|  |
| --- |
| Newcastle University | NCCPEDevelopment and Optimisation of Convolutional Neural Networks for Real-Time Individual Facial Recognition using Artificial Intelligence |
| *A hybrid approach combining machine learning and statistical techniques to build a real time facial detection system using Python.*  Student name: Idraq Siddiqui, Student number: 180637694, School of Engineering  Degree programme: BEng Mechanical Engineering (Hons)  A dissertation submitted in partial fulfilment of the requirements for the degree of Bachelor of Engineering with Honours in Mechanical Engineering  Submission Date: 30/09/2024 |

|  |  |  |  |
| --- | --- | --- | --- |
| ***Heading*** | ***Subheading*** | ***Contents*** | ***Page Number*** |
| **Introduction** |  |  | 1 |
| **Aims and Objectives** |  |  | 1-2 |
| **Background and Literature Review** |  |  | 2- |
|  | Machine Learning |  | 3 |
|  | Artificial Neural Networks |  | 4 |
|  |  | *Activation Functions* | 4-5 |
|  |  | *Training Process: Loss (Cost) Function, Backpropagation and Gradient based learning* | 5-6 |
|  | Convolutional Neural Networks | *Tensors, Convolution, Max Pooling* | 6-8 |
| **Methodology and Implementation** | Data Preprocessing | *Image Processing* | 9-10 |
|  |  | *Model Configuration* | 11-12 |
|  | Model Training |  |  |
|  |  | *K fold Cross validation* | 13-14 |
|  |  | *Testing and Live Feed Face detection* | 14 |
| **Results and Discussion** | Results |  |  |
|  |  | *Training Results* | 15 |
|  |  | *Testing Results* | 16 |
|  |  | *Live Face Detection (Model Implementation) Results* | 17-18 |
|  | Discussion |  | 19 |
| **Conclusion** | Improvements | *Dataset Improvements* | 20 |
|  |  | *Cross Validation Improvements* | 20 |
|  |  | *Model Ensembling Techniques* | 20 |
|  | Final Thoughts |  | 21 |

**Table of Contents**

Appendix Scripts 24

**Abstract**

This dissertation will focus on the development and optimisation of Convolutional Neural Networks using custom datasets. We will aim to development the goal of understand the limitations of small datasets and non-commercial computational power for real time face detection. A blend of Machine Learning and Deep Learning techniques are combines with statistical analysis to complete this. The code for this is included in the folder directory, the scripts mentioned directly in this dissertation are also included in the appendix.

Appendix Scripts: https://github.com/IdraqS/FaceDetection2/tree/main

1. Introduction:

AI has garnered an incredible amount of attention in recent years. Despite its recent furore, the term ‘Artificial Intelligence’ was first coined by John McCarthy (1956) [1]. The Turing Test (1950) proposed an ‘intelligent’ computer should be able to give written responses that are indistinguishable from a human being’s via 4 methods [2]:

1. Natural Language processing: Successful understanding of human language.
2. Knowledge Representation: Storing what it knows and hears
3. Automated Reasoning: Using this stored information to develop answers and new conclusions
4. Machine Learning: to adapt to new circumstances

And to do some of these, the computer will require:

1. Computer Vision: to ‘see’ the objects in its environment
2. Robotics: to manipulate the objects in its environment

You may have heard of some of these terms already because, 74 years later, each of these have become fields under the AI umbrella, each branching off into their own subfield and sub-subfields. ChatGPT uses Large Language Models (LLMs), a form of Natural Language Processing. We will be using OpenCV - a Python Library which uses Computer Vision. We will create Convolutional Neural Networks (CNNs), a subfield of Deep Learning (DL) via Machine Learning (ML).

1. Aims and Objectives:

The aim of this study is to explore the adaptability and performance of CNNs for real-time face detection, focusing specifically on small, custom datasets and hardware with limited computational power. Custom CNN model architectures will be developed, trained, optimised and trained for facial recognition of the writer's face. Please note, a CNN is a type of model amongst many. A model is a mathematical representation of the real world – and a CNN is a subset of ML models specifically.

Dataset size is crucial for CNNs. A model, just like a human being, will typically become wiser (*more accurate)* with more experience (*more data*). Models for this task in commercial use are trained on datasets of hundreds of thousands if not millions of images, which comes at significant computational cost [3]. Our custom dataset will be trained on a dataset in the order of 1000s to investigate the effect of limited dataset sizes on model performance and how we can mitigate this effect.

With a small dataset the variability of lighting conditions, angles and facial expressions is low. This increases the risk of what is known as overfitting. Overfitting occurs when a model learns the training set very well but fails to perform well on new, unseen data. Any given dataset can include noise, anomalies or specific pattern – when overfitting occurs it is because the model has learned this well in conjunction with the useful data [4]. Preventing overfitting is key to the success of this study so we must maximise performance from a low variability dataset. It is particularly important to balance accuracy, computational cost and risk of overfitting.

1. Background and Literature Review
2. **Machine Learning (ML):**

|  |
| --- |
| Mathematics 10 02552 g003 |
| **Figure 1. The Artificial Intelligence tree and its branches. Source: [5]** |

In this study we will use CNNs for binary classification tasks and implement the model created using machine vision techniques. Thus, we will combine Deep Learning (DL) techniques with Vision techniques. Our primary focus, however, is CNN technology.

|  |
| --- |
| A diagram of a machine learning  Description automatically generated |
| **Figure 2 Source 6** |

In a typical scenario, to automate tasks you create a program by defining rules and input for the program to deliver the output(s). In ML, however, you train the model on the data and give it an expected output. An ML model will learn its own rules thereby creating its own road from data to desired output.

DL is a subfield of ML using artificial neural networks (ANNs) to create a model of greater *depth* in number of layers of the model. The layers of the model are like steps of the model – a clustering of nodes in each step where each clustering performs its calculations. Deeper models allow us to learn larger, complex datasets to tackle more complex tasks.

1. **Artificial Neural Networks and how they work**

The neural networks (NN) described so far are distinct from ‘real’ neural networks; biological neural networks, like those of the brain are the source of inspiration for ANNs.

NNs contain an input layer, hidden layer(s) and output layer as per figure 3. Data flows forward from input to output. The input layer contains data in the form of numerical values. The hidden layer(s) is where the learning occurs and are typically much more densely populated than the simple figure shows.

|  |
| --- |
| A diagram of a network structure  Description automatically generated |
| **Figure 3. Simple NN structure. Source: [7]** |

Finally, the output layer also contains numerical values. Data is inputted as tensors, which are multi-dimensional arrays of greater dimensionality capability. However, tensors are broken into scalar values when computing at each neuron in a simple NN.

|  |
| --- |
| A diagram of a function  Description automatically generated |
| **Figure 4. Perceptron. Source: [2]** |

The nodes in figure 3 are all artificial neurons, called perceptrons, represented in figure 4. Each neuron of a neural network will have two main operations : weighted sum and activation, such that:

|  |  |
| --- | --- |
| *Weighted (input) sum between neurons i,j :* | *Activation Function:* |
| **Figure 5. a = inputs, w = weights. Bias =** 𝒘𝟎,𝒋∗𝒂𝒐 | |

Biases are initialised as a non-zero integer to allow the neuron to have a non-zero output even with zero input – increasing neuron availability. Inputs from all previous perceptrons with their associated weights and biases are summated and passed through the activation function. Larger weights provide greater neuron inputs; therefore, a weight is essentially the measure of the strength between neurons. This effect on the *layer* is described as the weights parameterising the layer [6]. The model adjusts weights to find a set of weight values for all the layers in each network such that inputs are mapped to their intended output [6] to minimise the differences between the predicted and actual values. This difference is known as the loss.

1. **Activation Functions**

The weighted sum operation is a linear one. With standard linear activation function, the output of a neuron is a linear function: . Consequently, the model learns linear relationships between inputs and the outputs. This works when the input-output relationship is actually linear, such as in a linear regression task. However, this is often not the case.

By applying nonlinear activation functions to neurons (and consequently, the layer) we transform the linear input to a nonlinear output, so the model learns nonlinear relationships between the inputs and the outputs. The most common nonlinear activation functions are Tanh, Sigmoid and ReLU. Sigmoid activation and tanh functions will squeeze all inputs to values between 0,1 and -1,1 respectively, restricting their output values, causing saturation of inputs at 0,-1 or 1.

|  |  |
| --- | --- |
|  |  |
| Tanh activation Function | Sigmoid Activation Function |

The gradients (derivative) of the activation functions are very important to help the model during backpropagation:

|  |  |
| --- | --- |
|  |  |

As the values of the input (z) tend away from 0 the derivatives of the sigmoid and tanh functions become smaller and smaller – hence vanishing. To solve this, we use a linear function called the Rectified Linear Unit (ReLU).

ReLU is now the standard as outputs are unrestricted, and its gradient is constant, so it doesn’t vanish. It provides far less saturation and greater sensitivity.

1. **Training Process: Loss (Cost) Function, Backpropagation and Gradient based learning**

To train an NN, there are two passes, forward and backwards. In the forward pass, a model will pass data from its input to output layer via hidden layers to generate an output prediction, which is used to calculate the loss described previously - essentially a measure of how wrong a prediction is. After random initialisation, the model uses loss function, backpropagation (backprop) and gradient based learning to update its weights and biases. The goal of the model is the find the optimal set of weights and biases to minimise loss. Neural pathways leading to the smallest loss are given greater weights making those connections stronger. This is how the model ‘maps’ its pathway from input to the *best* output. Losses for classification tasks commonly use cross entropy loss:

|  |  |  |
| --- | --- | --- |
| *Sparse Categorical Cross Entropy Loss function:* | *Categorical Cross Entropy Loss function:* | *Binary Cross Entropy Loss function:* |
| = predicted probability for true class, ytrue actual class label | C = Number of classes, yi =actual label for class i, = predicted probability for class i | y = actual label, predicted probability = |

Losses are calculated at each layer until it reaches the output layer where now the model will work backwards using backprop, which calculates the error (δL) of a layer by multiplying the loss function derivative by the derivative of the activation function of the previous layer. In a binary classification problem using ReLU:

|  |  |
| --- | --- |
|  | Where L = loss function no, Y = predicted, z = weighted sum of inputs |

The gradient descent algorithm uses this error function. Output Y is a function of Weights (W) and biases(b), therefore:

|  |  |
| --- | --- |
| Change in weights (W) using derivative of loss function with respect to weight | Change in bias(b) using derivative of loss function with respect to bias |

Where is the learning rate of the model. This is called gradient descent-based learning, where the weights and biases of each layer in the NN are updated via this gradient descent algorithm.

1. **Convolutional Neural Networks (CNNs)**

A rank-0 tensor is just a scalar number without axes [10]. A rank-1 (1D) tensor is a list of values of singular axis: ([1,2,3]). Rank 2,3,4 tensors are shown in figure 9:

|  |  |
| --- | --- |
|  |  |
| ***Rank 2 (2D)*** *Tensor of shape [3,2]* | ***Rank 3 (3D****) Tensors of shape [3,2,5]. Essentially 3 matrices of [2,5] shape* |
| ***Figure 9 Source [10]*** | |
|  |  |

|  |
| --- |
| ***A diagram of three different colored cubes  Description automatically generatedRank 4 (4D)*** *Tensors of shape [3,4,2,5]. 3 3D matrices of shape [4,2,5]* |
| **Figure 10. Source [10].** |

All NNs process their data in the different shapes of tensors. An image, be represented as a 3D tensor. Multiple images at a time are processed in batches, thus creating a 4D tensor of shape as per figure 9 where each 3D block is an image. Each individual block is an element representing a pixel value.

In normal, fully connected NNs individual weights are stored in 2D tensor [12] where the number of neurons determine the tensor dimensions. 4 neurons in the previous and 3 neurons in the current layer would be a 4x3 tensor where elements are weights between neurons i,j. This is flattened into a 1D tensor (list) : ([1,2,3…n]) where each element is given an individual neuron. Even a small dataset would contain 1000s of elements, so this becomes impractical. The use of CNNs allow us to maintain the dataset dimensionality.

CNNs use convolution instead of a general matrix multiplication [4], combining the input function and the kernel function, producing an output function or *feature map*. This is helpful for processing data that is grid-like – like in like an image tensor. Thus, if we apply convolution to combine a 2D image input I with a 2D kernel K[4]:

|  |  |
| --- | --- |
|  | S = output as function of row, column(i,j) of feature map, m,n = input row, column |

|  |
| --- |
|  |
| **Figure 10 Source 12, Convolution in CNN** |

The kernel of size (3,3) slides over the tensor and multiplies the weights within it by the corresponding values of the tensor and adds its calculated values. This is a dot product calculation, thus for every 9 elements there is now one associated value. The kernel continues sliding and calculating over the entire tensor resulting in a feature map of values of weighted sums associated to local regions in the data. This drastically reduces the number of connections and therefore, total computational cost. It also allows the model to learn localised patterns in the data. In an image – textures, corners or edges etc.

|  |
| --- |
| A diagram of a function  Description automatically generated |
| **Figure 11, Pooling Layer Calculation. Source [12]** |

Dimensionality is further reduced by the pooling layer of a CNN. The pooling function slides over the feature map, typically in 2x2 windows and take either the maximum value or the average value for each window of the feature map calculated by convolution. In figure 11, max pooling occurs at the top and average pooling at the bottom. This further reduces the computational cost by reducing the spatial dimension yet still it keeps the most prominent features. This also prevents overfitting by keeping the most prominent features – thereby reducing noisy data (inconsistencies).

1. Methodology and Implementation
2. **Data Preprocessing:**

Scripts are written in Python 3.12.5 with libraries OpenCV, NumPy, Pandas, Matplotlib and TensorFlow. Our model(s) will be constructed using Keras APIs and they will also be cross-validated using the Scikit-learn library. All files are in the project directory.

The task at hand is to create a CNN model to recognise the writer’s face in real time. Since it needs only to learn one face it only needs to distinguish between ‘Idraq’ and ‘Not Idraq’ - two states. In other words, its goal is **bi**nary. Hence, we use Binary Classification; training our model on datasets of images assigned binary labels ‘1’ for ‘Idraq’ and ‘0’ for ‘Not Idraq’. Once we have our trained model, this is used in real time to produce a predicted value between 0 and 1. The closer it is to 1 the surer the model is the image fed into it is of ‘Idraq’.

Given an objective of ours is to understand the effect of smaller datasets and lesser computational power on model performances, our dataset must be in the order of 1000s.

For our ‘Idraq’ dataset **take\_pictures.py** is written to take 5000 (approx.) images to create this dataset. This dataset contained some variety in a different facial expression, profile views, tilts and lighting conditions, but it cannot be varied too much as the dataset is small so learning my face in a frontal view takes absolute precedence and must take most of the dataset.

‘Not Idraq’ dataset is partially collected from Kaggle[13]. This dataset contains 20,200 images of celebrities, the variety of which will help the model learn a greater variety of features. The drawback is these images are collected ‘from the wild’ and contain random face expressions, angles, tilts and lighting conditions. In addition, different cameras (that took the celeb images) have different aperture, lenses, focal lengths…etc. All of these take the images that form our dataset, so they dictate its quality. We can mitigate this in the next step of our methodology, but we can also help mitigate this now. In addition to the Kaggle images, 2500 images of were taken of Person 2 (for future reference) in the same conditions as the ‘Idraq’ dataset (as per figure 11). Now we have introduced images taken by the same camera in the same conditions of two different people. The model’s ability to distinguish between ‘Idraq and ‘Not Idraq’ will improve while still maintaining variance in both datasets.

|  |
| --- |
|  |
| A screenshot of a math problem  Description automatically generated with medium confidence |
| **Figure 11. Source me and Source 10** |

1. **Image Preprocessing**

Images can be represented as an 3D array as shown in Figure 11 of [pixel height, pixel width, colour channels]. Initial images collected are therefore of shape [1080, 1920, 3] where the 3 colour channels are RGB. There are 3 1080x1920 arrays in which the elements are red, green and blue colour intensity values associated to that given pixel. For example, pixel (284,1024) may have a red value of 58 stored in channel 1, a green value of 249 in channel 2 and a blue value of 163 in channel 3.

These arrays are far too large to be useful for model training so we must reduce their size. **batch\_img\_cropping.py** is created for this.

To identify faces in all the images, we use OpenCV's pre-built Haar Cascades, which implements the Viola-Jones algorithm. Input images are converted to grayscale. The Haar Cascade detector creates rectangles around detected faces, which is then used to crop the images. We optimised the scaleFactor and minNeighbors arguments to maximize the number of correctly identified faces while minimizing false positives (detailed in 'Haar\_Optimiser.xlsx'). Applied to both datasets, our total processed dataset contains 9173 image files. As shown in Figure 12, each image tensor is a 75x75 array of grayscale values ranging from 0 to 255. Images are now of size [75,75,1] and will be the input shape of the tensors going into the model. This reduces the total number of pixel values in each image from 6,220,800 to 5625, reducing computational cost significantly.

|  |  |
| --- | --- |
|  |  |
| **Figure 12** | |

1. **Model Configuration**

To create our model, we will use Keras APIs that to construct CNNs by stacking its layers in sequential manner. If the model is too deep, it will learn patterns in the dataset too well such that it will not generalise well to new data [15] – consequently it may recognise any face as ‘Idraq’. So we will look out for this.

We write **model\_script.py** to produce the CNN shown in Figure 13. The input shape is [75,75,1]. The first convolution layer (Conv2D) contains 128 convolution filters, producing 128 distinct feature maps that learn their own features from the input data such as vertical edges, horizontal edges, textures, etc. The data continues through subsequent layers, reducing in dimensionality until the multidimensional tensor gets flattened by the flattening layer (not typically shown in CNN diagrams) resulting in a final outputting of a 1D tensor. This tensor passes into 128 neurons of a fully connected layer (like figure 3) before being processed all the way down to a singular value – the prediction value.

|  |
| --- |
|  |
| **Figure 13. Model CNN Configuration** |

We use ReLU activation functions, convolutional layer kernel sizes (3,3), pooling layer kernel sizes (2,2) at each layer. Between the 1st and 2nd fully connected layers, a dropout layer drops a random 50% of the neurons to mitigate overfitting risk. Finally, at the last output layer we use a ‘sigmoid’ activation function for binary classification tasks.

At the end of the script, we define the optimiser function ‘adam’ and the loss function ‘binary cross entropy’. ‘adam’ (adaptive moment estimation) is a gradient descent algorithm as mentioned in background section, with learning rate, . A learning rate determines how quickly a model learns by changing how often the weights and biases update. Adam uses an adaptive learning rate, leading to a faster convergence towards an optimal set of weights and a more stable training [4].

1. **Model Training:**

Now we train our model on the dataset which is in two: ‘Idraq’ and ‘Not Idraq’ assigned labels ‘1’ and ‘0’ respectively. A dataset is typically split into 80% for training and 20 % for validation [16]. In our total dataset of 9173 images 7333 are used for training and 1840 for validation. We split our images into batches of 32 to reduce computation cost. Our model is trained through 10 epochs. An epoch is a single run through the dataset. Weights and biases are updated after each batch until the end of the epoch. At the end of each of our epochs 4 values are given: (training) accuracy, (training) loss, validation accuracy, validation loss.

During validation, the validation dataset images are passed through this now trained model to gather predictions for each image. The correct value for an image in the ‘Idraq’ dataset is 1; 0 for ‘Not Idraq’. If the prediction value meets the 0.5 threshold for binary classification tasks it is considered correct. Accuracy is the average percentage of the correct predicted values within a batch and losses over the batch are averaged. Note no weights and biases are updated in validation.

|  |
| --- |
| A collage of images of a person's face  Description automatically generated |
| **Figure 14** |

**data\_preparation.py** was written using Image Data Generator from TensorFlow [17], a form of data augmentation that applies random transformations such as flips, tilts and rotations to the images to generate more images from our images. We essentially create more data from our limited dataset, allowing the feature maps to become more diverse which in may allow the model to recognise a wider variety of face angles or tilts. This script also assigns the label ‘1’ to the ‘Idraq’ dataset and ‘0’ to the ‘Not Idraq’ dataset. Finally, it normalizes all pixel values in every image to a range between 0 and 1 (from the original range of 0 to 255). This normalization smooths the gradient descent learning curve, facilitating faster convergence to the optimal sets of weights and biases[6].

1. **K Fold Cross validation**

If we split our dataset 80:20, the first 7338 images would be used for training and last 1835 used for training. Let’s say we train our model 5 times to reduce anomalies, with 10 epochs it is the same 7338 images to train on 50 times and the same 1835 images validated on 50 times.

This introduces overfitting risk by training and validating on the same sets repeatedly because don’t know how well the model generalises to unseen data if we keep using the same set to validate. We must introduce variance to these datasets – this is how we mitigate drawback of smaller dataset.

A commonly used method for this is called K Fold Cross Validation, a statistical technique implemented by the Scikit-learn library. With each training iteration will ‘move’ the validation dataset for us. In our script we use *Stratified* K Fold for binary classification tasks.

|  |  |
| --- | --- |
|  | **Figure 15 Source 18** |

Figure 15 shows how we split our data; validation (test) sets split into *k* folds where *k* = 5. Value *k* is chosen as 5 to match the 80:20 split. Where *k =* 1 the first 1835 images are used for validation. This is executed in **data\_process\_cross\_validation.py** which uses the image data generators described previously thus providing more variance within the dataset via 2 methods.

1. **Testing and Live Feed Face Detection**

Once we have our trained models, we must test them on unseen data once more to see how they generalise on a new dataset before implementation into the live feed. To do this we write **testing\_script.py.** We generate predictions using Keras model.predict() function, which we use to calculate our metrics: Accuracy, Precision, Recall, F1 Score, AUC. Each metric provides different insights which can be found in the Scikit-learn library tools for model evaluation [19]. These calculations are over the entire dataset, not batches as per model training.

Our best model is implemented in real time using **live\_face\_detection.py**. [75,75,1] images are passed through our model, frame by frame to generate prediction values displayed atop the face. A greater threshold is chosen to balance the risk of miss some true positives, for the reward of reducing false positives. We don’t want the model to overpredict ‘Idraq’ which it may do as our ‘Not Idraq’ dataset contains images of celebrities which have been collected in a variety of conditions, whereas the ‘Idraq’ dataset more *consistent* conditions. Consequently, the model may learn more *consistent* patterns within the ‘Idraq’ dataset – biasing its prediction toward that. This risk is offset by a greater threshold.

1. Results and Discussion
2. **Results:**
3. **Training Results**

Upon Training our model using K fold cross validation we compute the following results:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Model 1 K fold Results | Training accuracy | Training Loss | Validation accuracy | Validation loss | | 1 | 0.9581 | 0.1327 | 0.9696 | 0.0997 | | 2 | 0.9631 | 0.1107 | 0.9726 | 0.0755 | | 3 | 0.9577 | 0.1180 | 0.9695 | 0.0790 | | 4 | 0.9682 | 0.1028 | 0.9533 | 0.1157 | | 5 | 0.9761 | 0.0779 | 0.9807 | 0.0718 | | AVG | 0.9646 | 0.1084 | 0.9691 | 0.0883 | |
| **Figure 16 Model 1 K fold results** |

Our first model has performed excellently, producing a high number of correct predictions and has optimised its weights and biases to minimise losses. There is little discrepancy between training and validation accuracies indicating there is little to no overfitting – the model has learned the data well. High validation accuracy also indicates the model generalises very well.

|  |
| --- |
|  |
| **Figure 17 All Model Configurations** |

The number of filters within each hidden layer stays for constant for constant computational cost. The difference is in the activation functions used : Leaky ReLU, ELU, GELU, SELU and Swish. Each new model configuration is trained once to see how they perform during training and validation. Models 1, 2 and 11 were the most effective models; all used a form of ReLU. Model 6 was the only exception, perhaps because the alpha (a) used was too small.

Based on training results, the 3 best model configurations are then evaluated through K fold Cross validation. Models 7,8,9 and 10 showed plots like figure 18 and 19. They performed well in training but did not validate well - characteristic of overfitting.

1. **Testing**

We now have 20 trained models, 15 from cross validation. We test this on an unseen testing dataset of 501 ‘Not Idraq’ and 483 ‘Idraq’ images of shape [75,75,1], using **testing\_script.py**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Model | Accuracy | Precision | Recall | F1 | AUC | | model\_1\_fold\_1 | 0.9583 | 0.9300 | 0.9896 | 0.9589 | 0.9955 | | model\_1\_fold\_2 | 0.9654 | 0.9446 | 0.9876 | 0.9656 | 0.9977 | | model\_1\_fold\_3 | 0.9685 | 0.9708 | 0.9648 | 0.9678 | 0.9951 | | model\_1\_fold\_4 | 0.9644 | 0.9341 | 0.9979 | 0.9650 | 0.9987 | | model\_1\_fold\_5 | 0.9278 | 0.8759 | 0.9938 | 0.9311 | 0.9946 | | model\_2\_fold\_1 | 0.9533 | 0.9210 | 0.9896 | 0.9541 | 0.9956 | | model\_2\_fold\_2 | 0.9715 | 0.9633 | 0.9793 | 0.9713 | 0.9942 | | model\_2\_fold\_3 | 0.9807 | 0.9633 | 0.9752 | 0.9802 | 0.9984 | | model\_2\_fold\_4 | 0.9776 | 0.9675 | 0.9979 | 0.7461 | 0.9843 | | model\_2\_fold\_5 | 0.9807 | 0.9696 | 0.9917 | 0.9806 | 0.9987 | | model\_11\_fold\_1 | 0.9431 | 0.9194 | 0.9689 | 0.9435 | 0.9869 | | model\_11\_fold\_2 | 0.9553 | 0.9295 | 0.9834 | 0.9557 | 0.9950 | | model\_11\_fold\_3 | 0.9411 | 0.9032 | 0.9032 | 0.9426 | 0.9921 | | model\_11\_fold\_4 | 0.9837 | 0.9834 | 0.9834 | 0.9834 | 0.9975 | | model\_11\_fold\_5 | 0.9024 | 0.9143 | 0.8841 | 0.8989 | 0.9491 | | model\_6 | 0.8252 | 0.7772 | 0.9027 | 0.8352 | 0.9186 | | model\_7 | 0.7500 | 0.6720 | 0.9586 | 0.7901 | 0.9636 | | model\_8 | 0.7043 | 0.6500 | 0.9938 | 0.8613 | 0.7683 | | model\_9 | 0.8587 | 0.7915 | 0.9669 | 0.8705 | 0.9723 | | model\_10 | 0.7978 | 0.7178 | 0.9689 | 0.8247 | 0.9621 | |
| **Figure 20 – Testing results** |

Models 1, 2 and 11 again perform very well – suggesting they generalise well to unseen data. Model 7 and 8 provide the worst performance metrics. Model 6, 9 and 10 while better are still relatively poor. Thus, the relevant activation functions were not suited to this task and none of these models can be used for a live face detection. Producing an average for the best models:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Model | Accuracy | Precision | Recall | F1 | AUC | | 1 | 0.9569 | 0.9311 | 0.9867 | 0.9577 | 0.9963 | | 2 | 0.9728 | 0.9435 | 0.9567 | 0.9356 | 0.9892 | | 11 | 0.9451 | 0.9300 | 0.9446 | 0.9448 | 0.9841 | |
| **Figure 21 – Average Testing results for Model 1, 2 and 11** |

Model 2 returns the highest testing accuracy, suggesting it has generalised best to the unseen data. It also produces the best precision suggesting the model produces the fewest false *positives*. The highest recall comes from Model 1 which means it produces the fewest false *negatives*. Model 1 has the best F1 score, combines the previous two metrics and AUC score is highest for model 1 also. This suggests we should use model 1 despite it not having the highest accuracy to strike the best balancing between producing false negatives and positives.

1. **Live Face Detection (Model implementation)**

Upon implementing model 1 into our **live\_face\_detection.py** script we ran a somewhat annoying issue : the model predictions kept flickering. This flickering between ‘Idraq’ and ‘Not Idraq’ can be caused by sudden movements, shifts in focus or lighting. Since *every* frame is passed through the model each frame has its own prediction, explaining why our script keeps flickering between predictions.

Flickering is also caused by the Haar Cascades classifier was also identifying false positives in the frame. We mitigate the flickering effect by optimising Haar Classifier arguments to identify fewer faces. We also take mean average predictions of the last 5 frames and return this average prediction. This worked well to reduce flickering and produce the results shown in figures.

To test predictions between ‘Idraq’ and ‘Not Idraq’, we train in two settings on two different people. The first is in the same conditions we trained our data (webcam, lighting) and the second is with another person with different conditions.

|  |  |
| --- | --- |
| A person with a beard  Description automatically generated | A person with a beard  Description automatically generated |
| **Figure 23 – Prediction 100%** | **Figure 24 – Prediction 100%** |
| A person taking a selfie  Description automatically generated | A person with a beard  Description automatically generated |
| **Figure 25 – Prediction 99.9%** | **Figure 26 – Prediction 0%** |

The model performs well in real time using our live feed. Figures 23 and 24 are expected given these frames are very similar to the dataset’s images. There is a bit of lens flare in the frame of figure 25 – which would result in the image tensor inputted into the model having greater values than what it was trained on, yet still the model adapted and produced a correct prediction. This proves the model has learned the underlying patterns in the data well enough to be able to produce correct predictions even when conditions have deviated from ideal.

Figure 26 presented a challenge it could not overcome. The further away the person is from the camera, the fewer pixels are required to display their face thereby increasing pixel density (in each area) and reducing the resolution of the face decrease. This makes it harder for the Haar classifier to find identifiable features to find a face. Also, the head is tilted. The dataset contains mostly full front facing images as per Figures 23,24 so this, again, is not a surprise.

|  |  |
| --- | --- |
|  |  |
| **Figure 26 – Prediction 97.90% Not Idraq** | **Figure 27 – Prediction 75.31% Not Idraq** |
|  |  |
| **Figure 28 – Prediction 22.12% Not Idraq** | **Figure 29 – Prediction 42.57% Not Idraq** |

Again, the model correctly identifies people who are ‘not Idraq’ as such and produces a very high prediction score, to boot. This **solidifies** the models ability to distinguish between faces. Our model is robust enough to be able to distinguish between datasets and produce correct prediction values with consistency across a variety of conditions.

We did run into some limitations, however. In both ‘Idraq’ and ‘Not Idraq’ the live feed failed to identify faces when lighting conditions were poor. The Haar classifier is known to be sensitive to changes in light [20]. Additionally, despite our efforts to mitigate them the flickering did still occur. However – overall the model we produces has adapted well and performed its task.

1. **Discussion:**

To discuss these results, it’s important to touch upon our initial objectives:

*‘Explore limitations of a small dataset (size, variability) and limited computational power on CNN model performance* ***and*** *produce robust CNNs that balance overfitting risk, accuracy and computational cost.’*

The methods we have used to mitigate this have been proven effective. Overall, the methodologies implemented to mitigate the lack of size and variability in our dataset compared to those in commercial use were proven to be effective. K Fold cross validation and training regularly resulted in trained models with accuracies of 95%-97% with losses generally below 0.1 indicating that we created models whose performance did balance overfitting risk, accuracy and computational cost. This is further proved by real time results.

However, we did also establish where the limitations of our dataset lie. In Figure 22, 23 and 24 we have predictions indicating that the model is 100%, 100% and 99.9% confident the person in the frame is ‘Idraq’. When comparing to Figure 26, 27, 28 and 29 the model is 97.9%, 75.31%, 22.12% and 42.57% confident it is ‘Not Idraq’. This means that the model is biased toward the ‘Idraq’ dataset hence its superior confidence.

In the ‘Not Idraq’ dataset the celebrities images are not taken in controlled conditions like our ‘Idraq’ dataset. This made our model more prone to learning noise or inconsistent patterns from this dataset during training. Our model did, however, perform much better on Person 2 (figure 31) whose pictures were taken in stable conditions like those of ‘Idraq’ dataset. This is an important result which tells us that a models performance is highly dependent on clean, consistent data from stable conditions – otherwise your model is prone to learning noise rather than underlying patterns.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Figure 31 – Testing on Person 2 within ‘Not Idraq’ dataset. Predictions : 75.80%, 73.68%, 87.23%** | | |

The models in this study each took 10 minutes to train on a small dataset on a mid-tier GPU. In total, the 20 saved models therefore took 3hrs and 20 mins. If these models were trained on a typical commercial data in the order of 100,000s it would probably take days or even weeks. This establishes the limit of computational power on model performance. A large dataset requires a large amount of processing power Thus, proving the importance of computational power proportional to the size of your dataset.

1. Conclusions
2. **Improvements:**
3. **Dataset size and variance**

For real world applications the conditions of facial recognition are inconsistent. To apply these techniques in real time scenarios you must use a dataset that that contains a wide variety of facial expressions, lighting conditions, face angles and tilts. This proves the requirement for a large dataset to match the large variance of real – world applications.

1. **K fold Cross validation**

To improve our methodology, we should use the performance metrics to provide us insights into the dataset itself. For example, fold 5 in model 2 and 11 seemed to cause a dip in performance of the model. This suggests us that the model struggled to generalise to the last 1385 images of the dataset. To mitigate this, we unsuccessfully experimented more model configurations that also validated on the last 1385 images. To counter this, it would have been more appropriate to check for any inconsistencies in this fold that may have led to a reduction in performance and replace these inconsistencies with better data. This could not be performed, however due to time constraints.

1. **Model Ensemble Techniques**

A great way to mitigate the effects of a smaller dataset is to combine the outputs of your trained models, even the poor performing ones, to produce what is known as an ensemble model.

For our example, stacking, average weighted ensemble may have been useful. Stacking involves creating a meta model of all the model’s and combining their predictions. Average weighted ensemble techniques involve averaging the predictions of the models based on their weights, i.e. the strongest models are given greater weights (and contribute more of the combined predictions).

We did not implement these techniques owing to time and computational constraints. We decided this would not increase performance enough to warrant the extra time and computational cost an ensemble model would require. It may also slow down your real time feed because the model file would be larger.

1. **Final thoughts**

To conclude the objectives of this study were completed. The limitations of dataset size and variance on CNN model performance has been well established. The techniques, such as data augmentation and K fold Cross Validation adequately mitigated the effects of smaller datasets. Upon implementation in real time, these models performed well considering the effects of limited datasets on model generalisation. We have also done well to minimise the risk of overfitting and balance computational cost with accuracy.

There are some important takeaways we can take from this study. Our first model configuration proved to be the most effective. This is also a great lesson to learn, because it was also the simplest. Ironically, we created 8 different configurations to chase performance, yet our first model produced the best results! A common phrase in Engineering and many a field is to KISS : Keep It Simple, Stupid! This phrase is reflected in this study. It is often the simple, repeatable actions that build the best results.

At the start of this this dissertation we noted that a model is ‘mathematical representation of the real world’. The models we use many small, simple calculations repeated continuously to achieve great things. It is then fitting that in modelling, just like the real world, it is the continuous repetition action of small, simple tasks that lead to great things.

References

1. S. Dick, “Artificial Intelligence,” *Issue 1*, vol. 1, no. 1, Jun. 2019, doi: <https://doi.org/10.1162/99608f92.92fe150c>.
2. Russell, S.J. and Norvig, P. (2016). *Artificial Intelligence: a Modern Approach*. 3rd ed. Upper Saddle River: Pearson.
3. (Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C. and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), pp.211–252. doi:https://doi.org/10.1007/s11263-015-0816-y.
4. Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. Cambridge, Massachusetts: The Mit Press.
5. Mukhamediev, R.I., Popova, Y., Kuchin, Y. and Zaitseva, E. (2022). Review of Artificial Intelligence and Machine Learning Technologies: Classification, Restrictions, Opportunities and Challenges. *Mathematics (2227-7390)*, [online] 10(15), pp.2552–2552. doi:https://doi.org/10.3390/math10152552.
6. Chollet, F. (2018). *Deep Learning with Python*. Shelter Island (New York, Estados Unidos): Manning, Cop.
7. Said, A.M. and Ibrahim, F.S. (2018). Comparative Study of Segmentation Techniques for Detection of Tumors Based on MRI Brain Images. *International Journal of Bioscience Biochemistry and Bioinformatics*, 8(1), pp.1–10. doi:https://doi.org/10.17706/ijbbb.2018.8.1.1-10.
8. Brownlee, J. (2019). *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>.
9. Stansbury, D. (2020). *Derivation: Derivatives for Common Neural Network Activation Functions*. [online] The Clever Machine. Available at: https://dustinstansbury.github.io/theclevermachine/derivation-common-neural-network-activation-functions.
10. TensorFlow. (n.d.). *Introduction to Tensors | TensorFlow Core*. [online] Available at: <https://www.tensorflow.org/guide/tensor>.
11. Yamashita, R., Nishio, M., Do, R.K.G. and Togashi, K. (2018). Convolutional Neural networks: an Overview and Application in Radiology. *Insights into Imaging*, [online] 9(4), pp.611–629. doi:https://doi.org/10.1007/s13244-018-0639-9
12. Chu, H., Liao, X., Dong, P., Chen, Z., Zhao, X. and Zou, J. (2019). An Automatic Classification Method of Well Testing Plot Based on Convolutional Neural Network (CNN). *Energies*, 12(15), p.2846. doi:https://doi.org/10.3390/en12152846.
13. Singh, S. (2024). Casia Face Dataset. [online] Kaggle.com. Available at: https://www.kaggle.com/datasets/cybersimar08/casia-face-dataset [Accessed 10 Sep. 2024].
14. OpenCV (n.d.). *OpenCV: Cascade Classifier*. [online] docs.opencv.org. Available at: <https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html>.
15. Bejani, M.M. and Ghatee, M. (2021). A systematic review on overfitting control in shallow and deep neural networks. *Artificial Intelligence Review*, 54(8), pp.6391–6438. doi:https://doi.org/10.1007/s10462-021-09975-1.
16. “Image classification | TensorFlow Core,” *TensorFlow*. <https://www.tensorflow.org/tutorials/images/classification>
17. Tensorflow, “tf.keras.preprocessing.image.ImageDataGenerator,” *TensorFlow*. <https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator>
18. V. H. Phung and E. J. Rhee, “A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets,” *Applied Sciences*, vol. 9, no. 21, p. 4500, Oct. 2019, doi: <https://doi.org/10.3390/app9214500>.
19. “3.3. Metrics and scoring: quantifying the quality of predictions — scikit-learn 0.22.1 documentation,” *scikit-learn.org*. <https://scikit-learn.org/stable/modules/model_evaluation.html>
20. R. K. M, S. M. Bakkannavar, Arjun M.S, and Samarth Bhaskar Bhat, “Face Detection from CCTV Footage using OpenCV and Haar Cascade,” *International journal for research in applied science and engineering technology*, vol. 11, no. 7, pp. 1185–1189, Jul. 2023, doi: https://doi.org/10.22214/ijraset.2023.54816.

***Appendix***

**Take pictures.py**

import cv2

import os

import time

webcam = cv2.VideoCapture(0)

storage\_folder = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset\_2'

frame\_count = 0

recording = False

if not webcam.isOpened():

    print("Camera not open!!")

    exit()

#////////////////////////////////////////////////////////////////////////////////////////////////

while True:

    ret,frame = webcam.read()

    if not ret:

        print("Frames not being received, exiting")

        break

    cv2.imshow("Webcam feed", frame)

    if cv2.waitKey(1) == ord('p'):

        recording = True

        print('Recording started, frames gathering...')

        while recording:

            ret,frame = webcam.read()

            if not ret:

                print('Frames not being received, exiting')

                break

            file = os.path.join(storage\_folder, f'photo\_{int(cv2.getTickCount())}.jpg')

            cv2.imwrite(file,frame)

            print(f'Frame{frame\_count} saved as {file}')

            frame\_count += 1

            cv2.imshow("Webcam feed", frame)

    elif cv2.waitKey(1) == ord('q'):

        recording = False

        break

webcam.release()

cv2.destroyAllWindows()

**DATA PREPARATION SCRIPT**

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IMG

import os

import numpy as np

import matplotlib.pyplot as plt

# Dataset directory path

DATASET\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset(not\_in\_use)\processed'

# Image Data augmentation

# Splits into val and train sets then random flips, rotates, shifts etc

img\_datagen = IMG(

    rescale = 1./255,

    validation\_split = 0.2,  # 80% for training, 20% for validation

    rotation\_range = 40,

    width\_shift\_range = 0.2,

    height\_shift\_range = 0.2,

    shear\_range = 0.2,

    zoom\_range = 0.2,

    horizontal\_flip=True,

    vertical\_flip=True

)

# Training data generator

def TrainingDataGenerator(DATASET\_DIR, img\_datagen):

    training\_datagen = img\_datagen.flow\_from\_directory(

        DATASET\_DIR,

        target\_size = (75, 75),

        color\_mode = 'grayscale',

        batch\_size = 32,

        class\_mode = 'binary',

        subset = 'training'

    )

    return training\_datagen

# Validation data generator

def ValidationDataGenerator(DATASET\_DIR, img\_datagen):

    validation\_datagen = img\_datagen.flow\_from\_directory(

        DATASET\_DIR,

        target\_size = (75, 75),

        color\_mode = 'grayscale',

        batch\_size = 32,

        class\_mode = 'binary',

        subset = 'validation'  # Using 'validation' subset

    )

    return validation\_datagen

# Generate training and validation datasets

train\_gen = TrainingDataGenerator(DATASET\_DIR, img\_datagen)

val\_gen = ValidationDataGenerator(DATASET\_DIR, img\_datagen)

# Show random pics from generator to show if assigns 1 and 0 Correctly

plt.figure(figsize = (8,8))

for images, labels in train\_gen:

    for i in range(9):

        ax = plt.subplot(3, 3, i + 1)

        plt.imshow((images[i] \* 255).astype("uint8").squeeze(), cmap='gray') #multiply by 255 to normalise pixel values

        plt.title(int(labels[i]))

        plt.axis("off")

    break

plt.show()

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IMG

import os

os.environ['TF\_ENABLE\_ONEDNN\_OPTS'] = '0'

#Layers will stack in sequential order...

def Model\_CNN(input\_shape = (75,75,1)):

    model = Sequential()

    #Layer 1(Conv + Pool)

    model.add(Conv2D(filters = 128, kernel\_size = (3,3), activation = 'relu', input\_shape = (75,75,1)))

    model.add(MaxPooling2D(pool\_size = (2,2)))

    #Layer 2(Conv + Pool)

    model.add(Conv2D(filters = 64, kernel\_size = (3,3), activation = 'relu', input\_shape = (75,75,1)))

    model.add(MaxPooling2D(pool\_size = (2,2)))

    #Layer 3(Conv + Pool)

    model.add(Conv2D(filters = 32, kernel\_size = (3,3), activation = 'relu', input\_shape = (75,75,1)))

    model.add(MaxPooling2D(pool\_size = (2,2)))

    #Flatten layer

    model.add(Flatten())

    #Fully Connected Layer 1

    model.add(Dense(units = 128, activation = 'relu'))

    model.add(Dropout(0.5))

    #Fully Connected Layer 2. Sigmoid for binary, Softmax for multi class classification

    model.add(Dense(units = 1 , activation = 'sigmoid'))

    #Model Compilation

    model.compile(optimizer = 'adam', loss = 'binary\_crossentropy', metrics = ['accuracy'])

    return model

model = Model\_CNN()

model.summary()

**MODEL TRAINING SCRIPT**

import tensorflow as tf

import os

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IMG

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, roc\_auc\_score

from data\_process\_cross\_validation import img\_dataframe

# Directories

TEST\_DATASET\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\testing\_dataset\processed'

TRAINED\_MODELS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\_2'

SAVE\_TEST\_PLOTS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\metrics\_plots'

SAVE\_RESULTS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\excel\_sheets'

def main():

    results = []

    test\_datagen = IMG(rescale=1./255)

    test\_df = img\_dataframe(TEST\_DATASET\_DIR)

    test\_gen = test\_datagen.flow\_from\_dataframe(

        dataframe = test\_df,

        x\_col = 'filename',

        y\_col = 'label',

        target\_size = (75,75),

        color\_mode = 'grayscale',

        batch\_size = 32,

        class\_mode = 'binary',

        shuffle = False #no shuffling test set!

    )

    models = [

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_1\_1.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_1\_2.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_1\_3.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_1\_4.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_1\_5.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_2\_1.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_2\_2.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_2\_3.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_2\_4.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_2\_5.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_6.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_7.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_8.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_9.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_10.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_11\_1.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_11\_2.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_11\_3.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_11\_4.keras',

        r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2\model\_11\_5.keras'

    ]

    for i, model\_path in enumerate(models):

        model = tf.keras.models.load\_model(model\_path)

        test\_gen.reset()

        # generate binary prediction. Create 0.5 threshold

        y\_pred = model.predict(x=test\_gen, steps=len(test\_gen))

        y\_pred\_binary = (y\_pred > 0.5).astype(int) #if pred > 0.5 == 1, if pred < 0.1 == 0

        y\_true = test\_gen.classes

        #calculate metrics

        accuracy = accuracy\_score(y\_true, y\_pred\_binary)

        precision = precision\_score(y\_true, y\_pred\_binary)

        recall = recall\_score(y\_true, y\_pred\_binary)

        f1 = f1\_score(y\_true, y\_pred\_binary)

        auc = roc\_auc\_score(y\_true, y\_pred)

        # Print metrics for current iteration

        print(f"\n{os.path.basename(model\_path)}:")

        print(f"  Accuracy:  {accuracy:.4f}")

        print(f"  Precision: {precision:.4f}")

        print(f"  Recall:    {recall:.4f}")

        print(f"  F1 Score:  {f1:.4f}")

        print(f"  AUC:       {auc:.4f}")

        #append to empty results list

        results.append({

            'model': f'Model {os.path.basename(model\_path)}',

            'accuracy': accuracy,

            'precision': precision,

            'recall': recall,

            'f1': f1,

            'auc': auc

        })

    metrics = ['accuracy', 'precision', 'recall', 'f1', 'auc']

    for metric in metrics:

        plt.figure(figsize=(15, 8))

        plt.bar([r['model'] for r in results], [r[metric] for r in results])

        plt.title(f'{metric.capitalize()} for each model')

        plt.xlabel('Model')

        plt.ylabel(metric.capitalize())

        plt.ylim(0, 1)

        plt.xticks(rotation=90)

        plt.tight\_layout()

        plt.savefig(os.path.join(SAVE\_TEST\_PLOTS\_DIR, f'{metric}\_plot.png'))

        plt.close()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**BATCH IMAGE CROPPING SCRIPT**

import cv2

import os

input\_directory = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset\_2\unprocessed\Uncropped\_Me'

cascades\_directory = r'C:\Users\Idraq\Desktop\Project\Python Files\haarcascade\_frontalface\_default.xml'

output\_directory = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset\_2\processed\b\_me'

#set initial value for...

total\_faces\_detected = 0

#make directory if it doesnt exist

if not os.path.exists(output\_directory):

    os.makedirs(output\_directory)

#loop through filenames in input directory

for file\_name in os.listdir(input\_directory):

    input\_path = os.path.join(input\_directory, file\_name)

    #read imgs and conv to greyscale

    img = cv2.imread(input\_path)

    img\_grey = cv2.cvtColor(img,cv2.COLOR\_BGR2GRAY)

    #identify and store x,y,w,h in array

    haar\_cascades = cv2.CascadeClassifier(cascades\_directory)

    faces = haar\_cascades.detectMultiScale(img\_grey, scaleFactor = 1.1, minNeighbors = 4)

    num\_faces = len(faces)

    total\_faces\_detected += num\_faces

    print(f'Detected {num\_faces} in {file\_name}')

    #Rectangle around around face, then crop, then save

    for i,(x,y,w,h) in enumerate(faces):

        cv2.rectangle(img\_grey, pt1 = (x,y), pt2 = ((x+w),(y+h)), color = (0,255,0), thickness = 2)

        faces\_cropped = img\_grey[y: y+h , x: x+w]

        final\_face = cv2.resize(faces\_cropped, (75, 75)) # images will be (75,75,1)

        #save each processed image

        output\_filename = f'not\_me\_processed\_photo\_{int(cv2.getTickCount())}.jpg' #change me or not me based on use

        output\_path = os.path.join(output\_directory,output\_filename)

        cv2.imwrite(output\_path,final\_face)

print(f'total faces detected: {total\_faces\_detected}')

print(final\_face.shape)

print(faces)

cv2.destroyAllWindows()

**K FOLD CROSS VALIDATION**

import tensorflow as tf

import os

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model\_selection import KFold

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IMG

from sklearn.model\_selection import StratifiedKFold

from tensorflow.keras.callbacks import ModelCheckpoint

#import my model from model script

from model\_script import Model\_CNN #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*REMEMBER TO CHANGE!!!!!!!

# Directories

TRAINED\_MODELS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2'

DATASET\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset\_2\processed'

SAVE\_PLOTS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\metrics\_plots\metrics\_plots\_2'

# ImageDataGenerator with data augmentation

# also splits data into train val sets, while randomly flipping tilting etc images

img\_datagen = IMG(

    rescale=1./255, #set between 0 and 1 for input into model training

    validation\_split = 0.2,  # 80% for training, 20% for validation

    rotation\_range = 40,

    width\_shift\_range = 0.2,

    height\_shift\_range = 0.2,

    shear\_range = 0.2,

    zoom\_range = 0.2,

    horizontal\_flip=True,

    vertical\_flip=True

    )

# Image dataset conversion into a dataframe of two columns

# Column 1 = filenames, Column 2 = labels

def img\_dataframe(DATASET\_DIR):

    file\_paths = []

    labels = []

    #assign labels to folders

    for folder\_name in os.listdir(DATASET\_DIR):

        folder\_path = os.path.join(DATASET\_DIR, folder\_name)

        if os.path.isdir(folder\_path):

            if folder\_name == 'b\_me':

                label = '1'

            elif folder\_name == 'a\_not\_me':

                label = '0'

            else:

                continue

        #append filenames and labels to empty lists created earleir

        for file\_name in os.listdir(folder\_path):

             file\_path = os.path.join(folder\_path, file\_name)

             file\_paths.append(file\_path)

             labels.append(label)

    #create dataframe

    img\_df = pd.DataFrame({

         'filename' : file\_paths,

         'label' : labels

    })

    return img\_df

# \*\*\*\*Now must create function to perform k fold cross validation\*\*\*\*

def k\_fold\_cross\_validation(k=5, img\_df = None):

     results = []

     file\_names = img\_df['filename']

     labels = img\_df['label']

     skf = StratifiedKFold(n\_splits = k, shuffle = True, random\_state = 42)

     #for i, (train\_index, test\_index) in enumerate(skf.split(X, y)):

     for fold\_index, (train\_index, test\_index) in enumerate(skf.split(file\_names,labels)):

        #create dataframes for each fold iteration

        train\_df = pd.DataFrame({

             'filename': file\_names.iloc[train\_index].values,

             'label': labels.iloc[train\_index].astype(str).values

        })

        val\_df = pd.DataFrame({

             'filename': file\_names.iloc[test\_index].values,

             'label' : labels.iloc[test\_index].astype(str).values

        })

        #flow from dataframe

        training\_datagen = img\_datagen.flow\_from\_dataframe(

            train\_df,

            x\_col = 'filename',

            y\_col = 'label',

            target\_size=(75,75),

            color\_mode='grayscale',

            batch\_size = 32,

            class\_mode='binary',

            shuffle = True

        )

        validation\_datagen = img\_datagen.flow\_from\_dataframe(

             val\_df,

             x\_col = 'filename',

             y\_col = 'label',

             target\_size = (75,75),

             color\_mode = 'grayscale',

             batch\_size = 32,

             class\_mode = 'binary',

             shuffle = False

        )

        model = Model\_CNN(input\_shape = (75,75,1))

        #create path to save models from each fold

        model\_path = os.path.join(TRAINED\_MODELS\_DIR, f'model\_1\_{fold\_index + 1}.keras')#/////////////////////////////////////////////////////////REMEMBER TO CHANGE!!

        checkpoint = ModelCheckpoint(

            filepath = model\_path,

            save\_best\_only = True,

            mode = 'auto',

            verbose = 1

        )

        history = model.fit(

             x = training\_datagen,

             validation\_data = validation\_datagen,

             epochs = 10,

             callbacks = [checkpoint]

        )

        # Get results

        results.append({

            'fold': fold\_index + 1,

            'history': history.history

        })

     return results

# Function to plot and save metrics per fold

def metrics(metric\_name, results, title, ylabel):

    for i, result in enumerate(results):

        plt.figure(figsize=(12, 8))

        plt.plot(result['history'][metric\_name], label=f'Fold {i+1}')

        plt.title(f"{title} - Fold {i+1}")

        plt.xlabel('Epoch')

        plt.ylabel(ylabel)

        plt.legend()

        plt.savefig(os.path.join(SAVE\_PLOTS\_DIR,f"{metric\_name}\_fold\_{i+1}\_plot.png"))

        plt.close()

        print (f'Saved{metric\_name}plot for fold {i+1} to {SAVE\_PLOTS\_DIR}')

def main():

    # Run K-Fold Cross Validation

    img\_df = img\_dataframe(DATASET\_DIR)

    results = k\_fold\_cross\_validation(k = 5, img\_df = img\_df)

    # Now create and save plots for training/validation accuracy, loss vs epochs

    if not os.path.exists(SAVE\_PLOTS\_DIR):

        os.makedirs(SAVE\_PLOTS\_DIR)

    # Plot and save metrics

    metrics('accuracy', results, 'Training Accuracy vs Epochs', 'Accuracy')

    metrics('val\_accuracy', results, 'Validation Accuracy vs Epochs', 'Validation Accuracy')

    metrics('loss', results, 'Training Loss vs Epochs', 'Loss')

    metrics('val\_loss', results, 'Validation Loss vs Epochs', 'Validation Loss')

    # Print summary results for each fold

    for result in results:

        print(f"Fold {result['fold']} - Final val\_accuracy: {result['history']['val\_accuracy'][-1]:.4f}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**LIVE FACE DETECTION**

import cv2

import tensorflow as tf

import numpy as np

MODEL\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_final\model\_1\_4.keras'

CASCADES\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\haarcascade\_frontalface\_default.xml'

#load model and haar cascades classifier

model = tf.keras.models.load\_model(MODEL\_DIR)

cascades = cv2.CascadeClassifier(CASCADES\_DIR)

if model is None:

    raise Exception ("WHERE IS MY MODEL?!")

# Initialise video capture

webcam = cv2.VideoCapture(0)

# initialise list to enter predictions for average pred

latest\_predictions = []

while True:

    ret, frame = webcam.read()

    if not ret:

        break

    # Convert frame to grayscale for Haar cascade

    grey = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

    # Detect faces using Haarcascades to detect face and return values x,y,w,h (of the faces) in an array.

    grey\_faces = cascades.detectMultiScale(grey, scaleFactor = 1.3, minNeighbors = 9, minSize = (40, 40))

    for (x, y, w, h) in grey\_faces:

        #resize faces to (75,75) to fit in model using xywh values

        grey\_faces\_resized = cv2.resize(grey[y:y+h, x:x+w], (75, 75))

        # normalise face and add batch dimension to process 1 image at time

        normalised\_faces = grey\_faces\_resized / 255.0

        face\_input = np.expand\_dims(normalised\_faces, axis = (0, -1))

        # Make model predict me or not me

        prediction = model.predict(face\_input)

        prediction\_score = prediction[0][0]

        latest\_predictions.append(prediction\_score) #append latest predictions

        if len(latest\_predictions) > 5 : #change this number to how many frames you want it to average over

            latest\_predictions.pop(0)

        averaged\_predictions = np.mean(latest\_predictions)

        prediction\_percentage = averaged\_predictions \* 100 # pred as % to show in on label

        threshold\_value = 0.8 #threshold high bc binary classification + small dataset

        if averaged\_predictions > threshold\_value:

            label = f'Idraq: {prediction\_percentage:.2f}%'

            color = (0, 255, 0)  # Green if me

        else:

            label = f'Not Idraq: {100 - prediction\_percentage:.2f}'

            color = (0, 0, 255)  # Red if not me!!

        # Draw rectangle around face and label

        cv2.rectangle(frame, (x, y), (x+w, y+h), color, 2)

        cv2.putText(frame, label, (x, y), cv2.FONT\_HERSHEY\_SIMPLEX, 0.9, color, 2)

    # Display the result

    cv2.imshow('Face Detection', frame)

    # Break the loop if 'q' is pressed

    if cv2.waitKey(1) & 0xFF == ord('q'):

        break

# Release resources

webcam.release()

cv2.destroyAllWindows()

**MODEL TRAIN**

import tensorflow as tf

import os

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IMG

from tensorflow.keras.callbacks import ModelCheckpoint

from model\_script\_5 import Model\_CNN as Model\_5

from model\_script\_5\_2 import Model\_CNN as Model\_5\_2

from model\_script\_6 import Model\_CNN as Model\_6

from model\_script\_7 import Model\_CNN as Model\_7

from model\_script\_8 import Model\_CNN as Model\_8

from model\_script\_9 import Model\_CNN as Model\_9

from model\_script\_10 import Model\_CNN as Model\_10

#change based on which model(s) you'd like to train

MODELS = [

     Model\_5,

]

# Directories

TRAINED\_MODELS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\trained\_models\trained\_models\_2'

DATASET\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\dataset\_2\processed'

SAVE\_PLOTS\_DIR = r'C:\Users\Idraq\Desktop\Project\Python Files\metrics\_plots\metrics\_plots\_2'

# ImageDataGenerator for data augmentation

# Will randomly flip, shift, rotate images

img\_datagen = IMG(

    rescale = 1./255, # normalise to values between 0 and 1 to feed into model

    validation\_split = 0.2,  # 80% for training, 20% for validation

    rotation\_range = 40,

    width\_shift\_range = 0.2,

    height\_shift\_range = 0.2,

    shear\_range = 0.2,

    zoom\_range = 0.2,

    horizontal\_flip = True,

    vertical\_flip = True

)

# Image dataset conversion into a dataframe of two columns

def img\_dataframe(DATASET\_DIR):

    file\_paths = []

    labels = []

    # Assign labels to folders

    for folder\_name in os.listdir(DATASET\_DIR):

        folder\_path = os.path.join(DATASET\_DIR, folder\_name)

        if os.path.isdir(folder\_path):

            if folder\_name == 'b\_me':

                label = '1'

            elif folder\_name == 'a\_not\_me':

                label = '0'

            else:

                continue

        # Append filenames and labels to empty lists created earlier

        for file\_name in os.listdir(folder\_path):

            file\_path = os.path.join(folder\_path, file\_name)

            file\_paths.append(file\_path)

            labels.append(label)

    # Create dataframe

    img\_df = pd.DataFrame({

        'filename': file\_paths,

        'label': labels

    })

    return img\_df

def train\_model(model\_name, model\_fn, img\_df):

    training\_datagen = img\_datagen.flow\_from\_dataframe(

        img\_df,

        x\_col='filename',

        y\_col='label',

        target\_size=(75,75),

        color\_mode='grayscale',

        batch\_size=32,

        class\_mode='binary',

        subset='training',

        shuffle=True

    )

    validation\_datagen = img\_datagen.flow\_from\_dataframe(

        img\_df,

        x\_col='filename',

        y\_col='label',

        target\_size=(75,75),

        color\_mode='grayscale',

        batch\_size=32,

        class\_mode='binary',

        subset='validation',

        shuffle=False

    )

    # Call model function to get model architecture

    model = model\_fn(input\_shape=(75,75,1))

    # Create path to save models from each fold

    model\_path = os.path.join(TRAINED\_MODELS\_DIR, f'{model\_name}.keras')

    checkpoint = ModelCheckpoint(

        filepath=model\_path,

        save\_best\_only=True,

        mode='auto',

        verbose=1

    )

    # Train the model

    history = model.fit(

        x=training\_datagen,

        validation\_data=validation\_datagen,

        epochs=10,

        callbacks=[checkpoint]

    )

    return history

def plot\_save\_metrics(history, model\_name):

    model\_plot\_dir = os.path.join(SAVE\_PLOTS\_DIR, model\_name)

    if not os.path.exists(model\_plot\_dir):

        os.makedirs(model\_plot\_dir)

    # Plot accuracy

    plt.figure(figsize=(12, 6))

    plt.plot(history.history['accuracy'], label='Training Accuracy')

    plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

    plt.title(f'{model\_name} - Accuracy vs Epochs')

    plt.xlabel('Epochs')

    plt.ylabel('Accuracy')

    plt.legend()

    plt.savefig(os.path.join(model\_plot\_dir, f'{model\_name}\_accuracy\_plot.png'))

    plt.close()

    # Plot loss

    plt.figure(figsize=(12, 6))

    plt.plot(history.history['loss'], label='Training Loss')

    plt.plot(history.history['val\_loss'], label='Validation Loss')

    plt.title(f'{model\_name} - Loss vs Epochs')

    plt.xlabel('Epochs')

    plt.ylabel('Loss')

    plt.legend()

    plt.savefig(os.path.join(model\_plot\_dir, f'{model\_name}\_loss\_plot.png'))

    plt.close()

def main():

    # Load image dataframe

    img\_df = img\_dataframe(DATASET\_DIR)

    # Loop over all models

    for i, model\_i in enumerate(MODELS, start = 6):  # Starting index from 7

        model\_name = f'model\_{i}'

        print(f"Training {model\_name}...")

        # Train the model

        history = train\_model(model\_name, model\_i, img\_df)

        # Plot and save metrics

        plot\_save\_metrics(history, model\_name)

        print(f"{model\_name} training complete and plots saved.")

if \_\_name\_\_ == '\_\_main\_\_':

    main()