ngee ann polytechnic

School of InfoComm Technology

Deep Learning DLIR Assignment

Specialist Diploma in applied Generative AI (SDGAI)

Apr 2025 Semester

ASSIGNMENT

(60% of DLIR Module)

28th Apr 2025 – 11th June 2025

Submission Deadline:

Presentation: 9th OR 10th June 2025 (Mon/Tue)

Report: 11th June 2025 (Wednesday), 11:59PM

Student Name	**:**	**Lim Ai Sim Elizabeth**
Student ID	**:**	3440680F

Penalty for late submission:

10% of the marks will be deducted every calendar day after the deadline.

NO submission will be accepted after 21st June 2025 (Saturday), 11:59PM.

Contents

1. Introduction

In this assignment, the use of deep learning models for classification of images is explored. The data set provided consists of images grouped in 10 different food categories.

The model to be designed and implemented will be based on a convolutional neural network (CNN) architecture. CNN models are suitable for image classification as they are designed to convert images into vectors, enabling the machine to learn particular and unique features of given image for the classification task.

Classification of food images is a challenging task. The same food dish can also vary in arrangement, size and color depending on its preparation and presentation. In this task, the challenge is to design and train a model that can correctly classify an image of a food dish into the ten given classes.

2. Objective

To design and implement a convolutional neural network (CNN) based model that can accurately classify an appropriate food image as one of the 10 specified food categories.

3. Approach

i) A model based on Convolutional Neural Network (CNN) will be trained, validated, and tested using the provided image dataset.

ii) The dense layers of the custom-built CNN model will be integrated with the pretrained convolutional layers of a pretrain model. The performance of this transfer learning approach will be compared with the custom-built model in (i).

iii) The better model will be tested on at least three previously unseen food images to access its ability to generally classify the images.

4. Data Processing and augmentation

In this section, data preprocessing and initial model architecture will be discussed. The initial model containing convolutional 2D and dense layers, as well as the initial hyperparameters set used will also be explained.

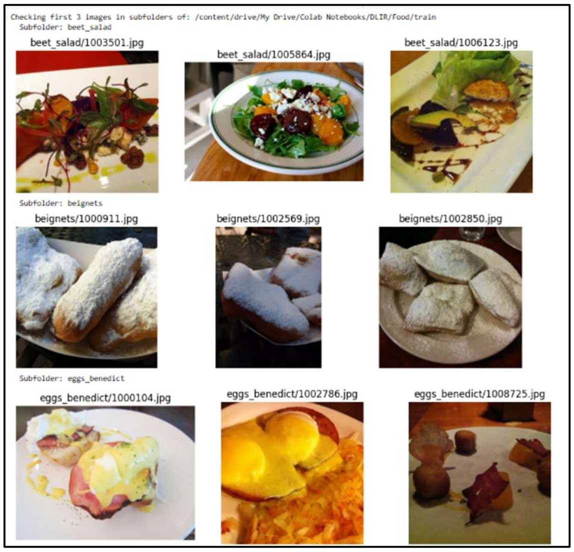
All codes and model training and evaluation are performed in Jupyter Notebooks on Google Colab or Amazon Web Services (AWS) SageMakerAI.

4.1. Data preprocessing

The data given for training, validation and testing consists of colored images of food in 10 different classes:

1. beet salad
2. beignets
3. eggs benedict
4. hamburger
5. hot and sour soup
6. huevos rancheros
7. lasagna
8. risotto
9. seaweed salad
10. strawberry shortcake

A total of 7500 images (750 per class) would be used for model training, 2000 images (200 per class) would be used as validation for model adjustments during training and optimization, and 500 images (50 per class) would be set aside as 'unseen data' for testing of model performance.



```
Checking first 3 images in subfolders of: /content/drive/My Drive/Colab Notebooks/DLIR/Food/train
  Subfolder: beet_salad
  Subfolder: beignets
  Subfolder: eggs_benedict
```

Figure 1: Some images included in the dataset. Each row represented a different 'class' of food. It is interesting to note how different a dish can look even though they belong to the same class.

The sizes of the images were adjusted to 150 x 150 pixels. Images are also normalized and rescaled by dividing the pixel values by 255, the result in which all values would be between 0 and 1.

4.2. Data augmentation

For data augmentation on the training images, images can be processed using Keras' ImageDataGenerator to randomly vary the training images and increase training data. This helped the model learn more effectively, mitigating overfitting and improving generalization.

The parameters are set based on the expected variation of images in the same food class.

i) Rotation range, shift range and horizontal flip would account for the same dish with different position in the image, either rotated at some angle, shifted off center (width/height shift) or completely mirrored (horizontal flip).

ii) Zoom range and shear range would be able to account for image distortions.

iii) Brightness range is required because photo images may sometimes be under- or over-exposures.

iv) Fill mode has been set to 'nearest' so that white areas (possibly created by the above image transformations) can be filled with the nearest pixel values.

The parameters used are detailed in the box below (Box1).

Code	Explanation

`train_datagen = ImageDataGenerator(`
`      rescale=1./255,`
`      rotation_range=40,`
`      width_shift_range=0.2,`
`      height_shift_range=0.2,`
`      shear_range=0.2,`
`      zoom_range=0.3,`
`      horizontal_flip=True,`
`      brightness_range=[0.8,1.2],`
`      fill_mode='nearest')`
	Rotation angle = 40 degrees left or right
	Shift range is in percentage.0.2 = 20% width or height position shift.
	Shear range is in radians.
	
	Zoom range is in percentage (0.3 = 30%)
	
	Brightness range is in a range 0.8 -1.2 means 20% darker to 20% brighter than original image.

Box 1: Parameters used for data augmentation

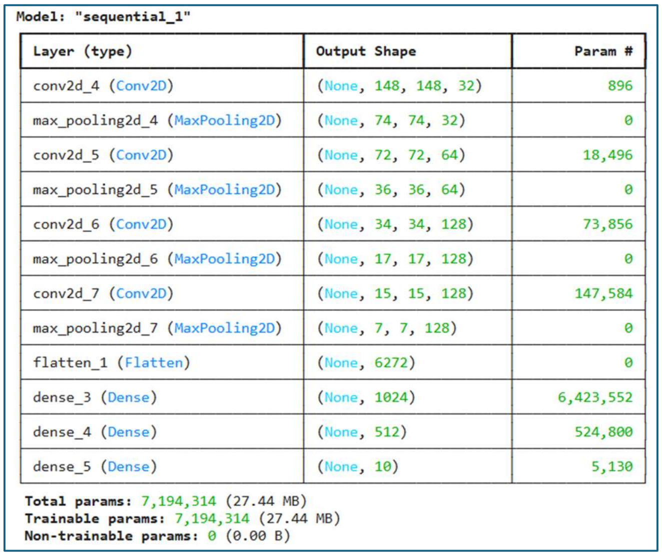
5. Model Design

5.1. Building the initial model

The initial model is a sequential model that takes in the image (150 by 150 pixels by 3 color channels).

There are four Convolutional 2D (Convo2D) layers followed by MaxPooling2D layer. Each Convo2D layer has 3x3 Kernel and filter sizes of 32, 64, 128, 128 respectively to capture the features. (Box 2) Stride and padding are left at default (1x1 for stride, and 'valid' respectively). The activation function for these layers is ReLU (Rectified Linear Unit).

The output from the Convo2D and MaxPooling2D layers are reshaped (or 'flattened') and then fed into the dense layers. The last dense layer outputs 10 values, representing the 10 classes. For prediction (likelihood of image belonging to one of the classes), softmax activation is used to convert the output values into probabilities.



Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_5 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_6 (Conv2D)	(None, 34, 34, 128)	73,856
max_pooling2d_6 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_7 (Conv2D)	(None, 15, 15, 128)	147,584
max_pooling2d_7 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_3 (Dense)	(None, 1024)	6,423,552
dense_4 (Dense)	(None, 512)	524,800
dense_5 (Dense)	(None, 10)	5,130

```
Total params: 7,194,314 (27.44 MB)
Trainable params: 7,194,314 (27.44 MB)
Non-trainable params: 0 (0.00 B)
```

Box 2: Initial model summary

5.2. Loss Function

Categorical cross entropy loss function is applied as the main task of the model is to perform multi-class prediction.

5.3. Optimizer

Root mean square propagation (RMSProp) is used for updating the weights via backpropagation during model training with learning rate of 1e-4 (0.0001).

Changing the optimizer method would require some understanding the underlying (https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be) It seem that the most commonly used optimizer are RMS or Adaptive Moment Estimation (Adam). Both optimizer functions are variations to stochastic gradient descent (SGD).

Gemini Flash 2.5 (Google. (2025, May 27)) was used to understand the difference between optimizing with RMSprop or Adam (See Annex B). In summary, RMSprop adjusts weights based on recent measurements on the magnitude of change towards correct prediction. Adam adjusts weights based on recent trends in addition to having RMSprop's adaptive properties.

5.4. Hyperparameters

i) Batch size: Batch size for training is set at 25. (A number divisible by the number of images in the training, validation and testing folders)

ii) Number of training cycles / epoch: the number of training cycles for the model to learn all the features of the training data.

5.5. Selection of metrics for evaluating model performance

For this assignment, loss and accuracy for both training and validation would be used to measure and evaluate model performance.

Ideally, model training results in a progressive increase in the accuracy score as the model learns while loss scores decrease with increased predictive confidence.

6. Custom-build CNN model training and optimization

6.1. Training progression of custom-build CNN model

Each training round yielded information for decision making on which part of the model or hyperparameter to adjust. Changes are made one component at a time to understand its effects on the performance of the model (See Table 1 for the summary of changes and observations made). See Annex F for the excel spreadsheet which records the changes made and the graphs of accuracy and loss scores.

Table 1: Customized model adjustments. This table shows the changes made to each model version and the results and observation on the effect of each change and alteration.

Model version	**Addition or change**	**Result / observation**
Training performed without data augmentation of training images.		
1-1	Initial model	Overfitting from epoch 5
1-2	Added dropout layer	Overfitting delayed to epoch 10
1-3	Changed first convolutional layer's kernel size to 5x5	Overfitting and stagnation at around epoch 10 Achieved 0.566 test accuracy score.
1-4	Added L2 regularization in the first dense layer	Similar to Model 1-3 Achieved 0.592 test accuracy score.
1-5	Changed the first convolutional layer's kernel size back to 3x3. Added another convolutional layers with 256 units and a MaxPooling layer.	Overfitting delayed to epoch 20. Achieved 0.60 test accuracy score.
1-6	Remove the convolutional layer with 256 layers along with its accompanying MaxPooling layer Added another convolutional layer with 32 units.	Training and validation loss graph remain in close synchrony through 30 epochs. However, test accuracy was found to have lowered.
1-7	Revert to Model 1-5 but changed **Dense layers with 1024 units to 512 units.**	Some overfitting. Achieved near 0.60 test accuracy score.
Training performed with data augmentation of training images. (Augments: `rotation_range = 40, width_shift_range = 0.2, height_shift_range = 0.2, shear_range = 0.2, zoom_range = 0.3, horizontal_flip = True, brightness_range = [0.8,1.2], fill_mode = 'nearest'`)		

Training epochs increased to 80.		
1-5A1	From Model 1-5.	Achieved 0.68 test accuracy score.
1-5A2	Added batch normalization before MaxPooling layers	Achieved 0.73 test accuracy score.
1-5A3	Changed all activation functions from ReLU to GeLU.	Accuracy and loss graphs seem to look smoother. Achieved 0.742 test accuracy score.
1-5A4	Changed the optimizer from RMSprop to Adam optimizer	Achieved 0.772 test accuracy score. Accuracy and loss graphs show some fluctuation.
1-5A5	Changed activation functions in the dense layers back to ReLU. Changed the optimizer back to RMSprop	Achieved 0.762 test accuracy score. Accuracy and loss graphs seem be more stable curve.
1-7A1	From Model 1-7.	Achieved 0.678 test accuracy score.
1-7A2	Batch normalization layers added after MaxPooling layers. Activation functions are all ReLU.	Achieved 0.728 test accuracy score. Slight overfitting occurring towards the end of the training epochs.
1-8A	From Model 1-5. Learning rate changed from 1e-4 to 2e-5. Optimizer was changed from RMSprop to Adam.	Achieved 0.578 test accuracy score. (Test accuracy score dropped.)

6.2. Rationale for the adjustments and changes made during training and evaluation

6.2.1. **Changing number of units in the Dense layers** – decreasing neural network by reducing the number of units in the dense layers is a common strategy used to mitigate overfitting. However, the number of units cannot be overly reduced, or the model would not be able to pick up all the features for the classification task.

6.2.2. **Adding a dropout layer and L2 regularization** – Initial training resulted in overfitting at around epoch 10. A Dropout layer and an L2 regularization were subsequently added to the model to reduce the complexity of the model and to mitigate overfitting.

The dropout layer functions to reduce model complexity by setting a random portion of node activations to zero during each training step (Srivastava et al (2014)) This would push the dense layers of the network to learn without relying too much on a particular feature and reduce overfitting.

L2 regularization keeps weights in the model small by penalizing the model based on the square of the magnitude of all weights in a layer. (Ref: https://builtin.com/data-science/l2-regularization) In brief, it reduces the influence of large weights from particular features and helps to promote better generalization.

6.2.3. **Adjusting the kernel size in the convo2D layer** – In order to improve training accuracy, the kernel size was increased from 3x3 to 5x5. Larger kernel sizes could capture more features but may miss out on the local fine details and increase computational complexity (Szegedy et al (2016)). However, when no improvement to the training/validation accuracy was observed and no delay in overfitting occurred, the kernel size was reverted to 3x3.

6.2.4. **Adding convo2D and MaxPooling2D blocks** – Adding convo2D+MaxPooling2D layers can be an effective strategy in improving the model performance [https://youtu.be/V9ZYDCnItr0?si=pVBCRVVTBdi7RGOZ]. The following adjustments were made and evaluated.

i) An extra convo2D+MaxPooling2D layer with a 32-output channels added to the start of the convolutional layers did not improve model learning and accuracy.

ii) Instead, adding a final convo2D+MaxPooling2D layer with 256 output channels seem to improve model accuracy.

6.2.5. **Adding batch normalization layers** – it was observed that many pretrained models include batch normalization within their convolutional layers. A query was performed to understand the purpose of batch normalization. (See Annex

C) In summary, batch normalization is used to ensure the inputs to the next layer are better distributed. This would improve learning rates and training performance. (https://www.geeksforgeeks.org/what-is-batch-normalization-in-deep-learning/)

6.2.6. **Changing the activation function from ReLU to GeLU** (Gaussian Error Linear Unit) – GeLU was implemented in attempt to improve the training capacity of the model, as it was noticed that training seemed to stall when training accuracy reached ~0.5-0.6.

6.2.7. **Adjusting learning rate** – in some models, learning rate of 2e-5 (0.00002) was used instead of the default 1e-4 (0.0001). It was observed that models trained using learning rate 2e-5 (0.00002) could not achieve validation accuracy between 0.5 to 0.6. Thus, the initial learning rate of 1e-4 was retained.

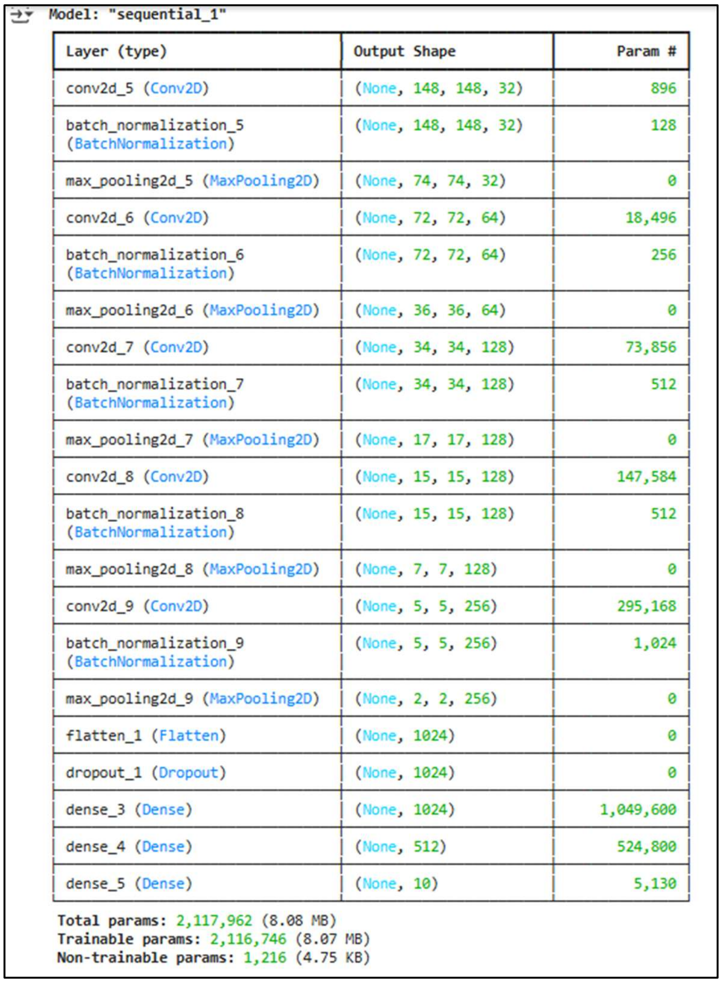
- Too High (e.g., 1e-2 or 0.01): The model might take huge steps when learning, which may result in overshooting the minimum loss, and fail to converge (loss might fluctuate wildly or even increase).
- Too Low (e.g., 1e-6 or 0.000001): The model may learn in extremely tiny steps. It will converge very slowly, potentially getting stuck in a local minimum or simply taking an impractically long time to train.

Reference: https://milvus.io/ai-quick-reference/how-do-learning-rates-affect-deep-learning-models

6.3. The final 'build-from-scratch' CNN model:

Model 1-5A5

This model consists of 5 convolutional 2D + Batch Normalization + MaxPooling 2D layers (32, 64, 128, 128, 256 output channels respectively) with GeLu activation function, the output is put through a flatten and dropout layer, followed by 2 fully connected Dense layers with ReLu activation functions. The final Dense layer has 10 output channels, and 'softmax' activation function for performing the class prediction. (See Annex D for the codes for build setup of Model 1-5A5)



```
Model: "sequential_1"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ conv2d_5 (Conv2D)               │ (None, 148, 148, 32)   │           896 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_5           │ (None, 148, 148, 32)   │           128 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_5 (MaxPooling2D)  │ (None, 74, 74, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_6 (Conv2D)               │ (None, 72, 72, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_6           │ (None, 72, 72, 64)     │           256 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_6 (MaxPooling2D)  │ (None, 36, 36, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_7 (Conv2D)               │ (None, 34, 34, 128)    │        73,856 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_7           │ (None, 34, 34, 128)    │           512 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_7 (MaxPooling2D)  │ (None, 17, 17, 128)    │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_8 (Conv2D)               │ (None, 15, 15, 128)    │       147,584 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_8           │ (None, 15, 15, 128)    │           512 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_8 (MaxPooling2D)  │ (None, 7, 7, 128)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_9 (Conv2D)               │ (None, 5, 5, 256)      │       295,168 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_9           │ (None, 5, 5, 256)      │         1,024 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_9 (MaxPooling2D)  │ (None, 2, 2, 256)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_1 (Flatten)             │ (None, 1024)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)             │ (None, 1024)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 1024)           │     1,049,600 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                 │ (None, 512)            │       524,800 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_5 (Dense)                 │ (None, 10)             │         5,130 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 2,117,962 (8.08 MB)
 Trainable params: 2,116,746 (8.07 MB)
 Non-trainable params: 1,216 (4.75 KB)
```

Figure 2: Model 1-5A5 model summary

7. Transfer Learning: Leveraging Pre-trained Models

7.1. Considerations for the selection of a pretrained model

A quick search on Google and Google scholar seem to show that Resnet50 and InceptionV3 have been recently used in recent publications that worked on food image classification. These two models are available in the Keras models library (https://keras.io/api/applications/)

7.1.1. **Resnet-50** (He, 2016) is part of Resnet family of models based on residual network architecture developed by Microsoft Research experts (Figure 3). This architecture was designed to solve the problem of vanishing or exploding gradients that occur when training very deep networks. Resnet contains 'skip connections' (also known as residual connections) within its convolutional blocks, the weight updates do not get degraded as it is backpropagated through the network to its early layers. The skip connections updates features and weights (or the learning signal) by 'skipping over' the layers that are not involved in learning.(Figure 4)

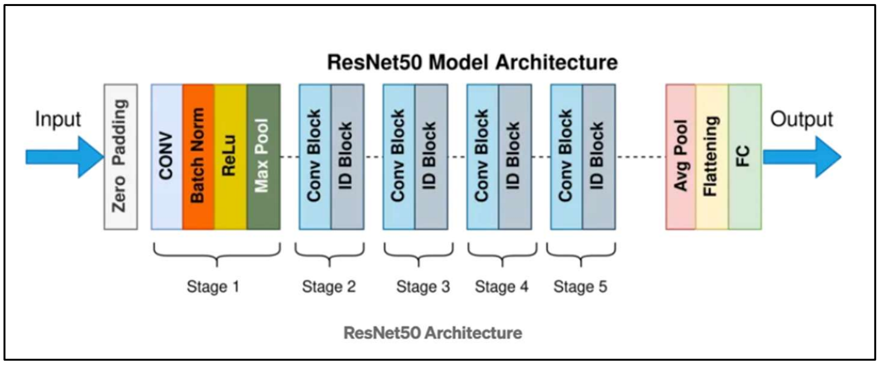


Figure 3: Resnet-50 Architecture Source: https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f

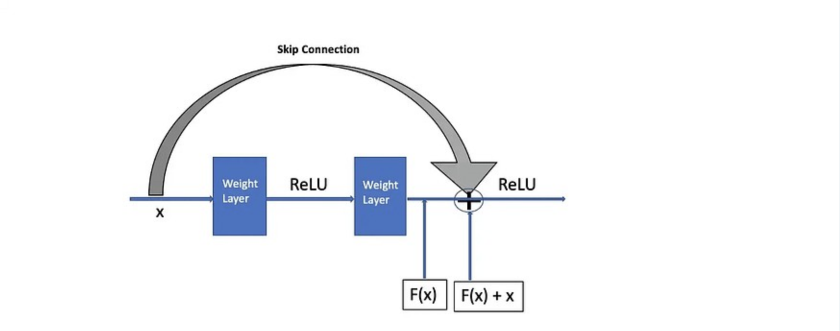


Figure 4: Resnet skipped connections. Source: https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f

7.1.2. **InceptionV3** (Szegedy, 2015) is part of the Inception family of convolutional neural networks that was developed to solve the problem of computation efficiency of very deep neural networks. Conventionally, layers are added to a neural network for image classification to capture more complex features and details, but this could lead to increased number of parameters and high computational costs. Inception network architecture applies multiple filters of different sizes, placed in parallel (each unit of this is referred to as ‘inception’ block or module. See Figure 5 and Figure 6.)

In addition, Inception networks implement ‘bottleneck layers’ (In Figure 5, labelled as ‘concat’.) which consist of 1x1 filters to reduce the number of output channels before they are fed to larger (and therefore, more computationally expensive) filters. This contrasts with earlier network models which tend to connect filters and convolutional blocks one after another (e.g. Resnet and VGG). This allows the network to learn complex and more intricate features of the image while moderating computational costs.

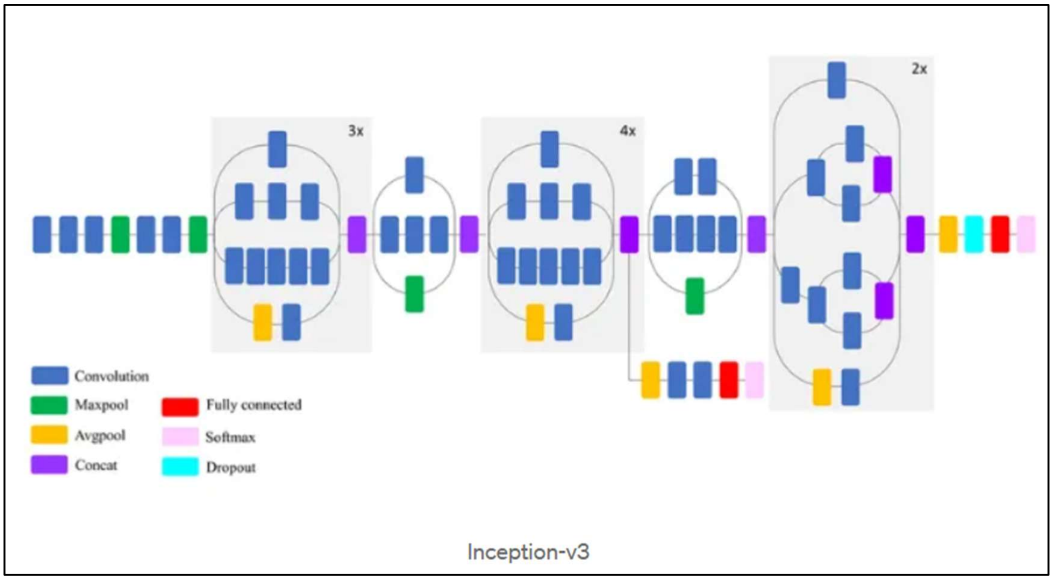


Figure 5: General architecture of InceptionV3
Source:https://medium.com/@akshayhitendrashah/understanding-inception-v3-d43fe7ca66a9

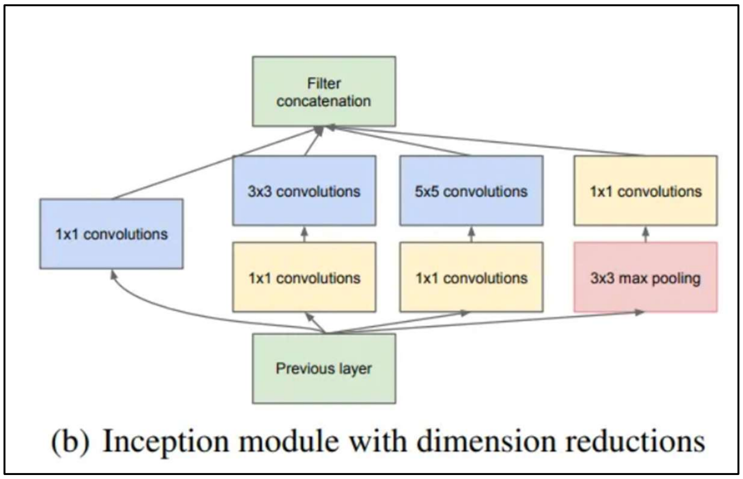


Figure 6: InceptionV3 contains convolutional kernels in parallel followed by pooling to extract the most prominent features of the given images. Source: https://arxiv.org/pdf/1409.4842v1

InceptionV3 was selected as it has been used in recent food classification model development (Burkapalli *et al*, 2020; Mawardi *et al*, 2024). It would be interesting to explore the effects of unfreezing its convolutional layers at different depths on overall model training and performance.

7.2. Training the InceptionV3 convolution layers with customized Dense Layers.

The convolutional layers of the InceptionV3 model were imported from the Keras Applications library. The imported convolutional base of InceptionV3 was integrated with the dense layers taken from Model 1-5A5.

Data augmentation was applied to the training data as described previously. (Section 4.2: Data augmentation)

The initial training phase focused on training the custom dense layers, while the weights of the pre-trained InceptionV3 convolutional base were kept frozen.

Changes and adjustments to the model made are listed in Table 2. See Annex G for the excel spreadsheet which records the changes made and the graphs of accuracy and loss scores of training and validation.

Table 2: Adjustments made to the InceptionV3+customized Dense layers. This table shows the changes made to each version and the results and observation on the effect of each change and alteration.

Model version	**Addition or change**	**Result / observation**

Training initially performed with frozen convolutional base. 50 training epochs.		
IN-1	Initial model	Training accuracy graph shows stagnation of accuracy at ~0.75 Achieved 0.768 test accuracy score.
IN-2	Reduced the complexity of the dense layers. First Dense layer with 1024 units reduced to 512 units, second dense layer with 512 units reduced to 256 units.	Achieved 0.769 test accuracy score. No improvement to accuracy and loss graphs. Training accuracy does not go above ~ 0.75.
Unfreezing of select layers in the convolutional base.		
IN-3	Similar to IN-1, but unfrozen from Layer ‘**mix 5**‘ (no. 164).	Training accuracy scores increase and taper off at about 0.95. Achieved 0.864 test accuracy score. Validation loss graph contains periodical spikes.
IN-4	Similar to IN-1, but unfrozen from Layer ‘**mix 9**‘ (no. 279).	Achieved 0.811 test accuracy score.
IN-5	Similar to IN-1, but unfrozen from Layer ‘**mix 4**‘ (no. 132).	Achieved 0.878 test accuracy score. Validation graph seems very unstable and trend suggest slight overfitting.
IN-6	Similar to IN-5 Optimizer was changed from RMSprop to Adam.	Achieved 0.8768 test accuracy score. Training and validation loss graphs do not show any spiking.

7.3. Finetuning - Unfreezing the modules of the convolutional base in InceptionV3

Unfreezing layers in convolutional base is commonly performed in transfer learning finetuning.

The convolutional base of InceptionV3 has already been trained on ImageNet. To adapt the model to our use-case, layers of the convolutional base may be unfrozen to enable it to adapt and learn the complex and intricate patterns that are more specific to our dataset.

There are 11 inception blocks or 'mix', numbered 0 to 10 (Figure 7). Each progressive module in the model is learning more and more complex features (Burkapalli *et al* (2020)), from edges to shapes to higher features.

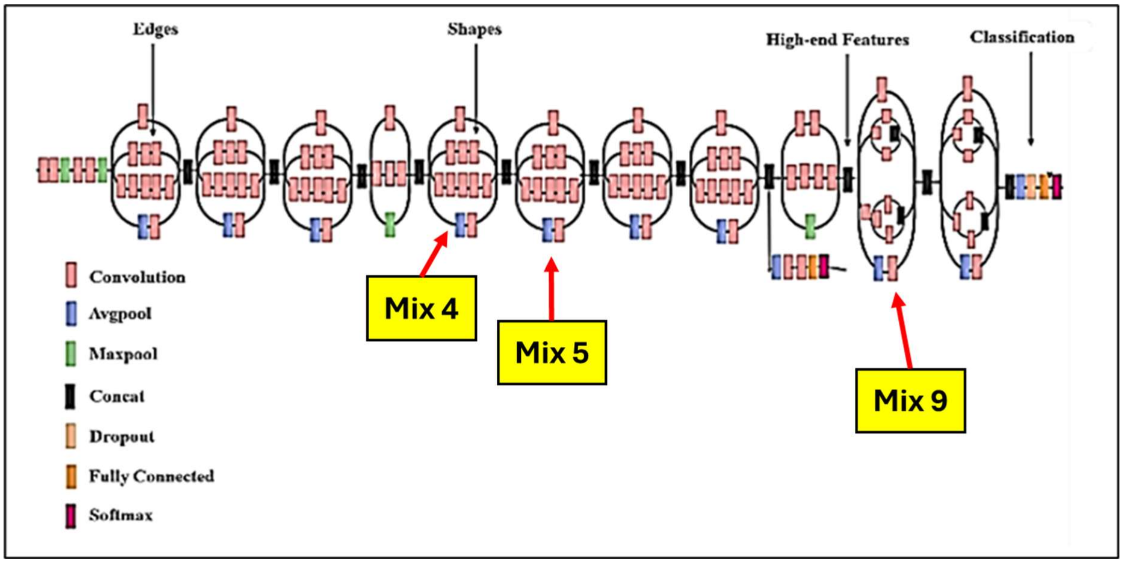


Figure 7: This diagram shows the structure of the InceptionV3 convolutional base with emphasis on the inception blocks/modules. Source: Burkapalli *et al* (2020)

i) **Mix 5** – When convolutional base was unfrozen from Mix 5 which was selected as it was the 'middle' layer (Table 2, Model versions IN3). It was observed that validation loss and test accuracy scores improved significantly (~70% accuracy to ~80%+ accuracy).

ii) **Mix 9** – The convolutional base was unfrozen from Mix 9 (Table 2, Model versions IN4) which represented the last few convolutional layers. It was observed that accuracy levels were lower.

iii) **Mix 4** – Lastly, the base was unfrozen from Mix 4 (Table 2, Model versions IN5), where it was found that the accuracy was slightly higher than unfreezing from Mix 5.

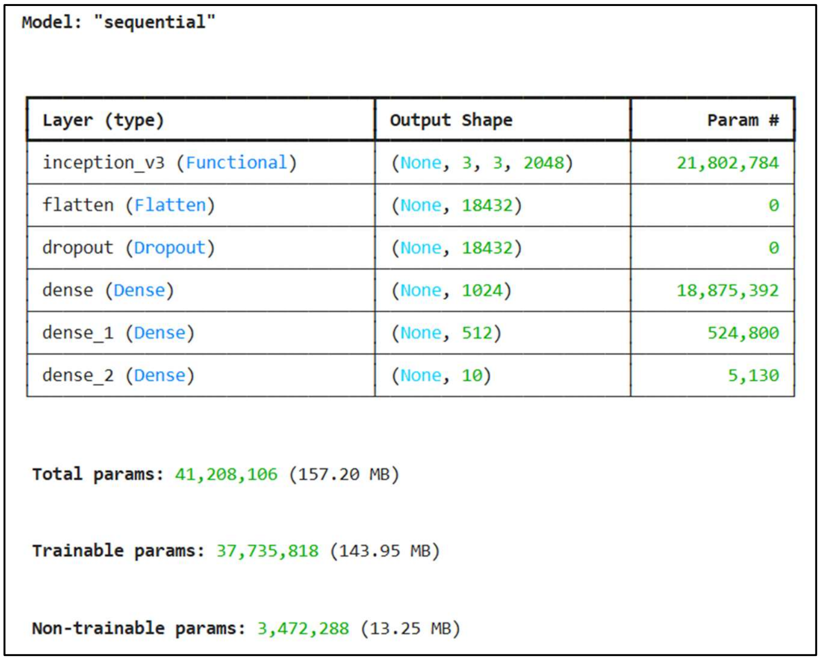
7.4. Changing the Optimizer

Finally, the last change was using Adam optimizer for the final training round (Table 2, Model versions IN6), which resulted in further improvement to accuracy and loss scores.

7.5. Final InceptionV3 convolutional layers + classification dense layers

Model: Food IN-6

This model uses convolutional base from InceptionV3 on top of Dense layers from Model 1-5A5 (flatten, dropout (0.5), Three Dense layers with outputs of 1024 units, 512 units and with a final 10 channel outputs with a SoftMax function. (See Annex E for the codes for final model build of Food IN-6)



```
Model: "sequential"

┌──────────────────────────────┬───────────────────────┬─────────────┐
│ Layer (type)                 │ Output Shape          │     Param # │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ inception_v3 (Functional)    │ (None, 3, 3, 2048)    │  21,802,784 │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ flatten (Flatten)            │ (None, 18432)         │           0 │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ dropout (Dropout)            │ (None, 18432)         │           0 │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ dense (Dense)                │ (None, 1024)          │  18,875,392 │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ dense_1 (Dense)              │ (None, 512)           │     524,800 │
├──────────────────────────────┼───────────────────────┼─────────────┤
│ dense_2 (Dense)              │ (None, 10)            │       5,130 │
└──────────────────────────────┴───────────────────────┴─────────────┘

 Total params: 41,208,106 (157.20 MB)

 Trainable params: 37,735,818 (143.95 MB)

 Non-trainable params: 3,472,288 (13.25 MB)
```

Figure 8: Model Food IN6

8. Evaluation of Models and Discussion

8.1. Final test evaluation scores

From the final test evaluation accuracy and loss scores, 'Food_IN6' is the better model, with 87% test accuracy compared to the build from scratch CNN model Model 1-5A5 (Table 3).

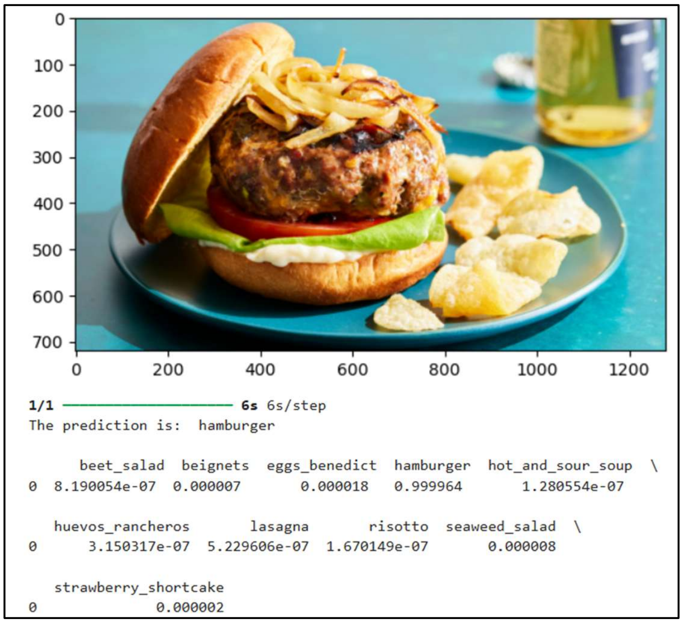
Table 3: Test accuracy results of the final models 1-5A5 and Food_IN6

Model	Test accuracy	Test Loss score
1-5A5 [Built-from-scratch CNN model]	0.7659	0.8871
Food_IN6 [InceptionV3 convolutional base + customized Dense layers]	0.8750	0.5881

8.2. Testing on 3 images taken from the internet

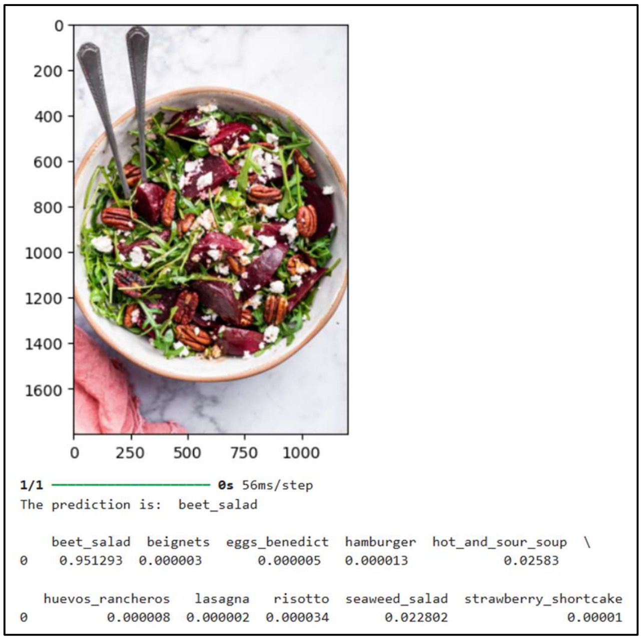
The better model Food_IN6.keras was able to correctly classify all 4 images taken from the internet.

8.2.1. Hamburger (Prediction: Hamburger, 0.999)



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 6s 6s/step
The prediction is:  hamburger

     beet_salad  beignets  eggs_benedict  hamburger  hot_and_sour_soup  \
0  8.190054e-07  0.000007       0.000018   0.999964       1.280554e-07

   huevos_rancheros       lasagna       risotto  seaweed_salad  \
0      3.150317e-07  5.229606e-07  1.670149e-07       0.000008

   strawberry_shortcake
0              0.000002
```

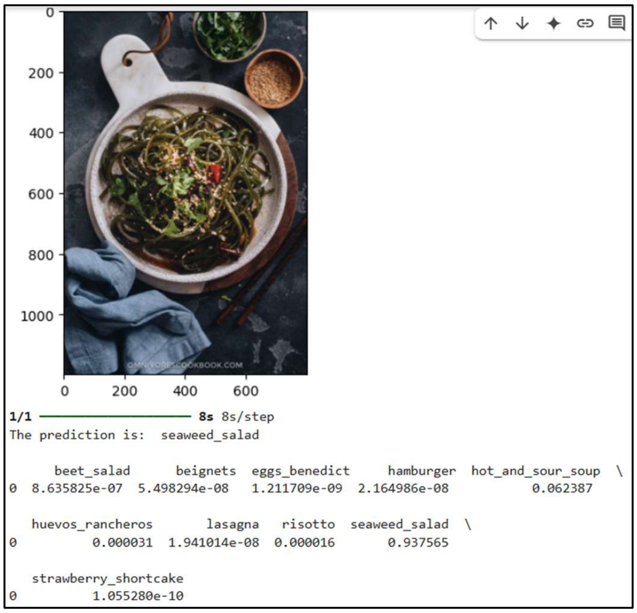
8.2.2. Beet salad (Prediction: Beet salad, 0.951)



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step
The prediction is:  beet_salad

   beet_salad  beignets  eggs_benedict  hamburger  hot_and_sour_soup  \
0    0.951293  0.000003       0.000005   0.000013            0.02583

   huevos_rancheros   lasagna   risotto  seaweed_salad  strawberry_shortcake
0          0.000008  0.000002  0.000034       0.022802               0.00001
```

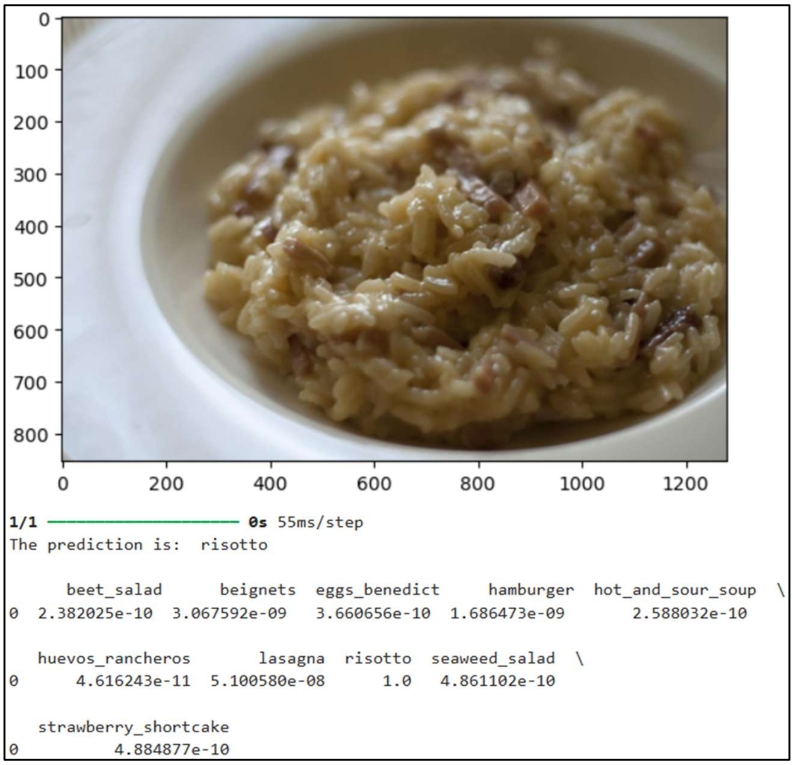
8.2.3. Seaweed Salad (Prediction: Seaweed salad, 0.937)

Seaweed salad looked very similar to beet salad. So, it was good that the model got it correct.



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 8s 8s/step
The prediction is:  seaweed_salad

     beet_salad      beignets  eggs_benedict     hamburger  hot_and_sour_soup  \
0  8.635825e-07  5.498294e-08   1.211709e-09  2.164986e-08           0.062387

   huevos_rancheros       lasagna   risotto  seaweed_salad  \
0          0.000031  1.941014e-08  0.000016       0.937565

   strawberry_shortcake
0          1.055280e-10
```

8.2.4. Risotto (Prediction: Risotto, 1.0)

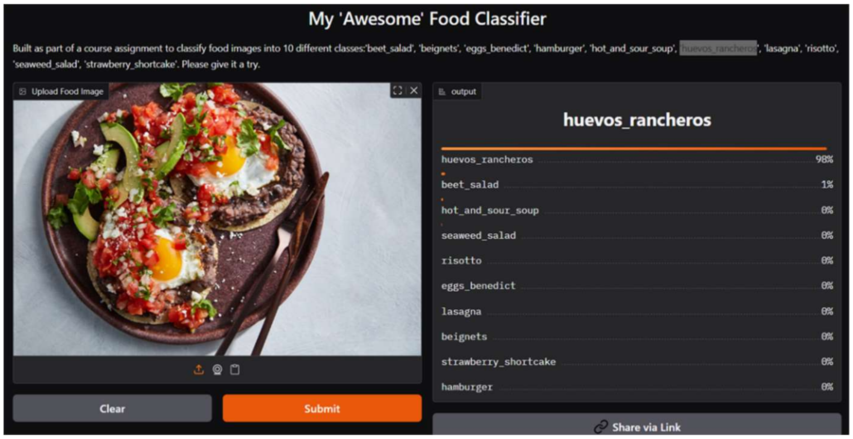


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
The prediction is:  risotto

     beet_salad      beignets  eggs_benedict     hamburger  hot_and_sour_soup  \
0  2.382025e-10  3.067592e-09   3.660656e-10  1.686473e-09       2.588032e-10

   huevos_rancheros       lasagna  risotto  seaweed_salad  \
0      4.616243e-11  5.100580e-08      1.0   4.861102e-10

   strawberry_shortcake
0          4.884877e-10
```

8.3. Model Demo

The model is available for testing on Hugging Face’s Spaces: https://huggingface.co/spaces/Ravenblack7575/DLIRfood .

The test application was created using Gradio and hosted on Hugging Face “Spaces” (https://huggingface.co/spaces).


My 'Awesome' Food Classifier

Built as part of a course assignment to classify food images into 10 different classes:'beet_salad', 'beignets', 'eggs_benedict', 'hamburger', 'hot_and_sour_soup', 'huevos_rancheros', 'lasagna', 'risotto', 'seaweed_salad', 'strawberry_shortcake'. Please give it a try.

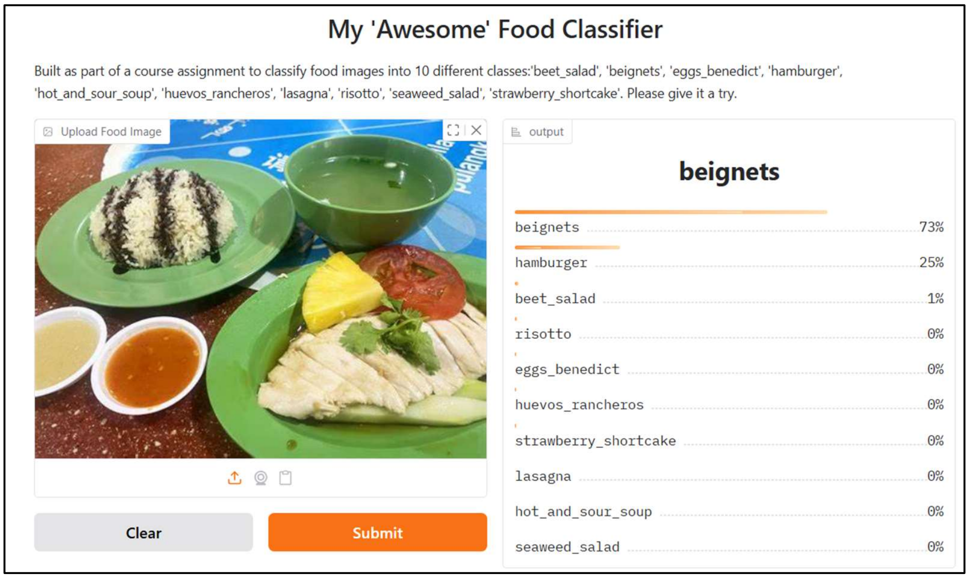
huevos_rancheros

```
huevos_rancheros ........ 98%
beet_salad .............. 1%
hot_and_sour_soup ....... 0%
seaweed_salad ........... 0%
risotto ................. 0%
eggs_benedict ........... 0%
lasagna ................. 0%
beignets ................ 0%
strawberry_shortcake .... 0%
hamburger ............... 0%
```

Figure 9: The assignment food image classifier on Gradio.

8.4. <u>Extra: What happens when you submit a food dish that is not part of the ten classes?</u>

The model would still report its prediction. However, it can be seen that the percentage for that prediction call would not be 100%, rather it could be at a lower percentage or the percentages are split among multiple classes. (Figure 10)



My 'Awesome' Food Classifier

Built as part of a course assignment to classify food images into 10 different classes:'beet_salad', 'beignets', 'eggs_benedict', 'hamburger', 'hot_and_sour_soup', 'huevos_rancheros', 'lasagna', 'risotto', 'seaweed_salad', 'strawberry_shortcake'. Please give it a try.

beignets

```
beignets ............................ 73%
hamburger ........................... 25%
beet_salad ........................... 1%
risotto .............................. 0%
eggs_benedict ........................ 0%
huevos_rancheros ..................... 0%
strawberry_shortcake ................. 0%
lasagna .............................. 0%
hot_and_sour_soup .................... 0%
seaweed_salad ........................ 0%
```

Figure 10: The results when an image of Chicken Rice meal set is submitted. The model calls 'beignets" but the prediction score is 73% beignets and 25% hamburger.

9. Limitations and recommendations

i) Currently, if the model was given an image that doesn't fall into any one of these categories, it would still give a prediction based on these ten classes. The model could include another class for images that it can't classify, or a prediction threshold could be implemented that allows the model to indicate when it cannot confidently classify a given image.

ii) Functions could be written so that the codes used for model training could be neater and more organized, which may translate to easier execution.

iii) In future model training, data augmentation will be performed before training commences.

10. Conclusion

In this assignment, two different models (one built from scratch and another using a pretrained model) were trained and evaluated. The training of the models involved changing, adding or removing model layers and components, and closely monitoring its effects of model learning through accuracy and loss scores.

The application of transfer learning demonstrated that the use of pretrained models could shorten the process of putting together a model for one's own image classification application. Understanding model architecture and rationale for its structure is key to adjusting and training it for a given task.

The model using the pretrained model convolution base, Model Food_IN6 proved to be more accurate than the built from scratch model.

11. References

1. Deep Learning Architectures Explained: ResNet, InceptionV3, SqueezeNet https://www.digitalocean.com/community/tutorials/popular-deep-learning-architectures-resnet-inceptionv3-squeezenet

2. ImageNet: VGGNet, ResNet, Inception, and Xception with Keras https://pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/

3. Kaggle Guide to VGG, Resnet and InceptionV3 https://www.kaggle.com/code/darthmanav/resnet-50-vgg-16-inception-v3-beginner-s-guide

4. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778). [Resnet models]

5. Burkapalli, V. C., & Patil, P. C. (2020). Transfer learning: Inception-V3 based custom classification approach for food images. ICTACT Journal on Image & Video Processing, 11(1). [InceptionV3 for food image classification]

6. B. Senapati, J. R. Talburt, A. Bin Naeem and V. J. R. Batthula, "Transfer Learning Based Models for Food Detection Using ResNet-50," 2023 IEEE International Conference on Electro Information Technology (eIT), Romeoville, IL, USA, 2023, pp. 224-229, doi: 10.1109/eIT57321.2023.10187288. [Resnet50 for food image classification]

7. Mawardi, C., Buono, A., & Priandana, K. (2024). Performance Analysis of ResNet50 and Inception-V3 Image Classification for Defect Detection in 3D Food Printing. International Journal on Advanced Science, Engineering & Information Technology, 14(2).

8. Pouya Bohlol, Soleiman Hosseinpour, Mahmoud Soltani Firouz, Improved food recognition using a refined ResNet50 architecture with improved fully connected layers, Current Research in Food Science, Volume 10,2025, 101005, ISSN 2665-9271, https://doi.org/10.1016/j.crfs.2025.101005. [Resnet50 for food image classification]

9. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

10. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9). [Inceptionv3 models]

11. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).

Annex A: Metrics used to evaluate machine learning models

Table 4: Metrics used to evaluate machine learning models for classification (Reference: https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide)

Metric	**Definition**
Accuracy	The number of correctly classified images out of the total number of images.
Confusion Matrix	A table showing the number of true positive, false positive, true negative and false negative scores, which may give a detailed view of what the model tends to classify correctly or incorrectly.
Precision	May be used to measure the number of true positive out of all correctly classified images out of all classified images.
Recall	May be used to measure the number of correctly classified images out of correctly and incorrectly classified classes.
Area under the curve (AUC)	May be used to measure the rate of correctly classified images over incorrectly classified images.

Annex B: Gemini Flash 2.5 explaining the difference between RMS and Adam optimisers

Prompt: Explain the difference between RMSprop and Adam optimizers in simple terms without the use of complex math.

Understanding the Challenge: Steering a Ship in a Fog

Imagine you're trying to steer a ship (your neural network) through a vast, foggy ocean to find the deepest point (the best solution).

- **The "Gradient" (Slope):** This is your compass. It tells you which way is downhill (towards better solutions).
- **"Learning Rate":** This is how much you turn the rudder. A small turn means slow, careful adjustments. A big turn means drastic, fast adjustments.
- **"Vanishing/Exploding Gradients":**
    - **Vanishing Gradients:** Sometimes, your compass suddenly points very, very weakly, almost not at all. It's like your ship is in a calm, flat area, and you don't know which way to go. Your adjustments become tiny, and you stop making progress.
    - **Exploding Gradients:** Other times, your compass spins wildly and strongly. It's like you hit a very steep wave, and the compass tells you to turn the rudder extremely hard. Your ship might overshoot the target, spin out of control, or even capsize.
- **"Mini-batch Learning":** Instead of steering based on looking at the *entire ocean* at once, or just *one tiny ripple*, you look at a *small, random patch of waves* around you to decide your next turn. This gives you a good enough idea of the general direction without being too slow or too jumpy.

RMSprop: The Smart Rudder Controller

RMSprop is like a **smart rudder controller** designed by Geoffrey Hinton (a pioneer in AI) to deal with those vanishing and exploding compass issues.

- **What it does:** Instead of always turning the rudder by the same amount (fixed learning rate), RMSprop keeps a **running memory** of how strongly the compass has been pointing recently.
    - If the compass has been pointing **very strongly** (large gradients, potential exploding), RMSprop says, "Whoa there, let's only turn the rudder a little bit, even if the compass is screaming at us." This **reduces the step size** to prevent overshooting.
    - If the compass has been pointing **very weakly** (small gradients, potential vanishing), RMSprop says, "Hmm, the compass isn't giving us much signal, let's try a *slightly bigger* turn than we normally would." This **increases the step size** to make progress even when the signal is weak.
- **Simply put:** RMSprop automatically adjusts how much you turn the rudder for *each part of the ship's steering system* based on its recent behavior. It's

an **adaptive learning rate** optimizer – the "learning rate" (how much you adjust) isn't fixed; it changes on the fly.

Adam: The Even Smarter Navigator with Momentum

Adam (Adaptive Moment Estimation) is like an **even more advanced navigation system** that builds on RMSprop's ideas. It's considered one of the best and most popular systems today.

- **What it adds:** Adam doesn't just remember how *strong* the compass has been (like RMSprop). It also remembers the *average direction* the compass has been pointing in.
    1. **"First Moment" (Average Direction/Momentum):** Imagine Adam remembers the *general trend* of your compass readings. If you've been consistently turning slightly left, Adam gives you a little extra push in that "left" direction, even if the current reading is slightly different. This is like **momentum** – it helps you keep moving steadily in the right direction and slide past small bumps.
    2. **"Second Moment" (Average Strength/Adaptive Step Size):** This is similar to what RMSprop does. Adam also remembers how *strongly* the compass has been pointing recently and uses that to adjust how much you turn the rudder for each part.
- **How it works together:** Adam combines these two memories. It uses the "average direction" to know *where* to go, and it uses the "average strength" to know *how much* to turn the rudder for that specific part of the ship's steering.
- **Simply put:** Adam is a super-smart navigator. It not only knows to adjust your rudder turns based on how erratic the signals have been (like RMSprop), but it also incorporates a "memory" of your past movements to keep you on a smoother, more efficient path towards your destination. It's adaptive and efficient, making it excellent for finding that deepest point in the foggy ocean.

Annex C: Gemini Flash 2.5 explains the reasoning behind adding Batch Normalization before MaxPooling2D block

Prompt / Question: Is it correct to assume that batch normalization should be generally be plaed before the MaxPooling2D layer?

Yes, **Batch Normalization (BN) is generally considered okay, and often beneficial, when placed *before* a MaxPooling2D layer.**

Here's a breakdown of why this is the common practice and why it makes sense:

1. **Standard Order:** The most common and empirically proven sequence in modern CNN architectures (like ResNet, Inception, MobileNet, EfficientNet) is: **Conv2D -> BatchNormalization -> Activation -> MaxPooling2D**
2. **What Batch Normalization Does:** BN normalizes the activations (the output of the previous layer) to have a zero mean and unit variance. This stabilizes the distribution of inputs to the next layer and allows for higher learning rates, leading to faster and more stable training.
3. **What Max Pooling Does:** Max pooling selects the maximum value within a given receptive field. It reduces the spatial dimensions and provides some translation invariance.
4. **Why Order BN before Max Pooling Works:**
    - **Normalization of Features:** BN normalizes the feature maps. Max pooling then operates on these *normalized* feature maps. This means that the relative importance (the maximum values) within a pooling window are preserved, but their absolute scale is consistent across batches and layers.
    - **Preservation of Relative Information:** Max pooling is a non-linear operation, but it's generally considered "monotonic." If a value is the maximum in a window, it remains the maximum after normalization, even if its absolute value changes. So, the core function of max pooling (selecting the strongest feature response in a region) is not fundamentally disrupted by pre-normalization.
    - **Reduced Input Variability for Next Conv Layer:** By normalizing the feature maps *before* pooling, and then pooling them, the subsequent convolutional layer receives inputs that are already more standardized and stable. This helps in continuing the positive effects of batch normalization throughout the network.

In contrast:

- **Placing Max Pooling before Batch Normalization (Conv2D -> MaxPooling2D -> BatchNormalization -> Activation):** This is less common. If you pool first, you've already reduced the spatial dimensions. BN would then normalize a smaller feature map. While not "wrong" in all cases, it's generally seen as less effective in standardizing the *full* range of feature maps throughout the network compared to normalizing before downsampling.

Your previous experience where adding BN made it worse:

It's important to reiterate that while Conv2D -> BN -> Activation -> MaxPooling2D is the *standard good practice*, your prior experience (where Conv2D(..., activation='relu') followed by MaxPooling2D worked better than introducing BN) is a valuable empirical observation for *your specific setup*.

This could be due to:

- Suboptimal learning rate for BN (BN often allows higher LRs).
- The overall simplicity of your model not gaining significant benefit from BN's stabilization.
- The specific characteristics of your dataset.

Conclusion:

Yes, **it is generally okay and recommended to put Batch Normalization before Max Pooling.** The sequence Conv2D -> BatchNormalization -> Activation -> MaxPooling2D is a very robust and common block design in modern deep learning. If you tried it and it performed worse, it's worth re-checking your learning rate or sticking with your previous best-performing model (perhaps just trying GELU and padding='same' on the Conv layers). Deep learning is often about empirical iteration!

Additional references

https://www.geeksforgeeks.org/what-is-batch-normalization-in-deep-learning/

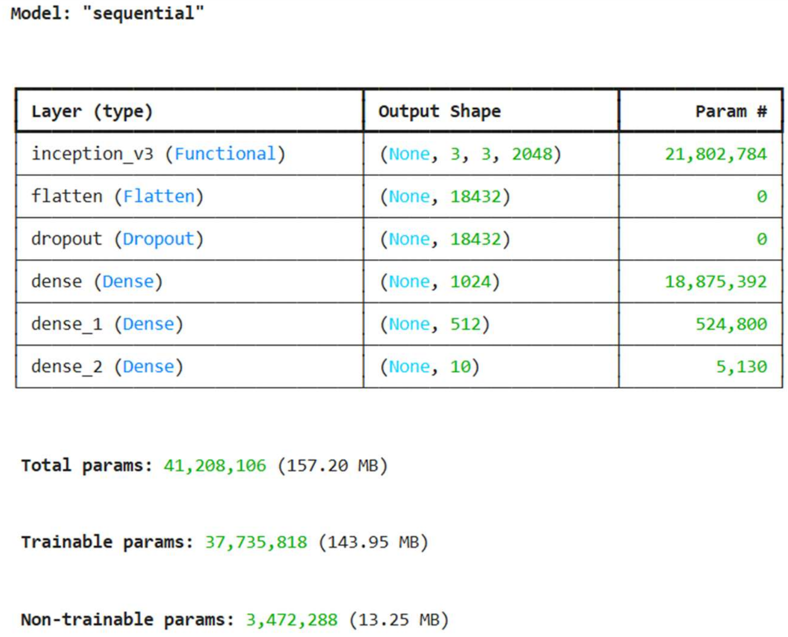
https://www.upgrad.com/blog/basic-cnn-architecture/

Annex D: Code set up of Model 1-5A5

```
model = models.Sequential()
model.add(keras.Input(shape=(img_size, img_size, 3)))
model.add(layers.Conv2D(32, (3, 3), activation='gelu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='gelu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='gelu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='gelu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='gelu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.summary()

# Compile the model

model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(learning_rate=1e-4),
              metrics=['acc'])
```

Annex E: Code set up of Model food_IN6

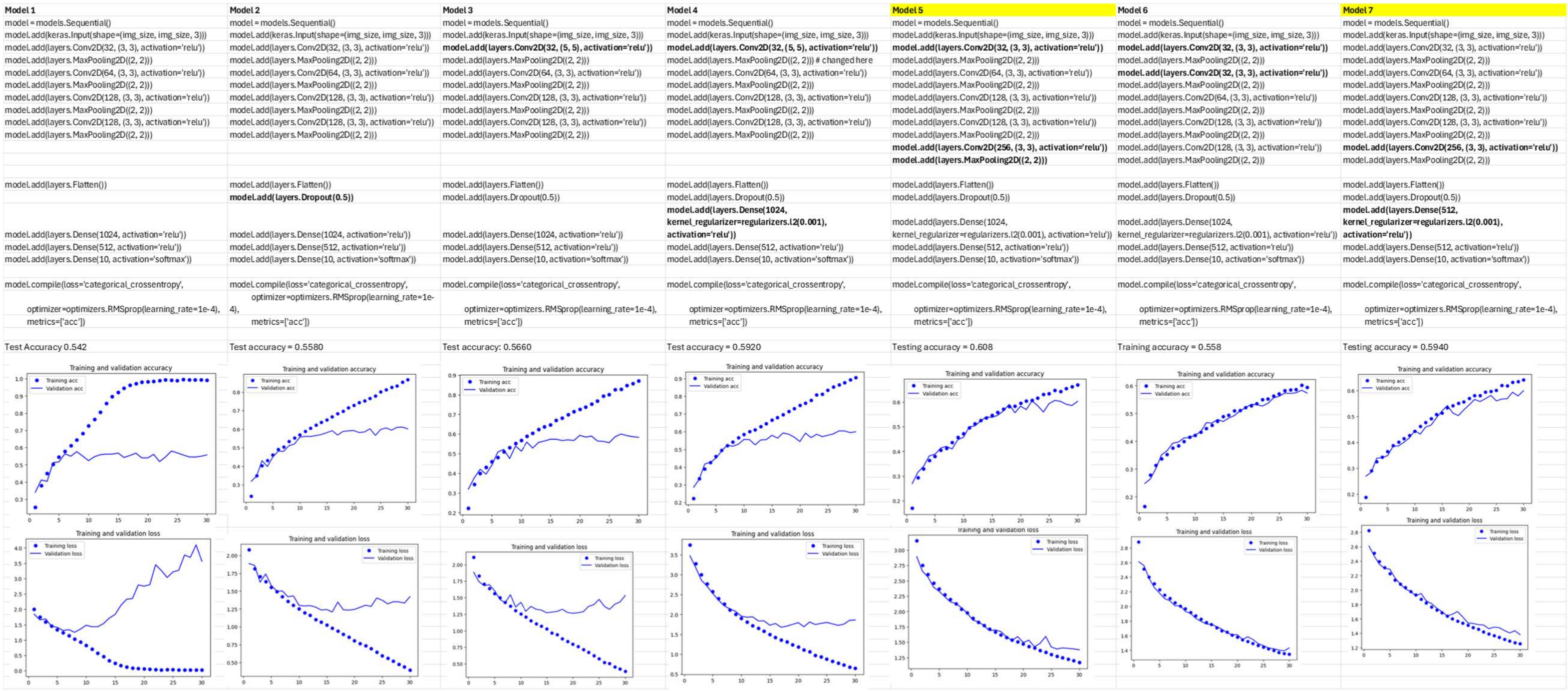
```
# get the model InceptionV3

from keras.applications.inception_v3 import InceptionV3
img_size = 150

conv_base = InceptionV3(
    include_top=False,
    weights="imagenet",
    input_tensor=None,
    input_shape=(img_size, img_size, 3),
    pooling=None,
)

conv_base.summary()

# transfer learning model setup

from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
import tensorflow as tf

model = models.Sequential()
model.add(conv_base) # InceptionV3 base
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# Freeze the convolutional base to prevent weight updates during initial training
conv_base.trainable = False


# codes for unfreezing layers

unfreeze_from_layer = 132 # mix4

for layer in conv_base.layers[:unfreeze_from_layer]:
   layer.trainable = False
for layer in conv_base.layers[unfreeze_from_layer:]:
   layer.trainable = True


# Compile the model after changing the trainable status of layers
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.Adam(learning_rate=1e-4),
              metrics=['acc'])

model.summary()
```



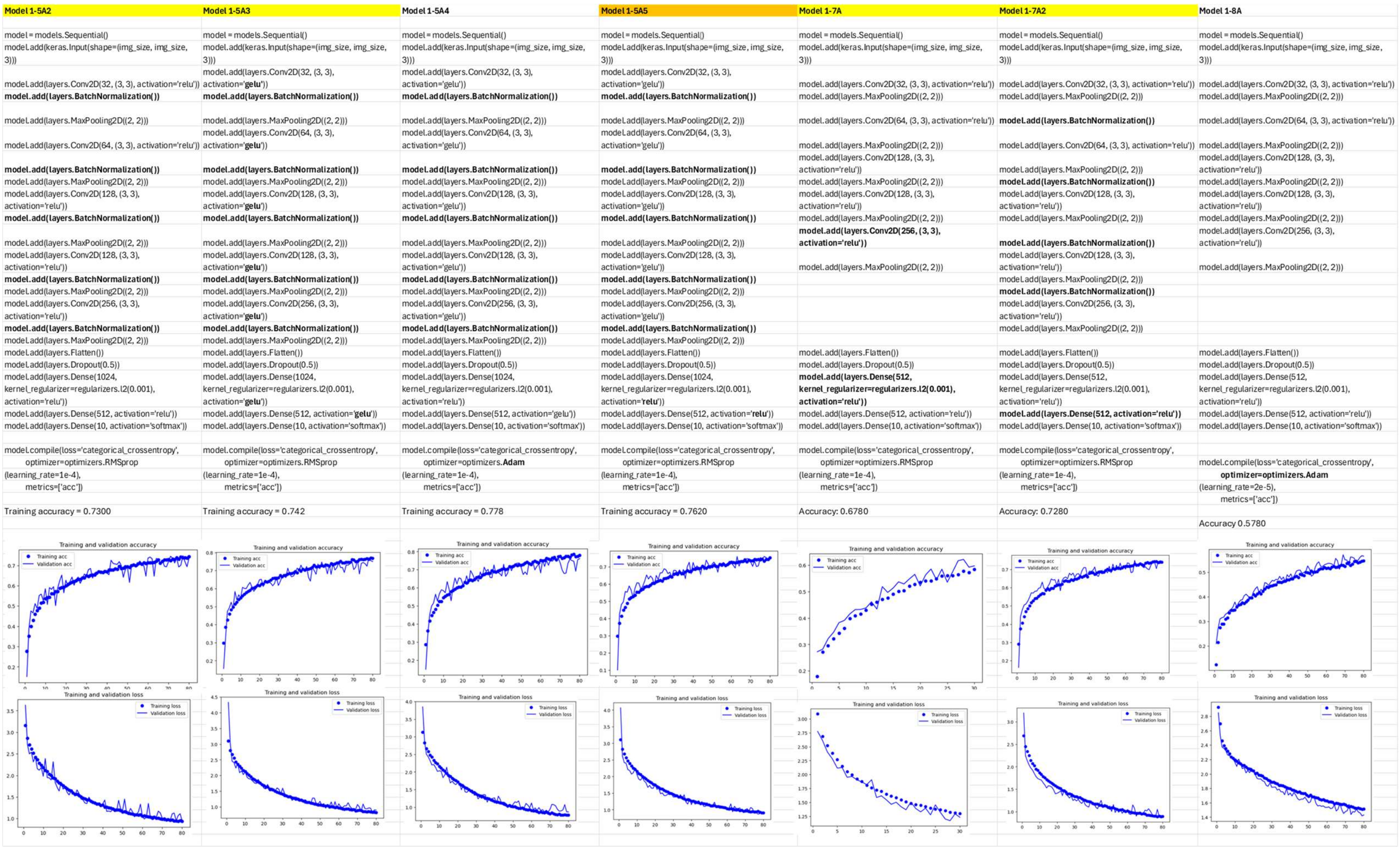
```
Model: "sequential"

┌──────────────────────────────┬──────────────────────┬──────────────┐
│ Layer (type)                 │ Output Shape         │      Param # │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ inception_v3 (Functional)    │ (None, 3, 3, 2048)   │   21,802,784 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ flatten (Flatten)            │ (None, 18432)        │            0 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dropout (Dropout)            │ (None, 18432)        │            0 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense (Dense)                │ (None, 1024)         │   18,875,392 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense_1 (Dense)              │ (None, 512)          │      524,800 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense_2 (Dense)              │ (None, 10)           │        5,130 │
└──────────────────────────────┴──────────────────────┴──────────────┘

 Total params: 41,208,106 (157.20 MB)

 Trainable params: 37,735,818 (143.95 MB)

 Non-trainable params: 3,472,288 (13.25 MB)
```

```
# model training
model.fit(
      train_generator,
      epochs=50,
      validation_data=validation_generator,
      verbose=1)
```

Annex F: Training progression of Customized CNN model: accuracy and loss graphs


Training and validation accuracy
Training acc
Validation acc
Training and validation loss
Training loss
Validation loss

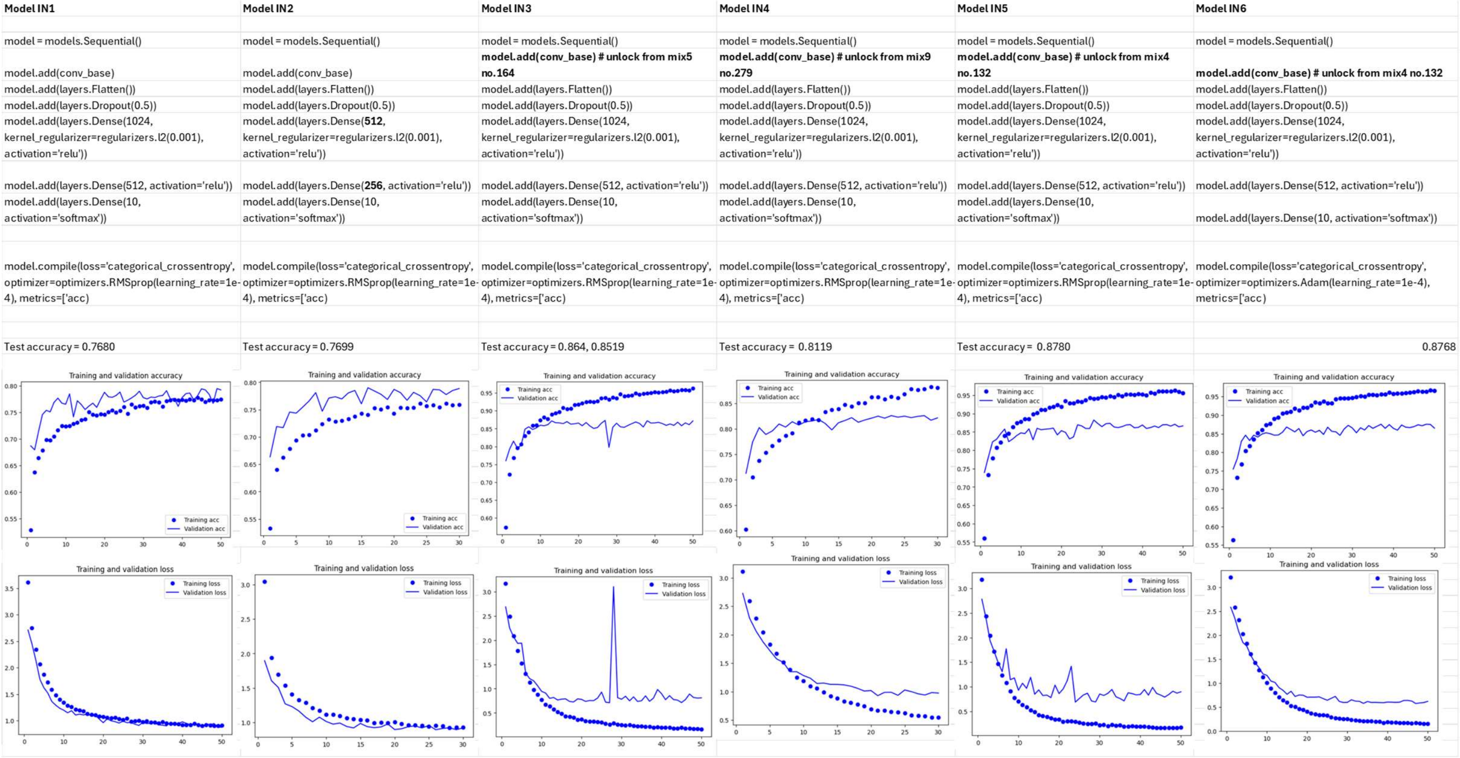
Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	Model 7
model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()
model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))	model.add(layers.Conv2D(32, (3, 3), activation='relu'))	**model.add(layers.Conv2D(32, (5, 5), activation='relu'))**	**model.add(layers.Conv2D(32, (5, 5), activation='relu'))**	**model.add(layers.Conv2D(32, (3, 3), activation='relu'))**	**model.add(layers.Conv2D(32, (3, 3), activation='relu'))**	model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2))) # changed here	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	**model.add(layers.Conv2D(32, (3, 3), activation='relu'))**	model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
				model.add(layers.Conv2D(256, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	**model.add(layers.Conv2D(256, (3, 3), activation='relu'))**
				model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())
	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, activation='relu'))	model.add(layers.Dense(1024, activation='relu'))	model.add(layers.Dense(1024, activation='relu'))	**model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))**	model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))	model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))	**model.add(layers.Dense(512, kernel_regularizer=regularizers.l2(0.001), activation='relu'))**
model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e- 4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'])
Test Accuracy 0.542	Test accuracy = 0.5580	Test accuracy: 0.5660	Test accuracy = 0.5920	Testing accuracy = 0.608	Training accuracy = 0.558	Testing accuracy = 0.5940



Model 1-5A2	Model 1-5A3	Model 1-5A4	Model 1-5A5	Model 1-7A	Model 1-7A2	Model 1-8A
model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()	model = models.Sequential()
model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))	model.add(keras.Input(shape=(img_size, img_size, 3)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))	model.add(layers.Conv2D(32, (3, 3), activation='**gelu**'))	model.add(layers.Conv2D(32, (3, 3), activation='gelu'))	model.add(layers.Conv2D(32, (3, 3), activation='gelu'))	model.add(layers.Conv2D(32, (3, 3), activation='relu'))	model.add(layers.Conv2D(32, (3, 3), activation='relu'))	model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	**model.add(layers.BatchNormalization())**	model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.Conv2D(64, (3, 3), activation='**gelu**'))	model.add(layers.Conv2D(64, (3, 3), activation='gelu'))	model.add(layers.Conv2D(64, (3, 3), activation='gelu'))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.Conv2D(64, (3, 3), activation='relu'))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	**model.add(layers.BatchNormalization())**	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='**gelu**'))	model.add(layers.Conv2D(128, (3, 3), activation='gelu'))	model.add(layers.Conv2D(128, (3, 3), activation='gelu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	**model.add(layers.Conv2D(256, (3, 3), activation='relu'))**	**model.add(layers.BatchNormalization())**	model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.Conv2D(128, (3, 3), activation='**gelu**'))	model.add(layers.Conv2D(128, (3, 3), activation='gelu'))	model.add(layers.Conv2D(128, (3, 3), activation='gelu'))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.Conv2D(128, (3, 3), activation='relu'))	model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**		model.add(layers.MaxPooling2D((2, 2)))	
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))		**model.add(layers.BatchNormalization())**	
model.add(layers.Conv2D(256, (3, 3), activation='relu'))	model.add(layers.Conv2D(256, (3, 3), activation='**gelu**'))	model.add(layers.Conv2D(256, (3, 3), activation='gelu'))	model.add(layers.Conv2D(256, (3, 3), activation='gelu'))		model.add(layers.Conv2D(256, (3, 3), activation='relu'))	
model.add(layers.BatchNormalization())	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**	**model.add(layers.BatchNormalization())**		model.add(layers.MaxPooling2D((2, 2)))	
model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))	model.add(layers.MaxPooling2D((2, 2)))			
model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())	model.add(layers.Flatten())
model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))	model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))	model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='**gelu**'))	model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))	model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='**relu**'))	**model.add(layers.Dense(512, kernel_regularizer=regularizers.l2(0.001), activation='relu'))**	model.add(layers.Dense(512, kernel_regularizer=regularizers.l2(0.001), activation='relu'))	model.add(layers.Dense(512, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
model.add(layers.Dense(512, activation='relu'))	model.add(layers.Dense(512, activation='**gelu**'))	model.add(layers.Dense(512, activation='gelu'))	model.add(layers.Dense(512, activation='**relu**'))	model.add(layers.Dense(512, activation='relu'))	**model.add(layers.Dense(512, activation='relu'))**	model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))	model.add(layers.Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy',	model.compile(loss='categorical_crossentropy',	model.compile(loss='categorical_crossentropy',	model.compile(loss='categorical_crossentropy',	model.compile(loss='categorical_crossentropy',	model.compile(loss='categorical_crossentropy',	
optimizer=optimizers.RMSprop	optimizer=optimizers.RMSprop	optimizer=optimizers.**Adam**	optimizer=optimizers.RMSprop	optimizer=optimizers.RMSprop	optimizer=optimizers.RMSprop	model.compile(loss='categorical_crossentropy',
(learning_rate=1e-4),	(learning_rate=1e-4),	(learning_rate=1e-4),	(learning_rate=1e-4),	(learning_rate=1e-4),	(learning_rate=1e-4),	**optimizer=optimizers.Adam**
metrics=['acc'])	metrics=['acc'])	metrics=['acc'])	metrics=['acc'])	metrics=['acc'])	metrics=['acc'])	(learning_rate=2e-5),
						metrics=['acc'])
Training accuracy = 0.7300	Training accuracy = 0.742	Training accuracy = 0.778	Training accuracy = 0.7620	Accuracy: 0.6780	Accuracy: 0.7280	
						Accuracy 0.5780

Training and validation accuracy
Training acc
Validation acc
Training and validation loss
Training loss
Validation loss

Annex G: Training progression of CNN model utilizing InceptionV3 convolutional base: Accuracy and Loss Graphs



Model IN1	Model IN2	Model IN3	Model IN4	Model IN5	Model IN6
model = models.Sequential() model.add(conv_base)	model = models.Sequential() model.add(conv_base)	model = models.Sequential() **model.add(conv_base) # unlock from mix5 no.164**	model = models.Sequential() **model.add(conv_base) # unlock from mix9 no.279**	model = models.Sequential() **model.add(conv_base) # unlock from mix4 no.132**	model = models.Sequential() **model.add(conv_base) # unlock from mix4 no.132**

model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
	model.add(layers.Flatten())
	model.add(layers.Dropout(0.5))
	model.add(layers.Dense(**512**, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
		model.add(layers.Flatten())
		model.add(layers.Dropout(0.5))
		model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
			model.add(layers.Flatten())
			model.add(layers.Dropout(0.5))
			model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
				model.add(layers.Flatten())
				model.add(layers.Dropout(0.5))
				model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
					model.add(layers.Flatten())
					model.add(layers.Dropout(0.5))
					model.add(layers.Dense(1024, kernel_regularizer=regularizers.l2(0.001), activation='relu'))

model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
	model.add(layers.Dense(**256**, activation='relu'))
	model.add(layers.Dense(10, activation='softmax'))
		model.add(layers.Dense(512, activation='relu'))
		model.add(layers.Dense(10, activation='softmax'))
			model.add(layers.Dense(512, activation='relu'))
			model.add(layers.Dense(10, activation='softmax'))
				model.add(layers.Dense(512, activation='relu'))
				model.add(layers.Dense(10, activation='softmax'))
					model.add(layers.Dense(512, activation='relu'))
					model.add(layers.Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc')	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc')	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc')	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc')	model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=1e-4), metrics=['acc')	model.compile(loss='categorical_crossentropy', optimizer=optimizers.Adam(learning_rate=1e-4), metrics=['acc')
Test accuracy = 0.7680	Test accuracy = 0.7699	Test accuracy = 0.864, 0.8519	Test accuracy = 0.8119	Test accuracy = 0.8780	0.8768