Non-Linear Dimensionality Reduction using Auto-Encoder for Optimized Malaria Infected Blood Cell Classifier

Aayush Dhakal

NON-LINEAR DIMENSIONALITY REDUCTION USING AUTO-ENCODER FOR OPTIMIZED MALARIA INFECTED BLOOD CELL CLASSIFIER

by

Aayush Dhakal

A thesis submitted to the faculty of The University of Mississippi in partial fulfillment of the requirements of the Sally McDonnell Barksdale Honors College.

Oxford, MS

May 2021

Approved by:

Advisor: Professor Yixin Chen

Reader: Professor Feng Wang

Reader: Professor Phillip Rhodes

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor Dr. Yixin Chen for being a mentor to me and providing his support in the course of this project. I am also thankful to my readers Dr. Feng Wang and Dr. Philip Rhodes who have provided strong support throughout my undergraduate studies.

I am also very thankful towards Sally McDonnell Barksdale Honors College and Department of Computer and Information Science for providing numerous opportunities to get involved in exciting research and providing adequate resources to follow through with it.

I would also like to thank the Office of Research and Sponsored Programs for sponsoring this project. This research project was funded through the University of Mississippi-Summer Undergraduate Research Fellowship (UM-SURF). I am thankful for their contribution to make this research project a success.

ABSTRACT

Neural Networks have been widely used in the problem of Medical Image Analysis. However, when dealing with large images, deep networks easily exhaust computer resources, which in turn hinders training. This paper shows the efficacy of using Auto-Encoders as a dimensionality reduction tool to increase the efficiency of a Malaria Infected Blood Cell Image classifier. We show that using an autoencoder, we can reduce the dimensionality of large blood cell images effectively such that the features in the new space retain all the essential information from the original input. Then we show that the new features obtained from the autoencoder can be used to train a classifier while maintaining the same accuracy. Using a Convolutional autoencoder with a Convolutional Neural Network(CNN) for malaria infected blood cell classification, gives us a significantly smaller model compared to a vanilla CNN model which performs similar in terms of accuracy.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

NN	Neural Network
DNN	Deep Neural Network
CNN	Convolutional Neural Network
FCNN	Fully Connected Neural Network
CAE	Convolutional Auto-Encoder

CHAPTER 1

INTRODUCTION

In recent years, Neural Networks have taken over the field of medical image analysis and gained success in fields of image-based diagnosis, disease prognosis and risk assessment. Neural Networks are not restricted by the current knowledge of disease related diagnosis pattern and thus have a high diagnostic accuracy [1]. Researchers are using deeper and deeper neural networks to increase the accuracy of image classification. This is because research has shown that deep neural networks are able to drastically increase the accuracy of image classification by learning to exploit complex nonlinear structures of the images [2]. However, deeper network exponentially increase the number of parameters in the network. Such models with very high number of parameters require huge computational resources for training and storing purposes.

One way to exponentially decrease the number of parameters is to reduce the size of the input feature vector. The number of trainable parameters in a neural network is a function of the network architecture and the size of the input vector. Thus, reducing the size of the input allows training a very deep neural network with much lower number of parameters.

Unsupervised learning methods are used to extract useful features from unlabeled data by removing redundancies and preserving only the essential aspects of the input data[3]. In the context of neural networks, autoencoders use unsupervised learning to map the input data into a bottleneck layer which extracts essential features from the input. Using this structure of a bottleneck layer, autoencoder can be used as a dimensionality reduction model which reduces the number of input dimensions to the output size of the bottleneck layer[4].

In this paper, we propose the use of Convolutional Auto-Encoder to reduce the dimensionality of the Blood Cell Images. We use both reduced input and original input on a CNN for classification. We show that we achieve similar accuracy when using either inputs but the CNN with the reduced input is a smaller model that requires significantly less storage space.

CHAPTER 2

CONCEPTS

2.1 Dimensionality Reduction

Dimensionality refers to the number of features in a given data. The size of a neural network model is directly proportional to the size of the input feature vector. Thus, it is inconvenient to use high dimensional data to train neural networks due to concerns of scalability. The size of a model can increase exponentially with larger number of input features. Thus, in this project, we will convert the high dimensional blood cell image data into lower dimensional data by extracting only the key features from the images. This can be done by mapping the original input to a new space with less number of features, such that it preserve the global characteristics of the input. If these features in the new space accurately preserve the important information from the original input images, they can be used to learn a classifier using a much smaller neural network model. To carry out this feature extraction, we will perform non-linear dimensionality reduction on the images using an auto-encoder.

2.2 Neural Network

Neural Networks are models that are able to learn highly complex and nonlinear decision boundaries purely from the input data. These networks were initially inspired by the biological neural systems of the human brain [5]. A simple Neural Network takes a set of inputs (X) and their corresponding outputs (y), and then tries to learn a function that can map X to y:

$$F(X) = \hat{y} \approx y$$

The model tries to learn this function by minimizing a cost function over a number of iterations. The cost function is a value that represents the distance of the model's prediction from the correct values. By using an optimization algorithm like gradient descent, after a number of iterations, the model can learn complex functions that minimize the cost and produce a good mapping from X to y.

The most basic neural network is a perceptron. It has an input layer, which represents the input features, and an output layer [5]. The output layer is a node that performs a mathematical computation:

$$\hat{y} = a\,(WX + b)$$

Where, W is the weight, b is a bias and a is an activation function.

This single node is only capable of learning linear functions, and therefore does not allow the network to solve nonlinear problems. However, we can stack these nodes on top of each other making a layer and further use multiple layers between the input and output layer, known as hidden layers. Such an architecture where we use multiple hidden layers between the input and output layer to learn complex non-linear functions is commonly known as Deep Neural Networks (DNN). Outputs from each layer in a DNN serve as inputs to the next layer. By passing outputs from one layer as inputs to the next, the degree of complexity of the learned function increases with deeper layers. Thus, Deep Neural Networks, enable us to learn highly complex nonlinear boundaries, which would not be possible using a perceptron.

By changing the operation at the nodes, or adjusting the architecture of the layers, these networks can be tuned to facilitate different purposes. In this project, we will use a Convolutional Neural

Network (CNN) to build a classifier for the Blood Cell Images and a Convolutional Auto-Encoder (CAE) to reduce the dimensionality of the Blood Cell Images.

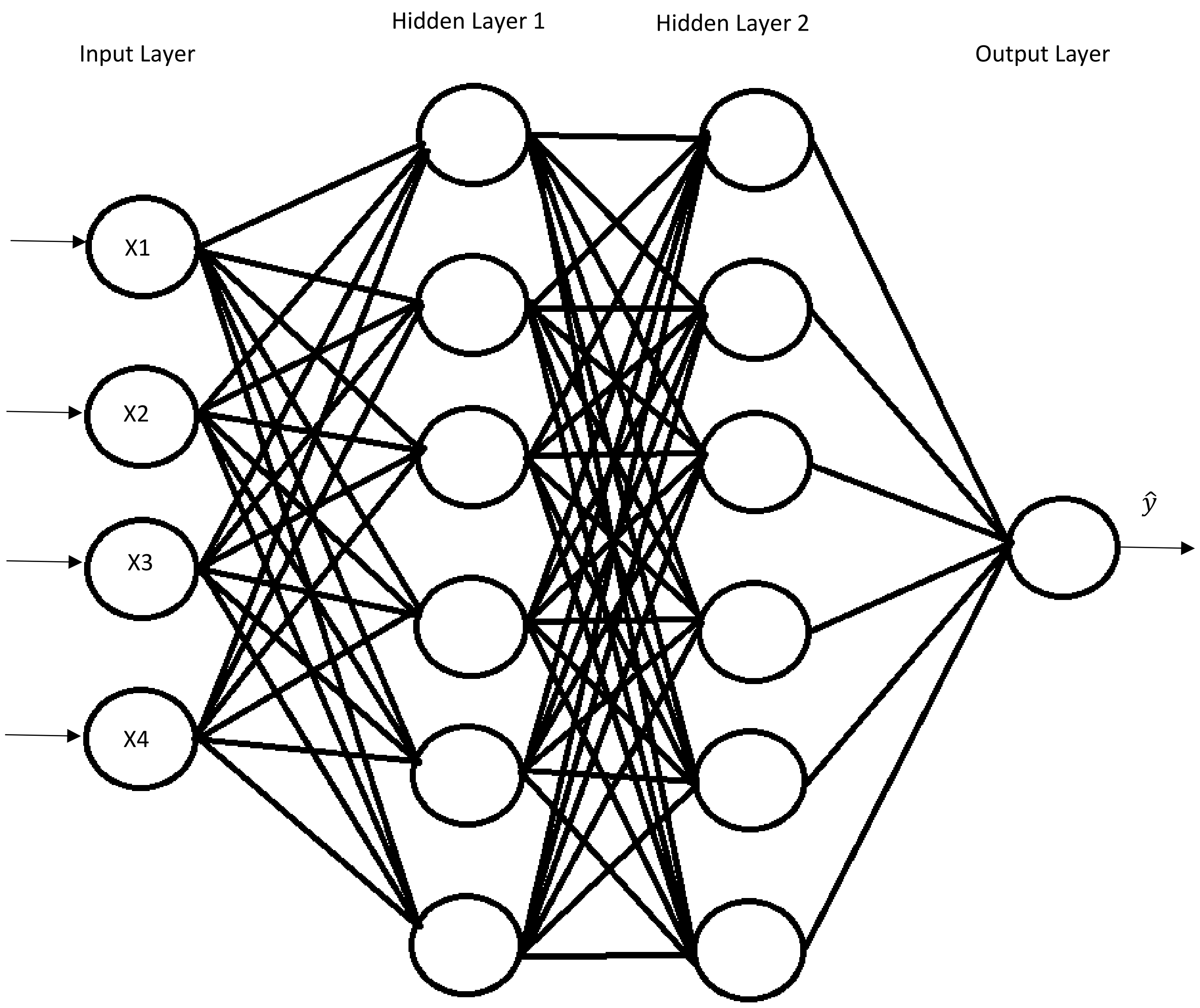


Fig. 2.1. A Simple Neural Network with 2 hidden layers

2.2.1 Convolutional Neural Network

Traditional Neural Network architectures (such as the fully connected NN described above), are not well suited for handling image data. As each pixel of an image acts as a feature, image data usually have a very high number of input features. For example, even if we take a small image of size 32x32x3 and use a perceptron, we still have 32x32x3 or 3,072 parameters to train. However, it is unlikely that a single node perceptron can extract useful features from such data [6]. Adding more nodes, however, will exponentially increase the number of parameters in this network.

Instead of connecting each node with each feature, a better way to deal with such data is to connect each neuron to a local region in the image. For example, one neuron activation may be dependent on a 3x3 region of an image. Furthermore, the local connection weights can be fixed for the entire neurons of the next layer [6]. By doing so, we drastically reduce the number of parameters; in a FCNN, to map the input of a 64x64 image to a layer with 64x64 neurons, we would need 16,777,216 (64x64x64x64) parameters, whereas using this new approach, if we use a 7x7 local region, we only need 49 parameters. In this particular example, this process can be viewed as sliding a window of size 7x7 across the image such that we perform the convolution operation for each pixel and map the output to the corresponding space in the next layer. The convolution operation over a pixel is defined as the sum of the product of each overlapping tensor values within the region of the kernel window. An example of a convolution operation is provided in figure 2.2.

A CNN generally uses multiple kernels (or filters) in each layer, such that each filter learns to extract a certain feature from the input. Furthermore, the deeper layers of the CNN learn abstract features of the input. For example, in facial recognition, the shallow layers usually learn different edges whereas the deeper layer learns high level features like eyes, nose, face etc. A convolution layer is often followed by a nonlinearity function (like Relu) which is followed by a pooling layer.

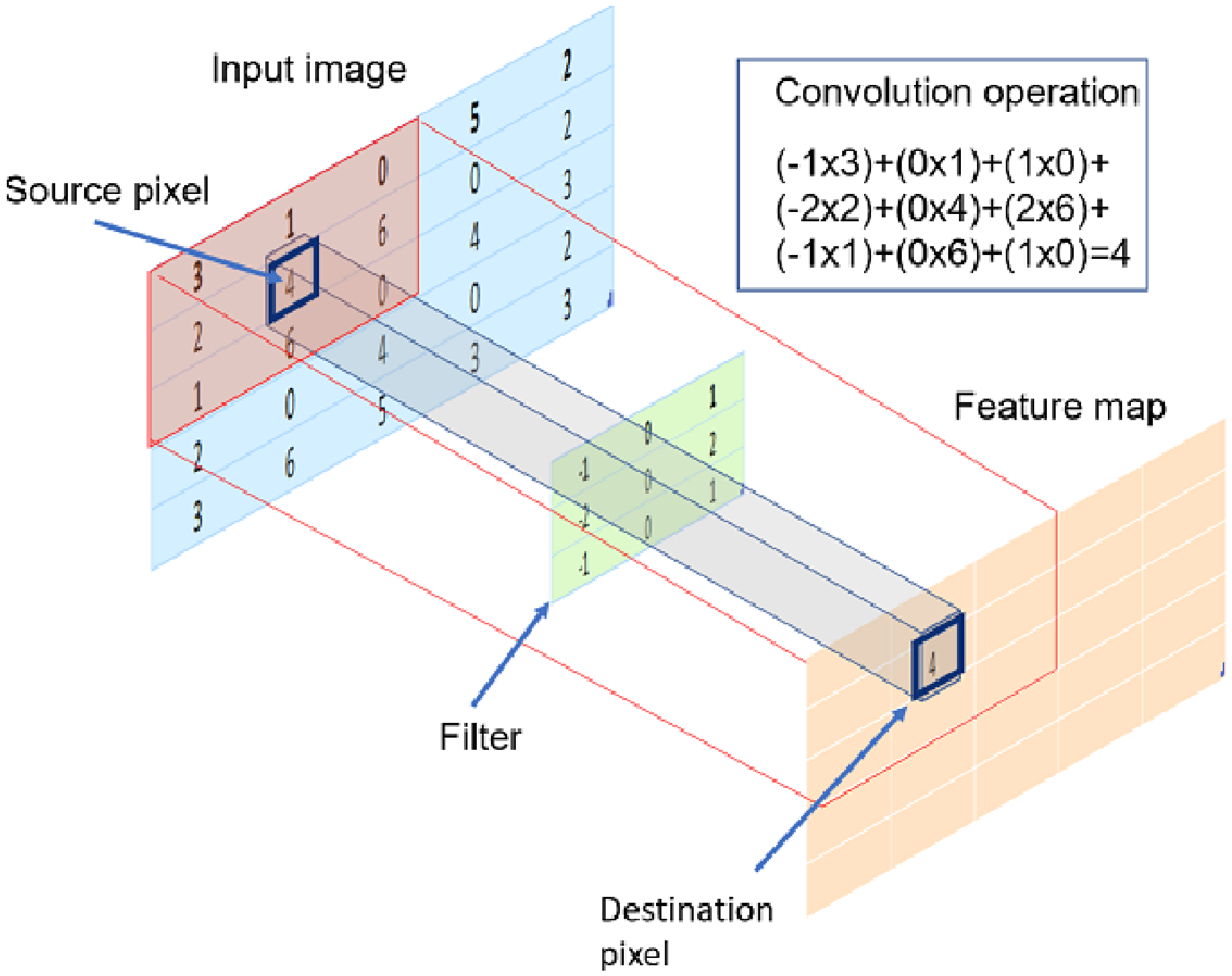


Fig 2.2. Convolution operation on a pixel over a 3x3 kernel [7]

2.2.2 Auto Encoder

Suppose we have an input x of n dimensions. An autoencoder is a special neural network that learns the function:

$$h_{W,b}(x) = y \approx x$$

The output for this network is set to be equal to the input x. Hence, the network learns this identity function by minimizing a cost function which represents the difference between the produced output y and the input x.

The output of an autoencoder is not particularly useful as it is close to the identity function of the input. However, by manipulating the hidden layers carefully, the autoencoder can be used for feature extraction and dimensionality reduction. Suppose we want to express our original input x of n dimensions in terms of lower number of dimensions m. We can use an autoencoder with a hidden layer whose output size is m. The values obtained from the activations of this layer after training will be the latent representation of the original input x [4]. This process can be viewed in three stages:

- The encoder maps the input to the latent representation $g(x) = a(Wx + b)$
- The decoder reconstructs the input by a reverse mapping of $h(g(x)) = a(W'g(x) + b')$
- The loss (L) is calculated as some distance between h and x, and the parameters are updated through backpropagation. E.g. If Mean Squared Error was used the loss function would be: $\text{Loss} = \frac{1}{2k} \sum_{i=0}^{k} (h(g(x)) - x)^2$

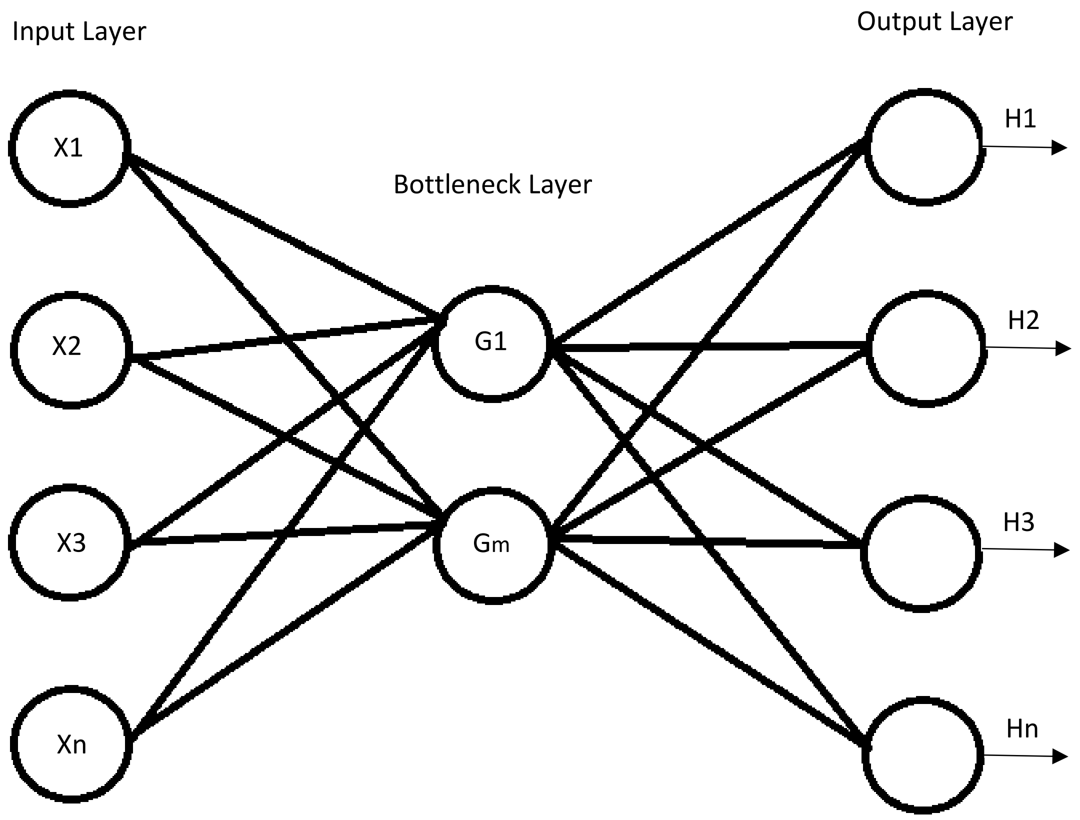


Fig. 2.3. Fully Connected Auto Encoder

CHAPTER 3

METHODOLOGY

3.1 Dataset

For this project, we used a publicly available dataset of 27,558 thin blood smear slide images from the Malaria Screener research activity [8]. The National Library of Medicine (NLM) maintains this dataset. The dataset is well balanced with equal instances of parasitized and uninfected cells. For this project, we spit the train, validation, and test set in the ratio of 8:1:1.

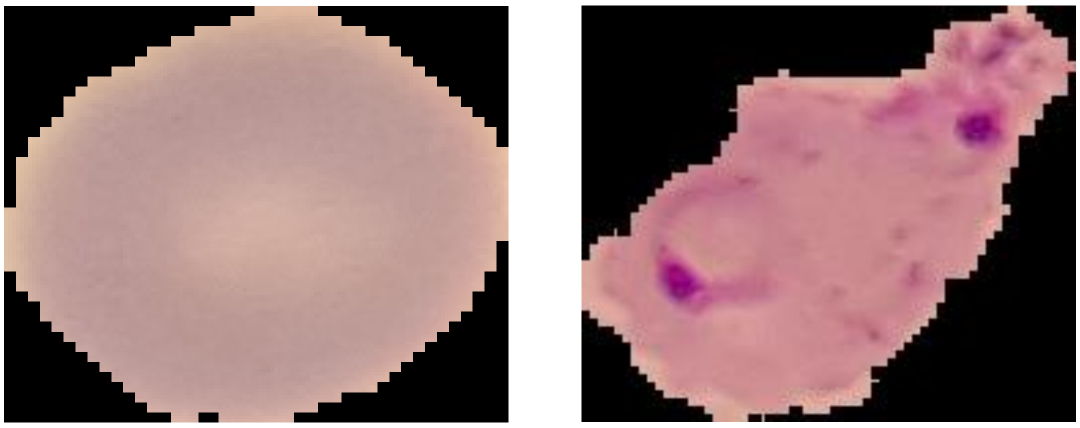


Fig 3.1. Example of Healthy Blood Cell (left) vs Infected Blood Cell (right) images from the dataset

3.2 Malaria Infected Blood Cell Classifier

In the first phase of the project, we built a convolutional neural network that functioned as a binary classifier. This classifier takes as input a blood smear image and predicts if the image belongs to parasitized class or uninfected class. This classifier acts as the benchmark for our approach. The goal was to reach this accuracy using our approach of using reduced features in the second phase.

3.2.1 Data Preprocessing

An important part of training Neural Network Model is preprocessing the data properly. Adequate preprocessing steps can help solve problems like overfitting, and incompatible dimensions in the training phase.

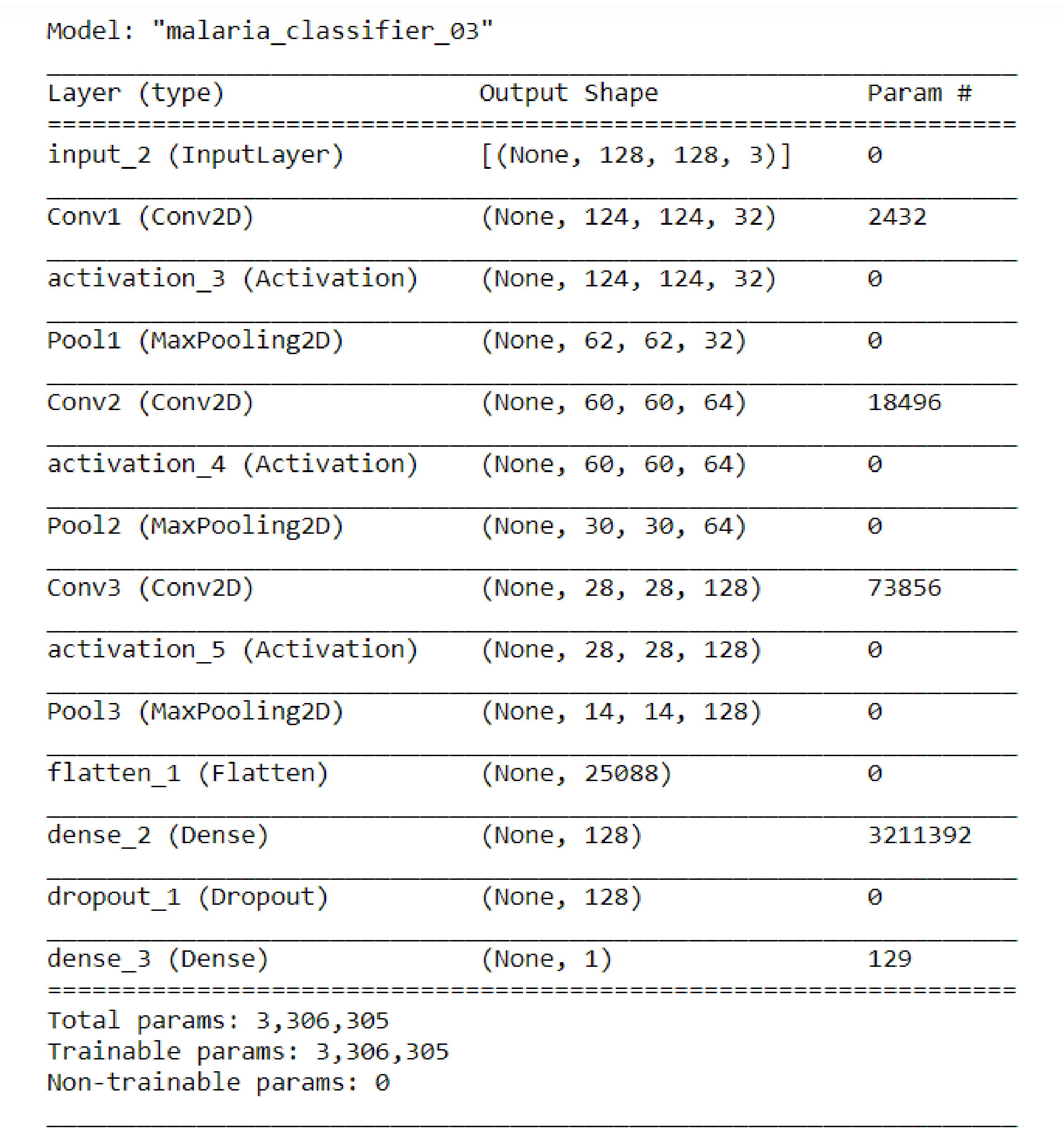
Firstly, Neural Networks cannot take inputs with varying number of input features. The images in our datasets do not have a consistent size. We calculated the mean for the x and y dimensions for all the images in the training set. The mean size for all the images was calculated as [134, 133, 3], where 3 refers to the color channels. By convention, it is best to use image sizes that are powers of 2. Therefore, we transformed all images to a size of [128,128, 3].

Another important step in data preprocessing is Data Normalization. Normalization ensures that the input features have a similar data distribution, which makes convergence fast while training. While we usually subtract the data by its mean and then divide the result by the standard deviation to normalize it, for image data, we can also simply divide the data by 255 changing the pixel range from [0-255] to [0-1]. We divided the test and train set by 255 and normalized our data.

To prevent overfitting, we used Data Augmentation in the training set. Using augmentation, we can give our model a richer variety of training data to learn from. This can prevent overfitting in the test set as the model sees a different variation of the training data in each epoch. We used rotation, height shift, width shift, random zoom, horizontal flip, and vertical flip as the augmentation methods. Each of these augmentation were performed in real time during training with a random possibility of applying any, all, or none of these augmentations on the input data.

3.2.2 CNN Classifier Architecture

The CNN classifier was built using TensorFlow and Keras libraries. There are three convolutional blocks in the model. Each convolutional block is comprised of a Convolutional Layer, which is followed by a Relu activation, which is followed by a Max Pooling Layer. The first, second and third convolutional blocks use 32, 64 and 128 filters respectively. After the third convolutional block, we have a Flatten Layer that flattens the 3-dimensional output from the convolutional block into a one-dimensional tensor. The output from this Flatten layer is passed to a Fully Connected layer (or Dense layer) with 128 neurons. Each neuron in this layer uses a Relu activation. The output from this Dense Layer is connected to a single neuron with a sigmoid activation. This single neuron behaves as our output layer. The output from this final layer is a value between 0 and 1 (because of the range of the sigmoid function). Hence, it acts as a Binary Classifier by rounding the output values to either 0 or 1.



```
Model: "malaria_classifier_03"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_2 (InputLayer)        [(None, 128, 128, 3)]     0

_________________________________________________________________
Conv1 (Conv2D)              (None, 124, 124, 32)      2432

_________________________________________________________________
activation_3 (Activation)   (None, 124, 124, 32)      0

_________________________________________________________________
Pool1 (MaxPooling2D)        (None, 62, 62, 32)        0

_________________________________________________________________
Conv2 (Conv2D)              (None, 60, 60, 64)        18496

_________________________________________________________________
activation_4 (Activation)   (None, 60, 60, 64)        0

_________________________________________________________________
Pool2 (MaxPooling2D)        (None, 30, 30, 64)        0

_________________________________________________________________
Conv3 (Conv2D)              (None, 28, 28, 128)       73856

_________________________________________________________________
activation_5 (Activation)   (None, 28, 28, 128)       0

_________________________________________________________________
Pool3 (MaxPooling2D)        (None, 14, 14, 128)       0

_________________________________________________________________
flatten_1 (Flatten)         (None, 25088)             0

_________________________________________________________________
dense_2 (Dense)             (None, 128)               3211392

_________________________________________________________________
dropout_1 (Dropout)         (None, 128)               0

_________________________________________________________________
dense_3 (Dense)             (None, 1)                 129

=================================================================
Total params: 3,306,305
Trainable params: 3,306,305
Non-trainable params: 0

_________________________________________________________________
```

Fig 3.2. Summary of the CNN Classifier Architecture

3.2.3 CNN Classifier Training

The training portion involves analyzing the performance and tuning the hyper parameters accordingly. After many trials, the most precise model was obtained when trained under the following conditions:

- Adam was used as the optimizer with a learning rate of 0.0001
- Binary cross entropy loss was used to computer loss
- A Dropout Layer with 25% dropout rate was used in the first Dense Layer
- Batch Size of 64 was used
- Early Stopping was used with a patience of 2 epochs
- The training was adjusted based on the performance on the validation set. Hence, the model had not seen the test set during training.

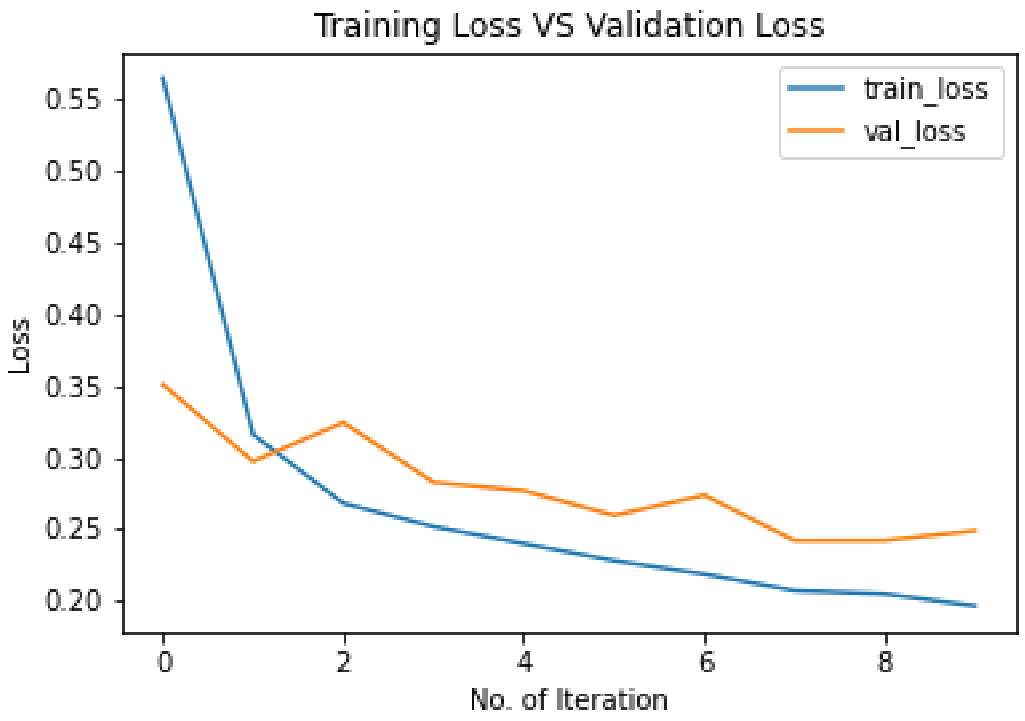


Fig. 3.3. Progression of Training and Validation Loss during training of the CNN classifier

3.3 Auto-Encoder for Dimensionality Reduction

In the second phase, we built an Auto-Encoder and trained the blood cell images to retrieve a latent representation of the data with reduced features. The goal of this phase was to encode the original input of size [128, 128, 3] into a tensor of smaller size while preserving all the important features.

3.3.1 Data Preprocessing

The images were transformed to a size of [128, 128, 3]. This was the closest value to the mean size for the entire dataset that could be expressed as a power of 2. The reason behind choosing the value to be a power of 2 lies with the pooling and upsampling layers of autoencoder. In the encoding phase we use Pooling Layers of pool size [2,2] i.e. the number of input tensors in every channel gets halved after each pooling operation. Similarly, in the decoding phase we use UpSampling Layers of size [2,2] i.e. the number of input tensors in each channel gets doubled after each UpSampling operation. Thus, using a value that can be expressed as a power of 2, we can use Pooling and UpSampling operations symmetrically in the encoder and decoder to match the dimensions at the input and output layer of the autoencoder.

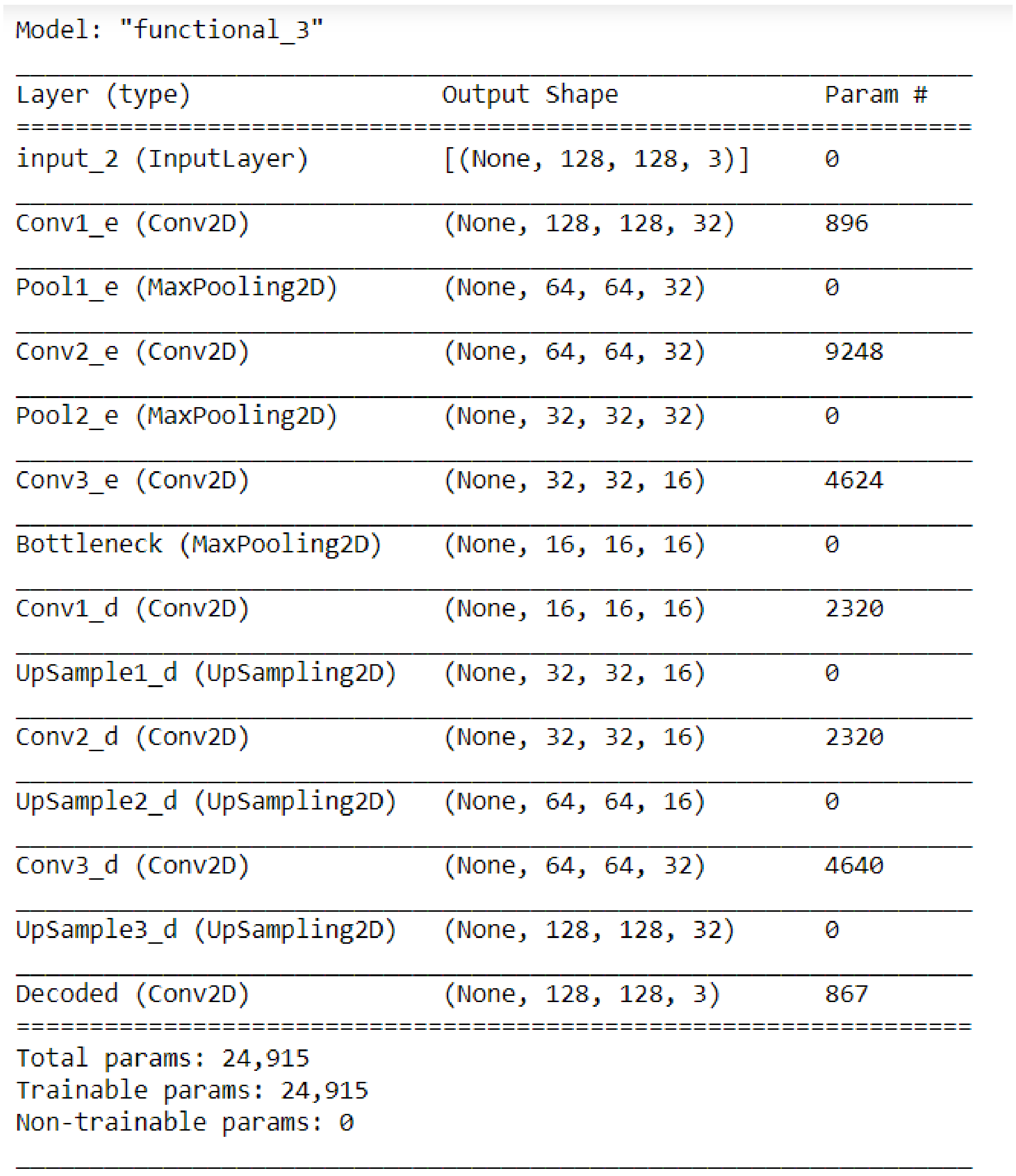
As we did for the CNN classifier, we normalized the image data by dividing the dataset by 255 and changing the pixel scale from [0-255] to [0-1]. Data augmentation was not used as good results were obtained with unaugment data.

3.3.2 Auto Encoder Architecture

The autoencoder was built using Tensorflow and Keras libraries. We used a Convolutional Auto-Encoder instead of a Fully Connected Auto-Encoder for two reasons: 1) Convolutional layers are better at handing image data as they preserve some spatial information, 2) Fully Connected Auto-Encoder would itself be a massive network with large amount of parameters to train.

The size of the input images is [128, 128, 3] or 49,153 features. The output of our bottleneck layer is of size [16, 16, 16] or 4,096 features. In the encoder, the input layer is followed by 3 Convolutional Blocks. Each Convolution Layer has a nonlinearity activation, which is followed by a Max Pooling Layer. The number of filters in the first, second and third convolution layers is 32, 32, and 16 respectively. All pooling layers use a pooling size of [2, 2]. Relu activation is used for nonlinearity.

In the decoder, the input size is [16, 16, 16], and the output size is [128, 128, 3]. The bottleneck layer is followed by 3 Convolutional Blocks. Each Convolution Layer is followed by a Relu activation, which is followed by an UpSampling Layer. The number of filters in the first, second and third convolution layers are 16, 32, and 32 respectively. All UpSampling Layers use a size of [2, 2]. The output from the final UpSampling layer is passed to a Convolutional Layer with 3 channels to match the dimensions with the original input.



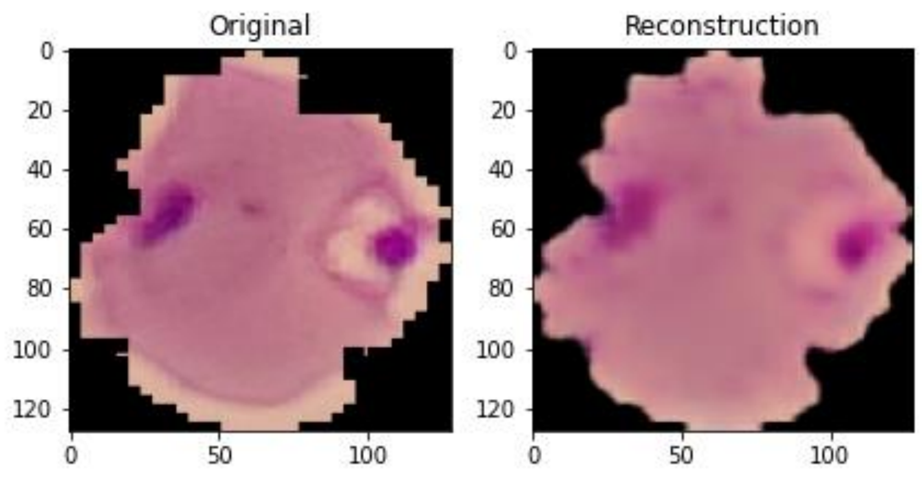
```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
Conv1_e (Conv2D)             (None, 128, 128, 32)      896
_________________________________________________________________
Pool1_e (MaxPooling2D)       (None, 64, 64, 32)        0
_________________________________________________________________
Conv2_e (Conv2D)             (None, 64, 64, 32)        9248
_________________________________________________________________
Pool2_e (MaxPooling2D)       (None, 32, 32, 32)        0
_________________________________________________________________
Conv3_e (Conv2D)             (None, 32, 32, 16)        4624
_________________________________________________________________
Bottleneck (MaxPooling2D)    (None, 16, 16, 16)        0
_________________________________________________________________
Conv1_d (Conv2D)             (None, 16, 16, 16)        2320
_________________________________________________________________
UpSample1_d (UpSampling2D)   (None, 32, 32, 16)        0
_________________________________________________________________
Conv2_d (Conv2D)             (None, 32, 32, 16)        2320
_________________________________________________________________
UpSample2_d (UpSampling2D)   (None, 64, 64, 16)        0
_________________________________________________________________
Conv3_d (Conv2D)             (None, 64, 64, 32)        4640
_________________________________________________________________
UpSample3_d (UpSampling2D)   (None, 128, 128, 32)      0
_________________________________________________________________
Decoded (Conv2D)             (None, 128, 128, 3)       867
=================================================================
Total params: 24,915
Trainable params: 24,915
Non-trainable params: 0
_________________________________________________________________
```

Fig. 3.4. Summary of the Convolutional Auto-Encoder Architecture

3.3.3 Auto Encoder Training

After many trials, the most precise model was obtained when trained under the following conditions:

- Adadelta optimizer was used with a learning rate of 2
- Binary cross entropy loss was used to computer loss
- Training progress was evaluated based on the loss as well as empirical analysis.
- The similarity of reconstructed image with the original image was used to estimate the quality of features on the bottleneck layer. A good reconstruction is only possible if the bottleneck layer accurately learns all the important features of the original data which can be seen in figure 3.5
- Training was progressed for 10 epochs. After 10 epochs the performance of the network did not increase significantly
- The training was adjusted based on the performance on the validation set. Hence, the model had not seen the test set during training.



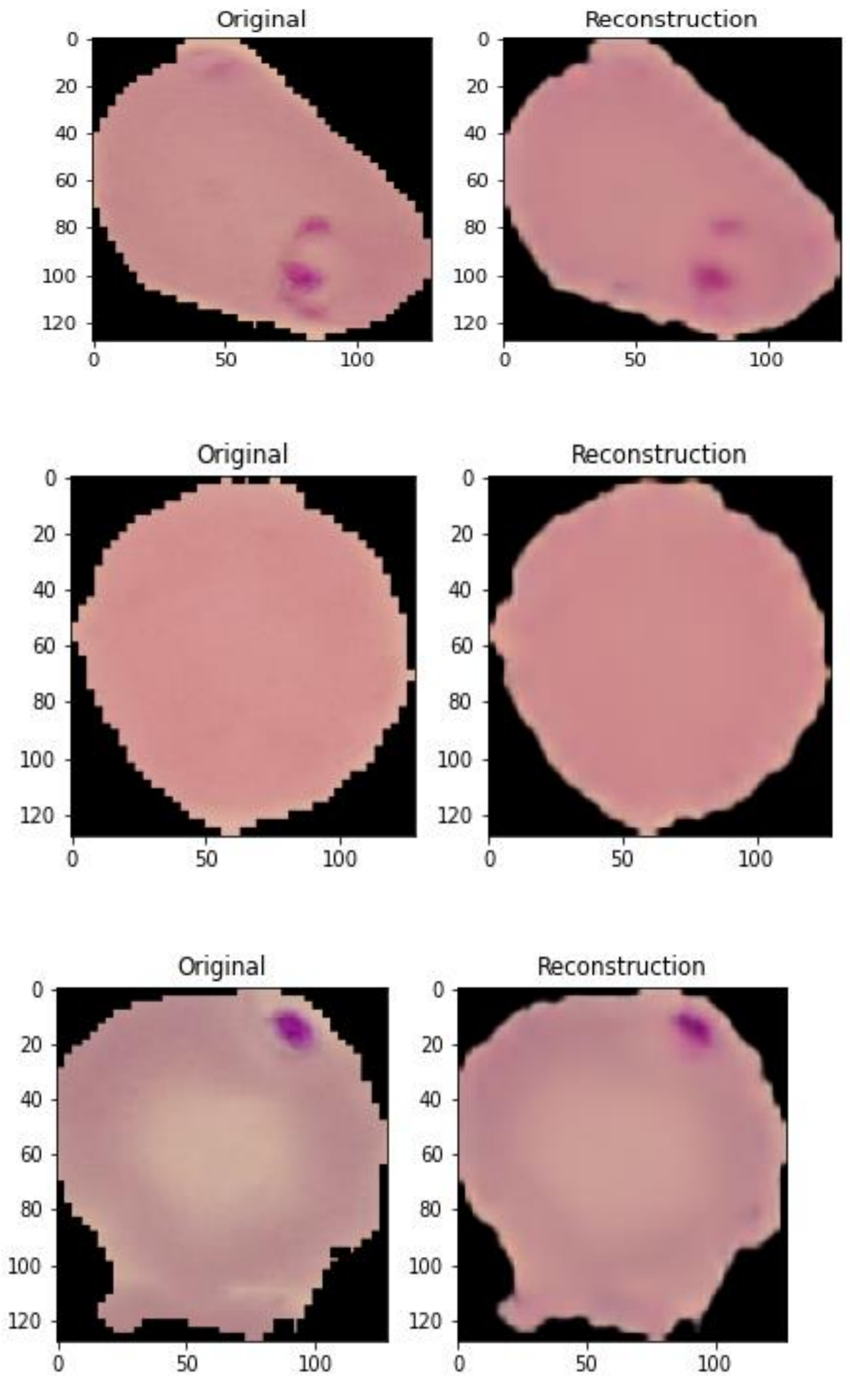


Fig 3.5. Original Vs Reconstructed Images obtained by the CAE on the test set

3.4 Malaria Infected Blood Cell Classifier with reduced features as input

The final phase of the project was to train the Malaria Infected Blood Cell Classifier using the new features obtained from the Convolutional Auto-Encoder's bottleneck layer. We used the encoder portion of the previously trained CAE to get the encoded input vectors and used it to train the Convolutional classifier that we had built in phase one. Since the input feature size is reduced, our model would have a smaller number of parameters to train. The objective of this phase was to see if we could build a smaller model using the reduced features without effecting the performance of the classifier.

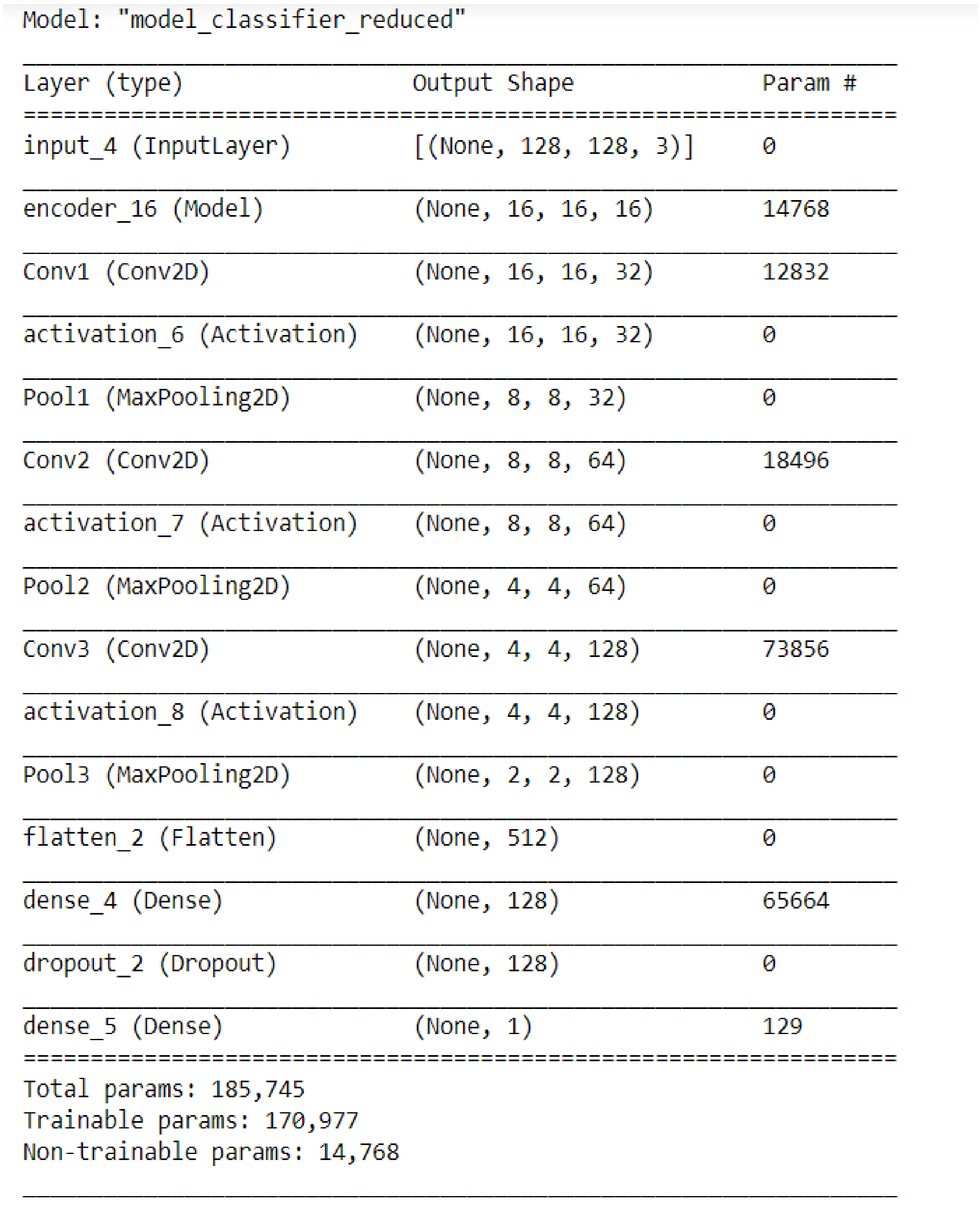
3.4.1 Data Preprocessing

The data preprocessing steps are identical to the ones used in the first phase to train the Malaria Infected Blood Cell Classifier described in section 3.2.1.

3.4.2 Combined Architecture of Encoder and CNN

The input data has size [128, 128, 3]. The input layer is first connected to the Encoder obtained from the Convolutional Auto-Encoder that was trained in phase 2. The encoder encodes the input images to a tensor of size [16, 16, 16]. Since this encoding is obtained by training the CAE to a good accuracy, the weights and biases need to be preserved. Hence, this encoding layer is frozen so that the weights and biases are not updated during training. Although we used an active approach for encoding, a passive approach can also be used where we first encode all the images, store it in the disk, and use that as our input to the classifier.

The encoder outputs a tensor of size [16, 16, 16], which is passed to a CNN classifier. The architecture of the CNN classifier is identical to the Malaria Infected Blood Cell Classifier which was used in phase 1 (shown in figure 3.2). However, as a result of the using reduced number of input features, significant difference can be seen in the amount of trainable parameters. The combined architecture is show below in figure 3.6. This architecture of this CNN classifier with an encoder attached to the input layer will be referred as the “CNN-Encoder Model” from this point forward in the paper.



```
Model: "model_classifier_reduced"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_4 (InputLayer)         [(None, 128, 128, 3)]     0         
_________________________________________________________________
encoder_16 (Model)           (None, 16, 16, 16)        14768     
_________________________________________________________________
Conv1 (Conv2D)               (None, 16, 16, 32)        12832     
_________________________________________________________________
activation_6 (Activation)    (None, 16, 16, 32)        0         
_________________________________________________________________
Pool1 (MaxPooling2D)         (None, 8, 8, 32)          0         
_________________________________________________________________
Conv2 (Conv2D)               (None, 8, 8, 64)          18496     
_________________________________________________________________
activation_7 (Activation)    (None, 8, 8, 64)          0         
_________________________________________________________________
Pool2 (MaxPooling2D)         (None, 4, 4, 64)          0         
_________________________________________________________________
Conv3 (Conv2D)               (None, 4, 4, 128)         73856     
_________________________________________________________________
activation_8 (Activation)    (None, 4, 4, 128)         0         
_________________________________________________________________
Pool3 (MaxPooling2D)         (None, 2, 2, 128)         0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 128)               65664     
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 129       
=================================================================
Total params: 185,745
Trainable params: 170,977
Non-trainable params: 14,768
_________________________________________________________________
```

Fig 3.6. Architecture of Malaria Infected Blood Cell Classifier with an added encoding layer (CNN-Encoder Model)

3.4.3 CNN-Encoder Model Training

The training conditions were tuned similar to the vanilla CNN classifier to properly estimate the effects of dimensionality reduction without interference from changes in other parameters. The final model was trained under the following conditions:

- Adam was used as the optimizer with a learning rate of 0.0001
- Binary cross entropy loss was used to computer loss
- A Dropout Layer with 25% dropout rate was used in the first Dense Layer
- Batch Size of 64 was used
- Early Stopping was used with a patience of 2 epochs

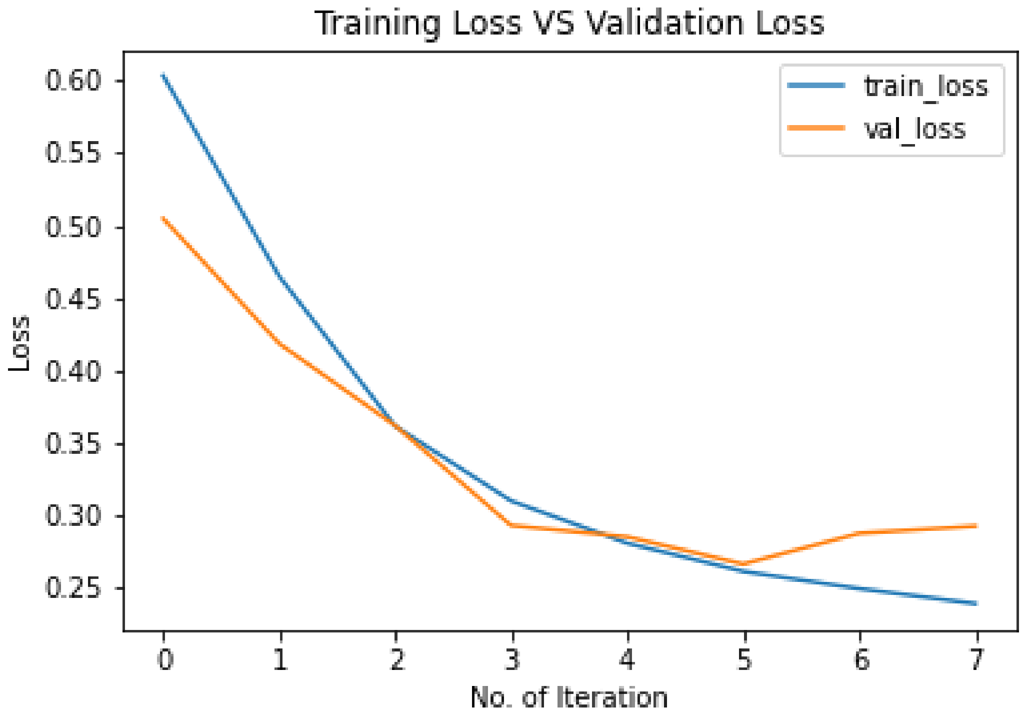


Fig. 3.7. Progression of Training and Validation Loss during training of the CNN-Encoder Model

CHAPTER 4

RESULTS

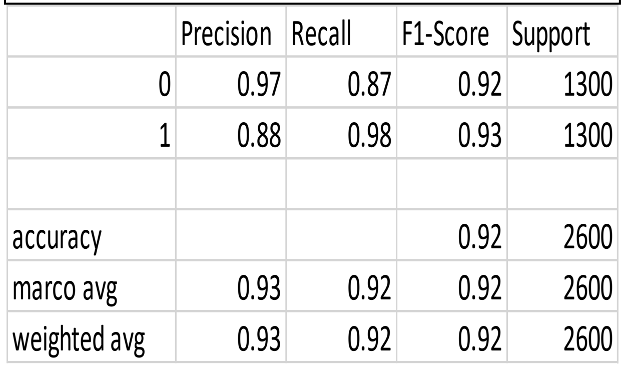
4.1 Classifier Comparison Metrics

To compare the efficiency and performance of the two models, two metrics were used, Accuracy and Model Size. Firstly, accuracy of the CNN-Encoder Model had to be comparable with the vanilla CNN classifier to posit that reduced features obtained from the auto encoders can be used to train a classifier without a significant impact on the classification accuracy. Secondly, the CNN-Encoder model would need to show a significant advantage in model size for us to conclude that the added step of dimensionality reduction can be worthwhile the increase the efficiency of the model.

4.1.1 Accuracy

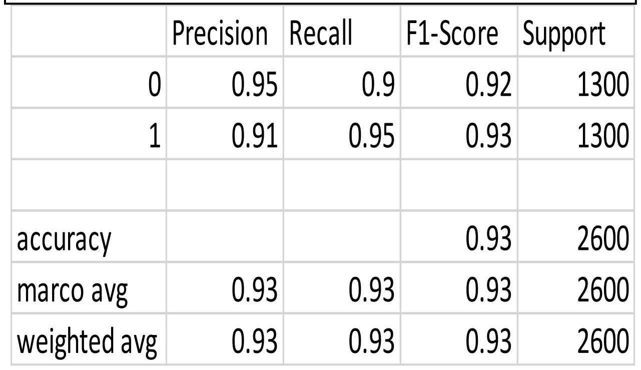
Both the models were trained under similar conditions. The performance of the models on the test set was used to generate the Classification Report. The classification reports for the two models are shown side-by-side in table 1.

Classification Report for CNN-Encoder Model



	Precision	Recall	F1-Score	Support
0	0.97	0.87	0.92	1300
1	0.88	0.98	0.93	1300
accuracy			0.92	2600
marco avg	0.93	0.92	0.92	2600
weighted avg	0.93	0.92	0.92	2600

Classification Report for Vanilla CNN Model



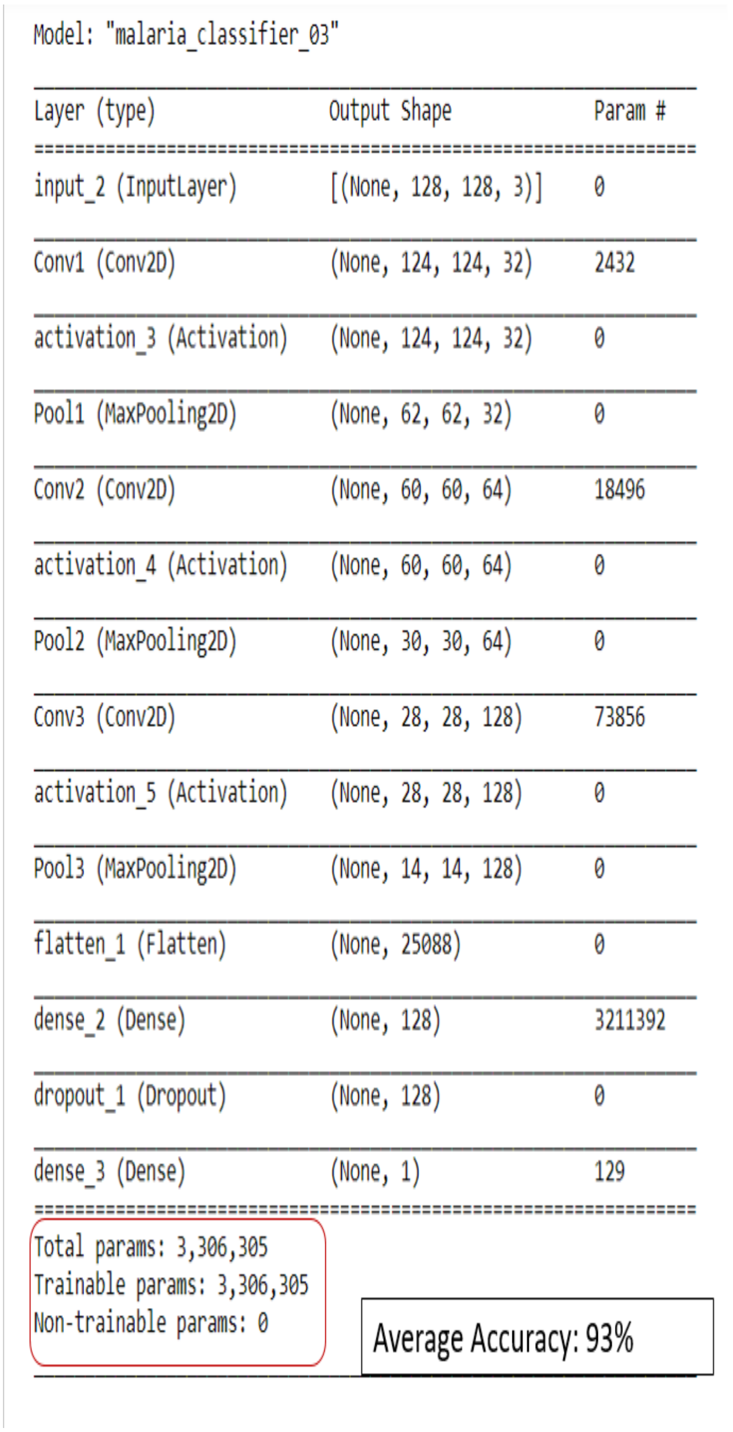
	Precision	Recall	F1-Score	Support
0	0.95	0.9	0.92	1300
1	0.91	0.95	0.93	1300
accuracy			0.93	2600
marco avg	0.93	0.93	0.93	2600
weighted avg	0.93	0.93	0.93	2600

Table 1. Classification Report CNN-Encoder Model Vs Vanilla CNN Model

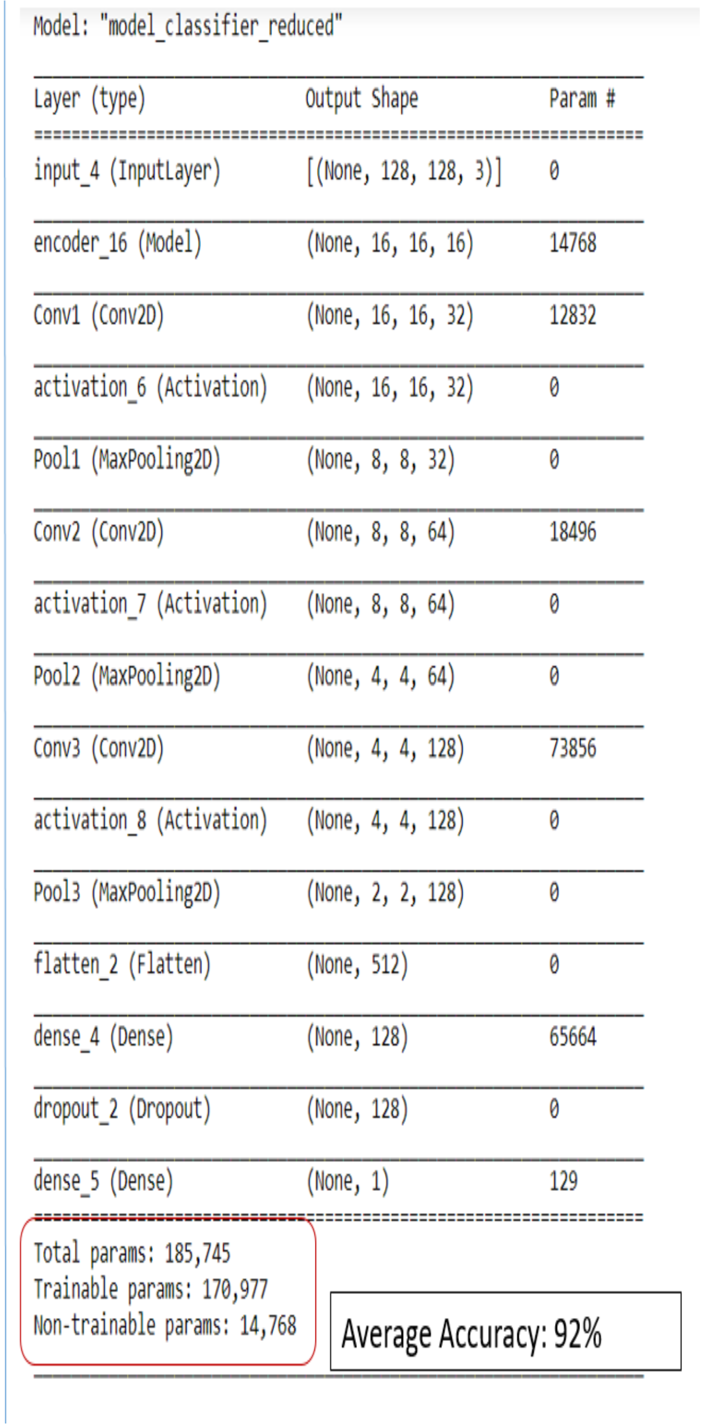
From table 1, we see that all the metrics in the Classification Report for both models are comparable to each other. The accuracy, precision, recall and F1 score for both model fall within an error bound of 1%, which is very close. Thus, looking at the classification report, we can conclude that the accuracy of the Vanilla CNN and CNN-Encoder Models are comparable to each other.

4.1.2 Model Size

The objective of our research project was to create efficient Neural Network models using non-linear dimensionality reduction. The efficiency we were striving for in this project was related to the model size i.e. building small NN models with comparable performance to their larger counterparts. Therefore, model size comparison is an important metric to evaluate the performance of our approach. Figure 4.1 shows the architecture of the Vanilla CNN and CNN-Encoder Model side by side with the number of parameters in each layer. We have established in the previous section that the accuracy for both these models is comparable. However, looking at the difference in parameter size, it is clear that the CNN-Encoder is a much compact model than the vanilla CNN classifier. While the vanilla CNN Model has 3,306,305 trainable parameter, the CNN-Encoder Model only has 185,745 total parameter and 170,977 trainable parameters; that makes a difference of an enormous 3,120,560 total parameters between the two models.


```
Model: "malaria_classifier_03"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
Conv1 (Conv2D)               (None, 124, 124, 32)      2432
_________________________________________________________________
activation_3 (Activation)    (None, 124, 124, 32)      0
_________________________________________________________________
Pool1 (MaxPooling2D)         (None, 62, 62, 32)        0
_________________________________________________________________
Conv2 (Conv2D)               (None, 60, 60, 64)        18496
_________________________________________________________________
activation_4 (Activation)    (None, 60, 60, 64)        0
_________________________________________________________________
Pool2 (MaxPooling2D)         (None, 30, 30, 64)        0
_________________________________________________________________
Conv3 (Conv2D)               (None, 28, 28, 128)       73856
_________________________________________________________________
activation_5 (Activation)    (None, 28, 28, 128)       0
_________________________________________________________________
Pool3 (MaxPooling2D)         (None, 14, 14, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               3211392
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 129
=================================================================
Total params: 3,306,305
Trainable params: 3,306,305
Non-trainable params: 0
_________________________________________________________________
```

Average Accuracy: 93%

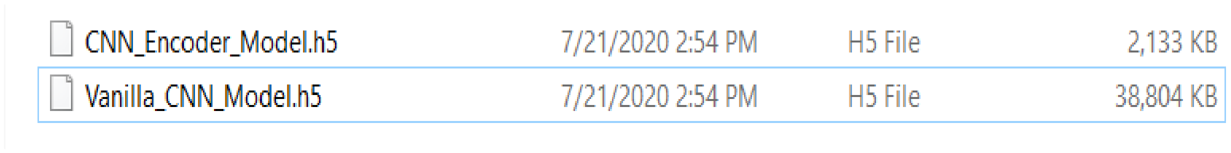

```
Model: "model_classifier_reduced"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
encoder_16 (Model)           (None, 16, 16, 16)        14768
_________________________________________________________________
Conv1 (Conv2D)               (None, 16, 16, 32)        12832
_________________________________________________________________
activation_6 (Activation)    (None, 16, 16, 32)        0
_________________________________________________________________
Pool1 (MaxPooling2D)         (None, 8, 8, 32)          0
_________________________________________________________________
Conv2 (Conv2D)               (None, 8, 8, 64)          18496
_________________________________________________________________
activation_7 (Activation)    (None, 8, 8, 64)          0
_________________________________________________________________
Pool2 (MaxPooling2D)         (None, 4, 4, 64)          0
_________________________________________________________________
Conv3 (Conv2D)               (None, 4, 4, 128)         73856
_________________________________________________________________
activation_8 (Activation)    (None, 4, 4, 128)         0
_________________________________________________________________
Pool3 (MaxPooling2D)         (None, 2, 2, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               65664
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 129
=================================================================
Total params: 185,745
Trainable params: 170,977
Non-trainable params: 14,768
_________________________________________________________________
```

Average Accuracy: 92%

Fig 4.1. Comparison of Vanilla CNN Model Architecture (left) with CNN-Encoder Architecture (right)

4.1.2.1 Model Size Evaluation on Disk Space

We saw the difference in the parameters size for the two models in the previous section. To have a better understanding of the impact of this difference in a real world scenario, we compared the difference of the model size on the computer disk. Both the Vanilla CNN and CNN-Encoder models were saved as an h5 file. The Vanilla CNN model when stored as an h5 file occupied a disc space of 37.8 MB or 38,804 KB. The CNN-Encoder model when stored as an h5 file occupied a disc space of 2.08 MB or 2,133 KB (a mere 5.5% of the original model).



CNN_Encoder_Model.h5	7/21/2020 2:54 PM	H5 File	2,133 KB
Vanilla_CNN_Model.h5	7/21/2020 2:54 PM	H5 File	38,804 KB

Fig 4.2 Comparison of disc space occupancy between Vanilla CNN Model and CNN-Encoder Model

4.2 Results Summary

The results show that, with proper dimensionality reduction, we can reduce the number of input features that are passed as input to a classifier to get a smaller model with good accuracy. The original Vanilla CNN model took the raw [128, 128, 3] images as input and the model had an accuracy of 93%. While the accuracy of the model was good, the model was very large with about three million parameters to train and occupied 37.8 MB in disc.

We built and trained a Convolutional Auto-Encoder to reduce the dimensionality of our Blood Cell Images from [128, 128, 3] to [16, 16, 16]. These new features although much smaller in size preserved all the important artifacts from the original image; we were able to verify this by looking at the reconstruction images obtained from the decoder. Since the decoder reconstructs an image from this latent space represented by [16, 16, 16] tensors, a good reconstruction would be possible only if all the important information from the original data was preserved in this latent space. We can see the high quality of reconstruction obtained from our bottleneck layer in figure 3.5.

We used this new reduced representation as input on the Vanilla CNN (as opposed to the raw image data) and trained this classifier under identical circumstances. This new model takes as input the images, encodes them into tensors of [16, 16, 16] using the pre-trained encoder, and uses this encoding to train the CNN classifier. This CNN with an encoding layer on top was called CNN-Encoder Model. This model had an accuracy of 92% on the test set (only 1% less than the vanilla CNN). However, the model was much smaller in size with only 185,745 total parameters and 170,977 trainable parameters. Furthermore, this CNN-Encoder only occupied 2.08MB in disc, which is less than 6% of the original model.

CHAPTER 5

CONCLUSION

5.1 Conclusion

Deep Learning and Artificial Intelligence have tremendous potential to solve complex problems in various domains. Deep Learning has been used to automate demanding tasks like detecting Cancer-causing tumors, studying complex genetic sequences, building autonomous vehicles, etc. However, the potential for Deep Learning stretches even beyond these tasks. 90% of the world's data was generated in the last couple of years [9]. With internet serving as the primary location for generation and storage of this data, much of this data is openly available for use. Deep Learning models thrive in the abundance of data, allowing them to perform complex tasks that would be impossible to fathom otherwise. But, are our Deep Learning Models scalable to keep up with the exponential growth of data and the expectations that come with it?

In 2019, Nvidia trained a model with 8.3 billion parameters where storing the parameters alone used up 33 Gigabytes on the disc [10]. It might not be possible to keep using more resources as the demand of resources also increase exponentially with larger DL models. Therefore, research in the scalability of DL models is a must if we wish to continue this progress that has been made in the field of Machine Learning in the era of Deep Learning.

In this project, we looked into non-linear dimensionality reduction as a feasible tool to reduce input data size features to significantly reduce the size of the model. By using autoencoder for dimensionality reduction, we were able to achieve a much smaller classification model whose performance was comparable to its larger counterpart. We used this technique to solve the problem of detecting Malaria Infected Blood Cell Images and achieved good results. However, the efficiency of this technique is not limited to this particular project. Any researchers or ML engineers who use Deep Learning Models abundantly and thus are at risk of exhausting their hardware resources are can benefit from this approach.

5.2 Future Work

For future work, we would like to test this approach on different problems to see if we can properly generalize the CNN-Encoder model for different problems. Furthermore, we also want to use the approach on some pre-built famous models like AlexNet or VGG. This will again help us to gauze at the generalization of this approach on different Neural Network architectures.

BIBLIOGRAPHY

[1]M. de Bruijne, "Machine learning approaches in medical image analysis: From detection to diagnosis", *Medical Image Analysis*, vol. 33, pp. 94-97, 2016. Available: 10.1016/j.media.2016.06.032.

[2]H. Lee and H. Kwon, "Going Deeper With Contextual CNN for Hyperspectral Image Classification", *IEEE Transactions on Image Processing*, vol. 26, no. 10, pp. 4843-4855, 2017. Available: 10.1109/tip.2017.2725580.

[3]J. Masci, U. Meier, D. Ciresan and J. Schmidhuber, "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction", *International Conference on Artificial Neural Networks*, pp. 52-59, 2011. [Accessed 20 March 2021].

[4]Y. Wang, H. Yao and S. Zhao, "Auto-encoder based dimensionality reduction", *Neurocomputing*, vol. 184, pp. 232-242, 2016. Available: 10.1016/j.neucom.2015.08.104.

[5]P. Tan, M. Steinbach, V. Kumar and A. Karpatne, *Introduction to data mining*, 2nd ed. Pearson, 2019, pp. 249-261.

[6]S. Albawi, T. Mohammed and S. Al-Zawi, "Understanding of a Convolutional Neural Network", *International Conference on Engineering and Technology*, 2017, doi 10.1109/ CEngTechnol.2017.8308186.

[7] D. Mellouli, T. M. Hamdani, J. J. Sanchez-Medina, M. Ben Ayed and A. M. Alimi, "Morphological Convolutional Neural Network Architecture for Digit Recognition," in IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 9, pp. 2876-2885, Sept. 2019, doi: 10.1109/TNNLS.2018.2890334.

[8]"Malaria Datasets. - LHNCBC Abstract", *Lhncbc.nlm.nih.gov*, 2021. [Online]. Available: https://lhncbc.nlm.nih.gov/LHC-publications/pubs/MalariaDatasets.html. [Accessed: 20- Mar- 2021].

[9]A. Sondur, "How much Data is generated every day?", *Medium*, 2021. [Online]. Available: https://medium.com/@amoghvs/how-much-data-is-generated-every-day-b94af8bcef4b#:~:text=We%20create%20around%202.5%20quintillion,of%20data%20worldwide%20by%202025. [Accessed: 20- Mar- 2021].

[10]J. Toole, "Deep learning has a size problem", *Heartbeat*, 2019. [Online]. Available: https://heartbeat.fritz.ai/deep-learning-has-a-size-problem-ea601304cd8. [Accessed: 20- Mar- 2021].