Victoria University of Wellington

COMP309
Project

Convolutional Neural Network for Image Classification

Author:
Luke Hartfield

Student ID:
300421352

October 21, 2020

Contents

1 Introduction

The purpose of this project is to build a convolutional neural network model (CNN) that is capable of accurately classifying sets of images. The images will be made up of three distinct classes: strawberry, cherry, and tomato. If implemented correctly, the model will be able to distinguish between the three fruit with a high level of success. To build this model, the Keras deep learning API, as part of the TensorFlow library, will be utilised. An initial multilayer perceptron model (MLP) will also be built as a baseline model in order to compare the results of the CNN model and to provide a basis for improvements.

2 Problem Investigation

2.1 Exploratory Analysis

The initial dataset is made up of 4,500 JPEG images split evenly between the three class: strawberry, cherry, and tomato, with each class having 1,500 images of their respective fruit. All images have also been scaled to the same size of 300x300. Upon investigating the data, some potential challenges do become apparent that may impact the accuracy of my model.

2.1.1 Multi-Label Images

The dataset appears to have images containing more than one of the classes, as seen in Figure 1. While the image is mostly made up of cherries, and is labelled as such, by having strawberries as well, the model may become less accurate as when the model sees new images of strawberries it may think it is an image of cherries as it has seen them in the cherries class too. These will need to be treated.

Figure 1: Example of an image with both cherries and strawberries present

2.1.2 Illumination

Another challenge is some that images have very low illumination (low light levels), which can make classification quite difficult as the effects of illumination are very significant at the pixel level. An example of low illumination from the training set can be seen in Figure 2. Some images also have the opposite, where illumination is high. This can sometimes be beneficial though as the model will become more robust to images of different light levels.

Figure 2: Example of an image low illumination

2.1.3 Occlusion

Some of the images are partly or mostly occluded (hidden) making classifying them potentially very difficult. This may have a significant impact on the accuracy of the model. An example of this can be seen in Figure 3.

Figure 3: Example of strawberries that are partially blocked from view

2.1.4 Outliers

The dataset contains some images that appear to not include any fruit, contain cartoon drawings of the fruit, or contain objects that only vaguely resemble the fruit (such as a cherry shaped handbag). This will cause the model to learn incorrect information and could consequently reduce accuracy. These will need to be removed.

2.1.5 Additional Comments

Other challenges found in the dataset include deformation: where a image has been cut in half or is misshapen, viewpoint variation: not all the fruit are shown from the front as some are from above or below, background clutter: where the background has a lot of noise, and color

variation: the fruit aren't all the same color even within the same class. This could possibly reduce accuracy, though it could also make the model more robust to new data.

2.2 Preprocessing

2.3 Remove Outliers

In order to alleviate some of the challenges found during the exploratory analysis, the data needs to be cleaned. Any images which do not contain any fruit and images that contain drawings of the fruit from their corresponding classes are to be removed. To do this, I used personal judgement. This resulted in 148 images being removed, reducing the dataset from 4,500 images to 4,352. An example of an outlier that was found in the strawberry class can be seen in Figure 4, there is clearly no strawberry in the image. This improved the accuracy of the model slightly.

Figure 4: Example of an outlier found within the strawberry class

3 Methodology

3.0.1 Data Augmentation

The ImageGenerator class from Keras is to be used to augment the data. The ImageGenerator will randomly apply transformations to batches of training images as they are loaded, making the training data more robust as it artificially creates more images. The ImageGenerator has a number of possible transformations such as brightness, rotation, flip, and zoom that can be randomly applied using values from within a user-specified range. It can also normalize the images and binarized the classes. For this model, I have set the following ImageGenerator parameters:

- Rescale = 1. / 255, this normalizes the images
- Shear Range = 0.2, randomly changes the viewing angle
- Zoom Range = 0.2, randomly changes the zoom
- Rotation Range = 0.45, randomly rotates the image

- Width & Height Shift = 0.2, randomly shifts the image horizontally or vertically
- Horizontal & Vertical Flip = True, randomly flips the image
- Brightness Range = 0.5 - 1.5, randomly changes the image brightness

3.1 Data Enrichment

One powerful strategy to improve model accuracy is to add more data to the training set. The idea behind this is that the model will be able to capture more of the real world variation as it will have seen more data, thus improving accuracy. I gathered 598 more images from the website 'Unsplash'. As removing the outliers imbalanced the classes, I have added a different number of images to each class in order to re-balance them, each class now has 1,650 images. This is an increase of around 12%. When gathering the images I tried to include ones that the model was unlikely to have seen before and made sure to include some more difficult to classify images that for example, had low lighting or were partially occluded, to help make the model more robust. This does increase training time slightly but resulted in an increase in accuracy. The dataset now has 4,950 images, split evenly into the 3 classes.

3.2 Splitting the Data

The dataset has been split into a training set containing 80% of the data and a validation set containing 20% of the data. This means that the training set contains 3,960 images, while the validation set contains 990 images. By doing this, the validation set can be used to see what the model's accuracy is likely to be if the model is used on new, unseen data, and adjust the model accordingly.

3.3 Loss Function

A loss function is some value that can be used to estimate the loss of a model. The loss function is then used to adjust the weights of the model in order to reduce the loss on the next evaluation. There are many loss functions available for use but the design of the model will decide which is most suitable. Some examples of loss functions include: Mean Squared Error, Mean Absolute Error, Binary Cross-Entropy, Sparse Categorical Cross-Entropy and Categorical Cross-Entropy. Each loss function has some criteria that needs to be met in order to be used. As this is a classification task with more than two classes that have been one-hot encoded, Categorical Cross-Entropy is the most appropriate loss function to use.

3.4 Activation Function

An activation function is a mathematical equation which decides whether a neuron should be activated or not, this determines if a neuron is relevant to the prediction output. Activation functions have a large impact on the predictions of a model and the model's accuracy, as well as the performance efficiency of the model. Some commonly used activation functions are: Sigmoid, Rectified Linear Unit (ReLu), and SoftMax. For the hidden layers of the model, I chose to use the ReLu activation function as it is non-linear and time efficient. It is important that a non-linear activation function is used to introduce non-linearity to the data as the images are non-linear. Time efficiency is also very important when using large models such as a Convolutional Neural Network. For my output layer, I chose to use the SoftMax activation function. SoftMax outputs a multi-class categorical probability distribution and because the Categorical Cross-Entropy loss function is being used (which outputs a probability value between 0 and 1), this is the most suitable choice.

3.5 Regularisation Strategies

In order to avoid overfitting, regularisation strategies are used. Overfitting occurs when a model learns the noise in the training data too well. This leads to poor results when using the model on new data as it too closely fits the training data. This can be seen in Figure 5. The most common regularisation strategies used for CNN models are Drop Out, L2 Regularisation, and Early Stopping. By using a combination of these, the chances of overfitting can be significantly reduced.

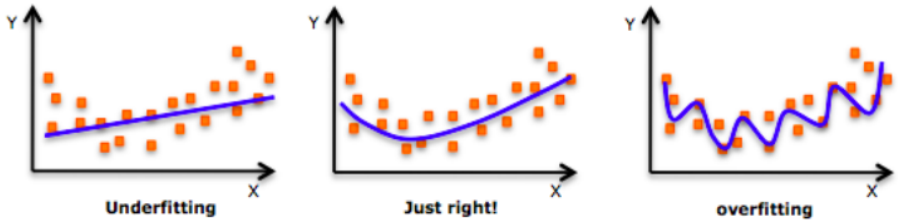

Figure 5: Example of a model underfitting, fitting well, and overfitting the data. Retrieved from Analytics Vidhya

3.5.1 Dropout

Dropout performs regularisation by randomly selecting some nodes at every iteration, and then removes them. Therefore, each iteration has a different set of nodes, resulting in a different set of outputs, helping to stop overfitting of the training data. The Keras Dropout function takes a parameter which determines the probability of choosing how many nodes should be dropped.

3.5.2 Early Stopping

Early stopping prevents overfitting by prematurely stopping training when performance on the validation set appears to be getting worse. This means that the model will stop training, ideally, before it begins to overfit the data. This can be achieved by using EarlyStopping from Keras. The main parameter of EarlyStopping is called 'patience' and specifies the number of epochs to observe with no improvement before terminating training. As early stopping can stop training when no improvements are being seen, it can also greatly improve training times as it can potentially reduce the number of epochs. For this model, validation loss is monitored for early stopping. One thing to consider is that if early stopping is activated, the most optimal model won't be saved as it wont have been the most recent model. To solve this, a ModelCheckpoint is created which saves the model with the lowest validation loss rather than the most recent model. Figure 6 shows what happens if early stopping isn't set. We can see that validation loss begins to level out and fluctuate around a point (where loss is around 0.65) from around epoch 25 but training loss continues to decrease, this suggests the model is beginning to overfit the training data. Early stopping will prevent this from happening.

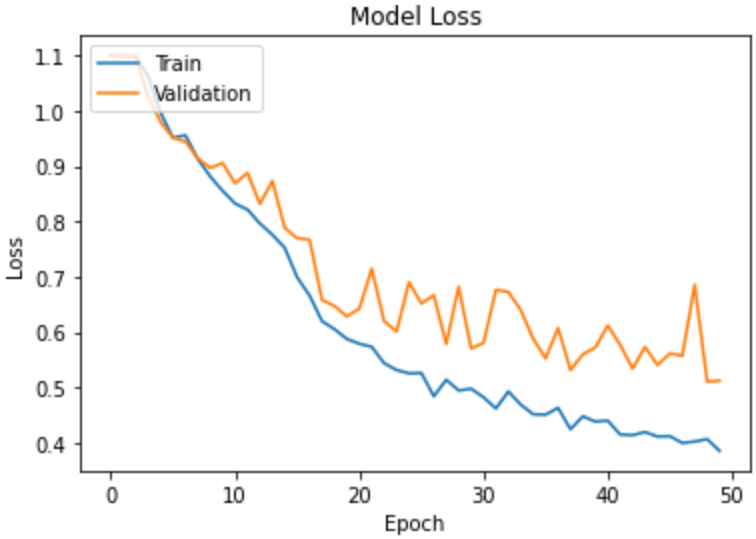

Figure 6: An example of overfitting the training data

3.5.3 L2 Regularisation

L2 regularisation updates a cost function by adding another term known as the regularization term. L2 regularization is also known as weight decay as it forces the weights to decay towards zero. The equation for L2 regularisation looks as follows, where λ is the regularization parameter:

$$Cost\ Function = Loss + \frac{\lambda}{2m} * \Sigma \|w\|^2$$

This can be done using the regularizers class from Keras. While I tested adding this to my model, it did decrease validation accuracy significantly so I decided not to include it in the final model.

3.6 Optimisation Method

An optimiser is an algorithm used to alter the attributes of the neural network with the purpose of reducing losses. For this model I experimented using three common optimisers: SGD, RMSprop, and Adam. When training the models on 10 epochs each with a learning rate of 0.001, Adam resulted in the highest validation accuracy. Based on this, Adam appears to be the best choice for this model. If I had more time to tune the hyper-parameters of the optimisers, this might change.

3.7 Hyper-parameter Tuning

Hyper-parameter tuning refers to the act of finding the set of most optimal hyper-parameters for a learning algorithm in order to increase model performance.

3.7.1 Convolutional Layers

For the model's convolutional layers, which are made up of Keras' Conv2D layers, there are a number of hyper-parameters that can be tuned in order to improve performance. The first parameter, filters, is the number of output filters in the convolution. I tested many combinations of filters ranging from 8 filters all the way to 512 filters. I found that a model with six Conv2D layers with 16, 32, 64, 64, 64, and 128 filters respectively worked best for this data. The next parameter which required tuning was strides which specifies the "step" of the convolution along

the x and y axis. For stride I tried the default of 1, then 2, and then 4. Leaving the stride as the default of 1 produced the best accuracy so I choose to leave it at that. Next, kernel size specifies the height and width of the filters. Common values are 1, 3, and 5, so these are the values I tested. A kernel size of 3 produced the greatest accuracy, therefore I choose to set kernel size to 3. The final hyper-parameter that was tuned is padding which, simply put, is an operation to change the size of the input. Padding can be set to the default 'valid' which leaves the data as is, and 'same' which produces output of the same size as the input. After testing both values, 'same' increased accuracy by around 3% so this appears to be the optimal choice.

3.7.2 Early Stopping's Patience

For this model, patience has been set to 4. After observing the model training without early stopping, the model would often go 1 or 2 epochs with worsening validation loss, but then begin to improve again. To ensure that the model training is not stopped too early, I have set patience to be reasonably large but not too large as to result in overfitting.

3.7.3 Number of Epochs

An epoch is defined as one whole pass over the dataset during training with performance being logged after each epoch. When using a validation set, such as in this project, evaluation is also run at the end of each epoch, giving the validation loss and accuracy. Increasing the number of epochs can increase accuracy but can also cause overfitting and will increase training time. As early stopping has been implemented, setting a high number of epochs has a much lower chance of overfitting although training time can be quite long if early stopping is not activated. I have chosen to set the number of epochs to 200, ensuring maximum model accuracy is achieved.

3.7.4 Optimiser Learning Rate

The primary hyper-parameter of the Adam optimiser is the learning rate. Learning rate determines the speed at which the model learns. There is, however, a trade-off between speed and convergence, where the model may learn quickly but end up at an undesirable local minimum or learn too slow and fail to converge. After testing learning rates of 0.1, 0.01, 0.001, and 0.0001 I found that the optimal rate was 0.001. At this rate, training time is still sensible yet ensures smooth convergence. I also implemented an adaptive learning rate using ReduceLROnPlateau from Keras. This will decrease the learning rate by 25% if the validation loss doesn't decrease after 2 epochs and therefore lowers the penalty of potentially setting the initial learning rate too high.

4 Results

4.1 Multilayer Perceptron Model

The baseline Multilayer Perceptron Model (MLP) was comprised of 3 dense layers (filter sizes 16, 32, and 64 respectively) and used the SGD optimiser with a learning rate of 0.001 and categorical cross-entropy as the loss function. The MLP was trained over 25 epochs and took around 14 minutes. It achieved a training accuracy of 49.6%, a validation accuracy of 46.5%, and a test accuracy of 33%. This model performed very poorly, though that was expected from no hyper-parameter tuning or regularisation. Figure 7a and 7b show the accuracy and loss for both the training and validation sets per epoch for the MLP model.

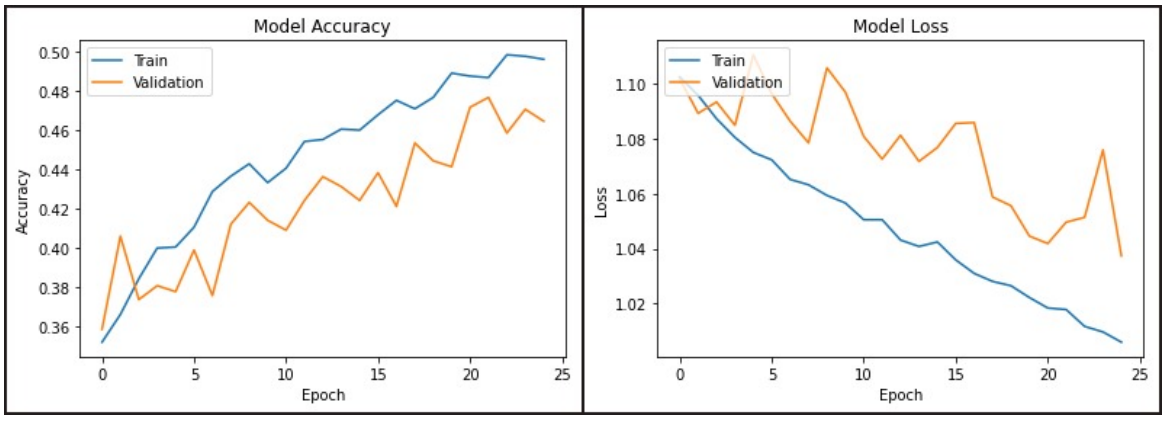

Figure 7: (a) MLP model accuracy per epoch, (b) MLP model loss per epoch

4.2 Convolutional Neural Network

The CNN model took 32 minutes to train, with early stopping activating after 28 epochs. It achieved a training accuracy of 75%, a validation accuracy of 74%, and a test accuracy of 80%, reached at epoch 24. This similarity between the two accuracies suggests very little chance of the model overfitting the training data and Figure 8, which shows that the training/validation loss and accuracy per epoch confirms this. This is a huge improvement when compared to the baseline model, increasing validation accuracy by around 27.5% and increasing test accuracy by 47%. This increase in accuracy was expected as MLP models generally don't perform as well as a CNN does when it comes to image classification. One reason for this is that MLP models are not spatially invariant while CNN models are. Spatially invariant means that the model is not sensitive to the position of the object in an image. For example, if a strawberry is first seen in the top left of an image and then again in the bottom right, the MLP will assume that the strawberry will be in the bottom right in the future. This is not an issue for CNN models as they are able to capture this information, significantly improving accuracy.

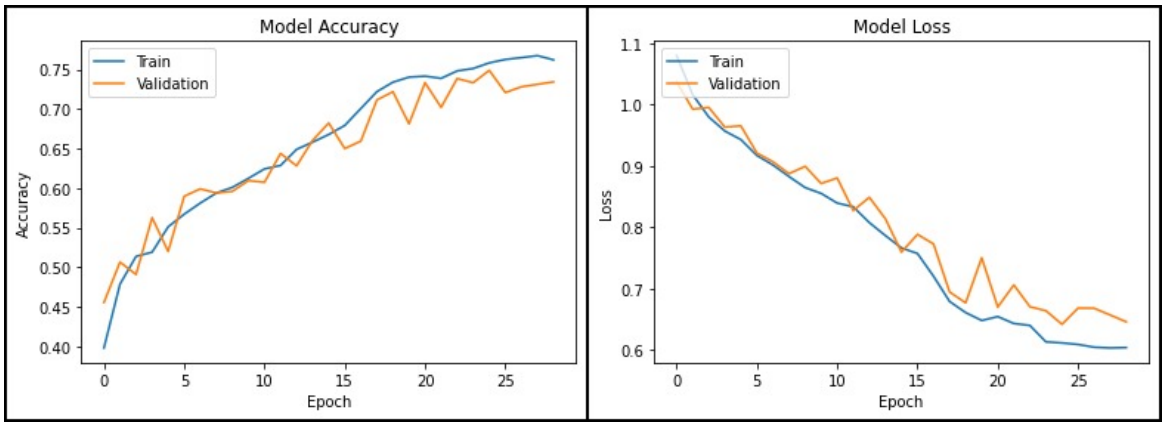

Figure 8: (a) CNN model accuracy per epoch, (b) CNN model loss per epoch

5 Conclusion

The purpose of this project was to create a convolutional neural network capable of accurately predicting various fruit in images. This has been accomplished. If the final CNN model is used on new images of strawberries, cherries, and/or tomatoes, then we expect it to successfully classify 80% of the images as the correct fruit. This is far better than if we were to just assume all images to be of one class, which would give us an accuracy of 33%. When compared to the

baseline multilayer perceptron model, we have greatly improved accuracy, helping to confirm that the CNN model works well. The CNN model was also very time efficient. Overall, the CNN performs very well.

6 Future Work

While the model did perform well, there are still some things that could be done to potentially improve it further. I would like to look into transfer learning. Transfer learning refers to the use of very powerful, pre-trained models, such as ImageNet, to classify the images. This can greatly improve both time and accuracy as they have already been trained on similar data. These large models typically take weeks to train but as they have been pre-trained, we can benefit from their complexity without having to train them ourselves. These models have the potential of reaching 90% accuracy or more. Another thing that would improve the model would be to get even more data in order to learn more of the real-world noise. Although I was able to add quite a few images, it was quite time consuming and I wasn't able to add as many as I would have liked to. Other more powerful models are often trained on considerably more data than what I used. It isn't uncommon to have a training set of 50,000 or more images. Finally, using ensemble learning could also improve model accuracy. Ensemble learning is where multiple models are trained and the predictions of those are combined to reduce the variance of predictions, often leading to increased accuracy. By looking into these strategies, I may be able to achieve even better performance than I already have.