Flower Recognition

March 17, 2019

CMPS 144, Winter 2019

Tarun Salh
Jack Baskins School of Engineering
University of California, Santa Cruz
tsalh@ucsc.edu

Prabmeet Gill
Jack Baskins School of Engineering
University of California, Santa Cruz
psgill@ucsc.edu

Tyler Hu
Jack Baskins School of Engineering
University of California, Santa Cruz
qhu27@ucsc.edu

ABSTRACT

Convolutional neural networks (CNNs) have created an advancement in computer vision applications such as image recognition, localization, and detection. A successful CNN known as VGG16 won the ImageNet Challenge in 2014 for creating a CNN that most successfully was able to identify objects in images for 1000 different categories in the competition dataset. In our work we decided to implement transfer learning on the VGG16 network and leverage it on detecting 5 different types of flowers: tulips, daisies, sunflowers, dandelions, and roses. We experimented implementing our own CNN on top of extracted layers of VGG16 for collecting features from our dataset and compared our performance with VGG16 model. From conducting our experiments, VGG16 performed exceptionally well on the dataset with accuracies of 93% and 87% training-testing sets respectively. Our model's results got a max of 74.7% and 74.2% on the training-testing sets respectively ultimately achieving a goal of having a network which does not overfit and could potentially increase its performance increasing the number of epochs.

Motivation and Objective

Our goal will be to classify correctly which image corresponds to which flower. In doing so, we wanted to create a model that performs well on not only the training set but also the test set. The typical approach to this type of problem is to use a Convolutional neural network (CNN). There are already several well known CNNs that would solve this problem with great success, such as the aforementioned VGG16. There is also VGG19, Resnet50, and Inceptionv3 just to name some of the few other models we tried. Since a lot of these well known models have been proven to be quite successful, we made it the goal of our project to use one of the pre trained CNNs, in our case we chose VGG16, and try to build on top of it after freezing the model after a few blocks of the CNN. After that we wanted to implement our own CNN, which is described below, on top of the VGG16 in order to see if something that we built ourselves would be able to create comparable results. However, we still wanted to use parts of VGG16 to help grab the features given the weights already provided by VGG16.

Figure 1: **The distribution of our dataset**

Dataset

Our dataset includes a raw set of 4,242 images of flowers that are 320x240. However the photos are not reduced to any fixed size so they have slightly different proportions. All the images were web-scraped from flickr, google images, and yandex images. The types of flowers in our dataset

(labels) are dandelions, daisies, roses, sunflowers, and tulips. From Figure 1 of the distribution of the dataset we see there is also bias in our dataset since the number of tulips and dandelions have a few hundred more pictures than daisies, roses, and sunflowers. Since there are more photos of those pictures our model would have better performance classifying tulips and dandelions since it has seen more data of those flowers during training.

Given that our data did not have a fixed size and are slightly different proportions we needed to augment the data to not only generalize the problem at hand but to also fix the input to our CNN so it meets the requirements of our input. During image augmentation our data had random rotation up to 180, zoom up to 40%, horizontal or vertical shift of 40 pixels, and even flipped the images horizontally. That way our problem becomes more generalized and our network can learn patterns of each flower from different locations in the images. A example of the data augmentation is shown in Figure 2. From there we performed a 80-20 split for training-testing our data with random shuffling to help with generalization of the classification so that our models are not training and being tested on the same images every time we have to retrain our model.

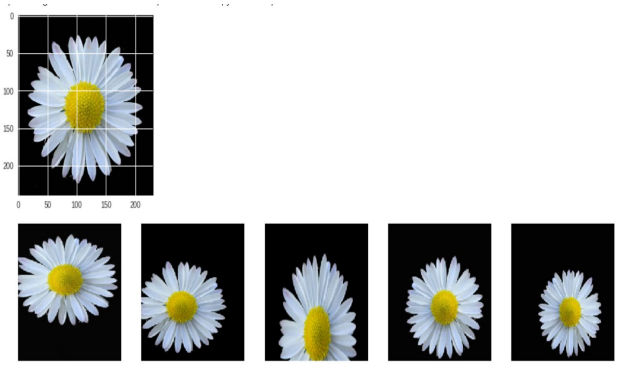

Figure 2: **Augmented Daisy**

Models and Algorithms

Before we ended up with our final model, our model went through a lot of different iterations. The first thing we tried was a basic CNN with one convolutional max pooling layer, with a dense layer and a flatten layer. And a final softmax layer for the multi classification. We found that after going through training our model only predicted a single class every time, so we needed to create a more complex model by increasing the depth of our network. In our next approach we took an input of a 320 x 240 image, and then had 3 convolutional max pooling layers. The filter size for

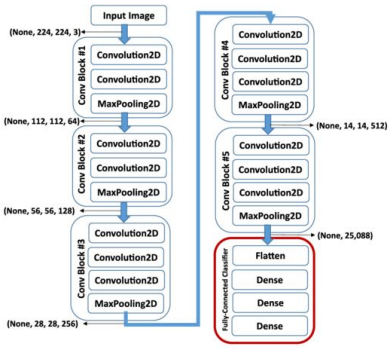

Figure 3: **The VGG16 model architecture**

each convolutional layer was 3x3, with max-pooling with a 2x2 filter, where the number of filters doubles after each layer starting with 16. The idea behind doubling the amount of features after every layer was to increase the middle layer and high level features as they produce features that grab the patterns from the flower within its images. All of our layers used the RELU activation function. Our dense layer had 512 hidden units with 50% dropout rate to prevent overfitting. This model resulted in a training-testing accuracy of 59%-60% for training-testing.Our next step was to try one of the pretrained models. We ultimately ended up trying VGG16, VGG19, ResNet50, and InceptionV3. All of these pretrained models were found through the Keras Applications Library. All 4 models utilize the ImageNet competition dataset, and produced relatively similar results with roughly low 90% on the training set and high 80 percent on the testing set when using their weights that were saved for each layer during the ImageNet competition. We ultimately went with the VGG16 model because for our specific problem it produced the best results, with 93% on the training set and 87% on the testing set. VGG16 was good at extracting features from the images so we decided to leverage its layers to implement our own customized model on top of VGG16.

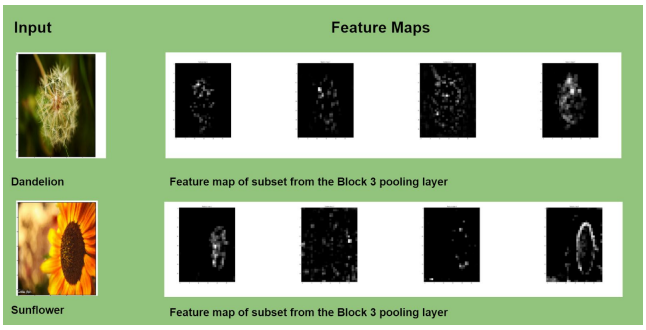

Figure 4: **Data and Feature Mapping**

After we settled with VGG16, we started to implement our final model. We extracted the first two convolutional max-pooling layers from VGG16 and froze them with the weights learned from the ImageNet competition to leverage feature extraction for our own CNN (these layers have two convolutional layers and 1 max-pooling). These layers in the network were able to extract good low and medium level features from the images. Then we created our own layers consisting of 3 convolutional max-pooling layers each with a single convolutional and max-pooling layer to customize how we extracted the higher level features from the images.

Figure 4 above shows the features extracted from a sunflower and a dandelion at the third max-pooling layer in our model, we can see that our model has obtained relatively high level features as it captures the shape and structure of both flowers. After flattening out the features from the last max-pooling layer, we used a Dense layer with a dropout rate 0.5 to prevent overfitting. In terms of some the hyperparameters of our model, The filter size for each convolutional layer is 3x3, max-pooling with 2x2 filter where the number of filters doubles after each convolutional max-pooling layer starting with 64 filters. Figure 5 shows the structure of our customized model with the inputs each layer takes and the activation for the convolution and dense layers.

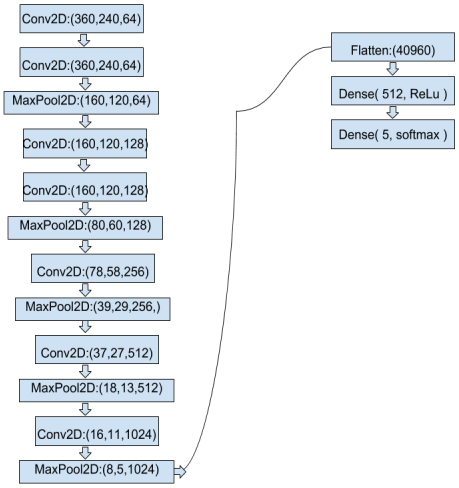

Figure 5: **Our final model, of our CNN built on top of VGG16**

Results and Analysis

For each of the experiments we ran trying and creating different networks we used the loss function of categorical cross entropy, the optimizer we used for minimizing the loss in our models is Adam with an initial learning rate of 0.0001, and we trained our model for 20 epochs. With the 80-20 split in our data we had 861 images for testing and 3462 images for training. The batch sizes for training were 91 images with 38 steps per epoch and for testing the batch sizes were 173 images with 5 steps per epoch. We used categorical cross entropy for our loss function because it generally is a good loss function to use when dealing with multi-classification. The way that it works is by taking the predicted probability that our network predicts a specific class, it increases when the probability diverges from the associated label of the given image. And so when the predicted probability gets closer to the label the loss the slower the loss decreases. The Adam optimizer is an optimizer that adjusts its learning rate for each of the parameters during training. It keeps track of the average decaying gradients similar to RMSprop and it also keeps track of the average decaying momentum in order to calculate the mean and variance of the gradients. They update its parameters by calculating the corrected biases for the mean and variance.

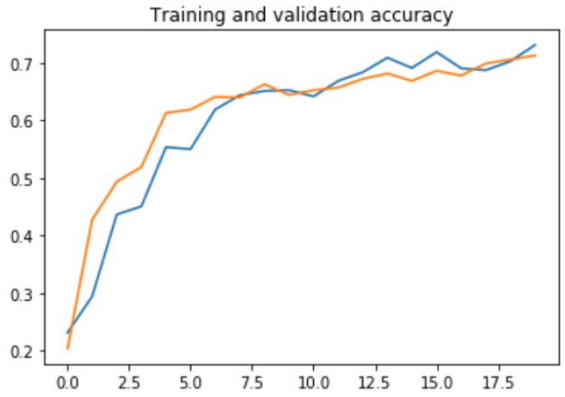

Figure 6: **Training (blue) and validation (orange) accuracy of our custom model**

In figure 6 and 7, we see our performance of our custom model, with our validation accuracy hovering in the mid 70% and our validation loss of roughly 67% in the final epoch. The graphs above are validation accuracy and validation loss, respectively, vs. the total number of epochs. We observed the benefit of leveraging VGG16 for our own customized network as a result. We have managed to boost our performance without overfitting our model by extracting the layers of VGG16 that learn the low and mid level patterns in the images. We believe freezing the first several layers of VGG16 with its ImageNet weights helped boosted

our results because VGG16 network complexity allowed it to converge to a minimal loss rather quickly, so we had the idea that if we were to leave the first few layers unaltered, and try to build our own custom CNN on top. We were then able to extract good high level features for the images and saw the loss increase dramatically from the first epoch up until the third-fourth epoch. We also believe that the loss decreases fairly quickly because our own customized network layers were randomly initialized while we set the weights for the VGG16 layers, and so with good features being extracted in the VGG16 layers our model needed to update its own parameters to reduce the loss. So we see our model update its trainable parameters and after the fourth epoch our model slowly starts to learn and decrease its loss at a constant rate. Looking at Figure 6 of the accuracies of the model at training and testing we observe that the accuracies may still improve without overfitting. We believe that since the accuracies are still increasing in general without plateauing and that our model will not overfit since the training and testing accuracies continue to intersect each with minimal accuracy distance.

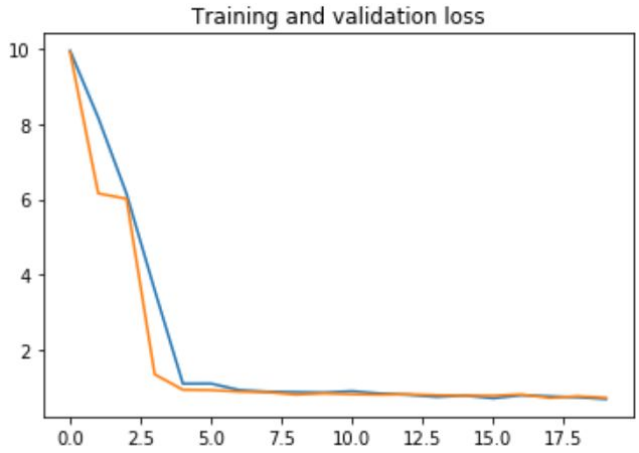

Figure 7: **Training (blue) and validation (orange) loss of our custom model**

Based on the graphs of figure 8 and 9, we can see that base VGG16 gives us the best performance, of 93% accuracy and a loss of 0.36 in the final epoch. Looking at Figures 7 and 8 we see a considerable difference in initial loss starting at the first epoch. We had originally hoped that our model would be able to produce slightly more comparable results to the base of VGG16, but this was not the case. One of the reasons the VGG16 model had performed considerably better in terms of loss initially (our model loss ≅ 10 and VGG16 loss ≅ 1) has to do with the initialization of the weights. In VGG16 we set each of the layer's weights to its ImageNet weights and from there allowed the network to train and update its parameters during each step in an epoch. And in our own custom model

Figure 8: **Training (blue) and validation (orange) loss of base VGG16**

all of the non-frozen layers were randomly initialized. So the initialization of the weights for VGG16 were much better allowing the model to generally have a good idea about what features of the images are important to the classification of each flower. This also explains why the VGG16 model was able to obtain significantly better accuracies, looking at Figures 6 and 9, in the same amount of epochs. However, we do believe that if we had enough time to run it for more epochs, we would have hit a comparable accuracy, assuming the model did not start to overfit.

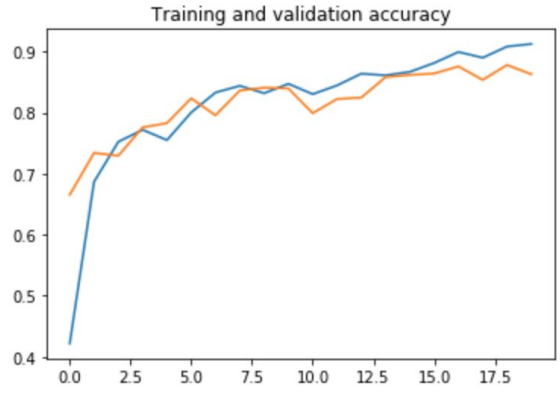

Figure 9: **Training (blue) and validation (orange) accuracy of base VGG16**

One thing that we found interesting was that our training loss for our custom model and VGG16 both initially have higher training loss than testing loss. We believe that the average training loss over each epoch is lower because it's the average of the losses calculated for each batch of data in one epoch. So to begin with the early batches of data during training time will have a high loss and the loss will generally decrease once it gets to the last few batches during training since the model is being updated after every batch with the optimizer implemented. So the initial losses at the beginning of each epoch are extreme values that throw off the average loss. And during testing, the loss returned is the loss at the end of evaluating the testing data. Overall we are satisfied to know that our model was able to perform relatively well at classifying flowers. Since we were able to produce a model that did not overfit the training data, it demonstrated that it did well in generalizing the problem at hand when it comes to dealing with data that it had never seen before. And that the model can produce similar results to VGG16 if it were to be trained for more epochs. We also learned how increasing the depth of our model may increase the computational complexity and how it benefits in the performance. So we have come to understand the trade-offs between depth, complexity of our network and how they affect the overall run time of our model.

Contribution

We all worked on the project equally. We would often meet during nights once a week during the start of the quarter, and almost three or four times a week towards the end of it on discord. Through our meetings we would often discuss various different approaches and work through different problems we were having with our code, such as trying to get our model to run on our entire dataset, since it would not work on the entire dataset on google collab, and other various errors we would be having with our model. Ultimately, through these meetings we all developed a full understanding of our project and worked together on essentially all portions of the project. However, we did specialize in some specific areas. Tyler wrote the abstract, perform data augmentation on the images, and organized the data charts, and helped build our final model. Prabhmeet worked on determining which pre trained model to use, implemented the neural network built on top of VGG16, worked on feature extraction, and helped build our final model. Tarun worked on properly reading in our dataset and creating a proper path between our code and the dataset, worked on initial iterations of our the model, and helped build our final model. Overall, we all feel that we contributed to the project an equal amount and definitely learned a lot about image recognition.

Future Work

In our future endeavours, we would like to try increase our validation accuracy without overtraining and overfitting on the training set. One thing we planned to do is "fine-tune" our model, which means we would use our previously learned network, and load all of the weights to the network, then we would freeze each layer of VGG16 up to the last CNN block, all the while making very small weight updates with different datasets. We will look into adjusting which layers would give us the best results and we will pick the best amongst them. The learning rate is probably the most important aspect of gradient descent, thus altering our learning rate could have a significant impact on our model. Our model here always made sure to make full use of the Keras' parameter for varying learning rate, if we were to use Keras' rmsprop optimizer, there is a good possibility that we can improve the accuracy. In addition, we could add more data for each of the five categories of flowers so that our model can better learn since more data usually mean better performance. We would also try to obtain a more equal distribution of flowers, since our dataset currently seems to be more weighted towards dandelions and tulips. There's also a matter of running more epochs, we ran 20 to get the accuracy 71%. We believe that if we run more, about 100 epochs, it could improve. We will also apply the idea of early stopping where we basically stop training if accuracy decreases, which might suggest some overfitting, and will utilize the best model. There's also "bagging", which is bootstrap aggregation. The idea is that we take the training data and separate them into their own packages, so there are now separate models with each having around 60% of the training data but 100% of the validation data. Then we take all the models and average them and the final models have better results. Ultimately, we would like to try more approaches that we were unable to get to during the 10 week quarter.

REFERENCES

[1] Espericueta, Rafael. Assignment 3 from Applied ML, CMPS 144 Winter 2019, Prof. Professor Narges Norouzi

[2] François Chollet. Keras Team .2019. Keras: Python] Deep Learning Library. Retrieved January 20, 2019 from https://keras.io/

[3] Kingma, Diederick P. and Ba, Jimmy Lei. Adam: A Method for Stochastic Optimization. 3rd International Conference for Learning Representations, San Diego, 2015.

[4] Li, Fei Fei. 2018. Convolutional Neural Networks for Visual Recognition. Retrieved February 10, 2019 from http://cs231n.stanford.edu/

[5] Simonyan, Karen & Zisserman, Andrew. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition.