FLOWER CLASSIFICATION USING TPU

Kaggle competition using TPUs to classify 104 types of Flowers: This competition challenges to build a machine learning model that identifies the type of flowers in a dataset of images.

Team: Apoorva Nerkar

Rohit Nagpal

Flower Classification using TPU (Kaggle Competition)

1. Introduction

a. Description

- For our project we decided to participate in an active Kaggle competition of Flower Classification with TPU's. TPU, which stands for Tensor Processing Unit, are powerful hardware accelerators specialized in deep learning tasks.
- TPUs were developed by Google to process large image databases, such as extracting all the text from Street View.
- The latest Tensorflow release (TF 2.1) was focused on TPUs and they are now supported both through the Keras high-level API and at a lower level, in models using a custom training loop.

b. Objective

- The objective of the competition is to build a Deep Learning model to classify 104 types of flowers using TPUs.
- As we know, training a deep learning model with high resolution images is very time consuming and resource exhaustive, without using GPUs or TPUs, it can take up to hours to train the model even for 1 epoch whereas it could be done within seconds using GPUs and TPUs.
- Therefore, the main objective of this competition is to make sure competitors learn to use the TPUs and analyze how the solutions are accelerated by TPU's.

c. Difference between CPU, GPU and TPU

- **CPU (Central Processing Unit)**
  CPU is a general-purpose processor means a CPU works with software and memory. A CPU has to store the calculation results on memory inside the CPU (so called registers or L1 cache) for every single calculation. CPU's Arithmetic Logic Units (ALU, the component that holds and controls multipliers and adders) executes them one by one, accessing the memory every time, limiting the total throughput and consuming significant energy.
- **GPU (Graphics Processing Unit)**
  To gain higher throughput than a CPU, a GPU uses a simple strategy: why not have thousands of ALUs in a processor? The modern GPU usually has 2,500–5,000 ALUs in a single processor that means we could execute thousands of multiplications and additions simultaneously. This GPU architecture works well on applications with massive parallelism, such as matrix multiplication in a neural network.
  But, the GPU is still a general purpose processor, for every single calculation in the thousands of ALUs, GPU need to access registers or shared memory to read and store

the intermediate calculation results. Because the GPU performs more parallel calculations on its thousands of ALUs, it also spends proportionally more energy accessing memory.

- **TPU (Tensor Processing Unit)**
  In case of TPU, instead of having a general-purpose processor, it has a matrix processor specialized for neural network workloads. They can handle the massive multiplications and additions for neural networks, at blazingly fast speeds while consuming much less power and inside a smaller physical footprint.

  At first, TPU loads the parameters from memory into the matrix of multipliers and adders. Then, the TPU loads data from memory. As each multiplication is executed, the result will be passed to next multipliers while taking summation at the same time. So, the output will be the summation of all multiplication results between data and parameters. During the whole process of massive calculations and data passing, no memory access is required at all.

d. Challenge

It is difficult to fathom just how vast and diverse our natural world is. There are over 5,000 species of mammals, 10,000 species of birds, 30,000 species of fish – and astonishingly, over 400,000 different types of flowers. In this competition, we are challenged to build a deep learning model that identifies the type of 104 different flowers in a dataset of images.

2. Data Description and Data Preparation

a. Dataset Description

In this competition we are classifying 104 types of flowers based on their images drawn from different public datasets. The images provided are of 4 different size:

i. 192x192
ii. 224x224
iii. 331x331
iv. 512x512

Some classes are very narrow, containing only a sub-type of flower (e.g. pink primroses) while other classes contain many sub-types (e.g. wild roses).

b. Data Format

The images provided are in TFRecord format. The TFRecord format is a container format frequently used in Tensorflow to group and shard data files for optimal training performance.

c. Data Preparation

- Each TFrecord format file contains the id, label (the class of the sample, for training data) and img (the actual pixels in array form) information for the images.
- All the different sizes of images have been classified into 3 segments - Train, Validation and Test
- For Train and Validation images, id's and labels are provided along with the images but for Test images, samples are without labels. We need to predict the classes of flowers for Test images.

d. Submission

As a submission we need to submit a csv file with two fields - id and label

- id - a unique ID for each test sample images
- label - the class of flower represented by the test sample

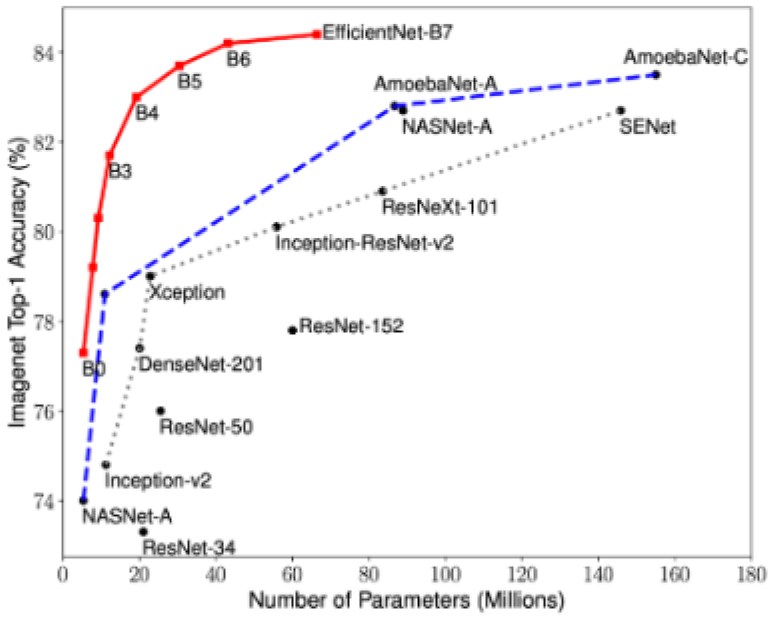
3. Model Description and Alternative Model Analysis

a. Model Description

i. EfficientNet-B7

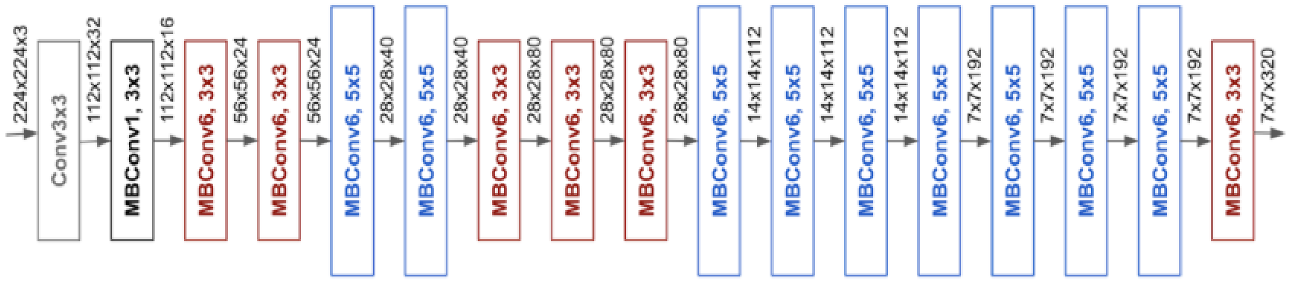
https://github.com/qubvel/efficientnet
https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet

EfficientNets rely on AutoML and compound scaling to achieve superior performance without compromising resource efficiency. The AutoML Mobile framework has helped develop a mobile-size baseline network, EfficientNet-B0, which is then improved by the compound scaling method to obtain EfficientNet-B1 to B7.



- **EfficientNet Architecture**
  https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html

  The effectiveness of model scaling also relies heavily on the baseline network. Developed a new baseline network by performing a neural architecture search using the AutoML MNAS framework, which optimizes both accuracy and efficiency.



  The resulting architecture uses mobile inverted bottleneck convolution (MBConv), similar to MobileNetV2 and MnasNet, but is slightly largerthen scale up the baseline network to obtain a family of models, called EfficientNets.

- **EfficientNet Performance**
  EfficientNets achieve state-of-the-art accuracy on ImageNet with an order of magnitude better efficiency:

  - In high-accuracy regime, EfficientNet-B7 achieves the state-of-the-art 84.4% top-1 / 97.1% top-5 accuracy on ImageNet with 66M parameters. At the same time, the model is 8.4x smaller and 6.1x faster on CPU inference than the former leader, Gpipe.
  - In middle-accuracy regime, EfficientNet-B1 is 7.6x smaller and 5.7x faster on CPU inference than ResNet-152, with similar ImageNet accuracy.
  - Compared to the widely used ResNet-50, EfficientNet-B4 improves the top-1 accuracy from 76.3% of ResNet-50 to 82.6% (+6.3%) under same constraints.

  Though EfficientNets perform well on ImageNet, to be most useful, they should also transfer to other datasets. To evaluate this, we tested EfficientNets on eight widely used transfer learning datasets. EfficientNets achieved state-of-the-art accuracy in 5 out of the 8 datasets, such as CIFAR-100 (91.7%) and ***Flowers (98.8%)***, with an order of magnitude fewer parameters.

- **Mobile inverted bottleneck convolution (MBConv)**
  (https://harangdev.github.io/papers/7/)

  The blocks have 3 features:
  - Depthwise Convolution + Pointwise Convolution It divides original convolution to two steps to reduce computational cost significantly with minimal accuracy loss.

    - In MBConv, blocks are composed with a layer that first expands channels, then squeeze them, so layers that have less channels are skip-connected.
    - Linear Bottleneck - It uses linear activation in the last layer in each block to prevent information loss from ReLU.

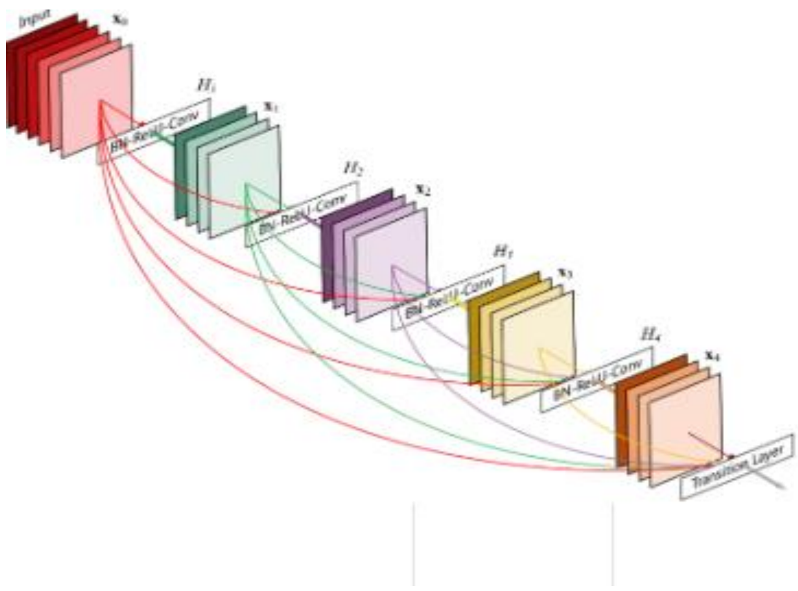
- **Weights**
  (https://arxiv.org/abs/1911.04252)

  - **Imagenet**: ImageNet is an image database. The images in the database are organized into a hierarchy, with each node of the hierarchy depicted by hundreds and thousands of images.
  - **Noisy-Student:** Self-training with Noisy Student improves ImageNet classification. How it works is first train an EfficientNet model on labeled ImageNet images and use it as a teacher to generate pseudo labels on 300M unlabeled images. Then train a larger EfficientNet as a student model on the combination of labeled and pseudo labeled images. Iterate this process by putting back the student as the teacher. During the generation of the pseudo labels, the teacher is not noised so that the pseudo labels are as accurate as possible. However, during the learning of the student, inject noise such as dropout, stochastic depth, and data augmentation via RandAugment to the student so that the student generalizes better than the teacher.

ii. DenseNet201

(*https://arxiv.org/abs/1608.06993*)
(*https://medium.com/intuitionmachine/notes-on-the-implementation-densenet-in-tensorflow-beeda9dd1504*)

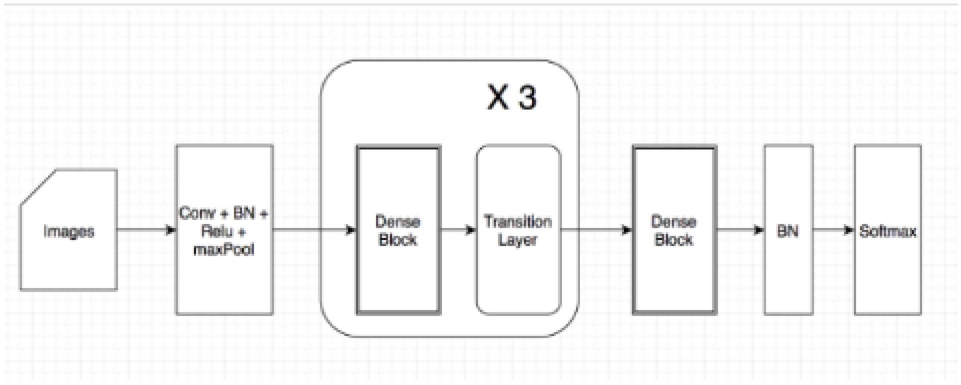
Dense Convolutional Network (DenseNet) connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connections - one between each layer and its subsequent layer - our network has L(L+1)/2 direct connections. For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters.



In DenseNet, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. Concatenation is used and each layer receives a "collective knowledge" from all preceding layers.

Since each layer receives feature maps from all preceding layers, network can be thinner and compact, i.e. number of channels can be fewer. So, it has higher computational efficiency and memory efficiency.

- **The highlights:** Densenet is more efficient on some image classification benchmarks. From the following charts, we can see densenet is much more efficient in terms of parameters and computation for the same level of accuracy, compared with resnet.
- **The structure:** Densenet contains a feature layer (convolutional layer) capturing low-level features from images, serveral dense blocks, and transition layers between adjacent dense blocks.



The architecture of Densenet

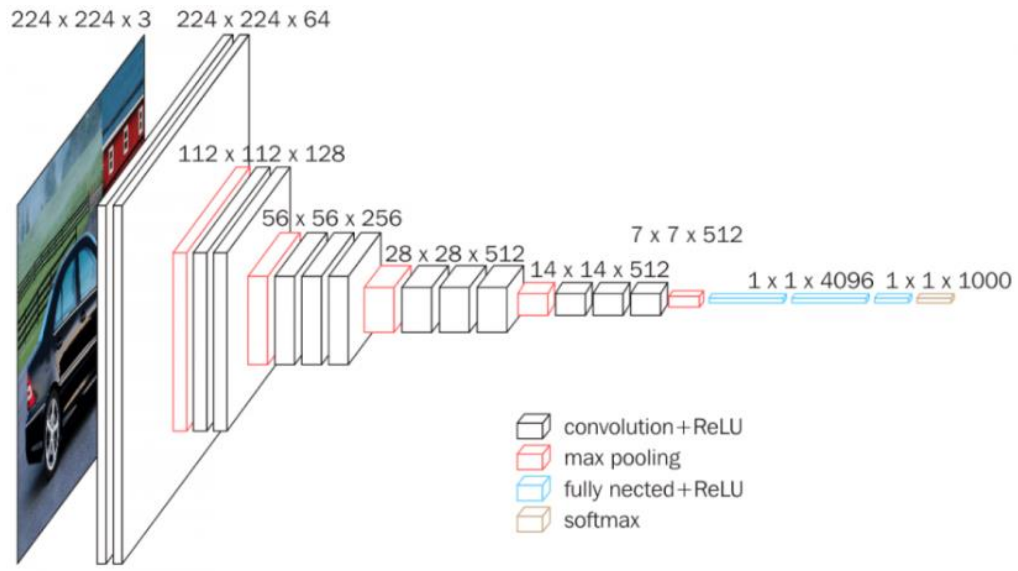
- **Dense block:** Dense block contains several dense layers. The depth of a dense layer's output is 'growth_rate'. As every dense layer receives all the output of its previous layers, the input depth for the kth layer is (k-1) *growth_rate +

input_depth_of_first_layer. This is also where the name of 'growth_rate' comes from. If we keep adding more layers in a dense block, the depth will grow linearly.

- **Transition layer:** As a tradition, the size of output of every layer in CNN decreases in order to abstract higher level features. In densenet, the transition layers take this responsibility while the dense blocks keep the size and depth. Every transition layer contains a 1x1 convolutional layer and a 2x2 average pooling layer with a stride 0f 2 to reudce the size to the half. Be aware that transition layers also receive all the output from all the layers of its last dense block. So, the 1*1 convolutional layer reduces the depth to a fixed number, while the average pooling reduces the size.
- **Parameter Efficiency:** As we have seen, it uses much fewer parameters than resnet. Therefore, the parameters in a densenet are more representative on average. the layers tend to use the information from its closer previous layers, which means the closer layers are not bypassed. Therefore, there are fewer redundant layers. Fewer redundant layers mean more parameter efficiency and less computation.
- **Weights= ImageNet:** ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images.

iii. VGG16

VGG16 is a convolutional neural network model. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes.



VGG-16 has 13 convolutional and 3 fully connected layers, carrying with them the ReLU tradition from AlexNet. This network stacks more layers onto AlexNet and uses smaller size filters (2×2 and 3×3). It consists of 138M parameters and takes up about 500MB of storage space

Challenges of VGG 16:

- It is very slow to train (the original VGG model was trained on Nvidia Titan GPU for 2-3 weeks).
- The size of VGG-16 trained imageNet weights is 528 MB. So, it takes quite a lot of disk space and bandwidth that makes it inefficient.

b. Alternative Models Built

We built the following alternative models:

- **Modelling using GPU and TPU:**
  As mentioned in the introduction GPU and TPU can be used to train models with lower computation time as compared to CPU. Considering the number of training and validation images and different sizes of images available we build following models:
  i. CNN model for input size of (192,192,3) and (224,224,3) using GPU
  ii. CNN model for input size of (331, 331, 3) and (512, 512, 3) using TPU

- **Modelling using different types of pre-trained Image Classification Models**:
  (*(https://www.learnopencv.com/keras-tutorial-fine-tuning-using-pre-trained-models/*)

  The task of fine-tuning a network is to tweak the parameters of an already trained network so that it adapts to the new task at hand. As we know, the initial layers learn very general features and as we go higher up the network, the layers tend to learn patterns more specific to the task it is being trained on. Thus, for fine-tuning, we want to keep the initial layers intact (or freeze them) and retrain the later layers for our task.

  Thus, fine-tuning avoids the limitations like:

1. The amount of data required for training is not much because of two reasons. First, we are not training the entire network. Second, the part that is being trained is not trained from scratch.
2. Since the parameters that need to be updated is less, the amount of time needed will also be less.

Considering pre-rained image classification, we build the following models:
  i. Fine tuning model using VGG16
  ii. Fine tuning model using VGG16 along with LearningRateScheduler
  iii. Fine tuning model using DenseNet and EfficientNet EB7 along with LearningRateScheduler

- **Modelling using GPU and TPU**

  I. **CNN model for input size of (192,192,3) and (224,224,3) using GPU:**
  Build a plain vanilla convolutional network model with following hyperparameters:
  Model:
  ❖ **Layers**: Conv2D, BatchNormalization. Dropout and GlobalAveragePooling2D (or Flatten)
  ❖ **Activation** function (in last layer): Softmax with 104 classes of Flowers

- **Loss function**: sparse_categorical_crossentropy
- **Optimizer:** Adam
- **Regularizer**: EarlyStopping

Dataset:

- 12753 training images
- 3712 validation images
- 7382 unlabeled test images

Performance: ~70-72% Validation Accuracy

Time taken:

- size 192 x 192 -> ~2-2.5 hours
- size 224 x 224 -> ~3.3.5 hours
- size 331 x 331 and 512 x 512 -> Got "Resource Exhausted" Error while training the model. considering the number of images (training + validation), the time taken by GPU is still very low compared to the CPU processor.

II. **CNN model for input size of (331, 331, 3) and (512, 512, 3) using TPU:**

Build a plain vanilla convolutional network model with following hyperparameters:

Model:

- **Layers**: Conv2D, BatchNormalization. Dropout and GlobalAveragePooling2D (or Flatten)
- **Activation function (in last layer):** Softmax with 104 classes of Flowers
- **Loss function:** sparse_categorical_crossentropy
- **Optimizer**: Adam
- **Regularizer**: EarlyStopping

Dataset:

- 12753 training images,
- 3712 validation images,
- 7382 unlabeled test images

Performance: ~74-76% Validation Accuracy

Time taken:

- size 192X192 -> ~1 hour
- size 224x224 -> ~1.5 hours
- size 331x331 -> ~2 - 2.5 hours
- size 512x512 -> ~3 hours

Conclusion:

From mentioned stats we can conclude that TPU not only helped in processing the model at faster rate but also helped in improving the accuracy of the model.

- **Modelling using different types of pre-trained Image Classification Models**

  From the above basic models' analysis, we observed that the best results can be obtained by using TPU with higher image size i.e. 512x512. Therefore, next we tried different type of pre-trained model (fine tuning) along with different model optimization techniques

  Fine tuning model using VGG16

  i. **Model:**
     - ❖ **Pre-trained Model:**
       - o *VGG19(weights='imagenet', include_top=False, input_shape=image_shape)*
       - o *include_top = False* means we have not loaded the last two fully connected layers which act as the classifier. We are just loading the convolutional layers.
       - o trained last two convolutional layers (as a part of fine tuning)
     - ❖ **Layers**: Conv2D, BatchNormalization. Dropout and GlobalAveragePooling2D (or Flatten)
     - ❖ **Activation** function (in last layer): Softmax with 104 classes of Flowers
     - ❖ **Loss function**: sparse_categorical_crossentropy
     - ❖ **Optimizer:** Adam
     - ❖ **Regularizer**: EarlyStopping

  ii. **Dataset:** Size 512x512
     - ❖ 12753 training images,
     - ❖ 3712 validation images,
     - ❖ 7382 unlabeled test images

  iii. **Performance:** ~82-85% Validation Accuracy

  iv. **Time taken:**
     - ❖ Less than 1.5 Hours

  Fine tuning model using VGG16 along with LearningRateScheduler

  i. **Model:** Build fine-tuned model using VGG16:
     - ❖ **Pre-trained Model:**
       - o *VGG16(weights='imagenet', include_top=False, input_shape=image_shape)*
       - o *include_top=False* means We have not loaded the last two fully connected layers which act as the classifier. We are just loading the convolutional layers.
       - o trained last two convolutional layers (as a part of fine tuning)
     - ❖ **Layers**: VGG16, GlobalAveragePooling2D (or Flatten)
     - ❖ **Activation function** (in last layer): Softmax with 104 classes of Flowers
     - ❖ **Loss function**: sparse_categorical_crossentropy
     - ❖ **Optimizer**: Adam

- ❖ **Regularizer**: EarlyStopping, LearningRateScheduler
- ❖ **Dataset:** Size 512x512
    - o 12753 training images,
    - o 3712 validation images,
    - o 7382 unlabeled test images

ii. **Performance:** ~95-96% Validation Accuracy

iii. **Time taken:** Less than 2 Hours

iv. **Prediction:** While predicting the class of test images, we calculated probabilities and then used *"argmax(probabilities, axis=-1)"* to get the class with highest probability.

v. **Comparison:** Compared with earlier model in this model we:
- ❖ only used VGG16 convolutional layers and did not use any other layers like BatchNorm, Dropout etc.
- ❖ Additional regularizer named "Learning Rate Scheduler" being used such that learning rate can be adjusted automatically

vi. **Observation:**
- ❖ Model took more time to converge due to low learning rate with higher epochs
- ❖ But the overall model's accuracy improved significantly from 85% to 95%.

Fine tuning model using DenseNet and EfficientNet EB7 with LearningRateScheduler

i. **Model:** Build two fine-tuned models using DenseNet and EB7.

Model 1: DenseNet
- ❖ **Pre-trained Model:**
    - o *DenseNet201(weights='imagenet', include_top=False, input_shape=image_shape)*
    - o include_top=False. We have not loaded the last two fully connected layers which act as the classifier. We are just loading the convolutional layers.
    - o trained all convolutional layers (as a part of fine tuning)
- ❖ **Layers**: DenseNet201, GlobalAveragePooling2D (or Flatten)
- ❖ **Activation function** (in last layer): Softmax with 104 classes of Flowers
- ❖ **Loss function**: sparse_categorical_crossentropy
- ❖ **Optimizer**: Adam
- ❖ **Regularizer**: EarlyStopping, LearningRateScheduler

Model 2: EfficientNetB7
- ❖ **Pre-trained Model:**
    - o *efn.EfficientNetB7(weights='noisy-student', include_top=False, input_shape=image_shape)*

   - include_top=False. We have not loaded the last two fully connected layers which act as the classifier. We are just loading the convolutional layers.
   - trained all convolutional layers (as a part of fine tuning)
 - ❖ **Layers**: DenseNet201, GlobalAveragePooling2D (or Flatten)
 - ❖ **Activation function** (in last layer): Softmax with 104 classes of Flowers
 - ❖ **Loss function**: sparse_categorical_crossentropy
 - ❖ **Optimizer**: Adam
 - ❖ **Regularizer**: EarlyStopping, LearningRateScheduler

ii. **Dataset:** Size 512x512
   - For Training used sets of both Training and Validation Images to provide the model with a greater number of images - 12753 training images + 3712 validation images
   - 7382 unlabeled test images

iii. **Performance:** ~99-100% Training Accuracy

iv. **Time taken:**
 - ❖ Less than 2 to 3 Hours

v. **Prediction:**
While predicting the class of test images, we calculated probabilities from both the models individually, then took the weighted average of the probabilities and last used *"argmax(probabilities, axis=-1)"* to get the class with highest probability.

vi. **Observation:**
 - ❖ Model took more time to converge due to low learning rate with higher epochs and having two different models
 - ❖ But the overall model's accuracy improved significantly, and we achieved the highest rank of 54 out of 700 participating teams.

4. Model Optimization

Below optimization techniques were used to optimize the results:

a. Batch Normalization

Used batch norm in order to speed up Neural Network learning process

b. Dropout

Used dropout in order to prevent Neural Network from overfitting

c. Transfer Learning

Used transfer learning to accelerate the training of neural networks as either a weight initialization scheme or feature extraction method. Pre-trained network on a large and diverse dataset like the ImageNet captures universal features like curves and edges in its early layers, that are relevant and useful to most of the classification problems.

d. Fine Tuning

Used fine tuning to accelerate the training of neural networks as either a weight initialization scheme or feature extraction method

e. Callbacks

A callback is a set of functions to be applied at given stages of the training procedure. We can use callbacks to get a view on internal states and statistics of the model during training. We can pass a list of callbacks (as the keyword argument callbacks) to the .fit() method of the Sequential or Model classes. The relevant methods of the callbacks will then be called at each stage of the training.

We define and use a callback when we want to automate some tasks after every training/epoch that help us to have control over the training process. This includes stopping training when you reach a certain accuracy/loss score, saving our model as a checkpoint after each successful epoch, adjusting the learning rates over time, and more.

i. Early Stopping

(*https://keras.io/callbacks/ https://www.kdnuggets.com/2019/08/keras-callbacks-explained-three-minutes.html*)

Used early stopping to avoid overfitting and underfitting as too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping stops training once the model performance stops improving on a validation dataset.

Patience: number of epochs with no improvement after which training will be stopped
min_delta: minimum change in the monitored value. For example, min_delta=1 means that the training process will be stopped if the absolute change of the monitored value is less than 1
mode: one of {auto, min, max}. In min mode, training will stop when the quantity monitored has stopped decreasing; in max mode it will stop when the quantity monitored has stopped increasing; in auto mode, the direction is automatically inferred from the name of the monitored quantity. For example, you set mode='min' if the monitored value is val_loss and you want to minimize it.
restore_best_weights: set this metric to True if you want to keep the best weights once stopped

ii. Learning Rate Scheduler

(*https://keras.io/callbacks/#learningratescheduler https://machinelearningmastery.com/using-learning-rate-schedules-deep-learning-models-python-keras/*)

It adjusts the learning rate over time using a schedule that we write beforehand. This function returns the desired learning rate (output) based on the current epoch (epoch index as input).

keras.callbacks.callbacks.LearningRateScheduler(schedule, verbose=0)

schedule: a function that takes an epoch index as input (integer, indexed from 0) and current learning rate and returns a new learning rate as output (float).
verbose: int. 0: quiet, 1: update messages.

In our model, we are reducing the learning rate over the time during the training. Here is the scheduler defined for the learning rate in our model:

a) For first 4 epochs, we start with learning rate = 0.0001 (lr_min) multiplied with some random number (between 0 to 1)

b) For the epochs from 5 to 10, we used learning rate = 0.0008 (lr_max)

c) for the epochs after 10, learning rate =
*(max_lr -min_lr)*decay_rate^(epoch-10) + 0.0001*

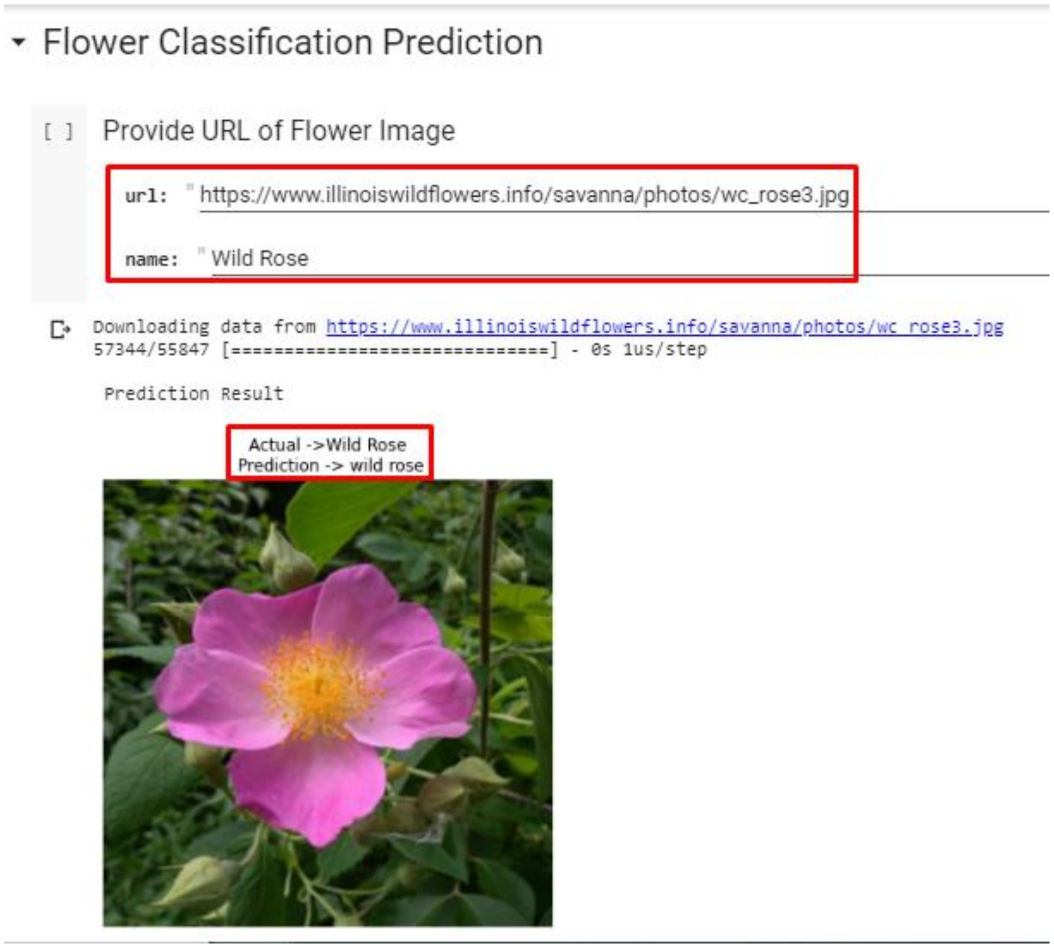
These have the benefit of making large changes at the beginning of the training procedure when larger learning rate values are used, and decreasing the learning rate such that a smaller rate and therefore smaller training updates are made to weights later in the training procedure. This has the effect of quickly learning good weights early and fine tuning them later.

5. Implementation (Prediction, Operationalization)

a. Interactive script using saved models:

From the implementation and demo perspective, we created an interactive script in Google Colab which does the following:

- Load the saved models trained by us. Models are: DenseNet201 and EfficientNetB7
- Asks user to provide any flower image url for the prediction
- Download the flower image from given url
- Process the image to convert it into size of (512, 512, 3) as our model is trained on that size
- Predict the class of the flower and shows with predicted result



Flower Classification Prediction

Provide URL of Flower Image

url: " https://www.illinoiswildflowers.info/savanna/photos/wc_rose3.jpg

name: " Wild Rose

```
Downloading data from https://www.illinoiswildflowers.info/savanna/photos/wc_rose3.jpg
57344/55847 [==============================] - 0s 1us/step

Prediction Result
```

Actual ->Wild Rose
Prediction -> wild rose

b. Created a gitHub repository:

We created a gitHub repository with the following:

- Description
- Difference between different processing units
- Data description
- Model description
- Results
- Complete script
- Demo Script

gitHub repository can be found here:
https://github.com/rohitnagpal92/CNN_Flower_Classification
https://rohitnagpal92.github.io/portfolio/
https://github.com/apoorvanerkar

c. Online Application for Flower Classification:

- As a potential next step of implementation, we can scale-up this model and build a online application where user would just need to provide the image url or upload the flower image and get to know the class of the flower it belongs to as output

6. Discussion (Business Impact, Potential Improvements, Challenges)

a. Business Impact

- **Use of TPU:**
  As one of the main objectives of this competition was to use the TPU for the model building, we got to know how using TPU can not speed up the training of the complex models but also helps in getting better accuracy.
  Although TPU is quite costly as compared to GPU but considering that TPU speeds up the model training better than GPU, TPU can be used in the industries where complex Artificial Intelligence applications are being vastly used for eg. Healthcare, Advertising and Media, Retail, Automotive & Transportation and BFSI

- **Use of Flower Classification Model/Application:**
  As our interactive script (using the saved models) only requires the user to provide the flower image url, the script can be used by Anthophile who likes to collect different types of flowers and classify them in different types. Similarly, the same can be used in floral businesses to classify the types of flowers based on the customer's request.

b. Potential Improvements

As a potential next steps and model improvements, we can scale-up this model by following way:

- **Add more Flower Classes:**
  In our present model we are classifying 104 types of flowers but as we know there are hundreds of different types of flowers available so we further update our script and add other flower types as well.

- **Predict flower classes with combination of different models:**
  For better and accurate prediction result we can train other models as well like VGG19, NASNET etc. and use the prediction result from all the models with different weightage so that we can have better accuracy of the test result.

- **Online application for flower classification:**
  As a final product we can build an online application where user would just need to provide the image url or upload the flower image and get to know the class of the flower it belongs to as an output.

c. Challenges

- One major challenge of using TPUs is the cost associated with it. Since, TPUs are optimal for large models with very large batch sizes and workloads that are dominated by matrix-multiplication like image classification, the process would get very slow if CPU is used for processing.
- Processing of TFrecord format file containing the id, label (the class of the sample, for training data) and img (the actual pixels in array form) information for the images.

7. SUBMISSION TO COMPETITION

a. Final Model:

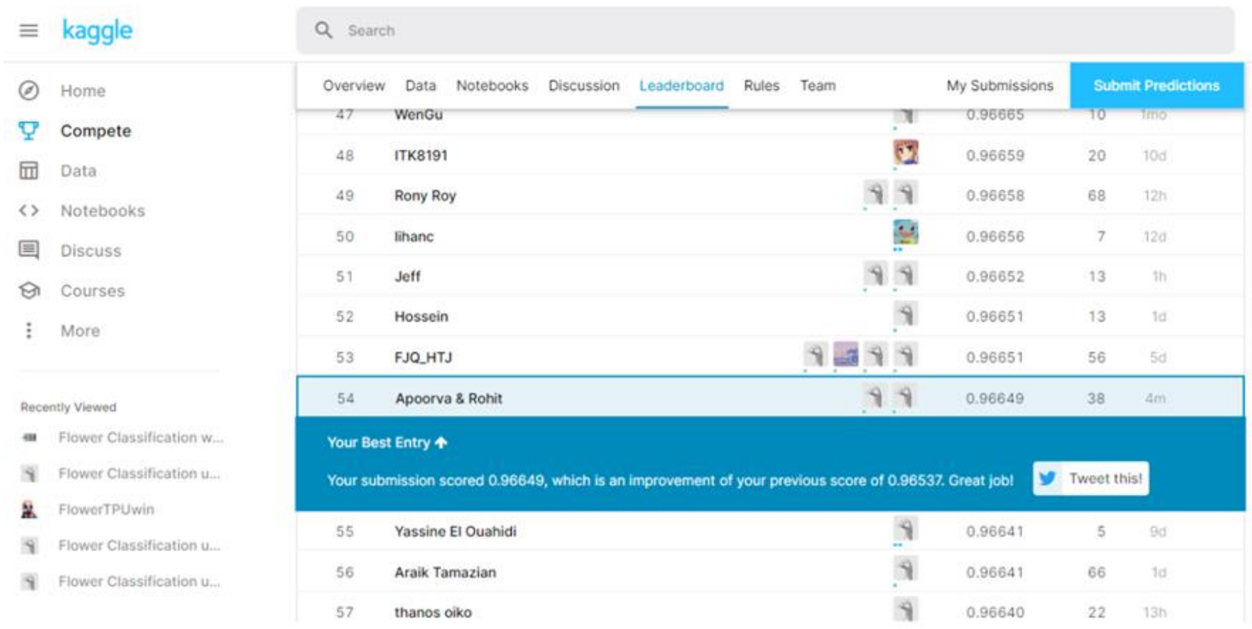
- For the prediction of test images, we used combined effect of two models – DenseNet and EfficientNet-B7
- We took the equal weightage (50%) of prediction from both the models and then used "argmax" function to get the class label with highest probability

b. Submission:

- As a submission we needed to submit a csv file with two fields - ID and label. ID - a unique ID for each test sample images and label - the class of flower represented by the test sample

c. Rank Achieved:

- We managed to achieve the least rank of 54 out of ~800 participated teams.



47	WenGu	0.96665	10	1mo
48	ITK8191	0.96659	20	10d
49	Rony Roy	0.96658	68	12h
50	lihanc	0.96656	7	12d
51	Jeff	0.96652	13	1h
52	Hossein	0.96651	13	1d
53	FJQ_HTJ	0.96651	56	5d
54	Apoorva & Rohit	0.96649	38	4m
55	Yassine El Ouahidi	0.96641	5	9d
56	Araik Tamazian	0.96641	66	1d
57	thanos oiko	0.96640	22	13h

Your Best Entry

Your submission scored 0.96649, which is an improvement of your previous score of 0.96537. Great job!