Comp309 Project Report

Introduction

The goal for this project is to use deep convolutional neural network to build a model to classier images into 3 different classes. The 3 classes are tomato, cherry, and strawberry. There are 6,000 images in total, 4,500 images are given as the training data to train/learn a CNN model and rest of 1500 images are use for test data to evaluate the CNN model that I builded. During build the model I will use my knowledge such as loss functions, optimisation techniques, regularisation strategy, activation ect to tune my model. Strive for achieve higher performance in my CNN model.

Problem investigation

Augmentation

From the 4,500 given images, I find out there are all different. Like the fruit's shape, position, brightness, the numbers of fruits in image ect. So I design to augmenting the existing data-set with perturbed versions of the existing images. Scaling, rotations and other affine transformations are typical. This is done to expose the neural network to a wide variety of variations. This makes it less likely that the neural network recognises unwanted characteristics in the data-set.

Normalization

Also in the model, I did set the rescale value as 1/255, it is a value by which will multiply the data before any other processing. Our original images consist in RGB coefficients in the 0-255, but such values would be too high for our models to process (given a typical learning rate), so I scaling with a 1/255, target values between 0 and 1.

The above preprocessing methods helps my model improve 2% (from 93% to 95%) of the accuracy, but this is not a good measurement for measure the preprocessing method preprocessing. If I apply those methods on a simple model, is should improve more accuracy.(from 72% - 84%). The follow preprocessing is I have been attempted but it do not improve the accuracy.

Standardisation

I have try to apply standardisation for my model, standardization is the process of centralizing data by de-averaging. According to the knowledge of convex optimization theory and data probability distribution, data centering conforms to the data distribution law, and it is easier to obtain the generalization effect after training. But seems it does not help for my model.

Dimensionality reduction

This method is choose to collapse the RGB channels into a single gray-scale channel. There are often considerations to reduce other dimensions, when the neural network performance is allowed to be invariant to that dimension, or to make the training problem more tractable. When it apply in model, the training speed for this model actually increases, but the model accuracy is decreases. This method can used in some ways for the test model during tuning parameter, but it should not apply on the final model.

Methodology

Data split

From 4500 provided image data , I did randomly split 80% of the data as training set and 20% of the data as validation set. That means training set contains 3600 images and validation set contains 900 images. The reasons for I split the data in this ways is because I want the model have enough training before testing, otherwise will be course under-training problem. But if I set all the image data as training set, it will cause training bias problem and it might cause overfitting problem. Without validation set, we cannot monitoring the model during training, to observe when over fitting problem occur and to judgment whether or not to execute early stop to avoid overfitting problem occur. But after I thought the final proportion of training data and test data is 4500:1500, so I design to tune the proportion of training data and validation data to 3:1 as well. Which is (3375:1125),this Is for make better simulate the final test environment.

Loss function

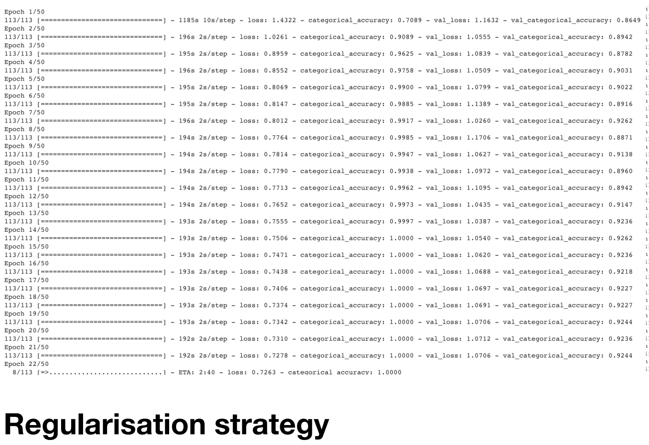
The loss function I used in my model is 'categorical_crossentropy', this is because my model is going to classier the images into 3 different classes that means my target is in categorical format. After using the 'categorical_crossentropy' loss, the target for each sample should be a 3 dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample.

Optimisation

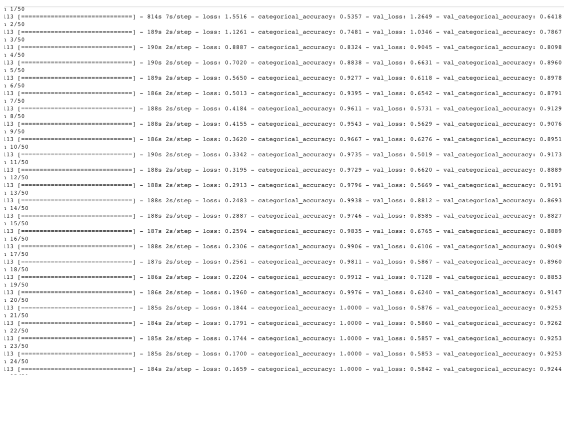
For the optimisation I have been used 'adam' and 'sgd' to test out which optimisation is the best fit for my model. During the test, I find out each of them have they own benefits and here is some interesting features I observed: Adam optimisation is an algorithm for first-order gradient-based optimisation of stochastic objective functions, based on adaptive estimates of lower-order moments. When it apply on my VGG16 model it usually stuck at the beginning (the validation loss keep decreasing, but accuracy did not change). After I did tune the Adam learning rate and decay and added some preprocessing step in my mode, the model does find a way to improve it's learning efficiency. But after it close to the bottleneck the learning efficiency will start turning low. This is because it's adaptive mechanism of estimates, the learning rate turning low. In spite of this, the accuracy is keep increasing and loss is keep decreasing and it needs very long time to train.

Another optimisation Stochastic gradient descent (sgd) does a different effect, the learning efficiency usually increasing at the beginning and I don't have to do much tuning for optimisation to let it apply for my model, and it reach the bottleneck quite fast, up to 20 epoch the validation accuracy is reach 92% But when it reach the bottleneck it is hard to breakthrough, it usually just hold the same performance.

Both of optimisation works well for my model, relatively sdg optimisation is more stable. Because use the same sdg dataset can work on different datasets, the applicable environment is widespread. The Adam optimisation have to do a bit tuning when it apply on a small datasets, otherwise the performance will drop a lot compare to other datasets. But Adam is suitable for break through the bottleneck, to achieve higher performances.



```
Epoch 1/50
113/113 [==============================] - 1185s 10s/step - loss: 1.4322 - categorical_accuracy: 0.7089 - val_loss: 1.1632 - val_categorical_accuracy: 0.8649
Epoch 2/50
113/113 [==============================] - 196s 2s/step - loss: 1.0261 - categorical_accuracy: 0.9089 - val_loss: 1.0555 - val_categorical_accuracy: 0.8942
Epoch 3/50
113/113 [==============================] - 195s 2s/step - loss: 0.8959 - categorical_accuracy: 0.9625 - val_loss: 1.0839 - val_categorical_accuracy: 0.8782
Epoch 4/50
113/113 [==============================] - 196s 2s/step - loss: 0.8552 - categorical_accuracy: 0.9758 - val_loss: 1.0509 - val_categorical_accuracy: 0.9031
Epoch 5/50
113/113 [==============================] - 195s 2s/step - loss: 0.8069 - categorical_accuracy: 0.9900 - val_loss: 1.0799 - val_categorical_accuracy: 0.9022
Epoch 6/50
113/113 [==============================] - 195s 2s/step - loss: 0.8147 - categorical_accuracy: 0.9885 - val_loss: 1.1389 - val_categorical_accuracy: 0.8916
Epoch 7/50
113/113 [==============================] - 196s 2s/step - loss: 0.8012 - categorical_accuracy: 0.9917 - val_loss: 1.0260 - val_categorical_accuracy: 0.9262
Epoch 8/50
113/113 [==============================] - 194s 2s/step - loss: 0.7764 - categorical_accuracy: 0.9985 - val_loss: 1.1706 - val_categorical_accuracy: 0.8871
Epoch 9/50
113/113 [==============================] - 194s 2s/step - loss: 0.7814 - categorical_accuracy: 0.9947 - val_loss: 1.0627 - val_categorical_accuracy: 0.9138
Epoch 10/50
113/113 [==============================] - 194s 2s/step - loss: 0.7790 - categorical_accuracy: 0.9938 - val_loss: 1.0972 - val_categorical_accuracy: 0.8960
Epoch 11/50
113/113 [==============================] - 194s 2s/step - loss: 0.7713 - categorical_accuracy: 0.9962 - val_loss: 1.1095 - val_categorical_accuracy: 0.8942
Epoch 12/50
113/113 [==============================] - 194s 2s/step - loss: 0.7652 - categorical_accuracy: 0.9973 - val_loss: 1.0435 - val_categorical_accuracy: 0.9147
Epoch 13/50
113/113 [==============================] - 193s 2s/step - loss: 0.7555 - categorical_accuracy: 0.9997 - val_loss: 1.0387 - val_categorical_accuracy: 0.9236
Epoch 14/50
113/113 [==============================] - 193s 2s/step - loss: 0.7506 - categorical_accuracy: 1.0000 - val_loss: 1.0540 - val_categorical_accuracy: 0.9262
Epoch 15/50
113/113 [==============================] - 193s 2s/step - loss: 0.7471 - categorical_accuracy: 1.0000 - val_loss: 1.0620 - val_categorical_accuracy: 0.9236
Epoch 16/50
113/113 [==============================] - 193s 2s/step - loss: 0.7438 - categorical_accuracy: 1.0000 - val_loss: 1.0688 - val_categorical_accuracy: 0.9218
Epoch 17/50
113/113 [==============================] - 193s 2s/step - loss: 0.7406 - categorical_accuracy: 1.0000 - val_loss: 1.0697 - val_categorical_accuracy: 0.9227
Epoch 18/50
113/113 [==============================] - 193s 2s/step - loss: 0.7374 - categorical_accuracy: 1.0000 - val_loss: 1.0691 - val_categorical_accuracy: 0.9227
Epoch 19/50
113/113 [==============================] - 193s 2s/step - loss: 0.7342 - categorical_accuracy: 1.0000 - val_loss: 1.0706 - val_categorical_accuracy: 0.9244
Epoch 20/50
113/113 [==============================] - 192s 2s/step - loss: 0.7310 - categorical_accuracy: 1.0000 - val_loss: 1.0712 - val_categorical_accuracy: 0.9236
Epoch 21/50
113/113 [==============================] - 192s 2s/step - loss: 0.7278 - categorical_accuracy: 1.0000 - val_loss: 1.0706 - val_categorical_accuracy: 0.9244
Epoch 22/50
  8/113 [=>............................] - ETA: 2:40 - loss: 0.7263 - categorical_accuracy: 1.0000
```

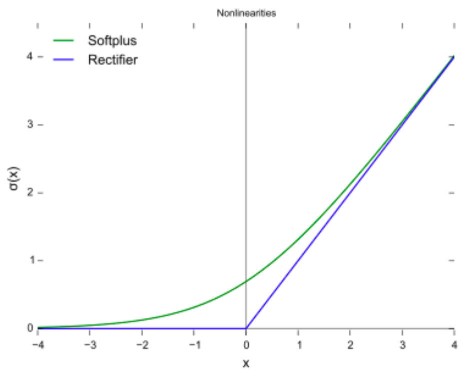


```
1/50
13 [==============================] - 814s 7s/step - loss: 1.5516 - categorical_accuracy: 0.5357 - val_loss: 1.2649 - val_categorical_accuracy: 0.6418
2/50
13 [==============================] - 189s 2s/step - loss: 1.1261 - categorical_accuracy: 0.7481 - val_loss: 1.0346 - val_categorical_accuracy: 0.7867
3/50
13 [==============================] - 190s 2s/step - loss: 0.8887 - categorical_accuracy: 0.8324 - val_loss: 0.9045 - val_categorical_accuracy: 0.8098
4/50
13 [==============================] - 190s 2s/step - loss: 0.7020 - categorical_accuracy: 0.8838 - val_loss: 0.6631 - val_categorical_accuracy: 0.8960
5/50
13 [==============================] - 189s 2s/step - loss: 0.5650 - categorical_accuracy: 0.9277 - val_loss: 0.6118 - val_categorical_accuracy: 0.8978
6/50
13 [==============================] - 186s 2s/step - loss: 0.5013 - categorical_accuracy: 0.9395 - val_loss: 0.6542 - val_categorical_accuracy: 0.8791
7/50
13 [==============================] - 188s 2s/step - loss: 0.4184 - categorical_accuracy: 0.9611 - val_loss: 0.5731 - val_categorical_accuracy: 0.9129
8/50
13 [==============================] - 188s 2s/step - loss: 0.4155 - categorical_accuracy: 0.9543 - val_loss: 0.5629 - val_categorical_accuracy: 0.9076
9/50
13 [==============================] - 186s 2s/step - loss: 0.3620 - categorical_accuracy: 0.9667 - val_loss: 0.6276 - val_categorical_accuracy: 0.8951
10/50
13 [==============================] - 190s 2s/step - loss: 0.3342 - categorical_accuracy: 0.9735 - val_loss: 0.5019 - val_categorical_accuracy: 0.9173
11/50
13 [==============================] - 188s 2s/step - loss: 0.3195 - categorical_accuracy: 0.9729 - val_loss: 0.6620 - val_categorical_accuracy: 0.8889
12/50
13 [==============================] - 188s 2s/step - loss: 0.2913 - categorical_accuracy: 0.9796 - val_loss: 0.5669 - val_categorical_accuracy: 0.9191
13/50
13 [==============================] - 188s 2s/step - loss: 0.2483 - categorical_accuracy: 0.9938 - val_loss: 0.8812 - val_categorical_accuracy: 0.8693
14/50
13 [==============================] - 188s 2s/step - loss: 0.2887 - categorical_accuracy: 0.9746 - val_loss: 0.8585 - val_categorical_accuracy: 0.8827
15/50
13 [==============================] - 187s 2s/step - loss: 0.2594 - categorical_accuracy: 0.9835 - val_loss: 0.6765 - val_categorical_accuracy: 0.8889
16/50
13 [==============================] - 188s 2s/step - loss: 0.2306 - categorical_accuracy: 0.9906 - val_loss: 0.6106 - val_categorical_accuracy: 0.9049
17/50
13 [==============================] - 187s 2s/step - loss: 0.2561 - categorical_accuracy: 0.9811 - val_loss: 0.5867 - val_categorical_accuracy: 0.8960
18/50
13 [==============================] - 186s 2s/step - loss: 0.2204 - categorical_accuracy: 0.9912 - val_loss: 0.7128 - val_categorical_accuracy: 0.8853
19/50
13 [==============================] - 186s 2s/step - loss: 0.1960 - categorical_accuracy: 0.9976 - val_loss: 0.6240 - val_categorical_accuracy: 0.9147
20/50
13 [==============================] - 185s 2s/step - loss: 0.1844 - categorical_accuracy: 1.0000 - val_loss: 0.5876 - val_categorical_accuracy: 0.9253
21/50
13 [==============================] - 184s 2s/step - loss: 0.1791 - categorical_accuracy: 1.0000 - val_loss: 0.5860 - val_categorical_accuracy: 0.9262
22/50
13 [==============================] - 185s 2s/step - loss: 0.1744 - categorical_accuracy: 1.0000 - val_loss: 0.5857 - val_categorical_accuracy: 0.9253
23/50
13 [==============================] - 185s 2s/step - loss: 0.1700 - categorical_accuracy: 1.0000 - val_loss: 0.5853 - val_categorical_accuracy: 0.9253
24/50
13 [==============================] - 184s 2s/step - loss: 0.1659 - categorical_accuracy: 1.0000 - val_loss: 0.5842 - val_categorical_accuracy: 0.9244
```

Regularisation strategy

For regularisation, it is a very important technique in machine learning to prevent overfitting. It adds a regularization term in order to prevent the coefficients to fit so perfectly to overfit. I did use keras_regularizer and activity_regularizer to apply penalties on layer activity during optimisation. In thoes two regularisation strategy, keras_regularizer is applied on the kernel weights matrix, and activity_regularizer is applied to the output of the layer. In the regulariser, there are two kinds of functions, L1 loss function and L2 loss function. The difference between the L1 and L2 is just that L2 is the sum of the square of the weights, while L1 is just the sum of the weights. By contrast, L1 regularisation can help generate a sparse weight matrix, which can be used for feature selection. In predicting or classification, so many features obviously difficult to choose, but if put those furthers into a sparse model, which only a few furthers have contribution to the model, but most of feathers have no contributions for the model, or have very few contribution (because they are in front of the coefficient is zero or is a very small value, even with is no effect on the model), then we can only focus on the feathers of the coefficient is not zero. For L2 loss function, the fitting process are generally inclined to let weight as small as possible, finally use all the small parameters to create a model. Because the model build by small parameters, it can adapt to different data sets and avoids over fitting phenomenon to some extent. So I apply L2 loss function in keras_regularizer and L1 loss function in activity_regularizer.

Activation function



The activation function in hidden layer I used is rectifier. We can see from the left rectifier figure, the range of x and a(x) is from 0 to infinite, if it compared to sigmoid function or similar activation functions, ReLU only need a threshold value can be activated, without having to calculate a complex set of operations, it allow for faster and effective training of deep neural architectures on large and complex datasets.

In the output layer, the activation function I used is softmax. oftmax used in classification process, transform the output of the neutron, mapped to (0, 1) range, can be said as a probability, thus to classify more types.

Hyper-parameter settings

For the hyper-parameter settings, the most of the parameters I adjusted is they learning rates. (the learning rates from L1 and L2 loss function and Adam optimiser learning rate) During tuning the model, I find out a rule: the more data I use for training, the lower learning rate I have to adjust. I thing this is because when my model recognize lots of features, it just need a very low learning rate to making sure that we do not miss any local minimum. Although we'll be taking a long time to converge—especially if we get stuck on a plateau region. If we use larger learning rate, gradient descent can overshot the minimum. It may fail to coverage, or even diverge.

Extra technical advancement

In my model, I did use call back function to apply early stop technique and check point technique. The early stop technique can Stop training when a monitored quantity has stopped improving. To observe this phenomenon, I did set the validation data during training and set the patience parameter in early shop technique as 5. This means that number of epochs with no improvement after which training will be stopped. The monitor I used is validation loss, if in next 5 epoch the validation loss did not decrease the model will stop training. After the training done, the check point will save the best model automatically, here I just the same monitor as the early stop. Which will save the model that have lowest loss. The reasons for apply those two techniques is because to avoid overfitting problem and get the best model after every epoch.

The use of existing models

My best model is basic build on the keras model VGG16, the figure on the left is the model after I transform VGG16. The processing of training shows as follow: Input a 300x300x3 image doing twice convolution using 64 filters (used only 3*3 size), then Max pooling layers (used only 2*2 size), then next block use 128 filter twice larger than last block, and so on. But the 5th block is the same as the 4th block, last fully connected with 256 nodes and output layer with softmax activation with 3 nodes. Because consider the time factor, I am not design to use the more complex model than VGG16, and also I did decrease the dense layer nodes from the original VGG16 model. In spite of this, the time for run single epoch of this model on google colab use full data set (4500 images) will cost 6 min. Then to run 20 epoch that will cost about 2 hours to train for one model.

```
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 300, 300, 64)      1792

block1_conv2 (Conv2D)        (None, 300, 300, 64)      36928

block1_pool (MaxPooling2D)   (None, 150, 150, 64)      0

block2_conv1 (Conv2D)        (None, 150, 150, 128)     73856

block2_conv2 (Conv2D)        (None, 150, 150, 128)     147584

block2_pool (MaxPooling2D)   (None, 75, 75, 128)       0

block3_conv1 (Conv2D)        (None, 75, 75, 256)       295168

block3_conv2 (Conv2D)        (None, 75, 75, 256)       590080

block3_conv3 (Conv2D)        (None, 75, 75, 256)       590080

block3_pool (MaxPooling2D)   (None, 37, 37, 256)       0

block4_conv1 (Conv2D)        (None, 37, 37, 512)       1180160

block4_conv2 (Conv2D)        (None, 37, 37, 512)       2359808

block4_conv3 (Conv2D)        (None, 37, 37, 512)       2359808

block4_pool (MaxPooling2D)   (None, 18, 18, 512)       0

block5_conv1 (Conv2D)        (None, 18, 18, 512)       2359808

block5_conv2 (Conv2D)        (None, 18, 18, 512)       2359808

block5_conv3 (Conv2D)        (None, 18, 18, 512)       2359808

block5_pool (MaxPooling2D)   (None, 9, 9, 512)         0

flatten (Flatten)            (None, 41472)             0

dense (Dense)                (None, 256)               10617088

dense_1 (Dense)              (None, 256)               65792

dense_2 (Dense)              (None, 3)                 771
=================================================================
Total params: 25,398,339
Trainable params: 25,398,339
Non-trainable params: 0
```

Result discussion

20 Epoch	Time	Loss	Accuracy
MLP	118s per Epoch	0.9907	0.5023
VGG16-SGD	265s per Epoch	1.0706	0.9234
VGG16-ADAM	289s per Epoch	0.5876	0.9253

Compare the above results, MLP model have lower accuracy and fast training speed, it achieve 0.5023 accuracy and cost 118s per epoch. On the VGG16 model, it achieve 0.92 accuracy after 20 epoch but need to cost 265s per epoch. This is because for the MLP model, it just the same setup as the VGG16 model (Eg flatten and dense layer batch size ect), but without use regularisation and convolution layer. In the flatten layer, just for flattens the input and the first two dense layer it just the regular densely connected NN layer. And the last layer is the output layer. The setting number 256 is the number of hidden nodes, I think this number might have good balance between accuracy and time cost, and the setting 3 is the number of the classes. The most important reasons for they have different results is because in the VGG16 model it added convolution layer, since it cost time for the images go through convolution layer and it helps the model better to recognize the features. Therefore VGG16 model achieve higher accuracy and cost more time than MLP model.

Conclusions and future work

In conclusion, VGG16 using Adam optimiser achieve highest performance for classier the images into tomato, cherry, and strawberry 3 different classes. The benefits for this model is it can achieve high accuracy and low loss. But on the other hand, it will cost very long time to achieve high accuracy, it toke about 40 epoch (about 3 hour) to achieve 94% accuracy on the validation dataset. Also Adam optimiser is not suitable for all the datasets, if I apply this model on different dataset I have to adjust the parameter. For example If I have a larger dataset I have adjust to a smaller learning rate. But if I use sgd optimiser the problem will not exist, and it will take a bit shorter time to train. The disadvantages is do not have the higher accuracy like use Adam optimiser.

About the future work, to achieve higher accuracy it might need a better GPU to run my program, then I can use more complex models such as InceptionResNetV2, Xception NASNetLarge ect. Currently use the google Colab to do tuning on those models is too show. Another ways to achieve higher performances is to collect more images to train the model, that will increase the learning efficiency. Further more, is to do tuning on the model, on the condition of using the same configuration to find out the most suitable parameters.