Machine Learning Engineer Nanodegree

Capstone Project

Khang Vu
April 7th, 2019

I. Definition

Project Overview

Background

Consider the case when we are passionate about flowers, and we are curious at the same time about what type of flower they are, what name people usually call them, etc.

What if we can do just that with a little help from technology? Imagine we can use our own phone to take pictures of the flowers, and right away, its name appears afterwards on the screen, and we feel satisfied. Here comes a use case where we can apply Machine Learning (ML) algorithm to make predictions.

This classification use case is one of the problem hosted by `Kaggle`, where:

> Kaggle is a platform for predictive modelling and analytics competitions in which statisticians and data miners compete to produce the best models for predicting and describing the datasets uploaded by companies and users. This crowdsourcing approach relies on the fact that there are countless strategies that can be applied to any predictive modelling task and it is impossible to know beforehand which technique or analyst will be most effective.[Kaggle]

Click here for more information in Kaggle, about `Oxford 102 Flower Pytorch – 102 Flower Classification Created by Enthusiast's` competition. Even though the competition requires a solution in Pytorch, we will instead use Keras in this project.

References

- Kaggle

Purposes and Motivation

The main goal for this project is to create an intelligent Machine Learning (ML) model using different techniques to help us recognize most of the common flower types. And this model could be used to integrate into mobile apps, devices to help predict and open more opportunities for developers to innovate, especially if they are passionate about flowers.

This project will be very helpful and diverse in technical term, by using various ML techniques from Supervised Learning, Exploratory Data Analysis (EDA), to Deep Learning, etc. Apart from that, recognizing objects has been an interesting topic in recent years since it can make our applications smarter by learning and making predictions by themselves in different categories without being explicitly programmed, which is interestingly motivated to put in the efforts.

Datasets and Inputs

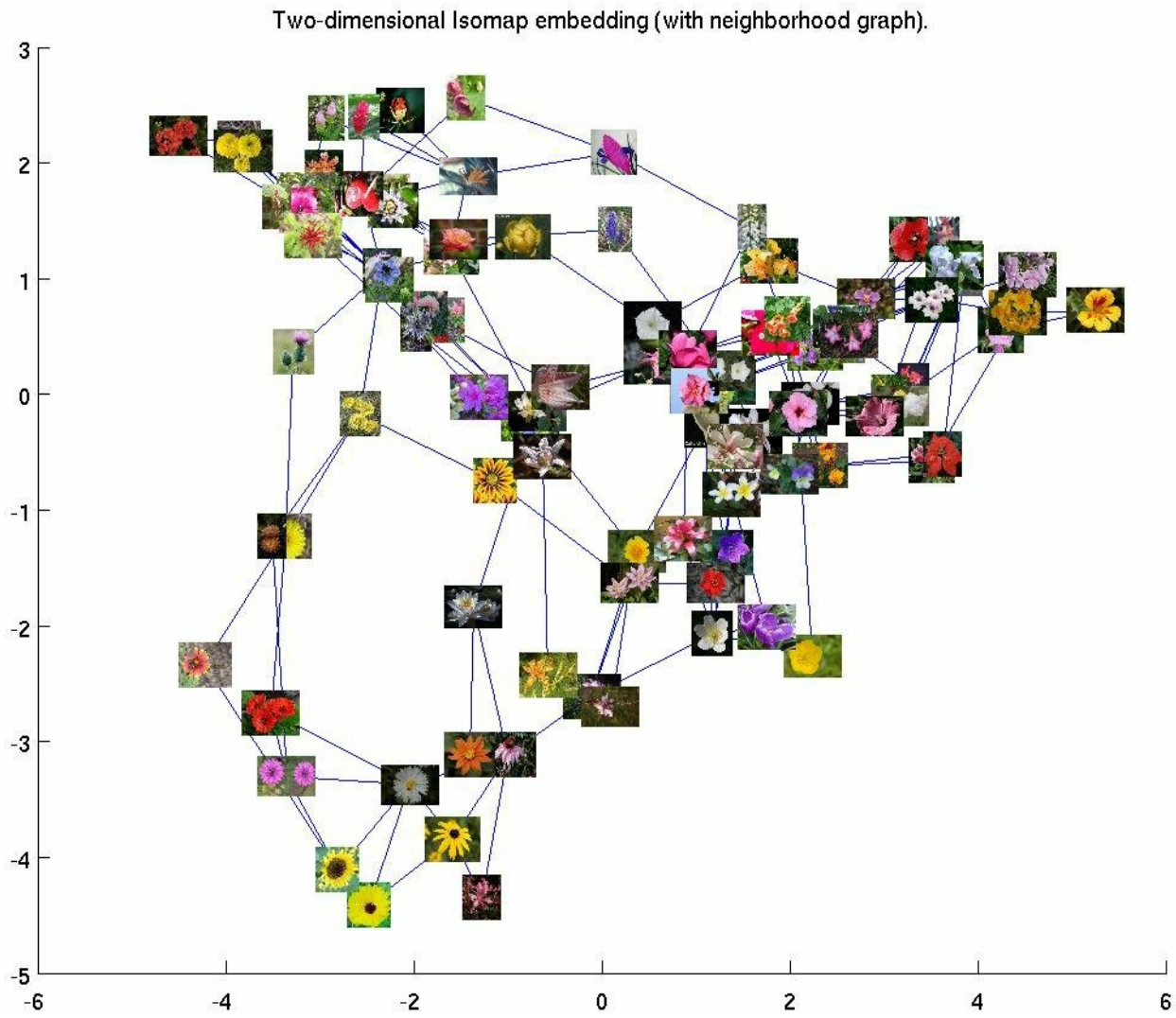
There are 102 flower categories commonly occurring in the United Kingdom. *Maria-Elena Nilsback* and *Andrew Zisserman*, in *Department of Engineering Science* at the University of Oxford, have decided to create a dataset, corresponding to the aforementioned 102 flower categories, or so-called classes interchangeably. In details, each class consists of 40 to 258 images. Visualization about each class (name, image, label number, etc.) can be found at this site.

According to Visual Geometry Group at the University of Oxford:

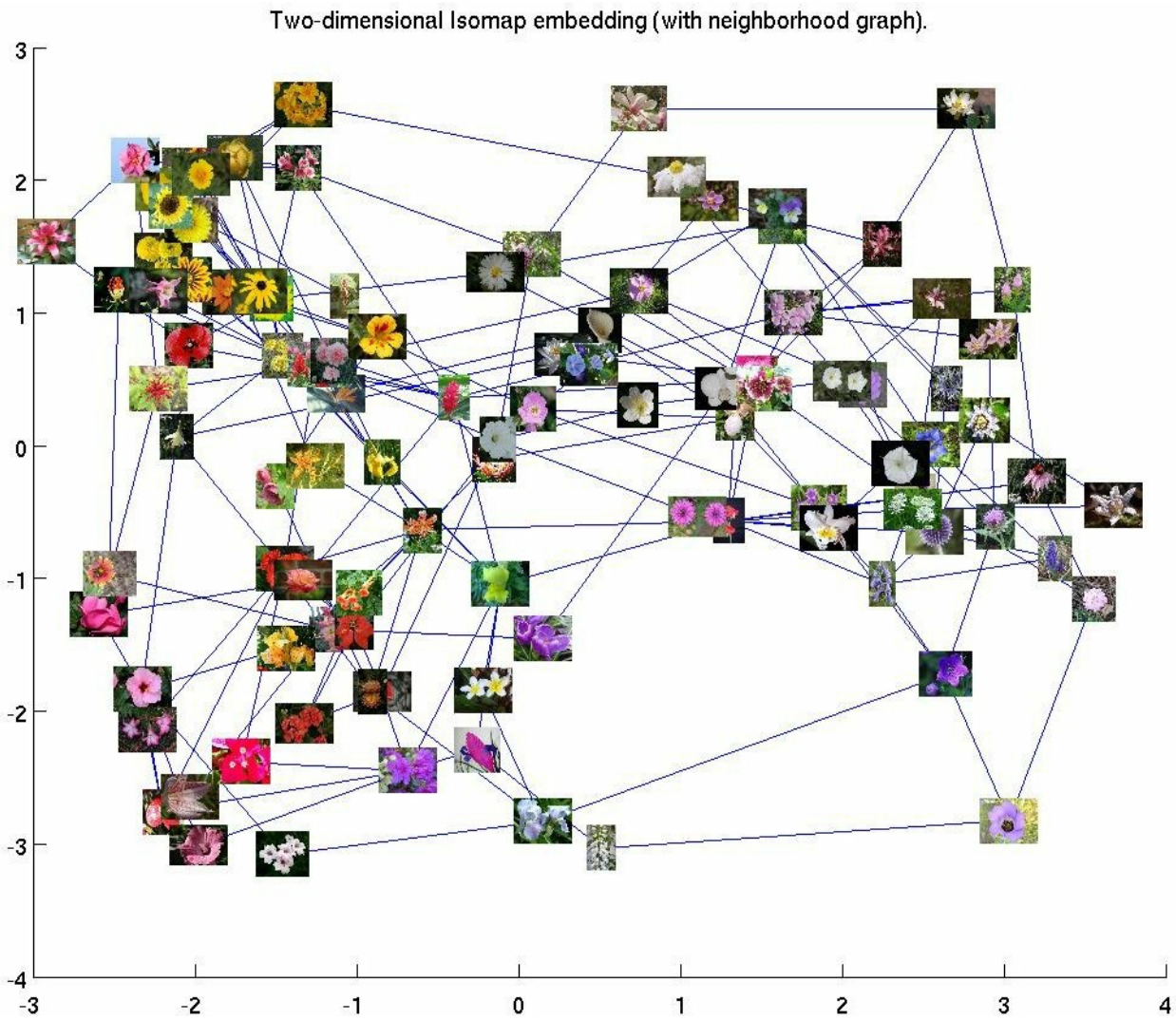
> The images have large scale, pose and light variations. In addition, there are categories that have large variations within the category and several very similar categories. The dataset is visualized using isomap with shape and colour features.(4)

And the following are the `Isomap` for `Shape` and `Colour` for the flowers provided in Visual Geometry Group:

Shape Isomap



Colour Isomap



References:

- http://www.robots.ox.ac.uk/~men/
- http://www.robots.ox.ac.uk/~az/
- http://www.robots.ox.ac.uk/~vgg/data/flowers/102/categories.html
- http://www.robots.ox.ac.uk/~vgg/data/flowers/102/

Problem Statement

Quantifiable

Given a batch of different types of flowers, and we want to classify them by matching the flowers to their corresponding type names. In order to figure out the corresponding names for the flowers, we can calculate the probabilities for each classes represented by an output layer from a Deep Neural Network, which should produces the maximum likelihood of those classified names.

> A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship. The network moves through the layers calculating the probability of each output.(5)

From there, we can predict the corresponding name for the image by taking the class with largest probability within 102 output predictions. These outputs are represented in form of probabilities because we use a Softmax function to calculate probabilities distribution across our classes.

> In mathematics, the softmax function, also known as softargmax or normalized exponential function, is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities.

Measurable

There are few metrics we can use to measure our model performance such as `Accuracy`, `Precision`, `Recall`, and `F1-score`. `Accuracy` is pretty much a straight forward metric which will give us an idea on how much accurate our model can predict, by observing the percentage of how many images being correctly classified out of 102 flower types. However, in some cases, such as for an unbalanced dataset, the other three metrics can be used to measure predicting performance better. Our dataset is on the edge to be unbalanced (ranging from 40 to 258 images per classes), even though not really significantly. Hence, so we will use all four of these metrics for measurement to be safea and more convinced.

Replicable

This classification problem should be reproducible by taking images of different flowers and making predictions accordingly again and again.

Solution Statement

Firstly, we need to make sure our dataset is clean by pre-processing it using various Exploratory Data Analysis (EDA) concept. Then we will be using a Deep Neural Networks (DNN) at the core implementation to train our data, and calculate the probabilities as final output after making predictions, which will potentially tell us the flower types. We will also use Image Augmentation technique to vary the input types so that the network can learn better in terms of diversity, as part of data pre-processing step. We then use the aforementioned metrics to see how accurate our model performs after training.

> Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis

> and to check assumptions with the help of summary statistics and graphical representations.(6)

After all, developers can make use of CoreML (iOS) or ML Kit (Android) to convert and incorporate this ML model into their platforms and start making predictions right on their respective devices as real applications.

References:

- Softmax function - wikipedia
- Deep Neural Network - wikipedia
- https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15
- https://developer.apple.com/machine-learning/
- https://developers.google.com/ml-kit/

Metrics

As mentioned in Problem Statement, we will be using `Accuracy` to measure our model's performance along with some other metrics. Because in some cases, `Accuracy` is not enough to measure our performance properly and we need to use F-beta score (F-1 to be exact, for beta is 1 to shift the result towards `Precision` side) by determining whether we need a high `Recall` score (if the cost for `False Positives` is high) or a high `Precision` score (if the cost for `False Negatives` is high).

The four metrics can be calculated as following:

```
Accuracy = (True Positives + True Negatives) / 102 (dataset size)
Precision = True Positives / (True Positives + False Positives)
Recall = True Positives / (True Positives + False Negatives)
F1-score = (2 x Precision x Recall) / (Precision + Recall)
```

where:

- True Positives is number of images *X*, which are correctly classified / predicted as *X*.
- True Negatives is number of images which are predicted as not *X* and actually not being *X*. This is almost similar to True Positives.
- False Positives is number of images that the model thinks is *X* but actually is some other classes *Y* or *Z*.
- False Negatives is number of images which are predicted as some other flower classes *Y* or *Z* rather than its actual flower label *X*.

Hence the formula can be shortened as:

```
Accuracy = True Positives / Total number of samples
Precision = True Positives / Total predicted positives
Recall = True Positives / Total actual positives
F1-score is the same as above
```

Optionally, we can multiply the result by 100 to turn it into percentage format, if necessary, for displaying purposes.

As in our classification problem, what we really need from the solution model is to make sure we can correctly classify the flower's names as many as possible. Hence, we expect our model to have high `Accuracy`, and also high `Recall` as well. For `Precision` and `F1-score`, we can observe the results and compare to the other two since this is a unique flower classification problem, for `Precision` should be similar to `Recall` case. So is F1-score as it it built based on `Precision` and `Recall`.

II. Analysis

Data Exploration

The dataset contains a good amount of different flower types (102), with a number of images represent each class.

Within this dataset, we have two folders *Training*, *Validation*, which stand for their own purposes, respectively. Each folder should contains 102 categories, and each category contains 40 to 258 images, so it is quite a bit unbalanced but not too significant since we have at least 40 images per class.

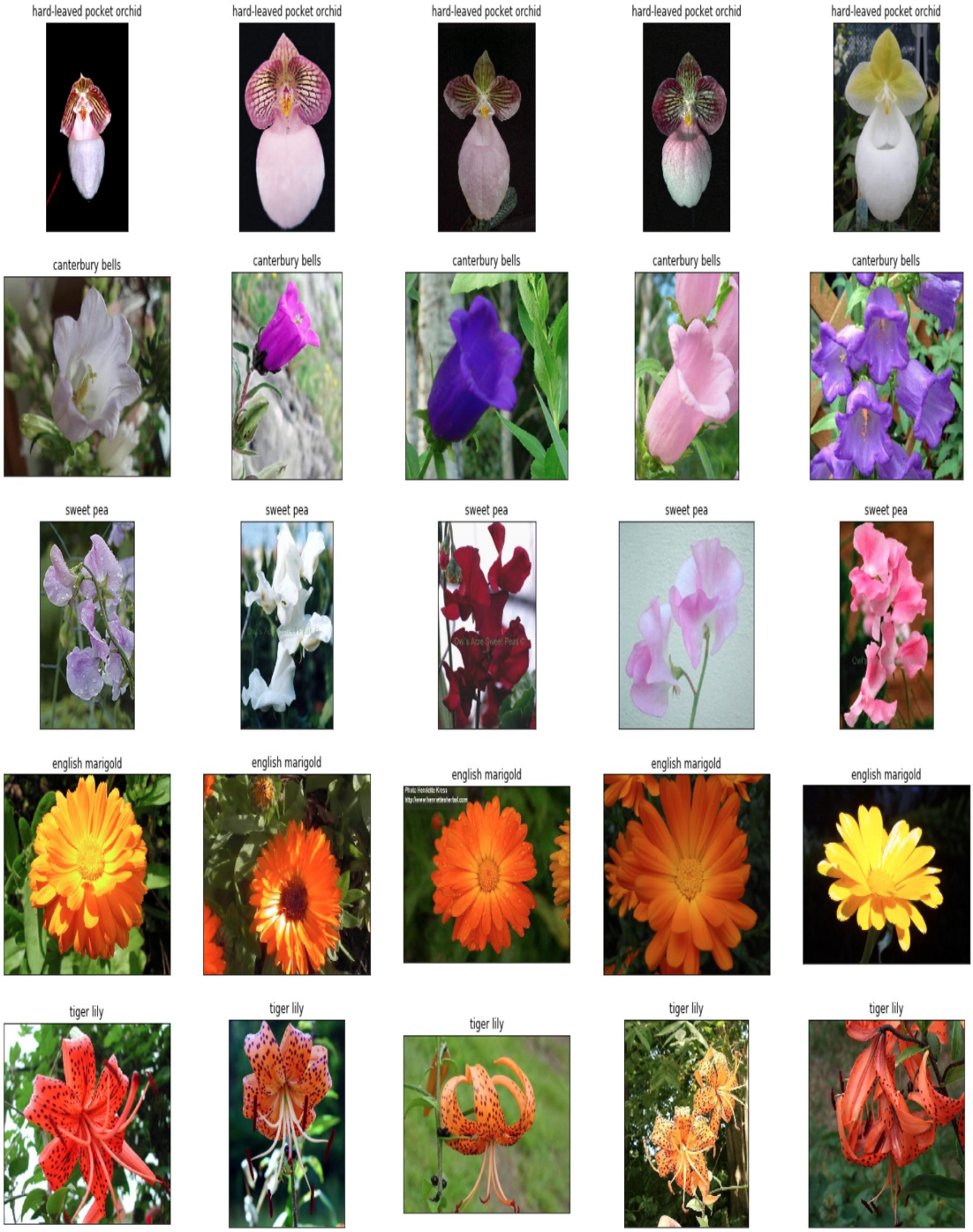
Training folder is used for training our model. While loading in images for training folder, we will split up the images dataset into 90-10% fraction and use the 10% of dataset to create a validation generator. This will help prevent overfitting problem since the data using for validating the model while training is totally different from the validation set that we loaded in originally. *Validation* set contains of 609 images, while *Training* set contains 5943 images after splitting up training data.

Then we use the original *Validation* dataset to test our model after training since this dataset has stayed intact the whole time and will play a role as unseen data. In other words, we need this *Validation* set in order to verify the accuracy of our model after training so that our model does not learns too deep into the testing dataset and unable to predict accurately new unseen

data (Overfitting).

Exploratory Visualization

The following screenshot shows five different flower types, with five images per each category (per row):



Our model will extract the basic characteristics from the images and try to learn patterns from

its features, then combine these knowledges to learn even more common complex patterns between the same type of flower and predict the likelihood of flower types eventually as a goal.

Algorithms and Techniques

In order to solve this flower classification problem, a Deep Learning architecture incorporated with Transfer Learning technique will be used to learn patterns, extract feature characteristics from images and then combine these knowledges to come up with the most likelihood probabilities for the predicted flowers. We, hence, use this result to classify our flower categories.

Before developing our own Deep Learning architecture, we need to make sure our problem is solvable and can be improved, and fine-tuned later. A simple benchmark model will be used in this case as a starting point, which will be discussed in the next section.

After comparing with the aforementioned benchmark model, we now can use Transfer Learning technique to benefit the pre-trained model (VGG-19) by using its *pre-trained weights* to continue training and fit our own problem.

Parameters

A pre-trained model like VGG-19 will need mandatory parameters such as:

- *weights*: '*imagenet*' - is chosen in this case, since we want to reuse the pre-trained weights through 1000 images in imageNet database.
- *include_top*: '*False*' - is set so we can ignore the output trained from imageNet and use our own custom output layers.
- *input_shape*: this parameter specifies an expected size to feed in VGG-19 network (224 width x 224 height x 3 color channels).
- *loss*: '*categorical_crossentropy*' - since our problem is multiclass classification, this parameter is proved to produce more accurate result then '*binary-crossentropy*' option.
- *optimizers*: '*Adam*', and *Learning rate*: '*0.00005*'' - this optimizer and learning rate combination has been experimented to reach the optimal result better, even though it takes quite some times to converge to a local minimum (with help from GPU power). Especially, VGG networks have been known as having a long time to sucessfully train.

Strategy (Algorithm)

Since our dataset is considered as a small dataset (less than 10,000 images), and quite similar to imageNet database, we need a strategy to make sure we train the pre-trained model in an optimized way to fit our problem. For this scenario, we will `freeze` all the pre-trained layers in the original network. In other words, we don't train the original layers again, but using the pre-trained weights from imageNet instead.

Doing this will allow us to add new extra custom layers in place of the original output layer, so that the model can produce the desire result uniquely to our flower problem.

After adding new custom layers, we are then ready to train this new architecture network. And since we already froze all the original layers from the pre-trained network, when we train this new entire network again, the original layers' weights will not get updated in the process, but only in the extra layers. This way we can purposely train our extra layers to make sure they gain enough basic knowledges about this classification problem.

Until this point, we will discuss more about two approaches, which have been tried out for this project, in the section `Implementation and Refinement` below. In high level, after a preset number of `epochs` of training, when this new model is supposed to learn some new knowledges, we will un-freeze a few layers in the original network and let the entire network be trained again for another amount of epochs. This time we should be able to get a model with decent enough accuracy to make predictions.

Note:

- `epochs`: number of times we do a forward pass and a backward propagation to update the weights in each layers.

- `forward pass`: this procedure happens when we feed an image into our DNN from the first layer and forward pass it through each layers until the last layer in the network to get the features extracted along the way.

- `backward propagation`: this procedure happens when the errors have been calculated from the last layer by using a *loss function* (`Cross Entropy` in this case) and passed backward to update the weights in each layers.

Benchmark

It is a good idea to setup a benchmark model, which plays a role of making sure that our classification problem is solvable and giving us an idea on how this problem can be solved with just a simple model architecture. Later on, we can use this result to have confidence that a DNN using Transfer Learning will definitely produce a better result.

This simple benchmark model only includes one Convolutional layer (2D), and one Dense layer as core layers. Then we add a Pooling layer to help reduce space info to filter out features, and a Dropout layer to give each layer a chance to train in the network. Lastly, we need a Dense layer as an output layer to make prediction using a `Softmax` activation function to make sure the results are in form of probabilities. The architecture is as following:

```
bench_model = Sequential()
bench_model.add(Conv2D(256, kernel_size=2,
  input_shape=(224, 224, 3), activation='relu'))
bench_model.add(GlobalAveragePooling2D())
bench_model.add(Dropout(0.5))
bench_model.add(Dense(128, activation='relu'))
bench_model.add(Dropout(0.2))
bench_model.add(Dense(train_generator.num_classes,
  activation='softmax'))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1_input (InputLayer)  (None, 224, 224, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 223, 223, 256)     3328
_________________________________________________________________
global_average_pooling2d_1 ( (None, 256)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 102)               13158
=================================================================
Total params: 49,382
Trainable params: 49,382
Non-trainable params: 0
_________________________________________________________________
```

After five epochs, this benchmark model gives an accuracy of `3.48%`, which is very bad but as expected, since this model is too simple to learn enough patterns because it has not been trained to learn foundation patterns enough like in pre-trained model.

We will definitely need another model with more knowledge on lower levels, to extract the foundation features out from the given images, using a pre-trained model such as VGG-19, which has already been trained on ImageNet through 1000 images, so it knows the basic features to learn higher level patterns. This way we can modify this pre-trained model using Transfer Learning technique to solve our unique problem with customized Dense layers for a better result.

III. Methodology

Data Preprocessing

For this classification problem, in order to make our dataset more diverse in terms of shape and uniqueness, we make use of Image Augmentation technique, where various transformations will be applied to each images before loading in for training. For example, some transformations such as rotation, scale, horizontally flip, zoom, etc. can be used to generate more data to feed into our model. This way, we can have more data to train with and make our model learns the features better for each image category (since each can be learned in different angles).

Other than applying transformations, the data images also need to be converted into shape of (width: 224, height: 224, color channels: 3) because the pre-trained model like VGG-19 (or VGG in general) will expect to have input shape of these dimensions.

In `Keras` APIs, we can perform `Data Augmentation` technique appropriately by importing:

```
from keras.preprocessing.image import ImageDataGenerator
```

and for `preprocessing`:

```
from keras.applications.vgg19 import preprocess_input
```

In the process of loading images from given input directories, the above two techniques will be used to load and prepare the data accordingly so that they are ready for training later. The results from this procedure give us `Image Data Generators`, which hold all the preprocessed images that are ready to be used.

Important Note:
There are two parameters that need to be mentioned before training, which are related to the dataset we loaded in earlier: `train_step_size` and `valid_step_size`. These two parameters are calculated based on the number of images in the respective folders divided by the number of images per batches specified in each Image Generator using for those respective folders.

```
## Step sizes to fit and train our model later
train_step_size = train_generator.n // train_generator.batch_size
valid_step_size = valid_generator.n // valid_generator.batch_size
```

At this moment, the preparation step is done and our data images are ready to be used for training process.

Implementation and Refinement

Overview

In order to solve this classification problem, there are many approaches/strategies we can use use to train our model. In this project, we will try two different approaches to help us reach the optimal performance:

1. The first and most trivial approach is to freeze all the original Convolutional layers in the original pre-trained network and add some extra custom layers in place of the output layers (last/top layer in the original network), then let it be trained after about `30 episodes` to fully make use of the pre-trained weights from ImageNet. At the beginning, the validation accuracy for this approach only reaches about `60%`. The parameter at this time was:

    - Batch size: `32`
    - Optimizer: `Adam`
    - Learning rate: `0.001` (or `1e-3`)
    - Epochs: `20`

Afterwards, I changed the above combination to:

- Batch size: `64`
- Optimizer: `Adam`
- Learning rate: `0.00001` (or `1e-5`)
- Epochs: `30`

With this new set of parameters, the model has been recorded to produce a result around `80%`, which is not a bad result since there was some improvements compared to the previous one, but also not the best performance at the same time.

Note: we use 30 episodes in this case because the model has been observed that it stops improving for both the training and validation accuracy afterwards.

2. Another approach we can try, which has been used for this project as final solution, is to also freeze all the Convolutional layers in the original pre-trained network and also add extra custom layers just like in the above approach. However, we only let it be trained for about `10 episodes` to give the custom layers a chance to learn about the data. Then after

these `10 episodes`, we will un-freeze the last few layers in the original layers and let the model learn again. This time we leave it run for another `20 episodes` and the result has been recorded to stay stable afterwards. `90-92%` has been recorded consistently after a few times re-training from scratch.

Note: this result has been recorded with the same set of parameters as above:

- Batch size: `64`
- Optimizer: `Adam`
- Learning rate: `0.00001` (or `1e-5`)
- Epochs: `10` for the first time, `20` for the second time.

Discussion

In the following, we will discuss in details about when and why the second approach is useful and has been chosen as final solution strategy.

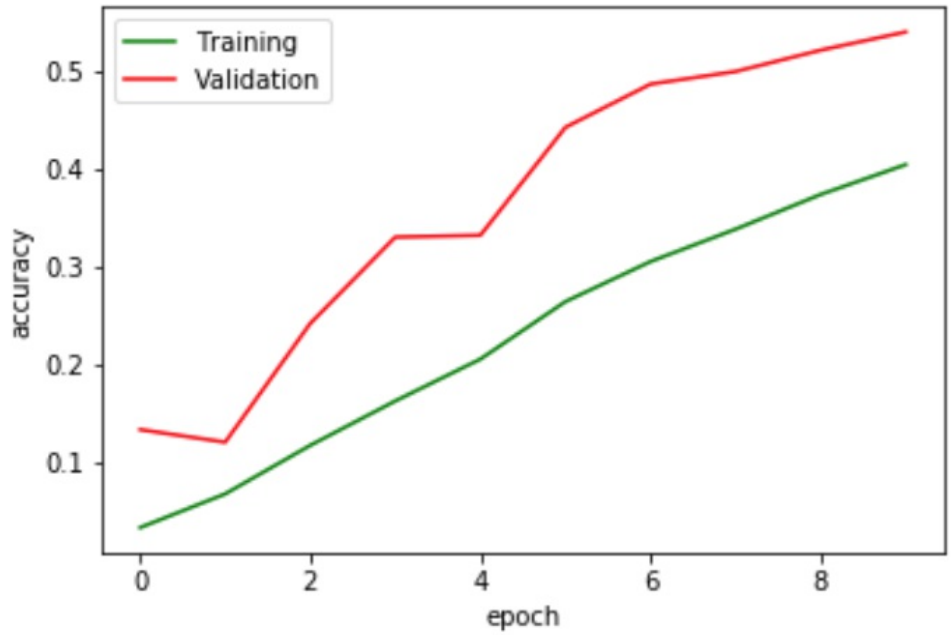
In details, the `include_top` parameter, when the VGG-19 network is initialized, makes sure we exclude the original output layer. In order to fit to our problem, we need to create few custom layers to replace the top layer (output layer) from the original network. A few `Dense` layers will be added as well as some `Dropout` functions to avoid Overfitting. It will look like following:

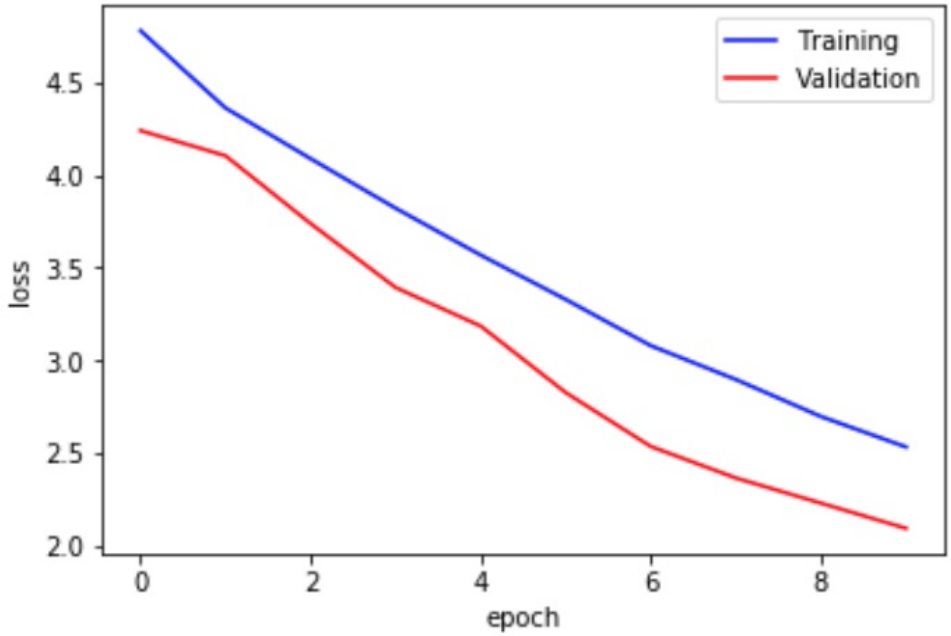
```
## Freezing all the layers from the original network
for layer in base_model.layers:
    layer.trainable = False

# Add some custom layers
net = Flatten(name='flatten')(base_model.output)
net = Dense(4096, activation='relu')(net)
net = Dropout(0.5)(net)
net = Dense(4096, activation='relu')(net)
net = Dropout(0.5)(net)
net = Dense(train_generator.num_classes,
            activation='softmax')(net) # 102 output classes
```

This way, once the training process has started, the spending time is mostly to train the newly added custom layers while making use of the existing weights for the original layers, which gives our custom layers chances to learn our dataset to some extend. I mentioned `to some extend` because this entirely new architecture does not need to train for too long (just about 10 epochs and then stop).

At this point, after `10 epochs` of experimentation, the model has been noticed to produce an accuracy of `65%`, not the best performance but it does show that the learning progress has good potential to learn even more and better if we keep training since both *Training* and

Validation accuracies are simultaneously increasing with a consistent distant gap in between, as following:



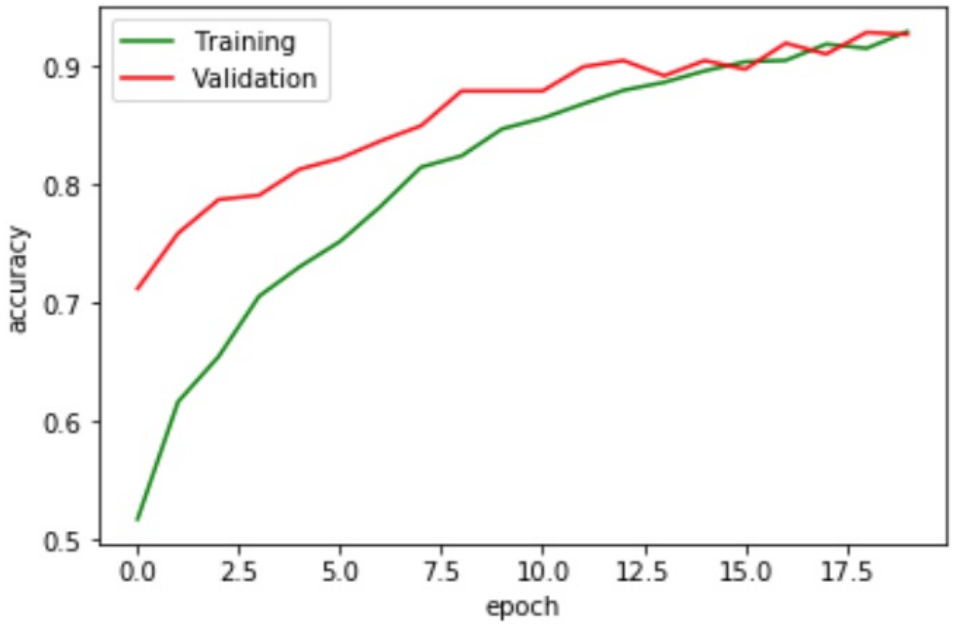


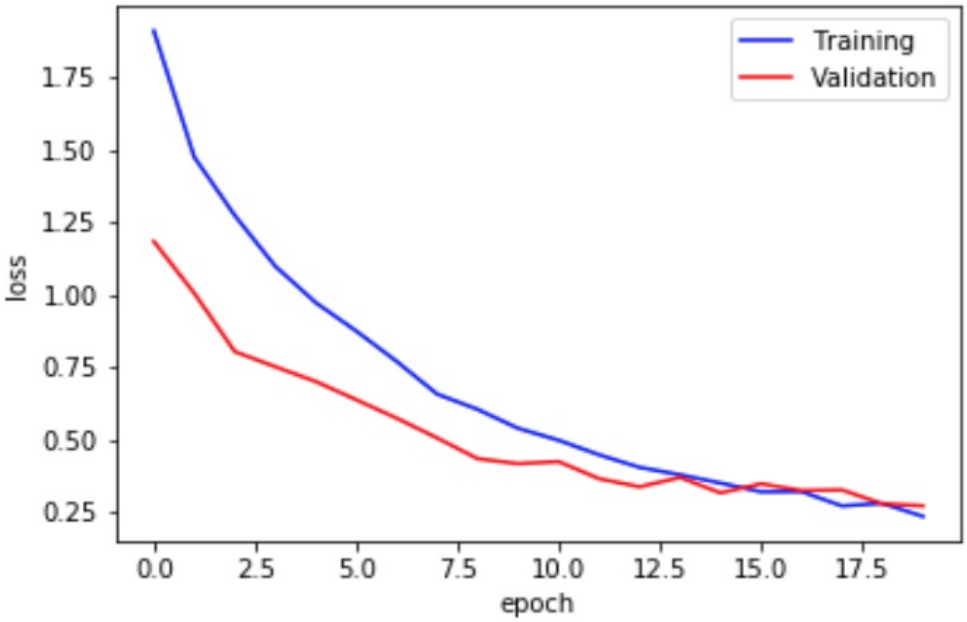
Before proceeding, we save the weights that have been trained so far as a checkpoint for later use.

Next, we will give our model another chance to learn more. But this time, we will un-freeze the last eight layers of the original network (from the Convolutional layers) and let them be trained along with our custom layers. This strategy is known to be quite efficient to train network with

a small dataset. What we will do is to load the weights from out checkpoint earlier and start training again using the loaded weights for about `20 epochs`. This process has been experimented and we get an accuracy of `92%`, which is not a bad performance after all for a classification problem, especially when compared to the benchmark model!

The training/validation progress has been recorded as below:





After acquiring a desired result, we can plot out the `accuracy` and `loss` values for both Training and Validation folders to validate/evaluate our model. This will be discussed more in

the below.

IV. Results

Model Evaluation and Validation

During the the process of training our model, we kept track of the `training` and `validation` accuracy at every epochs. As shown in the previous screenshots for both `accuracy` and `loss`, both `training accuracy` and `validation accuracy` were increasing together consistently until about the `15th` epoch, then they started crossing path each other and kept increasing slightly for about a few more epochs before staying stable at around `90%` accuracy.

The `losses` also reflect the same behavior as `accuracy` but in the exact opposite manner. And finally produce a loss value around `0.27`.

To achieve this result, I fine-tuned and eventually used the same set of parameters as above:

- Batch size: `64`
- Optimizer: `Adam`
- Learning rate: `0.00001` (or `1e-5`)
- Epochs: `10` for the first time, and `20` for the second time.

These parameters proved to yield the best result so far. I have tried learning rate of `1e-4` or `1e-3` but the model learns pretty fast with them and did not actually manage to converge to the optimal result, even up to `30 episodes`.

In order to make sure our model is performing well up to our expectations, we can use the function `evaluate_generator` from Keras API, which will take in `validation generator` along with `steps`, batch size for images in the generator, as following:

```
evaluation = predicted_model.eveluate_generator(valid_generator,
    steps=valid_generator.batch_size)
```

And get validated results:

```
[0.29231724081289434, 0.9150895146152858]
or
[0.29, 91.5%]
```

Loss is about `0.29` and accuracy is about `91.5%`, which is not really bad for making predictions.

With this result, it performs much better than the benchmark model (with `3.48%`). Obviously, as a machine learning model, we cannot expect a perfect predictor with `100%` of accuracy when trying to name any kind of flowers. Within its capabilities, a model that can produce an accuracy of around `90–92%` should be reasonable enough to consult from.

However, for a classification problem, where the use case is more about a hit-or-miss kind of problem, `90%` itself is not really robust enough to clearly tell users whether a flower is associated to a certain names. For example, out of 100 times we consult this model for a flower's name, and 10 times we get wrong answers. For some users, this is not enough to trust, and I agree.

In the other hand, when a scenario of predicting a flower's name does not actually damage anything seriously, most of the users can compromise with its performance by giving it another try or just do not take the result seriously. In that case, then this model is appropriate for such applications where entertainment is more preferred than the prediction correctness.

Another point about this model is that even if the input data is changed in any ways, it should not affect much the final result since the learning rate is small in this case (`1e-5` or `0.00001`) so it can better find the optimal solution (easier to converge to minimum point using `Gradient Descent`), and also because training images were shuffled when loaded for training, so it should be trained in a different orders when testing.

In my opinion, it really comes down to the use cases where we want to apply this model to predict flowers' names and user's strictnesses on result accuracy.

Justification

As discussed above, the benchmark solution we have derived earlier does not have good performance initially, but it did serve as a baseline. Then we derived another model with thorough strategy, which yields a much better result and much more reliable predictions (`92%` compared to `3.48%` from benchmark model). And the final model with the chosen strategy has been discussed along with shown statistics. The main reason came from the fact that we make use of the pre-trained weights from ImageNet database by using VGG-19 network.

Hence, we should be able to make use of the model with `92%` performance for our applications in most cases.

V. Conclusion

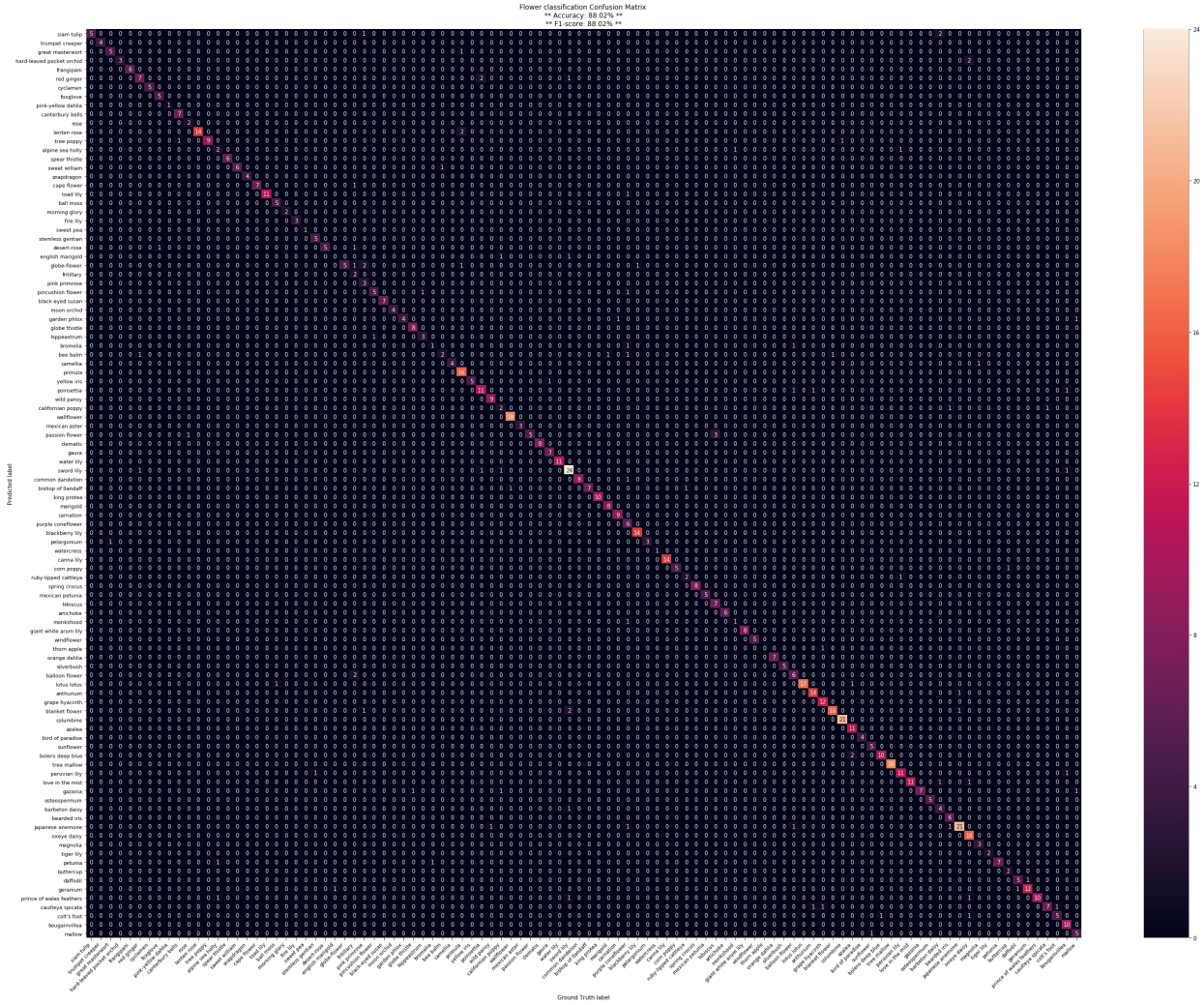
Free-Form Visualization

As a result, the following screenshot captures what has been predicted from our final solution model, which displays each filenames in the separate testing images datasets along with its predictions labels and its actual labels (associated with corresponding order indices).

	Filename	Prediction	Predicted label	Ground truth	Grouth truth label
0	image_06739.jpg	1	pink primrose	1	pink primrose
1	image_06749.jpg	1	pink primrose	1	pink primrose
2	image_06755.jpg	1	pink primrose	1	pink primrose
3	image_06756.jpg	86	tree mallow	1	pink primrose
4	image_06758.jpg	1	pink primrose	1	pink primrose
5	image_06763.jpg	1	pink primrose	1	pink primrose
6	image_06765.jpg	51	petunia	1	pink primrose
7	image_06769.jpg	1	pink primrose	1	pink primrose
8	image_07094.jpg	10	globe thistle	10	globe thistle
9	image_07101.jpg	10	globe thistle	10	globe thistle
10	image_07102.jpg	10	globe thistle	10	globe thistle
11	image_07107.jpg	10	globe thistle	10	globe thistle
12	image_07895.jpg	100	blanket flower	100	blanket flower
13	image_07904.jpg	100	blanket flower	100	blanket flower
14	image_07905.jpg	100	blanket flower	100	blanket flower
15	image_07917.jpg	100	blanket flower	100	blanket flower
16	image_07929.jpg	100	blanket flower	100	blanket flower
17	image_07931.jpg	100	blanket flower	100	blanket flower
18	image_07951.jpg	89	watercress	101	trumpet creeper
19	image_07962.jpg	101	trumpet creeper	101	trumpet creeper
20	image_07963.jpg	101	trumpet creeper	101	trumpet creeper
21	image_07970.jpg	89	watercress	101	trumpet creeper
22	image_07985.jpg	101	trumpet creeper	101	trumpet creeper
23	image_08002.jpg	102	blackberry lily	102	blackberry lily
24	image_08006.jpg	102	blackberry lily	102	blackberry lily
25	image_08014.jpg	102	blackberry lily	102	blackberry lily
26	image_08038.jpg	102	blackberry lily	102	blackberry lily
27	image_08040.jpg	102	blackberry lily	102	blackberry lily
28	image_08041.jpg	102	blackberry lily	102	blackberry lily

Here is just a sample screenshot with 28 images, a full report will be attached along with this project in the `images` folder.

If the above result is a bit hard to really capture how well the model performs, we can also look at the `confusion matrix`, which includes the number of images which are correctly classified across the main diagonal axis from the top left corner, as following:



The `brighter` the color, the more images were correctly classified, and this happens across 102 flower categories. And this confusion matrix really did do a good job in summarizing well the correctness from our model's performance.

Note:

- x-axis: Actual labels.
- y-axis: Predicted labels.

Model evaluation with Metrics

We use calculate al four metrics, which have been described above to see how well our model is performing, and the results are pretty consistent!

The calculations have been done using `sklearn` library.

```
Flowers not being predicted: ('14', 'spear thistle')

Flowers not being predicted: ('7', 'moon orchid')

Accuracy: 90.95%
Precision score: 91.88%
Recall score: 91.18%
F1-score: 90.90%
```

Note:
There are two flower classes which have not been predicted so they are just ignored when calculating these metrics because there are no predictions for them eventually.

Reflection

In general, for a flower classification problem like this, we need an appropriate set of data images beforehand, which was provided from the University of Oxford as stated at the beginning of this report.

Next, we load the images into `Image Data Generator` so we can apply `Augmentation` technique to the images along with various other parameters, such as `batch_size`, `shuffle`, `color_mode`, `class_mode`, etc. for more diverse characteristics for training and validating.

Then we visualize the input images with their respective labels to see how our data really looks like as batches of images.

We need a benchmark model to make sure this problem is solvable by building a Deep Neural Network, even as simple as having a few `Dense` layers. And we get a result of `3.48%` of accuracy.

As a must, we need to improve the above accuracy significantly than just `3.48%`. So `Transfer learning` technique is used in this case with the pre-trained weights from VGG-19, which has been trained through 1000 images from `ImageNet` database. Since our dataset is small (less

10,000 images) and quite similar to images from `ImageNet`, we can apply an appropriate strategy by excluding the top layer (output/predicted layer) from the original network, and adding extra custom layers to fit our problem. Also make sure to freeze all the original layers from VGG-19, and let it trained for about `10 episodes` in order to give our new custom layers a chance to gain some knowledge about our dataset. Afterwards, we un-freeze more Convolutional layers in the original network (VGG-19), about `8 layers`, then let it be trained again for another `20 episodes` to reach the optimal performance.

After getting a desired output, we can evaluate the model and visualize the results by putting the predicted labels with its corresponding images.

From this step onwards, we can export the model and convert to appropriate format to integrate to either `Android` or `iOS` platforms, or just use it in any applications from websites as a backend infrastructure.

One of the interesting aspects about this project is that we can play around with the different types of layers, and also the pre-trained model from `ImageNet` to help ourselves achieving better results eventually. At the same time, tuning parameters or try different training strategies, as well as time consuming process for training (even on GPU) makes the entire process more difficult to achieve an optimal results.

In the end, this model is somewhat targeted for different types of entertainment applications rather than serious ones. Hence, I think the results achieved so far was a great research experience and even when it is just used for general setting to solve similar object classification problems, since most of them are identical in many aspects.

Improvement

After creating a Machine Learning model solution for this problem, I have recognized a few things that could lead to some improvements as a result:

1. Since our dataset are considered as small compared to the amount of data usually collected by a company through years, we can add more data images so that our model can have more knowledge on each categories. Because a Machine learning model is essentially built based on data, and the more data the better the performance.

2. Also related to data perspective, since each category varies from 40 to 258, it is not really a `balanced` dataset. And this could affect the performance in some ways when training because of the knowledge from minorities. For example, consider a scenario where a flower category only has few images compared to another category which could contains a lot more images, and the minorities will be more likely picked as prediction when the model hesitates between different multiple predictions. On some other dataset types, we

can overcome this situation by using an `oversampling` method like SMOTE (Synthetic Minority Over-sampling Technique) by generating more similar data points using `k neighbors` technique. In our case, we can generate more images using `Data augmentation` for different transformations for certain images and add to each minor categories only. This is one of the technique I would love to research more in the future.

3. We can try different extra custom layers and try to un-freeze more `Convolutional layers` in the original network.

4. We can use `K-fold validation` technique as well to make sure each data images is giving a chance to train and validate the model at the same time. This should make sure our model can make use of all the images without locking any of them just for validation purposes only.

Last but not least, if this final model is made as a benchmark model, after applying more algorithms / techniques as well as adding more data images. There definitely will be better solutions which can be achieved if we spend more time to train and refine the model. There are no end to this kind of problem in the first place since we can use of different pre-trained model other than VGG-19 such as Xception, Inception, ResNet etc. We have so many possibilities to try out with help from GPU power as well.

References:

- SMOTE: https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis#SMOTE