COMP4211: Project Report

Studying CNN Models for Defects Detection in Cookies

by

Leung King Suen

Student ID	Email
20770625	ksleungac@connect.ust.hk

This report is formatted base on this template [5].

Contents

1
Introduction

In this project, I trained 3 CNN models using the **Industry Biscuit (Cookie) dataset** [2]. The first and the second model are built based on AlexNet and VGGNet taught in class. The third model is a lite version of VGGNet in hope to achieve a balance between training time and model performance metrics.

1.1. the Dataset and Preprocessing

This dataset is made up of 1225 samples of cookies. Among which 751 samples were with defects like incomplete cookie (465 samples), strange object in cookie (158 samples) or color defect found (128 samples), the remaining 474 samples were the ones without defects. The dataset is augmented by the author such that each image is rotated 90 degrees 3 times to create 3 new samples for a image. So there are 4900 cookies images in total. Each of the image is 256×256 in size, has 3 channels (RGB image). Pixels are normalized from range [0, 255] to [0, 1] before training. The following illustration shows the different classes of cookies.

Figure 1.1: Color Defect

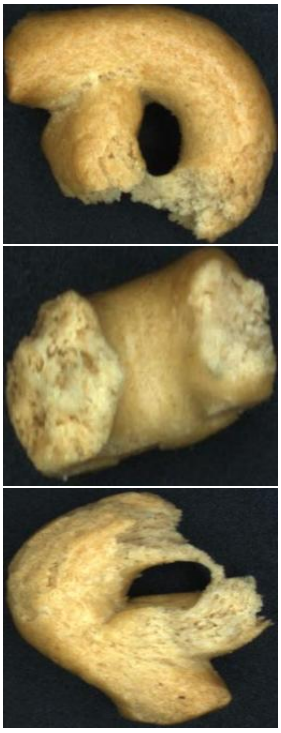

Figure 1.2: Incomplete

Figure 1.3: Normal

Figure 1.4: Strange Object

The 4900 image samples are split-ed into 3 sets. The first 20% in alphabetical order is used as test set, the remaining 80% is used as training set. 20% from the training set are used for validation. At the end, there are 3137 images used for training, 784 images for validation and 979 images for testing.

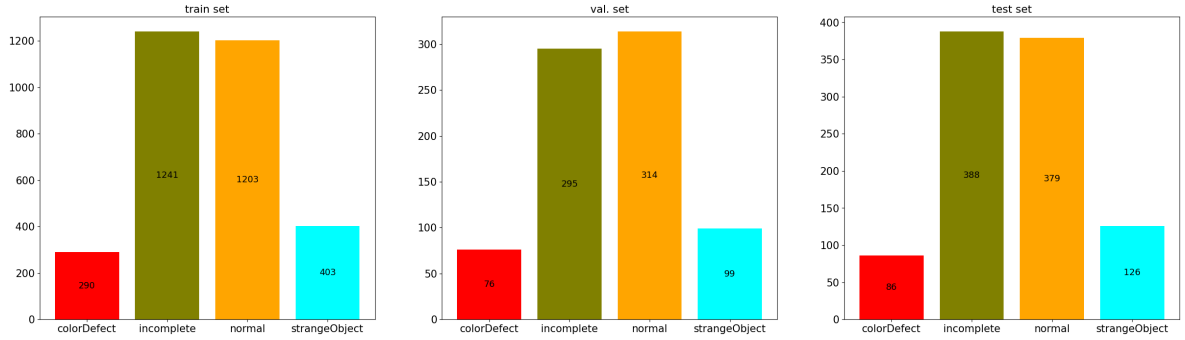

Figure 1.5: Distribution of Samples

This dataset is unbalanced as seen from figure 1.5. Where 'color defect' samples were approximately 1/4 of the majority classes such as 'incomplete' and 'normal' in the 3 sets. while strange object were about 1/3 of the majority classes. This will be tackled by re-weighting classes and by using the learning rate scheduler in the training process.

1.2. Multi-class Classification using CNN

I will train a model using AlexNet CNN, and then another model using VGG16 for feature extraction along with 2 fully connected layers. At last, I will train a simpler VGG16 by removing some of the convolutional layers. Then I will be comparing and select the best model for this dataset specifically, in balance of metrics such as F1-score, number of trainable parameters and training time.

One thing worth noting is this dataset is originally for Anomaly Detection task, where it's used for binary classification, so as the script provided by the author only splits the dataset to OK/notOK. But it comes with a .csv file that has labelled each image to each of the 4 classes, therefore, I split-ted them to 4 folders corresponding to the 4 classes, and do a multi-class classification task. However, because the labelling were done manually, there are some wrongly labelled samples, I have corrected most of them.

2
Training Process and Demo

Computation Environment
Before getting into the training process, below is a list for the environment and hardware used for training:

- **Software:** TensorFlow 2.8.2, Python 3.8 running Anaconda 3 on Windows 11 Home;
- **Hardware:** NVIDIA GeForce RTX 3080 (CUDA Compute Capability 8.6, TensorFlow GPU support on Windows Native);

2.1. General model training settings
Each model is trained with following settings across the board.

Callbacks
Using tf.keras.callbacks.Callback() instance to save model weight that has the best validation accuracy after each epoch.

Learning Rate Scheduler
Using tf.keras.callbacks.ReduceLROnPlateau to control the learning rate while training, monitors the validation loss. Starts at learning rate=0.001, reduce learning rate by a factor of 2 if validation loss does not improve after 3 epochs, stop reducing learning rate if learning rate has reached the minimum=0.00001. It's a step decay.

Class Weights Re-weighting
Class weights were adjusted to combat imbalanced data set. Each of the class samples. Each sample of the class 0 (colorDefect) was treated as 4 samples, each of the class 3 (strangeObject) was treated as 3 samples, while the majority class was 1:1 .

2.2. Training and Testing methodology
The training and the testing process will be following this flow:

- **1:** Building the CNN architecture
- **2:** Simple refining on the architecture for more stable performances and customize it for the dataset (Investigating the uses of number of units in FC layers, Dropout, BatchNormalization), done based on the maximizing training and validation accuracy.
- **3:** Analyze the training history chart (loss, accuracy w.r.t. epochs generations)
- **4:** Select the best performing model to do evaluation on the test set. Evaluating accuracy and f1-score.
- **5:** Generate the confusion matrix in reference to this example and evaluate it [4].

At last, I will be comparing models based on the step **3, 4, 5**.

2.3. AlexNet

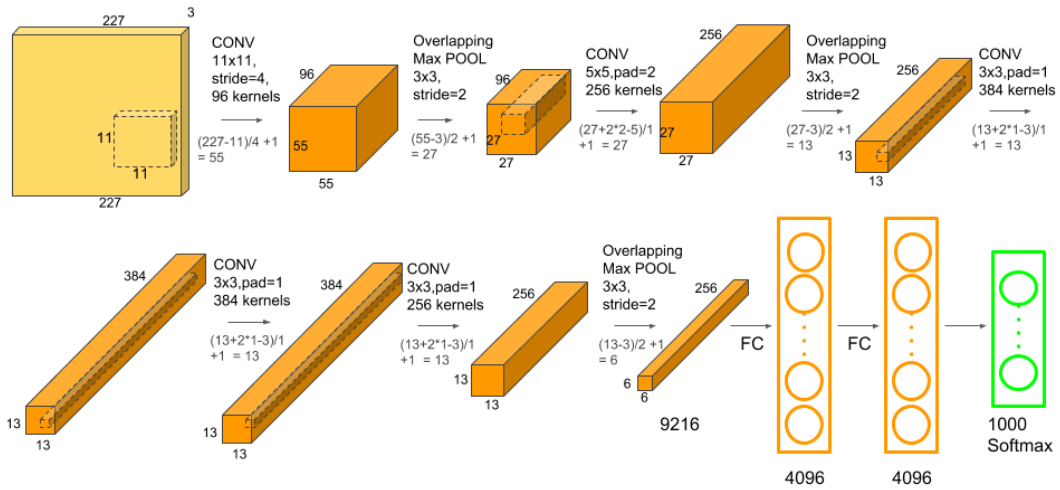

Figure 2.1: AlexNet Architecture

My AlexNet architecture was built based on the figure 2.1 and this kaggle notebook [6]. I made few changes on this architecture to adapt for my task. Firstly, the output layer is changed to 4 units, since the dataset has only 4 classes instead of 1000 classes in ILSVRC. Secondly, because my dataset is relatively short, for over 14 million trainable parameters, it's very easy to over-fit, therefore, I reduced the number of units in the 2 FC layers from 4096 to 1024. Dropout of 0.5 was added after them. BatchNormalization() after each CNN layers. The model takes input in shape (227, 227, 3).

The model was trained 20 epochs at the end. Reaching a validation accuracy of 0.96 . The training time was 55 seconds. Each epoch took 2 seconds. Along the training of AlexNet, I noticed the validation loss is quite unstable without batch normalization and the dropout. I have experimented run with/without batch normalization and dropout, it still produced some random spikes, and this behaviour worsened if the input image is in a smaller size (64, 64, 3).

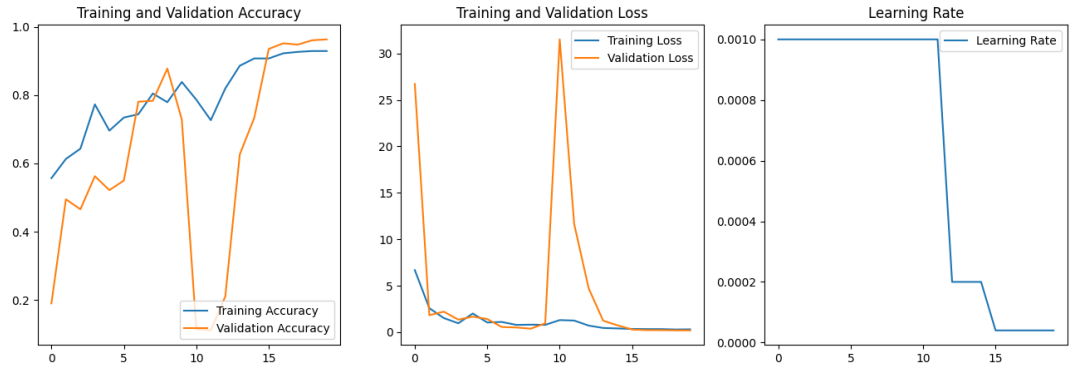

Figure 2.2: Training History

Then I conducted evaluation on the test set. It has overall good performances (0.96 accuracy and 0.96 f1-score). However, there are some interesting findings in the result.

Classification Report			
	precision	recall	f1-score
colorDefect	0.97	1.00	0.98
incomplete	0.97	0.95	0.96
normal	0.95	0.99	0.97
strangeObject	0.96	0.87	0.92
Accuracy			0.96
macro avg	0.96	0.95	0.96
weighted avg	0.96	0.96	0.96

In the table above, note that the recall for the for strangeObject is merely 0.87, which is lower than the average. Upon inspection on the confusion matrix, there are 15 samples in total were wrongly classified as incomplete or normal. Moreover, 12 incomplete cookies were classified as normal cookies.

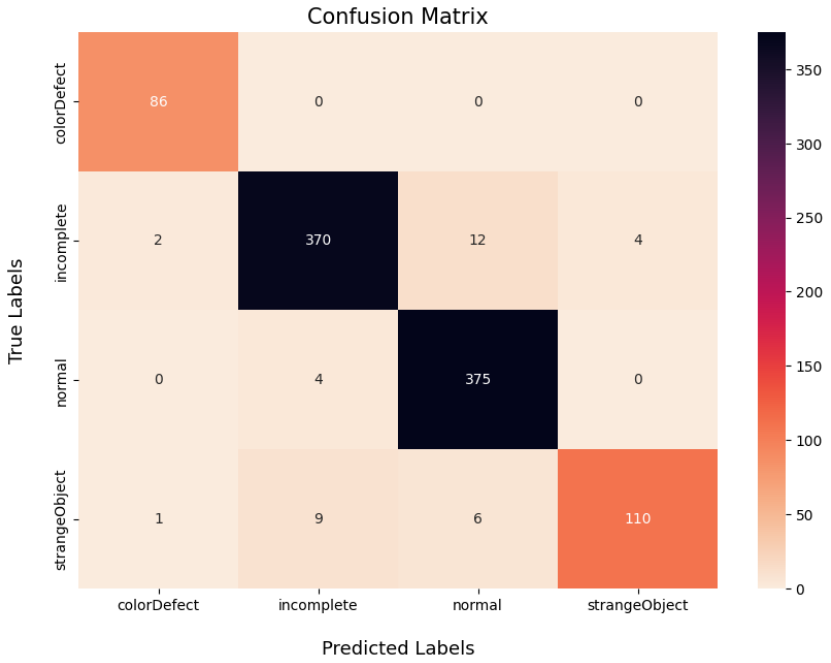

Figure 2.3: Confusion Matrix

Inspecting the test set, I have found this example, where it indeed belongs to strange object class (no class labelling error). It indeed has that foreign object behind the cookie, just that it is so small that it will very likely be classified as a normal cookie.

Figure 2.4: Strange Object example

At the end, the overall performance is good, albeit with some wrongly classified outliers. Next I will train again on the VGG16 network to see if it is any better.

2.4. VGGnet

I built the VGGnet architecture using tf.keras.application.VGG16() in reference to this kaggle notebook [7]. The overall structure looks like below [1]

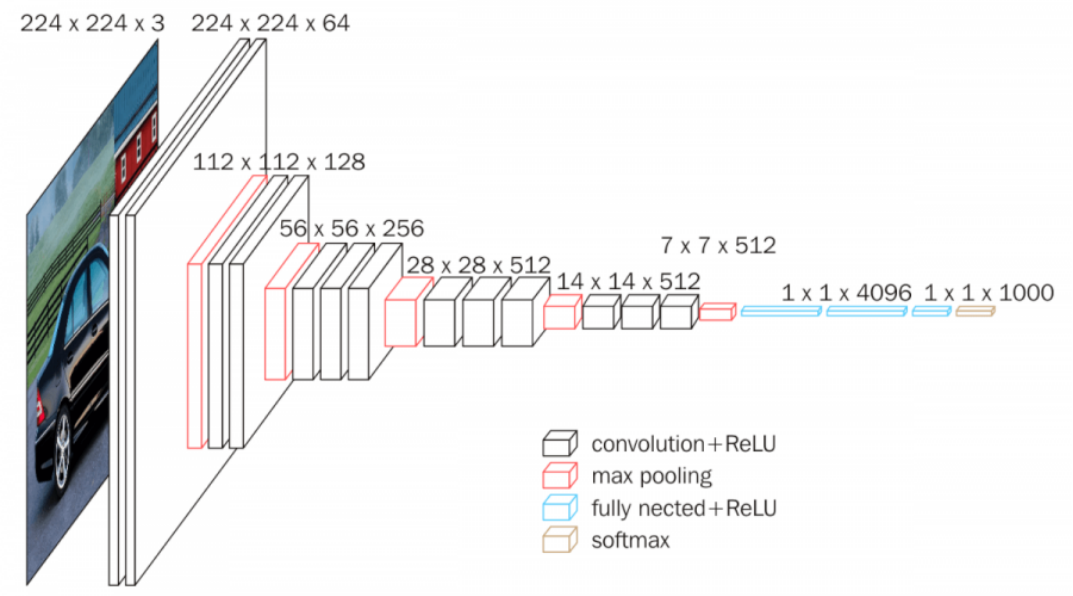

Figure 2.5: VGGNet Architecture

The VGG16 model for feature extraction was loaded using tf.keras.applications.VGG16(), then 2 fully-connected layers with 512 units, using relu as activation function and a output layer with 4 units, using softmax. I found using 2 fully-connected layers of 512 layers performs better than using Dropout() in combination with 1024 units in each layer. I have loaded VGG16 weights from ILSVRC to the feature extraction model, the convolutional layers before the FC layers were set to untrainable as well. The model takes input in shape (224, 224, 3), same to the original VGG.

The model was trained 15 epochs at the end. Reaching a validation accuracy of 0.98 . The training time was 94 seconds. The VGGnet model converges faster than the AlexNet, mostly because I have the weights loaded, however, it takes a considerable longer time to train than the AlexNet (94 seconds vs 55 seconds). Each epoch would take 6 seconds rather than 2 seconds. Another observation is VGGNet's training history is very stable compare to AlexNet, there is no spikes in the training history (Figure 2.6). It did not trigger the learning rate to decay. In theory I should let it run for 25-30 epochs to see it if further converge, but I have experimented that stopping at 15 epochs is a wise choice. Because it has reached steady state and showing excellent performance. More epochs with smaller learning rate does not necessarily improve accuracy, it might even make the model over-fit. The total number parameter is around 15 million.

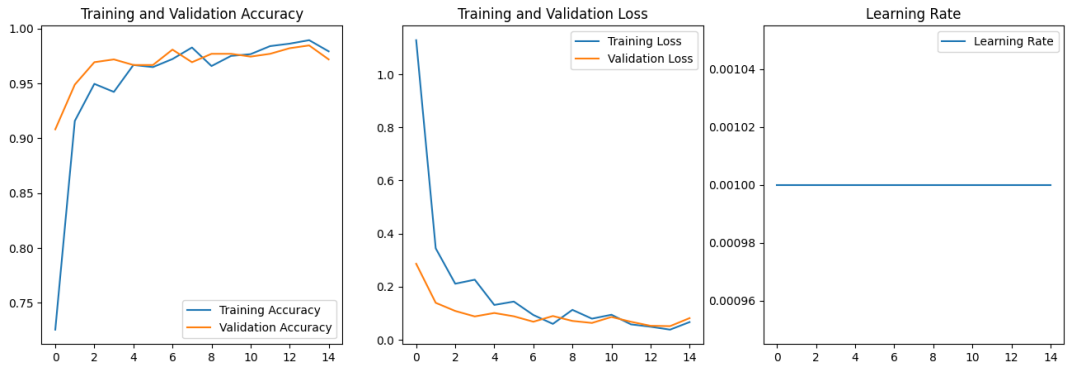

Figure 2.6: Training History

Then I conducted evaluation on the test set. It has overall extremely good performances (0.99 accuracy and 0.99 f1-score).

Classification Report			
	precision	recall	f1-score
colorDefect	1.00	1.00	1.00
incomplete	0.99	0.98	0.99
normal	0.98	1.00	0.99
strangeObject	0.99	0.97	0.98
Accuracy			0.99
macro avg	0.99	0.99	0.99
weighted avg	0.99	0.99	0.99

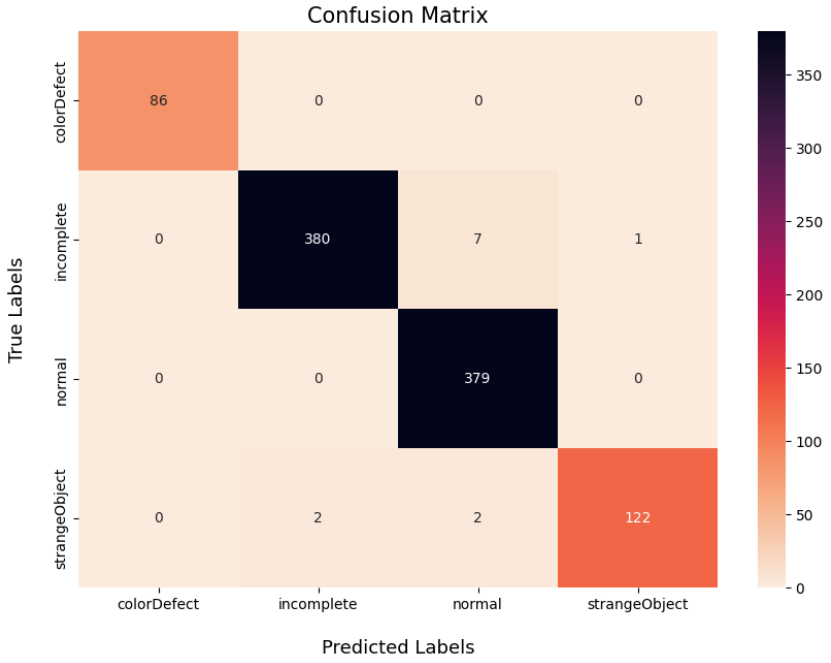

Figure 2.7: Confusion Matrix

In the confusion matrix, the hot spot [3,1], [3,2] (strangeObject classified as incomplete or normal) is greatly reduced, summing up to 4. Meanwhile, less incomplete samples were classified as normal.

In conclusion, the overall performance is extremely good, next I will try a simplified VGGNet model, to see if I can reduce the number of parameters and training time by exchanging some performances.

2.5. VGGnet Lite Version

The AlexNet and VGG16 above were used for ILSVRC before. In ILSVRC, it is built for a 1000 classes classification tasks. However, this is an overkill for my project of dataset with only 4 classes. Both AlexNet and VGG16 have multiple layers with over 10 billion parameters. Therefore, I decided to build a simpler VGGnet architecture with less convolutional layers based on this kaggle notebook [7].

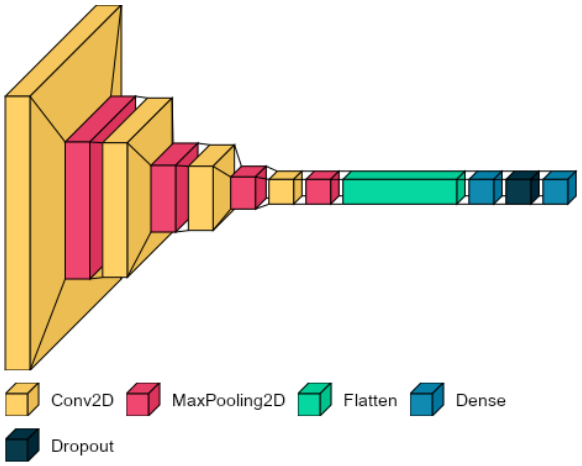

Figure 2.8: VGGNet Lite Architecture

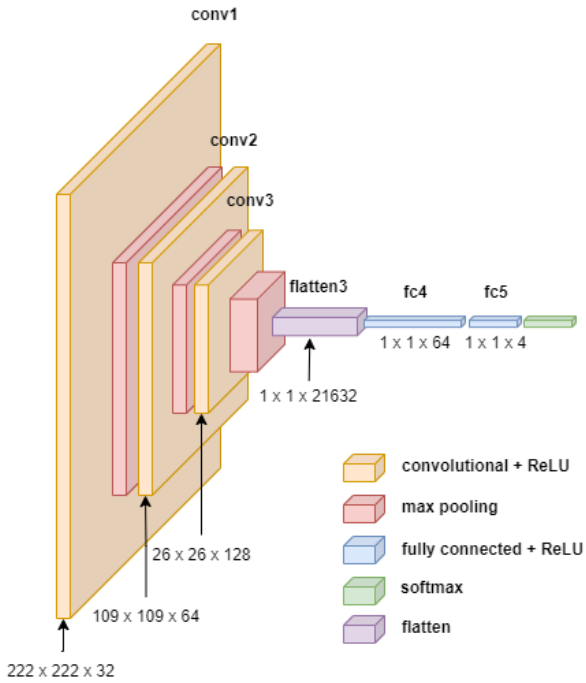

Figure 2.9: VGGNet Lite Actual implementation

Figure 2.9 shows the actual design of the VGGnet Lite, the figure is made based on the drawing template created by kennethleungty [3]. Conv4 and Conv5 were removed, and the number of convolutional layers is greatly reduced in each of the Conv stage, the number of filters in each of the convolutional layers is reduced by half as well. The total number of parameters is only 2 millions now. The model takes input in shape (224, 224, 3), the image size it reduced to (222, 222) after the first convolutional layer because no padding was added.

The model was trained with 15 epochs. Reaching a validation accuracy of 0.98 as well. The training time was 46 seconds, faster than both the previous models, I think this is mainly because the number of weights is 7 times lesser, but this time is still impressive when there's no load pre-trained weights loaded. Each epoch would take 3 seconds. The training history regarding validation and training loss is very stable as well, similar to the VGGnet.

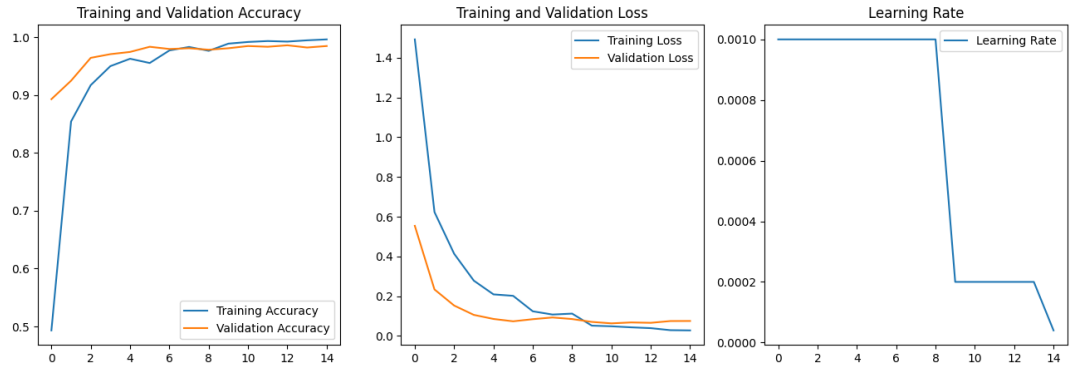

Figure 2.10: Training History

Conducting evaluation on the test set. It has overall good performances (0.98 accuracy and 0.98 f1-score). We see some trade-offs in performance here, the f1-score and accuracy is worse than the full VGGnet, albeit only by a little.

Classification Report			
	precision	recall	f1-score
colorDefect	0.99	0.90	0.94
incomplete	0.97	0.98	0.98
normal	0.97	1.00	0.98
strangeObject	1.00	0.94	0.97
Accuracy			0.98
macro avg	0.98	0.96	0.97
weighted avg	0.98	0.98	0.98

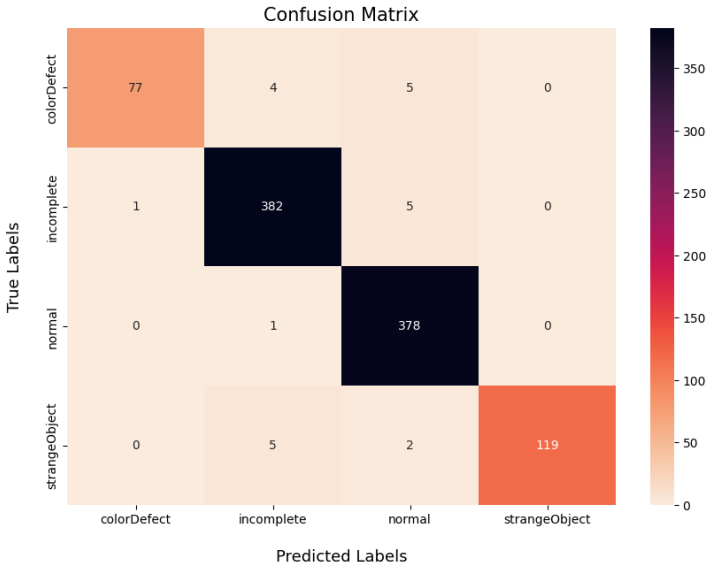

Figure 2.11: Confusion Matrix

There are no obvious hot stop in the confusion matrix besides the truth diagonals. Except the true-incomplete versus predicted-normal cell, but this is always the same area where the original VGGnet and AlexNet fall short.

The overall performance is a little bit worse than the original VGGNet, but I think it is still performing very good, and better than the AlexNet as well.

2.6. Demo

Below shows a demo of the model predicting cookie images. The model used is VGGNet Lite with an accuracy of 98%. The text above shows the model's prediction and its confident about it.

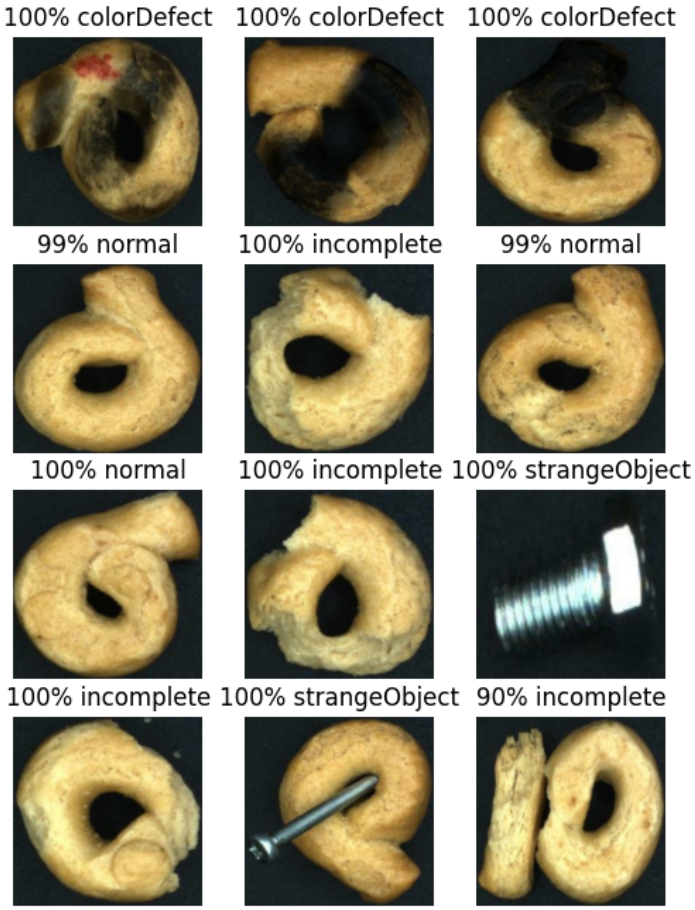

Figure 2.12: Prediction on some images

3
Comparison of 3 models and Conclusions

3.1. Cross Comparison

	AlexNet	VGGNet	VGGNet Lite
Trainable Params	14,241,860	15,242,052	2,862,788
Accuracy (Test)	0.96	0.99	0.98
f1-score (Test)	0.96	0.99	0.98
Time(seconds)/epoch	2	6	3
Total Time(seconds)	55	94	46

Judging on this table, VGGnet Lite is a better choice for this specific dataset, mainly because its quicker training time and significant fewer number of parameters. Although the advantage in the total training time is rather small, but this advantage in training time will scale significantly w.r.t. a larger training dataset. Such as the the dataset in the PA2 (42000 images).

3.2. Conclusion

Lastly I think VGGnet Lite is the best choice to do multi-class classification on the cookies dataset. To conclude the report, my main takeaway is bigger or more powerful architecture does not necessarily be the best model to a specific dataset, it depends on the nature, the quality, the size of the dataset. While it might have absolute advantage on the metrics (accuracy or f1-score), but more often it is overkill for most of the simpler tasks out there. It is then more wise to go with a smaller model with little to no trade-off in model performances to save computation power and time.

3.3. Extension

I understand that, there are way more model settings to tune to optimise certain model performances. Which means the 3 models in my report could be tuned even better, to the point where they are performing the same. But there exists too much combinations and would require much more time.

Moreover, during my task on this dataset, I found wrongly labelled samples, like image for incomplete cookies marked for color defect. While I guess it's the author injecting some noises in training, but I think it would confuse the model when the dataset it so small. Consider although there are 10 wrongly labelled images on the colorDefect class, it is already 10% of the samples in the colorDefect class, which would surely confuse the model. My further investigation could be, using Convolutional Autoencoders to do defect detection (taught in Autoencoders lecture). Because Autoencoders can be an unsupervised task, then when the labelling data is not reliable, it could avoid the model being confused.

References

[1] Muneeb ul Hassan. *VGG16 – Convolutional Network for Classification and Detection*. 2018. URL: https://neurohive.io/en/popular-networks/vgg16/.

[2] Karel Horak et al. *Industry Biscuit (Cookie) dataset*. 2022. DOI: 10.34740/KAGGLE/DSV/4311115. URL: https://www.kaggle.com/dsv/4311115.

[3] Kennethleungty. *GitHub - kennethleungty/Neural-Network-Architecture-Diagrams: Diagrams for visualizing neural network architecture (Created with diagrams.net)*. URL: https://github.com/kennethleungty/Neural-Network-Architecture-Diagrams.

[4] Nkitgupta. "Evaluation Metrics For Multi-class Classification". In: *Kaggle* (Mar. 2021). URL: https://www.kaggle.com/code/nkitgupta/evaluation-metrics-for-multi-class-classification.

[5] *TU Delft Unofficial Report/Thesis Template*. URL: https://www.overleaf.com/latex/templates/tu-delft-unofficial-report-slash-thesis-template/swythjmksywm.

[6] PARAS VARSHNEY. *AlexNet Architecture: A Complete Guide*. 2020. URL: https://www.kaggle.com/code/blurredmachine/alexnet-architecture-a-complete-guide/notebook.

[7] ALBERTO VIDAL. *Defect detection using CNN with keras & tensorflow*. 2023. URL: https://www.kaggle.com/code/albertovidalrod/defect-detection-using-cnn-with-keras-tensorflow.