Fingerprint Recognition using Deep Learning

Alexander Abrantes Figueiras
ID: 40007320
Concordia University
Montreal, Quebec
https://github.com/Figgueh/FreshPrints

***Abstract*—This report explores the use of Convolutional Neural Networks (CNNs) in classifying fingerprints for biometric identification. The study uses the Sokoto Coventry Fingerprint Dataset (SOCOFing) and trains two models to detect the original image by classifying altered images. The report outlines the dataset used, the preprocessing steps taken, the model architecture, the proposal of a new more accurate model, and experimental results. The main goal of the project was to achieve high accuracy and robustness while feeding the network with the least amount of lowest quality input possible, using Google Colab to train the model. The report demonstrates the effectiveness of using CNNs for fingerprint classification and provides insights into the potential of deep learning techniques in biometric identification.**

***Index Terms*—Deep Learning, Fingerprint classification, Convolution, Neural Networks, Convolutional Neural Network.**

I. Introduction

Our fingers are a unique part of our body that can provide insight into our lifestyle, profession, and personality. Whether they are covered in dirt and callouses or pampered and manicured, the appearance and condition of our fingers can reveal a lot about us. In this report we take a closer look at them particularly, the prints they leave behind and the telling story of their origins. Prints have long been used as a reliable method for identifying individuals and with the increasing demand for secure and accurate bio-metric authentication, fingerprint recognition technology has become a crucial aspect of many applications.

While conducting research on the feasibility of performing fingerprint classification, various methods were considered. It is notable, however, that Convolutional Neural Networks (CNNs) have emerged as a formidable tool for image classification tasks, and by extension a potentially viable solution fingerprint recognition. This observation served as a catalyst for the inception of the current report.

We explore the application of CNNs in classifying fingerprints, with the goal of achieving high accuracy and robustness while feeding the network with the least amount of input possible. We describe the architecture of our CNN model, the dataset used for training and evaluation, and the experimental results. Our findings demonstrate the effectiveness of using CNNs for fingerprint classification and provide insights into the potential of deep learning techniques in bio-metric identification.

II. Motive

During the research phase of the report, the discovery of a dataset and project which will be used for the base of the application was made. Conversely, without the access to a fancy Nvidia graphics card training the model would have been rather time consuming to execute on the CPU considering that the architecture of the model required that two models be trained. This is to explain the dependence on Google Colab, as they offer the limited use of a GPU for free. This dependence comes with its own restrictions, mainly the restriction of available RAM caused an issue due to the large number of images in the dataset. This conceived the main goal of the project: To be able to get similar accuracy to the original project, however removing one set of images from the dataset to be able to fit in the environment provided by Google Colab.

III. Source Methodology

The following is a recapitulation of the original model developed by Brian Zhang [2].

A. Dataset

The dataset consumed by this CNN consists of images that are from the Kaggle website, by the name of Sokoto Coventry Fingerprint Dataset (SOCOFing).

"Sokoto Coventry Fingerprint Dataset (SOCOFing) is a biometric fingerprint database designed for academic research purposes. SOCOFing is made up of 6,000 fingerprint images from 600 African subjects and contains unique attributes such as labels for gender, hand and finger name as well as synthetically altered versions with three different levels of alteration for obliteration, central rotation, and z-cut." [1]

Upon downloading the dataset from the website and extracting its contents, two additional folders will be revealed. The first named 'Real' which contains the original images of the fingerprints. The second named 'Altered' which contains 3 additional folders with levels of difficulty varying from easy to hard. These folders contain the altered versions of the fingerprints that will be use to train our CNN with, in hopes of being able to detect which image was the original.

Each image in the dataset is named with the following formatting:

'ID'__'sex'_'Right/Left'_'finger'.BMP

- ID – Identifying number of that fingerprint
- Sex – The sexual orientation of the person
- Right/Left – The hand from which the print came
- Finger – The particular finger of the print

B. Preprocessing

The dataset came with images which have different levels of alteration for obliteration, central rotation, and z-cut. To maintain an acceptable aspect ratio and uniform size, each image is reshaped into 96 by 96 before processing. The image data is then normalized between values 0 to 1 and split into two partitions of a 80/20 split for training and testing data respectively.

C. Dependencies

- **OpenCV2**

OpenCV2 is an open-source computer vision and machine learning software library that provides a comprehensive set of tools and functions for image and video processing, object detection and recognition, feature detection and matching, and camera calibration, among other tasks. Its ability to work with both 2D and 3D images and video, and its set of tools for machine learning make it a powerful tool for computer vision research and development. It is used in our application for initial processing of images.

- **Numpy**

NumPy is a Python library designed for scientific computing, providing support for large multi-dimensional arrays and matrices, and a collection of optimized mathematical functions. It's optimized for numerical computations on large datasets and is often used in combination with other libraries for data analysis, visualization, and machine learning.

- **Mathplotlib**

Matplotlib is a popular Python library for creating high-quality visualizations, including line plots, scatter plots, bar plots, histograms, and more. It offers a wide range of options for controlling the appearance of plots, including line styles, colors, fonts, and labels, and supports multiple plotting backends, including interactive backends for use in Jupyter notebooks and web applications. With its ability to handle large datasets and create complex visualizations, Matplotlib is widely used in data exploration and presentation in scientific and engineering fields.

- **Keras**

Keras has become one of the most popular deep learning libraries due to its simplicity, versatility, and ease of use. It has a large and active community of developers who contribute to its development and provide support for users. Keras also integrates well with other popular Python libraries such as NumPy, Pandas, and Matplotlib, making it easy to incorporate deep learning into existing data science workflows. Overall, Keras is a powerful tool for anyone looking to experiment with deep learning and create advanced neural network models.

- **Scikit-Learn**

Scikit-learn is designed to be user-friendly and flexible, with a focus on enabling rapid experimentation and model building. It offers a wide range of tools for preprocessing data, feature extraction and selection, model selection and evaluation, and performance metrics. It also provides a variety of pre-built models, including popular algorithms such as decision trees, random forests, support vector machines, and neural networks, making it easy to get started with machine learning.

D. Model architecture

This network trains 2 models and later cross references them as a means of validation. It contains 14 layers and totals to 2,935,512 parameters for the first model and 2,935,512 parameters for the second model. For a visual representation of the sequence of operations and size of each layer see figure 3. Both models mirror each other and consist of the following operations:

1) Convolutional layers:
   Seen as the major building block of a Convolutional Neutral Network, where a set of filters is applied to an input image to produce a set of feature maps. Each filters is a small matrix which slides across an input image. Computing the dot product between the filters and each local patch of the input image of which it is currently processing. The resulting values are then summed up to a single value, which is placed in the corresponding position of the output feature map. This model contains 3 different convolutional layers, 2 of size 5x5 and an additional one of size 3x3.

   Both 5x5 filters are positioned before the 3x3 which is surprising given that most models usually apply the smaller filters first then the larger ones. Due to the nature of convolution, where starting with the smaller filters first allows for lower computation costs, captures more fine-grained features, and increases the receptive field. This would translate to giving network more of an opportunity to capture the important details about high frequency parts of the image such as edges or swirls before considering larger parts of the fingerprint.

2) Batch Normalization layers:
   Batch normalization is a technique used in convolutional neural networks (CNNs) to improve the performance of the network by reducing internal covariate shift. Internal covariate shift refers to the phenomenon where the distribution of the input to a layer change as the parameters of the previous layers are updated during training. This can make it difficult for the network to learn and slow down the training process. Our models use the batch normalization layer after each convolutional layer.

3) Max Pool Layers:
   Pooling layers have several benefits for neural networks.

First, it reduces the dimensionality of the feature maps, which helps to reduce the computational burden of the network and prevent overfitting. Second, it helps to make the network more invariant to small translations of the input image, since the maximum value within a pooling window is likely to be the same regardless of the exact location of the feature within the window. Finally, it helps to introduce some degree of spatial invariance into the network, since the output of the pooling operation is not affected by small changes in the exact location of the feature within the pooling window.
In our network it is used after each batch normalization layer, this is done to increase the effectiveness of the Max Pooling results. Normalization can help reduce the impact of outliers which will in turn produce a more robust model.

4) Flattening Layers:
Used to convert the output of the previous convolutional layer or pooling layer into a one-dimensional feature vector, which can be inputted into a fully connected layer.

5) Dense Layers:
A type of fully connected layer where each neuron in the layer is connected to every neuron in the previous layer. It performs a linear operation on its input, followed by the activation of the ReLU function in our specific model. Dense layers are often used towards the end of the network to perform classification or regression tasks based on the input features. Lastely they enable the network to learn complex, non-linear relationships between the input and output features.

6) Dropout Layers:
During training each neuron in the dropout layer is retained with a given probability. Which means that each time an input is fed to the network during training, some of the neurons in the dropout layer are randomly selected to be set to zero (dropped out), and the remaining neurons are scaled up by a factor of 1/'given probability' to maintain the overall magnitude of the layer's output. By dropping out neurons during training, the dropout layer helps to prevent the network from relying too much on any particular input feature or neuron, forcing it to learn more robust and generalizable representations. During inference, the dropout layer is typically turned off and all neurons are used to make the prediction.

IV. Alterations

As previously mentioned, there needed to be some adjustments to program to be able to run on Google collab.

A. Constraints

- 12.7 Gigabytes of system RAM
- 15.0 Gigabyte of GPU RAM
- 78.2 Gigabyte of disk space

TABLE I
ACCURACY OF BOTH ORIGINAL MODELS BEING TRAINED WITH THE ORIGINAL NETWORK USING DIFFERENT CONFIGURATIONS OF DATA.

Dataset configuration	ID recognition accuracy	Finger recognition accuracy
Easy & Medium	99.6999979019165%	99.86666440963745%
Easy & Hard	99.6333360671997%	99.6833324432373%
Medium & Hard	97.81666398048401%	91.41666889190674%

B. Warnings

Seldomly the compiler would issue the following warning when attempting to process the dataset

Listing 1. Compiler warning.

```
<__array_function__ internals>:180:
    VisibleDeprecationWarning: Creating an
     ndarray from ragged nested sequences
    (which is a list-or-tuple of lists-or-
    tuples-or ndarrays with different
    lengths or shapes) is deprecated. If
    you meant to do this, you must specify
     'dtype=object' when creating the
    ndarray.
```

This was rectified by changing the following line:

Listing 2. Line with the error.

```
Altered_data = np.concatenate([Easy_data,
     Medium_data, Hard_data], axis=0)
```

To:

Listing 3. Corrected line.

```
Altered_data = np.concatenate([np.array(
    Medium_data, dtype=object), np.array(
    Hard_data, dtype=object)], dtype=
    object, axis=0)
```

Due to the structure of how the array is built, a conversion to a Numpy array is necessary to indicate that the type of the variable is in fact an object, more specifically an object containing array entries with the image ID, finger number and resized image.

C. Modifications to dataset

To be able to get the application to run within the specified limit, one of the directories of the images needed to be removed. The decision of which one will be removed was made based on which one the model seemed to rely on more. We hypothesised that removing the easy images would have the greatest impact since they would theoretically be closer to the original image, nether the less, the model was trained with all possible configurations of the 3 difficulties and their accuracy recorded in table I.

See figures 4 and 6 for the graphical representations of the 'medium and hard' accuracy functions. Figures 5 and 7 represent the loss function with the same dataset.

As predicted the model preformed worse when we removed the easy images. Admittedly, there isn't much room for improvement and simply adjusting or removing the dropout layers might have accounted for the adjustment in training data. This whoever does increase its likely hood to overfit which inspired the creation of a new model.

D. Experimentation

Since this was a pioneering project into neutral networks, some time was spent adjusting hyper parameters, changing the sequence and kernel size of the existing model then analyzing their impact on the results.

1) Additional convolutional 2D layer: An additional convolutional layer was placed ahead of the original models first 5x5 convolutional layer. It was of size of 7x7, this was done in hopes of allowing the algorithm to consider more of the whole image instead of just looking at smaller ones. This however did not have a positive effect during the fitting process until adjustments for an extra batch normalization layer and max pool 2D layer were made. Which lead to the discovery that the model would have better accuracy if the smaller kerels are processed at the begging.

2) Activation function change: Since the images we are looking at are the more obfuscated version, an attempt was made to change the activation function to be leaky ReLU instead of the original version in hopes of being able to have more control on what we want to learn from the image. The idea was that we might want to not learn as much from the portions of the image were altered. However, this modification didn't affect our model as expected and produced worse results.

E. Proposed model

The final proposed model consists of 4 stages. It was heavily influenced by the original model proposed by Brian Zhang but with the addition of a couple different sized convolutional layers. The inspiration for the kernel size came from the Fibonacci spiral.

The first stage consists of two convolutional layers, which learns 32 features each and have a kernel size of 1x1 for the first, and 1x2 for the second. These layers are then normalized with batch normalization and passed through a Max Pool layer with kernel size 2x2.

The second stage involves an additional 2 convolutional layers, which learn 64 features each and have kernel size of 2x1 for the first, and 2x2 for the second. These layers are also Normalized and Pool with the same methods from the previous stage.

The third stage is comprised of 3 final convolutional layers, which learn 128 features each. Their size starts at 3x1 and increments to 3x3. These layers then follow the same normalization and pooling as the stages before. All instances of convolutional layers are activated with the ReLU function.

The last stage includes the addition of a dropout layer before being flatten, then two Multilayer perceptions (dense layers). The first dense layer has a total of 256 units, then a dropout layer is applied follow by the second dense layer with a total of either 600 or 10 units depending on which model is being trained. Both dense layers layers are active with ReLU function

All these layers combine to be a total of 3,070,264 parameters for the first model, and 2,918,634 for the second. For a visual representation of the sequence of operations and size of each layer of the proposed model see figure 8. We opted to use multiple smaller layers than the original model since stacking the convolutional layers like so would have the same effective receptive field as a larger kernel, but results in a deeper network with more non-linearities.

V. Evaluation and results

A. Hyper-parameters

When training the model, the following hyperparameters were used:

- **Batch size**

References the number of training examples used within one iteration of the optimization algorithm. Larger values can improve accuracy but at the expense of more memory which in turn demands more processing time to complete the algorithm. In our application, the best results were realized when setting the batch size to 64.

- **Optimizer: Adam**

Adam (Adaptive Moment Estimation) is a popular optimization algorithm used in deep learning that has advantages over both the AdaGrad and RMSProp algorithms. The Adam optimizer adapts the learning rate for each parameter based on the past gradients and updates, allowing it to converge faster and more efficiently than traditional stochastic gradient descent (SGD). It is called Adam due to it requiring only first and second order gradients to adapt the learning rate. The decision was made to stick with Adam due to the hyperparameters having intuitive interpretations and hence required less tuning.

- **Loss function: Categorical cross entropy**

For multi-class classifications programs, categorical cross entropy would be the most performant since it measures the dissimilarity between the predicted probability distribution of class labels and the actual distribution. probability distribution is obtained by applying a SoftMax activation function to the final layer of the model, which ensures that the output values are between 0 and 1 and sum up to 1. Actual distribution is usually represented as a one-hot encoded vector.
The goal of the optimization process is to minimize the value of the categorical cross-entropy, which effectively maximizes the similarity between the predicted and actual distributions.

- **Epochs**

Epochs refers to one full iteration of the training dataset during the training process. After the completion of one epoch, the algorithm would have used 64 images to update the model's parameters. Choosing a fair number of Epochs could improve the accuracy of the model, whoever too many Epochs can lead to overfitting the model causing it to perform poorly on unseen data. For this application, the decision to run the model for 20 Epochs was seen as the happy medium between execution time and amount of memory use.

B. Evaluation Criteria

Referred to as metrics which measure the quality and correctness of a model. These metric makes use of the predictions made by our model using the 20% of the data reserved for testing.

With Keras, the calculation of accuracy is dependent on what is selected as our loss function. Since we have categorical crossentropy selected, Keras will calculate the accuracy function with Categorical Accuracy. This measurement of accuracy considered how often the predictions made match the one-hot labels. Then simply divided the number of correct matches with the total number of possible matches.

Although there are other forms of evaluation metrics such as precision and recall we are primarily concerned with the accuracy of our models. Both models are later cross validated to ensure that either model did not incorrectly classify the fingerprint.

C. Confusion-Matrix

Used to evaluate the performance of a classification algorithm. It summarizes the number of correct and incorrect classifications of the given model.

Refer to Figure 1 for the confusion matrix of the original model trained with the medium and hard images. Figure 2 represents the confusion matrix of the proposed model trained with the medium and hard images

Fig. 1. Original model confusion matrix results.

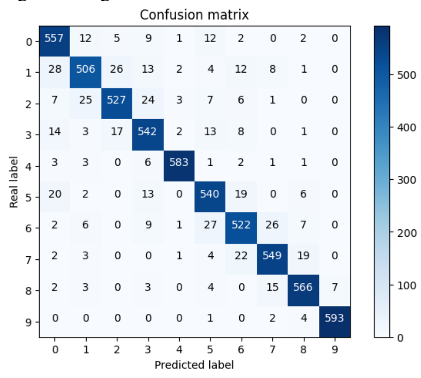
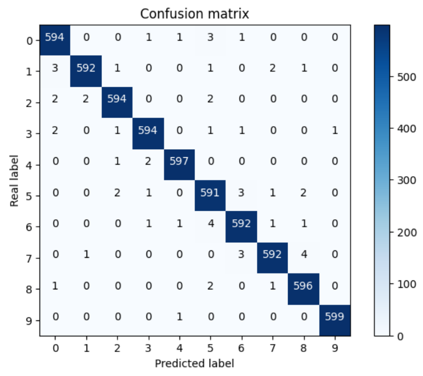


TABLE II
ACCURACY OF BOTH MODELS BEING TRAINED WITH THE PROPOSED NETWORK USING ONLY THE MEDIUM AND HARD IMAGES.

Model:	Accuracy:
Id recognition	98.93333315849304%
Finger recognition	99.01666641235352%

Fig. 2. Proposed model confusion matrix results.



D. Results

From the experiments ran using our CNN model, we were able to conclude that our proposed model does in fact increase in accuracy from the original model. See table II for the numerical representation of the accuracy of the proposed model.

As previously mentioned, there was not much room for improvement, but the end goal was achieved. This model was able to achieve similar scores when tested with all 3 combinations of training data and as a result was able to be executed within Google Colab's environment.

Figures 4 & 6 show the accuracy while figures 5 & 7 show the loss of the original models ID detection & finger recognition respectively. Figures 9 & 11 show the accuracy while figures 10 & 12 show the loss of the proposed models ID detection & finger recognition respectively.

Analyzing all four graphs generated for the two models, we can see that the proposed model was able to consistently get closer to the target values. The confusion matrix also supports this analysis as it shows that the proposed model was also able to get a better overall accuracy.

VI. CONCLUSION & FUTURE WORK

To conclude although our proposed model was able to get a better accuracy, we did not have very much time to be able to extensively experiment with different model architecture. This is amplified by the fact that each execution would take around 30 minutes and that we would be limited on how many times a day we were able to train the network. It would be interesting to see if we can apply some inception modules like the ones used in GoogLeNet to be able to get more accurate results with an even more heavily altered or sparce dataset.

Another interesting approach would be to incorporate some pretrained model. Unfortunately, we could not take this route with the original model as we would have needed to have it pretrained. A more feasible alternative would have been to take some other popular model and fine tune it to work with our image dataset. However, whilst experimenting we discovered that the model we proposed performed well and decided that showing our proposed model would have been more impressive than altering a pretrained one.

References

[1] "Sokoto Coventry Fingerprint Dataset (SOCOFing) — Kaggle." https://www.kaggle.com/datasets/ruizgara/socofing (accessed Apr. 17, 2023).

[2] "SubjectID&Finger_CNNRecognizer — Kaggle." https://www.kaggle.com/code/brianzz/subjectid-finger-cnnrecognizer (accessed Apr. 17, 2023).

VII. Appendix A - Original model

Fig. 4. Original model, Medium and Hard images validation accuracy for ID detection.

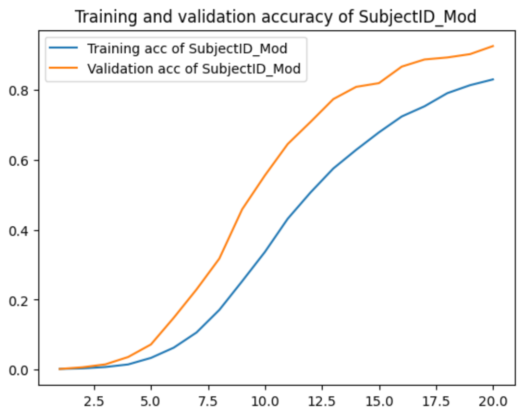


Fig. 5. Original model, Medium and Hard images validation loss for ID detection.

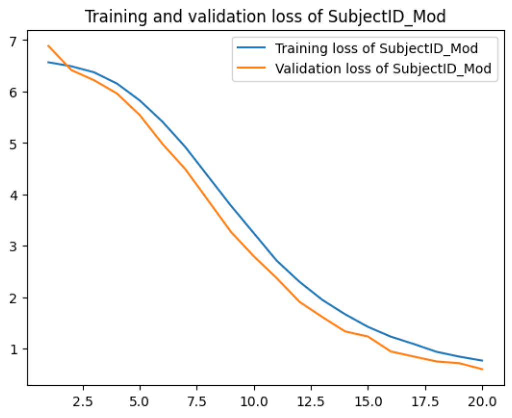


Fig. 3. Original model architecture.

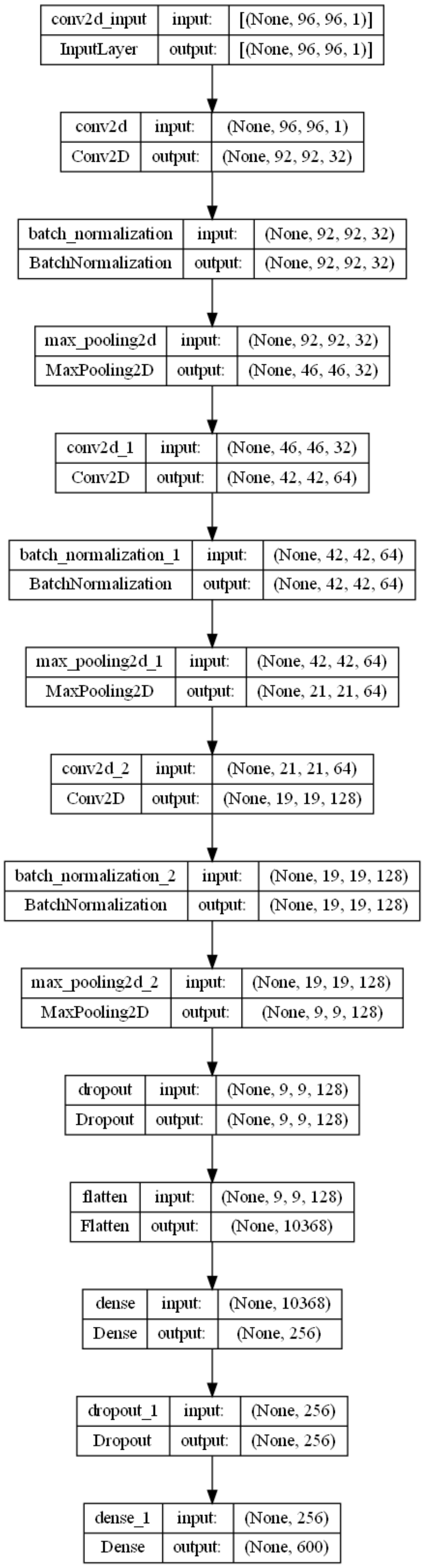

conv2d_input | input: | [(None, 96, 96, 1)]
InputLayer | output: | [(None, 96, 96, 1)]
conv2d | input: | (None, 96, 96, 1)
Conv2D | output: | (None, 92, 92, 32)
batch_normalization | input: | (None, 92, 92, 32)
BatchNormalization | output: | (None, 92, 92, 32)
max_pooling2d | input: | (None, 92, 92, 32)
MaxPooling2D | output: | (None, 46, 46, 32)
conv2d_1 | input: | (None, 46, 46, 32)
Conv2D | output: | (None, 42, 42, 64)
batch_normalization_1 | input: | (None, 42, 42, 64)
BatchNormalization | output: | (None, 42, 42, 64)
max_pooling2d_1 | input: | (None, 42, 42, 64)
MaxPooling2D | output: | (None, 21, 21, 64)
conv2d_2 | input: | (None, 21, 21, 64)
Conv2D | output: | (None, 19, 19, 128)
batch_normalization_2 | input: | (None, 19, 19, 128)
BatchNormalization | output: | (None, 19, 19, 128)
max_pooling2d_2 | input: | (None, 19, 19, 128)
MaxPooling2D | output: | (None, 9, 9, 128)
dropout | input: | (None, 9, 9, 128)
Dropout | output: | (None, 9, 9, 128)
flatten | input: | (None, 9, 9, 128)
Flatten | output: | (None, 10368)
dense | input: | (None, 10368)
Dense | output: | (None, 256)
dropout_1 | input: | (None, 256)
Dropout | output: | (None, 256)
dense_1 | input: | (None, 256)
Dense | output: | (None, 600)

Fig. 6. Original model, Medium and Hard images validation accuracy for finger detection.

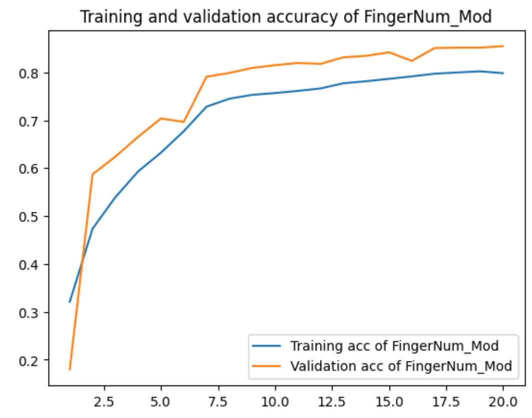
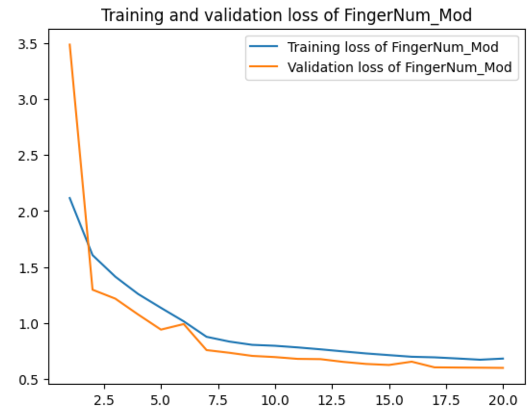


Fig. 7. Original model, Medium and Hard images validation loss for finger detection.



VIII. Appendix B - Proposed model

Fig. 9. Proposed model, Medium and Hard images validation accuracy for ID detection.

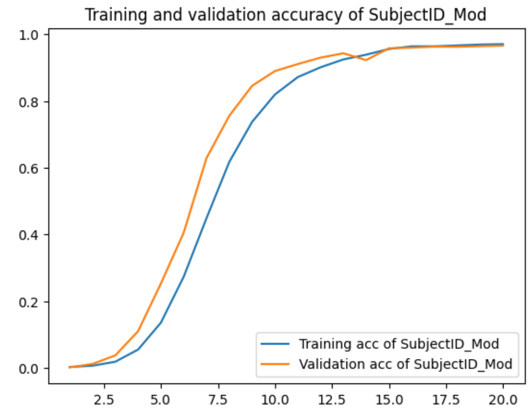
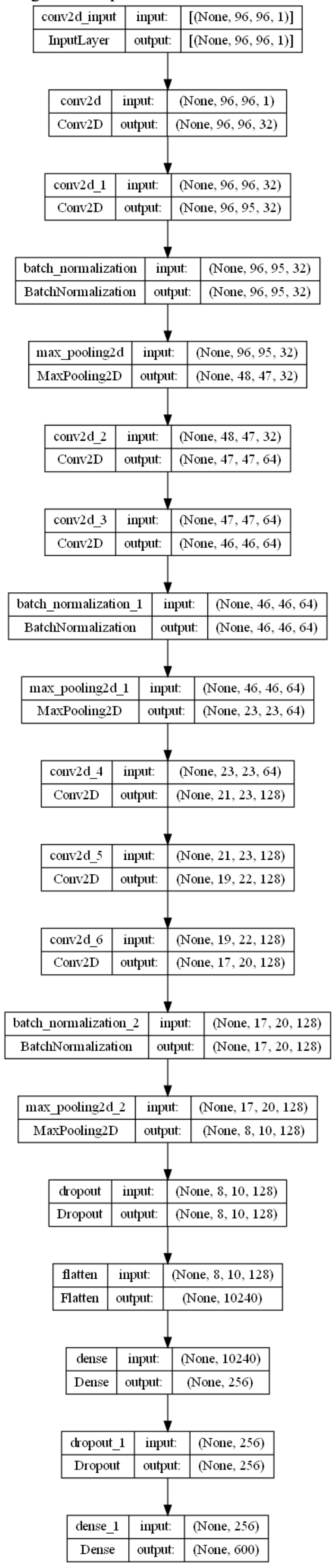


Fig. 8. Proposed model architecture.



conv2d_input	input:	[(None, 96, 96, 1)]
InputLayer	output:	[(None, 96, 96, 1)]
conv2d	input:	(None, 96, 96, 1)
Conv2D	output:	(None, 96, 96, 32)
conv2d_1	input:	(None, 96, 96, 32)
Conv2D	output:	(None, 96, 95, 32)
batch_normalization	input:	(None, 96, 95, 32)
BatchNormalization	output:	(None, 96, 95, 32)
max_pooling2d	input:	(None, 96, 95, 32)
MaxPooling2D	output:	(None, 48, 47, 32)
conv2d_2	input:	(None, 48, 47, 32)
Conv2D	output:	(None, 47, 47, 64)
conv2d_3	input:	(None, 47, 47, 64)
Conv2D	output:	(None, 46, 46, 64)
batch_normalization_1	input:	(None, 46, 46, 64)
BatchNormalization	output:	(None, 46, 46, 64)
max_pooling2d_1	input:	(None, 46, 46, 64)
MaxPooling2D	output:	(None, 23, 23, 64)
conv2d_4	input:	(None, 23, 23, 64)
Conv2D	output:	(None, 21, 23, 128)
conv2d_5	input:	(None, 21, 23, 128)
Conv2D	output:	(None, 19, 22, 128)
conv2d_6	input:	(None, 19, 22, 128)
Conv2D	output:	(None, 17, 20, 128)
batch_normalization_2	input:	(None, 17, 20, 128)
BatchNormalization	output:	(None, 17, 20, 128)
max_pooling2d_2	input:	(None, 17, 20, 128)
MaxPooling2D	output:	(None, 8, 10, 128)
dropout	input:	(None, 8, 10, 128)
Dropout	output:	(None, 8, 10, 128)
flatten	input:	(None, 8, 10, 128)
Flatten	output:	(None, 10240)
dense	input:	(None, 10240)
Dense	output:	(None, 256)
dropout_1	input:	(None, 256)
Dropout	output:	(None, 256)
dense_1	input:	(None, 256)
Dense	output:	(None, 600)

Fig. 10. Proposed model, Medium and Hard images validation loss for ID detection.

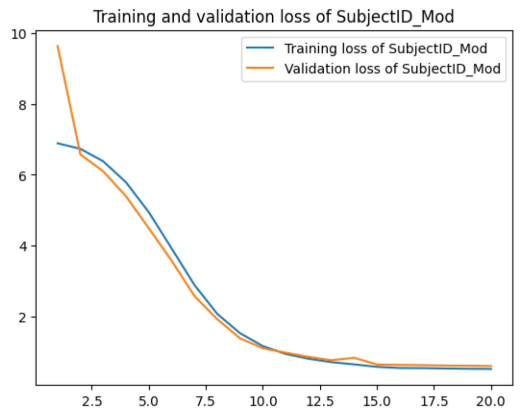


Fig. 11. Proposed model, Medium and Hard images validation accuracy for finger detection.

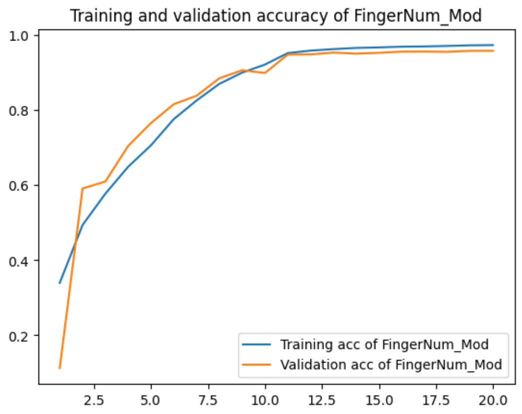
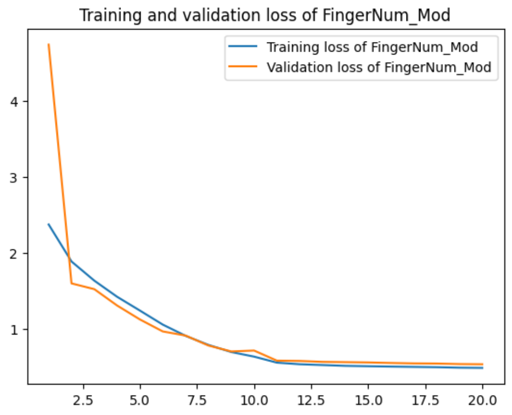


Fig. 12. Proposed model, Medium and Hard images validation loss for finger detection.


Training and validation loss of FingerNum_Mod
Training loss of FingerNum_Mod
Validation loss of FingerNum_Mod