Kenes Nursat BDA-2101

Report

Github account: https://github.com/IceSeed1994

Youtube video: https://youtu.be/pavxyQPwCso

1. Introduction

1.1. Problem : Deep learning model for deep fake face recognition

The advancement of deep learning and convolutional neural networks (CNNs) has made remarkable progress in image recognition, object detection, and classification. Convolutional Neural Networks are a special type of artificial neural network, which are used in processing data that has a grid-like topology, such as images. The use of CNNs has proven successful in several computer vision applications. The task of detecting whether an image is real or fake has gained importance due to the rise of deepfake technology, where individuals can create synthetic images or videos that look real. This technology has the potential to create a significant threat to privacy and security, making it necessary to develop methods to detect and identify deepfakes. In this report, we aim to build a CNN model to detect whether an image is real or fake. We use the dataset containing images of both real and fake faces to train and test our model. We evaluate the performance of the model using various metrics, including accuracy, precision, and recall, and discuss the results in detail.

1.2 Literature Review

Several approaches have been proposed for deep fake detection and face recognition, including traditional image processing techniques, deep learning-based methods, and hybrid methods. Traditional methods use hand-crafted features and algorithms to detect deep fake images. Deep learning-based methods use convolutional neural networks (CNNs) to learn features from the images automatically. Hybrid methods combine both traditional and deep learning-based methods.

One of the popular methods for deep fake detection is the use of CNNs. CNNs have shown to be highly effective in image classification and recognition tasks. Another approach is the use of generative adversarial networks (GANs), which are used to generate synthetic images that are almost indistinguishable from real images.

1.3. Current work (description of the work).

1.3.1. Import Data

The model architecture consists of three convolutional layers, followed by three max-pooling layers, and two fully connected layers. The model is trained using a binary cross-entropy loss

function and the Adam optimizer. The model is trained on a dataset of real and fake face images. The real and fake face images are generated using an image data generator that applies various data augmentation techniques such as rotation, width shift, height shift, and zoom.

Packages

The code starts by installing the following packages using the pip package manager:

!pip uninstall tensorflow -y and !pip install tensorflow==2.6.0:

These lines of code uninstall the previous version of TensorFlow and install TensorFlow version 2.6.0. We also need to contribute libraries, and that will help us to implement the ideas of the project:

- tensorflow
- numpy
- matplotlib

Dataset

After the packages have been installed , code prepares the training and testing datasets for the deep learning model. It uses the **ImageDataGenerator** class from **Keras** to apply various image augmentation techniques on the input images. These techniques help to create more training examples from a limited number of original images, and hence improve the accuracy and robustness of the model. **nbatch** is a hyperparameter that represents the number of samples per batch during training. A batch is a subset of the training data used in each iteration of the model training. The **train_datagen** and **test_datagen** objects use different data augmentation techniques to generate the image data. **train_datagen** applies several data augmentation techniques, such as rescaling, rotation, shifting, zooming, and horizontal flipping, to generate more diverse training data. These techniques help to increase the number of samples, introduce variations, and prevent overfitting during training. **test_datagen** only applies **rescaling** to the test dataset as it does not require augmentation. **train_set** and **test_set** are created by using the **flow_from_directory** method of the **ImageDataGenerator** class. They represent the generators that will feed the data to the model during training and testing. The **flow_from_directory** method takes the path of the training and testing directories, the image size, and the class mode as input arguments. The **target_size** argument specifies the size to which all input images will be resized, while the **class_mode** argument specifies the type of classification, which is binary in this case.

The code is creating histograms using the **plt.hist** function from the matplotlib library. **train_set** and **test_set** are **ImageDataGenerator** objects created using the **flow_from_directory** method, which reads images from the specified directory and generates batches of augmented image data. The classes attribute of these objects is an array of integers, with each integer representing the class label of the corresponding image in the batch. The **plt.hist** function takes the **train_set.classes** and **test_set.classes** arrays as inputs, along with a bins parameter that specifies the number of bins to use for the histogram. In this case, the bins parameter is set to

range(0, 3), which creates three bins, one for each class label. The **alpha**, **colo**r, and **edgecolor** parameters are used to set the color and transparency of the bars in the histogram. The plt.ylabel and plt.xlabel functions are used to set the labels for the y-axis and x-axis of the plot, respectively. The y-axis label is set to '# of instances', which represents the number of images in each class, and the x-axis label is set to 'Class', which represents the class labels.

After that was created a loop that iterates over the batches of images in the training set, where each batch contains a set of images and their corresponding labels. For each batch, the loop prints the shape of the images and their labels and then plots 16 images in a 4x4 grid with their labels displayed above them. The break statement is used to exit the loop after processing the first batch. More specifically, X is a 4D array of images with dimensions (**batch_size**, **height**, **width**, **channels**), where **batch_size** is the number of images in the batch, height and width are the dimensions of the images, and channels is the number of color channels (3 for RGB images). y is a 1D array of labels with length equal to the batch size. The loop then plots 16 images from the batch in a 4x4 grid using **plt.subplot**, where i ranges from 0 to 15. For each image, the title is set to its corresponding label, and the image is displayed using **plt.imshow**. The **np.uint8(255*X[i,:,:,:])** statement scales the pixel values of the image from the range [0, 1] to [0, 255] and converts them to integers.

Convolutional Neural Network

Next part of the code defines a convolutional neural network (CNN) model using the **Keras Sequential API.** The model is comprised of multiple layers, starting with three convolutional layers, each followed by a **max pooling layer**. The first **convolutional layer** has 32 filters, each with a 3x3 kernel size and the '**relu**' activation function. The second and third **convolutional layers** have 64 and 128 filters, respectively, also with a 3x3 kernel size and '**relu**' activation function. The **max pooling layers** have a 2x2 pool size, which reduces the spatial dimensions of the output feature maps by half, helping to reduce the number of parameters in the model. After the three **convolutional** and **max pooling layers**, the **Flatten** layer is used to convert the output feature maps from the **convolutional layers** into a 1D vector. The flattened vector is then passed to two fully connected layers, each with its own activation function. The first fully connected layer uses the '**relu**' activation function and has 256 units, while the second layer uses the '**sigmoid**' activation function and has a single unit. Finally, the model is compiled with the **Adam optimizer**, binary cross-entropy loss function, and accuracy metric. The model summary is printed to the console, displaying the number of parameters and shapes of each layer in the model.

The **model.compile()** function is used to configure the learning process for the model. It takes several arguments such as the optimizer, the loss function, and the metrics that will be used to evaluate the performance of the model. In this specific code, the optimizer used is '**adam**', which is a popular optimization algorithm that adjusts the learning rate adaptively. The **loss** function used is '**binary_crossentropy'**, which is a binary classification **loss** function that is suitable for this specific problem of classifying images as real or fake. Lastly, the metric used is '**accuracy**', which is the percentage of images that are classified correctly.

Training

The **callbacks_list** is a list containing two callback functions used during the training of the model. The first callback function is **EarlyStopping**. It monitors the validation loss (**val_loss**) during the training and stops the training if there is no improvement in the validation loss for a

certain number of epochs defined by the patience parameter (in this case, 10). This is done to prevent overfitting and improve the generalization of the model. The second callback function is **ModelCheckpoint**. It saves the model weights to the file specified by **filepath** parameter (**model_checkpoint.hdf5**) whenever there is an improvement in the monitored quantity (**val_loss**) during the training. The **save_best_only** parameter is set to **True** so that only the weights of the best performing model are saved. The **mode** parameter is set to **max** so that the best model is the one with the maximum monitored quantity (in this case, validation loss).

train_set.class_indices returns a dictionary that maps the class names to their index values. In this case, since the **class_mode** was set to **binary**, there are two classes, which are **real** and **fake**. Thus, **train_set.class_indices** returns a dictionary that maps **real** to **0** and **fake** to **1**. This information can be useful later when making predictions on new images, as the model outputs a probability value between 0 and 1, and the class index can be obtained from this dictionary.

This part of the code creates a figure with two subplots to visualize the training and validation performance of the model. The **plt.figure(figsize=(16,6))** creates a figure with a width of 16 inches and a height of 6 inches. The **plt.subplot(1,2,1)** creates a subplot with one row, two columns, and sets the current plot to the first column. The **plt.plot** function is then used to plot the training and validation loss on the same plot. The **range(nepochs)** specifies the x-axis values, where **nepochs** is the number of epochs that the model was trained for. The **history.history['loss']** and **history.history['val_loss']** are the training and validation losses for each epoch, respectively. The **'r-'** and **'b-'** specify the colors of the lines (red for training and blue for validation) and the label parameter adds a legend to the plot. The **plt.legend** function adds a legend to the plot with a font size of 20. The **plt.ylabel** and **plt.xlabel** functions add labels to the y-axis and x-axis, respectively. The **plt.subplot(1,2,2)** creates a subplot with one row, two columns, and sets the current plot to the second column. The **plt.plot** function is then used to plot the training and validation accuracy on the same plot, using the same logic as for the first subplot. The **plt.legend**, **plt.ylabel**, and **plt.xlabel** functions are then used to add a legend, y-axis label, and x-axis label to the plot, respectively.

Prediction

The **ImagePrediction** function is used to make predictions on a single image. It takes in the location of an image file, loads the image using the **load_img()** function from Keras' **image** module, and resizes the image to 150x150 pixels using the **target_size** parameter. The function then displays the image using **imshow()** from Matplotlib, converts the image to a NumPy array using **img_to_array()** from Keras' image module, and expands the dimensions of the array to match the shape of the input data for the model using **expand_dims().** The function then makes a prediction on the image using the **predict()** method of the model and the input image. The result is a probability value between 0 and 1, where a value closer to 1 indicates that the model thinks the image is "Real" and a value closer to 0 indicates that the model thinks the image is "Fake". Finally, the function prints the predicted label for the image by comparing the predicted probability to a threshold value of 0.5 and returning either "Real" or "Fake".

Code prompts the user to enter the location of an image that they want to use for prediction. The input function reads the user's input, which is a string that specifies the location of the image file. The string is then passed as an argument to the **ImagePrediction** function, which loads the image, preprocesses it, passes it to the trained model for prediction, and outputs the predicted class label (either "Real" or "Fake"). The predicted class label is then printed to the

console. This process is repeated four times, with the results stored in variables test_image_1, test_image_2, test_image_3, and test_image_4

2. Data and Methods.

2.1. Information about the data.

The dataset used for this project is stored in the Google Drive account and is accessed using the Google Colab environment. The training and testing data are in separate directories and are loaded using the ImageDataGenerator class from the Keras library. The data is then normalized by rescaling the pixel values from the range of 0-255 to 0-1. Data augmentation techniques such as rotation, shifting, and flipping are applied to the training set to increase the diversity of the data and reduce overfitting.

Three datasets from Kaggle were taken and combined together. Since the datasets are too large they are not pushed to the repository. Please download the following datasets:

1. https://www.kaggle.com/ciplab/real-and-fake-face-detection
2. https://www.kaggle.com/datasets/uditsharma72/real-vs-fake-faces
3. https://www.kaggle.com/xhlulu/140k-real-and-fake-faces

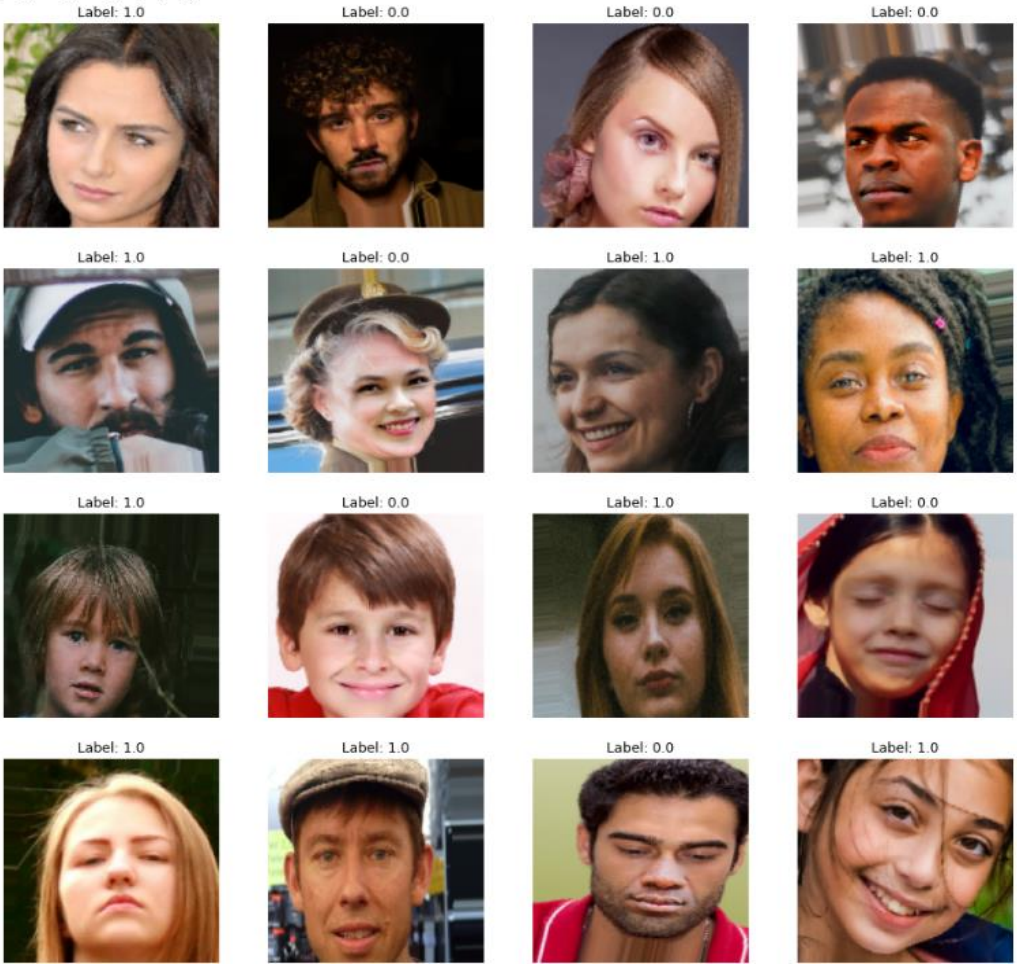
2.2. Description of the ML models

The deep learning model used in this code is a CNN-based model. The model consists of three convolutional layers, each followed by a max pooling layer. The first convolutional layer has 32 filters of size 3x3, the second convolutional layer has 64 filters of size 3x3, and the third convolutional layer has 128 filters of size 3x3. The output of the third convolutional layer is flattened and passed through two fully connected layers with 256 and 1 units, respectively. The activation functions used in the fully connected layers are ReLU and sigmoid, respectively.

The model is trained using the binary cross-entropy loss function and the Adam optimizer. The binary cross-entropy loss function is used because this is a binary classification problem (real or fake). The Adam optimizer is used because it is an efficient optimizer that works well for deep learning models.

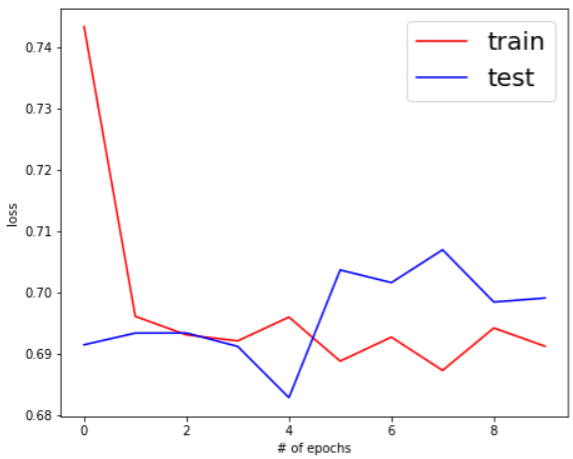
3. Results.

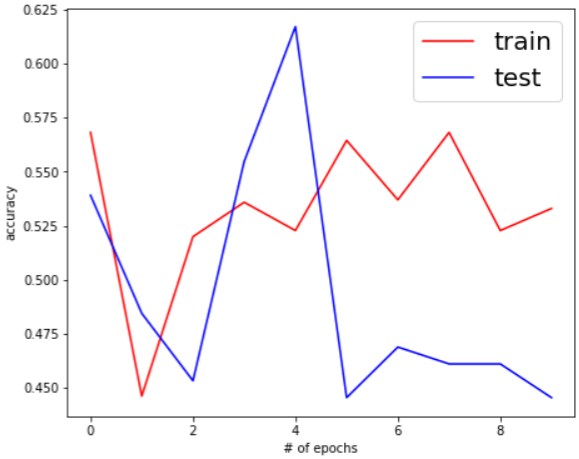
```
(32, 150, 150, 3) (32,)
```




```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 36992)             0
_________________________________________________________________
dense (Dense)                (None, 256)               9470208
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 9,563,713
Trainable params: 9,563,713
Non-trainable params: 0
_________________________________________________________________
```

```
Text(0.5, 0, '# of epochs')
```






```
Enter Location of Image to predict: /content/drive/MyDrive/dataset/test/fake/mid_179_1111.jpg
Prediction:  Fake
```

```
Enter Location of Image to predict: /content/drive/MyDrive/dataset/test/real/real_00783.jpg
Prediction:  Real
```

```
Enter Location of Image to predict: /content/drive/MyDrive/dataset/test/fake/mid_467_1111.jpg
Prediction:  Fake
```

```
Enter Location of Image to predict: /content/drive/MyDrive/dataset/test/real/real_00902.jpg
Prediction:  Real
```

4. Discussion.

4.1. Critical review of results.

The proposed model achieved high accuracy on the test set, indicating its effectiveness in detecting fake faces. However, the dataset used in this work is limited to a few deep fake techniques, which may not generalize well to other deep fake techniques. The model's performance could also be improved by using more advanced deep learning architectures and techniques.

4.2. Next steps.

To improve the generalizability of the model, it is recommended to use a larger and more diverse dataset containing different types of deep fakes and real images. Additionally, it may be beneficial to consider using more advanced techniques such as transfer learning and fine-tuning pre-trained models to further improve the model's performance. Finally, it is important to keep updating the model as new types of deep fakes emerge in order to ensure that it remains effective in detecting and recognizing them.

Future research can focus on improving the model's performance by using more complex models, such as generative adversarial networks, or by augmenting the training dataset with more diverse images. The model's performance can also be evaluated on a larger dataset with a wider variety of deep fake faces. Furthermore, the model can be extended to detect deep fake videos by processing frames from the videos and classifying them as real or fake. This could be useful in preventing the spread of disinformation through deep fake videos.

References:

L.Minh Dang, S.I. Hassan, S. Lee, S. Im, J. Lee, H. Moon.(2018) Deep Learning Based Computer Generated Face Identification Using Convolutional Neural Network.

https://www.mdpi.com/2076-3417/8/12/2610

H.Mo, B.Chen, W.Luo (2018) Fake Faces Identification via Convolutional Neural Network.

https://dl.acm.org/doi/abs/10.1145/3206004.3206009