University of Cape Town

EEE4114F

Digital Signal Processing

Hand-Drawn Circuit Component Classification

Ryan Jones
JNSRYA006

Natasha Soldin
SLDNAT001

21/05/23

Contents

1 Introduction

The following project details the construction of a machine-learning algorithm that aims to identify Hand-Drawn Circuit Components. This will be achieved through the use of a Convolution Neural Network (CNN). This report includes (1) a literature review that critiques the relevant literature that specifically applies to the project, (2) a project description that details the project's purpose of hand-drawn circuit components classification and appropriate data selection, (3) a method that details the project's proposed implementation, (4) a results section that evaluates the success of the project and finally (5) a conclusion.

2 Literature Review

2.1 Image Classification

The prevalence of digital media in modern society has introduced a need to understand digital images [1]. Computers find this task particularly difficult, but the advancements in Machine Learning (ML) algorithms, have allowed classification to become an important part of computer vision technologies [2]. Classification as a whole, is a supervised learning technique, meaning, the algorithm is provided with both the input data, as well as the corresponding ground truths [3]. Image classification can be generally defined as the labeling of images into a specific class out of a set of predefined classes [2]. Some applications of which could be the labeling of medical scans - such as X-rays, quality control in the manufacturing process, and the association of names with particular faces in an image [4].

2.2 Convolution Neural Network

A CNN is a subclass of Artificial Neural Networks (ANNs) which are machine learning methods well suited for deep learning. ANN's name and structure are inspired by that of a human brain and therefore are comprised of layers of nodes (or neurons). These nodes are connected to others either in their own layer or across other layers, this connection has an associated weight and threshold [5]. The weight determines the node's contribution and the threshold determines whether information is transmitted over the connection or not. The difference between ANNs and CNNs is that instead of matrix multiplication being the calculation occurring between the different layers and weights, CNN's implement convolution in at least one of their layers [6].

CNN's are most commonly used to interpret visual imagery [7] or any data of grid-like topology. CNNs, unlike ANNs, are very skilled at reducing the number of model parameters while maintaining dataset quality and spatial positioning. Due to images' high dimensionality and the resulting large number of model features and parameters, they greatly benefit from CNN operation in that their pixel integrity is maintained [8] and the image can be re-generated un-corrupted without loss of pixel or image data.

The structure of a CNN and its layers can be graphically explained by the diagram in Figure 2.2.1 below:

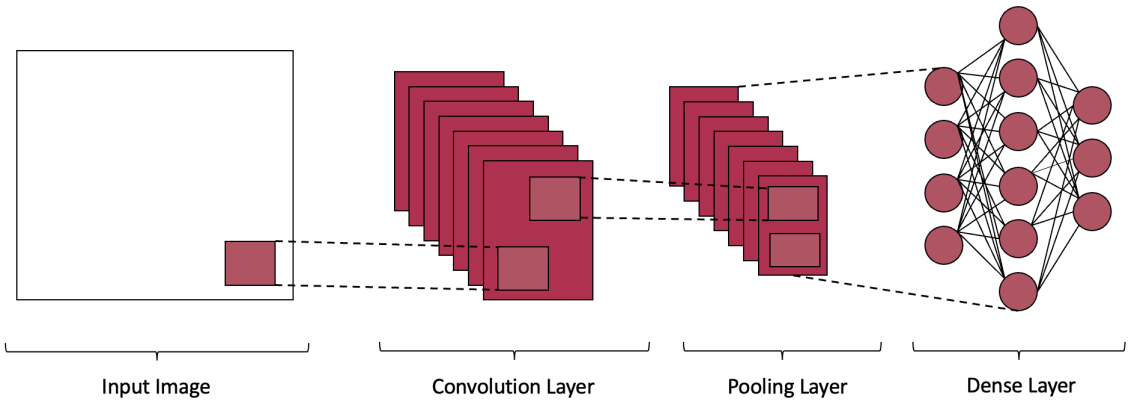


Figure 2.2.1: Diagram of the Structure of a CNN

In this application, the input to the CNN will be an image with a specified height, width, and depth. The height and width refer to the image's dimensions and the depth refers to the channel or number of colours that the image comprises (i.e. RGB has 3 channels and Greyscale has 1 channel). The output of the CNN will be an image classification label determined by the CNN's internal layers, the types of layers that a CNN may include are convolution, pooling, flattening, and dense described in Sections 2.2.1, 2.2.2, 2.2.3 and 2.2.4 below respectively.

2.2.1 Convolution Layer

The convolution layer takes in the input and its characteristics (height, weight, depth) and applies 2-dimensional weighted matrices to them, these matrices are called filters [9]. The filters move across the image, identifying learnable features of an image and condensing them into an output matrix or feature maps, which when compared to the original images are considerably smaller in size. Multiple filters can be applied to produce multiple feature maps. A feature map is then passed through an activation function which alters the output matrix such that it is ready to be transmitted to the next layers. The most commonly used activation function used is the Rectified Linear Activation (ReLU) function which effectively rids the feature map of all negative numbers [10] which increases the accuracy and decreases the computation required to utilise a CNN.

2.2.2 Pooling Layer

The pooling layer aims to reduce the dimensionality of the convolution output. It applies a 'window' filter over a certain number of pixels (frame size n x m) and simplifies that collection of pixels into a single pixel value. This overall dimension reduction in turn reduces the computational complexity required to process the data. There are two different types of pooling layers: Average Pooling takes the average value of all the pixels within the window and Maximum Pooling takes the maximum value pixel included within the window and effectively

'drops' the rest of them. This method not only decreases dimensionality but also acts as a noise suppressant and greatly aids the CNN's performance and is, therefore, the preferred method.

2.2.3 Flattening Layer

This is a transitional layer necessary to be placed before the Dense Layer. It takes the two-dimensional output of the Pooling Layer and transforms it into a single-dimension vector which is the necessary input format for the dense layer.

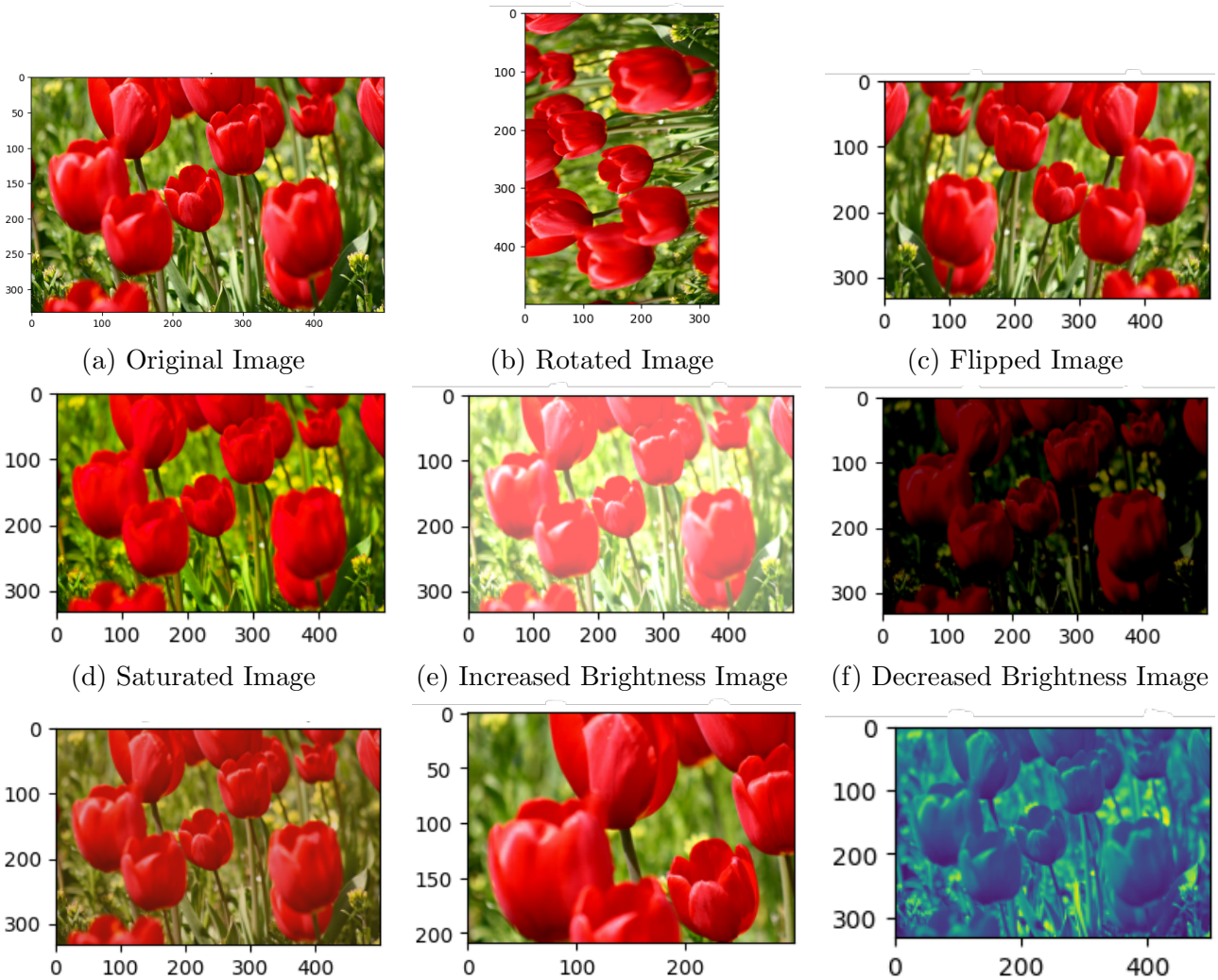
2.2.4 Dense Layer

This layer is also known as the Fully-Connected (FC) layer because each node in one layer is connected to every node in the adjacent layer. It starts with a flattened input and is passed through the FC layer. This classifies the inputted images, it uses an activation function that outputs the probabilities that an image belongs to each of the classes available. The class with the highest probability is outputted as the classification of the image.

2.3 Data Augmentation

Data augmentation is an approach used in the creation of a dataset on which the model is trained. The use of data augmentation is beneficial for two main reasons, namely, to increase the size of a limited dataset that the model is trained on, and to increase the variation of data that the model is trained on [11]. This means that the model is unable to learn general mappings of the data and this in turn prevents the model from over-fitting [2].

Different approaches can be taken to achieve augmentation of the data. Some of these include rotating, flipping, saturating, lightening, darkening, contrasting, cropping, or grey-scaling the image. The results of the aforementioned augmentation techniques are shown graphically in Figure 2.3.1 below.



(a) Original Image (b) Rotated Image (c) Flipped Image

(d) Saturated Image (e) Increased Brightness Image (f) Decreased Brightness Image

(g) Changed Contrast Image (h) Cropped Image (i) Greyscaled Image

Figure 2.3.1: Data Augmentation Techniques [12]

Depending on the application, some of the augmentation techniques explored in Figure 2.3.1 above, are more useful than others. For instance, if the input images are black-and-white, then augmentation techniques like grey-scaling or contrasting, shown in Figures 2.3.1i and 2.3.1g above, will not be beneficial and will not prevent a model from over-fitting - because the image will appear unchanged. Black-and-white images will benefit more from rotating or flipping the image, as shown in Figures 2.3.1b and 2.3.1c above, as these will alter the appearance of the image, thus providing the model with new data that can be learned to improve its performance [13].

2.4 Transfer Learning

This aptly named method refers to the re-use or transfer of previous model's training onto a new problem [14]. It, therefore, offers the ability to deeply and effectively train neural networks with minimal data input. Transfer learning works by focusing on edges in the earlier layers, general forms in the central layers, and task-specific features in the later layers. This forms

a strong foundation for any neural network whereby only the later layers will be retrained to suit the specific project scope. There are a multitude of Transfer Learning models available for image classification, however, some of the most popular, and most accurate include VGG, Inception, ResNet50, and EfficientNet and associated variations [15].

3 Project Description

This project aims to classify electrical circuitry components from hand-drawn schematic images. This application has been limited to five electrical components, namely: resistors, capacitors, inductors, diodes, and ground for simplicity. Once an appropriate model has been developed, these components can be extended to others but for initial implementation and testing, this limited scope can be beneficial.

3.1 Data Set Selection

There are limited online resources for hand-drawn electrical component databases. The one used in this project is the Kaggle database under the title Hand-Drawn Electric Circuit Schematic Components. This is an ideal database for the project as it involves hand-drawn circuit components with varying levels of accuracy and 'neatness' which accounts for real-life hand-drawn circuit component classification. However, this dataset has a limited number of images per classification which may hinder the performance of a CNN. As mentioned above, although this dataset has 15 different components, only five will be integrated into the machine learning model. A notable feature of this sub-dataset is that the diodes and ground components and the resistor and inductor components can look similar and thus it may be difficult for the model to distinguish between them - even more so if their orientations are altered (i.e. data augmentation).

4 Method

4.1 Data Pre-Processing

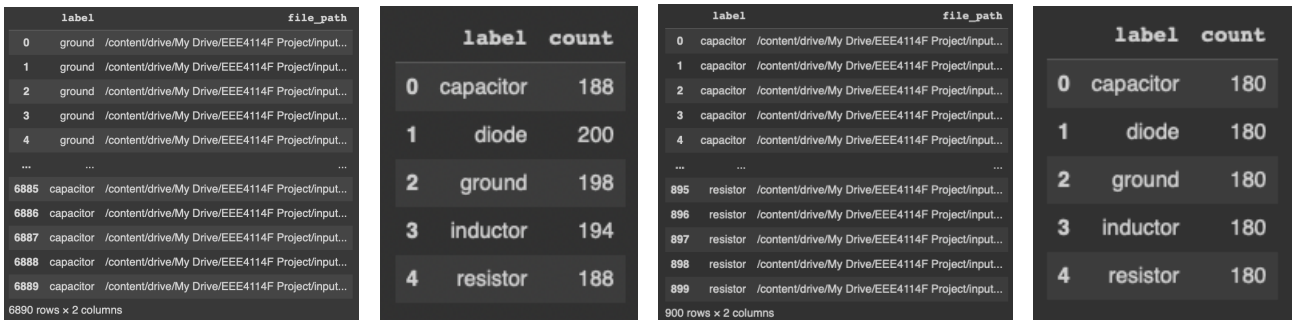
The images were stored in a Pandas Dataframe, initially using their relative file paths and corresponding component labels, as seen by Figure 4.1.1a below, during the data processing and 'cleaning' phase and later as their image. This cleaning phase involved ensuring that there were equal amounts of images per component classification. This was not the case, as seen by Figure 4.1.1b below, and thus the code in Listing 1 below was used to rectify this - producing equal data splits of 180 images per class as seen by Figures 4.1.1c and 4.1.1d below.

```
1 # this checks how many images per label - ideally we would like there to ...
      be the same number per classification
2 df_grouped_by_class = df.groupby('label', ...
      as_index=False).agg({'file_path': ...
      'count'}).rename(columns={'file_path': 'count'})
3
4 # this creates a new data frame with equal amounts of images per class
```

```
df_balanced = df.groupby('label').apply(lambda x: ...
    x.sample(n=180)).reset_index(drop=True)

# sanity check: ensure df_balanced is balanced (same number of images per ...
    classification)
df_balanced.groupby('label', as_index=False).agg({'file_path': ...
    'count'}).rename(columns={'file_path': 'count'})
```

Listing 1: Python Algorithm for Data Cleaning and Balancing



	label	file_path
0	ground	/content/drive/My Drive/EEE4114F Project/input...
1	ground	/content/drive/My Drive/EEE4114F Project/input...
2	ground	/content/drive/My Drive/EEE4114F Project/input...
3	ground	/content/drive/My Drive/EEE4114F Project/input...
4	ground	/content/drive/My Drive/EEE4114F Project/input...
...	...	...
6885	capacitor	/content/drive/My Drive/EEE4114F Project/input...
6886	capacitor	/content/drive/My Drive/EEE4114F Project/input...
6887	capacitor	/content/drive/My Drive/EEE4114F Project/input...
6888	capacitor	/content/drive/My Drive/EEE4114F Project/input...
6889	capacitor	/content/drive/My Drive/EEE4114F Project/input...

6890 rows × 2 columns

(a) Initial Dataframe

	label	count
0	capacitor	188
1	diode	200
2	ground	198
3	inductor	194
4	resistor	188

(b) Initial Image Groups per Class

	label	file_path
0	capacitor	/content/drive/My Drive/EEE4114F Project/input...
1	capacitor	/content/drive/My Drive/EEE4114F Project/input...
2	capacitor	/content/drive/My Drive/EEE4114F Project/input...
3	capacitor	/content/drive/My Drive/EEE4114F Project/input...
4	capacitor	/content/drive/My Drive/EEE4114F Project/input...
...	...	...
895	resistor	/content/drive/My Drive/EEE4114F Project/input...
896	resistor	/content/drive/My Drive/EEE4114F Project/input...
897	resistor	/content/drive/My Drive/EEE4114F Project/input...
898	resistor	/content/drive/My Drive/EEE4114F Project/input...
899	resistor	/content/drive/My Drive/EEE4114F Project/input...

900 rows × 2 columns

(c) Balanced Dataframe

	label	count
0	capacitor	180
1	diode	180
2	ground	180
3	inductor	180
4	resistor	180

(d) Balanced Image Groups per Class

Figure 4.1.1: Data Pre-processing process of Balancing Image Classes

The data was now split into separate train, validate and test data sets which account for 60%, 20%, and 20% of the total data respectively - a common split method where emphasis is placed on the majority of the data being used to train the model. This is achieved by the code in Listing 2 below.

```
# split the data into train and temp sets
df_train, df_temp = train_test_split(df_balanced, test_size=0.4, ...
    stratify=df_balanced['label']) # Check function shuffles while keeping ...
    splitting equally

# split the temp set into validation and test sets
df_validate, df_test = train_test_split(df_temp, test_size=0.5, ...
    stratify=df_temp['label'])
```

Listing 2: Python Algorithm for Train, Validate and Test Data Split

The final step of data pre-processing is to convert the relative file paths into their corresponding images. This is achieved by the TensorFlow ImageDataGenerator function as seen by the code in Listing 3 below. This code pertains to the training dataset but was repeated for the validation and testing datasets as well.

```
# generator for the training data
train_image_generator = ImageDataGenerator(rescale=1./255)

```

```
# converts file paths into an image object to be stored in the data frame ...
    for the training data set
train_data_gen = train_image_generator.flow_from_dataframe(
    dataframe=df_train,
    x_col='file_path',
    y_col='label',
    target_size=(img_height, img_width),
    # color_mode='grayscale', # transfer requires RGB images
    batch_size=batch_size,
    shuffle = False, # shuffling would not line with up ground truths
    class_mode='categorical',
    classes=['capacitor', 'diode', 'ground', 'inductor', 'resistor'],
)
```

Listing 3: Python Algorithm for Image Retireval from Dataframe

4.2 Model Training

A CNN will be implemented using TensorFlow to stack multiple layers, as discussed in Section 2.2 of the Literature Review, on top of each other in order to create a sequential neural network model. Several configurations were explored (i.e. different number and order of layers) and the best model was produced by the code in Listing 4 below. Which contains three convolution layers, three max-pooling layers, one flatten layer, and two dense layers.

```
model = tf.keras.Sequential([
        layers.experimental.preprocessing.Rescaling(1./255),
        layers.Conv2D(32, 3, activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(128, 3, activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.Dense(5, activation='softmax')
])
```

Listing 4: Python Algorithm for CNN Model

Once created, the model will be compiled with specified metrics and fit to the training dataset, and provided with the validation dataset. This should produce the final model which is ready to be tested on the testing dataset.

It is hypothesised that this CNN implementation will not be as accurate as wanted. This is because the Kaggle database explored in Section 3.1 above has a limited number of images per class or in this case per circuit components. These range from 188 to 200 components per class. Two methods of model improvement could be Data Augmentation or Transfer Learning, both of which will be implemented and tested to attempt better model performance and accuracy.

4.2.1 Data Augmentation

A founding team member of TensorFlow, Pete Warden, stated that in order to have a successful machine learning image classification algorithm, each class should contain no fewer than 1000 images [16].

This would suggest that the current dataset is insufficient. Therefore data augmentation, as discussed in Section 2.3 of the Literature Review, will be employed to increase the data set size and as added benefit will advantage the model in manners explored above. Due to the simplistic structure and black-and-white coloring of the electrical components, only image rotation was implemented as it was decided to be the only form of data augmentation that would benefit the model. This image rotation was implemented in 6 different degrees: 45°, 90°, 135°, 180°, 270° and 315°. It was accomplished using the code shown in Listing 7 in Appendix A.1.

The augmentation machine learning algorithm still involves the same Data Pre-Processing as before, just with a larger number (i.e. 1310) of images per class. It also implements the same CNN Model Training as before, as shown by Listing 4.

4.2.2 Transfer Learning

This method does not modify the dataset size but provides the model with a greater foundation such that the smaller dataset does not disadvantage the model as much. This is discussed in Section 2.4 of the Literature Review and is implemented using the code in Listing 5 below. This code shows the implementation of a specific type of transfer learning: ResNet50, however, it will be adjusted to test several transfer learning models to determine which is the most effective. The choice of models is based on the discussion in Section 2.4 of the most popular and accurate models. For some of these models, two different versions were tested.

```
from tensorflow.keras.applications.resnet50 import ResNet50     #transfer ...
    learning model

# setting up transfer learning model (this may need to be changed with ...
    different transfer learning models)
base_model = ResNet50(input_shape=(img_height, img_width, 3), ...
    include_top=False, weights='imagenet')

# model training (layers are variable, more can be added - only after ...
    flatten)
model = Sequential()
model.add(base_model)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(5, activation='softmax'))
```

Listing 5: Python Algorithm used for to implement Transfer Learning Model ResNet50

This just creates the model, the manner in which it is compiled and tested is the same as for the regular CNN model shown in Listing 4 above.

To compare the three models in terms of their training and validation accuracy and loss, plots will be generated to illustrate these features.

4.3 Model Testing

All three models as mentioned above will be trained on the training and validation datasets and then tested on the test dataset - which has been kept separate to avoid overfitting the model. All three models are tested in the same manner using the code in Listing 6 below and will produce an overall accuracy and loss value.

```
# sending in the testing data set
scores = model.evaluate(test_data_gen)
```

Listing 6: Python Algorithm used to test a Pre-Trained Model

To compare the three models in terms of their performance on testing data, confusion matrices will be plotted to illustrate the model's accuracy in terms of each component classification. The model with the highest accuracy will also be tested on a different dataset of electrical components that were hand drawn by engineering students. This will test if the model has been overfit to the Kaggle dataset or if it generalises well to other data, signaling a well-trained model.

The code of all three of the above methods was implemented in three separate Jupyter Notebooks hosted on Google Colab. All the necessary code to implement the above methods is hosted in the GitHub repository available here.

5 Results

5.1 Model Comparison

5.1.1 CNN

The simple CNN implementation with the Kaggle dataset has the following results.

```
Epoch 1/5
16/16 [==============================] - 148s 9s/step - loss: 1.1468 - accuracy: 0.5709 - val_loss: 0.9580 - val_accuracy: 0.6375
Epoch 2/5
16/16 [==============================] - 83s 5s/step - loss: 0.5184 - accuracy: 0.8110 - val_loss: 0.6618 - val_accuracy: 0.7875
Epoch 3/5
16/16 [==============================] - 74s 5s/step - loss: 0.1152 - accuracy: 0.9606 - val_loss: 0.8385 - val_accuracy: 0.8000
Epoch 4/5
16/16 [==============================] - 74s 5s/step - loss: 0.0650 - accuracy: 0.9843 - val_loss: 1.1196 - val_accuracy: 0.8250
Epoch 5/5
16/16 [==============================] - 79s 5s/step - loss: 0.0501 - accuracy: 0.9823 - val_loss: 0.9944 - val_accuracy: 0.7500
```

(a) Train and Validation Results over all 5 epochs

```
Test Accuracy: 78.89%
Test Loss: 0.85%
```

(b) Test Accuracy and Loss

Figure 5.1.1: CNN Train, Validation and Test Results

5.1.2 Data Augmentation

As discussed in Section 2.3, the hyperparameters are kept the same except the number of images per class increased. The number of epochs was kept constant at 5. With this in mind, the simple CNN implementation with the augmented dataset has the following results.

```
Epoch 1/5
123/123 [==============================] - 350s 3s/step - loss: 0.9953 - accuracy: 0.6089 - val_loss: 0.6550 - val_accuracy: 0.7641
Epoch 2/5
123/123 [==============================] - 346s 3s/step - loss: 0.2985 - accuracy: 0.8926 - val_loss: 0.5598 - val_accuracy: 0.8122
Epoch 3/5
123/123 [==============================] - 336s 3s/step - loss: 0.0606 - accuracy: 0.9802 - val_loss: 0.8614 - val_accuracy: 0.7977
Epoch 4/5
123/123 [==============================] - 334s 3s/step - loss: 0.0374 - accuracy: 0.9873 - val_loss: 0.8019 - val_accuracy: 0.8214
Epoch 5/5
123/123 [==============================] - 345s 3s/step - loss: 0.0116 - accuracy: 0.9967 - val_loss: 0.8160 - val_accuracy: 0.8267
```

(a) Train and Validation Results over all 5 epochs

```
Test Accuracy: 82.29%
Test Loss: 0.88%
```

(b) Test Accuracy and Loss

Figure 5.1.2: CNN Train, Validation and Test Results

Comparing Figures 5.1.1b and 5.1.2b, it can be seen that the use of augmentation increased the test accuracy.

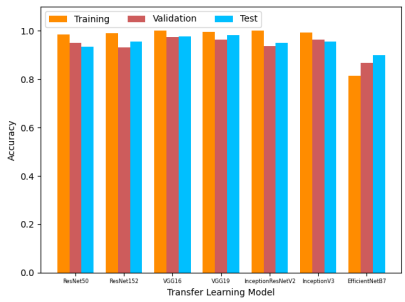
5.1.3 Transfer Learning

Each Transfer Learning model was run using the same train, validation, and test split. Each model's performance was evaluated on accuracy, loss, and time taken at each stage of the training, validation, and test process. This data is tabulated in Table 1 below and graphically shown by Figure 5.1.3a, 5.1.3b and 5.1.3c in terms of accuracy, loss, and time taken per classification respectively.

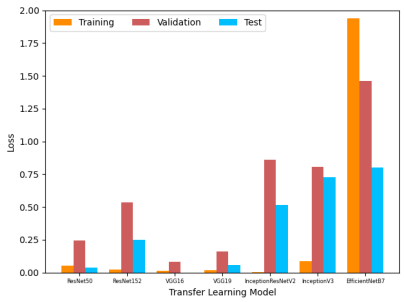
Model Name	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss	Train and Validation Time (s)	Number of Images	Time per image (s)	Testing Accuracy	Testing Loss	Testing Time	Number of images	Average Time per Classification (s)
ResNet50	0.9843	0.0504	0.95	0.2464	787	2700	0.291481481	0.9333	0.037	39	180	0.216666667
ResNet152	0.9902	0.0214	0.9312	0.5332	2091	2700	0.774444444	0.9556	0.251	92	180	0.511111111
VGG16	1	0.013	0.975	0.084	2343	2700	0.867777778	0.9778	0.0007	113	180	0.627777778
VGG19	0.9961	0.0189	0.9625	0.1609	2690	2700	0.996296296	0.9833	0.0584	137	180	0.761111111
InceptionResNetV2	1	0.0018	0.9375	0.8609	1196	2700	0.442962963	0.95	0.5152	50	180	0.277777778
InceptionV3	0.9941	0.0878	0.9625	0.8078	474	2700	0.175555556	0.9556	0.7297	24	180	0.133333333
EfficientNetB7	0.815	1.9407	0.8687	1.46	2363	2700	0.875185185	0.9	0.8036	111	180	0.616666667

Table 1: Comparison of all Transfer Learning models used

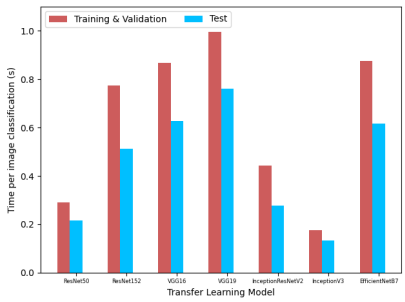
Plotting the results shown above in Table 1 as histograms, results in the following plots.



(a) Accuracy



(b) Loss



(c) Time per image classification

Figure 5.1.3: Comparison of Different Transfer Learning Models

From the data in Table 1 above and the associated plots in Figure 5.1.3 above, it is evident that certain transfer learning models are better suited to this project and thus more accurate. The model with the greatest test data accuracy was VGG16. The results also highlight the notable presence of a trade-off between accuracy and the time per image classification or ultimately the overall model's execution time. This project has placed emphasis on data accuracy and therefore will not take this trade-off into consideration. However, for other applications where this trade-off is relevant, careful consideration must be taken to ensure that the best model is chosen which does not favour accuracy over execution time or vice versa.

The CNN implementation with the VGG16 transfer learning model has the following results, bearing in mind that the number of epochs was again stated to be 5:

```
Epoch 1/5
16/16 [==============================] - 487s 31s/step - loss: 2.8789 - accuracy: 0.5945 - val_loss: 0.2827 - val_accuracy: 0.9125
Epoch 2/5
16/16 [==============================] - 472s 30s/step - loss: 0.1728 - accuracy: 0.9390 - val_loss: 0.1703 - val_accuracy: 0.9438
Epoch 3/5
16/16 [==============================] - 466s 30s/step - loss: 0.0604 - accuracy: 0.9882 - val_loss: 0.1075 - val_accuracy: 0.9688
Epoch 4/5
16/16 [==============================] - 469s 30s/step - loss: 0.0287 - accuracy: 0.9961 - val_loss: 0.0987 - val_accuracy: 0.9750
Epoch 5/5
16/16 [==============================] - 467s 30s/step - loss: 0.0146 - accuracy: 1.0000 - val_loss: 0.0914 - val_accuracy: 0.9812
```

(a) Train and Validation Results over all 5 epochs

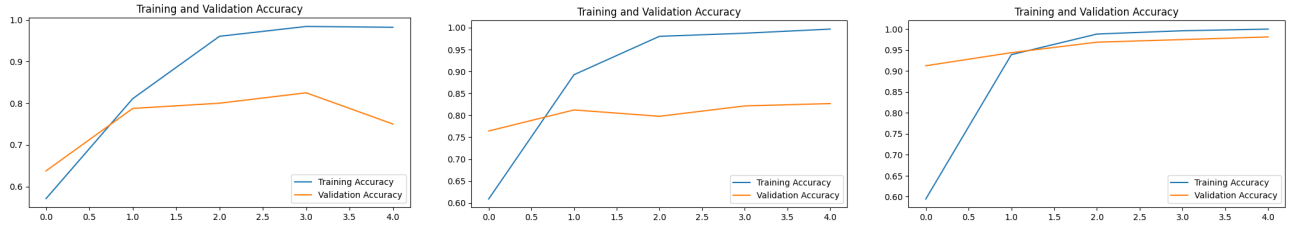
```
Test Accuracy: 82.29%
Test Loss: 0.88%
```

(b) Test Accuracy and Loss

Figure 5.1.4: CNN Train, Validation and Test Results

Comparing Figures 5.1.1b and 5.1.2b with Figure 5.1.4b, it can be seen that the use of transfer learning increased the test accuracy even further.

The three models' results can be plotted graphically for ease of comparison. The train and validation data accuracy for each model can be seen in Figure 5.1.5 below and the train and validation data loss for each model can be seen in Figure 5.1.6 below. A confusion matrix is generated for the test data for each model, as shown by Figure 5.1.7 below, which will allow for the comparison of classification accuracy.

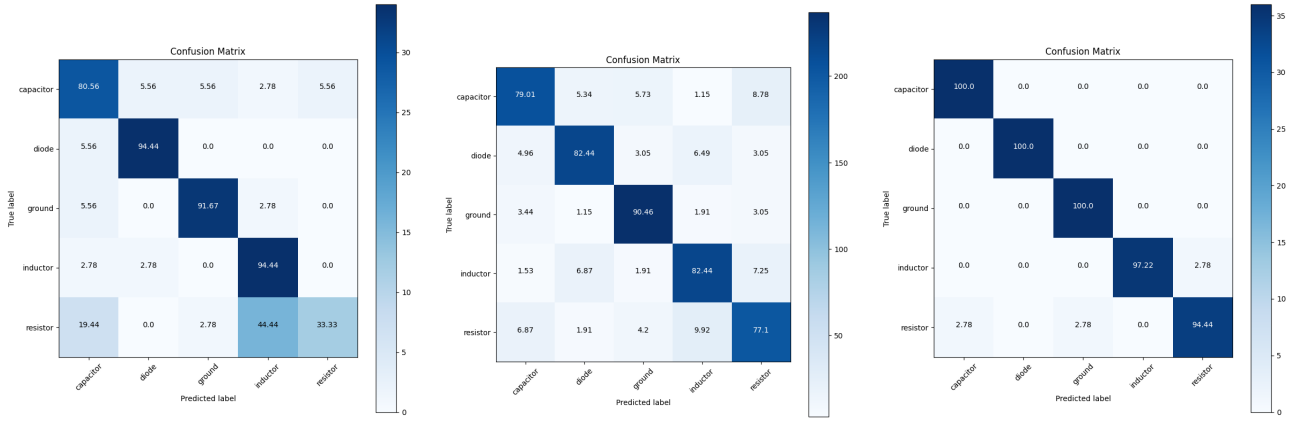


(a) CNN (b) CNN with Augmented Data (c) CNN with Transfer Learning

Figure 5.1.5: The Accuracy of the Training and Validation Datasets on Different Models

(a) CNN (b) CNN with Augmented Data (c) CNN with Transfer Learning

Figure 5.1.6: The Loss of the Training and Validation Datasets on Different Models



(a) CNN (b) CNN with Augmented Data (c) CNN with Transfer Learning

Figure 5.1.7: Confusion Matrices Indicating Classification Accuracy on Different Models

From Figure 5.1.7 above, two results can be seen. Firstly, an increase in classification accuracy can be seen between the CNN confusion matrix in Figure 5.1.7a and the CNN with augmented data confusion matrix in Figure 5.1.7b. This is seen through the darkening of the squares depicting accurate resistor and inductor classifications. Further improvement is seen between the CNN with augmented data confusion matrix in Figure 5.1.7b and the CNN with Transfer Learning in Figure 5.1.7c. All diagonal squares of the confusion matrix have darkened, and thus, the classification accuracy improved.

The second observable result from Figure 5.1.7, is the fact that the CNN in Figure 5.1.7a struggled the most with classifying resistors and inductors. This confirms the hypothesis of misclassification of similar-looking components expressed in Section 3.1. The CNN classified resistors as inductors more than correctly classifying resistors as resistors. The CNN classified resistors as inductors 44.44 % of the time, whereas it classified resistors as resistors 33.33 % of the time. However, when comparing the confusion matrices of the CNN with augmentation and the CNN with Transfer Learning, this issue can be seen to be mitigated.

5.2 Hand-Drawn Test Data

The best-performing model was the Transfer Learning VGG16 model. This model was supplied with unseen hand-drawn electrical component data and produced the following results:

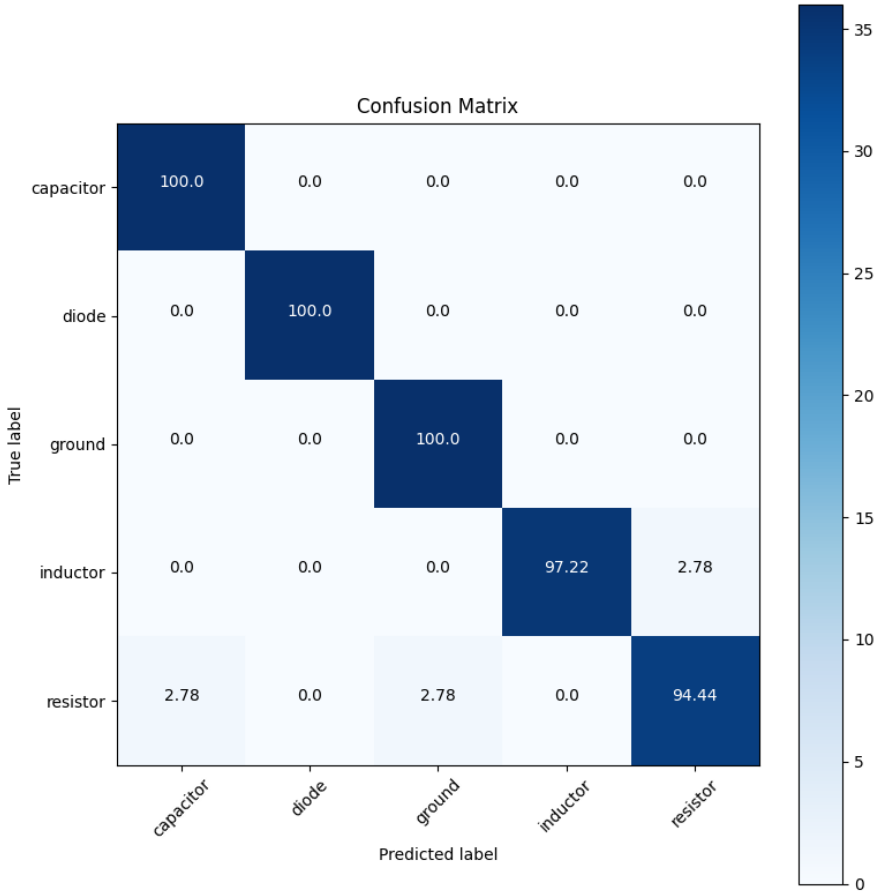


Figure 5.2.1: Confusion Matrix for Transfer Learning CNN Model Performance on Hand-Drawn Test Data

Due to the adequate accuracy obtained, it can be concluded that the model performance is irrespective of the dataset provided. This is indicative of a well-trained model with little to no over-fitting.

6 Conclusion

In this project the simple CNN struggles to produce accurate classification on test data. In order to improve the classification accuracy, two different approaches were taken. The first approach was the augmentation of the input dataset which in turn increased the size of the dataset. This allowed the CNN model to improve its test classification accuracy slightly, however, not to the desired level. The second approach of implementing the CNN with Transfer Learning models significantly improved the test classification accuracy. Various models were tested and the VGG16 proved to be the most accurate. This model was then tested on hand-drawn test data, as opposed, to a test split of the Kaggle dataset. Comparing the results of the hand-drawn test data to the test split returned a comparable accuracy of approximately 98 %. This meant that the model was well-trained and did not display any evidence of being over-fit. This signifies the success in implementing a CNN for image classification as well as demonstrates the project's success in improving upon the accuracy of the model.

References

[1] E. d'Archimbaud, "Programming image classification with machine learning." [Online]. Available: https://kili-technology.com/data-labeling/computer-vision/image-annotation/programming-image-classification-with-machine-learning#1

[2] T. Jain, "Basics of machine learning image classification techniques," Aug 2019. [Online]. Available: https://iq.opengenus.org/basics-of-machine-learning-image-classification-techniques/

[3] J. Son, *Digital Signal Processing - EEE2045F Module 1 Notes.* University of Cape Town, 2023.

[4] S. Javaid, "Image classification: 6 applications; 4 best practices in 2023," Feb 2023. [Online]. Available: https://research.aimultiple.com/image-classification/

[5] [Online]. Available: https://www.ibm.com/topics/neural-networks

[6] [Online]. Available: https://www.deeplearningbook.org/

[7] M. Mandal, "Introduction to convolutional neural networks (cnn)," Apr 2023. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

[8] P. Mishra, "Why are convolutional neural networks good for image classification?" Jul 2019. [Online]. Available: https://medium.datadriveninvestor.com/why-are-convolutional-neural-networks-good-for-image-classification-146ec6e865e8

[9] U. Udofia, "Basic overview of convolutional neural network (cnn)," Sep 2019. [Online]. Available: https://medium.com/dataseries/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17#:~:text=The%20activation%20function%20is%20a,neuron%20would%20fire%20or%20not.

[10] J. Brownlee, "How to choose an activation function for deep learning," Jan 2021. [Online]. Available: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

[11] A. D'Cruz, "What is data augmentation in deep learning?" May 2022. [Online]. Available: https://www.calipsa.io/blog/what-is-data-augmentation-in-deep-learning

[12] Dec 2022. [Online]. Available: https://www.tensorflow.org/tutorials/images/data_augmentation

[13] B. Raj, "Data augmentation — how to use deep learning when you have limited data - part 2," Apr 2018. [Online]. Available: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

[14] P. Sharma, "Understanding transfer learning for deep learning," Oct 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/10/understanding-transfer-learning-for-deep-learning/

[15] P. Huilgol, "Top 4 pre-trained models for image classification with python code," May 2023. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/

[16] P. Warden, "How many images do you need to train a neural network?" Dec 2017. [Online]. Available: https://petewarden.com/2017/12/14/how-many-images-do-you-need-to-train-a-neural-network/

A Appendix

A.1 Data Augmentation Code

The following python script was used to augment the images, the codein Listing 7 below illustrates the 90° image rotation - this was repeated for all required degree rotations.

```
from PIL import Image
import os

# specify the source folder (kaggle dataset images) and destination ...
    folder where the rotated images will be saved
source_folder = 'input_png/resistor'
destination_folder = 'input_png_rotated/resistor'

# specify the expected image extension
extension = 'png'

# retrieves images stored in the source folder to a image list
image_files = [f for f in os.listdir(source_folder) if ...
    os.path.isfile(os.path.join(source_folder, f))]

# loops through each image list to apply rotation to each image in the ...
    source folder
for file_name in image_files:

# ensures that any hidden files (e.g. .DS_Store) are disregarded
if file_name[-3:] == extension:

# opens the image file using the source folder file path and name of image
image_path = os.path.join(source_folder, file_name)
img = Image.open(image_path)

# rotates the image by 90 degrees clockwise
rotated_img = img.rotate(90)

# specifies the destination folder where the rotated images will be ...
    stored and their image name
destination_path = os.path.join(destination_folder, "90_degrees_"+ file_name)

# saves the rotated image to the destination folder
rotated_img.save(destination_path)
```

Listing 7: Python Algorithm used for 90° Rotation of all Images in a specified folder