CA2 - Food Image Classification

29-Sep-2019 | Version 1.0

Team Name: A22G

Alfred Tay Wenjie
Wang Zilong
Wong Yoke Keong

Prepared for NUS-ISS Master of Technology in Intelligent Systems ISY5002 Pattern Recognition and Machine Learning Systems Group Project

Table of Contents

1. Introduction

Singapore is considered a food haven and a favourite national topic by many spawning popular food blogs and long queues across the island. Given the huge focus on food and growing emphasis on healthy lifestyle, there is great potential in applying Computer Vision Deep Learning techniques to food classification. In this assignment, the A22G team showcases the motivation behind the project and the effectiveness of Image Classification with Convolutional Neural Networks and its variants.

The team also strives to share with the reader the team's journey to develop highly effective Deep Learning vision models while overcoming challenges like overfitting and training deep models by applying best practices. The team would also like to share empirical nuggets of wisdom gained from this journey when implementing various network architecture from scratch and techniques employed.

2. Project Scope

Proposed Solution

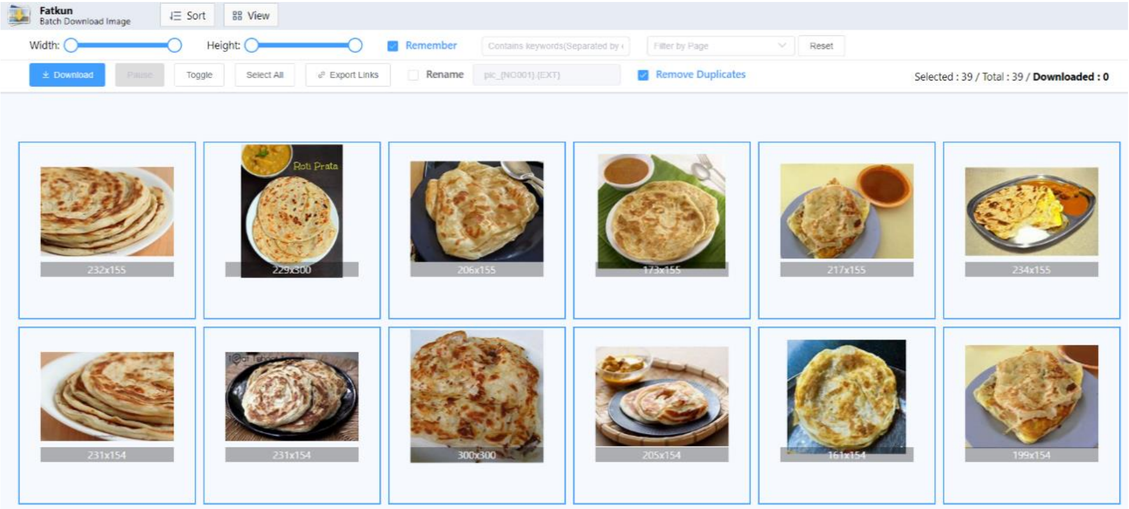
The larger proposed solution is a food diary and nutrition manager which will keep track of nutrition and calories intake. The solution will be in a form of a mobile application which the user will use to take pictures of their meals and image recognition will recognise the food item and record the nutrition and calories intake. After all, a lot of people already has the habit of "feeding their phone first" before they start eating. The scope of this CA focus on developing the image recognition module which is able to recognise three common local food, namely, Chicken Rice, Bak Chor Mee (minced pork noodle) and Roti Prata.

Value Proposition

The value proposition of the solution is that it will help users manage their nutrition needs by suggesting nearby food options which will help them eat a balanced diet for the day. For example, if the user has had bread for breakfast and roti prata for lunch, the application may suggest a nearby mix vegetable brown rice for dinner which will come with portions of vegetable and meat. The application could potentially be connected to local food ordering network to provide added convenience to the user.

3. Dataset Construction

Food images were collected from the Internet using various search engines (e.g. Google, Yahoo). An image mass download tool, Fatkun was used to select and mass download the images.



A total of 3,428 images of Chicken Rice, Bak Chor Mee and Roti Prata was collected.

Food Item	Number of Images
Chicken Rice	1175
Bak Chor Mee	1190
Roti Prata	1063
Total	3428

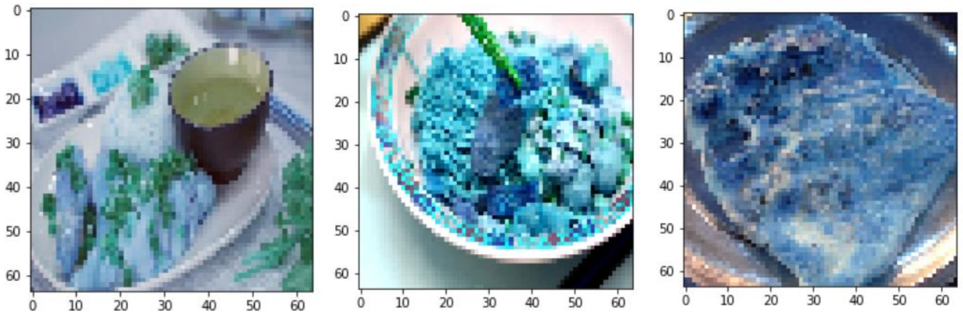
The Dataset is partitioned into Training Dataset and Testing Dataset.

Dataset	Number of Images	Percentage
Training Dataset	2860	83.5%
Testing Dataset	568	16.5%
Total	3428	100.0%

4. Dataset Pre-processing

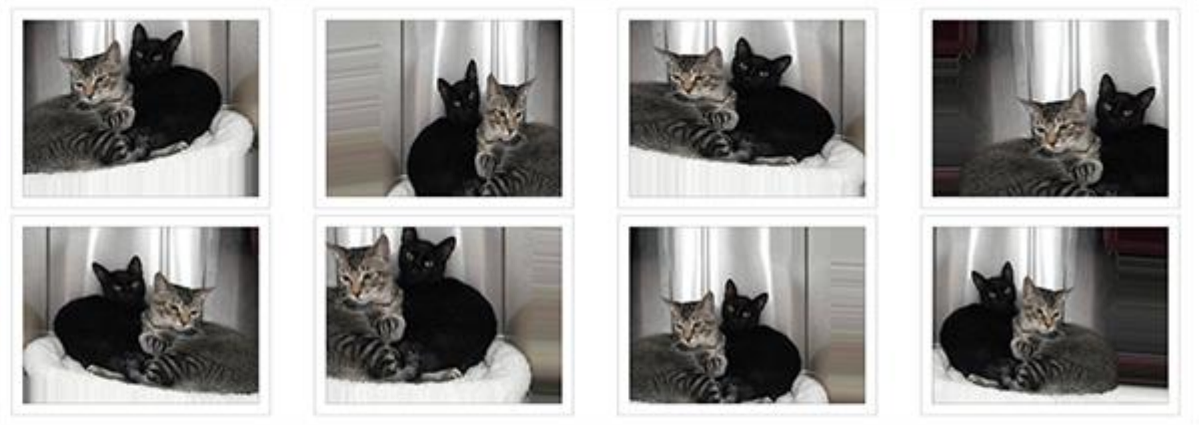
Image Resizing and Packaging

The images dataset was downsized to 64 by 64 pixels and packaged into Pickle file for ease of sharing and access. Different image sizes were tested, and accuracy of the various models did not show improvement for images larger than 64 by 64 pixels.



Data Augmentation

Image Augmentation refers to the introduction of noise through randomized applications of image transformations like rotation, shear, zoom and flips (horizontal and/or vertical) as shown in the diagram below.



Source: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

Note that these augmentations are typically done in real-time like in the case of Keras through the use of the ImageDataGenerator class and applied only on the training data.

Slowness of ImageDataGenerator vs 1-time pre-processing **-** One observation is that the use of ImageDataGenerator with fit_generator method in Keras is very slow. It seems that there are a couple of open issues and on Github as the report is written.

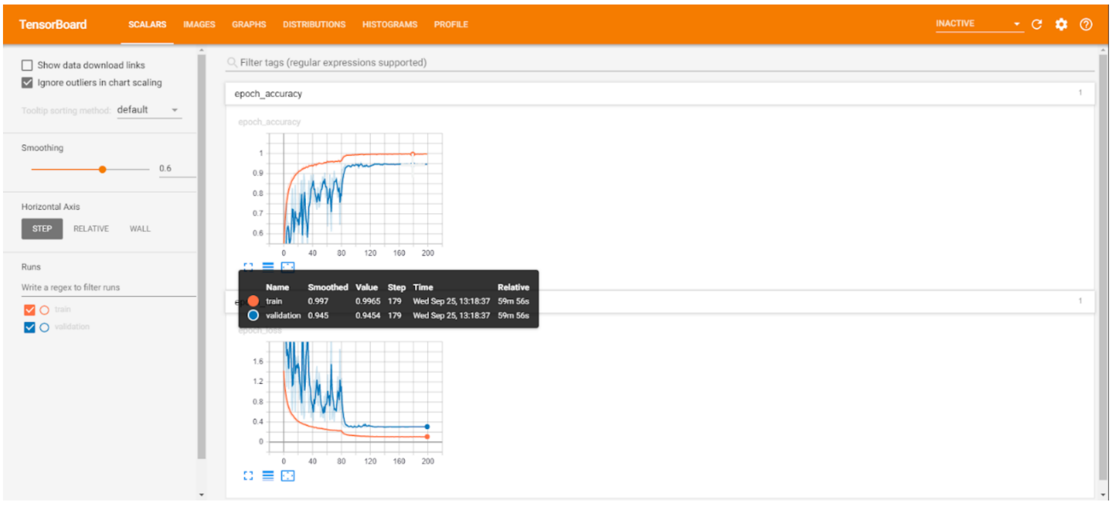
When training the Inception and Inception ResNet-based network described later, the observation is that each epoch takes around 50+s regardless of model size at Cycle 1 (1M+ parameters) or Cycle 11 (11M+ parameters). It has been observed that if the image data is not read directly from image files using flow_from_directory method but pre-processed into numpy arrays and the flow method is used instead, each epoch only requires 8s on average. Please note that these timings is based on the Nvidia Tesla T4 on Google Colab.

Testing of suggested workarounds like setting max_queue_size=100 and the use of multiprocessing with 2 threads does not solve the issue. Moreover, it has been observed that the enabling of multi-processing causes the training to stop at random epochs, typically at the first or last step of the impacted epoch.

The takeaway is that 1-time pre-processing of images before augmentation will help speed up training with minimal impact.

5. Model Training Visualisation Tool

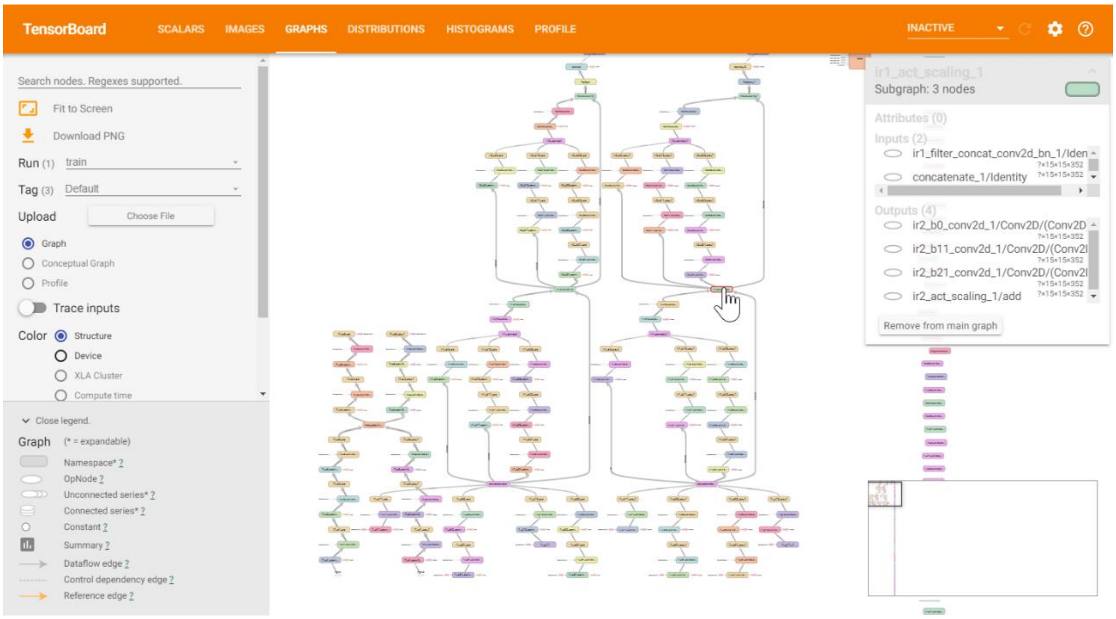
In addition to static visualization, the team also explored the use of interactive visualization through the use of Tensorboard, which is provided as a standard callback by Keras. It is very useful for tracking loss and accuracy per epoch in an interactive manner as compared to static images.



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis: STEP
☑ ○ train
☑ ○ validation

	Name	Smoothed	Value	Step	Time	Relative
●	train	0.997	0.9965	179	Wed Sep 25, 13:18:37	59m 56s
○	validation	0.945	0.9454	179	Wed Sep 25, 13:18:37	59m 56s

Tensorboard also allows an operational view of the graph generated. In complex models like InceptionResnet, where there are multiple connections coming from the various concatenated convolutional layers and residual connections, it is very convenient to just click on the particular layer (for example the Lambda layer featuring the residual connection and concatenate layer and see its inputs and outputs, along with their respective parameter settings as shown in the diagram below.



Run (1) train
Tag (3) Default
● Graph
○ Conceptual Graph
○ Profile
Color ● Structure
○ Device
○ XLA Cluster
○ Compute time

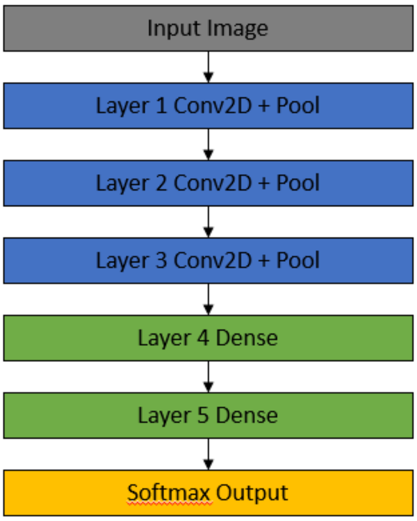
Graph (* = expandable)
Namespace* ?
OpNode ?
Unconnected series* ?
Connected series* ?
Constant ?
Summary ?
Dataflow edge ?
Control dependency edge ?
Reference edge ?

ir1_act_scaling_1
Subgraph: 3 nodes
Attributes (0)
Inputs (2)
ir1_filter_concat_conv2d_bn_1/Iden ?×15×15×352
concatenate_1/Identity ?×15×15×352
Outputs (4)
ir2_b0_conv2d_1/Conv2D/(Conv2D ?×15×15×352
ir2_b11_conv2d_1/Conv2D/(Conv2I ?×15×15×352
ir2_b21_conv2d_1/Conv2D/(Conv2I ?×15×15×352
ir2_act_scaling_1/add ?×15×15×352

This allows practitioners to understand the architecture and make it easier to troubleshoot issues visually.

6. Convolutional Neural Network

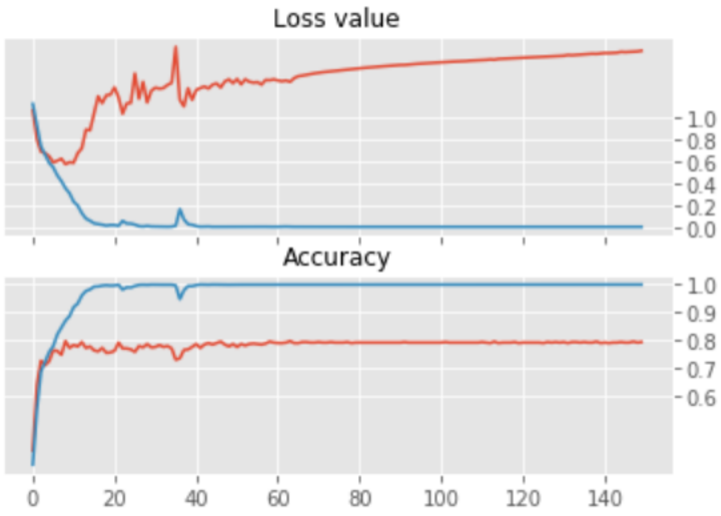
The Convolutional (Conv) layer automates the feature extraction from the image which greatly simplifies the task of building an image recogniser without the need to have specialised knowledge in image processing. The team started out to build a basic CNN with 3 Conv layers and 2 Dense layers.



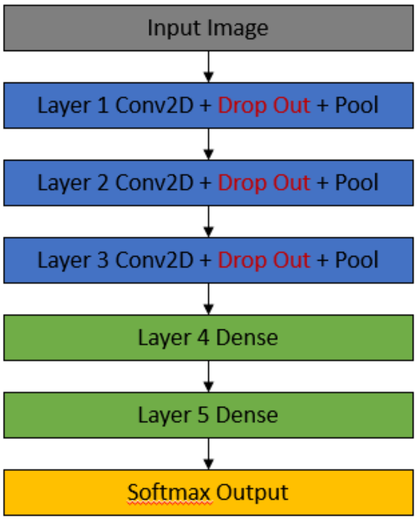
Relu activation function was used for the all the 3 Conv layers and the first Dense layer to avoid gradient vanish which occurs in Sigmod activation function.

Softmax activation function was used for the last Dense layer to classify the image into 1 of the 3 classes.

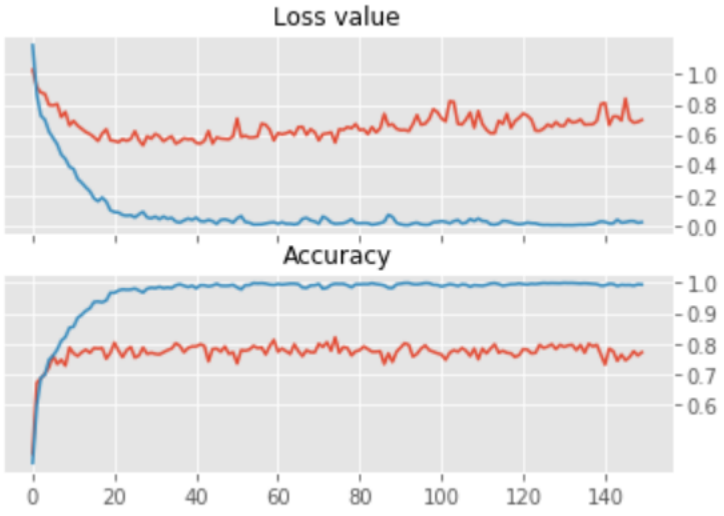
This first model achieved an accuracy of 79.75% and based on the loss value graph below, it looks like the model has overfitted.



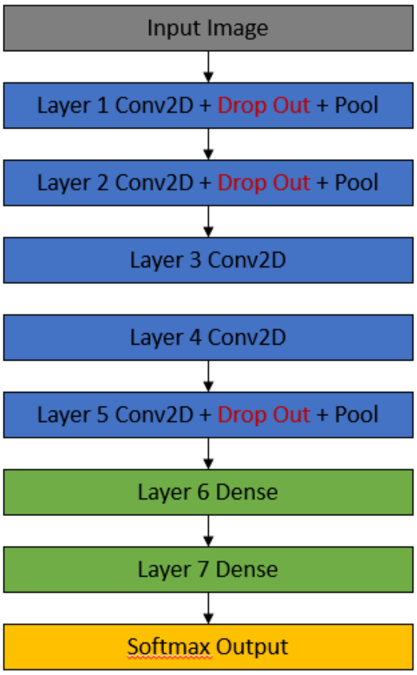
To solve the overfitting issue, drop out was added after each conv layer to achieve better generalisation.


Input Image
Layer 1 Conv2D + Drop Out + Pool
Layer 2 Conv2D + Drop Out + Pool
Layer 3 Conv2D + Drop Out + Pool
Layer 4 Dense
Layer 5 Dense
Softmax Output

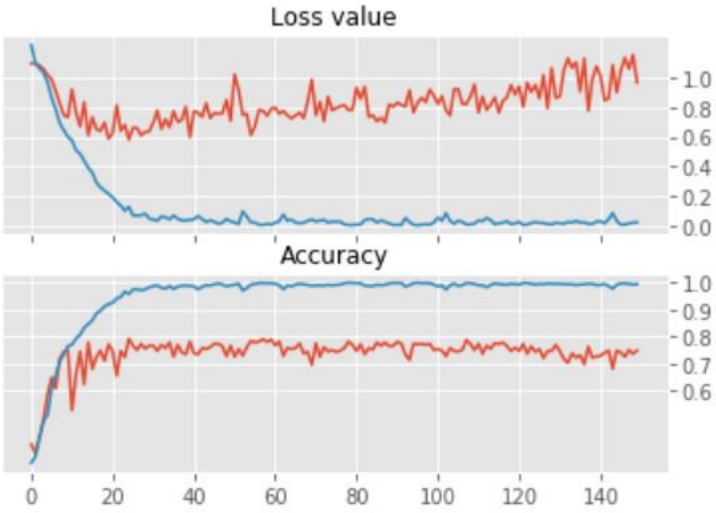
The validation loss value has decreased, and accuracy has increased to 82.04%.



We explored going deeper by adding 2 more Conv layers.


Input Image
Layer 1 Conv2D + Drop Out + Pool
Layer 2 Conv2D + Drop Out + Pool
Layer 3 Conv2D
Layer 4 Conv2D
Layer 5 Conv2D + Drop Out + Pool
Layer 6 Dense
Layer 7 Dense
Softmax Output

The model became unstable and accuracy dropped to 80.11%. More measures needs to be put in to build a deeper net.

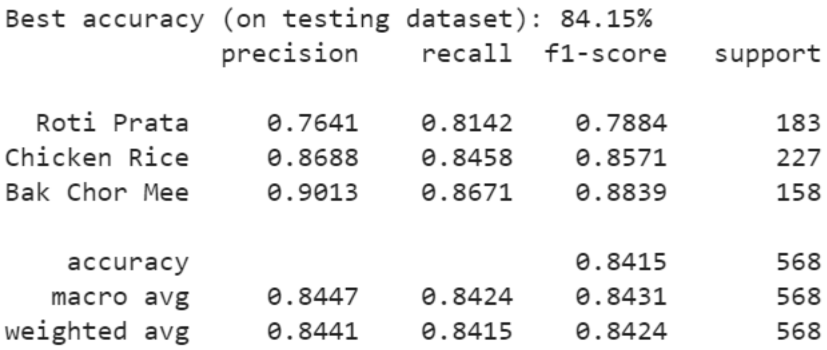


7. ResNet

The basic CNN provided a reasonable accuracy of 82.04% but we would like to see how deep we could go with this dataset (In general, a deeper net requires larger amount of data to train). We next set out to build a 20 layer ResNet.

To prevent overfitting in such a deep neutral network, Regularization and Batch normalization was deployed.

The accuracy improved to 84.15%.

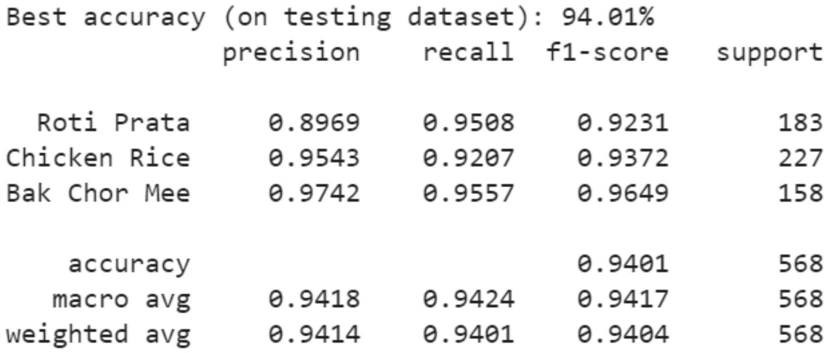


```
Best accuracy (on testing dataset): 84.15%
              precision    recall  f1-score   support

  Roti Prata     0.7641    0.8142    0.7884       183
Chicken Rice     0.8688    0.8458    0.8571       227
Bak Chor Mee     0.9013    0.8671    0.8839       158

    accuracy                         0.8415       568
   macro avg     0.8447    0.8424    0.8431       568
weighted avg     0.8441    0.8415    0.8424       568
```

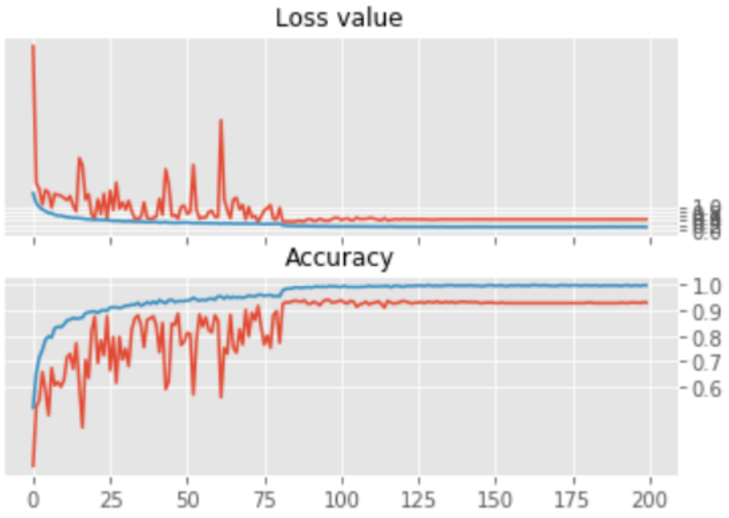
Next we set out to try 32 layers and 44 layers. The 20 layers ResNet still has the highest accuracy.

Number of layers	Accuracy
ResNet (20 layers)	84.15%
ResNet (32 layers)	82.75%
ResNet (44 layers)	83.45%

Fixing the number of layers to 20, we added image augmentation and learning rate scheduler. The accuracy shot up to 94.01%



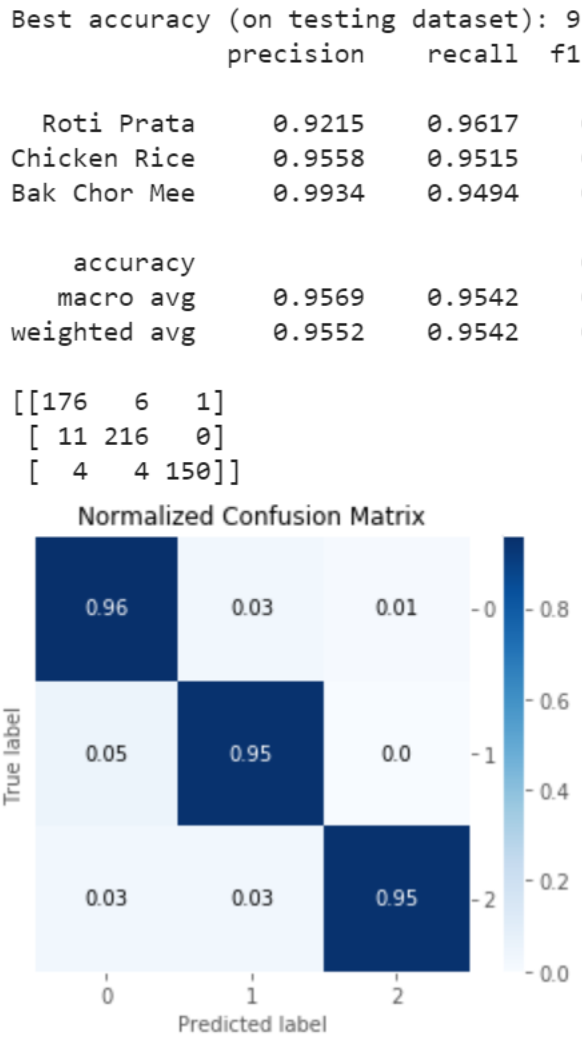
```
Best accuracy (on testing dataset): 94.01%
              precision    recall  f1-score   support

  Roti Prata     0.8969    0.9508    0.9231       183
Chicken Rice     0.9543    0.9207    0.9372       227
Bak Chor Mee     0.9742    0.9557    0.9649       158

    accuracy                         0.9401       568
   macro avg     0.9418    0.9424    0.9417       568
weighted avg     0.9414    0.9401    0.9404       568
```



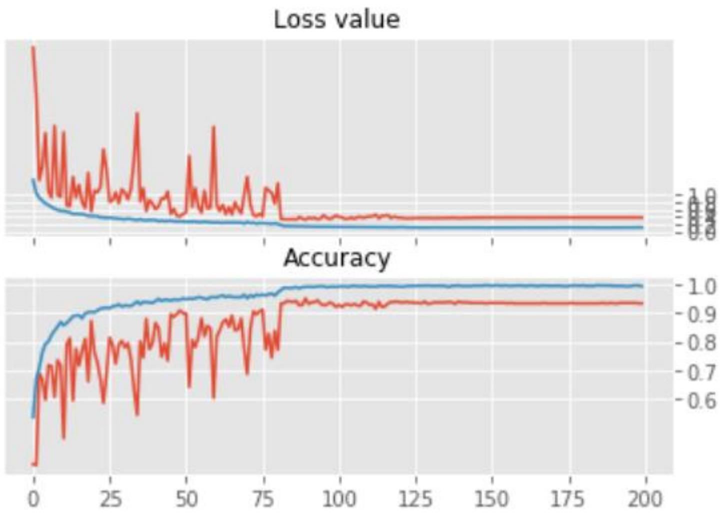
Next we performed hyper parameter tuning and found the below hyper parameters which gave the highest accuracy.

Hyper Parameter	Value
Initialisation	He Initialisation
Filter size	32,64,96
Batch size	48

The final ResNet (20 layers) model achieved an accuracy of **95.42%.**



```
Best accuracy (on testing dataset): 95.42%
              precision    recall  f1-score   support

  Roti Prata     0.9215    0.9617    0.9412       183
Chicken Rice     0.9558    0.9515    0.9536       227
Bak Chor Mee     0.9934    0.9494    0.9709       158

    accuracy                         0.9542       568
   macro avg     0.9569    0.9542    0.9552       568
weighted avg     0.9552    0.9542    0.9544       568

[[176   6   1]
 [ 11 216   0]
 [  4   4 150]]
```



8. Inception & Inception ResNet

With the experience and learning from the standard convolutional neural networks, one track the team explored are alternative architectures like Inception and Inception ResNet (i.e. Inception with residual connections).

This section shows the various attempts made to improve the Inception-architecture based model (with different parameters) and shows the long march to 10x deeper networks, while combating overfitting and the fickleness (instability) of neural networks.

A bite-sized overview of the attempts and the impact on accuracy and other factors like network stability ('Swings' in loss values) will first be illustrated through a table and nuggets of learning shall be highlighted afterwards. As the main metric, new highs achieved on accuracy shall also be shown in **bold** during the journey.

Process Taken and Finetuning

The below shows the summary of process taken in 13 steps or 'cycles'.

Cycle	Description of Additions/Replacements made	Accuracy and Impact
1	• Used simplified dimension reduction inception module with use of concatenate layer • Used ImageDataGenerator + Augmentation • Used adam optimizer, default learning rate • Number of epochs = 100 • Batch size = 128	Base accuracy of 73.11%; 1,181,215 parameters (total = trainable)
2	• Replaced simplified inception module with configurable inception module based on Inception 3a module	**78.85%**; Swing: relatively stable

	• Addition of Convolution 'stem' made of Conv2D and MaxPooling2D layers	
3	• Replace inception module with configurable InceptionResnet 35x35 block A • Understanding of InceptionResnet Layer with Lambda Layer • Implement Conv2D with BatchNorm to support InceptionResnet module	79.27%, similar accuracy (within 0.5%)
4	• Stack 1 inception module and 1 InceptionResnet module together	78.43%, accuracy dropped; Swing: huge swings in loss value

5	• Replaced Convolution 'stem' to Conv2D + BatchNorm
	• Added L2 Regularization (0.00005) to Conv2D with BatchNorm layer only (not Dense)
		81.09%;
		First time broke 80% at epoch 97;
		Swing: smaller swings relatively;

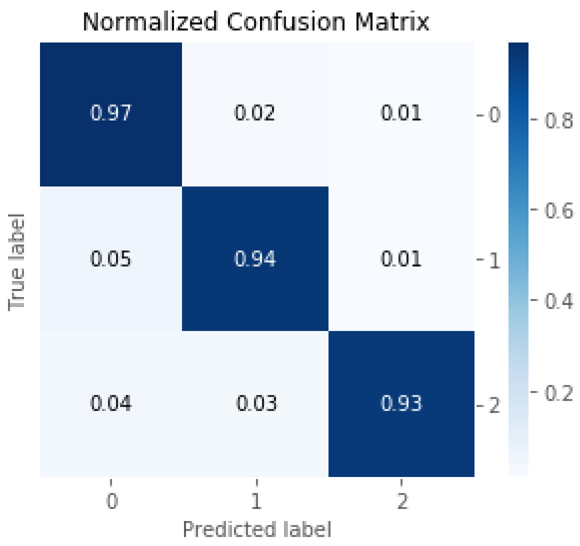
6	• Changed kernel initialization to He Normal • Change stack to 1 Inception module and 2 InceptionResnet modules • Added L2 Regularization (0.00005)to Dense layers	**83.43%**; Broke 80% at epoch 41, Swing: more stable loss trends than at cycle 5
7	• Increased image size from 32x32 to 48x48	**85.81%**; Broke 80% at epoch 52
8	• Added learning rate scheduler • Increased number of epochs from 100 to 150	83.29%, accuracy dropped; Swing: Smaller swings, stable after epoch 80; Broke 80% at epoch 93, slower increment
9	• Replaced adam optimizer with rectified adam optimizer	**85.11%**; Broke 80% at epoch 60, faster increment comparable to no learning rate scheduler
10	• Increased number of epochs from 150 to 160 • Reduced batch size from 128 to 32	**89.61%**; Broke 80% at epoch 25 Swing: Medium swings before epoch 80, stable after
11	• Increased number of epochs from 160 to 200 • Increased batch size from 32 to 48 again	**92.60%**; Broke 80% at epoch 18;

	• Change feed direct from image directory to 1-time read and pre-processed pickle file • Image size increased from 48x48 to 64x64	Training time per epoch dropped drastically from 50s to 8s on average using Tesla T4
12	• Decreased batch size from 48 to 32	**93.16%**, slight increase Broke 80% at epoch 21; Swing similar to cycle 10
13	• Use cleaned and de-duplicated image source as base with a new pickle file to be loaded	**94.19%**, final repeat run at **94.54%**, slightly better performance Broke 80% at epoch 19 11,691,363 parameters, 11,687,267 trainable

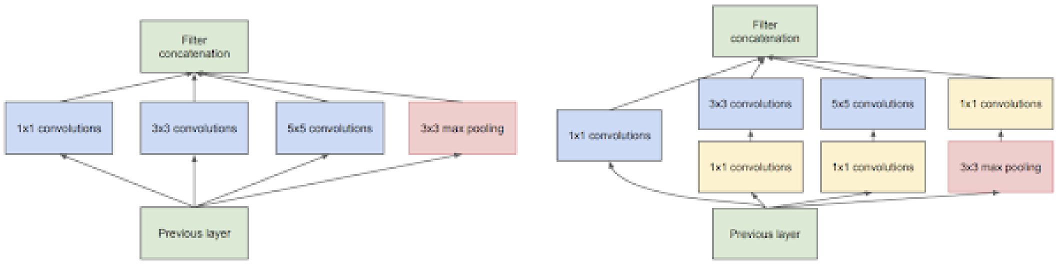
Model Performance

On the final repeat run, the model achieved an accuracy of 94.54% with the following confusion matrix and f1-scores.

```
Best accuracy (on testing dataset): 94.54%
              precision    recall  f1-score   support

  Roti Prata     0.9031    0.9672    0.9340       183
Chicken Rice     0.9638    0.9383    0.9509       227
Bak Chor Mee     0.9735    0.9304    0.9515       158

    accuracy                         0.9454       568
   macro avg     0.9468    0.9453    0.9455       568
weighted avg     0.9469    0.9454    0.9456       568

[[177   4   2]
 [ 12 213   2]
 [  7   4 147]]
```



Understanding the Inception Module



(a) Inception module, naïve version

(b) Inception module with dimension reductions

The journey begins with understanding what does Inception refers to. Referring to the diagram from the now-famous paper ‘Going Deeper with Convolutions’ by Szegedy et al [1]. we see that Inception refers to the stacking of variants of ‘Inception modules’, where each module is a horizontal combination of different convolutional layers, much like the similarly titled movie where the protagonists enter deeper and deeper levels of dream sequences.

To understand how this will be coded in practice, inspiration is taken from [2] and [3] to create a simplified dimension reduction Inception module using Concatenate and Conv2D Keras layers without the single 1x1 convolution tower, and subsequently the configurable Inception Block. Combined with image augmentation and the use of adam optimizer thus established the baseline of 73.11% with 1,181,215 parameters (total/trainable) Subsequently, the Inception 3a module structure presented in [1] and [3] is used.

Understanding the Inception Resnet Module

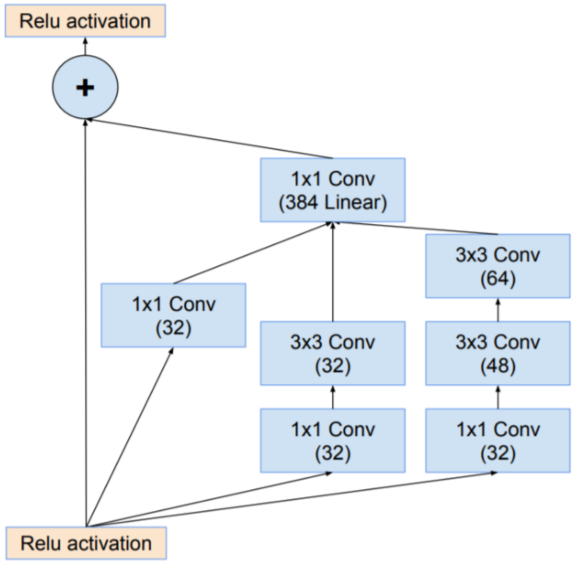


Figure 16. The schema for 35 × 35 grid (Inception-ResNet-A) module of the Inception-ResNet-v2 network.

Building on the understanding Inception, further research leads us to the 2016 paper ‘Inception-v4, InceptionResNet and the Impact of Residual Connections on Learning’ [4] by Szegedy et al on building in residual connections into the inception module and its variants. For the purpose of this project, the focus is on building the Inception-ResNet-A, shown above from the paper in [4] with reference to the Keras Implementation in [5]

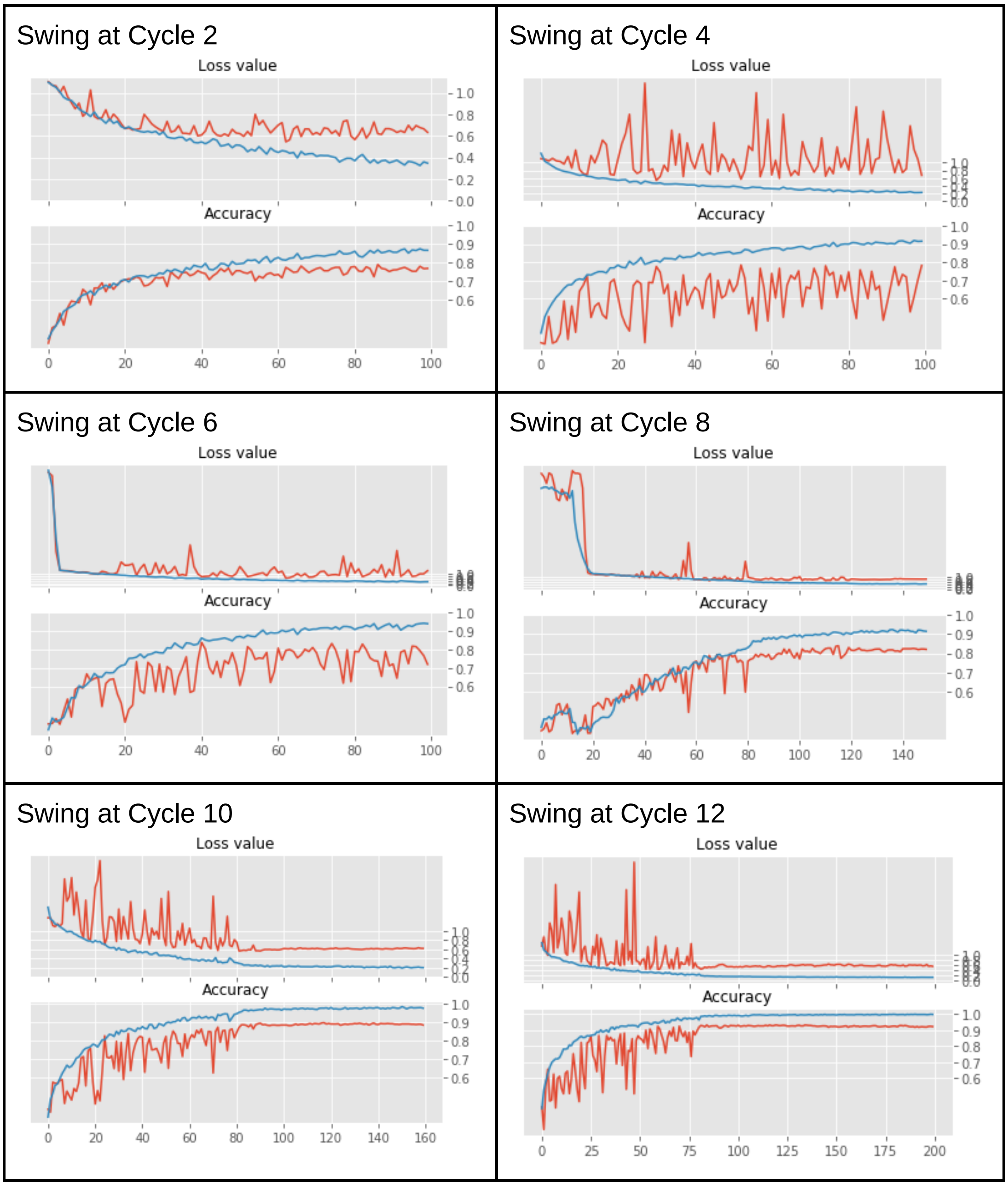
The residual connection is implemented through the use of a Lambda Keras layer where outputs from the new Inception Module are added with a scaled product of the previous Relu activation outputs.

Difference from the previous Inception Module includes the standardized use of 1x1 and 3x3 convolution layers (no more 5x5), where each convolution layer are in fact combined Conv2D layers coupled with BatchNormalization and regularization. Early in the journey, regularization is removed to observe the impact, which leads to the next highlight.

Impact of Regularization - It has been observed that just using Inception or Inception Resnet modules does not make much impact (See cycle 2 and 3). Plainly adding layers can only bring you so far, and stacking the Inception blocks makes accuracy worse (Cycle 4). It can be deduced that the effect of Batch Normalization is very small and residual connections helps more in later layers not overfitting.

Adding Dropout and L2 regularization increases accuracy to 80+%, which is a significant jump. It also contributed to reduction in loss value swings, hence the stability of learning. At this point it brings us to another highlight.

Stability of Neural Network Learning and Swings in Loss Values - Illustration of Unstable 'Swings' in loss values



Swing at Cycle 2

Swing at Cycle 4

Swing at Cycle 6

Swing at Cycle 8

Swing at Cycle 10

Swing at Cycle 12

The illustration above shows the swings in loss values at even cycles. The wild swings in loss values are an indication of poorly conditioned neural networks, where small changes in values results in large swings in classification labels, hence high accuracy variance. Addition of more layers (cycle 4) also resulted in overfitting. Addition of regularizations mentioned early makes the swings smaller (Cycle 6) and reduced overfitting.

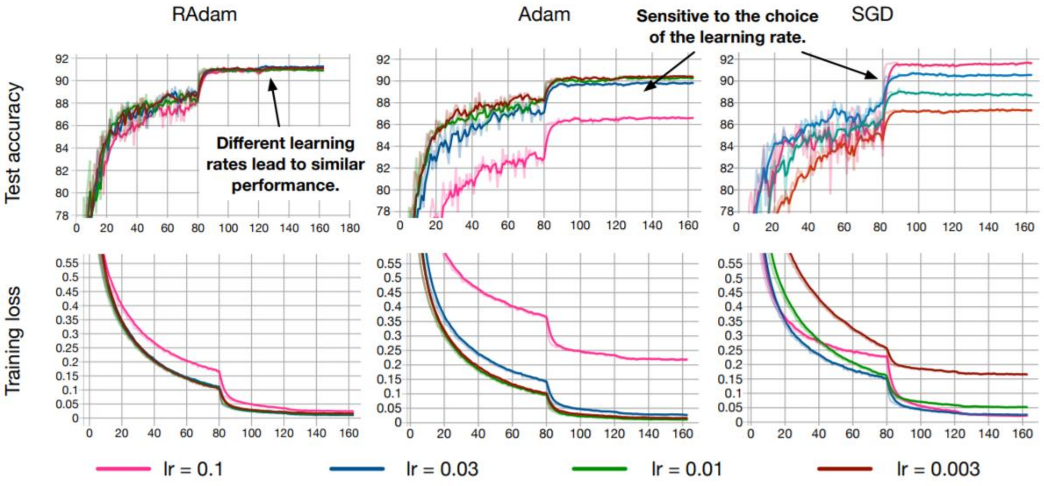
Addition of learning rate scheduler also enhances learning and stability (See Swings at cycle 8, 10 and 12), however this seems to be done at the expense of slower learning as seen at cycle 8. At first glance, the network seems to be less accurate even with longer training time at 150 epochs. However, as later cycle results show, this is more likely due to slower increase in accuracy due to smaller step sizes taken.

Intuitively, if we can see better results earlier, we can have a more stable network, make more changes and iterations and thereby moving faster. Thus this brings us to the next highlight.

Rectified Adam - Before adding the rectified adam implementation by HG Zhao (CyberZHG) [6], high accuracy (considered as 80% in during this journey) needs to take more than 40 epochs. With the introduction of Learning Rate Scheduler, we only see accuracy greater than 80% at epoch 97.

With rectified Adam, the first time the training broke 80% is at a considerably faster pace at epoch 60.

To understand what is happening, we need to look to the paper 'On the Variance of the Adaptive Learning Rate and Beyond' by Liu et al [7] where the team noted a weakness in adam optimizer that caused slower convergence due to undesirably big variance in the adaptive learning rate in early stage training. Liu et al proposed a rectification of this variance issue and allowing the learning rate to 'warm up' more quickly, as seen in the figure below from the paper, where optimization becomes less dependent on the choice of learning rates.


RAdam

Adam

SGD

Sensitive to the choice of the learning rate.

Different learning rates lead to similar performance.

Empirically, when coupled with a reduction in batch size, 80% accuracy is reached at much earlier epochs (around epoch 20 +/- in cycles 10 to 13 vs epoch 90+ at cycle 8). Coupled with the learning rate scheduler, both higher accuracies without using more than 200 epochs and less swings in loss values are achieved.

Deeper Networks with He Normalization - It is observed that deep models can be trained to achieve even higher accuracy (Cycle 6) with He Normalization even though another Inception ResNet Block has been added. It can be inferred that He Normalization, in tandem with Residual Connections and Batch Normalization enables the training of deeper networks with less impact from overfitting.

Increase of image size - Increasing image size per epoch increases accuracy. A good approach is thus to use smaller images to train and tune the model then use larger image sizes to realize gains in accuracy.

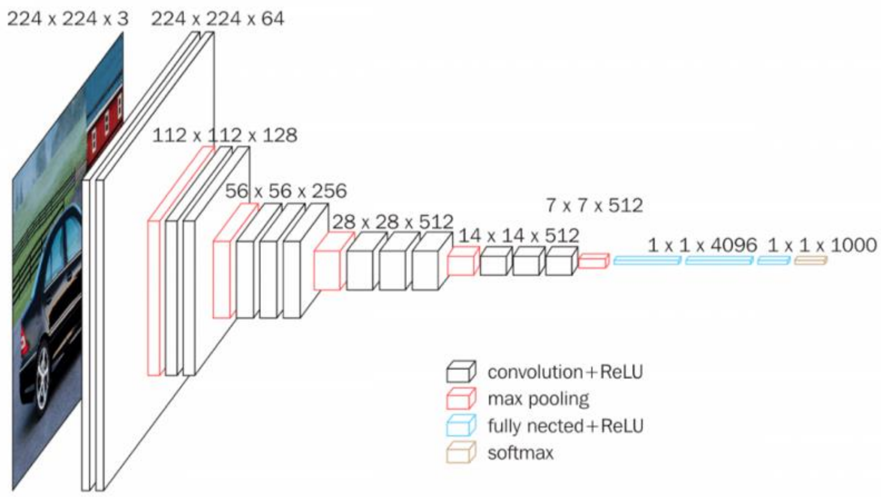
Reduction of Batch Size - It has been observed that reduction in batch size improves accuracy (Cycle 12).

Removal of duplicate images - It is observed that removal of duplicate images improves training by about 1%. While still an improvement, it is not very large and it can be attributed to the robustness of Neural Networks to noise.

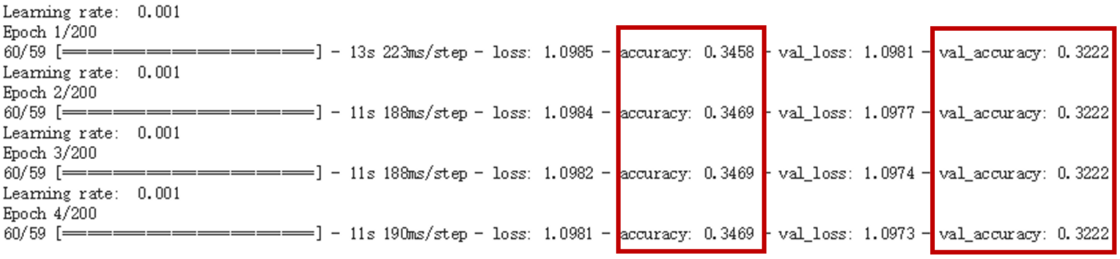
9. VGGNet

VGGNet was born out of the need to reduce the number of parameters in the CONV layers and improve on training time. The idea behind having fixed and small size kernels is that all the variable size convolutional kernels (11x11, 5x5, 3x3) can be replicated by making use of multiple 3x3 kernels as building blocks. The replication is in terms of the receptive field covered by the kernels [9]. For example, the effect of one 7x7 (11x11) conv layer can be achieved by implementing three (five) 3x3 conv layer with stride of one. This reduces the number of trainable variables by 44.9% (62.8%). Reduced number of trainable variables means faster learning and more robust to over-fitting.

For the food image classification task, we started out building a VGG16 model. VGG16 architecture consists of twelve convolutional layers, some of which are followed by maximum pooling layers and then four fully-connected layers and finally a 1000-way softmax classifier.



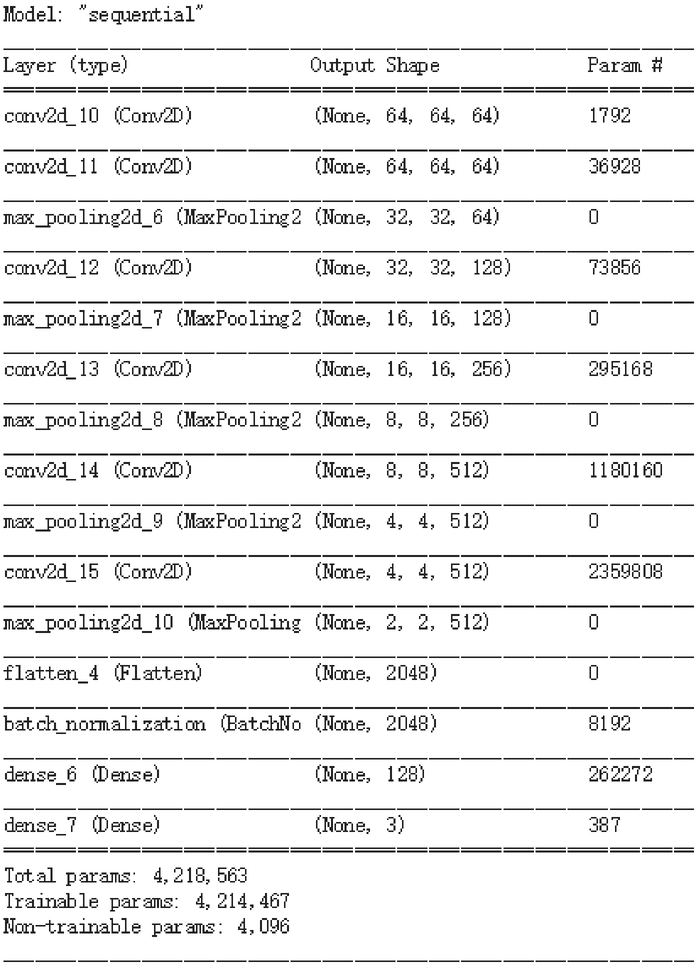
To adapt it our application, the last dense layer was changed to 3-way softmax classifier. We fit the dataset to the model and discovered that the model has failed to learn and the training accuracy was stuck at 34%.



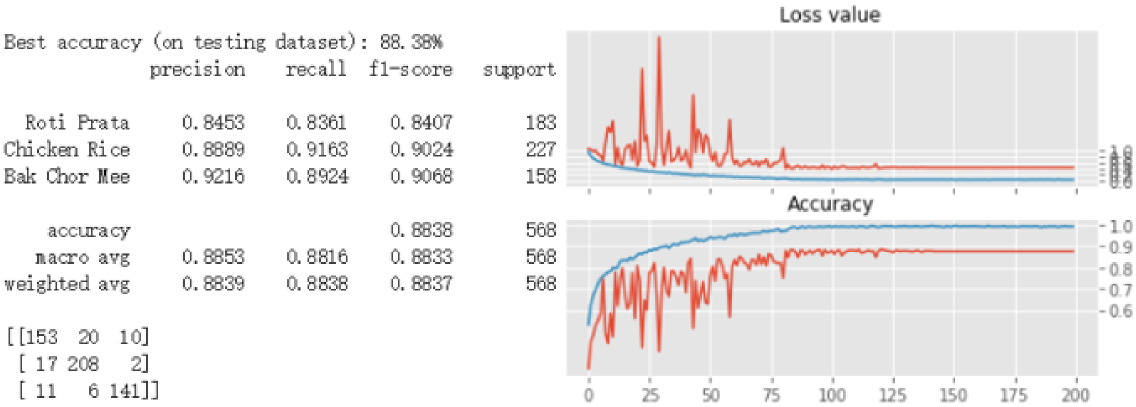
```
Learning rate:  0.001
Epoch 1/200
60/59 [==============================] - 13s 223ms/step - loss: 1.0985 - accuracy: 0.3458 - val_loss: 1.0981 - val_accuracy: 0.3222
Learning rate:  0.001
Epoch 2/200
60/59 [==============================] - 11s 188ms/step - loss: 1.0984 - accuracy: 0.3469 - val_loss: 1.0977 - val_accuracy: 0.3222
Learning rate:  0.001
Epoch 3/200
60/59 [==============================] - 11s 188ms/step - loss: 1.0982 - accuracy: 0.3469 - val_loss: 1.0974 - val_accuracy: 0.3222
Learning rate:  0.001
Epoch 4/200
60/59 [==============================] - 11s 190ms/step - loss: 1.0981 - accuracy: 0.3469 - val_loss: 1.0973 - val_accuracy: 0.3222
```

We realised that the 16 layer VGG was too deep for our small dataset without the skip connections in ResNet. We suspect that although ReLU activation function has been used to counter the vanishing gradient problem, there were simply too many layers and routes for the error signal to backpropagate and many of the weights tend to approach zero and this has caused the model not to be able to learn.

To continue experimenting with this model, we took out 7 x Conv2D layers and 2 x Dense layers and at the same time added Batch Normalisation and Learning Rate Scheduler. The model was now able to learn and has achieved an accuracy of 88.38%.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 64, 64, 64)        1792
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 64, 64, 64)        36928
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 32, 32, 128)       73856
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 16, 16, 256)       295168
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 8, 8, 512)         1180160
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 4, 4, 512)         0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 4, 4, 512)         2359808
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 2, 2, 512)         0
_________________________________________________________________
flatten_4 (Flatten)          (None, 2048)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 2048)              8192
_________________________________________________________________
dense_6 (Dense)              (None, 128)               262272
_________________________________________________________________
dense_7 (Dense)              (None, 3)                 387
=================================================================
Total params: 4,218,563
Trainable params: 4,214,467
Non-trainable params: 4,096
_________________________________________________________________
```



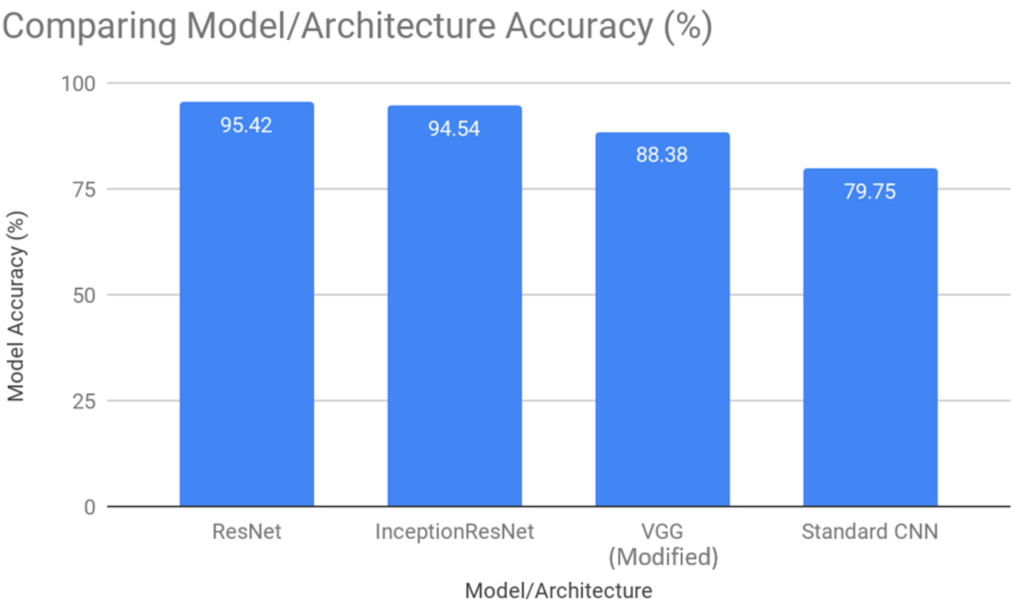
```
Best accuracy (on testing dataset): 88.38%
              precision    recall  f1-score   support

  Roti Prata     0.8453    0.8361    0.8407       183
Chicken Rice     0.8889    0.9163    0.9024       227
Bak Chor Mee     0.9216    0.8924    0.9068       158

    accuracy                         0.8838       568
   macro avg     0.8853    0.8816    0.8833       568
weighted avg     0.8839    0.8838    0.8837       568

[[153  20  10]
 [ 17 208   2]
 [ 11   6 141]]
```

After making these modifications to the VGG16, we realised that the model looks similar to the earlier CNN model (in section 6) we have built. However, this VGG (modified) model has higher validation accuracy than the CNN model because it was able to generalise better.

10. Comparison Between Models

Comparison between CNN, ResNet, VGG (Modified) and Inception/InceptionResNet Models

The below chart summarizes the accuracy scores achieved on the testing set. The ResNet model achieved the highest accuracy of 95.42%, which indicates that the model is production ready.



When comparing the size of the model (see table below), the InceptionResnet model is the largest while ResNet model is the smallest. This comparison shows us that should we need to deploy the model to mobile platforms, where space is an important consideration, the ResNet model will be highly considered.

Model/Architecture	File Size (MB)
InceptionResnet	134
VGG (Modified)	32.2
Standard CNN	28.6
ResNet	8.66

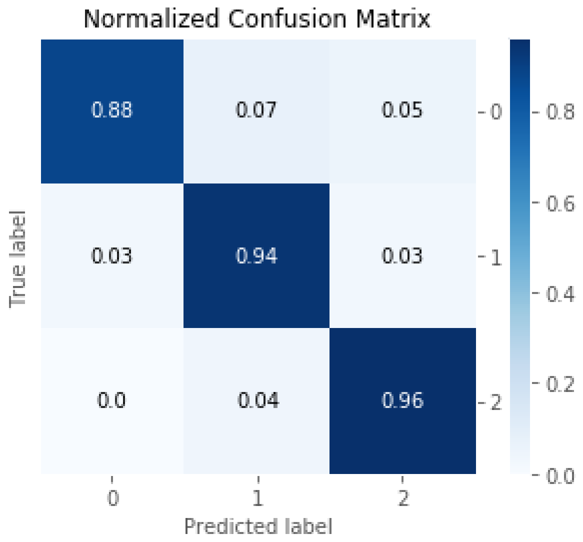
Comparison with Transfer Learning

A comparison was also made with the Transfer Learning approach, where pre-trained models are adapted and fine-tuned for similar tasks. In view of the project requirements, a pre-trained MobileNetV2 model has been adapted and fine-tuned. This particular architecture has been chosen due to it being known to an efficient network targeted for mobile platforms and features many similar elements like residual connections and the use of relu [8].

It has been observed to achieve an accuracy of 92.42% with only 20 epochs of training. While the ResNet and InceptionResnet networks trained from scratch outperformed this transfer learning model, there is a high potential in using transfer learning to build prototype models at lower computational cost (20 epochs in 1 run vs 200 epochs and many runs) and use it as a useful comparison.

The confusion matrix and f1-scores are shown below for the transfer learning model.

```
Best accuracy (on testing dataset): 92.42%
              precision    recall  f1-score   support

bak_chor_mee     0.9587    0.8782    0.9167       238
chicken_rice     0.8988    0.9407    0.9193       236
  roti_prata     0.9186    0.9575    0.9376       212

    accuracy                         0.9242       686
   macro avg     0.9254    0.9255    0.9245       686
weighted avg     0.9257    0.9242    0.9240       686

[[209  17  12]
 [  8 222   6]
 [  1   8 203]]
```



11. Findings and Conclusion

The team set out on a journey to train a deep learning computer vision model from scratch, starting from business case definition and data collection to building several image classification models. The team achieved the highest accuracy of 95.42% with ResNet.

The team demonstrated how different techniques can be used to finetune and build an effective deep model, from the use of data augmentation techniques, regularization (L2 and Dropout) and Batch Normalization to the use of different optimizers (adam and rectified adam) and tuning the batch size. Add, concatenate and lambda layers were also used to implement different neural architectures like ResNet and InceptionResnet.

The team also demonstrated how some deep learning tricks enabled a smoother or faster training process such as the use of 1-time pre-processing with pickles and the use of interactive visualizations like Tensorboard. Last but not least, a comparison with Transfer Learning - a technique that has been gaining popularity in the last few years - has also been done.

Reference

[1] Christian Szegedy and Wei Liu and Yangqing Jia and Pierre Sermanet and Scott Reed and Dragomir Anguelov and Dumitru Erhan and Vincent Vanhoucke and Andrew Rabinovich (2014), Going Deeper with Convolutions: https://arxiv.org/abs/1409.4842

[2] Shuhei Kishi (2018 June 10)), How to write Inception module: understanding and coding with Keras https://marubon-ds.blogspot.com/2018/06/how-to-write-inception-module.html

[3] Jason Brownlee (2019 July 5), How to Develop VGG, Inception and ResNet Modules from Scratch in Keras: https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/

[4] Christian Szegedy and Sergey Ioffe and Vincent Vanhoucke and Alex Alemi (2016), Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning: https://arxiv.org/pdf/1602.07261.pdf

[5] Keras Team on Github (2019 September 25), Inception-ResNet V2 model for Keras: https://github.com/keras-team/keras-applications/blob/master/keras_applications/inception_resnet_v2.py

[6] HG Zhao (CyberZHG) on Github (2019 September 25): https://github.com/CyberZHG/keras-radam

[7] Liyuan Liu and Haoming Jiang and Pengcheng He and Weizhu Chen and Xiaodong Liu and Jianfeng Gao and Jiawei Han (2019), On the Variance of the Adaptive Learning Rate and Beyond: https://arxiv.org/pdf/1908.03265.pdf

[8] Google AI Blog (2018 April 3),
MobileNetV2: The Next Generation of On-Device Computer Vision Networks: https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html

[9] Aqeel Anwar (2019 Jun 7), Difference between AlexNet, VGGNet, ResNet and Inception: https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

[10] Keras Issue #12683 on Github (2019 September 25), fit_generator with ImageDataGenerator is much slow than fit: https://github.com/keras-team/keras/issues/12683

[11] Keras Issue #12120 on Github (2019 September 25), Training takes too long when I am using ImageDataGenerator with a generator: https://github.com/keras-team/keras/issues/12120

[12] The Keras Blog (2019 September 25), Building powerful image classification models using very little data: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html