

# School of InfoComm Technology

**Deep Learning Assignment**

Diploma in DS / IT

Oct 2023 Semester

**ASSIGNMENT 1**

(30% of DL Module)

**Submission Deadline:**

**Presentation: 10th Dec 2023 11:59PM**

**Report and Code: 10th Dec 2023 11:59PM**

|  |  |  |
| --- | --- | --- |
| **Tutorial Group** | **:** | **T01 / T02** |
| **Student Name** | **:** | Markell Wong |
| **Student Number** | **:** | S10242300 |

**Penalty for late submission:**

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

**NO** submission will be accepted after 17th Dec 2022, 11:59PM.

Contents

[1 Overview 3](#_Toc153029316)

[2 Data Preprocessing and Data Loading 4](#_Toc153029317)

[3 Developing the Image Classification Models 6](#_Toc153029318)

[3.1 Baseline Model 7](#_Toc153029319)

[3.2 Final Scratch Model: 15](#_Toc153029320)

[3.3 Pre-trained Models 15](#_Toc153029321)

[4 Evaluate models using Test images 19](#_Toc153029322)

[5 Use the Best Model to perform classification 21](#_Toc153029323)

[6 Summary 23](#_Toc153029324)

[6.1 Further Improvements 23](#_Toc153029325)

# Overview

Problem: A social media platform catered to food enthusiasts is looking to elevate user engagement and enhance search functionalities by implementing a robust image recognition model. The company has hired machine learning engineers to create classification models for their platform.

Objective: Build an image classification model to recognize and classify 10 different types of food commonly shared on the platform.

Approach:

For this project, I have created two kinds of Convolutional Neural Network models for multiclass image classification. The first is trained from scratch using Conv2D and fully connected dense layers. The second is a model trained using transfer learning, which is importing models built by professionals and companies for competitions, also known as pre-trained models and fine-tuning them to best suit my dataset. The dataset used is Kaggle’s food images.

I will use Accuracy and Loss when training the model, where I will check the scores for each epoch to see how I can tune the model. Additional metrics such as precision, recall and f1 score will be used at the end when I perform evaluations on the test dataset. The final evaluation metrics were not used during training as these are global metrics which are used to evaluate the entire dataset rather than monitor the training process at the end of each epoch.

The loss function used for most of the first few models would be using categorical\_crossentropy as we are predicting multiple, mutually exclusive classes, which uses softmax activation in the output layer. After some experimentation during the final scratch model, I changed the loss function to sparse\_categorical\_crossentropy and continued to use it for the pre-trained models.

To train and evaluate the model, I will split the data into train, validation, and test. Training will be done with train and validation and evaluation will be done with test.

The project will follow the universal machine learning workflow, where I will first start with a baseline model, scale up the model until overfitting occurs, and then regularize the model accordingly. Between the final model built from scratch and the pre-trained, I will then compare and evaluate the two to choose the best model for my dataset.

For the project, I used a reaction approach to tune my models. I would first see how the model performs with the current set of variables to see whether it overfits or underfits etc and then react accordingly by adding new variables or tuning existing ones to help with the problem. I also performed some supplementary experimentation with different loss functions and dropout types to see if my model would improve.

The report is structured a bit like a journal, where I document my observations and reasoning for my actions. I will explain why I decided to implement certain methods, explore the methods in more depth and the outcome.

# Data Preprocessing and Data Loading

First, I downloaded the image dataset from Kaggle. The dataset contained images of food, with 101 types of food. There are 1000 images for each type of food. Since I had to classify 10 different types of food, I had 10 classes, totaling around 10000 images.

A computer screen shot of a program code

Description automatically generated

Figure 1: Directories

As seen in figure 1, I created the base directory and created train, validation and test directories, joined with the base directory.

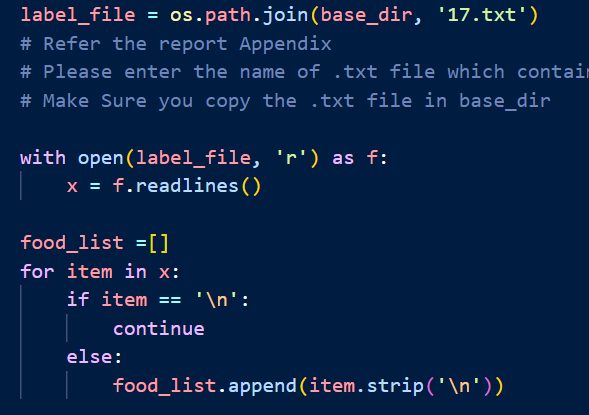


Figure 2: Reading food text file

Next, I ran the function in figure 2 which joined the text file I was assigned containing the list of foods I had to classify, opened it, and appended the contents into a list.

The foods I had to classify are as follows: beef\_carpaccio, crab\_cakes, escargots, gnocchi, lasagna, macaroni\_and\_cheese, samosa, spring\_rolls, strawberry\_shortcake and sushi.

A computer screen shot of a program code

Description automatically generated

Figure 3: Appending images to directories

Next, I had to split each class’s 1000 images into 750 train, 200 validation and 50 test. Using the directories I had created earlier, I ran the function in figure 3 which created a directory path for each food category in the training directory (‘train\_dir’). Next, it reads the list of images in the directory corresponding to the current food category and selects the first 750 images from that list. It then creates a destination path, and the images were copied from the source to the destination path. This function was run for the 200 and 50 validation and test images as well.

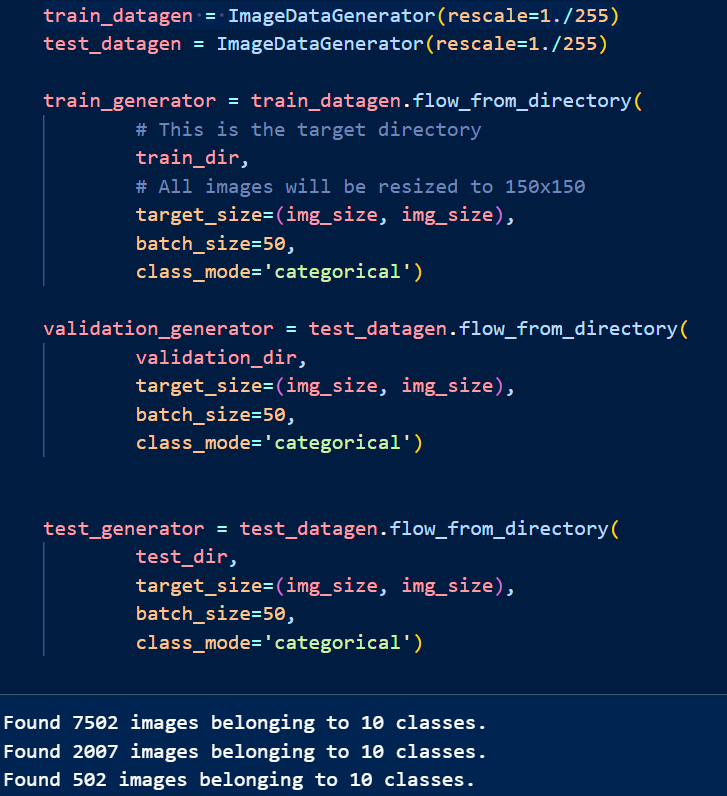


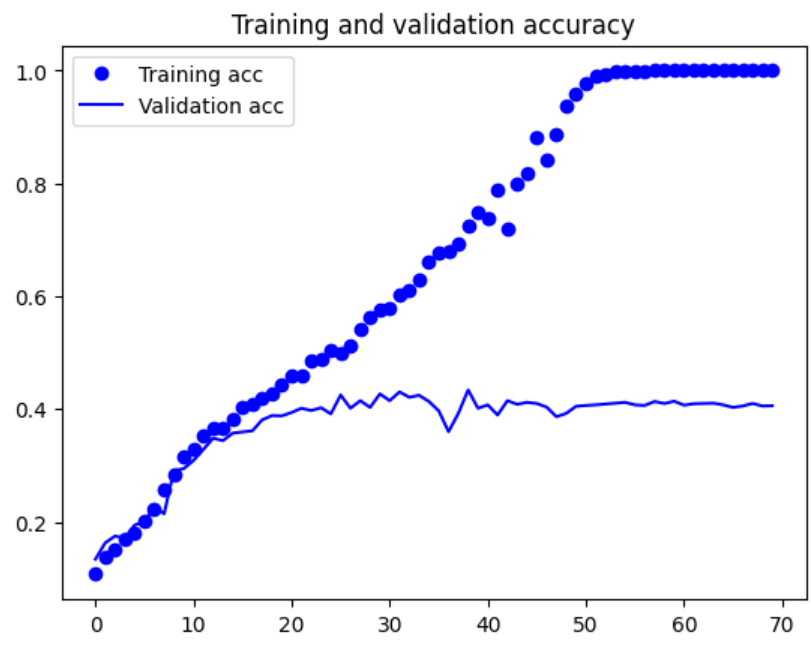
Figure 4: ImageDataGenerator

Then, I loaded the data into my assignment file. I then ran the images through the Image Data Generator in figure 4 where it rescales the pixel value of the image to a range of 0 and 1 by dividing each pixel value by 255. This ensures each pixel value is within the same scale and prevents larger pixel values from dominating the learning process. I also defined the image size as 150 x 150.

For the pre-trained models, each model has a different preprocessing input. For example, MobileNet requires the input pixel values to be scaled between -1 and 1, which is different from the one defined for the scratch model. Hence, for each Image Generator for each pre-trained model, I imported the preprocess input module unique to that model before implementing the Image Generator. I also changed the feature shape according to the model’s final feature map.

# Developing the Image Classification Models

In this report, I will be using the terms underfitting and overfitting a lot. Here, I want to clearly define what those terms mean.

 A graph of a graph

Description automatically generated with medium confidence

Figure 5: Overfitting training and validation curves

Overfitting is when a model overtrains on the train data, making it too accustomed to training data and causing it to misclassify new/unseen examples. When referring to metrics, Figure 5 is a clear example of model overfitting. There is a large difference between training and validation curves with the model having close to perfect training scores while having a poor validation score. We can also see it when the validation loss starts to increase but the training loss continues to decrease.

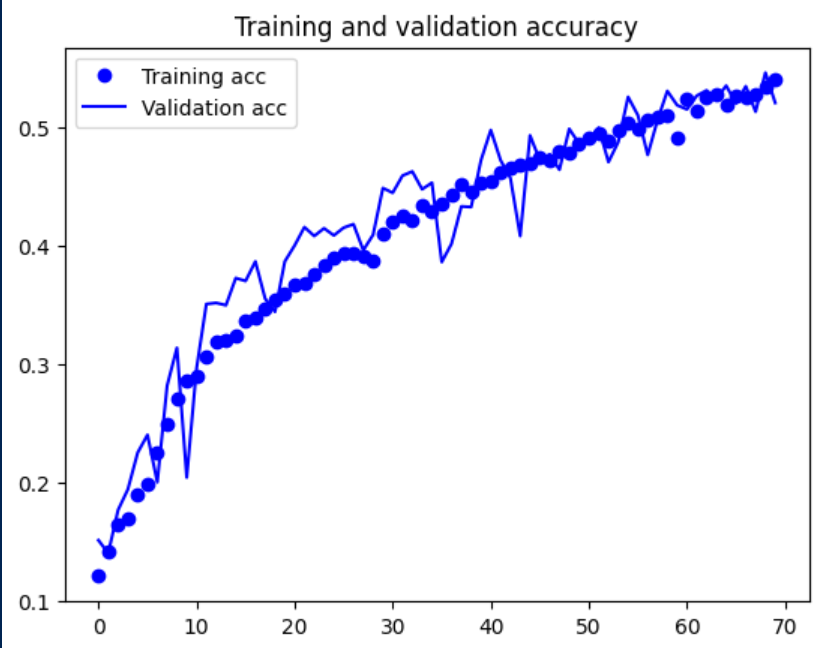
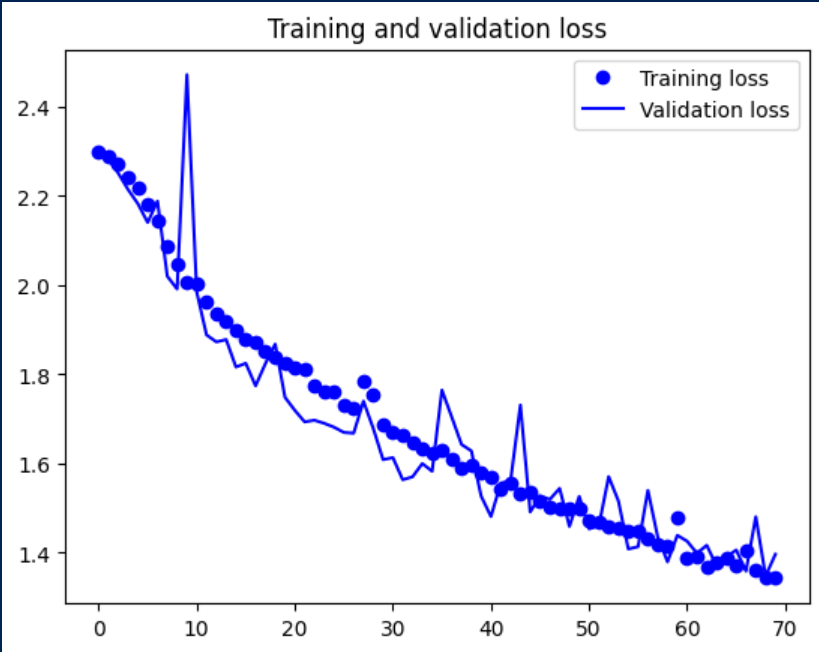
 

Figure 6: Underfitting training and validation curves

Underfitting is when the model is unable to learn the patterns in the data properly. It is characterized by low training and validation accuracies and high loss as seen in figure 6.

A model of best fit would be when we cut the model training before overfitting happens. Using figure 5 as an example, we would stop the model training at 30 epochs, which is before the loss starts to increase.

## Baseline Model

A computer screen shot of a program code

Description automatically generated

Figure 7: Baseline model

The baseline model is a simple model with 4 Conv2D layers and 2 dense layers. The layers have an increasing number of nodes (16,32,64 etc). This is to allow the model to capture simpler features in the earlier layers and then more complex features in the deeper layers.

The output dense layers have 10 nodes due to the 10 classes I am predicting, and I used the softmax function. The softmax function is an activation function that scales numbers into probabilities of each possible outcome summing up to one for all classes. For the Conv2D layers, I used Rectified Linear Unit (ReLU) which helps introduce non-linearity to the model and solves issues such as vanishing gradient. Non-linearity is important as it allows the model to capture more complex relationships and capture intricate patterns in the images.

Maxpooling layers were added to reduce the dimension of the feature maps before passing on to the next layer, it effectively retains the most important and relevant features while reducing the computational load in subsequent layers. Flatten layers are also added after the Conv2D layers and before the Dense layers, which helps to reshape the input tensor into a 1D tensor before feeding it into the fully connected layers (Dense) as FC layers require 1D inputs for classification tasks.

I set the learning rate to a default rate of 0.01 and used the optimizer Stochastic Gradient Descent (SGD). Categorical cross-entropy was used as the loss function. The baseline model had a total of 463,338 trainable parameters.

A graph of a graph

Description automatically generatedA graph of a graph

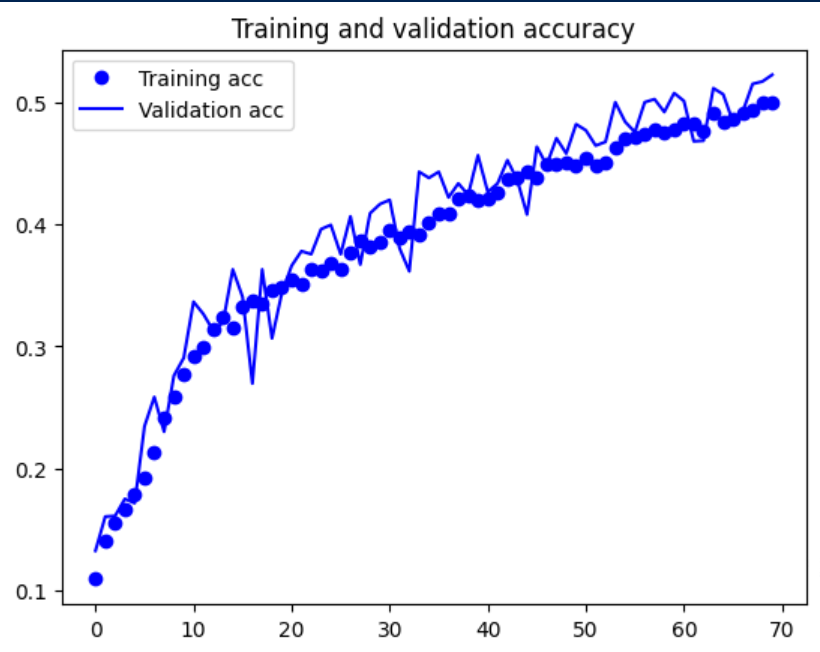
Description automatically generated with medium confidence

Figure 8: Baseline model performance

After running the baseline model for 70 epochs, I immediately observed clear overfitting with the training and validation curves deviating from each other and the validation loss starting to increase rapidly after the 30th epoch. To counter the overfitting, I implemented data augmentation.

**Data Augmentation**

Data Augmentation is a technique of artificially increasing the training dataset by creating modified copies of existing data. This helps introduce noise into the data, allowing the model to generalize better by exposing it to more variation which can help reduce overfitting.

A graph of training and validation

Description automatically generated

Figure 9: After data augmentation

After introducing data augmentation, the model started to underfit. While the validation accuracy improved to 0.5225, the training scores worsened considerably. To counter the underfitting, I increased the model capacity to allow the model to capture more complex patterns.

I tried two ways of increasing model capacity. First, I doubled the number of nodes for every layer and second, I introduced an additional layer of 512 nodes before the flatten layer. By increasing the number of nodes and layers, I aimed to increase the number of trainable parameters in the model, increasing model capacity.

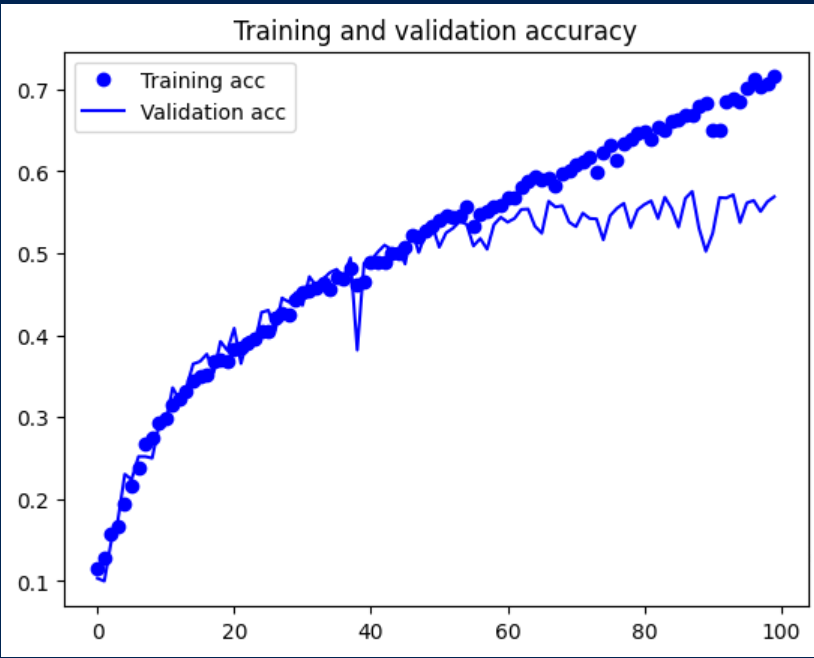
A screen shot of a computer program

Description automatically generated

Figure 10: Removing Maxpooling layer

Subsequently, I did not add MaxPooling layers for Conv2D layer I included as seen in figure 10. This is because I wanted to increase the number of trainable parameters. By adding a Maxpooling, the dimensions of the feature map would be reduced, and the number of trainable parameters would drop. After increasing model complexity, I had 11,011,402 trainable parameters.

However, even after increasing the model complexity by close to 10 million parameters, the model was still underfitting. It is possible the data augmentation could have introduced too much noise into the train data for the model to be able to capture patterns while training. Hence, I removed some data augmentation parameters to see if it helps the underfitting. I also increased the number of epochs to 100.

 A graph of training and validation

Description automatically generated

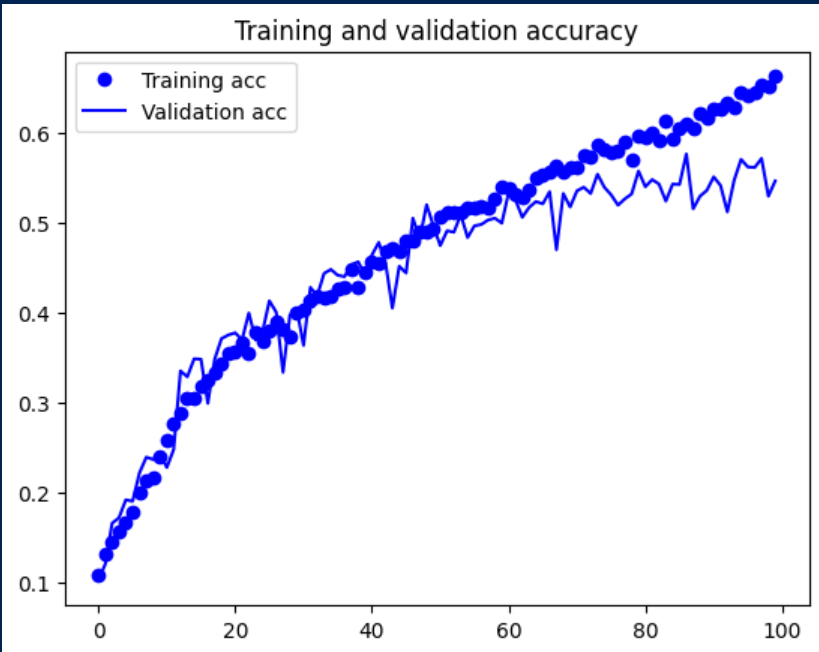
Figure 11: After removing height\_shift\_range

First, I tried removing the parameters shear\_range and zoom\_range, but my model continued underfitting. After removing height\_shift\_range as well my model started to overfit as seen in figure 11.

**Dropout Layers**

To counter the overfit, I decided to introduce a new method, dropout layers. Dropout layers randomly deactivate neurons in the network based on the rate defined by setting their outputs to zero, making them unusable for forward and backward propagation. This helps reduce the network’s reliance on specific neurons, forcing the model to learn more robust and generalized features. Dropout layers are a form of regularization and help reduce overfitting.

I added a dropout layer of 0.5 before the output softmax layer. I eventually tweaked the dropout rate to 0.2, deactivating 20% of the neurons instead as 0.5 had caused underfitting. 0.2 resulted in slight overfitting.

A graph of training and validation

Description automatically generated

Figure 12: After 0.2 dropout layer

**Batch Normalization**

Next, I introduced batch normalization. For each mini-batch during training, batch normalization standardizes the input values by subtracting the mean and dividing by the standard deviation. This ensures that the input values are now on the same scale, allowing the gradient descent to proceed smoothly down to the minimum of the loss curve, which allows for faster convergence.

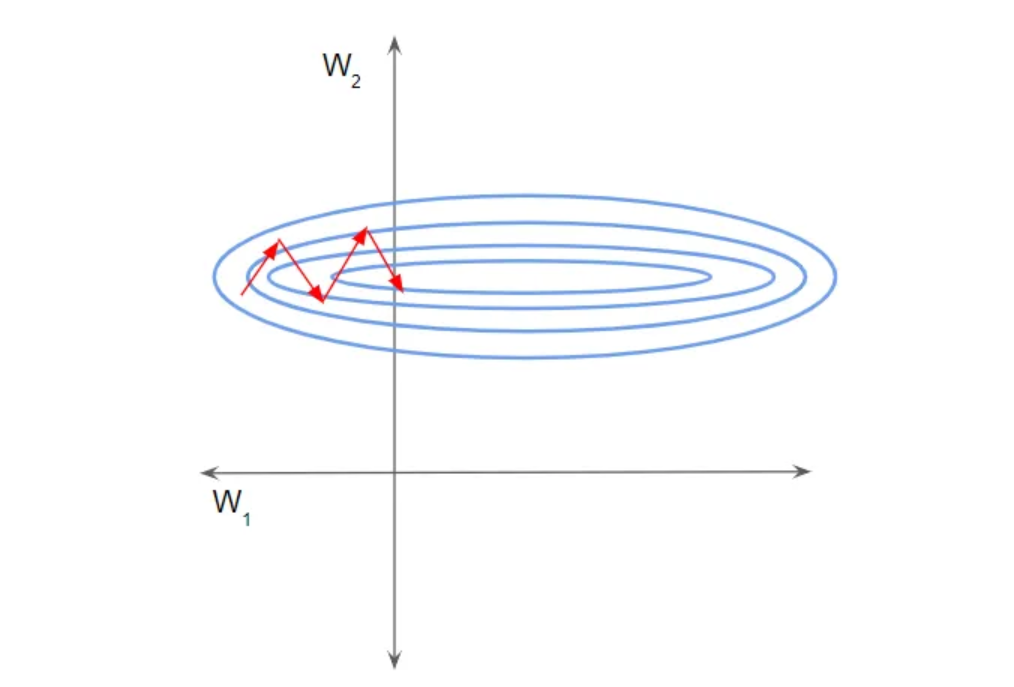


Figure 12: Gradient Descent trajectory when input is on different scales

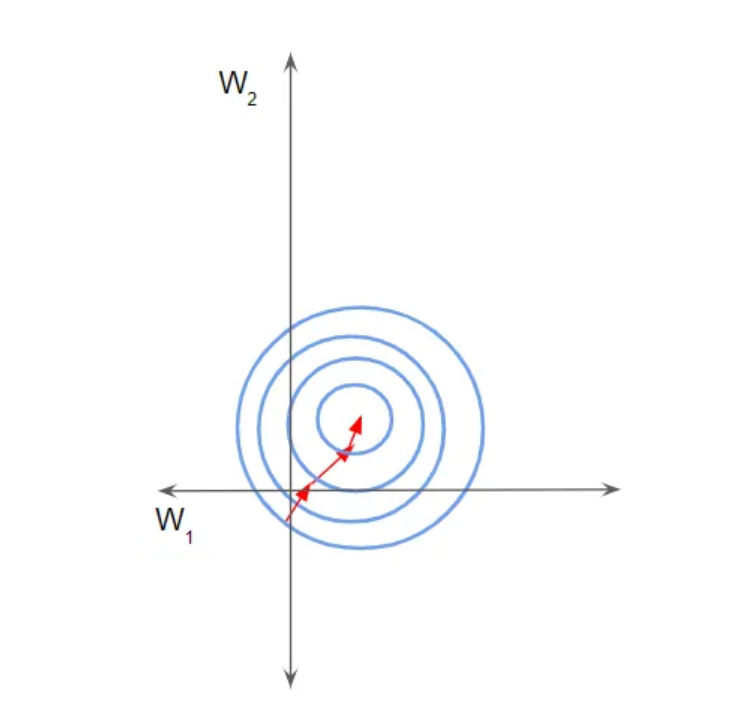
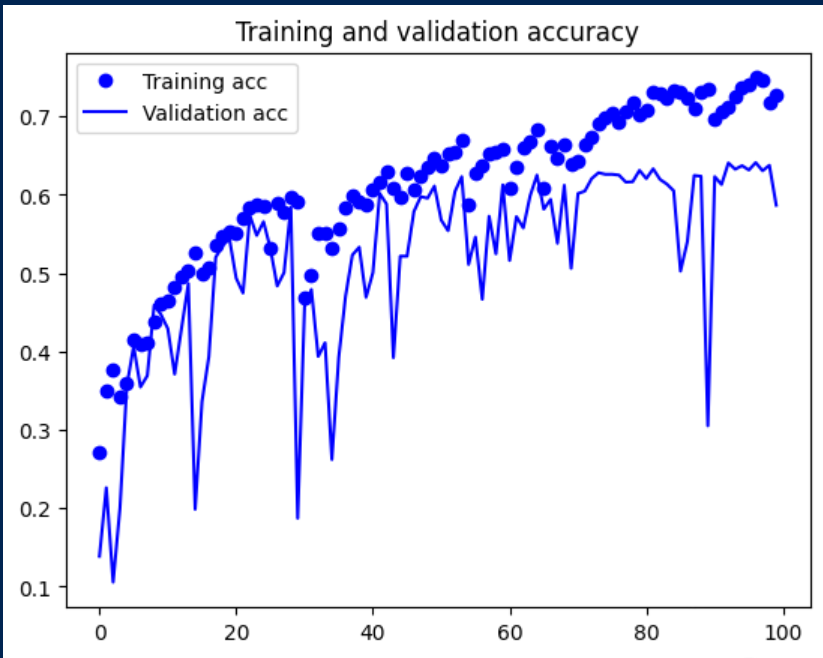


Figure 13: Gradient Descent trajectory when input is on same scale

Batch normalization offers some regularization effect by adding noise to each mini-batch caused by the normalization of mini-batch statistics. This helps reduce overfitting as well as quickens convergence.

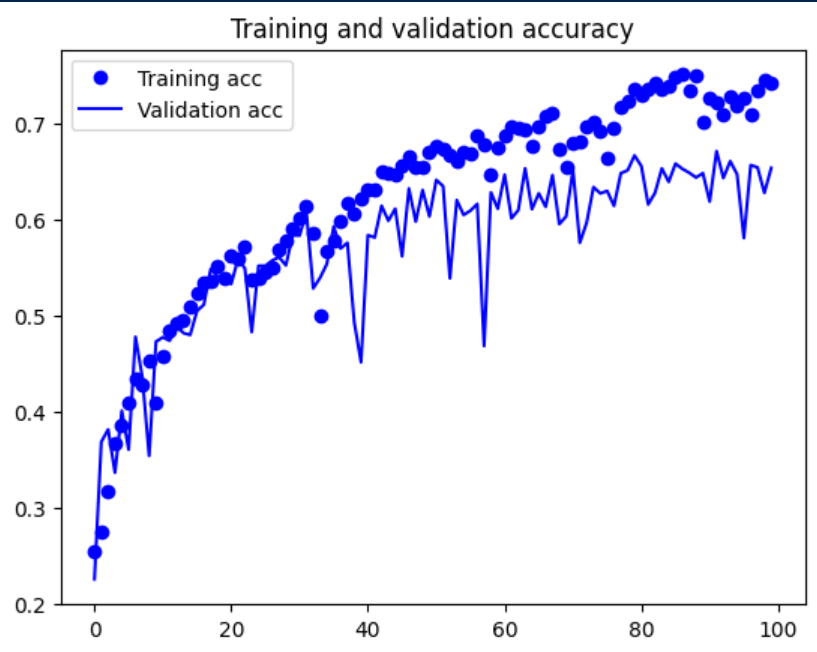
 A graph of training and validation

Description automatically generated

Figure 14: After batch normalization

After adding batch normalization, overfitting was reduced and validation accuracy was able to improve from 0.547 to around 0.63, however, the validation scores started to fluctuate erratically. The fluctuations might be due to differences in mini-batch statistics calculated during training and validation. Batch normalization uses training statistics to normalize the data which may not resemble the statistics in validation. This can lead to fluctuations in validation scores.

To counter that, I gradually reduced the momentum in the batch normalization layers from the default of 0.99 to 0.8. The momentum parameters control how quickly the moving average of mean and variance are updated during training. Reducing the momentum means that the current mini-batch’s statistics contribute more to the updated moving averages and less to the existing moving averages. This helps stabilize the training process and reduce fluctuations in validation scores.

A graph of training and validation

Description automatically generated

Figure 15: Reduced Momentum

After reducing momentum to 0.8, the fluctuations decreased a lot, and the model was more stable. However, there was still quite a bit of fluctuation, so I tried increasing the batch size as well.

**Batch Size**

For the next model, I increased the batch size from 50 to 64. Larger batch sizes provide a more stable gradient estimate since average gradients are computed over a larger number of samples. This leads to smoother updates of parameters, reducing noise and fluctuations in the training process. The fluctuations were now much lesser for training scores and lesser for validation scores but the model was also overfitted.

To counter the overfitting, I increased regularization by increasing the dropout rate back to 0.5 and added an additional dropout layer with a 0.5 rate. This led to lesser overfitting, however, towards the last few epochs, fluctuations in validation scores increased as seen below in figure 16. To counter this, I tried different optimizers such as RMSProp and Adam as they have learning rate schedulers.

A graph of a training and validation accuracy

Description automatically generatedA graph of training and validation

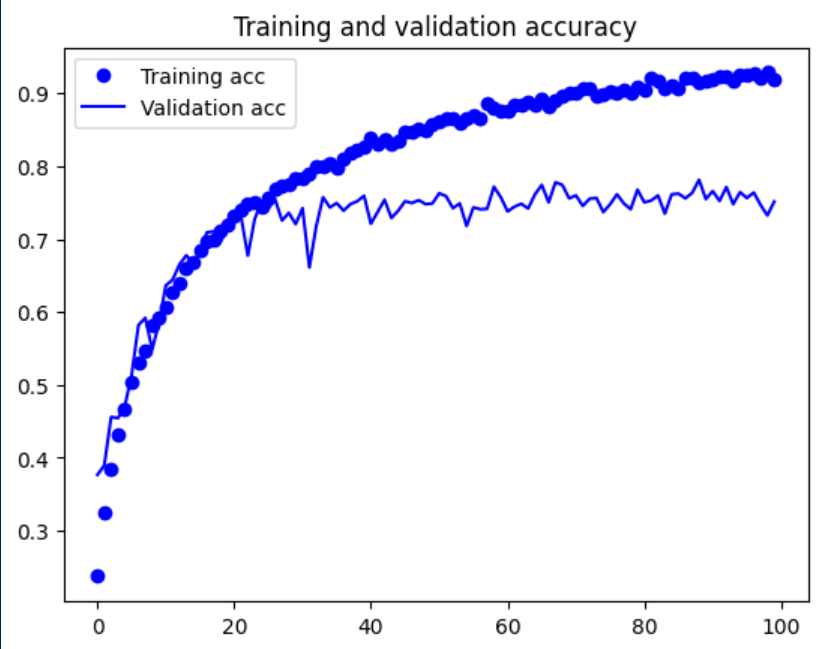
Description automatically generated

Figure 16: After increasing batch size and dropout

**Optimizers**

Before I could train using the optimizers, I encountered a ResourceExhaustedError problem. My GPU ran into a memory problem. I first tried to limit the memory for each model I ran but it didn't work. In the end, I reduced the batch size from 64 to 32 which helped reduce the computational load required to run my models.

RMSProp and Adam optimizers can adaptively adjust the learning rate for each parameter and enable the usage of larger learning rates. While in SGD, the learning rate remains constant. The adaptive learning rate can help the model converge with varying gradients, smoothing the training process.

A graph of training and validation

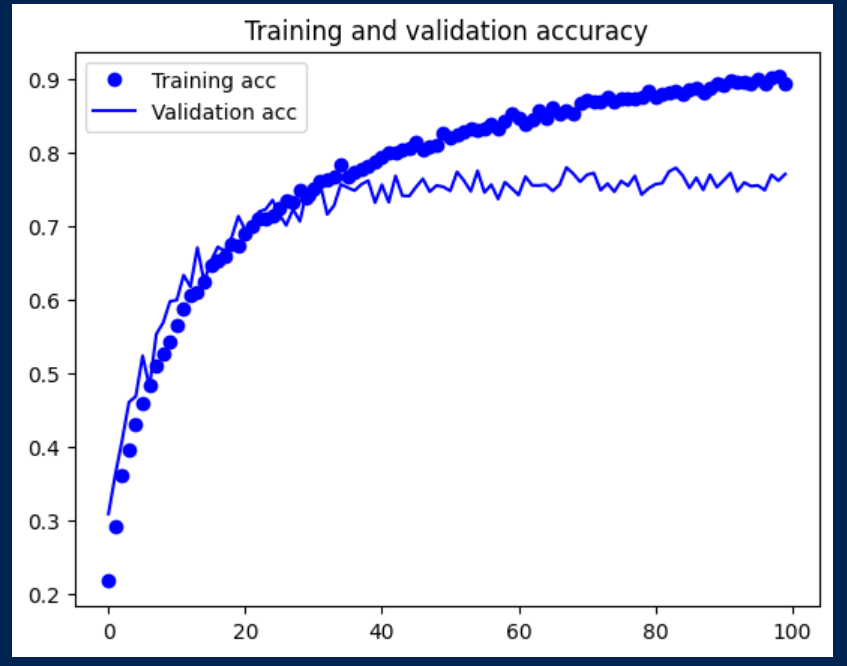
Description automatically generated

Figure 17: With Adam Optimizer

With Adam optimizers, I was able to reach a much better 0.75 validation accuracy and 1.0527 loss. RMSProp also saw a substantial improvement over SGD with around 0.72 accuracy and 1.2 loss but not as good as Adam. Both optimizers were able to reduce validation fluctuations effectively with Adam optimizer performing better. However, both models started to overfit.

**Adjusting regularization**

Here, I added more regularization. Referring back to data augmentation, I added back the removed data augmentation parameters which helped reduce the overfitting. I also noticed that the training and validation scores were quite good and that loss only started to increase around the 55th epoch

 A graph of training and validation

Description automatically generated

Figure 18: Model 6.2 where data augmentation parameters were added back

Later on, I tried more regularization techniques such as L2 and L1 Weight Regularizers to see if I could further reduce overfitting. L1 regularization forces the weights of uninformative features to be zero by subtracting a small amount from the weight at each iteration, eventually making the weight zero. L2 regularization forces weights toward zero but does not make them exactly zero. The weight regularizers handle two common reasons that cause the model to overfit, the total number of features (solved by L1) and the weights of features (solved by L2).

After trying the two weight regularizers separately with a lambda value of 0.001, both regularizers caused the model to underfit, with L1 causing the model to severely underfit, decreasing training and validation accuracy to around 0.55 and loss to around 2.5. I tried to reduce the implementation of L2 regularizer layers from 5 to 4 layers (I had previously added L2 weight regularizers to every Conv2D layer) to reduce the regularization effect. However, the model still seemed to underfit.

In the end, I decided to use Model 6.2 in Figure 18 and since loss increased after 55 epochs, I cut the number of epochs to 55.

**Additional Experimentation**

Next, I experimented with a different loss function, Sparse Categorical Cross-entropy and another kind of dropout Spatial Dropout 2D.

Sparse Categorical Cross-entropy is essentially the same as categorical cross-entropy but inside of one-hot encoding labels (eg: [1,0,0], [0,1,0]), the labels are instead just replaced with integers ([1],[2]). The advantage of using sparse is it saves time in memory as well as computation as it simply uses a single integer for a class, rather than a whole vector. Changing to sparse did not change my model’s scores, but I decided to use it as it saves computation power.

Spatial Dropout 2D performs the same function as regular dropout layers but it drops entire 2D feature maps instead of individual neurons. It helps with pixels within feature maps that are strongly correlated and promotes independence between feature maps. Spatial dropout is supposedly more suitable for CNN models.

Because spatial dropout cannot be used after the flatten layer (flatten changes input into 1D tensor), I moved the spatial dropout layers to the first 3 Conv2d layers. I ran the model for 55 epochs, however, it was underfitting so I pushed the model for another 45 epochs. After around 10 epochs (65 epochs total) the model started to overfit. When looking at the model scores, there wasn't much change compared to regular dropout so I continued using regular dropout.

## Final Scratch Model:

A screen shot of a computer program

Description automatically generated

Figure 19: Final Scratch Model Architecture

A graph of a training and validation accuracy

Description automatically generatedA graph of training loss

Description automatically generated

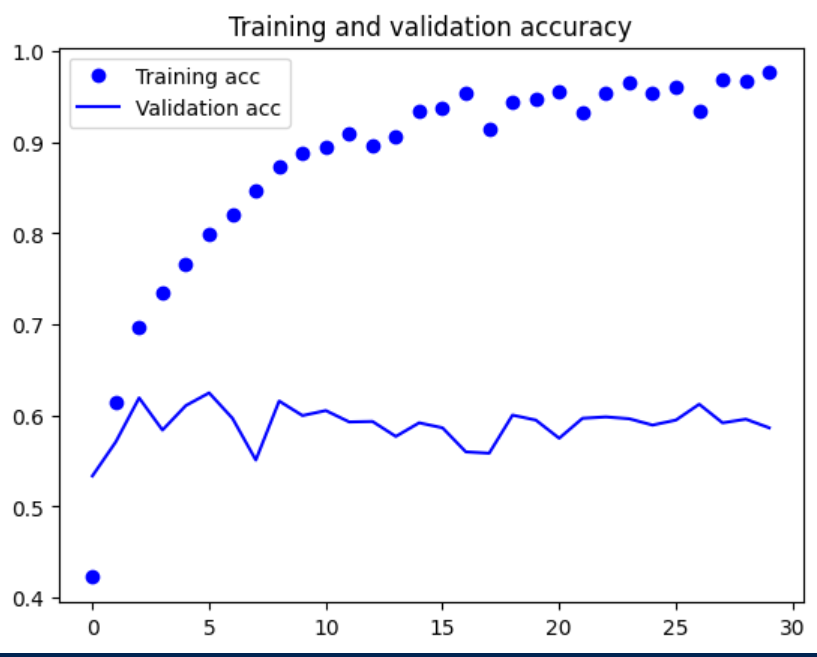
Figure 20: Final Scratch Model performance

Performance: loss: 0.4730 - accuracy: 0.8394 - val\_loss: 0.9080 - val\_accuracy: 0.7424

## Pre-trained Models

For the pre-trained models, I tried the InceptionV3 model. InceptionV3 assists in image analysis and object detection and got its start as a module for GoogLeNet. Its design was intended to allow deeper networks while keeping the number of parameters from growing too large. The final feature map for InceptionV3 is (3,3,2048).

First, I did feature extraction without data augmentation. I created the base model from InceptionV3, then added a simple classification head of one dense layer of 256 nodes and the output layer with 10 nodes.

A graph of a graph with blue dots

Description automatically generated

Figure 21: Without data augmentation

When training without data augmentation, my model overfitted very quickly with the loss increasing greatly from the first few epochs. My validation accuracy was hovering around 0.6. To counter the overfitting, I implemented data augmentation.

A graph with blue lines and white text

Description automatically generatedA graph of training and validation loss

Description automatically generated

Figure 22: With data augmentation

For data augmentation, I defined the InceptionV3 preprocess\_input and froze the base model which prevents the weights in the layers from being updated during training. After data augmentation, overfitting was reduced and my validation accuracy had improved to around 0.68 but training accuracy had also dropped to around 0.7.

To improve accuracy, I tried fine-tuning the base InceptionV3 model by unfreezing some layers. I unfroze the last few layers as it allows the model to be tuned specifically towards my dataset, allowing it to specialize and improve performance. The earlier layers contain more general features whilst the last layers contain more specific.

I unfroze at the ‘mixed9’ layer, resulting in 10.7 million out of 26.5 million trainable parameters. I also ensured to reduce the learning rate after unfreezing as I am training a much larger model and want to readapt the pre-trained weights. Not reducing the learning rate can lead to severe overfitting.

A graph with blue dots

Description automatically generatedA graph with blue lines

Description automatically generated

Figure 23: After fine-tuning

Upon training, the validation accuracy improved to around 0.75. However, the model faces overfitting again. Next, I added dropout layers to the classification head with a rate of 0.5. The validation accuracy improved very slightly around 0.76. The model also was starting to overfit around the last 10 epochs.

I also tried using GlobalAveragePooling layers to counter overfitting. Global average pooling is used to replace fully connected layers. It does that by generating one feature map for each corresponding class of the classification task. It takes the average of each feature map and the result is fed directly into the softmax layer. As there are no parameters to optimize in the GlobalAveragePooling layer, this reduces the model complexity, reducing overfitting.

After trying GlobalAveragePooling, it seems the method did very little to stop the overfitting as there is still a big difference in training and validation scores. Hence, I decided to continue using dropout layers instead.

From here on, I tried to tune the model hyperparameters such as increasing and decreasing batch size and learning rate as well as trying different optimizers like RMSProp and increasing model complexity by increasing dense layers. However, the tuning did not do much in terms of model performance and only helped determine whether the model would overfit earlier or later. The only change I implemented was to reduce the batch size from 20 to 10 as it reduced the overfitting and the model only started to overfit towards the last few epochs as seen below.

A graph of a training and validation accuracy

Description automatically generatedA graph with blue dots

Description automatically generated

Figure 24: Batch size 10

Hence, for my final InceptionV3 model, I had a validation loss of around 0.9 and a validation accuracy of around 0.76.

Next, I tried another pre-trained model, MobileNet. MobileNet is a computer vision model open-sourced by Google and designed for training classifiers. For MobileNet, I performed the same procedure as InceptionV3 all the way up to fine-tuning.

The MobileNet model performance without data augmentation was quite good at a validation accuracy of 0.70 and was overfitting. After data augmentation, the overfitting was reduced considerably, and the validation accuracy was 0.76. For fine-tuning, I unfreezed at the ‘conv\_dw\_12\_relu’ layer resulting in 5.7 million out of 7.4 million trainable parameters. After fine-tuning, the validation accuracy doesn’t improve and the model overfits.

InceptionV3: val\_loss: 0.9089 val\_accuracy: 0.7785

MobileNet: val\_loss: 1.0685 val\_accuracy: 0.7628

Based on the training performance, I will use the final Inception V3 model as my pre-trained

# Evaluate models using Test images

For evaluation with test images, I added sklearn classification\_report, adding the metrics precision, recall, f1\_score and support. To better understand the metrics, I will use a simple binary confusion matrix to explain.

A diagram of positive negative and negative

Description automatically generated

Figure 25: Binary Confusion Matrix

Positive: Data classfied as a member of the class the classifer is trying to identify (model looking for 1s would classify photos with 1 as positive if correct)

Negative: Data classified as not a member of the class the classifier is trying to identify (model looking for 1s would classify photos with 0 as negative)

True positives: data labeled as positive that is actually positive

False positives: data labeled as positive that is actually negative

True negatives: data labeled as negative that is actually negative

False Negatives: data labeled as negative that is actually positive

Precision is the ability of a classification model to return positive predictions that are correct (true positives). Recall is the ability of the model to identify how many of the positive cases the classifier correctly predicted, over all the positive cases in the data. F1 score is a single metric that weights the two metrics (Precision and Recall) in a balanced way with harmonic mean. Support is just the number of occurrences of each class.

These metrics help us to understand what classes the model performs better on and the classes that it doesn’t, providing us with a better understanding of the model.

First, I evaluated the final scratch model on the test dataset. The model resulted in a loss of 1.0235 and a test accuracy of 0.7146.

A screenshot of a computer screen

Description automatically generated

Figure 26: Classification Report of Scratch Model

When looking at the classification report, we can see that the scratch model performs the worst on crab cakes with a f1-score of 0.60 while it performs the best on beef carpaccio and strawberry shortcake at a f1-score of 0.80.

Next, I evaluated the final pre-trained model. The pre-trained model resulted in a loss of 1.0479 and a test accuracy of 0.7300.

A screen shot of a blue screen

Description automatically generated

Figure 27: Classification Report of Pre-trained Model

When looking at the classification report, we can see that the pre-trained model performed the worst on crab cakes at a f1\_score of 0.62 and the best on strawberry shortcake with a f1\_score of 0.81.

While the pre-trained model has a slightly worse loss score than the scratch, it's test accuracy and f1\_scores for worst and best classes were better than the scratch model. When we look at all the classes in terms of f1\_score, the pre-trained model performs better in 7 classes while the scratch model only performs better in 1 class, the two other classes are equal. Both models give sub-optimal precision for identifying Gnocchi.

As seen from the performance, I will use the pre-trained model.

# Use the Best Model to perform classification

For the pre-trained model, I decided to use images of strawberry shortcakes and gnocchi. Strawberry shortcake was the class with the highest f1 score while gnocchi was one of the poorer performing classes. This will allow us to assess the model's strengths and weaknesses in identifying food types.

The process I used to apply the model to real life images:

A computer screen shot of code

Description automatically generated

Figure 28: Data preprocessing and prediction functions

First, I load a text file list with the class labels of my dataset. Next, I ran two functions: image\_process() and prediction(). The image\_process function loads the image from the file path defined in the input and resizes the image to a target size of 150 x 150 (this was defined at the start). The loaded image is then converted to an array and normalized by dividing by 255 to scale the pixel values between 0 and 1. The function then returns the image as an array.

The prediction function uses the pre-trained model to predict the class probabilities for the image array reshaped to the required format for prediction. The predicted probabilities are then stored in a dataframe with columns of the class labels earlier loaded. The function then identifies the predicted class by finding the class label with the highest probability.

I tested the model on 3 images of strawberry shortcake, all of which the model was able to predict with high confidence (0.99, 0.71 and 0.99). I also tested the model on 3 images of gnocchi, 2 of which the model was able to predict with high confidence (0.93 and 0.99).

A screenshot of a computer screen

Description automatically generated

Figure 29: Prediction outcome of gnocchi picture

For this image, the model predicted the gnocchi as crab cakes (to be fair it does look like crab cakes) with a low confidence probability of 0.343. Gnocchi and sushi were also the next two classes at 0.238 and 0.172 respectively. A possible reason the model did not predict correctly could be that the model could not capture the complex patterns or variations the image introduces, for example, the bright red sauce on top. It is also possible that the model has not seen enough diverse examples during training to confidently differentiate between the classes, leading to uncertainty.

# Summary

In summary, out of the different models built, the pre-trained InceptionV3 model performed the best at a test loss of 1.0479 and a test accuracy of 0.7300. Two different types of models were built and tuned, a model from scratch and models from transfer learning (pre-trained). When training both types, I ensured to first overfit the model before starting regularization and tuning.

Different methods and tuning were used during this project to counter both under and overfitting. This included regularization methods such as dropout, weight regularizers, data augmentation, and batch normalization. Regularization methods were also further tuned when the regularization effect was too large. I also commonly encountered fluctuations in validation scores which were solved by tuning hyperparameters and using different optimizers.

After building the models, I evaluated them using metrics such as accuracy, loss and f1-score to find the best model. InceptionV3 was the best model and when applied to real-life images, performed quite well and was able to successfully class 5 out of 6 images.

## Further Improvements

**Data and preprocessing improvements**

For example, using larger image sizes such as 224 x 224 or 512 x 512. Larger image sizes, while more computationally expensive, allow for more information and better feature extraction which can help the model’s ability to learn complex representations. Another one would be to introduce more training data or clean the images in the dataset before training. Increasing training data helps the model generalize better to unseen images by increasing the variation of data. Cleaning the dataset includes removing irrelevant images to ensure the model learns meaningful patterns.

**Model Improvements**

We can also try more variations of hyperparameters and pre-trained models to see if the model is still able to improve. We can also try ensemble methods such as concatenation ensemble to combine predictions from multiple models, which could lead to better performance and robustness.

**References**

Image of Binary Classification Matrix: <https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826>

Batch Normalization Images: <https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739>