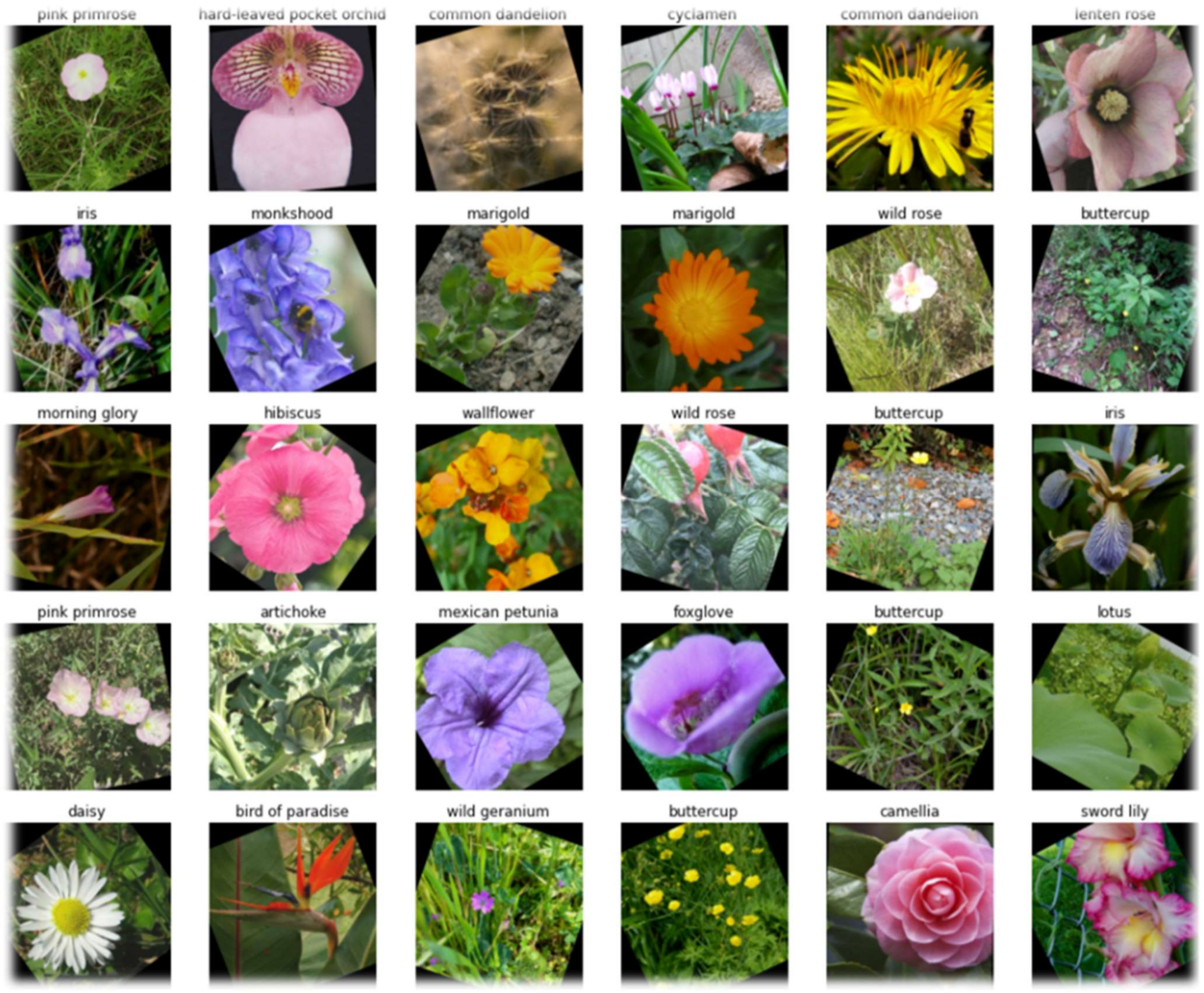


Techniques to Build and Train a Flower Classification Convolutional Neural Network

By: Sau Kha

Fall 2020

Executive Summary

The purpose of this study is three-fold: a) to get experience on cloud computing platform using Tensor Processing Units on Kaggle.com to create and train a convolutional neural network for flower image classification; b) to investigate various techniques to improve accuracy and performance of the network; and c) to create a publicly viewable web presence in the data science field by participating in a code competition named "Petals to the Metal – Flower Classification on TPU[1]" on Kaggle.com.

The research questions for this study are:

1. How to choose a pretrained model to build a convolutional neural network for image classification?
2. What techniques can be used to improve the classification accuracy of the network?

Methodology includes: 1) a trial-and-error exploration phase to search for a pretrained model to build a convolutional neural network, and 2) a hyperparameter tuning phase to find the optimal parameter settings to train the model for optimal performance.

Pretrained DenseNet201 and pretrained EfficientNetB7 were selected to each build a model, by adding a dropout layer, a Global_Average_Pooling2D layer, a batch normalization layer, and the final dense layer with "softmax" activation function for the multiclass classification model.

The model created with EfficientNetB7 achieved a higher Kaggle score of 0.95403 with efficiency. It was trained with less additional external data, without the need for data augmentation and with less training time than the one built with pretrained DenseNet201. The latter model achieved a Kaggle score of 0.95219, which was only slightly lower.

Valuable knowledge and lessons were learned in this study. Experience in cloud computing was gained. A public presence was created at Kaggle.com. Kaggle is an online community of data scientists and machine learning practitioners. Users are provided with free cloud computing resources, free courses, opportunity to participate in different types of competitions. Areas for further study were identified: a) Ops from tf.image and tfa.image modules for random cutout augmentation, b) CutMix technique for data augmentation, c) Methods to overcome the common problem of deep learning: data with class inbalance distribution, and d) Keras Tuner library for hyperparameter tuning.

[1] Tensor Processing Unit

Table of Contents

List of Figures

List of Tables

1 Introduction

Flower classification is one of the most challenging tasks in machine learning. This is largely due to the huge number of known flowering plant species in the world. Flowering plants are so diverse that they comprise of 64 orders, 416 families, approximately 13,000 known genera and 300,000 known species. (Flowering Plant, 2021)[1] The subtle differences within species and similarities between species make the task complicated. Additionally, such classification bases on static images and must overcome variance introduced when the images are captured. Typical factors include the different seasons and times of day the pictures were taken, types of equipment used, developmental stages of the flowers and local immediate environment factors when the static pictures are taken, like shades, angles, background, and so on.

Over the years some prominent classification models have been developed using deep learning libraries such as Keras. (7 Best Models for Image Classification using Keras, 2018)[2] They include: Xception, VGG16 and VGG19, ResNet50, InceptionV3, DenseNet, MobileNet, NASNest, etc. Scholars and Convolutional Neural Network (CNN) developers continuously research and/or scale up earlier classification models for improved accuracy and speed. EfficientNet is a family of models that bases on a uniform way to scale up depth/width/resolution dimensions to improve MobileNets and ResNet (Mingxing Tan, 2019, p. 1)[3]

TensorFlow is a large-scale machine learning open-source system. It incorporates different computer languages and libraries and a sophisticated Application Programming Interface (API) with a multitude of computing hardware and software. It also hosts a great learning/competing environment and community interactions platform for data science students and developers. (An end-to-end open source ML platform, n.d.)[4]

1.1 Purpose of Study

The purpose of this study is three-fold. One is to get experience on cloud computing platform using Tensor Processing Units on Kaggle.com and Google Colab to create and train a convolutional neural network for flower image classification. Second is to investigate various techniques to improve accuracy and performance of the network. Third is to create a publicly viewable web presence in the data science field by participating in a code competition named “Petals to the Metal – Flower Classification on TPU[2]” on Kaggle.com.

1.2 Research Questions

This study attempts to answer the following questions:

1. How to choose a pretrained model to build a convolutional neural network for image classification?
2. What techniques can be used to improve the classification accuracy of the network?

1.3 Intended Audience

Intended audience of this paper is for data science students who are novice in the field of convolutional neural networks.

2 Methodology

The method of this study is first to explore the results from building and training a model with different pretrained models and different input sizes. It started with building a simple model by choosing a pretrained image classification model, adding some top layers as needed and training the model with data of different dimensions. The choice of the pretrained model was to establish a desirable baseline level of performance (low bias) on the training, (low variance) on the validation datasets and a desirable Kaggle score for further improvement in the next phase.

The next step is to apply various techniques to fine tune the model to overcoming overfitting and underfitting to improve the accuracy and speed of training. In this study , the following areas are explored:

- Input Data Factors: a) data dimension, b) whether using external data and how much, c) use of data augmentation or not, d) types of data augmentation, if any.
- Pretrained Models: VGG16, InceptionV3, DenseNet201 and EfficientNetB7. In addition to “imagenet”, the ”noisy-student” were also explored for EfficientNetB7 as initial weights.
- Dropout Regularization: a) positioning of the layer, and b) dropout rate.
- Kernel Regularization: a) types of regularization, and b) regularization factor.
- Batch Normalization Layer.

[2] Tensor Processing Unit

The hyperparameters of the network can make a big difference in the performance of the model. There are two types of hyperparameters: a) model hyperparameters that relate to the model, such as the number and types of hidden layers and units, initial network weights, and activation function, and b) algorithm hyperparameters that influence the speed and quality of learning, such as learning rate, number of epochs and batch size. (Radhakrishnan, 2017)[5] As shown in the bulletized list above, both types of hyperparameters are explored in this study.

In his video "Hyperparameter Tuning in Practice" from the Improving Deep Neural Networks series (Course 2 of the Deep Learning Specialization) on DeepLearning.AI, Andrew Ng recommended babysitting one model as opposed to training many models in parallel when computation capacity is limited. Building and training for an optimal model for image classification is a highly iterative process as illustrated in Figure 1 below. (Ng, 2020)[6]

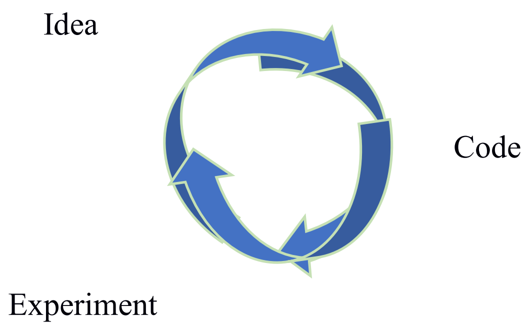


Figure 1. Iterative Process of Hyperparameter Tuning.

2.1 Limitations

To use the competition data to develop and train a flower image classification model, one must join the competition and agree to the official rules. Prediction submission must be made by an output from a Kaggle notebook or script. Notebook run-time per session is capped at 180 minutes. If a notebook run-time exceeds the limit, the "Submit" button will be disabled. Kaggle users have a quota of 30 hours of TPU time per week but no more than 5 submissions are permitted per day. There are rules that govern the use of external data also. All of these put limits on the optimal model in terms of size of network and how to train it.

2.2 Approach

Due to limited resources, the babysitting approach was used to build and fine tune one model at a time. The fine tuning of hyperparameters of this project is done using manual search method. The selection for the value for hyperparameters, such as number of epochs, dropout rate, L2 regularizer factor, and position of dropout layer and addition of batch normalization layer for each training was done by manual search. The values of the parameters, number of training images, Kaggle scores, notebook names and versions, the number of epochs ran, types of data augmentation, run time and so on, were tabulated, evaluated and re-adjusted as the search progressed.

2.3 Performance Measure

Since the ultimate real challenge for the study is to measure how well a model classifies unseen/unlabeled test images, the best way to gauge and compare different designs of my model is to use the public scores from Kaggle's competition on the unlabeled test dataset. Kaggle evaluates submissions on macro F1 score.

F1 is calculated as follows:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

where:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

In "macro" F1 a separate F1 score is calculated for each class / label and then averaged.

2.4 Data

Data for this project is from the Getting Started Code Competition on Kaggle.com, "Petals to the Metal - Flower Classification on TPU" (https://www.kaggle.com/c/tpu-getting-started). Appendix B presents links to the competition data and the external data that is valid to use according to the rules of the competition.

There are four sets of data grouped by the image dimensions: 192x192, 224x224, 331x331 and 512x512. In each group there is a training set, a validation set and a test set. Images in the training and validation sets are labeled with a numeric id while those in the test set are not. There are 104 types of flowers based on images drawn from five different public datasets. Some classes are narrow in the sense that they contain only a particular sub-type of flower, such as pink primroses. Other classes contain many sub-types, such as wild roses.

The dataset contains imperfections, making the competition challenging. As described on Kaggle's competition page, some images of flowers are in odd places, or as a backdrop to modern machinery. Further details of about the data are available from the competition page on Kaggle.[7] Figures 2 through 4 display samples of training, validation, and test images, respectively. Repeated sampling of the training and validation datasets revealed more imperfections, such as an image of bridge without apparent flowers in it, a full picture of a little girl standing and holding a tiny flower with her two hands behind her back, a tattoo of flower pattern on a person's leg, a blank picture, a picture of a fountain with no conspicuous flower in it, etc.

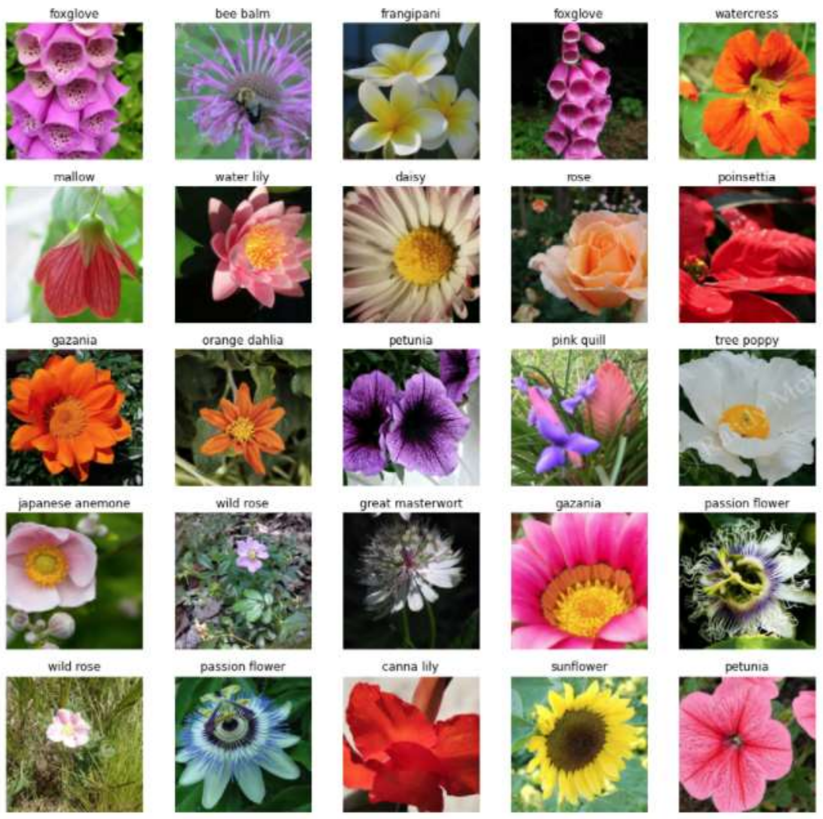


Figure 2. Samples of Non-Augmented Training Images (Labeled).

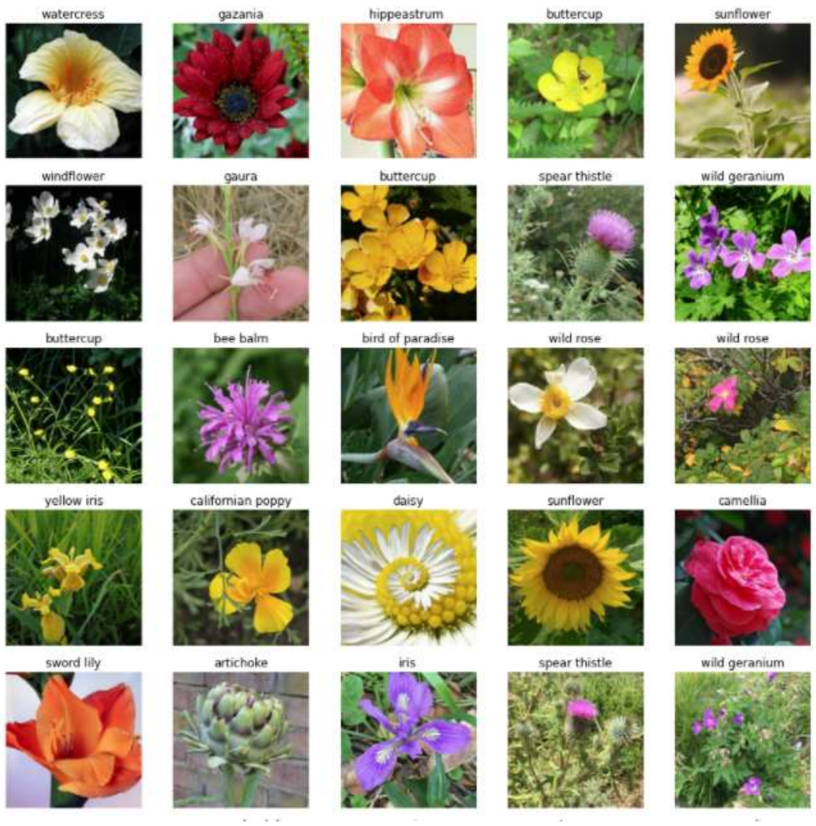


Figure 3. Samples of Validation Images (Labeled).

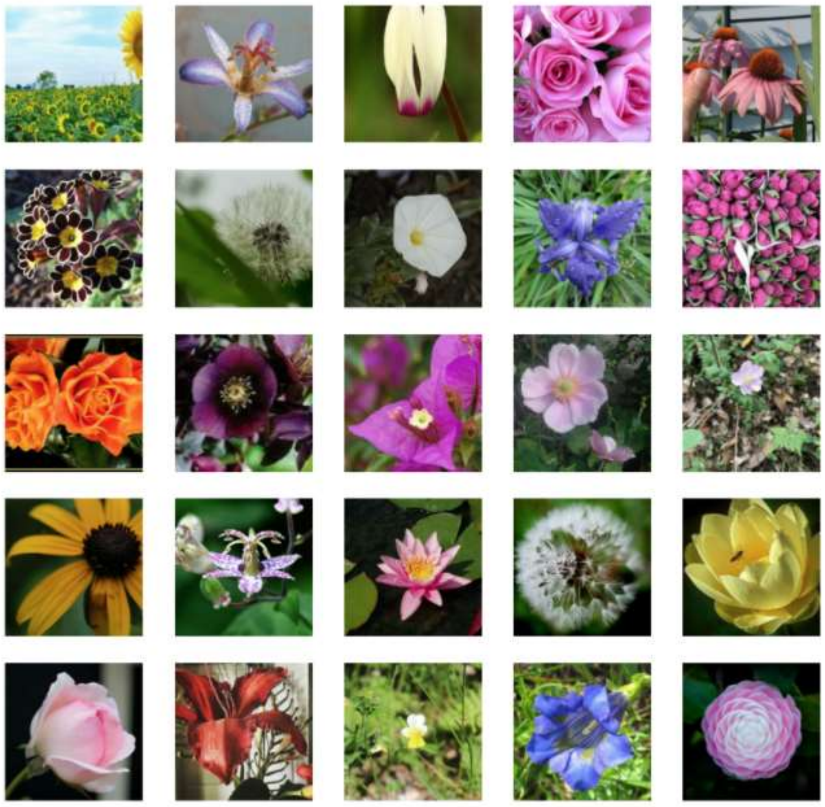


Figure 4. Samples of Test Images (Unlabeled).

3 Pretrained Models and Image Dimensions Exploration

As an exploration, basic models were constructed and trained using the following pretrained models:

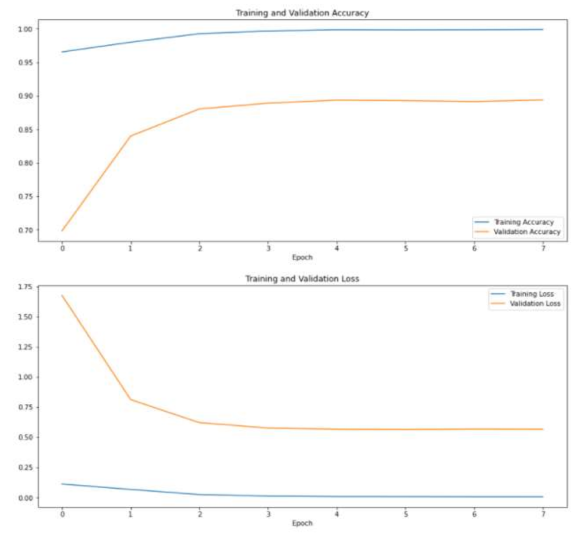
- VGG16: initial weights = “imagenet”
- InceptionV3: initial weights = “imagenet”
- DenseNet201: initial weights = “imagenet”

To begin, very basic models were built with pretrained model (without top layers), and just a GlobalAveragePooling2D layer and a Dense layer as the final layer with 104 nodes and “softmax” activation. As the exploration went on, use of data augmentation, dropout layer and exponential learning rate schedule were added to explore the prospect of continued fine-tuning the model. Charts were generated using data from the history object outputted from each training session, namely sparse_categorical_accuracy and loss metrics from both the training dataset and from the validation dataset. The charts were used to determine whether there was overfitting or underfitting problem that could be overcome to improve the model. Table 1 presents some notable results from the exploration.

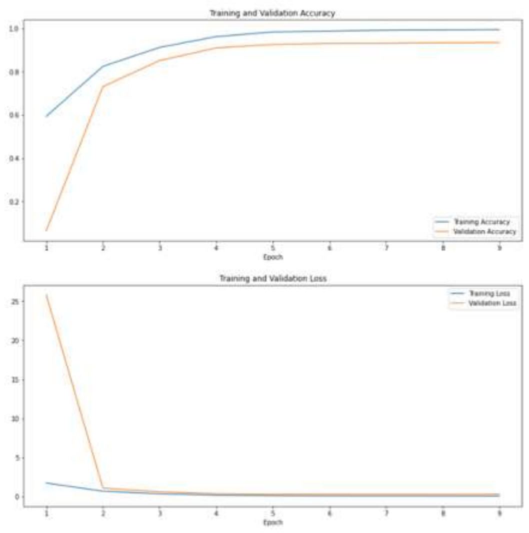
Table 1. Exploratory Results of Model Constructs with Different Pretrained Models.

notebook[3] version	**Pretrained Model**	**Input Dimensions**	**epochs**	**Random Data Augmentation**	**Dropout Factor**	**L2 Regularizer**	**Overfitting Problem?**	**Public Score**
8	VGG16	224x224	25	Not Implemented	0.2	N/A	Yes	0.52431
10	InceptionV3	224x224	25	random L/R flip	0.2	N/A	Yes	0.77868
15	InceptionV3	224x224	8/25[4]	random L/R flip	0.2	N/A	Yes (Figure. 5a)	0.87534
16	InceptionV3	512x512	12/20[4]	random L/R flip	0.2	N/A	Yes	0.92800
23	InceptionV3	512x512	9/20[4]	random L/R flip rdm contrast rdm brightness	0.2	0.00011	Not bad (Figure. 5b)	0.92889
34	InceptionV3	512x512	10/20[4]	random L/R flip rdm contrast rdm brightness rdm saturation	0.2	0.00011	Not bad	0.92906
39	InceptionV3	512x512	10/20[4]	**Turned Off**	0.2	0.00011	Not bad	0.91957
44	DenseNet201	512x512	11/20[4]	random L/R flip rdm contrast rdm brightness rdm saturation	0.2	0.00011	Not bad	0.94279
46	DenseNet201	512x512	9/20[4]	**Turned Off**	0.2	0.00011	Not bad	0.94002

[3] Kaggle Notebook “Petals-to-the-Metals-Flower-Classification” (https://www.kaggle.com/saukha/petals-to-the-metals-flower-classification))
[4] Exponential Decay Learning Rate Scheduler and EarlyStopping were implemented. Training stopped early.



a)



b)

Figure 5. Charts of Sparse Categorical Accuracy and Loss by Epochs
from Notebook Version 15 (left) and Version 23 (Right)

The preliminary results and charts were evaluated and showed that pretrained DenseNet201 for 512x512 images, was a good candidate to proceed with the next phase of this study, hyperparameter tuning. The following lists the takeaways from the exploration:

1. **Pretrained Models.** InceptionV3 scored higher at 0.77868 while VGG16 scored 0.52431 with the same 224x224 input dimension. (See score from notebook version 10 versus version 8.) When trained with 512x512 data, DenseNet201 outperformed InceptionV3 both with random data augmentation (see score from notebook version 44 versus 34) and without random data augmentation (see score from notebook version 46 and 39).
2. **Image Dimensions.** Models got higher scores when trained with data of higher dimensions. (Compare score from notebook version 16 with version 15.) This observation concurred with the fact that higher resolution for each image carries more digital information as inputs for the network to process. This also explains why some deep learning classifiers are scaled up using input resolution methods.
3. **Data Augmentation.** The use of data augmentation got better training results. (Compare score from notebook version 34 with version 39, and version 44 with version 46.)
4. **EarlyStopping and Learning Rate Scheduler with Exponential Decay Schedule.** The model got better training results when EarlyStopping and Exponential Decay Learning Schedule were utilized. They helped the model quickly trained to higher accuracy at the beginning and then fine-tuned gradually to increase the accuracy with exponentially reduced learning rate. EarlyStopping stopped the training early when the loss metric from the validation

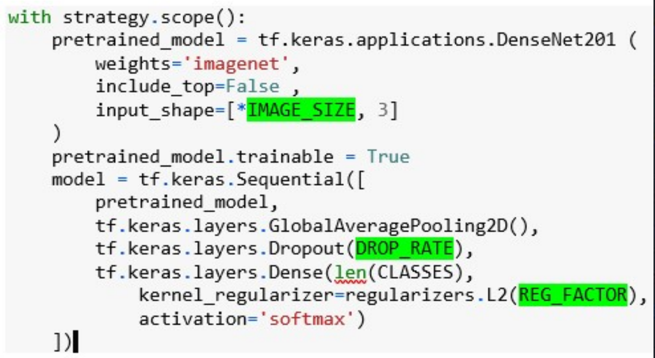
data stopped improving two epochs in a row, and thus prevented overfitting if training were to continue and saved computation time. (Compare score from notebook version 15 to version 10.)

5. **Kernel Regularizer.** When L2 kernel regularizer applied penalty to the weights on the dense layer's kernel and two additional forms of data augmentation was used, overfitting was reduced. The training completed with less epochs and achieved a slightly higher score. (Compare score from notebook version 23 to version 16.)
6. **Combined Regularization Effect.** The combined regularization effect of data augmentation and L2 kernel regularizer reduced overfitting tremendously (See Figure 5 above). Coupling with the use of higher-dimension data input, the model trained in notebook version 23 achieved a higher Kaggle score of 0.92889 than the one trained in version 15 (0.87534).

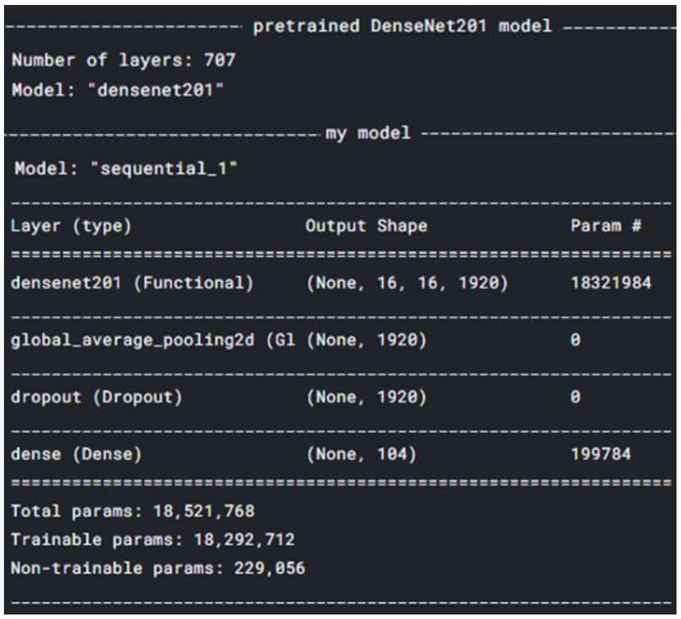
3.1 Selected Pretrained Models and Data Dimension

3.1.1 DensesNet201 for 512x512 Images

Since DenseNet201 pretrained with "imagenet" has a potential to achieve higher scores with hyperparameter tuning than InceptionV3 and VGG16, it was selected to build a convolutional neural network to proceed with this study. Figure 6 presents the structure of the model using DenseNet201 as the pretrained model, initialized with "imagenet", and IMAGE_SIZE = 512x512.



```
with strategy.scope():
    pretrained_model = tf.keras.applications.DenseNet201 (
        weights='imagenet',
        include_top=False ,
        input_shape=[*IMAGE_SIZE, 3]
    )
    pretrained_model.trainable = True
    model = tf.keras.Sequential([
        pretrained_model,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dropout(DROP_RATE),
        tf.keras.layers.Dense(len(CLASSES),
            kernel_regularizer=regularizers.L2(REG_FACTOR),
            activation='softmax')
    ])
```



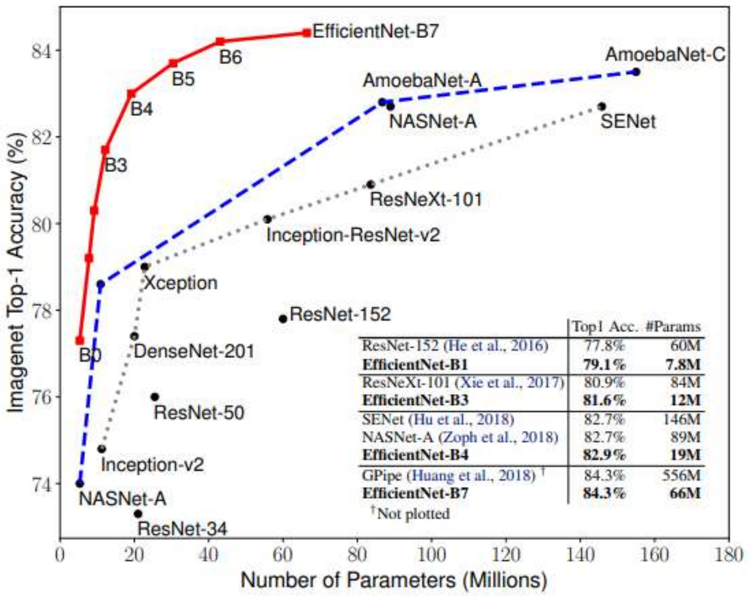
```
---------------------- pretrained DenseNet201 model ----------
Number of layers: 707
Model: "densenet201"
----------------------------- my model -----------------------
Model: "sequential_1"
______________________________________________________________
Layer (type)                 Output Shape              Param #
==============================================================
densenet201 (Functional)     (None, 16, 16, 1920)      18321984
______________________________________________________________
global_average_pooling2d (Gl (None, 1920)              0
______________________________________________________________
dropout (Dropout)            (None, 1920)              0
______________________________________________________________
dense (Dense)                (None, 104)               199784
==============================================================
Total params: 18,521,768
Trainable params: 18,292,712
Non-trainable params: 229,056
______________________________________________________________
```

Figure 6. Model Built with Pretrained DenseNet201.

Note that the dropout layer should have been placed strategically between the last layer of the pretrained model and the added Global_Average_Pooling2D layer. Each of those two layers contains 1920 nodes. It would have produced a greater regularization effect and helped with the overfitting problem. This was corrected after many trials of model training and boosted the Kaggle score at the end.

3.1.2 EfficientNetB7 for 512x512 Images

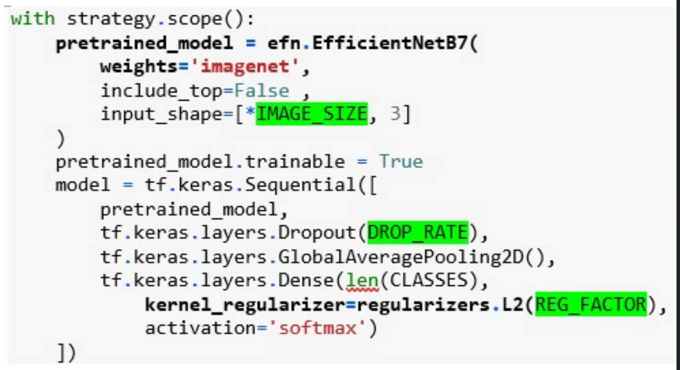
To achieve a higher public score in the competition, another pretrained model was selected to create and train another convolutional neural network with additional external data. EfficientNetB7 is part of a family of models that were developed with an effective scaling method that uniformly scales all dimensions of depth/width/resolution of MobileNets and ResNet, using a simple yet highly effective compound coefficient. Figure 7 compares the model size and ImageNet accuracy of EfficientNet models to other models, showing the achievement of higher Imagenet Top-1 Accuracy rates than DenseNet201. (Mingxing Tan, 2019, p. 1). According to information from code examples on TensorFlow.org, EfficientNet B6 and B7 are scaled up for 528x528 and 600x600 input sizes, respectively. Thus, both are good candidates to take this study to the next level. EfficientNetB7 has a bigger size and more trainable parameters than DenseNet201. Figure 8 presents the structure of the model built with pretrained EfficientNetB7 for hypertuning.



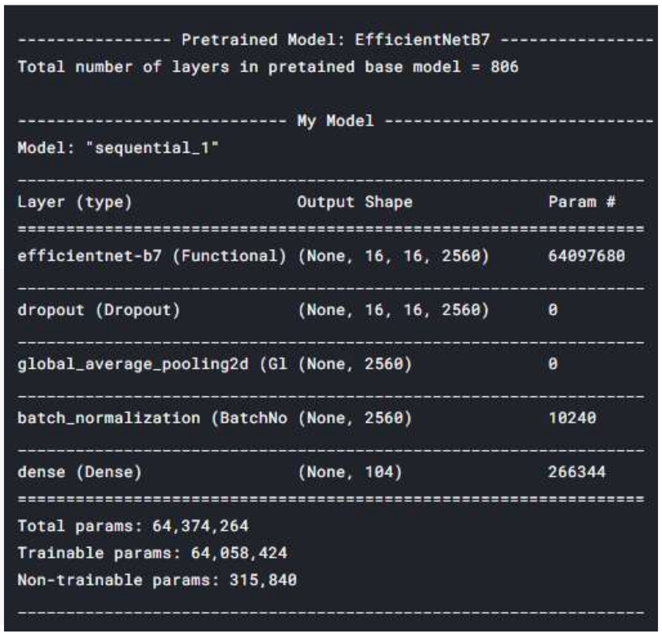
	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) [†]	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

[†] Not plotted

Figure 7. Model Size vs ImageNet Accuracy.



```
with strategy.scope():
    pretrained_model = efn.EfficientNetB7(
        weights='imagenet',
        include_top=False ,
        input_shape=[*IMAGE_SIZE, 3]
    )
    pretrained_model.trainable = True
    model = tf.keras.Sequential([
        pretrained_model,
        tf.keras.layers.Dropout(DROP_RATE),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(len(CLASSES),
            kernel_regularizer=regularizers.L2(REG_FACTOR),
            activation='softmax')
    ])
```



```
--------------- Pretrained Model: EfficientNetB7 ---------------
Total number of layers in pretained base model = 806

--------------------------- My Model ---------------------------
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
efficientnet-b7 (Functional) (None, 16, 16, 2560)      64097680
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 2560)      0
_________________________________________________________________
global_average_pooling2d (Gl (None, 2560)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 2560)              10240
_________________________________________________________________
dense (Dense)                (None, 104)               266344
=================================================================
Total params: 64,374,264
Trainable params: 64,058,424
Non-trainable params: 315,840
_________________________________________________________________
```

Figure 8.   Model Built with Pretrained EfficientNetB7

Note that the dropout layer was placed correctly between the last layer of the pretrained model and the added Global_Average_Pooling2D layer, where there were the greatest number of hidden nodes. A Batch Normalization layer was also added to normalize the inputs feeding to the last dense layer, which optimizes network training.

4 Hyperparameter Tuning

4.1 Discussion: Accuracy Vs Efficiency

The training of the model must deal with practical limitations, which is true for all real-life deep learning network applications. The limiting factor for this study was to keep the run time of the notebooks under 180 minutes. The limit covers run time for installing software, if needed; importing modules; downloading pretrained models, declaring variables and functions; setting parameters; building, compiling, and training the model and saving checkpoint data; calculating predictions; saving the submission file, and saving the notebook (markdown, code, and output cells).

Even though the competition is evaluated by classification accuracy, the training for an optimal model is inevitably a result of tradeoff between accuracy and efficiency. Figure 9a showed that training in Trial 1 resulted in overfitting and should have room for improvement. Thus, to reduce overfitting, random data augmentation was implemented in Trial 2. But the training had to be capped to a much smaller number of epochs (14) for fear of running over time. Metrics from the training indicated that the model could have improved, if only training continued. And the Kaggle score indeed suffered even though overfitting was reduced as shown in Figure 9b. In trial 3, both random data augmentation and more external data were used to reduce overfitting. This was considered a better technique, since natural features from more and different samples is better than artificially augmented data from the same pool of features. The chart in Figure 9c showed that overfitting was further reduced quite successfully. However, due to time constraint, even though metrics and the charts from the training showed promising results with continued training, Trial 3 had to cut short after epochs 10 and ended with the lowest score. To summarize, the training in Trial 1 completed with efficiency and achieved the highest score of all: with least run-time, less data, without data augmentation, but for more epochs. So, it is not always true that the more data the better. There were in fact a total of 54,005 external images available[5] for this competition. But optimization under all constraints is the key.

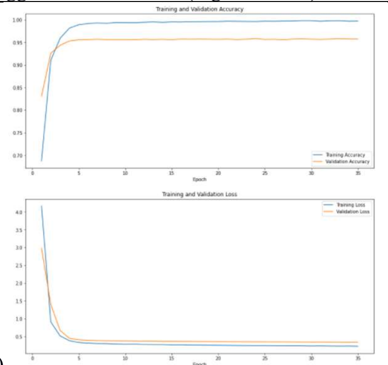
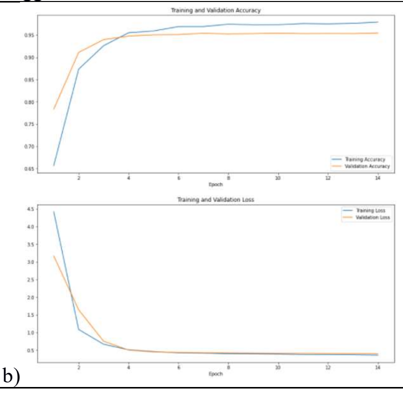
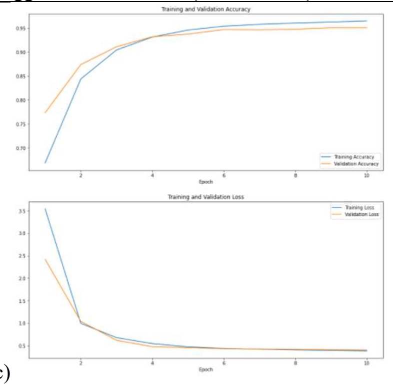
Trial 1	**Trial 2**	**Trial 3**
Notebook: lower-CNN-EfftNetB7_Imagenet_NoAug_ExD_BatchNorm, v14	Notebook: Flower-CNN-EfftNetB7_Imagenet_RA_XD_DO_BN, v3	Notebook: Flower-CNN-EfftNetB7_Imagenet_RA_XD_DO_BN, v4
No Data Augmentation	**Random Data Augmentation**	Random Data Augmentation
Competition Data: 12753	Competition Data: 12753	Competition Data: 12753
External Data: 941	Data: 941	**Data: 6724 (More)**
Epochs: 35/35	Epochs: 14/14	Epochs: 10/10 **(least epochs)**
Run Time: 5379.3 seconds **(shortest run time)**	Run Time: 9006.3 seconds	Run Time: 9182.1 seconds **(longest run time)**
Kaggle Score: 0.95403 (highest score)	Kaggle Score: 0.94570	Kaggle Score: 0.94254 **(lowest score)**
 a)	 b)	 c)

Figure 9. Overfitting, Accuracy and Efficiency

4.2 Optimization

For the phase of trial-and-error hyperparameter tuning, the following were manually adjusted to find the optimal performance of each model:

a) Input Data: External Data (None/How Much), Random Data Augmentation (On/Off/What Type)

b) Model Structure: the inserting, repositioning and the number of hidden nodes of the dropout layer; the addition of a batch normalization layer, and,

c) Algorithm hyperparameters that influenced the speed and quality of learning: EPOCHS; DROP_RATE; REG_FACTOR (for L2 kernel regularizer); initial weights, callbacks for

[5] https://www.kaggle.com/kirillblinov/tf-flower-photo-tfrec

learning scheduler with exponential decay learning rate schedule, EarlyStopping and Checkpoints saving the best model during training.

4.2.1 Inputs

As discussed in Section 4.1, the decision about input data is more a practical one. Training with more data and with random data augmentation both bear computational cost. Inputs to the network affects how it learn and should be decided before training and hyperparameters tuning begins. For real-life situation, there are other practical aspects that relate to data, such as data acquisition, processing, and managing, all of which bear operational cost. Is it better to divert valuable resources to acquire additional data? Would it better to devote all resources to developing and training the model? What about associated cost in hardware, such as graphics processing units (GPUs), required to handle the larger amount of data?

During the hyperparameters tuning phase, it was discovered that after a model was trained to it optimal performance, re-training the same model using more external data did not necessarily boost the performance. The reason was that the optimal values for the hyperparameters may need to adjust for such different set of inputs. The whole process for regularization to handle overfitting and to find the optimal set of parameters had to start over. My intuition was that this mainly depends on the quality of the additional data and how they relate to the validation data that the model uses as a measure for how well it performs. For this study, the amount of data inputs was determined such that the model could fit the data for a reasonable number of epochs to go through a good portion of the exponential decay learning rate schedule, and most importantly, such that the run time of the notebooks on Kaggle falls within the limit of the competition.

4.2.1.1 Data Augmentation

Random data augmentation is a way to reduce overfitting by randomly altering the inputs to the network during training. In this study, resources from tf.image, tfa.image and tf.keras.layers.experimental.preprocessing modules were used.

The arguments for the individual data augmentation functions were also fine-tuned to produce desirable effects. Figures 10 through 13 presents samples of randomly augmented training images using functions from the modules listed above. Refer to the notebooks published on Kaggle for details of settings and implementation under Sau Kha's profile. (URL: https://www.kaggle.com/saukha/petals-to-the-metals-flower-classification?scriptVersionId=58751347&cellId=20) It is worth noting that the notebook "Petals-to-the-Metals-Flower-Classification" was the first notebook created in this study and has evolved with many versions to conduct pretrained models and data dimensions exploration, and hyperparameter tuning. This notebook documented some lessons and ideas to share with the Kaggle community and received a silver medal.

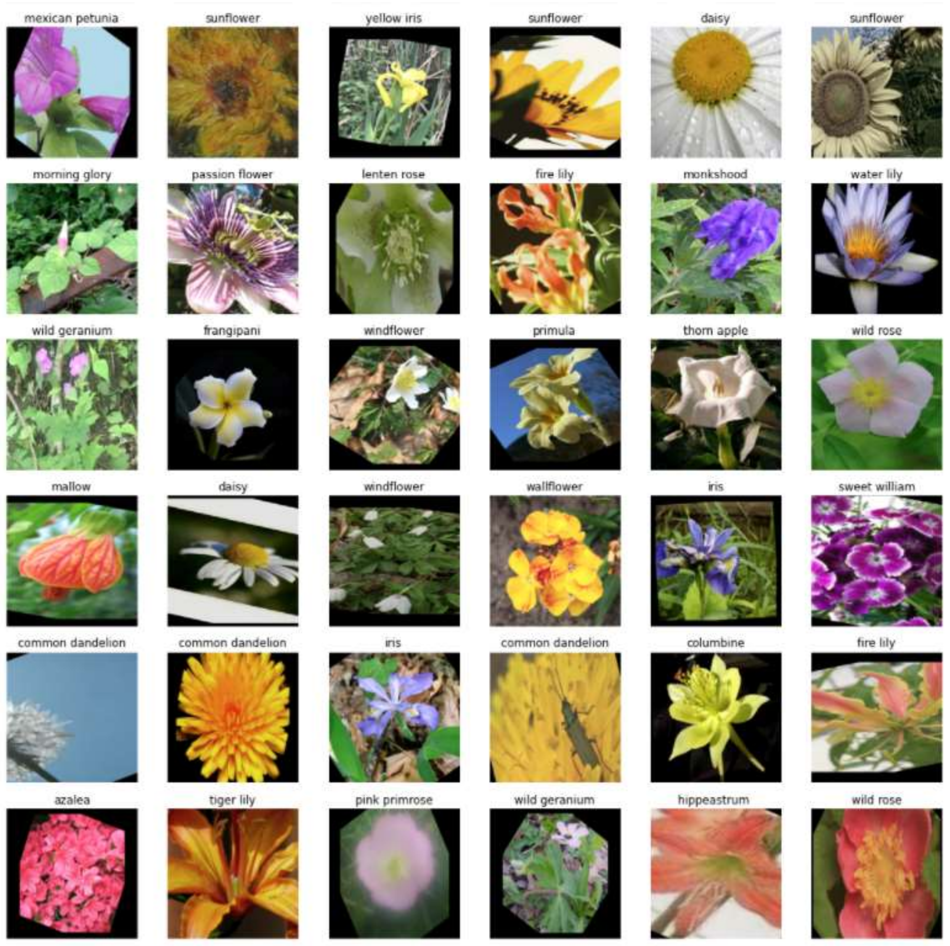


Figure 10. Samples of Augmented Training Images (Labeled).

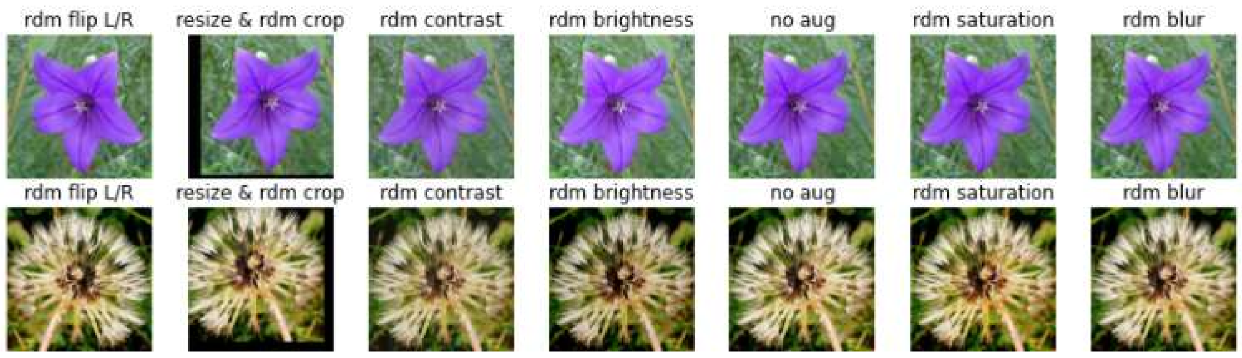


Figure 11. Augmented Training Images using tf.image and tfa.image Modules.

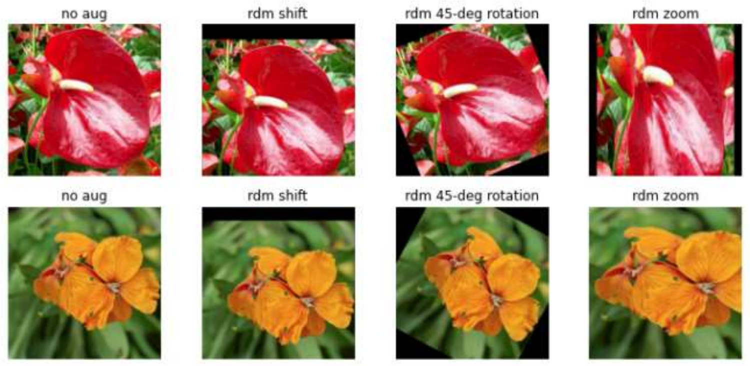


Figure 12. Augmented Training Images using keras.preprocessing.image Module.

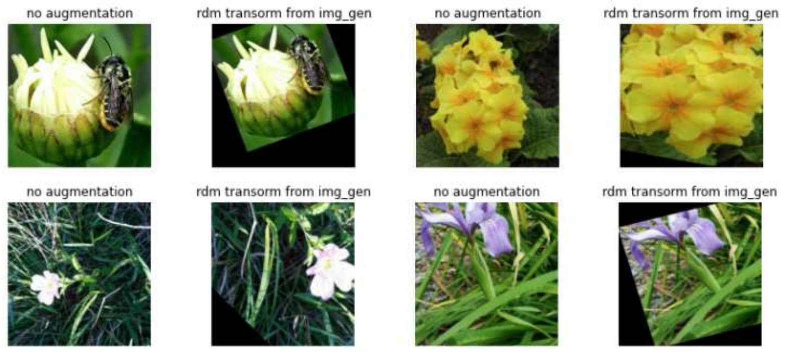

no augmentation
rdm transorm from img_gen
no augmentation
rdm transorm from img_gen
no augmentation
rdm transorm from img_gen
no augmentation
rdm transorm from img_gen

Figure 13. Augmented Training Images using random_transform Method from keras.preprocessing.ImageDataGenerator Module.

4.2.2 Model Structure Optimized: Dropout Layer and Batch Normalization Layer

A dropout layer was added to the model structure as a regularization technique to prevent overfitting. The technique involves randomly dropping out nodes in the dropout layer during training to create the effect of training with many different and randomly reduced network architectures (versus the full sized convolutional neural network). Since any node in dropout layer may be dropped randomly, the nodes of the next layer cannot rely heavily on any one node (feature) in the dropout layer for input but must spread out the weights on many features. This helped to avoid overfitting and increased the generalizing power of the model. By spreading out weights on the hidden nodes, the occurrence of large weights was lowered and thus brought the effect of L2 kernel regularization in the dense layer. When the dropout layer was moved up to between the last layer for the pretrained model and the Global_Average_Pooling2D layer, it had a greater regularization effect with a smaller drop rate and helped train the model for a higher score. Table 1 showed how the model built with pretrained DenseNet201 improved with the proper use of a dropout layer.

Table 1. Dropout Layer Regularization

Line #	Nb Version	Notebook	Average Score	Dropout Layer Position	DROP_ RATE	L2 REG_FACTOR
1	3	Flower-CNN-DenseNet201-Aug	**0.94299**	Moved up	0.1	0.00011
2	44, 47 - 51	Petals-to-the-Metals-Flower-Classification	0.93912		0.2	0.00011
3	93	Petals-to-the-Metals-Flower-Classification	0.93346	Not added	N/A (0.0)	N/A

Additionally, batch normalization was implemented by adding a batch norm layer before the activation dense layer. The batch norm layer standardized the inputs for each node of the next layer. It was done for all the training examples in each mini batch, keeping the value mean = 0 and variance =1. Thus, it improved the performance of the model by reducing the impact of parameter updates and variance from the Global_Average_Pooling2D layer for each node in the dense layer. This made the model more stable and sped up learning. Figure 14 a and b demonstrate the contrasts between training with and without batch normalization layer.

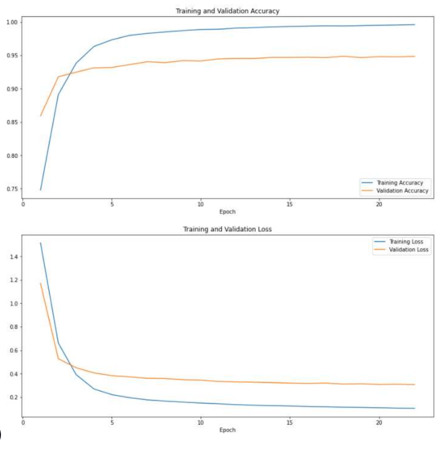
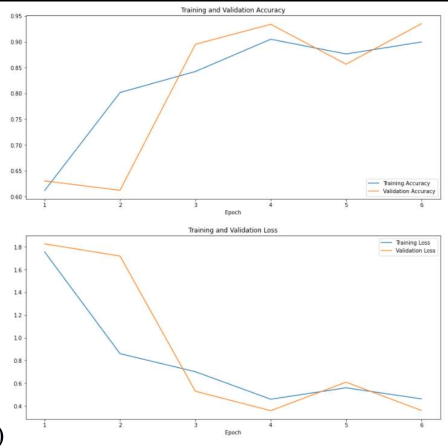
Trial 1 with Batch Normalization		**Trial 2 WITHOUT Batch Normalization**	
Flower-CNN-EfftNetB7_Imagenet_NoAug_ExD_BatchNorm, versio 6		My-Flower-CNN-Augmentation-External-Data, version 13	
Epochs: 22/22	Run Time: 7596.5 seconds	Epochs 6/15	Run Time: 3080.5 seconds
Score**: 0.94709**		Score**:** 0.91930	
 a)		 b)	

Figure 14. Batch Normalization Regularization

Training in Trial 1 (Left) was sped up with batch norm and achieved a higher score. 22 epochs were completed more than 3 times the number of epochs of training in Trial 2 in a little more than double of the run time. The chart in Figure 10b indicated unstable training in Trial 2.

4.2.3 Algorithm Hyperparameters Influencing Training

Two algorithm hyperparameters were investigated in this study: dropout rate for the dropout layer and regularization factor for the L2 kernel regularizer in the dense layer.

Dropout is a technique to prevent overfitting. The dropout rate argument for the dropout layer determines the ratio of hidden nodes to be "eliminated" from the network during training. At the same time, the in-going and out-going links of the randomly selected nodes in the dropout layer are removed from the network, resulting in a smaller network. The higher the dropout rate, the more reduced the network becomes. Thus, the dropout rate directly influences how the network trains. Training, which includes both forward training and back propagation, on a smaller network produces a regularizing effect and is less likely to overfit. The random dropout is done on each individual training example. For each training example, a unique and random set of hidden nodes are dropped out. Since any hidden node and its out-going links may get dropped at random, nodes in the next layer may not rely on any one node (feature) but must spread out weights among all its input nodes. This creates a similar effect as L2 regularization, which puts penalty on large weights for the loss function. The search for the optimal dropout rate in the study ranged from 0.0 to 0.5.

Regularization is a technique to prevent over-fitting by penalizing a model that has large weights. There are three different types of regularization available to dense layers in keras: a) kernel regularizer applied to weights, b) bias regularizer applied to bias unit, and c) activity regularizer applied to layer activation. These penalties are summed into the loss function that network optimizes. L2 kernel regularization was used in this study to apply penalty to the weights on the dense layer's kernel during optimization. The search for the optimal L2 regularization factor in this study ranged from 0 to 0.005.

4.3 Results

Tables 2 and 3 presents results from some of the trial-and-error training sessions during the hyperparameter tuning. Results are organized and sorted in blocks according to the number of images used for training. While parameters were being searched, random data augmentation was turned on or off by a Boolean variable for individual training session. Thus, random data augmentation is listed as a variable setting.

The highest score in each block of results is ***bolded and italicized.*** The highest score achieved with each pretrained model as base is highlighted in green. As shown in Table 2 line 1, EfficientNetB7 achieved a slightly higher score of 0.95403, with less training images and a little bit shorter training time than the one built with DenseNet201. When set with the optimal dropout rate (0.4) and L2 regularization factor (0.035), data augmentation was no needed to achieve the score that topped the model built with DenseNet201.

Table 2. Kaggle Scores from Pretrained EfficientNetB7

Line #	Kaggle Score	Training Time	Pretrained Model	Initial Weights	Number of Images	Data Aug	Dropout Rate	L2 Factor	Batch Norm
1	***0.95403***	5379.3	EfficientNetB7	imagenet	13694	No	0.4	0.035	Yes
2	0.95188	5456.2	EfficientNetB7	imagenet	13694	No	0.45	0.035	Yes
3	0.95173	4219	EfficientNetB7	imagenet	13694	No	0.4	0.003	Yes
4	0.95093	4606.3	EfficientNetB7	imagenet	13694	No	0.425	0.035	Yes
5	0.94882	5929.7	EfficientNetB7	imagenet	13694	No	0.4	0.035	Yes
6	0.94869	7033.4	EfficientNetB7	imagenet	13694	No	0.415	0.035	Yes
7	0.94763	4290.6	EfficientNetB7	imagenet	13694	No	0.5	0.003	Yes
8	0.94656	5405.1	EfficientNetB7	imagenet	13694	No	0.4	0.004	Yes
9	0.94439	3668.3	EfficientNetB7	imagenet	13694	No	0.3	0.003	Yes
10	***0.95119***	3707.2	EfficientNetB7	noisy-student**	13694	No	0.5	0.002	Yes
11	0.95070	3653.5	EfficientNetB7	noisy-student	13694	No	0.4	0.002	Yes
12	0.94651	2279.6	EfficientNetB7	noisy-student	13694	No	0.3	0.002	Yes
13	***0.94900***	6995	EfficientNetB7	imagenet	37131*	No	0.3	0.002	Yes
14	0.94831	5621.4	EfficientNetB7	imagenet	37131	No	0.2	0.0015	Yes
15	0.94709	7596.5	EfficientNetB7	imagenet	37131	No	0.3	0.003	Yes
16	0.94616	3915.7	EfficientNetB7	imagenet	37131	No	0.1	0.001	Yes
17	0.94281	4744.8	EfficientNetB7	noisy-student	37131	No	0.2	0.003	Yes

** More data was not better. ** The scores appeared to be lower when the models were Initialized with "noisy-student".*

Table 3 line 1 shows that the model built with DenseNet201 required training with more images (16061), implementation of random data augmentation, and the optimal settings of dropout rate (0.1) and L2 regularizer factor (0.003) to achieve a score of 0.95219. Note that a batch normalization layer was not included in the model. If a batch norm layer were added, the training time could have been reduced.

Table 3. Kaggle Scores from Pretrained DenseNet201

Line #	Kaggle Score	Training Time	Pretrained Model	Initial Weights	Number of Images	Data Aug	Dropout Rate	L2 Factor	Batch Norm
1	***0.95219***	6113.0	DenseNet201	imagenet	16061	4 tf.image() + 1 keras layer	0.1	0.003	No
2	0.94932	6097.5	DenseNet201	imagenet	16061	4 tf.image() + 1 keras layer	0.2	0.003	No
3	**0.94299**	1402.8	DenseNet201	imagenet	12753	4 tf.image()	0.2	0.00011	No

5 Conclusion

5.1 Research Question 1: How to choose a pretrained model?

Section 3 explained how the exploration phase of the study helped select a pretrained model for this project. However, in real-life applications, there is no simple answer to the first research question. How to choose a pretrained model to build a convolutional neural network for image classification depends on the available resources and the constraints. One must find out how much computational capacity is available, in terms of hardware and software. A deep learning network may take days or weeks to train to arrive at a desirable performance. A deep, wide, and high-input resolution pretrained model may consist of a huge number of trainable parameters and have greater potential to arrive at higher classification accuracy. However, a decision needs to be made when it comes to accuracy and efficiency tradeoff. How much time is allocated, and how much data is available to develop and train the model? Also, there is human resources question: is it practical to divide the project team for network development and for additional data acquisition? Would devoting time and resources in implementing random data augmentation be more promising without burdening for additional data acquisition and warehousing cost?

Even after a pretrained model family is selected based on its characteristic and capabilities, there are many different options to build and train a model with it. For example, EfficientNet is a family of models from B0 through B7. The depth, width and input resolution of the models scales up from B0 to B7. Each individual model consists of different number of blocks and layers and has different complexity. Which model in the family is more suitable for the project at hand?

On the other hand, the application of the neural network may put a constraint in the size of the network. Take MobileNet as another example. The architecture of MobileNet makes it special in that it requires less computation power to use or to apply transfer learning to. As suggested by its name, it is perfect for mobile devices, for computers without GPU. Thus, it will be a good pick for project that does not require high accuracy. To build a flower classification app for mobile device, MobileNet will be an ideal pretrained model. So, users may take a picture a flower with their cellphone camera for the model to determine what type of flower it is. However, for scientific botanical research, a more sophisticated pretrained model will be more desirable.

So, to answer the first research question of this study: a project needs to predicate with project requirements. Management decisions concerning available resources, constraints and model application must be made before a team of data scientists may be assembled to start the work.

5.2 Research Question 2: What techniques can be used to improve the classification accuracy of the network?

The answer to the second research question, on the other hand, can be very long. As discussed throughout different parts of this report, techniques may include:

a) selecting an optimal pretrained model to build your model,
b) using optimal amount of input data
c) implementing appropriate random data augmentation, and determine whether to turn this on or off for training,

d) model hyperparameter tuning, that is to find the optimal parameter settings that changes the model, such as:
    i. adding a dropout layer,
    ii. adding a batch normalization layer,
    iii. adding a Global_Average_Pooling2D convolution layer,
    iv. adding a keras preprocessing layer for random data augmentation,
    v. repositioning a hidden layer, such as a dropout layer,
    vi. adding a dense layer, with activation function and regularization function, and,
    vii. changing the number of hidden nodes in a layer, etc.

e) algorithm hyperparameter tuning, that is to find the optimal parameter settings that influences the quality and speed the model trains, such as:
    i. dropout rate to determine ratio of hidden nodes in the dropout layer to drop while training individual training example,
    ii. L2 regularization factor,
    iii. Learning rate,
    iv. Batch size,
    v. Initialized weights, and,
    vi. Epochs, etc.

5.3 Lessons Learned

There is the tf.keras.preprocessing.image.ImageDataGenerator module from keras that seems more efficient for data augmentation since the random_transform method can randomly run all selected types of augmentation at one pass as each training example is fit to the model. However, it was not used in the notebooks due to user code errors at runtime. This may relate to the docker image the notebook was created on. On the Kaggle computing platform, users may set preference for programming environment to pin to original environment or to always use latest environment. When pinning to the original environment, users will not get new packages, but users' code is less likely to break. There were times a training session halted due to strange errors like the socket closed, project id error, etc. The following lists a few lessons learned from this study:

1. To be aware of and take control of your computing environment, like the version of packages, dependencies, etc.

2. If you must pay for cloud computing services, the cost of training a deep convolutional neural network may add up very quickly. So, budget and time constraints must be considered when building the model. Trade-offs is not just theoretical.
3. When using free services that comes with a quota, like on Kaggle.com, learning how to switch off the TPU when not needed for an interactive session and how to start the editor without automatically starting an interactive session with TPUs is a plus.
4. If possible, have backup computing resources, such as cloud computing platform like Google Colab, DigitalOcean, or maybe, just a second home device like a Raspberry Pi.
5. Get involved with the data science communities. Share something and take something. Be friendly and grow together.

5.4 Further Study

5.4.1 Data Augmentation

For further study, additional data augmentation methods may be explored. The Ops listed below from the tf.image and tfa.image modules cut out and replace part of an image by black pixels.

- tfa.image.random_cutout,
- tf.image.crop_to_bounding_box,
- tf.image.sample_distorted_bounding_box, tf.image.draw_bounding_boxes, and tf.slice.

Another data augmentation strategy, CutMix, has been demonstrated to be effective in increasing classification accuracy, localization, and object detection rate. Patches of image are cut and pasted among training images where the ground truth labels are also mixed proportionally to the area of the patches. (Sangdoo Yun, 2019, pp. 1, 2, 8)[8]

5.4.2 Class Inbalance

Figure 15 below is obtained from a published notebook from Kaggle (https://www.kaggle.com/georgezoto/computer-vision-petals-to-the-metal), showing the imbalance class distribution the training dataset for the competition. Imbalance class distribution is a common problem in real life applications of deep learning. For this study, insufficient training images for some of the classes could have a big impact on the Kaggle public score, which takes an average of the F1 scores over all classes.

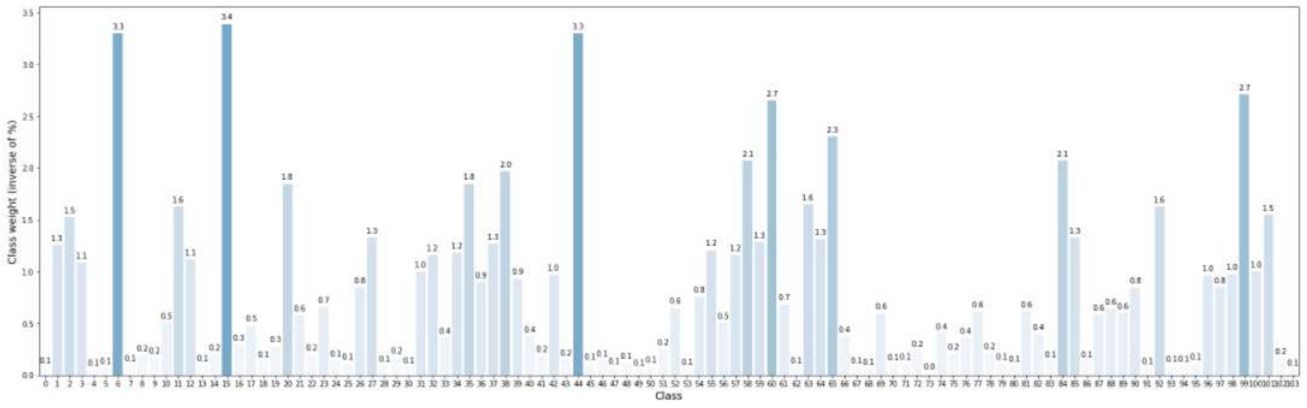


Figure 15. Class Distribution of Training Dataset

Related work, "A systematic study of the class imbalance problem in convolutional neural networks" by Mateusz Buda, et al has been published in *Neural Networks*.[9] The widely used methods to handle class imbalance is at the data level method, which uses sampling methods to increase the balance (oversampling) and to remove a random portion of examples from majority classes (undersampling). The other method category covers classifier (algorithmic) level methods. (Mateusz Buda, 2018, p. 2) Data level methods may be implemented at the mini-batch level and may easily be explored in further study. However, it may put additional challenge to the required computational capacity and cost.

5.4.3 Keras Tuner

This study covered tuning for only a few parameters, while it usually involves a lot more hyperparameters in real-life applications. Hyperparameter optimization is a huge part of deep learning, and ultimately determines how well the final model performs.

Keras Tuner is a library that may help in selecting the optimal set of hyperparameters. Keras is tightly integrated into TensorFlow. The tutorial "Tuning Hyperparameters with the Keras Tuner" on TensorFlow.org (https://www.tensorflow.org/tutorials/keras/keras_tuner), will be a good place to start for the next project. Additional resource "Hyperparameter Tuning with the HParams Dashboard" is also available under TensorBoard > Guide. (https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams).

6 APPENDIX A: Source Codes

Source codes and notebooks for this project are available on my profile page on Kaggle:
https://www.kaggle.com/saukha/code

7 APPENDIX B: Data Source

Competition Data for this project is available from the “Petals to the Metal - Flower Classification on TPU” competition on Kaggle at https://www.kaggle.com/c/tpu-getting-started/data.

External Data that is external to the competition, but approved to use according to the competition rules, is shared by Kaggle user, Kirill Blinov, and available at https://www.kaggle.com/kirillblinov/tf-flower-photo-tfrec.

8 References

[1] "Flowering Plant" *en.wikipedia.org*, 5 March 2021. https://en.wikipedia.org/wiki/Flowering_plant

[2] "7 Best Models for Image Classification using Keras" *IT4nextgen*, 17 Noveember 2018. https://www.it4nextgen.com/keras-image-classification-models/

[3] Mingxing Tan, Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", arxiv.org, last update 11 September 2020, v5. https://arxiv.org/abs/1905.11946

[4] "An end-to-end open source machine learning platform", *TensorFlow,* https://www.tensorflow.org/

[5] Pranoy Radhakrishnan, "What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?", *Towards Data Science*, 9 August 2017, https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a#:~:text=Hyperparameters%20are%20the%20variables%20which,optimizing%20the%20weights%20and%20bias).

[6] Andrew Ng. "Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization (Course 2 of the Deep Learning Specialization)" YouTube, 16 January 2020. https://www.youtube.com/playlist?list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc

[7] "Petals to the Metal – Flower Classification on TPU", *Kaggle*, https://www.kaggle.com/c/tpu-getting-started/data

[8] Sangdoo Yun, et al., "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features". *Openaccess.thecvf.com*, https://openaccess.thecvf.com/content_ICCV_2019/papers/Yun_CutMix_Regularization_Strategy_to_Train_Strong_Classifiers_With_Localizable_Features_ICCV_2019_paper.pdf

[9] Mateusz Budz, Atsuto Maki, Maciej A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks", *Neural Networks, 106:249–259, 2018*, 13 October 2018, https://arxiv.org/pdf/1710.05381.pdf