CS4287 Neural Computing
Assignment 2: CNNs

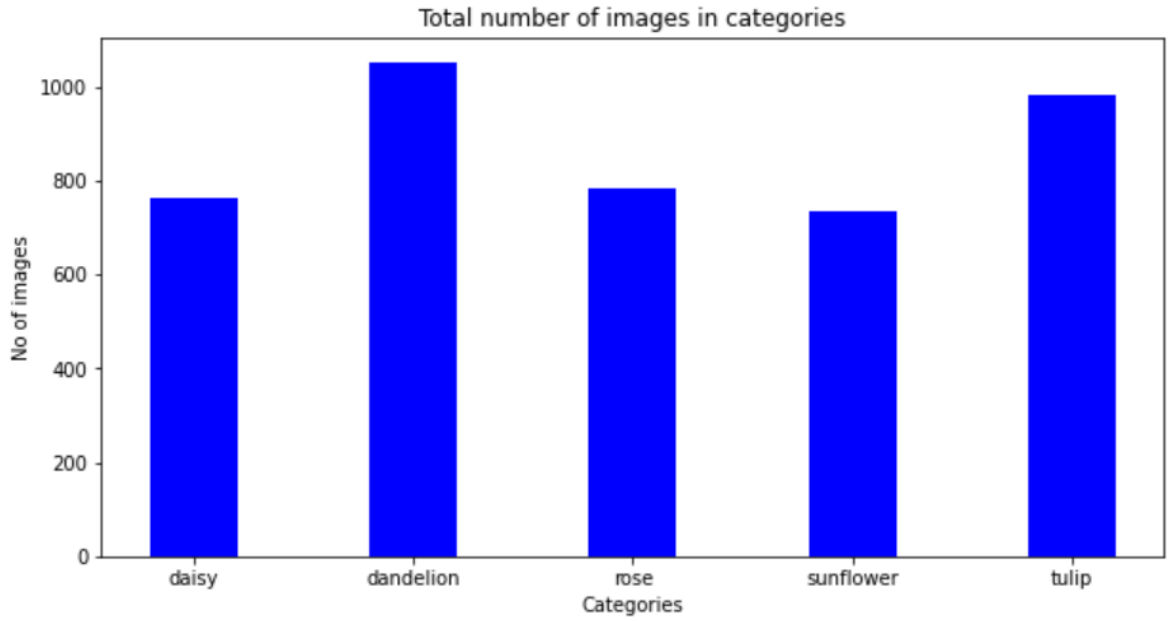
Dylan Kearney 18227023
Cyiaph McCann 17233453

Dataset

https://www.kaggle.com/alxmamaev/flowers-recognition

This dataset contains 4317 images of flowers. All images are divided into five classes splitting up five different types of flowers: Daisy, Rose, Dandelion, Sunflower and Tulip. All images are in .jpg format and have different dimensions.

The distribution of the data is pretty even so we did not have to balance the data before training the model. Distribution can be seen in the graph below.



Correlations
Since we only have the Flower Class to correlate we cannot perform correlation on the data.

Pre-processing
We read in each of the flower images from their relevant folders and place them into a list called training_data. Each element of this list will contain an image array and the class of flowers that the image represents. Since CV2 reads in the images as BGR values we need to convert this to RGB. We also use the CV2 library to resize the image arrays to be 48x48. This is to reduce the size to make the model train faster. We then normalise the image pixel values to be valued between zero and one.

Since the images are read in from the flower class folders, they will be arranged in order from the first class to the last (i.e the first 800 or so images in the list will be Daisies). When the model is trying to fit on the data, it is very likely that it would overfit each flower class before moving on to the next. For this reason, we shuffle the training data before moving on.

Preparing Test and Train sets

First of all, we separate the training data into features (the image arrays) and labels (the flower classes). Then, we reshape the features list to be 48x48x3 to match up with the size of the images - 48 pixels by 48 pixels, each of RGB values hence the '3'. We then convert our label array of integers to a binary class matrix - changing shape from (4315,) to (4315, 5).

Using the sklearn train_test_split function, we divide the features and labels lists into 20% test sets and 80% train sets. We apply a random state of 44 to control the shuffling before the splitting.

Cross Fold Validation (3 fold)

Cross Fold Validation is a method to measure the skill of machine learning models. Cross Fold Validation is used to evaluate models using a limited data sample by resampling. This will give us an estimated prediction on how the model is expected to perform when given unseen test data. Seen below is where we split our data in order to perform cross-fold validation in our model.

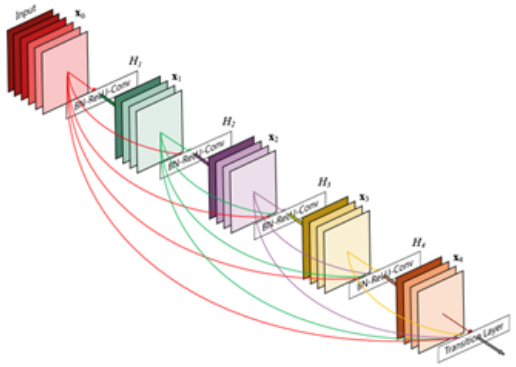
```
trainx, testx, trainy, testy = train_test_split(X,
                                                 y,
                                                 test_size=0.2,
                                                 random_state=44)
```

```
hist = model.fit(datagen.flow(trainx,trainy,batch_size=32),
                 epochs=10,
                 validation_data=(testx,testy)
)
```

Network

DenseNet 201

A DenseNet is a type of convolutional neural network that uses “Dense Blocks” to connect all layers with matching feature-map sizes directly.



[Fig 1.]

A Dense Layer uses the output from convolutional layers to classify images. A DenseNet is a form of CNN that uses Dense Blocks as seen above to through dense connections between layers. We thought that DenseNet would be a good choice as it reduces the vanishing gradient problem and reduces the number of parameters. As a result of this, it has been shown to have better feature-use efficiency and can outperform Resnet.

DenseNet-201 is a CNN with 201 layers. It is possible to load pre-trained weights of the network that has trained images from the ImageNet database. This pre-trained network can class images into a thousand different object categories and has learned representations for a wide range of images. We altered the network to have an image input size of 48x48 due to the time it was taking to fit the model with the original images scaled size of 224x224.

Weight Initialisation

When setting up the DenseNet201 model, we opted to initialise the weights from the ImageNet database. This helped us to speed up the time it took to perform the CNN. Since ImageNet has learned many representations for images, we believed that it would perform well with our data.

Batch Normalisation

Batch Normalisation is a method used to normalise the output of previous layers within a Neural Network. This is done by normalising the activation vectors from the hidden layers in a network. It uses the Mean and Variance of the current batch to perform this normalisation. The first calculation done by Batch normalisation is to determine the mean μ and standard deviation σ using the formulas below.

$$\mu = \frac{1}{n}\sum_i Z^{(i)} \qquad \sigma = \frac{1}{n}\sum_i (Z^{(i)} - \mu)$$

The next step is to normalise the activation vector Z to ensure that each output from the neurons has a normal distribution.

$$Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 - \epsilon}}$$

It then computes the output of the layer using linear transformation with two trainable parameters. Through trial and error of changing these two parameters, the model will be able to find the best distribution for each hidden layer. Adjusting γ will change the standard deviation and adjusting β will change the bias.

$$\check{Z} = \gamma * Z^{(i)}_{norm} + \beta$$

These formulas were taken from https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338#b93c

DenseNet uses Batch Normalisation as the third of its 4 layers with the other three being convolution layer, pooling layer, and non-linear activation layer in that order. Below is a segment of our *model.summary()* displaying the use of the BatchNormalisation layer in our CNN.

```
conv2_block1_0_relu (Activatio  (None, 12, 12, 64)  0           ['conv2_block1_0_bn[0][0]']
n)

conv2_block1_1_conv (Conv2D)   (None, 12, 12, 128)  8192        ['conv2_block1_0_relu[0][0]']

conv2_block1_1_bn (BatchNormal  (None, 12, 12, 128)  512        ['conv2_block1_1_conv[0][0]']
ization)
```

Regularisation

Regularisation is a type of regression. It generates constraints on the coefficient estimates bringing them towards 0. This method is used to minimise the risk of overfitting the model so that it generalises better. This will improve the performance of the model specifically on the test set. Using regularisation will dramatically reduce the variance of the model without affecting or increasing the bias significantly.

L1 and L2 Regularisation:

These are the most used types of regularisation and they work by adding the term known as the regularisation term to the overall cost function, which updates it.

Cost function = Loss (binary cross entropy) + Regularisation term

Because it is assumed that a NN with lower weights leads to simpler models, the values of weight matrices decrease when this regularisation component is added. As a result, it will significantly reduce overfitting.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

As seen above, the L2 where lambda is a hyperparameter that represents regularisation.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

Above is L1 where the weights can be reduced down to 0. This is useful for the compression of the model.

Another technique used to apply regularisation is Dropout. As the name suggests dropout will randomly select nodes within the model and it will remove them, this includes all of the related connections to do with that node. A dropout model will generally perform better than a standard model since each iteration is randomised.

We initially investigated using regularisation techniques however we discovered that densenet doesn't use this technique in its structure.

Transfer Learning

Transfer Learning occurs when we take a model that has been trained on a very large dataset and we transfer the knowledge gained onto a smaller dataset. The information that is learnt from the previous model can be used to better generalise another model. In our case, we were able to use ImageNet to set the weights of our model. This is considered a

pre-trained model approach as we are using data from a model that has already been trained.

Activation Functions

ReLU

We chose to use the ReLU activation function for the middle layers of our DenseNet neural network. ReLU is an activation function that acts like a linear function but is actually a nonlinear function that allows complex relationships in the dataset to be understood.

The ReLU (Rectified Linear Activation) is a linear function that, if the input is positive, outputs the input directly, otherwise it outputs zero. A model with ReLU as the activation function is quicker to train and generally produces higher performance. We thought about using the sigmoid activation function but we found out that it cannot be used in neural networks that have many layers due to the vanishing gradient problem.

Softmax

Softmax is a type of logistic regression that converts an input value into an array of values of probability distributions. These values have a sum of one. The output values are between 0 and 1. This is convenient as it allows us to avoid binary classification and fit as many classes as we like for our neural network model. For our output, we had 5 different flower classes so we used softmax as our output activation function.

Loss Function

Categorical Crossentropy

Categorical crossentropy is a loss function that helps the model to learn to give a high probability to the correct class (in our case, a flower class) and a low probability to the others. It compares predictions with the ground truth labels. It produces a one hot vector with the probable match for each class. Below is how the loss is calculated with Categorical crossentropy.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

When we first made our model, since it is a multi-class classification task, we chose Categorical Crossentropy as the loss function. We were getting some poor results (around 70% loss on the validation data). When using softmax as the activation function, it is recommended to use categorical cross-entropy as the loss function.

Mean Squared Error
Mean Squared Error or MSE is a function that is used to calculate the loss of a model. This is done by taking the difference between the model's results (predictions) and the actual expected results. This value is then squared and is averaged out across the entire data set.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

The equation for calculating the MSE:
N represents the total number of samples we test against. We chose to use this function to calculate the loss as it is great for mitigating any outlier predictions that may have significant errors.

We used MSE as the loss function of our model in order to estimate the loss of the model so that the weights could be changed to reduce the loss. MSE is non convex for binary classification problems and since it works well on multi-classification, we decided to use it as our loss function.

Sparse Categorical Crossentropy
Sparse Categorical Crossentropy is similar to Categorical Crossentropy in that they both compute categorical cross-entropy. The only difference is the way the targets are encoded. It produces the index of the most probable matching flower class. Sparse Categorical Crossentropy is more efficient for use with datasets with a lot of categories.

We attempted to use Sparse Categorical Crossentropy as our loss function but it did not match the shape of our data. Since our labels were of a binary class matrix (i.e. shaped like (4315, 5)) it did not match the expected shape.

Optimizer

Adam
Adam optimization algorithm is an extension to stochastic gradient descent. Adam is used to updating the network's weights. Adam is described by the authors to be a combination of two stochastic gradient descent extensions. The two are Adaptive Gradient Algorithm and Root Mean Square Propagation. Adam will calculate the exponential moving average of the gradient. The decay rates of these moving averages can be manipulated and controlled by adjusting the parameters beta1 and beta2.

Using Adam on our model yielded the highest levels of accuracy. Giving us 70.5% accuracy compared to SGD returning 70% and RMSProp returning 67%. After some research into comparing Adam to SGD, we learnt that Adam is much faster than SGD so we decided to use this as our optimisation function (Zhou et al., 2021).

Impact of Varying Hyperparameters & Data Augmentation

Changing the Loss Function

We experimented with different loss functions. When we first used Categorical crossentropy as the loss function we were getting about 70% loss on the validation data. After changing to MSE we got about 7-8% loss on the validation data.

ImageDataGenerator

As our model was overfitting on the data, we decided to perform some data augmentation in order to solve this issue. We used the Keras ImageDataGenerator to create tensor image data in batches with real-time random data augmentation.

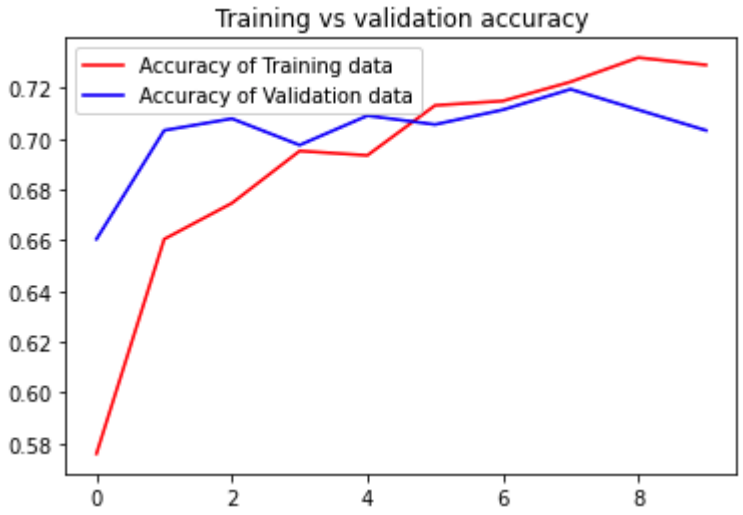
```
datagen = ImageDataGenerator(horizontal_flip=True,vertical_flip=True,
                             rotation_range=20,zoom_range=0.2,
                            width_shift_range=0.2,height_shift_range=0.2,
                             shear_range=0.1,fill_mode="nearest")
```

This will flip images vertically and horizontally, rotate, zoom, and perform other image augmentations. When fitting the model, we pass this in, giving it the training x and y sets in order to perform the augmentations on the X training set (X contains the image arrays).

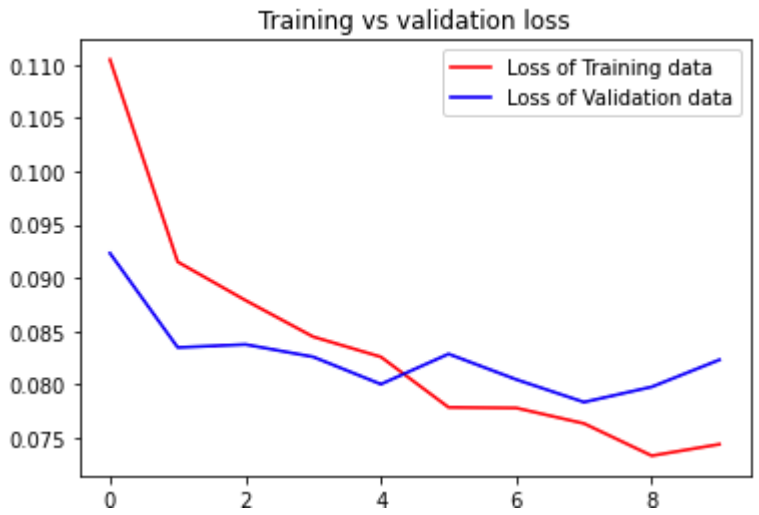
Results

Plot Model History over the Epochs

We plotted how the accuracy changed over the Epochs on both the train and validation data.



Next, we plotted the validation set loss over each Epoch.



We made sure that in each of these plots we plotted the accuracy and loss on the validation data. This gives the indication if the model is overfitting or underfitting on the training data. When these lines are compared with that of the training data lines, it is easy to spot issues with fitting.

Top 5 Accuracy, Precision, Recall, F1-Score, Support

	precision	recall	f1-score	support
0	0.79	0.67	0.73	150
1	0.65	0.86	0.74	188
2	0.60	0.70	0.64	171
3	0.83	0.66	0.73	154
4	0.76	0.62	0.68	200
accuracy			0.70	863
macro avg	0.72	0.70	0.70	863
weighted avg	0.72	0.70	0.70	863

The precision, recall, f1-score and support for each flower class is listed above.

Precision = Calculates the ratio between the number of positive images correctly classified and the total number of images classified as positive.

Recall = Recall represents the ratio between the number of positive images correctly classified as positive and the total number of positive images. Essentially measuring the model's ability to detect positive images in the dataset.

F1-Score = The F1 score will combine both precision and recall relative to a specific positive class. This value can be viewed as a weighted average of the two. 1 represents its best value and 0 represents its worst.

Support = Support value represents the actual number of occurrences of the class in a specified dataset. An imbalance here could mean a weakness in the model's scores, this could indicate a need for rebalancing.

Evaluation of Results

Evaluate the Model using the Test Set

```
[ ]  score = model.evaluate(testx, testy, verbose=1)
     print("loss = " + str(score[0]*100) + "% \naccuracy: "+ str(score[1]*100) + "%")
```

```
27/27 [==============================] - 2s 69ms/step - loss: 0.0823 - accuracy: 0.7034
loss = 8.232889324426651%
accuracy: 70.3360378742218%
```

After the model was finished fitting, we evaluated it on the test sets. From the image above, you can see that we achieved about 70% accuracy with ~8% loss. We were satisfied with the results as there was a lot of computational constraints when developing the model. We had to use images with a size of 48 by 48 pixels and limit the number of epochs to 10.

Overfitting

At first, the model was overfitting on the training data. This was visible by observing the validation loss to see if it is increasing. We solved this issue by using Image Augmentation as described above. We also could have applied Regularisation techniques to attempt to solve the overfitting but we were satisfied with the result after performing the Image Augmentation.

We also shuffled the training data so that the model wouldn't overfit any particular flower class.

References

[Fig 1]: https://www.pluralsight.com/guides/introduction-to-densenet-with-tensorflow

Zhou, P., Feng, J., Ma, C., Xiong, C. and HOI, S., 2021. Proceedings.neurips.cc. Available at: <https://proceedings.neurips.cc/paper/2020/file/f3f27a324736617f20abbf2ffd806f6d-Paper.pdf>.