Hair Type Classification with Convolutional Neural Networks

Advanced Intelligent Systems (CS-ELEC1A) Laboratory Activity

Mala-ay, Amiel Christian E.

I. INTRODUCTION

Image classification is one of the most prominent applications of machine learning models in modern society and a cornerstone of computer vision. Having found usage in several important fields, like healthcare, manufacturing, commerce, and security, it has cemented itself as a significant component of modern society. Image classification algorithms are used for a wide variety of applications, like autonomous driving, medical image analysis, and facial recognition, among others.

Convolutional neural networks (CNNs) are the foremost machine learning models for image classification. CNNs are feed-forward neural networks that learn feature engineering via filter optimization. They rely on matrices known as kernels, which are used to perform convolution on an input matrix, such as an image, in order to produce a feature map. The kernels of a convolutional layer are collectively known as the filter, and they capture patterns and features at different locations in the image. These filters are learned throughout the process, with their weights being optimized through the forward and backward pass.

The architecture of a common convolutional neural network is defined by convolutional layers, but there is more to CNNs than convolutional layers. Pooling layers, for instance, reduce the resolution of the preceding matrix, reducing both computational costs and the model's sensitivity to slight variances in features. Activation functions are also commonly used to introduce non-linearity into the convolutional layers.

CNNs are versatile in that one can have a simple architecture consisting of only a few layers or an incredibly deep and complex architecture designed for large amounts of data. This versatility makes it so that they can be tailored to fit the specific requirements of any given problem. In this paper, the problem to be tackled is hair type classification using a convolutional neural network model.

II. METHODOLOGY

The data set used by the model consists of 981 images containing three types of hair: curly hair (332 data points), wavy hair (331 data points), and straight hair (318 data points). This data set was obtained after removing all images of file types that the TensorFlow library does not accept from the original data set.

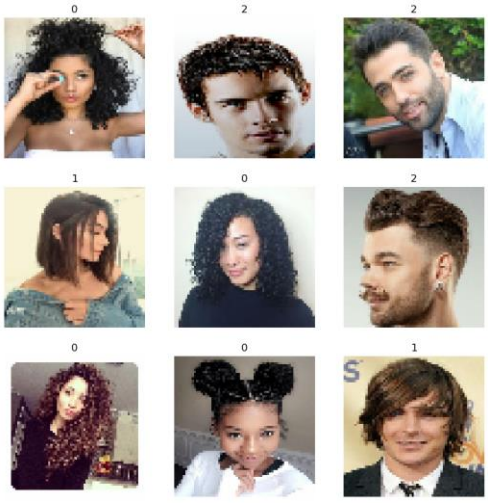

Figure 2.1 A preview of the images loaded into the initial training set

Afterwards, the images in the data set were resized to 64x64 pixels and fed to the model. The data was split into training and validation sets, with 80% of the data set (785 data points), being allocated to the training set and the remaining 20% (196 data points) being set aside for validation.

A baseline architecture was provided for the model. The images in the training and testing set were resized into 64x64 pixels. Rescaling was also applied to the images in order to normalize their pixel values, which could initially range between 0 and 255, to a range between 0 and 1.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling (Rescaling)       (None, 64, 64, 3)         0

 conv2d (Conv2D)             (None, 49, 49, 4)         3076

 activation (Activation)     (None, 49, 49, 4)         0

 conv2d_1 (Conv2D)           (None, 42, 42, 8)         2056

 activation_1 (Activation)   (None, 42, 42, 8)         0

 conv2d_2 (Conv2D)           (None, 39, 39, 16)        2064

 activation_2 (Activation)   (None, 39, 39, 16)        0

 global_average_pooling2d (  (None, 16)                0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 64)                1088

 activation_3 (Activation)   (None, 64)                0

 dense_1 (Dense)             (None, 3)                 195

 activation_4 (Activation)   (None, 3)                 0

=================================================================
Total params: 8479 (33.12 KB)
Trainable params: 8479 (33.12 KB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 2.2 Summary of the initial model's architecture

The rescaling layer is followed by two convolutional layers activated with the rectified linear unit (ReLU) function. These are then followed by a global average pooling layer for reducing the dimensions of the feature map to a single value. This is then followed by two dense layers, the first containing 64 neurons and activated with the ReLU function, and the second containing 3 neurons and activated by the softmax function. The last dense layer is responsible for classification, hence the 3 neurons.

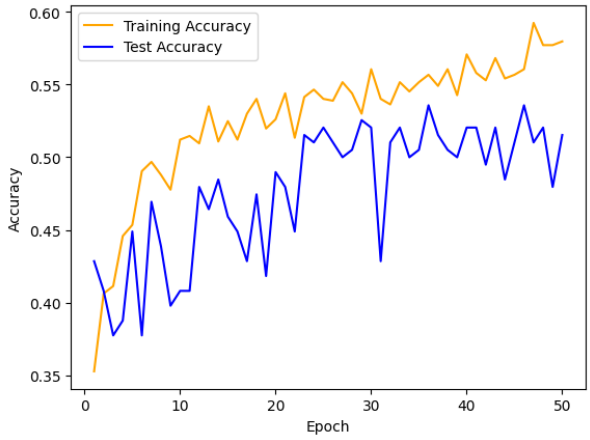

Figure 2.3 Initial model's performance with regard to accuracy

The performance of this model left much to be desired in terms of its ability to make accurate classifications. As such, there was a need to make major changes to the architecture and other parts of the model.

This was done by adding not only convolutional and pooling layers, but batch normalization, flattening, and dropout layers as well. Changes were also made to the model's hyperparameters, like the learning rate and batch size, among others.

Additional data preprocessing techniques were also used in order to address the relatively small size of the data set. These include random reorientation, rotation, and zooming, among others.

Other methods were used to help with training. Early stopping was used for ending the training process once the model's performance has stopped improving. Learning rate scheduling was also used for adjusting the model's learning rate over time. Automated hyperparameter tuning using KerasTuner was utilized to help with adjusting the model's hyperparameters.

The model was trained using a computer with an Intel Core i7-8750H processor, an NVIDIA GeForce GTX 1050 Ti graphics card, and 16 gigabytes of RAM. These hardware specifications, while by no means inadequate, place certain constraints on the complexity of the model. Model architectures that are too deep and complex might take too long to train.

This is not of grave concern, however, given that the data set provided for the model is relatively small. Overly deep and complex models do not generalize well when trained on too little data. For these reasons, it was decided that the model's architecture should be kept relatively simple.

III. EXPERIMENTS

In order to stop the training process when the model's performance in the middle of training seems less than promising, early stopping was utilized. This sped up the experimentation process by indicating much earlier that a given iteration of the model is not learning well, with validation loss being used as the metric for stagnation.

To compare trials, it was decided that a model iteration's performance at the last epoch would be analyzed and compared to that of previous iterations, irrespective of whether the epochs were completed or halted by early stopping. For consistency, the initial model was trained again with early stopping, where it stopped at epoch 18, providing the following metrics as a baseline.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.9703	0.4994	1.1651	0.4031

Figure 3.1 Initial model's final performance after halting at epoch 18

3.1 Adjusting Image Size

The size of the images in the initial model was set to 64x64 pixels. While this allowed for fast training times, this meant that the model had access to a low-resolution version of the training images, which could lead to important features in the images being obscured.

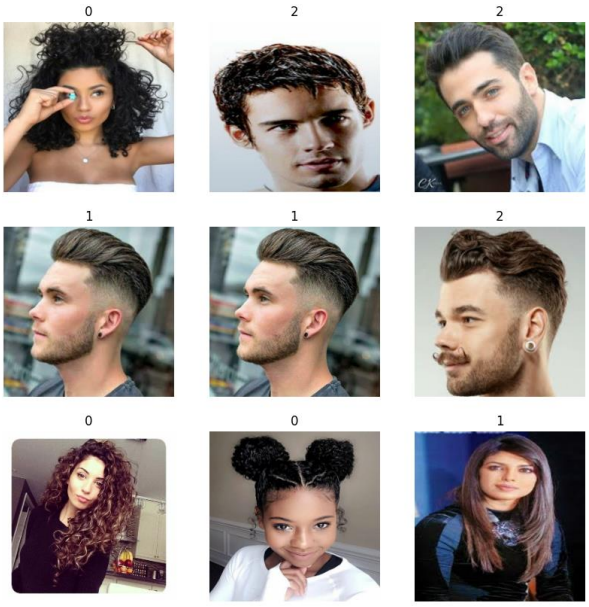

Figure 3.1.1 A preview of the training set with image sizes set to 256x256 pixels

As such, the image size was multiplied four-fold to 256x256. However, the initial model was not constructed with such a large image size in mind. Retaining the original architecture would result in the model taking too long to train. As such, the convolutional layers' hyperparameters were adjusted. These adjustments were largely arbitrary and only served to decrease the time for the model to train using the new input size.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.9096 (-0.0607)	0.5529 (+0.0535)	1.0670 (-0.0981)	0.4388 (+0.0357)

Figure 3.1.2 Results after resizing to 256x256 pixels

Even with arbitrary and poorly conceived adjustments to the original architecture, the model's performance saw improvements on all fronts, albeit not by much. Regardless of these results, however, expanding the image size allows for more flexibility with respect to the architecture. A larger number of pixels allows for larger strides and layer dimensions. This decision will be consequential for making modifications to the architecture.

3.2 Data Augmentation

With only 981 images, the provided data set is unfortunately quite small. The size of the data set significantly hampers the ability of the model to generalize well. To remedy this, data augmentation was performed on the data set.

For this purpose, Keras' ImageDataGenerator utility was used as the data loader. This utility allows for simpler application of data augmentations. However, it is critical to be careful in choosing which augmentations to apply to the data and the degree of frequency and intensity with which these augmentations are applied.

The augmentations that were applied to the training set included vertical and horizontal reorientation, zooming, rotation, shearing, and shifting, with varying results. Over the course of several trials, the intensity and frequency of these augmentations were modified.

The most interesting phenomenon observed over the course of experimenting with augmentations is the model's validation accuracy being higher than its accuracy on the training data. This could be because the augmentations were making it too hard for the model to classify the training data, which makes it easier for the model to classify non-augmented data like that found in the validation set.

Loss	Accuracy	Validation Loss	Validation Accuracy
1.0985 (+0.1889)	0.3363 (-0.2166)	1.0984 (+0.0314)	0.3385 (-0.1003)

Figure 3.2.1 Best final results after applying data augmentation

The results of experimenting with data augmentation were less than promising. Not only was there a decline on all fronts, but there was also the peculiarity of the validation accuracy being higher than the training accuracy. In the end, it was decided that the model would be better off without any augmentations to its training set.

3.3 Hyperparameter Tuning

Experimentation with learning rates was performed on the model. However, such experimentations require adjustments to the epochs as well, as the model's

learning rate and the number of epochs for which it is trained go hand in hand.

Numerous experiments were conducted with different epoch numbers and learnings rates, nearly all of which had negligible or adverse effects on performance. However, experimenting with learning rate scheduling resulted in much better results.

After much experimentation, it was decided that the learning rate would be set to 0.001, declining by a factor of 0.1 after every 12 epochs, and the number of epochs was set to 20. While this is a significant departure from the initial epoch amount, setting the learning rate and epoch amount as such yielded positive results.

Note: this follows from the results in Figure 3.1.2, not 3.2.1

Loss	Accuracy	Validation Loss	Validation Accuracy
0.8520 (-0.0576)	0.6089 (+0.0560)	0.9984 (-0.0686)	0.5077 (+0.0689)

Figure 3.3.1 Results after applying learning rate scheduling and setting epoch amount to 20

Experimentation with the batch size was also performed. Lower batch sizes helped with the system's relatively low memory, but it also helped with the model's performance by allowing it to converge faster.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.3830 (-0.4690)	0.8522 (+0.2433)	1.1007 (+0.1023)	0.5692 (+0.0615)

Figure 3.3.2 Results after setting batch size to 16

In the end, the batch size for the model was set to 16. The increase in validation loss by doing so is negligible and counteracted by the benefits observed in the other metrics.

It can be observed as early as now that the training accuracy and validation accuracy of the model are quite far apart from each other. This is a sure sign of overfitting, which will have to be addressed.

3.4 Architecture Modification

3.4.1 Convolutional and Pooling Layers

Additional convolutional layers were added to the model. After several experiments, it was decided that only one convolutional layer would be added, resulting in a total of four convolutional layers.

Resizing the images to 256x256 pixels allowed for larger kernel sizes and bigger strides. The configuration for the convolutional layers that showed the most promise after several trials is shown below.

- 3×3 convolutional layer with a stride of 2 and 16 filters
- 15×15 convolutional layer with a stride of 2 and 32 filters
- 5×5 convolutional layer with a stride of 1 and 64 filters
- 5×5 convolutional layer with a stride of 1 and 128 filters

Odd-numbered kernels were used in accordance with common practice, as such kernels have a valid coordinate at their center. Strides greater than 1 were used simply to improve training times.

While pooling was applied in the initial model, it was only global pooling, where data is aggregated globally across the feature map for use by the fully connected layers. Local pooling was added to the architecture, specifically max pooling, where the feature map's most prominent features within a predefined area are obtained by getting the maximum value. 2×2 max pooling layers with strides of 2 were added after the first two convolutional layers.

The global average pooling layer was also replaced with global max pooling, as doing so was found to be beneficial to the model's performance.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.4235 (+0.0405)	0.8280 (-0.0242)	1.0981 (-0.0026)	0.5949 (+0.0257)

Figure 3.4.1.1 Results after making changes to convolutional and pooling layers

While there is a decline in the model's training performance, its performance on unseen data also improved. However, there is still much work to be done to address the issue of overfitting.

3.4.2 Batch Normalization

Batch normalization is a method that was introduced relatively recently. It is used for increasing the speed and stability of a convolutional neural network by normalizing the inputs of a layer.

In following with the traditional ResNet architecture, the batch normalization layer was placed before the application of non-linearity with ReLU. For this model, batch normalization was applied only on the first layer, as this was the only place in which it had a significant impact.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.5733 (+0.1498)	0.7783 (-0.0497)	0.8654 (-0.2327)	0.6513 (+0.0564)

Figure 3.4.2.1 Results after adding batch normalization layer

Adding batch normalization negatively impacted the model's training performance, but it did reduce overfitting, as can be seen in the improvement in its validation performance.

3.4.3 Flattening

In addition to the global pooling layer, another option for converting the output of a series of convolution and pooling layers is to flatten it into a one-dimensional vector. While in other models, global pooling and flattening are used exclusively, for this model, flattening was used together with global pooling, yielding positive results.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.2402 (-0.3331)	0.9236 (+0.1453)	0.6798 (-0.1856)	0.7487 (+0.0974)

Figure 3.4.3.1 Results after adding flattening

Adding flattening increased the model's performance in all respects. However, there is still a clear overfitting problem that needs to be addressed.

3.4.4 Dense Layers and Dropout

The fully-connected layers at the end of the convolutional neural network were also modified. The first dense layer's neuron count was cut in half from 64 to 32.

In addition to this, dropout was used to reduce overfitting. Dropout removes a fraction of neurons in a layer during each forward and backward pass, introducing variability in the training process.

Loss	Accuracy	Validation Loss	Validation Accuracy
0.5370 (+0.2968)	0.7567 (-0.1669)	0.6855 (+0.0057)	0.7333 (-0.0156)

Figure 3.4.4.1 Results after modifying dense layers and adding dropout

While it may seem at first glance as though the model was adversely affected by these modifications, the close proximity of the model's training accuracy and validation accuracy shows that the overfitting problem has finally been overcome.

IV. RESULTS AND DISCUSSION

4.1 Final Architecture

```
Model: "sequential_20"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_20 (Rescaling)    (None, 256, 256, 3)       0

 conv2d_77 (Conv2D)          (None, 127, 127, 16)      448

 batch_normalization_18 (Ba  (None, 127, 127, 16)      64
 tchNormalization)

 activation_117 (Activation  (None, 127, 127, 16)      0
 )

 max_pooling2d_40 (MaxPooli  (None, 63, 63, 16)        0
 ng2D)

 conv2d_78 (Conv2D)          (None, 25, 25, 32)        115232

 activation_118 (Activation  (None, 25, 25, 32)        0
 )

 max_pooling2d_41 (MaxPooli  (None, 12, 12, 32)        0
 ng2D)

 conv2d_79 (Conv2D)          (None, 8, 8, 64)          51264

 activation_119 (Activation  (None, 8, 8, 64)          0
 )

 conv2d_80 (Conv2D)          (None, 4, 4, 128)         204928

 activation_120 (Activation  (None, 4, 4, 128)         0
 )

 global_max_pooling2d_20 (G  (None, 128)               0
 lobalMaxPooling2D)

 flatten_20 (Flatten)        (None, 128)               0

 dense_40 (Dense)            (None, 32)                4128

 activation_121 (Activation  (None, 32)                0
 )

 dropout_36 (Dropout)        (None, 32)                0

 dense_41 (Dense)            (None, 3)                 99

 activation_122 (Activation  (None, 3)                 0
 )

=================================================================
Total params: 376163 (1.43 MB)
Trainable params: 376131 (1.43 MB)
Non-trainable params: 32 (128.00 Byte)
```

Figure 4.1.1 A summary of the model's final architecture

The final architecture consists of 4 convolutional layers using the ReLU function, the first two of which are followed by max pooling layers. The first convolutional layer is also followed by a batch normalization layer. To reduce the output of these layers to a one-dimensional vector, global max pooling and flattening were used. These layers were then followed by the two dense layers that are responsible for classification.

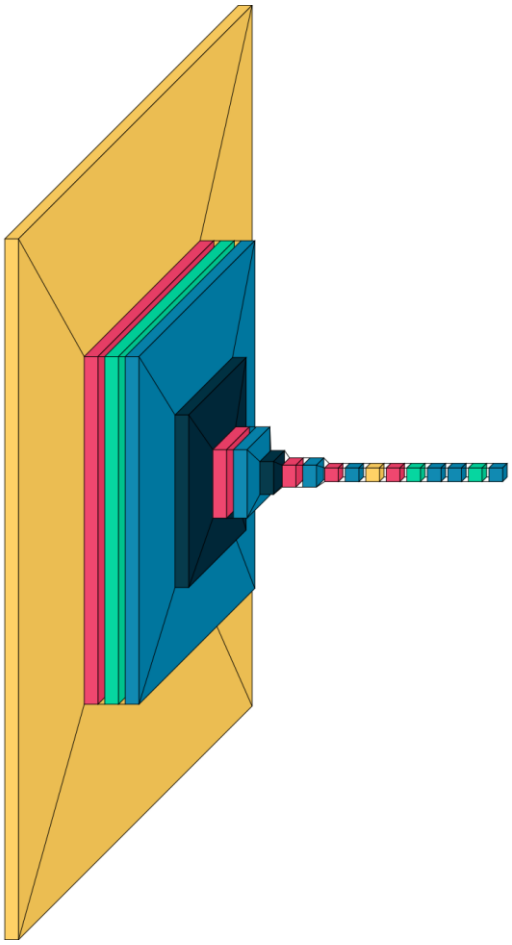

Figure 4.1.1 A visual representation of the model's final architecture

The resulting model was trained over 20 epochs, which it was able to accomplish fully without stopping due to test loss stagnation.

4.2 Model Loss

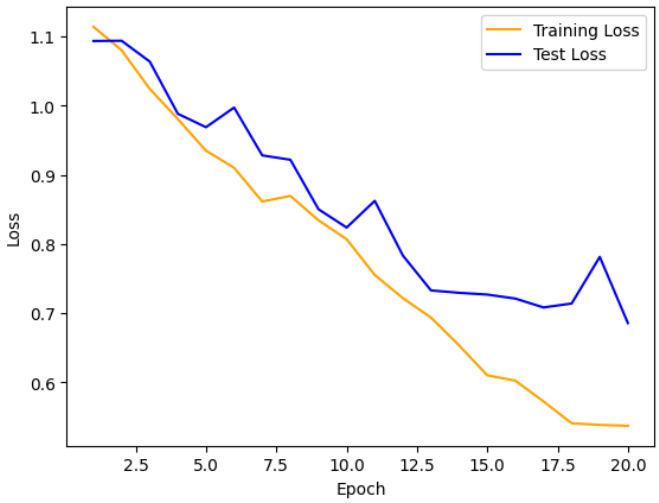

Figure 4.2.1 A graph of the model's training and testing loss over training

The model's training loss over time ranges from 0.53 to 1.11. Its testing loss, on the other hand, ranges from 0.68 to 1.09. Both of these lows are attained at the very last epoch. There is something to be said about the smoothness of the loss's decline over time, which indicates that the model was converging well and learning stably.

4.3 Model Accuracy

Figure 4.3.1 A graph of the model's training and testing accuracy over training

The model's training accuracy over time ranges from 0.36 to 0.76. Its testing accuracy, on the other hand, ranges from 0.31 to 0.73.

In total, the metrics of the final model are satisfactory. While far from perfect, the model's accuracy on unseen data falls within the threshold of what can be considered a good score (≥70%).

V. CONCLUSION AND RECOMMENDATIONS

5.1 Summary of Findings

With a final accuracy of 73.33% on unseen data, which falls within the threshold of what can be considered a good accuracy, it can be said that the model, while not quite perfect, can classify hair types as straight, wavy, or curly with a reasonable degree of correctness. This is a perfectly adequate score, even if it is not all that close to perfect.

The model's tendency to overfit in the early stages of experimentation, even with relatively simple architectures, served as evidence that the data set is simply too small. The overfitting problem was fortunately resolved by making adjustments to the layers of the model and using methods such as batch normalization and dropout, which goes to show just

how important such adjustments and methods are in constructing a convolutional neural network.

There is reason to believe that the data set's small size, with only 981 images that can be accepted by Keras, could be blamed for the model's shortcomings. While the model's performance on unseen data was satisfactory, it is difficult to make assessments on the model's abilities simply because the training set, or indeed the data set as a whole, is just too small. While its performance on the validation data set is acceptable, it does not stand to reason that it is reflective of its performance on unseen data in larger and more diverse data sets. Moreover, the data set, in addition to being small, seems to be inadequately treated, with the presence of mislabeled images and duplicates having been observed. These flaws of the data set are likely to have been detrimental to the performance of the model as well.

5.2 Recommendations

While the model's performance is adequate, there are quite a few improvements that could be made. Experimenting with the model's architecture by adding or removing convolutional, pooling, or dense layers might be worth exploring. Applying dropout more aggressively, for instance, could be something that could be experimented with. Batch normalization was not sufficiently explored for this model, so it is also worth experimenting with.

While the model does have some shortcomings, it is necessary to note that the data set on which it was trained is relatively small, consisting of only a little under 1000 images. A convolutional neural network trained on such a small data set cannot be expected to generalize particularly well or be particularly accurate in its classifications.

The small size of the data set also places constraints on the complexity of the model's architecture, as deep and overly complex models have a tendency to overfit on small data sets. Future explorations with the provided data set would be best done with relatively simple and shallow architectures, like that of the model featured in this paper, instead of deeper and more complex architectures which require larger data sets than the one used by the model.

The data set could also benefit from more intensive preprocessing techniques. It was observed that duplicates and mislabeled images existed within the data set. Such imperfections in the data set could be part of the reason for the model's poor performance. However, manually cleaning a data set of 900 images is a time and labor-intensive task, which is why it was not accomplished. Future undertakings with this data set could see better results by treating the data set for these imperfections.

VI. REFERENCES

Devron AI. (n.d.). *How to choose the optimal kernel size? | Devron*. Www.devron.ai. Retrieved November 27, 2023, from https://www.devron.ai/kbase/how-to-choose-the-optimal-kernel-size

Kandel, I., & Castelli, M. (2020). The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT Express*, *6*(4). https://doi.org/10.1016/j.icte.2020.04.010