A picture containing text, clipart

Description automatically generated

School of InfoComm Technology

**Deep Learning Assignment**

Diploma in CSF / FI / IT

April 2021 Semester

**ASSIGNMENT 1**

(30% of DL Module)

11st May 2021 – 13rd Jun 2021

**Submission Deadline:**

**Presentation: 13rd Jun 2021 (Sunday), 11:59PM**

**Report: 13rd Jun 2021 (Sunday), 11:59PM**

|  |  |  |
| --- | --- | --- |
| **Tutorial Group** | **:** | **P03** |
| **Student Name** | **:** | Tan Zi Xu |
| **Student Number** | **:** | S10185168 |

**Penalty for late submission:**

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

**NO** submission will be accepted after 20th Jun 2021 (Sunday), 11:59PM.

Contents

[Overview 4](#_Toc74507409)

[Data Preprocessing and Data Loading 5](#_Toc74507410)

[Image Classification Model Training 7](#_Toc74507411)

[CNN Model Trained from Scratch 7](#_Toc74507412)

[Base Model 1 – Foundation 7](#_Toc74507413)

[Base Model 2 – Refine convolution 10](#_Toc74507414)

[Base Model 3 – Data Augmentation 11](#_Toc74507415)

[Model 1 Prototype 1 – Scale Up 13](#_Toc74507416)

[Model 1 prototype 2 – Stacking Conv2d 15](#_Toc74507417)

[Model 1 Prototype 3 – Dropout 17](#_Toc74507418)

[Model 1 Prototype 4 – L2 Regularization 18](#_Toc74507419)

[Model 1 Prototype 5 – Adam Optimizer 19](#_Toc74507420)

[Final Model 1 – Early Stopping at epoch 60 20](#_Toc74507421)

[Pretrained CNN Model 22](#_Toc74507422)

[VGG16 – Base 22](#_Toc74507423)

[VGG16 – Feature Extraction with Data Augmentation 23](#_Toc74507424)

[VGG16 – Fine Tuning 26](#_Toc74507425)

[VGG16 – Fine Tuning with Regularization Techniques 27](#_Toc74507426)

[DenseNet121 28](#_Toc74507427)

[DenseNet121 – Base 29](#_Toc74507428)

[DenseNet121 – Fine Tuning with Regularization Techniques 30](#_Toc74507429)

[DenseNet121 – Adam Optimizer 32](#_Toc74507430)

[Final Model 2 – DenseNet121 Early Stopping at Epoch 20 33](#_Toc74507431)

[Model Evaluation 35](#_Toc74507432)

[Test Accuracy 35](#_Toc74507433)

[Model 1– Confusion Matrix 36](#_Toc74507434)

[Model 1 – Classification Report 37](#_Toc74507435)

[Model 2 – Confusion Matrix 38](#_Toc74507436)

[Final Model 2 – Classification Report 39](#_Toc74507437)

[Best Model Verdict 39](#_Toc74507438)

[Application of Best Model on Real Life Image 40](#_Toc74507439)

[Chosen Images 41](#_Toc74507440)

[Results – Carrot Cake 41](#_Toc74507441)

[Results – French Toast 42](#_Toc74507442)

[Results – Lobster Roll 43](#_Toc74507443)

[Summary 44](#_Toc74507444)

[References 45](#_Toc74507445)

# Overview

This is an individual report of a Deep Learning module assignment, this report is the documentation of the progression in this project. The objective of this assignment is to build an image classification model that can recognize and classify 10 different types of food.

Food has always been a necessity for living creatures to sustain their daily activities, it fuels our bodies to live. To humanity, food as a topic is broad but it is also deeply rooted to different cultures and cuisines. There could be some visual similarity or semblance between 2 different foods of different culture and cuisine. In the modern context, food has evolved greatly to satisfy our cravings and there is a call to eat healthy. To do that, there are ideas of deploying deep neural networks to identify food. It may sound weird, but the application is to make use of the identification to calculate the amount calories intake in healthy lifestyle apps. That is just an example of what application of deep learning technology is capable of in food industry.

With every big objective or goal, there will be problems, challenges and obstacles that follow along. The very first problem is to identify an approach that delivers a developed model capable of application with real life images. The next problem would be quality of prediction made by the model, some of the features amongst the 10 foods are similar. The 10 foods that I am tasked to classify are beet salad, carrot cake, chicken quesadilla, French toast, garlic bread, lobster roll sandwich, mussels, pad thai, peking duck and tuna tartare. I think in some cases, the model could misclassify French toast and garlic bread. Lastly, the final problem is that there is no “one size fit all” model, there will never be an exact model that can be fitted in all problems and perform well. Each case and model need to develop delicately based on the observation made throughout development.

My approach for this project would be to develop a Convolutional Neural Network (CNN) model of my own design and another that contain components of pretraining by others. After developing, I will make a fair evaluation of both models based on their predictions and select the final model to be applied on real life images. I will attempt to follow the universal workflow of machine learning during my model development phase of this project. I also intend on deploying confusion matrixes and classification report during the model evaluation to gain deeper insights on the model performance.

# Data Preprocessing and Data Loading

Before working on modelling, preparing the data is the first task on the list. The dataset I will be working with is a set of 101,000 images of 101 different foods, there will be 1000 images for each type of food. However, the actual dataset I am working with will be a scaled down version to the 10 assigned food that I have to classify. Here are the 10 different food that I am tasked to classify:

Text

Description automatically generated

To help me dig out the images of those 10 different foods, a jupyter notebook was provided. The provided file will also help me prepare the images to be split for training, validation and testing purposes during the model training. For each of the 10 foods, there are 750 images for training, 200 for validation and 50 for testing. Effectively, giving me a total of 10000 images as the final dataset for this project.

After I got my data, the next step is to read the images and convert it into something readable for my model. The images need to be in a certain size, transformed into the RGB channels and the pixel values to be rescaled. This is because during the model training later, there might be models that require specific image input sizes. The RGB channels requirement is there as I would like my model to train with images that has colors instead of grayscale images. Lastly, the rescaling of pixel values will assist in speeding up the convergence of my network training.

As the data are stored in my directory, I will be using the Image Data Generator from the Keras library through the Tensorflow API. The Image Data Generator is a tool that helps to create batches of the image data into tensors as the model input. This tool is capable of resizing images to my desired size, converting jpeg files to RGB format and it also allows me to rescale each pixel values, which checks out all the requirement for data preprocessing and data loading.

Text

Description automatically generated

Referring to the image above, this is how I used the Image Data Generator and conduct the data preprocessing and loading step. Firstly, I set the proper directory address for the training, validation and testing data. Then, I set the image size to be 150 and this will used in a moment. Next, I instantiated an Image Data Generator as “base\_IDG”, this will be a basic generator that will at least rescale the pixel values of the images. Lastly, I will generate the data itself into tensors with the base generator. The generator will use the “flow\_from\_directory” function to draw out each image from the appropriate folders based on the directory addresses set earlier. Generating the training, validation and test data, I had the images to be 150 pixels by 150 pixels with ‘target\_size’. Using “class\_mode = ‘categorical’”, the generator also can automatically label which type of food the image is. Lastly, I decided the batch size of the data was something I would not like to be small. Because to my knowledge, small batch sizes would make a model update its weight frequently, which is something I want to avoid. Hence, I pick batch sizes that is relatively a specific ratio to the whole data it came from, or the batch size would at least be 50. For example, training data are packed into batches of 75 with the 7500 training data images, which would be a 1 to 100 ratio.

# Image Classification Model Training

With the images loaded in and preprocessed, I will create a Convolutional Neural Network (CNN) model from scratch with my knowledge of using Keras’s sequential model with Conv2d and Dense layers. I will also develop another CNN model in the Keras library that was already pretrained.

## CNN Model Trained from Scratch

### Base Model 1 – Foundation

As I will be building a model from scratch, I believe it would be best to create a simple base model first. Then, I could scale it up to perform the best it can get before overfitting. Lastly, I can do some tuning or regularizing it.

Text

Description automatically generated with medium confidence

Starting with this simple CNN model, I make it simple as I could possibly think of. Which is just to have only 1 convolution layer with relu activation first, then do max pooling and flattening before connecting to the dense layer of the network. Going in order of this network topology, the convolution layer will receive images with “input\_shapes” of the 150 by 150 image size and the 3 representing the RGB channels. The convolutional layer uses relu activation as the relu is known to help models converge faster. With filters generated to identify features in the images, there is still a lot of pixels that does not really contribute much to predict an image. To pull out simple or even key features of an image, pooling needs to be done and it come in the form of either max pooling or average pooling. I will be using max pooling all the time as max pooling only takes out some of the most defined pixel within its pooling size. With the tensors coming out of the conv2d layer, its shape that is influence by the number of filters would create depth and the upcoming dense part of the network would not be able to be fed in. Thus, the tensors would be flattened and feed through the first dense layer with relu activation. This layer was set with 32 neurons just to be simple and relu activation will still be used for its convergence. All of which would be connected to the final dense layer as the output. This final layer only has 10 neurons to represent the 10 different classes of food there are. The layer will also use softmax activation instead of relu to help determine what food the image is in this single label, multiclass classification project. It will calculate the possibility of each possible option the food could be pick the one with the highest possibility as the predicted choice.

I think is a simple model that lays what are some fundamental foundations the final model will look like or have at the end. Such that, the model can perform convolution, pooling, being fed into a dense network and make prediction based on the features it “sees” in the convolution to be qualified as a CNN model.

Text

Description automatically generated

With the first iteration of the model completed, I need to do compiling such that the model will learn using the proper metric and how to calculate its loss during the training. In here, I first set the loss to be ‘categorical\_crossentropy’ as I am doing a single label, multiclass classification. I am neither doing binary classification nor using sparse data by one hot encoding. Thus, I will not be using ‘binary\_crossentropy’ or ‘sparse\_categorical\_crossentropy’ as loss. The optimizer I will use is RMSprop, it is one of the many optimizers available in the keras library, but it does typically work well. I do intend on trying out another optimizer towards the end of developing this model to see how it performs. Next, the learning rate of the RMSprop was set to the default of 0.001, I could have left it out to use the default, but I want to be aware of the learning rate. Hence, I manually set the learning rate with a value. Lastly, the metric is ‘acc’, which is accuracy. Since we are doing a classification task and not a regression problem, accuracy is right metric for measuring the model’s performance.

Following compiling, comes the fitting. This is step where my model will take in the training data and improve over iterations while also running through validation data to read the signs of how the model trained. In the codes, there is the ‘steps\_per\_epoch’ and ‘validation\_step’, which is stating number of training and validation data batches the model goes through in each epoch. It is dependent on the batch size, and it also shows how frequent the model will tune its weight. As small batch sizes with large data will give a huge number of batches, the number of times that the model will update its weights in each epoch is the equivalent of that number of batches. Lastly, I set the epoch to be 20 as I want to start out small and see how the model will fare. It would be inefficient to run many trainings epochs, just for it to overfit or underfit.

Chart, scatter chart

Description automatically generated

To make further decisions as to what I can do to improve the model, I have created a chart tracking the model’s accuracy, validation accuracy, loss and validation loss during the training. Referring to the chart above, I find that the model is already grossly overfitted at the fifth epoch. The performance of the model is also rather low at around the ~28% for validation accuracy, there is a lot of room for improvement. It is at this stage that I find having 1 Conv2d relu layer and max pooling layer is very limiting for the model. Such that, when I look at a sample image of the data, there are refined curves and having 1 Conv2d layer at best could only picked up simple lines or features. I can support this thought process with the fact that the first base model has around 3 million trainable parameters, which suggest there are a lot of small features for the model work with. The small features are not able to make bigger and much definitive features of the food. I feel that I should rerun this model with another conv2d layer that has more depth will increase the accuracy. With a higher expectation, I felt that I could also add on more training epochs. Hence, I created the second base model with the very much needed extra Conv2d relu layer, max pooling layer and additional 10 training epochs.

### Base Model 2 – Refine convolution

Graphical user interface, chart

Description automatically generated

In this second iteration, I feel I only achieve a small goal where the model’s validation seems to have improve on average by ~5-6%. The validation accuracy even peaked around 40%, which motivates me to push model beyond that in the remaining iterations of the model. However, there is a glaring problem with the overfitting. In fact, the model has overfitted even earlier than the first model and there are 2 reasons to this problem that comes to my mind. Firstly, the dataset may not be enough, it might be a lack of variation in the images, the model looks to be over reliant on the features of the training data and ignoring actual features that it should be considering. The other reason being the model is not complex enough to learn effectively. However, the model would more likely be underfitting if it was the second reason. Hence, I need to find a way to work with more data, adding data from external sources is not an option and my solution is to use data augmentation. By augmenting the image, I will be feeding my model images from different angles and perspective, which will exponentially increase the variation of images. To execute this solution is also very quick and simple, the answer lies in the image data generator that I used for preprocessing earlier. I can load in the training data with the augmentation setting in the generator and it is unlikely to have the same image being framed the same way. I am not necessarily increasing the number of data but the variation it comes in.

### Base Model 3 – Data Augmentation

Graphical user interface, text

Description automatically generated

Here is how I used the Image Data Generator to do data augmentation with the training data. There are various augmentations that can done to the images, and I just need to specify to what extend the generator can modify it to. I felt that the configuration above would be a decent fit, each of the specification has reason behind it. Firstly, I want the images to be seen varying angles, which I will set a high rotation range for. Next, I would not want the images to be shifted too far out of the frame, so the range was limited. Lastly, the zoom range and horizontal flip helps to increase the variation of images as well.

With the generator specification made, the same process of creating training data batches is applied here. This new augmented training data will be used primarily for the remainder of the project. There is also an example of an augmented data in the picture above.

The augmentation will only be done on training data as the validation and test data should remain untainted to retain its integrity for testing purposes.

Table

Description automatically generated with medium confidence

With augmented data available, I ran through the same model without any changes to design or topology. However, the model is now fitted with the new data and put through more training epochs.

Chart, scatter chart

Description automatically generated

Referring to the charts above, I am generally satisfied with the implementation of using augmented data. The validation accuracy has skyrocketed beyond baseline of 40%, the improvement was at least an increase by 13% into ~53%. There is a sign I am worried about, such that the variance in validation loss might be something I have to tackle soon. However, looking at the overall progress of this model performance, I think it would right to scale up the model to achieve greater heights. I intend to include a few more convolution and pooling layers. I also think it would be nice to add in a hidden layer for the dense network portion of this model, but I would also need to greatly increase the width of each dense layer.

### Model 1 Prototype 1 – Scale Up

Table

Description automatically generated with medium confidence

In this scaled up version of the base model training on augmented data, there are many new details equipped to the model. Firstly, the model has deeper initial convolution at the first layer while the filter size remained 3 by 3. Secondly, the number of convolution and pooling has increased to 4 times, with each conv2d and max pooling layer adding a greater depth. Lastly, the dense portion of the network is significantly widened to 512 neurons for the first layer and introducing a 128-neurons hidden layer with relu activation. This scaled up model sees a tremendous increase of parameters to 14.5 million for training that is at least 6 times more. As I foresee the possibility of overfitting with this new model, I decided to slow down the learning rate. Accompanying those changes, I have also increased the number of training epochs to see how the learning will curve out.

Chart, scatter chart

Description automatically generated

Referring to the charts above, I think the model’s panned out decently. It showed that it is capable of learning, raising the validation accuracy to a new height around 65% while the accuracy peaked close to 80%. However, it is important to note that it is overfitting after around the 40th epoch and the variance in loss does look worrying. I feel that I could try a slightly different approach in terms of the model design to improve performance. I will be referencing how the VGGnet structure look like and attempt to implement it. The feature that I want to implement into my model is how the VGGnet does their convolution.

### Model 1 prototype 2 – Stacking Conv2d

Diagram

Description automatically generated with low confidence

The image above is an overview of the VGG16 architecture, and I want to try replicating its convolution steps into my model. Seeing how there are conv2d layers stacking on each other before doing the pooling, I think it is an approach I could try. Because by having stacked conv2d layers, I could do more convolution before having information loss between pooling and the next layer. It is something worth trying and I could cross check how my model performs against VGG16 network.

Table

Description automatically generated

With the VGG16 model as inspiration, my model now has stacked conv2d layers before doing the max pooling. The additional conv2d layers are set to have the same filter size as the level they are respectively at. This stack conv2d model has 3 million less training parameter than the previous one. Although, it can be attributed to how the final tensor size before feeding into the dense layers is smaller from (7, 7, 512) to (5, 5, 512). This reduction in size would result in lesser parameters connected to the first layer after flattening the tensor. The compiling specification were left as it is and reused for this model.

Chart, scatter chart

Description automatically generated

Frankly, the model did not show a major improvement in terms of accuracy, averaging around 68% before overfitting and training accuracy peaking at 80%. However, I do like how this model performed much more stable when compared to non-stacked conv layer model. We can see in how the overfitting starts in later epochs and the gap between the accuracy and validation accuracy is much closer than before. Another good sign is variance in validation loss is also much more controlled with model. Seeing these results, I would continue developing this stacked conv2d layer model.

Moving on, I think the overfitting issue can be addressed now and I think I should deploy a couple of regularization techniques, between implementing dropout or L2 regularizer. Personally, I think starting off with implementing dropout layers is better than regularizers. As randomly dropping some neurons for each training epoch would allow my model to train in a way that is not monotonous. Regularizers would only slow the time the model takes to overfit, and I think I would deploy it if dropout were not enough.

### Model 1 Prototype 3 – Dropout

Graphical user interface, text, application

Description automatically generated

This is next iteration of the model with dropout layers introduced to reduce overfitting. I have set the dropout to 0.5, representing that half of the parameter will dropped off from the network during training. Dropping half may seem too much but I think it is justified as the first and hidden layer is already decently wide. Even if I were to lose half in the first layer, there is still 256 neurons connecting to the hidden. Without any other changes. I proceed with fitting the data into the model.

Chart, scatter chart

Description automatically generated

Looking at the charts above, I am very satisfied to see the overfitting being reduce from a difference of 10% between accuracy and validation accuracy to around 5%. It is effectively removing half of the overfitting that I had before, and I believe I should introduce L2 regularizer to the model. Regarding about the model performance, I have a feeling the limit of accuracy I get is around 68 to 70%. Which is why I will dedicate the next remaining improvement goal towards minimizing the overfitting by the end of 100 epochs.

### Model 1 Prototype 4 – L2 Regularization

Text

Description automatically generated

In this L2 regularizer implemented model, I have only attached it to the dense layers of the network excluding the output. Reason being that I feel only neurons deciding the presence of features in the first and hidden layer should be regularized. Such that, the conv2d layers and the final output layer should remain as it is. I do not think it make sense for those layers to be regularized to ensure my prediction output is the same.

Chart, scatter chart

Description automatically generated

Looking the model performance, I am surprised to see high variance in the validation loss and unstable validation accuracy. I was expecting the validation accuracy to track along with training accuracy closely. There is marginal overall improvement to the accuracy, averaging around 69-70%, but it is worrying to see the validation accuracy ending at a bad note in the last epoch. As for further improvement, I think I could try see how the model would work with another optimizer, which was an intended experiment I wanted to try earlier.

### Model 1 Prototype 5 – Adam Optimizer

Text

Description automatically generated

Keeping the same model with l2 regularizer and the dropout layers, I will be compiling the model with the Adam optimizer. It is also a popular optimizer that rivals RMSprop in popularity as I was told, so it would not hurt to try it out.

Chart, scatter chart

Description automatically generated

Referring to charts above, I really like how the model performance curved out. The variance in accuracy is much more control when compared to using RMSprop. Using Adam optimizer might work better in this problem. However, for my model to perform consistently, I might have to deploy my last resort in early stopping the model. Judging by the charts above, I think 60 epochs is a good stopping point before it starts overfitting. As the validation accuracy can consistently hit 70% without the training accuracy being too far off at ~72%.

### Final Model 1 – Early Stopping at epoch 60

Table

Description automatically generated

This will be the final iteration of the model I built from scratch, inspired from VGG16 architecture. The model does convolution with 2 conv2d layers stacked between before going through max pooling. The convolution and pooling process will be done 4 times, creating a tensor that has a shape (5, 5, 512) before being flattened. The flattened tensor will be fed into a dense layer with 512 L2 regularized neurons. It will be then connected to a hidden layer of 128 L2 regularized neurons and finally connected to 10 different outputs in the last layer. There are dropout layers between each of the dense layers leading up to the final output.

Chart, scatter chart

Description automatically generated

I am satisfied with this final model ending its 60 training epochs at 73.65% training accuracy and 70.05% validation accuracy. The validation loss throughout the training has demonstrated control in variance, the training was done in a stable manner. I will be using this final model to compare against the pretrain model in the testing stage. Hence, it will be saved as a h5 file, named as ‘food\_model\_1.h5’, to be loaded again later.

## Pretrained CNN Model

Starting right off with pretrained models, there are a variety of models in keras library for me to use. In this project, I am thinking of deploying 1 pretrained model and connecting it to the same dense portion of my made from scratch model above. I do not think I will be using any regularization techniques first and I should go straight to working with augmented data. My pretrained model of choice is the VGG16 as I am more familiar with it and my built from scratch model was inspired from the architecture. Doing a comparison would nice as there are still key differences between my model and the VGG16.

### VGG16 – Base

Table

Description automatically generated

Referring to the picture above, what I am doing with VGG16 model is that I have downloaded it to do convolution. I will not need to create any conv2d or pooling layers, but I still must create a dense network to make prediction later. In the VGG16 model I have downloaded, I have also downloaded the imagenet weights along, that the model was already trained on. This model has already been trained to recognize various lines and features from the big imagenet dataset, which will be used to train and recognize food in our dataset. Looking at convolution model itself, the number of training parameters is already more than the convolution and dense part of my built from scratch model combined.

### VGG16 – Feature Extraction with Data Augmentation

Table

Description automatically generated

Starting out with a basic model that can handle augmented data, I have the VGG16 model as the convolution base, a flatten layer to receive the tensor input for my dense portion of the model. The dense network has the same structure as my built from scratch model above without any regularization or dropout. However, I will still need to freeze the parameters in the conv\_base before I start any training. As I want to retain what the model has learn from imagenet, the parameter must be frozen for it to not change throughout the training.

Table

Description automatically generated

With the conv\_base frozen, my model only needs to train 4 million parameters and I can start the compiling and fitting process.

Text

Description automatically generated

In the compiling specification, I will follow a similar setting of RMSprop optimizer, categorical crossentropy loss and accuracy metric. As there is no change to the problem requirement of classifying 10 different foods. However, I will be using a slower than default learning rate, which is used in my current built from scratch model. The reason being that I foresee overfitting issues with such a big network, hoping that this learning rate that worked for other models would help in delaying the overfitting.

Moving onto the fitting, I am using the same augmented training data as before and the steps per epoch will remain the same. Although, I will be training this model for 30 epochs for a quick view of how the model would behave.

Chart

Description automatically generated

Referring to the charts above, the model did not overfit on one hand. On the other hand, I am unable to tell if it is underfitted. I do know that it has at least predicted more than half of the images correctly at around 60%. I think the model is at least learning properly, for the model to improve its accuracy I am thinking of fine tuning instead of scaling up. As pretrained models are already complex enough, I do not think I need to further scale up the model complexity. Fine tuning would loosen up the rigidness of the frozen parameters while I turn the model to learn more toward the current requirement. Instead of being fully reliant on the weight the model has learn with imagenet, unfreezing the convolutions would help it learn according to the current problems. However, I still must retain most of what it has learn through imagenet as it is still valuable learning for the model. Hence, I will be unfreezing the last block of convolution to let my model train on.

### VGG16 – Fine Tuning

**Table

Description automatically generated**

With the last block of the VGG16 unfrozen, I have a total of around 11 million parameters to train in this model. Keeping the same compiling specification, I will also run the model to train for 50 epochs now as 30 epochs may not be enough to see when the model overfits.

Chart

Description automatically generated

Looking at the result in the charts above, it is nice to see the model being capable of reaching accuracy around 73% before overfitting. However, it is a problem that the model had overfitted early on after the first 10 epochs. I think I will deploy regularization techniques to reduce the overfitting. Since it is massively overfitted, I will implement both dropout and L2 regularizer to tackle this issue.

### VGG16 – Fine Tuning with Regularization Techniques

Text

Description automatically generated

This is the model with dropout layers between the dense layers and L2 regularizer on the neurons in the dense layers too. The setup is like the one used for the model built from scratch with similar reasons backing this decision. The model will also have it last conv block unfrozen for fine tuning, the compiling and fitting process will also remain the same with exception of extending training epochs to 100.

Graphical user interface

Description automatically generated with medium confidence

Referring to the charts above, I still got back somewhat similar results with exception of observing more noise in the loss throughout the training. I think I might have to work around this problem from another angle. In hindsight, I think at this stage, I could draw some comparisons between this model and the one I replicate from scratch. The pretrained VGG16 model provides a slightly higher accuracy but at the cost of overfitting early when compared model from scratch. I am glad that I got close to replicating the accuracy. Although I feel that in this project, I am still limited in approach from the model design perspective, I think I will look to deploy a different pretrained model.

### DenseNet121

My new approach would be to use the DenseNet121 pretrained model. During my search for another pretrained model, I wanted to use a model with slightly lesser complexity than VGG16 and the architecture would be something different. This is where I came across the deep convolutional network, DenseNet. DenseNet was created by the author’s motivation to simplify connectivity pattern. (Ruiz, 2018) As information from input to output of the convolution can be lost, DenseNet will use fewer parameters to eliminate learning of redundant feature maps. (Ruiz, 2018)

Diagram, engineering drawing

Description automatically generated

The image above is to help visualize the structure of DenseNet, it shows how the model connect each layer with one another. Where quoted from an article, “each layer has direct access to the gradients from the loss function and the original input image”. (Ruiz, 2018)

### DenseNet121 – Base

Table

Description automatically generated with medium confidence

Following the same step of downloading pretrained models, I have downloaded the DenseNet121 with their imagenet weights. While the model looks very deeply complicated and densely interconnected with different layers, the model has lesser training parameters than VGG16 by half. Proceeding with this new model, I have decided to immediately start on fine tuning with the other specification of dropout and regularization. This is because I do not see the need to start all over again from ground up. I have already seen the progression of moving from feature extraction with data augmentation to fine tuning with regularization techniques, I think it is alright pick up where I left off.

### DenseNet121 – Fine Tuning with Regularization Techniques

Table

Description automatically generated with low confidence

The model above is newly created with DenseNet121 as the convolution base, and there are total of 15 million parameters, not too far from VGG16 base model. I will proceed to freeze all DenseNet121 layer except for the last few layers as fine tuning.

A picture containing table

Description automatically generated

After freezing all layers except for the last few layers, I am left with 8.6 million trainable parameters, which mostly came from connecting with the first dense layer. The compiling and fitting process will remain the same as it was in the previous model.

Graphical user interface, application

Description automatically generated

Referring to charts above, the validation accuracy peaked around the 73% mark before overfitting with the training. I am surprised to see the DenseNet121 base to exhibit similar behavior to the VGG16 base. The problem with fine tuning pretrained models is that they seem to have tendency to overfit easily. I think I would like to try using the Adam optimizer to invoke a different training curve. I think it might be redundant to train the model for 100 epochs. Hence, I will also greatly reduce the number of training epochs to 50.

### DenseNet121 – Adam Optimizer

Table

Description automatically generated

Chart

Description automatically generated

Looking at the charts above, I think the Adam optimizer was only able to help reduce the variance in the loss. While the accuracy still curves out similarly, I think that the best I could do. I am once again left with early stopping as the final resort and I think the 20th epoch is the right place to stop. As the validation accuracy is averaging around 72% before overfitting at that epoch range, I am expecting the model to produce similar result.

### Final Model 2 – DenseNet121 Early Stopping at Epoch 20

Text

Description automatically generated

This is the final iteration of my model 2, built with DenseNet121 as the convolution base loaded with imagenet weights. Attaching to the end of the DenseNet121 is a flatten layer and dense network to make the classification. The dense portion of the model are connected with regularized neurons and dropout layers between dense layers. The dense network mimics the structure that is found in model 1 as well. While the model is relying on the imagenet weights in the convolution, fine tuning of unfreezing the last few layers in the convolution base are made as well. The model is compiled with Adam optimizer, categorical cross entropy loss and accuracy metric. The model is also fitted with augmented data for 20 training epochs in this final training of model 2.

Chart, scatter chart

Description automatically generated

The charts above are performance of model 2’s final iteration, it is unfortunate to see the validation ending at bad note of 70.9% while the training accuracy is 75.05%. It fell under my expectation, and it is marginally closed to the final validation accuracy of model 1. Taking this model performance as it is, the model is saved in a h5 file as ‘food\_model\_2.h5’.

# Model Evaluation

In this section, I will be evaluating both models that was developed earlier to determine the better model between the 2. My evaluation will not solely rely on the testing accuracy, I will be using confusion matrix and classification report assess the models’ prediction. I will explore the quality of the models’ prediction and what are some of the interesting finds.

### Test Accuracy

Text

Description automatically generated

Model 1’s testing accuracy came out to be 71% and I think it is nice considering the model’s validation accuracy was 70.05%.

Graphical user interface, text

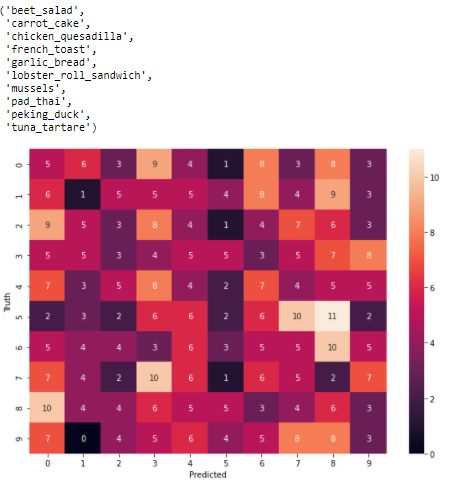
Description automatically generated

Model 2’s testing accuracy came out as a surprise, the model ended training with a validation score of 70.9% but still ended the testing with 74.2% accuracy. It is a little weird as testing and validation accuracy are typically very close to one another.

Looking at the testing accuracy purely, model 2 has the edge over model 1. However, the intended evaluation was to be done with confusion matrix and classification report.

### Model 1– Confusion Matrix

I do not think having testing accuracy would suffice in helping me determine the better of the 2. To assist me in making my decision, I believe I should see how the model predicts going against the true labels. I would be able to know what the model’s strength and weaknesses are. Hence, I am using confusion matrix to visualize the model’s predictions here.



Looking at a brief overview, model 1 is very bad at correctly classifying carrot cake and lobster roll sandwich. The model was better at classifying peking duck but, the model seems to be predicting a lot more wrong peking duck when compared to the other food. Such that, it had frequently predicted lobster roll and mussels to be peking duck. Some other common misprediction also comes in the form of beet salad as peking duck, french toast as pad thai and pad thai as lobster rolls. Most these misprediction are 1 way such that there is lesser case of misprediction if I swap position of predicted label to true label. Overall, the distribution of misprediction is broad and the number of true predictions is low. I think this heatmap is accurate considering my model has a meagre accuracy of 71% instead of high 80 or 90%. Which is why I will be using classification report to get deeper insights on the classification on individual classes.

### Model 1 – Classification Report

Table

Description automatically generated

Looking at the classification report, model 1’s top 3 most precise prediction of foods are chicken quesadilla, mussels and pad thai. The precision refers to how many correct predictions my model made against the prediction of a particular predicted label. The foods with the top 3 precisions all scored 9%. Whereas the least precise food was carrot cake at 3% precision, significantly lower than the precision of other food classes. Which proofs the model’s weakness in not properly identifying carrot cake correctly.

Moving onto the recall score, which refers to the number of correct predictions against the prediction of a particular true label. As expected, the recall score for carrot cake is the worst out of all the food classes. The highest recall score being peking duck reinforces the point that model 1 is great at classifying peking duck.

Looking across the whole performance of each individual classification of the classes, I will use the f1-score as a measure between precision and recall. Referring to the f1 scores above, the performance of all foods is even at around 7% to 10% except for carrot cake and lobster roll. I think the read of this classification report enforces the model’s weakness in predicting carrot cake and lobster roll while perform evenly across the other food classes.

### Model 2 – Confusion Matrix

A picture containing treemap chart

Description automatically generated

The visualization of this confusion matrix was not what I expected, in the overview, it seems that model generally did poorly to correct predict the labels. It is very shocking to see the model not being able to correctly predict a single tuna tartare image. Noteworthy points of this confusion matrix are that model 2 seems to predict a lot of French toast and lobster roll into the wrong true labels. French toast was predicted to be beet salad, chicken quesadilla and lobster roll multiple times. While lobster roll was predicted into mussels, pad thai, peking duck and tuna tartare. I have a feeling model 2 will not turn out to well in real life application when compared to model 1.

### Final Model 2 – Classification Report

Table

Description automatically generated

Looking at the classification report, model 2’s top 2 most precise prediction of foods are pad thai and peking duck. Both pad thai and peking duck scored 11% and 12% precision, respectively. Whereas the least precise food was tuna tartare at 0% precision, it is the only food with 0 correct prediction. Which showed the model’s weakness in being unable to properly identify tuna tartare correctly.

The recall score for tuna tartare is the worst out of all the food classes as expected with 0 correct prediction. The highest recall score being lobster roll instead of pad thai and peking duck is surprising.

Referring to the f1 scores above, the performance of all foods does not look consistent at all. Only lobster roll, pad thai and peking duck seems to be able to be predicted well by model 2.

### Best Model Verdict

I think I will choose model 1 as my best model. Despite having a higher testing accuracy in model 2 over model 1, comparing results with confusion matrix and classification report enforces my decision. I would feel very disturbed to use model 2 as best model because of its glaring weakness to not predict 1 of the food classes correctly at all. The accuracy in the classification report for model 1 is also higher than model 2, which indicates model 1 display more stability in prediction, showing less biasness for certain food. With that I saved model 1 as my best model for this project.

Graphical user interface, text, application

Description automatically generated

# Application of Best Model on Real Life Image

To apply model on real life images, it can be done easily in a few steps.

Graphical user interface

Description automatically generated with medium confidence

Firstly, there needs to be a model that is already trained. Since I had the best model saved beforehand as a h5 file, I can load and reconstructed the model along with its parameters’ weight.

Text

Description automatically generated

Next, I need to produce a list of possible classification output, which would be the 10 food I am assigned to classify. This list will be used as reference for what output there are with my model.

Text

Description automatically generated

Thirdly, I will need to define some functions to preprocess the real-life images for the model to read and another to predict.

Graphical user interface, text, application

Description automatically generated

Lastly, I can begin applying my model by calling the function to preprocess the image I want to feed in and call for prediction. The image above is an example of the application, and it would return a set of probabilities that the image could across the 10 different food classes.

### Chosen Images

As model 1 is chosen to be best model, I feel that I should pick food images that would be hard for my model to truly put it to the test here. My model is bad at classifying carrot cake and lobster hence I will be using those. As for the last image, I will use a French toast image that might come off as garlic bread. In fact, the first time I saw the image, I thought it was a garlic bread instead of French toast. I think my selection of these 3 types of images would be a good proving ground to test my model.

### Results – Carrot Cake

A piece of cake on a plate

Description automatically generated with medium confidence

Starting with carrot cake image, my model was able to correctly predict with confidence it is close to 100% a carrot cake. It is great that the model was able to identify key features of a carrot cake and I am very happy with this outcome. The probability of the image being other food was also very low.

### Results – French Toast

A picture containing text, food, screenshot

Description automatically generated

This is the test that I looked forward to the most, I had initially felt that differentiating a French toast and garlic bread will be a big challenge for my model. The result of the prediction is 75.4% French toast, 23% garlic bread and the remaining probability is distributed among the other food. The model has accurately identified and classified this image as a French toast, it is a great success in overcoming the big obstacle.

### Results – Lobster Roll

A plate of food

Description automatically generated with low confidence

Finally, the last image of a lobster roll. The model has predicted the image to be a confident 98.8% lobster roll. This is another successful prediction to close out this project.

# Summary

Concluding this project, I have stick through base approach that I intended to make. I have created to different models, one that is build from scratch and the other utilizing pretrained models. While built from scratch model was design with a pretrained model design as inspiration, I was able to develop a model achieving 73.65% training accuracy, 70.05% validation accuracy and 71% testing accuracy after 60 training epochs. As for the pretrained model, it has achieved 75.05% training accuracy, 70.9% validation accuracy and 74.2% testing accuracy after 20 training epochs.

While the pretrained model boast a better testing accuracy, the flaw in being unable to correctly predict a food type from looking at the confusion matrix and classification report. The flaw has discouraged me from using model 2 over model 1 as best model. Hence, model 1 was chosen as best model and it delivered wonderful results by applying it on real life images. Model 1 was able to accurately predict 2 of the food types it had struggle in the testing evaluation prediction. Model 1 has also overcome the challenge of differentiating a French toast and garlic bread, which is something I thought be a challenge for models.

Reviewing back the process of this project, I certainly feel that I am lacking in experience of developing model networks. The approach I took to enhance my models could have been more detailed. All these problems are something I would like to work on to improvement my models. I intend on honing my skills after project and over the next few years. I think I could also review techniques of Kaggle competition winner’s solution and try to implement it in the future when I give this project another attempt.

# References

Ruiz, P. (11 October, 2018). *Understanding and visualizing DenseNets*. Retrieved from https://towardsdatascience.com: https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a