COMP309 – Project

Image processing and CNN

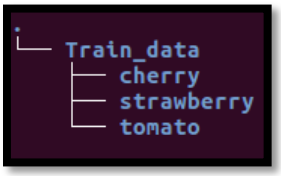
Andrew Garrett – 300352955

Introduction

The task is to use a CNN to classify an image into one of the following classes: “Cherry”, “Tomato”, “Strawberry”. The classification must be done using Kera's, a library based on TensorFlow which a state-of-the-art and user-friendly tool is designed to enable fast experimentation with deep CNNs. There are 4500 images to be processed use EDA and image preprocessing, some techniques to use when tuning the CNN model include **Organizing data, Image augmentation, regularization strategy, loss functions, hyper parameters, optimization techniques.**

Exploratory data analysis (Problem investigation)

We have a training set with three classes, tomatoes, cherries, and strawberries. The training set is split into three folders, with 1500 instances in each folder respectively. Below is a screenshot of our current data structure.



Import these statements first as these are the tools we will need for EDA.

```
# Imports
import glob
import numpy as np
import os.path as path
import numpy as np
import random
import tensorflow as tf
from scipy import misc
```

Retrieve the file name of both cherry, tomato, and strawberry.

```
IMAGE_PATH_STARWBERRY = 'drive/My Drive/data/Test_data/cherry'
IMAGE_PATH_TOMATO = 'drive/My Drive/data/Test_data/tomato'
IMAGE_PATH_CHERRY = 'drive/My Drive/data/Test_data/cherry'
```

Load the images into a single variable and convert to a numpy array

```
#load file paths
cherry_file_paths = glob.glob(path.join(IMAGE_PATH_CHERRY, '*.jpg'))
strawberry_file_paths = glob.glob(path.join(IMAGE_PATH_STRAWBERRY, '*.jpg'))
tomato_file_paths = glob.glob(path.join(IMAGE_PATH_TOMATO, '*.jpg'))

# Load the images
cherry_images = [misc.imread(path) for path in cherry_file_paths]
cherry_images = np.asarray(images)

strawberry_images = [misc.imread(path) for path in strawberry_file_paths]
strawberry_images = np.asarray(images)

tomato_images = [misc.imread(path) for path in tomato_file_paths]
tomato_images = np.asarray(images)
```

Now retrieve the size of the images

```
# Get image size
cherry_image_size = np.asarray([cherry_images.shape[1], cherry_images.shape[2], cherry_images.shape[3]])
print(cherry_image_size)

strawberry_image_size = np.asarray([strawberry_images.shape[1], strawberry_images.shape[2], strawberry_images.shape[3]])
print(strawberry_image_size)

tomato_image_size = np.asarray([tomato_images.shape[1], tomato_images.shape[2], tomato_images.shape[3]])
print(tomato_image_size)
```

```
[300 300   3]
[300 300   3]
[300 300   3]
```

This image_size array will let us know the dimensions of the images. If we print this variable, we learn that the dimensions of the images are [300, 300, 3]. This means that each image in the dataset has 300 rows, 300 columns, and a depth of 3 (or 3 channels, Red, Green, and Blue). These numbers define the spatial resolution of the image, now we need to sort our current data out into images and labels.

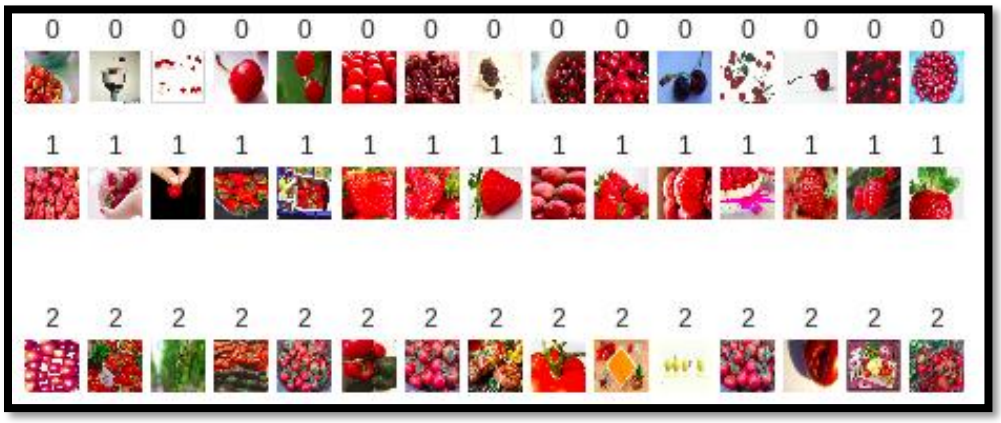
Seeing the screenshots above we can go ahead and create our labeled lists using the first character at the beginning of each image name to decide what class the image belongs to. We can do this by passing the starryberry_images, cherry_images, tomato_images and their corresponding file paths to the method shown below.

```
def retreive_image_labels(images, file_paths):
  n_images = images.shape[0]
  labels = np.zeros(n_images)
  for i in range(n_images):
    #0 for cherry, 1 for strawberry and 2 for tomato
      filename = path.basename(file_paths[i])[0]
      if(filename[0]=='c'):
        labels[i]==0
      elif(filename[0]=='s'):
        labels[i]==1
      elif(filename[0]=='t'):
        labels[i]==2

  return labels
```

```
#retrieve image labels
cherry_labels = retreive_image_labels(cherry_images, cherry_file_paths)
strawberry_labels = retreive_image_labels(strawberry_images, strawberry_file_paths)
tomato_labels = retreive_image_labels(tomato_images, tomato_file_paths)
```

Now we would like to finally visualize our data within our python code to analyze the data and ensure the labels have been matched correctly, we can do this using the “visualize_data” method below.

```
def visualize_data(cherry_images,strawberry_images, tomato_images, cherry_labels, strawberry_labels, tomato_labels):

    figure = plt.figure()
    count = 0
    for i in range(cherry_images.shape[0]):
        count += 1

        figure.add_subplot(10, cherry_images.shape[0], count)
        plt.imshow(cherry_images[i, :, :])
        plt.axis('off')
        plt.title(cherry_labels[i])

        figure.add_subplot(2, strawberry_images.shape[0], count)
        plt.imshow(strawberry_images[i, :, :])
        plt.axis('off')
        plt.title(strawberry_labels[i])

        figure.add_subplot(1, tomato_images.shape[0], count)
        plt.imshow(tomato_images[i, :, :])
        plt.axis('off')
        plt.title(tomato_labels[i])

    plt.show()
```



Now looking at the screenshot above and drawing on our previous observations we know that all the images have been re-sized to the same size/dimensions, e.g. 300 × 300. However, as you can see above, the images show different properties in terms of the background, light condition, number of objects, resolution. Having a closer look with in the files themselves we can see that the files seem to also include noisy images, noisy objects or outliers, which need to be handled properly (minimal amount).

Image pre-processing (Problem investigation)

Given that a large amount of preprocessing has already been conducted on the set prior to us using it, we will not need to spend too much time on this part. Quickly going through the data files and removing images to the obvious eye we are able to remove an image of a raspberry, a girl, strawberries and tomatoes which can confuse the model as both are part of the class, and a quiche.

Also noticing there are a few black and white images that are not RGB as seen below. We will need to handle this.

Writing the code below allows use to both convert the image to our desired size and also change any given image to an RGB image. Setting our image size to 32*32 will allow for our model to compute epochs a lot quicker than 300*300 and save us time while still preserving the accuracy of our model.

```
# Load the images
images = np.array([np.array((Image.open(fname).convert("RGB")).resize([32, 32])) for fname in file_paths])
```

Now we want to rescale the images using the code snippet provided below. (essentially using normalization)

```
#scale the images
strawberry_images = strawberry_images / 255
cherry_images = cherry_images / 255
tomato_images = tomato_images / 255
```


```
#scale the images
strawberry_images = strawberry_images / 255
cherry_images = cherry_images / 255
tomato_images = tomato_images / 255
```

Tuning the CNN (Methodology)

Organizing data

We have a training set with three classes, tomatoes, cherries, and strawberries. The training set is split into three folders, with 1500 instances in each folder respectively. Firstly, we will need to create a validation set. A validation set will help prevent over fitting of our model, our model will use the validation set while training, specifically it will test the validation set after each epoch and compare that to the accuracy of the training set. To do this we need to separate the given training set of 4500 instances into 20 percent validation and 80 percent training. That means for our validation there is a total of 900 instances.

Image augmentation

```
def train_model(model):
  train_datagen = ImageDataGenerator(rescale = 1./255,
                                     shear_range = 0.2,
                                     zoom_range = 0.2,
                                     horizontal_flip = True)

  valid_datagen = ImageDataGenerator(rescale = 1./255)

  training_set = train_datagen.flow_from_directory('./drive/My Drive/data/Train_data',
                                                   target_size = (300, 300),
                                                   batch_size = 35,
                                                   class_mode = 'categorical')

  valid_set = valid_datagen.flow_from_directory('./drive/My Drive/data/Valid_data',
                                                target_size = (300, 300),
                                                batch_size = 45,
                                                class_mode = 'categorical')
  model.fit_generator(training_set,
                      steps_per_epoch = 100,
                      epochs = 10,
                      validation_data = valid_set,
                      validation_steps = 5)
  return model
```

Using the code above we use keras “ImageDataGenerator” to effectively generate augmented images with combinations of rotational shifts, width and height shifts, zooming and horizontal flips giving us a distinctive set of 20 images per each image that we load in. Loading in 5 images per class will give us 5*20*3 which is 300 images added to our training data set giving us 1800 in total to train on. A small screenshot of a couple of augmented tomato images are shown below.

After enriching the training set by 300 new images, our model’s accuracy increased on the train set by 5%, and the test set by 2%

Investigate the regularization strategy

While we have used validation sets to avoid overfitting in our model, we can do better. We can use regularization. This is a technique which makes slight modifications to the learning algorithm such that the model generalizes better, this improves the model's performance on unseen data. We can also use drop outs set at 0.5 to regularize our model as displayed in the code below.

```
def construct_model(size):
    model = Sequential()
    model.add(Conv2D(filters=64,
                 kernel_size=(3, 3),
                 activation='relu',
                 kernel_initializer='he_normal',    # better for relu based networks
                 input_shape=size))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(filters=252,
                 kernel_size=(3, 3),
                 activation='relu',
                 kernel_initializer='he_normal'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(512, activation='relu',
                kernel_regularizer=regularizers.l2(0.05)))
    model.add(Dropout(0.7))
    model.add(Dense(3, activation='softmax'))

    # Compiling the CNN
    sgd = optimizers.SGD(lr=0.01, momentum=0.4, decay=0.0, nesterov=False)
    model.compile(loss='mean_squared_error',
              optimizer=sgd,
              metrics=['accuracy'])
    model.summary()

    return model
```

Investigate optimization techniques.

Possible optimization techniques include Adam, Adagrad, AdaDelta or RMSProp and SGD. Stochastic gradient descent performs a parameter update for each training example, it is a relatively fast technique and easy to implement, we can define our SGD as seen below and pass through to our model when compiling.

```
# Compiling the CNN
sgd = optimizers.SGD(lr=0.01, momentum=0.4, decay=0.0, nesterov=False)
model.compile(loss='mean_squared_error',
          optimizer=sgd,
          metrics=['accuracy'])
model.summary()
```

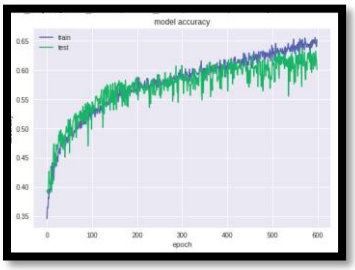
Optimizing the SGD - to optimize our learning rate, momentum, and decay, we can use the history callback to display our results for different parameters on the SGD to optimize our gradient descent.

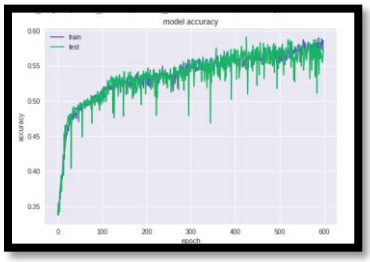
Max pooling

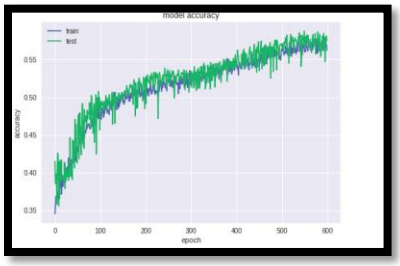
Max pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned. As Seen below we can define our model with a max pooling layer for each convolutional layer

```
def construct_model(size):
    model = Sequential()
    model.add(Conv2D(filters=64,
                     kernel_size=(3, 3),
                     activation='relu',
                     kernel_initializer='he_normal',    # better for relu based networks
                     input_shape=size))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(filters=252,
                     kernel_size=(3, 3),
                     activation='relu',
                     kernel_initializer='he_normal'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(512, activation='relu',
                    kernel_regularizer=regularizers.l2(0.05)))
    model.add(Dropout(0.7))
    model.add(Dense(3, activation='softmax'))

    # Compiling the CNN
    sgd = optimizers.SGD(lr=0.01, momentum=0.4, decay=0.0, nesterov=False)
    model.compile(loss='mean_squared_error',
                  optimizer=sgd,
                  metrics=['accuracy'])
    model.summary()

    return model
```

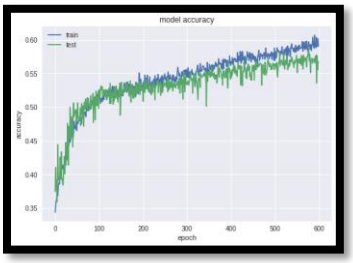
Investigate loss functions.

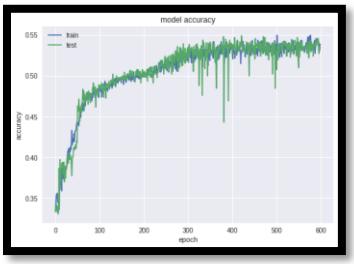


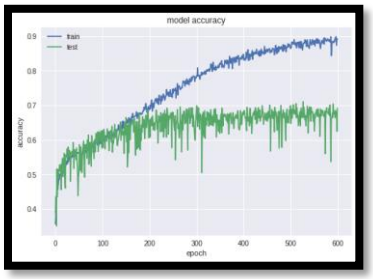




MSE **mean_absolute_error** **mean_squared_logarithmic_error**







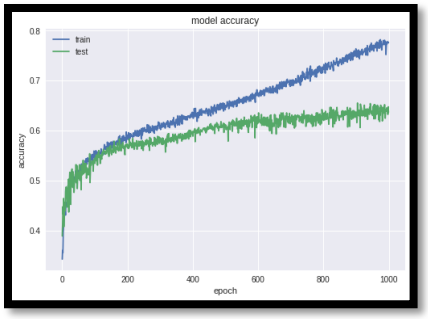
squared_hinge **Hinge** **categorical_crossentropy**

For our classification problem, mean squared error (L2 loss) and cross entropy loss are widely used. Cross-entropy - Provides us with faster learning when our predictions differ significantly from our labels (usually when training has just started). Looking at the results above, MSE overall performs well with the highest accuracy on the validation set and minimal fluctuations with only a slight overfit on the training. MSE will be the loss function we currently will continue with.

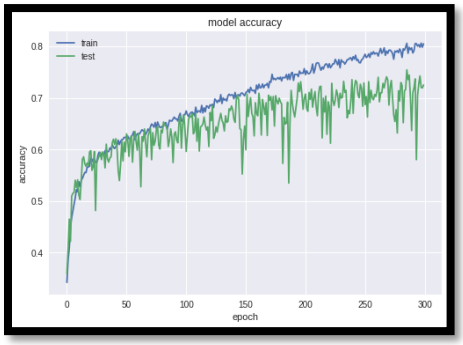
Setting hyper parameters

Tuning hyperparameters for deep neural networks is difficult as it is slow to train a dep neural network and there are numerous parameters to configure.

- Learning rate: Learning rate controls how much to update the weight in the optimization algorithm. We can use fixed learning rate, gradually decreasing learning rate, momentum-based methods or adaptive learning rates, depending on our choice of optimizer in our case we are using SGD, on top of this we can use our history instance we created earlier to pass in as a callback parameter and see our learning rate performs for each different value, this will be set to 0.025 as this is a safe value.
- Activation functions: Activation functions are functions that decide, given the inputs into the node, what should be the node's output. Because it's the activation function that decides the actual output, we often refer to the outputs of a layer as its "activations". One of the popular activation functions in cnn is relu, so we will be using this.
- Epochs: After setting the activation functions, learning rate and setting epochs to 1000 we get the graph below (validation accuracy = 63%)



However, Seeing the accuracy of the model in the graph above, we can see that the model starts to overfit on the training data around 500 epochs with only 63% on the validation set, this can be due to the fact that our model may be too complex, so decreasing the filter sizes and the connecting layer in our network may yield better results.



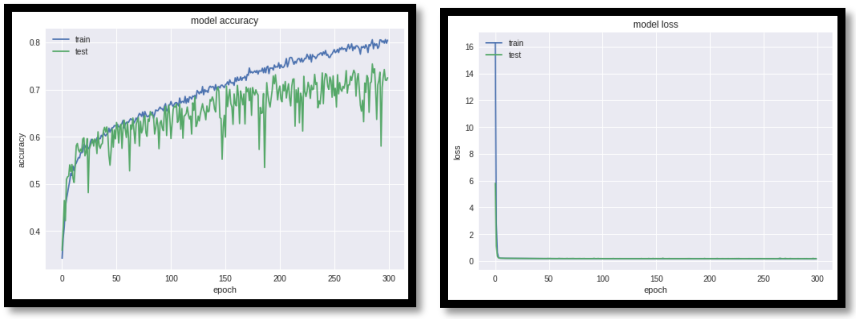
After simplifying the model and increasing the drop out threshold, we get the model above after 300 epochs (approximately 5minutes to compute) with a validation accuracy of 74%. Now testing this model on the given test set we get 80% accuracy as shown below.

```
[32 32  3]
[32 32  3]
[32 32  3]
Accuracy: 80.0000011920929
```

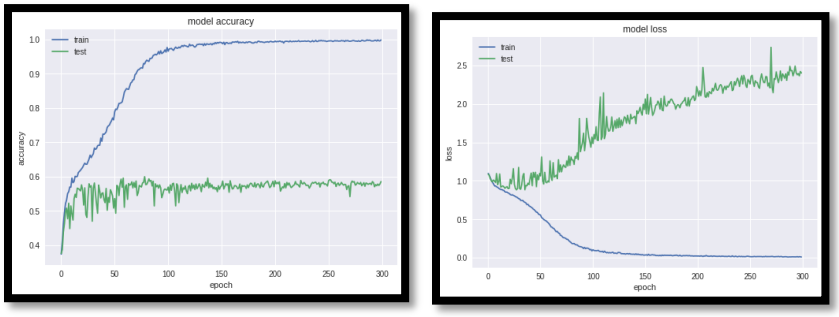
Results discussion

Method	Epochs	Training accuracy	Testing accuracy	Time taken
MLP	300	99.9%	40%	15MIN
CNN	300	81%	80%	5MIN

CNN



MLP



As you can see MLP's validation set only reach's 60% while its accuracy on the training set sharply increases to 100% very quick meaning the MLP model heavily over fits on the training data.

In conclusion the model works well with 80 percent accuracy on the testing set in only 300 epochs that only takes 5minutes to compute, compared to earlier models of CNN where it took 1500 epochs and 30minutes to compute, or compared to using keras flow_from_dir with optimal epochs which can take up to over an hour. The speed of my implementation was due to the fact I used one hot encoding and setting image sizes to 32 by 32. As seeing the CNN accuracy and validation graphs above we can see that there is still more room for higher accuracy.

The final implementation as shown above I still ever so slightly over fitting on the training data with a slight fluctuation in the validation accuracy. Future work may include tuning the parameters further to reduce over fitting and more stability, also using bottle necking or pre-trained models maybe something further to look into. Should the model be able to train on more than 3500 images then this also would increase the accuracy.