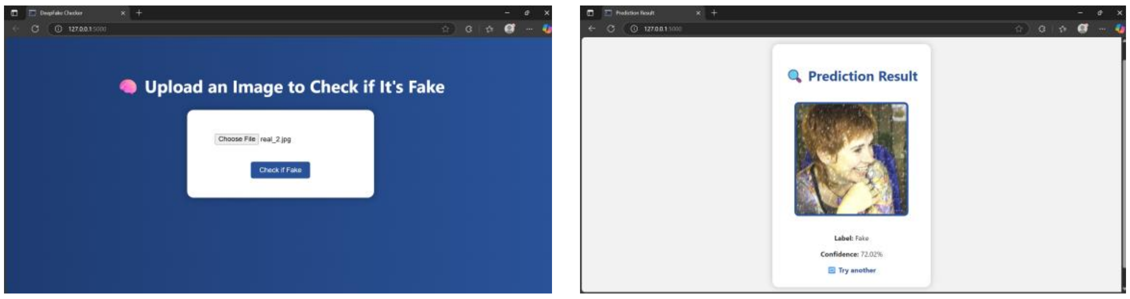
This helped us understand where the model is confident and where it struggles — especially in cases where fake images look highly realistic or real ones appear distorted.

Web Application (Flask)

To make our model easier to use, we built a simple web app using **Flask**. This app lets users upload any face image and get a prediction — telling them whether the image is **Real** or **Fake**, along with the confidence score.

How It Works

- The app has a drag-and-drop interface for uploading an image.
- The image is saved in a temporary folder and processed.
- Our saved CNN model (cnn_model.h5) is loaded in the background.
- The app runs prediction and shows the result instantly on the webpage.



Screenshots of web interface created

This helped turn our project into something interactive and user-friendly. Anyone with basic browser access can test images without running code.

Conclusion

In this project, our team successfully built a DeepFake image classification system using deep learning. We trained a custom Convolutional Neural Network (CNN) from scratch, which turned out to be the most accurate and reliable model among all the approaches we tried.

We used a large dataset of real and fake face images, applied preprocessing for efficiency, and trained the model with proper tuning. The CNN achieved around **90% accuracy** on the test set, with balanced precision and recall for both classes. It was able to clearly distinguish between AI-generated and real images most of the time.

We also experimented with:

- MobileNetV2 (transfer learning) to test model efficiency
- Data Augmentation to reduce overfitting
- Grad-CAM to understand model attention
- Hard Example Mining to focus on tricky cases

While these experiments gave us useful insights, our original CNN still performed best.

To make the system easy to use, we deployed it as a web application using Flask. This allows users to upload an image and get real-time predictions with confidence scores.

Our project code is available at the below link:

Data-Science-Assignments/DeepFake at main · saitejauppuluri/Data-Science-Assignments

Final Thoughts

This project helped us apply everything we learned about data preparation, model building, evaluation, and deployment. It also showed us the real-world challenges of working with DeepFake detection — especially how small details can make a big difference in classification.

We believe this work provides a solid foundation for future improvements, like:

- Using unfrozen transfer learning models
- Training on higher-resolution images
- Adding video support for frame-wise DeepFake detection