Image Classification with Convolutional Neural Network

Elise Bishoff
University of Washington
Department of Applied Mathematics
1410 NE Campus Parkway, Seattle WA 98105

Abstract

We built a Convolutional Neural Network (CNN) for flower classification. In particular we tried to understand how the Convolutional Neural Network is working. We include a through theoretical background to help introduce CNN's. Then, we give a deep look into our process of choosing a model for this particular dataset. We specifically focused on how the different number of nodes, convolutional layers, maxpooling layers, and dense layers had on how classification accuracy. We also explored the effects of dropout rate, data augmentation, and many other things. In the end we were able to achieve a 75.3% and a .74 loss on our test set.

Sec. I. Introduction and Overview

Our overarching question is how well can we classify images of flowers based off a snapshot of them? Someday could we create an app that allows us to tell you the type of flower you are seeing based off just the image? Another focus of this project was to have a better understanding of Convolutional Neural Networks (CNN). Convolutional Neural Networks are known for working well with image classification. The CNN can take in a $h \times w \times 3$ colored image and output a single classification value. In fact, there are many great pretrained CNN's out in the field that only require minimum hyperparameter tuning to get great results. Since the goal for this project was not to only get great results, but to have a deeper understanding of how CNN's work we created a CNN somewhat from the ground.

In this paper we try to give a thorough discussion of the theoretical background of Convolutional Neural Networks. Then, we dive deeper into the algorithmic process of creating our CNN. Later, we discuss the overall process of exploring different models and testing different hyperparameter before settling on a final model. Then we try to understand possible explanations of the driving factors of what our results are telling us. Lastly, we conclude with some suggestions of other factors to consider to better train our model.

Sec. II. Theoretical Background

CNN's are Neural Networks that have convolutional layers. They are known for working well with image classification. Mathematically speaking, CNN's are just a huge matrix of weights and biases and when we put in our data we get an output based on what those weights and biases tell us. Usually the images are the input where they are sent to a different number of convolutional layers and pooling layers before being flattened to go through our dense layers. This process can be seen somewhat from Figure 2.1. We have our image, then our feature learning which we assume our model is learning important defining structures of our different classes through our convolutional layers with our pooling layers. Then we flatten our image to prepare for the fully connected layers and classification.

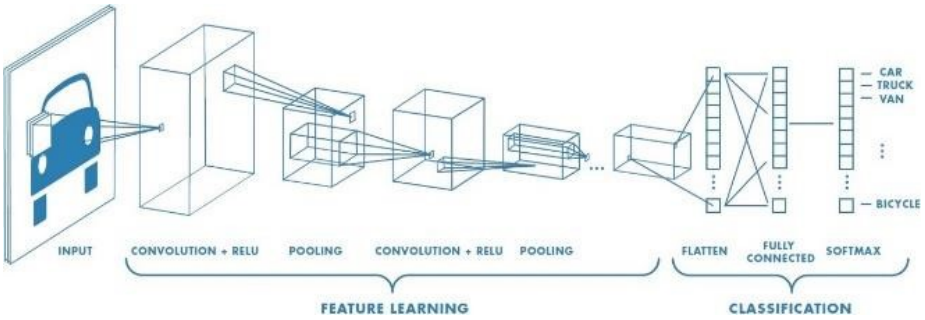

Figure 2.1 This tries to give a visual representation of what is going on in our CNN. First it takes the image where it learns the different features within the convolutional layers and maxpooling layers before flattening the image data into a vector and using a fully-connected layer to output the classification values.[1]

Forward propagation and Backpropagation

One of the most important things to understand for neural networks is what is going on in the forward and backward pass. This is how the model is developed and how we change our different weights and bias terms. In most simplistic terms we can think of neural networks as just a huge matrix of weights and biases that when we put in the input we simply are just calculating the output with those weights and biases.

So initially, in our neural network we have a forward propagation calculation which essentially calculates our output value based on our input. When using supervised training, the model recognizes when it has incorrectly labeled our data. A corrective step is then in place to change our weights and biases which is usually called backpropagation or backward pass.

The backpropagation algorithm helps us calculate the partial derivatives of our network variables to help calculate our gradient descent since we have such a big system. These partial derivatives from our backpropagation are then used in optimization algorithms such as gradient descent to minimize our loss function.

Gradient descent finds the path to the largest rate of change. Gradient descent is used to find local minimums or can easily find global minimums if we are dealing with a convex problem. The backpropagation algorithm shown in Eq. 2.2 is repeated to help with calculating our gradient descent. In this equation W are our weights, α is our learning rate, and $J(W)$ is our loss function that we are trying to minimize.[5] The learning rate essentially controls how fast we want to compute our gradient descent. The parameters of our neural network are then adjusted according to our gradient descent results.

$$W^j = W^j - \alpha\left(W^j\right)' J(W) \quad (Eq. 2.2)$$

The loss function that we used in this problem was categorical loss cross entropy. Cross entropy is calculated as shown in Eq 2.3. J is our loss function while y is true labels and $\hat{y}$ is our predicted labels. Also, M is the number of classes we have.

$$J = -\sum_{j}^{M} y_j ln\, \hat{y}_j \quad (Eq. 2.3)$$

Convolutional Layer

The great thing about CNN's is that their architecture takes into account that we are dealing with images. In the convolutional layer we have our neurons in 3D which are width, height, and depth which can be seen in Figure 2.4. Also, the convolutional layers are not fully connected which makes them distinctly different from dense layers. They are only connected to a certain region before it. Then, each layer takes our 3D image input and outputs a 3D output of neuron activations.

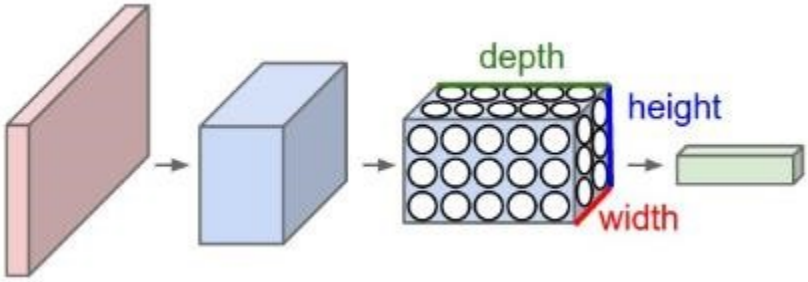

Figure 2.4 The red block represents our image input that would be $a \times b \times 3$ *in size. The blue blocks represent our convolutional layers that take in a 3D input and output some 3D volume. The green block represents our dense layer or output layer.*[2]

This layer consists of a set of learnable filters. These filters can be 3D in size to handle image's color components. During the forward pass, the filter slides across the image and computes dot products to produce a 2D activation map.[2] The idea that the layer is learning important feature components of the image. It learns edges, color, and patterns of the image that make it distinct.

Pooling Layer

Pooling layer is usually in-between different convolutional layers. It helps with overfitting and reduces model complexity by decreasing the size of parameters using a filter. Maxpooling is what was used and is a very common pooling layer. It takes the maximum value within the feature map. This can be shown in Figure 2.5.

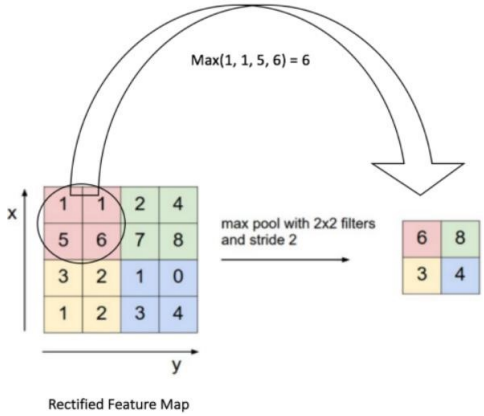

Figure 2.5 This shows how the maxpooling filter works. This one has a $(2,2)$ *filter that takes the max values in each filter.* [3]

Dense Layer (Fully-Connected Layer)

The fully-connected layer is a normal dense layer in a neural network. We flatten our matrix into a vector to get our data into a format to be able to work within the fully-connected layer.

Activation functions

Activations functions usually are nonlinear functions that we put our data through and essentially act like a summation. One of the most popular activation functions is RELU because they have been shown to work well with neural networks. There are a lot of different activations functions like Sigmoid (logistic), tanh, Leaky RELU, Gaussian, and many more. For this assignment, we mostly used RELU which function is in Eq. 2.7 The activation function essentially calculates a value that allows us to know whether that node should be activated or not in determining what our image is.

$$activation_w(x) = \sum_i w_i x_i \quad (Eq. 2.6)$$

$$f(x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases} \quad (Eq. 2.7)$$

Drop Out and Data Augmentation

Drop Out helps with preventing our model from overtraining. It randomly drops units and connections from the network while training. This helps so that the model doesn't become too dependent on a particular connection. This is shown in Figure 2.8.

Data augmentation takes into consideration a small dataset for a neural network. It allows us to 'add' more data to our dataset by creating more images from our already available images. It does this by randomly adding the same images but rotated, zoomed in, or modified in some way. This helps prevent overfitting and helps train the model.

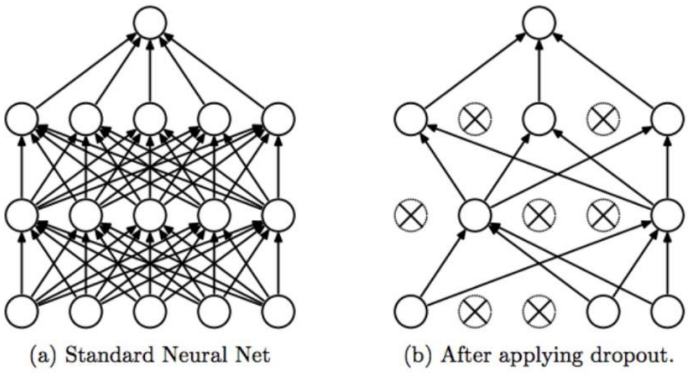

Figure 2.8 This shows us the difference between a normal neural net and a neural net that has had dropout applied.[4]

Limitations

Now, it is important to point out that there are many limitations with neural networks. One of the biggest complaints is their interpretability. It is extremely difficult to actually pinpoint what the

actual model is learning. Is it recognizing shapes? Is it recognizing curves? Or is it just memorizing the given dataset. It turns out to be very difficult to answer.

Also, they are computationally expensive because of how the sheer number of parameters needed to be calculated over and over again in the training process. The number of training samples needed also adds to the computational expense. Usually, to train a good neural network we need 1,000's of images within each class. With images, we need these 1,000's of images to be pictures at different angles, times, lighting, etc.

Also, neural networks can be very sensitive to small changes. Changes to the coloring of the image can cause the whole network to break.[8] Lastly, neural networks usually can only be used for interpolation problems and not extrapolation.

Sec. III. Algorithm Implementation and Development

The overall process in developing our CNN was to first perform essentially a grid search on most of the hyperparameters. Then, we performed some more hyperparameter tuning before deciding on a single model. Below we highlight the key points of our algorithmic process with reference to the code.

- In Section 1 we simply are loading all the Python packages needed to create our CNN.
- In Section 2 we load our images. We resize them to be $100 \times 100 \times 3$.
- In Section 3 we plot various images from our dataset to get an idea of the kind of images we are dealing with for this project.
- In Section 4 we normalize our images by dividing all values by the max value of 255. Then we create the labels for the data for our various flowers.
- In Section 5 we create our training, validation, and test dataset. Then, we covert our labels for each dataset to one hot encoding so that way they can work within the framework of neural networks.
- In Section 6 we test an initial model.
- In Section 7 we do a grid search for the best number of dense layers, convolutional layers, and layer sizes.
- Since I've heard that we should build our model and then regulate it, then in Section 8 we prepare to do data augmentation with our model.
- In Section 9, with our three best models we test then with dropout and data augmentation.
- In Section 10, we do some more hyperparameter tuning by trying out three different activation function on our best model to see if we get a better result.
- In Section 11, we test to see if more epochs on our final model improve it's overall accuracy between 3 and 4 convolutional layers.
- In Section 12 we have our final model and we create a confusion matrix on the results of our final model with the test data to try to better understand the misclassifications.
- In Section 13 we create a classification report on the test dataset.
- In Section 14 we plot misclassified images to try to better understand what is going on.

Sec. IV. Computational Results

Exploration of Models

First off, our flower dataset was obtained from Kaggle.com[5], which is a well-known site that contains many datasets. The dataset contains a total of 4,423 flower images. We have five different classification categories which include 769 daisies, 1052 dandelions, 784 roses, 734 sunflowers, and 984 tulips. We specifically chose this dataset because some of the flowers of the images are very distinct while others seem to be hiding or less recognized in the images as shown in Figure 4.1. We wanted to see if this mix of images could still work in training a CNN. One of the problems with this dataset is that we have a very small dataset considering that we are trying to train our model using a CNN.

Figure 4.1 Sample images from our dataset. The top represents images where the flower seems to be front and center whereas the bottom images are flowers that seem to be in the background or other objects are in the image that seem to detract from the flower.

Before working with the data, we conducted a lot of research on CNN's. We tried to understand everything about convolutional networks. Then we started with preprocessing the data. We resized all images to be $100 \times 100 \times 3$ simply because the flowers still looked distinct. Then we separated the data into 80% training data, 10% validation data, and 10% test data because that is usually what is standard practice.

Next, we worked with Python's popular packages Keras and TensorFlow to create our CNN. Initially, we randomly chose a CNN that had 2 convolutional layers, with 2 maxpooling layers,1 dense layer, and 32 nodes for each layer other than the first convolutional layer. This gave us around a 55% accuracy rate on the validation dataset after 5 epochs and around a 90% accuracy rate on the training data which probably means that we are overtraining. I only did 5 epochs because of my computer limitation. Each model took about 10 minutes to run.

After initial models, we tried every combination of dense layers being 0, 1, or 2, convolutional layers being 1, 2, 3 and node sizes being 32, 64, 128. The results for these models can be seen in Figure 4.2. We see from Figure 4.2 that we have a wide range of accuracies based on the different models. Also, in Figure 4.3 we see how the loss value decreases over time with these different models. In Figure 4.4 and 4.5 we see the same scores but for the validation set.

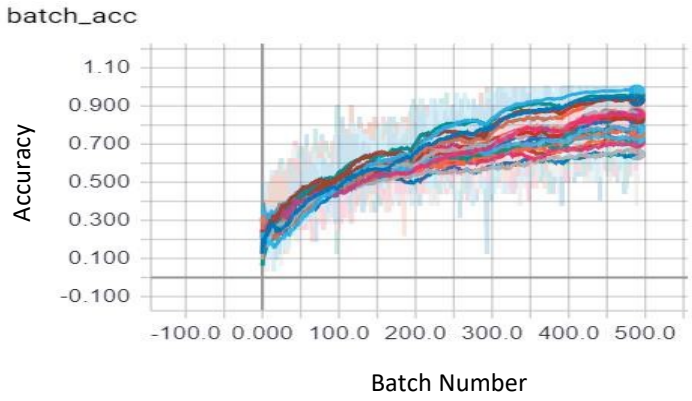

Figure 4.2 The accuracy score on each batch. Since batch size was 32 and we had about 3200 images in the training dataset which then caused 100 iterations for each epoch and since we did 5 epochs we ended up with 500 iterations. Used linearization so the faded colors in back are the actual values for each iteration.

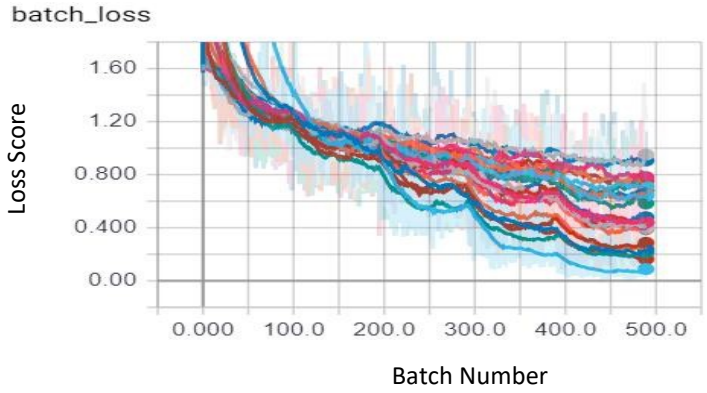

Figure 4.3 The loss value for each iteration of the training dataset. Our loss is categorial cross entropy.

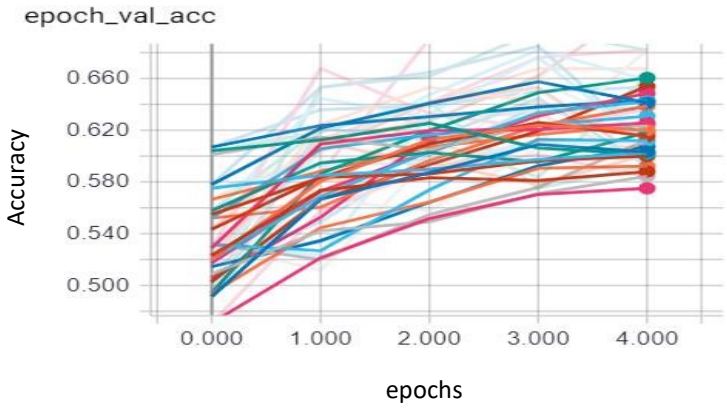

Figure 4.4 The accuracy score on validation set for each epoch. The washed-out colored lines are the actual values.

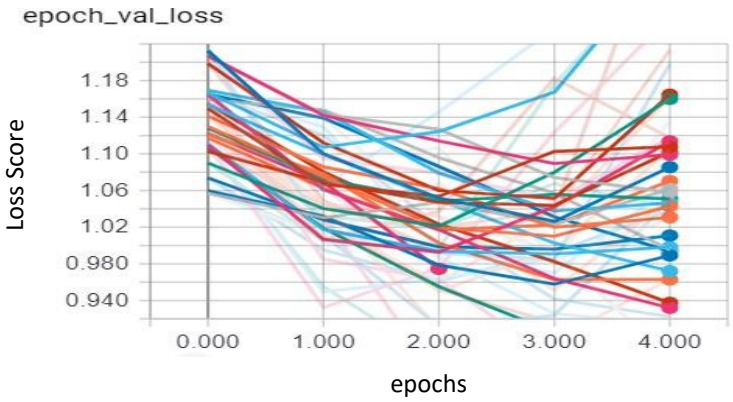

Figure 4.5 The loss score on validation set for each epoch. Used some linearization. The washed-out colored lines are the actual values. We can see that some models continued to decrease their loss value while others started to increase after a certain threshold.

We specifically only used the values from Figure 4.4 and 4.5 to pick the best three models. We picked three models that had high validation accuracy with low loss values. The three models picked we called Model 1, Model 2, and Model 3. All the models have 3 convolutional layers. Model 1 has 0 dense layers and 32 nodes. Model 2 has 1 dense layer and 64 nodes. Model 3 has 2 dense layers and 64 nodes.

With these three models, I performed data augmentation with a dropout rate of 20% on 10 epochs. I chose 20% dropout rate because it seemed to be a standard dropout rate. This takes some of the nodes randomly and ignores them during training.

In Figure 4.6 we can see the batch accuracy for the final three models. Figure 4.7 and Figure 4.8 look at the accuracy score and loss value for the validation dataset. I think it is important to recognize that in Figure 4.6 with our linearization we get that our accuracy scores of around 70% for our training data instead of around 90% for some cases in Figure 4.2.

Again, from the accuracy score and loss value on the validation dataset, I picked the top performing model. Model 3, which is in silver in the graphs, outperformed the other models in both loss value and accuracy.

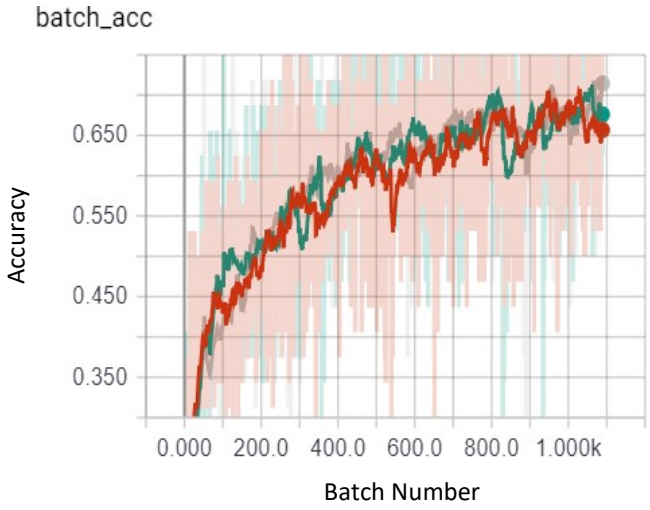

Figure 4.6 The final three model's accuracy score on each iteration. Used linearization so the washed-out colors in back are the actual values for each iteration. Model 1(Red) Model 2(Green) Model 3 (Silver)

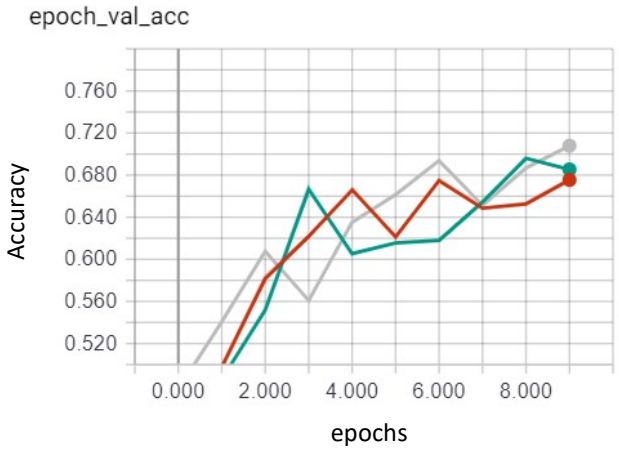

Figure 4.7 The accuracy score on the validation dataset for each epoch. Model 1(Red) Model 2(Green) Model 3 (Silver)

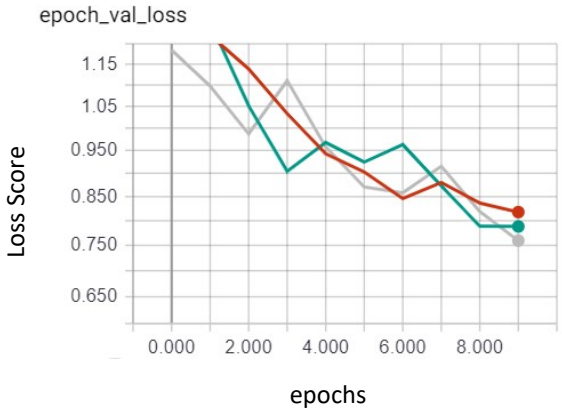

Figure 4.8 The loss value on the validation set for each epoch. Model 1(Red) Model 2(Green) Model 3 (Silver)

The next step I took with hyperparameters was testing out different activation functions. I tried out Sigmoid and Tanh against our RELU which can be seen in Figure 4.9. We weren't really surprised by the results, but were glad to conduct them. We see that RELU, which is in blue, performed the best our of all three. This just validated our suspicions on the matter. Afterwards, we tried some different hyperparameter tuning such as changing the node sizes to be all different instead of the same, and adding stride within our maxpooling layer, but nothing interesting came out of these tests.

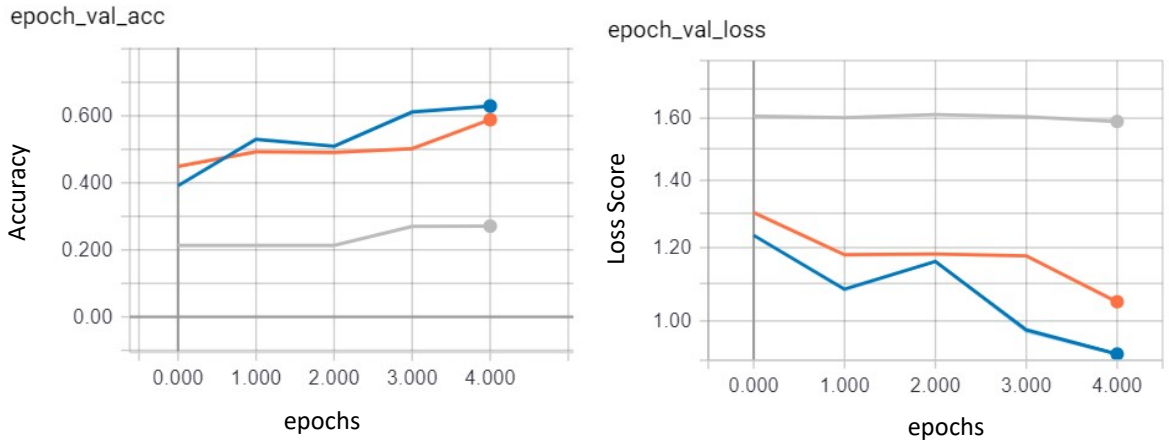

Figure 4.9 The accuracy and loss score for our three different activation functions. Blue: RELU activation. Orange: Tanh activation. Silver: Sigmoid activation.

Now, we decided to see what would happen if we added one more convolutional layer. We ran three different times of having four different convolutional layers and compared it to our best model, Model 3. The light blue, dark blue, and red are models with four convolutional layers and the silver is the model with three convolutional layers. We can see in Figure 4.10 that the extra layer seemed to make a significant improvement.

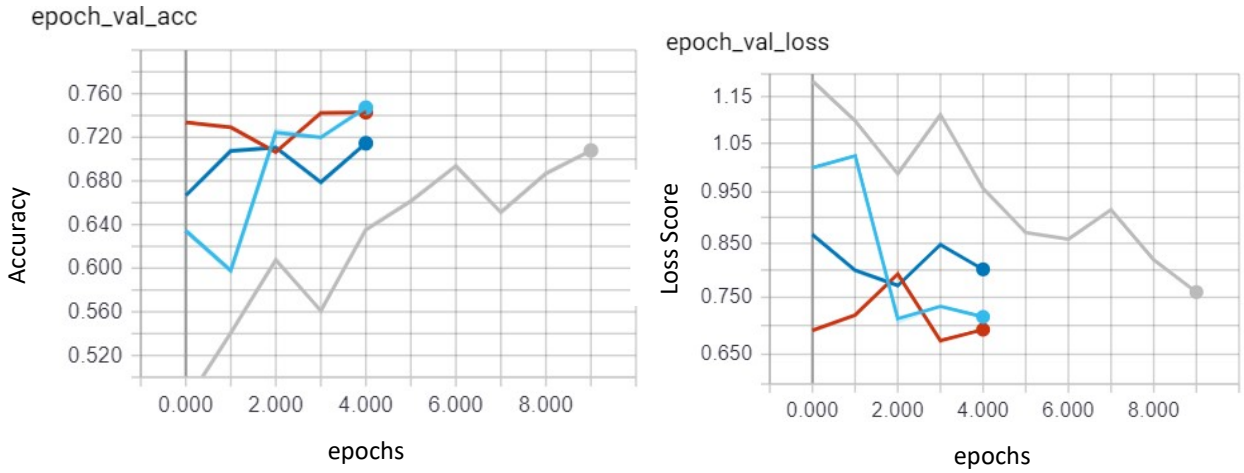

Figure 4.10 Light blue, dark blue, and red all have 4 convolutional layers whereas the silver model has 3 convolutional layers. Every other parameter is the same. We can see that we have better results with a model that has 4 convolutional layers.

Thus, as a final comparison, I compared a 25 epoch iteration 4 convolutional layers again a 25 epoch iteration 3 convolutional layers which can be seen in Figure 4.11. Each took over 1.5 hours to run. I thought the 4 convolutional layer would blow the 3 convolutional layer model out of the water, but they were actually pretty comparable in validation accuracy and validation loss. For the last epoch though, our 4 convolutional layered net(pink) outperformed our 3 convolutional layered net(blue). Therefore, we decided that our final model would be the four convolutional layered neural net. We are now ready to test it on our test dataset.

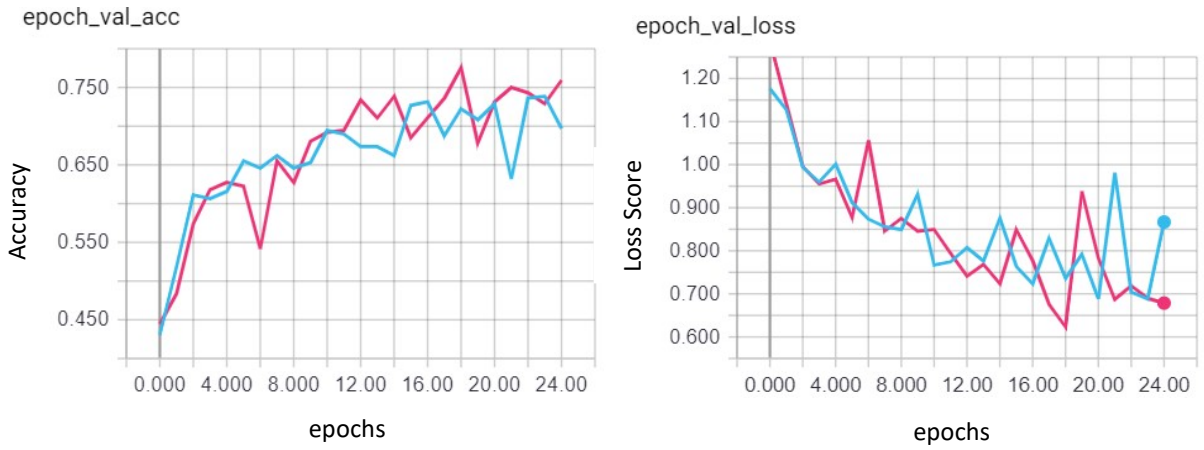

Figure 4.11 Here we compared our final models. The blue one is our Model 3 from above where we have only 3 convolutional layers whereas our pink is our new model with 4 convolutional layers. All other hyperparameters are the same for each one.

Final Results

Our baseline model was choosing the most frequent class. This happened to be our dandelions which would give us a 24% accuracy. Also, I implemented the data into a SVM with no blows or whistles and got an accuracy of 25.5%. We improved significantly from this baseline to getting an accuracy of 75.3% and a .74 loss value on our test set with our final neural net model.

In the end our final model had the following features:

- Input: $100 \times 100 \times 3$ colored flower image
- Convolutional Layers: 4 where each had a (3,3) filter with RELU activation.
- Maxpooling Layers: 4 with (2,2) filter where each maxpooling layer followed a convolutional layer
- Dense Layers: 2 with RELU activation
- Last layer contains Softmax activation
- Nodes for each layer: 64
- Loss Function: Categorical Cross Entropy
- Learning Rate Optimizer: Adam Optimizer with initial learning rate of .001

- Epochs: 25
- Dropout Rate: 20% after dense layer
- Image Augmentation
- Total parameters: 182,661

I chose convolution layer, maxpooling layers, dense layers, and nodes based off of the analysis in the section above. Epochs decision was based off of the capacity of my computer. Dropout rate was based off what seems to be the standard dropout rate. I implemented data augmentation because of our limited dataset size for a neural network. I chose (3,3) filters and (2,2) filters for convolutional layers and maxpooling layers respectively based on what seemed to be standard. I chose RELU based off of the research that has been shown that RELU works well with neural nets. I chose Adam optimizer for our learning rate because it has been proven to be a better optimizer usually compared to others. I arbitrarily chose image input size based on whether I could see the details in the image.

Analysis of Results

Here we have some analysis on our results. Once again, our overall accuracy on our test dataset was 75.3% with a low loss value of .74. This was a significant improvement from our baseline. Mostly, I looked at where the model failed and tried to find reasons for why that could have happened.

In Figure 4.12 we have our classification report. Precision deals with the number of accurately classified elements over the sum of the true positives and false positives. Recall is the number of accurately classified elements in a class over the sum of the true positives and false negatives. The f1 score is the average between precision and recall. Here we see that the sunflower and rose seemed to struggle with precision. Since the sunflower has high recall and low precision then our we are classifying many things as sunflower, but more of them are incorrect than correct. The rose is just overall low compared to the other classifiers. The dandelion did the best overall.

	precision	recall	f1-score	support
daisy	0.80	0.76	0.78	72
dandelion	0.89	0.79	0.84	112
rose	0.68	0.61	0.65	88
sunflower	0.62	0.92	0.74	62
tulip	0.76	0.72	0.74	99
avg / total	0.76	0.75	0.75	433

Figure 4.12 Classification report on our test data.

In Figure 4.13 we have our confusion matrix on our test data. We see that the Sunflower classified best with getting an 92% accuracy, but we also saw from the classification report that the sunflower had high recall and low precision.

The rose performed the worst with only getting a 61% accuracy classification rate. It follows to then make sense that the highest misclassification was with rose with the tulip. It was predicted most often to be a tulip when in fact it was a rose. Also, next in misclassification was that it was predicted to be a rose when in fact it was a tulip. Our model really struggled to recognize the rose.

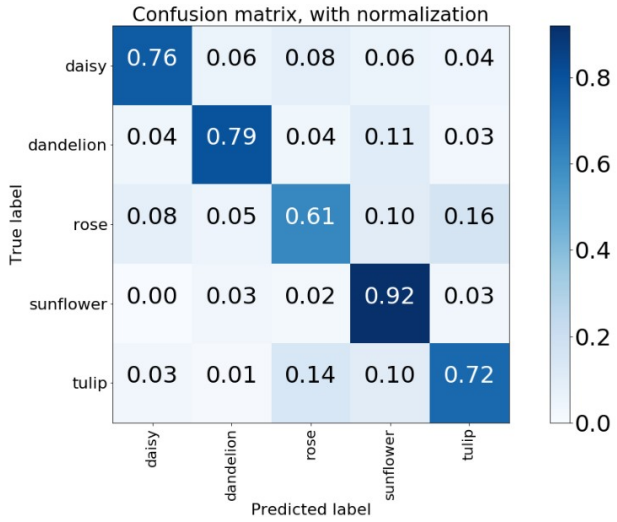

Figure 4.13 Normalized Confusion Matrix on our test dataset

I tried to make sense of this struggle. In Figure 4.14 I have some of the images that were improperly classified. For the top row we have predicted tulips when in fact they were roses. It seems to be that the busy images caused it some issues. Also, it does seem that roses and tulips have a similar curve of their petal shape. In the middle row we have predicted roses when in fact they were tulips. These all look distinctly like tulips to me and they weren't 'busy' images which caused me to question my previous assumptions and be a little concern about what the neural net is actually learning.

Obviously, I can't be certain the actual problem the CNN has with tulips and roses. I also included some other top misclassification images in the bottom in Figure 4.14.

Figure 4.14 Images that were incorrectly misclassified on the test dataset. Top: Predicted Tulip when True label was Rose. Middle: Predicted Rose when True label was Tulip. Bottom Left Two: Predicted Sunflower when it was a Rose. Bottom Right Two: Predicted Sunflower when True label was Tulip.

Sec. V. Summary and Conclusion

In conclusion, we were able to create a Convolutional Neural Net for this particular flower classification problem. We were able to obtain 75.3% with a low loss value of .74 on our test dataset with our final model. We looked into what our top misclassifications which happened to be with the Tulip and Rose.

If I had more time, I would have worked more on preprocessing of the flower dataset. I would have loved to somehow extract the face of the flower and perform Principal Component Analysis on the images and see if that helped the same structured CNN perform better. Or if the image size had any affect on the CNN. Could I get as good of results even if my image were $50 \times 50 \times 3$ or even $10 \times 10 \times 3$?

Lastly, I found a pretrained CNN, Resnet[6], that seemed to work really well on this dataset. The baseline for the Resnet for this dataset was an 88% accuracy and with some finetuning of the hyperparameters, a 94% accuracy was achieved. This could have been an improvement of working with a pretrained CNN, but I really wanted to gain the experience of working on creating a CNN from scratch.

Overall, we were able to outperform our given baseline by a significant amount. While there is a lot more that can be done, we have successfully started creating a CNN that classifies flowers.

Appendix A Python functions used and brief implementation explanation

Keras/ TensorFlow: Keras builds runs on top of TensorFlow. It allows us to create neural networks using a high level language.

- *Activation:* This allows us to determine the activation function we will use in our model.
- *Conv2D:* This allows us to create a convolutional layer.
- *Dense:* This allows us to create a fully connected layer within our neural network.
- *Dropout:* This allows us to add a dropout component to our neural network.
- *Flatten:* This reformats our data into a manageable format for fully connected layers.
- *ImageDataGenerator:* This allows us to do data augmentation during our training process for the neural network.
- *MaxPooling2D:* This allows us to create a maxpooling layer.
- *Model.compile(loss=a, optimizer=b, metrics=[c]):* This allows us to compile our neural network where we are focused on the give loss, optimizer, and metric measures.
- *Sequential:* This allows us to build a linear stack of layers within our model for our neural network.
- *TensorBoard:* Visual tool that allows you to visualize information about your different models for your neural network.
- *To_categorical:* This changes our class data into a one hot encoded class label.

Itertools: This allows us to create iterator for efficient looping.

Matplotlib.pyplot: This library allows us to create plots in python that usually are the same syntax as MATLAB plots.

Numpy: This packages stands for Numerical Python. It is a scientific computing library.

os: This package allows us to use operating system functionality.

- *os.path.join:* returns a string that joins the two paths together.

Pandas: This package allows us to have structed datasets.

PIL: This package stands for Python Imaging Library. This helps us manipulate images.

- *Image:* module that allows us to work on images. For example, Image.open opens an given image file.

Sklearn: This package is a machine learning package. It has lots of tools available for data analysis.

- *Accracy_score:* Gives us the accuracy score of our model with the given labels.
- *Classification_report:* Builds a report that shows the different accuracy scores for our different classes.
- *Confusion_matrix:* This creates a confusion matrix on the given inputs.
- *Train_test_test_split(x,y, test_size=a):* This allows us to split our data into training and test dataset.

Time: This allows us to get the time.

References

1 Prabhu, and Prabhu. "Understanding of Convolutional Neural Network (CNN) - Deep Learning." Medium. March 04, 2018. Accessed March 21, 2019. https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.

2 CS231n Convolutional Neural Networks for Visual Recognition. Accessed March 21, 2019. http://cs231n.github.io/convolutional-networks/.

3 Nehme, Adel. "Understanding Convolutional Neural Networks." Towards Data Science. May 24, 2018. Accessed March 22, 2019. https://towardsdatascience.com/understanding-convolutional-neural-networks-221930904a8e.

4. Budhiraja, Amar. "Learning Less to Learn Better - Dropout in (Deep) Machine Learning." Medium. December 15, 2016. Accessed March 22, 2019. https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5.

5 Mamaev, Alexander. "Flowers Recognition." Kaggle. June 28, 2018. Accessed March 21, 2019. https://www.kaggle.com/alxmamaev/flowers-recognition.

6 "Flower Recognition - FastAI 94% Accuracy." Kaggle. Accessed March 21, 2019. https://www.kaggle.com/sominwadhwa/flower-recognition-fastai-94-accuracy.

7 "Everything You Need to Know about Neural Networks and Backpropagation - Machine Learning Made Easy..." Towards Data Science. January 14, 2019. Accessed March 22, 2019. https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

8. Hosseini, Hossein, Baicen Xiao, Mayoore Jaiswal, and Radha Poovendran. "On the Limitation of Convolutional Neural Networks in Recognizing Negative Images." *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017. doi:10.1109/icmla.2017.0-136.

Appendix B Python Code

Section 1: Here we have the code needed to create a Convolutional Neural Network for classifying a flower image dataset.

In [ ]:
```
#Here we have all the packages needed to create our Neural Net
import os
import numpy as np
from PIL import Image
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, Conv2
D, MaxPooling2D
import pandas as pd
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import TensorBoard
import time
from sklearn.cross_validation import train_test_split
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import classification_report, confusion_matrix,accuracy_s
core
import itertools
```

Section 2: Here we upload the images and resize them to by 100 by 100 by 3

In [ ]:
```
#this function just takes the images and opens them, resizes them,
#and puts them in an array
def LoadDir(dirname):
    imgs = []
    for imgname in os.listdir(dirname):
        img = Image.open(os.path.join(dirname, imgname))
        img = img.resize([100, 100])
        img = np.squeeze(np.array(img)[:, :, :])
        imgs.append(img)
    return np.array(imgs)
#here I load the images
daisy_imgs = LoadDir('C:\\Users\\ebish\\Documents\\data analysis 582\\flowers
\\daisy')
print(daisy_imgs.shape)
dandelion_imgs = LoadDir('C:\\Users\\ebish\\Documents\\data analysis 582\\flow
ers\\dandelion')
print(dandelion_imgs.shape)
rose_imgs = LoadDir('C:\\Users\\ebish\\Documents\\data analysis 582\\flowers\\
rose')
print(rose_imgs.shape)
sunflower_imgs = LoadDir('C:\\Users\\ebish\\Documents\\data analysis 582\\flow
ers\\sunflower')
print(sunflower_imgs.shape)
tulip_imgs = LoadDir('C:\\Users\\ebish\\Documents\\data analysis 582\\flowers
\\tulip')
print(tulip_imgs.shape)
```

Section 3: Here we create a plot of some of the images in the dataset just to get an idea of what we are working with.

In [ ]:
```
#plotting the different images
plt.subplot(1,5,1)
plt.imshow(daisy_imgs[1])
plt.axis('off')
plt.subplot(1,5,2)
plt.imshow(dandelion_imgs[1])
plt.axis('off')
plt.subplot(1,5,3)
plt.imshow(rose_imgs[1])
plt.axis('off')
plt.subplot(1,5,4)
plt.imshow(sunflower_imgs[1])
plt.axis('off')
plt.subplot(1,5,5)
plt.imshow(tulip_imgs[8])
plt.axis('off')
```

Section 4: We normalize the images and then create labels for the flowers and combine all of it together.

In [ ]:
```
#normalize the data
daisy_imgs=daisy_imgs/255
dandelion_imgs=dandelion_imgs/255
rose_imgs=rose_imgs/255
sunflower_imgs=sunflower_imgs/255
tulip_imgs=tulip_imgs/255
#creating the labels for the data by finding their lenght and then
#creating an array that holds thier class as an integer
total=len(daisy_imgs)+len(dandelion_imgs)+len(rose_imgs)+len(sunflower_imgs)+l
en(tulip_imgs)
y=np.zeros(shape=(total,1))
a=len(daisy_imgs)
b=len(dandelion_imgs)
c=len(rose_imgs)
d=len(sunflower_imgs)
e=len(tulip_imgs)
y[a:a+b-1]=1; #dandelions
y[a+b:a+b+c-1]=2; #rose
y[a+b+c:a+b+c+d-1]=3; #sunflower
y[a+b+c+d:a+b+c+d+e-1]=4; #tulip
#combine all the flower data
allflowers=np.vstack([daisy_imgs, dandelion_imgs, rose_imgs, sunflower_imgs,tu
lip_imgs])
```

Section 5: Created the training, validation, and test dataset

In [ ]:
```
X_train, X_testval, y_train,y_testval=train_test_split(allflowers,y,test_size
=.20)
X_validate, X_test,  y_validate, y_test=train_test_split(X_testval, y_testval,
test_size=.5)
```

In [ ]:
```
#our nerual net will take the y dataset and turn in essetially into
#one hot encoding
y_train=to_categorical(y_train, num_classes=5)
y_test=to_categorical(y_test, num_classes=5)
y_validate=to_categorical(y_validate, num_classes=5)
```

Section 6: Here we created an initial convolutional nerual net. We have 2 convoltuional layers with maxpooling layers followed after each one. Then we go straight to the output layer.

In [ ]:
```
#initial model has 2 convolutaional layers where our filter is (3,3)
#with a RELU activation function. Two maxpooling layers that follow
#each convolutional layer with a filter of the size (2,2)
#No dense layer other than to compute the output with a softmax
#activation function. Loss=categorical crossentropy with a Adam
#optimizer.
model1=Sequential()
model1.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
model1.add(Activation("relu"))
model1.add(MaxPooling2D(pool_size=(2,2)))

model1.add( Conv2D(64,(3,3) ))
model1.add(Activation("relu"))
model1.add(MaxPooling2D(pool_size=(2,2)))

model1.add(Flatten())

model1.add(Dense(5))
model1.add(Activation('softmax'))

model1.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model1.fit(X_train, y_train, batch_size=32, epochs=5, validation_split=.1, cal
lbacks=[tensorboard])
```

Section 7: Here we create the loop that tests out every combination of dense layer, nodes, and convolutional layers.

In [ ]:
```
#different testing values
dense_layers=[0, 1, 2]
layer_sizes=[32, 64, 128]
conv_layers=[1,2,3]

#for loop to try out the different models where it will try every
#combination of dense_layers, layer_sizes, and conv_layers.
for dense_layer in dense_layers:
    for layer_size in layer_sizes:
        for conv_layer in conv_layers:
            #this connects with tensorboard and allows us to plot the
            #accuracy of each model that we iterate through
            NAME="{}-conv-{}-nodes-{}-dense-{}".format(conv_layer, layer_size,
dense_layer, int(time.time()))
            tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
            print(NAME)
            model1=Sequential()
            model1.add( Conv2D(layer_size,(3,3), input_shape=X_train.shape[1
:]) )
            model1.add(Activation("relu"))
            model1.add(MaxPooling2D(pool_size=(2,2)))

            for l in range(conv_layer-1):
                model1.add( Conv2D(layer_size,(3,3) ))
                model1.add(Activation("relu"))
                model1.add(MaxPooling2D(pool_size=(2,2)))

            model1.add(Flatten())
            for l in range(dense_layer):
                model1.add(Dense(layer_size))
                model1.add(Activation("relu"))

            model1.add(Dense(5))
            model1.add(Activation('softmax'))
            #compile the model
            model1.compile(loss="categorical_crossentropy",optimizer="adam",me
trics=['accuracy'])
            #train the model with the given data
            model1.fit(X_train, y_train, batch_size=32, epochs=5, validation_s
plit=.1, callbacks=[tensorboard])
```

Section 8: Here we prep to do data augmentation

In [ ]:
```
#Here we do some data augmentation to help us have more training samples
X_train_ch=X_train
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the da
taset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees, 0
 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.2,  # randomly shift images horizontally (fraction
of total width)
        height_shift_range=0.2,  # randomly shift images vertically (fraction
 of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images


d=datagen.fit(X_train_ch)
```

Section 9: Here we create our model1, model2, and model3 with newly added dropout and data augmentation

In [ ]:
```
#model 1
NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-dgm".format(3, 32, 0, int(time.time
()))
tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
print(NAME)
model1=Sequential()
model1.add( Conv2D(32,(3,3), input_shape=X_train.shape[1:]) )
model1.add(Activation("relu"))
model1.add(MaxPooling2D(pool_size=(2,2)))


model1.add( Conv2D(32,(3,3) ))
model1.add(Activation("relu"))
model1.add(MaxPooling2D(pool_size=(2,2)))

model1.add( Conv2D(32,(3,3) ))
model1.add(Activation("relu"))
model1.add(MaxPooling2D(pool_size=(2,2)))

model1.add(Flatten())
model1.add(Dropout(0.2))

model1.add(Dense(5))
model1.add(Activation('softmax'))

model1.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model1.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                 epochs = 10, validation_data=(X_validate,y_valid
ate), callbacks=[tensorboard])
```

In [ ]:
```
#model 2
NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-dgm2".format(3, 64, 1, int(time.tim
e()))
tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
print(NAME)
model2=Sequential()
model2.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
model2.add(Activation("relu"))
model2.add(MaxPooling2D(pool_size=(2,2)))


model2.add( Conv2D(64,(3,3) ))
model2.add(Activation("relu"))
model2.add(MaxPooling2D(pool_size=(2,2)))

model2.add( Conv2D(64,(3,3) ))
model2.add(Activation("relu"))
model2.add(MaxPooling2D(pool_size=(2,2)))

model2.add(Flatten())
model2.add(Dense(64))
model2.add(Activation("relu"))
model2.add(Dropout(0.2))

model2.add(Dense(5))
model2.add(Activation('softmax'))

model2.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model2.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                  epochs = 10, validation_data=(X_validate,y_valid
ate), callbacks=[tensorboard])
```

In [ ]:
```
#model 3
NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-dgm2".format(3, 64, 2, int(time.tim
e()))
tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
print(NAME)
model3=Sequential()
model3.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
model3.add(Activation("relu"))
model3.add(MaxPooling2D(pool_size=(2,2)))


model3.add( Conv2D(64,(3,3) ))
model3.add(Activation("relu"))
model3.add(MaxPooling2D(pool_size=(2,2)))

model3.add( Conv2D(64,(3,3) ))
model3.add(Activation("relu"))
model3.add(MaxPooling2D(pool_size=(2,2)))

model3.add(Flatten())
model3.add(Dense(64))
model3.add(Activation("relu"))

model3.add(Dense(64))
model3.add(Activation("relu"))
model3.add(Dropout(0.2))

model3.add(Dense(5))
model3.add(Activation('softmax'))

model3.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model3.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                  epochs = 10, validation_data=(X_validate,y_valid
ate), callbacks=[tensorboard])
```

Section 10: Here we test different acitvaiton functions on our best model from Section 9.

In [ ]:
```
#testing different activaiton functions
act=["sigmoid","tanh", "relu"]


for act_layer in act:
    NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-{}-activation-dgm2".format(3, 6
4, 2,act_layer, int(time.time()))
    tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
    print(NAME)
    model4=Sequential()
    model4.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
    model4.add(Activation(act_layer))
    model4.add(MaxPooling2D(pool_size=(2,2)))


    model4.add( Conv2D(64,(3,3) ))
    model4.add(Activation(act_layer))
    model4.add(MaxPooling2D(pool_size=(2,2)))

    model4.add( Conv2D(64,(3,3) ))
    model4.add(Activation(act_layer))
    model4.add(MaxPooling2D(pool_size=(2,2)))

    model4.add(Flatten())
    model4.add(Dense(64))
    model4.add(Activation(act_layer))
    model4.add(Dropout(0.2))

    model4.add(Dense(64))
    model4.add(Activation(act_layer))


    model4.add(Dense(5))
    model4.add(Activation('softmax'))

    model4.compile(loss="categorical_crossentropy",optimizer="adam",metrics=[
'accuracy'])

    model4.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                      epochs = 5, validation_data=(X_validate,y_va
lidate), callbacks=[tensorboard])
```

Section 11: now test between our two final models with 3 convolutional layers vs 4 convolutional layers. We test to see if more epochs will increase the overall accuracy on our validation dataset.

In [ ]:
```
NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-{}-activation-dgm2".format(3, 64, 2
,"relu", int(time.time()))
tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
print(NAME)
model6=Sequential()
model6.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
model6.add(Activation("relu"))
model6.add(MaxPooling2D(pool_size=(2,2)))


model6.add( Conv2D(64,(3,3) ))
model6.add(Activation("relu"))
model6.add(MaxPooling2D(pool_size=(2,2)))

model6.add( Conv2D(64,(3,3) ))
model6.add(Activation("relu"))
model6.add(MaxPooling2D(pool_size=(2,2)))

model6.add(Flatten())
model6.add(Dense(64))
model6.add(Activation("relu"))
model6.add(Dropout(0.2))

model6.add(Dense(64))
model6.add(Activation("relu"))


model6.add(Dense(5))
model6.add(Activation('softmax'))

model6.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model6.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                  epochs = 25, validation_data=(X_validate,y_v
alidate), callbacks=[tensorboard])
```

In [ ]:
```
NAME="{}-conv-{}-nodes-{}-dense-{}-dropout-{}-activation-dgm2".format(4, 64, 2
,act_layer, int(time.time()))
tensorboard=TensorBoard(log_dir='logs/{}'.format(NAME))
print(NAME)
model7=Sequential()
model7.add( Conv2D(64,(3,3), input_shape=X_train.shape[1:]) )
model7.add(Activation(act_layer))
model7.add(MaxPooling2D(pool_size=(2,2)))


model7.add( Conv2D(64,(3,3) ))
model7.add(Activation(act_layer))
model7.add(MaxPooling2D(pool_size=(2,2)))

model7.add( Conv2D(64,(3,3) ))
model7.add(Activation(act_layer))
model7.add(MaxPooling2D(pool_size=(2,2)))

model7.add( Conv2D(64,(3,3) ))
model7.add(Activation("relu"))
model7.add(MaxPooling2D(pool_size=(2,2)))

model7.add(Flatten())
model7.add(Dense(64))
model7.add(Activation(act_layer))
model7.add(Dropout(0.2))

model7.add(Dense(64))
model7.add(Activation(act_layer))


model7.add(Dense(5))
model7.add(Activation('softmax'))

model7.compile(loss="categorical_crossentropy",optimizer="adam",metrics=['accu
racy'])

model7.fit_generator(datagen.flow(X_train_ch,y_train, batch_size=32),
                                    epochs = 25, validation_data=(X_validate,y_v
alidate), callbacks=[tensorboard])
```

Section 12:We have finally picked a final model. Model 7 is the best that we have. Here we create a confusion matrix to try to understand the misclassification

In [ ]:
```
y_pred_test_model7=model7.predict_classes(X_test)
y_test_returned=np.argmax(y_test,axis=1)
```

In [ ]:
```
#this function creates a confusion matrix.
def plot_confusion_matrix(cm, classes,
                          normalize=True,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)
    plt.rcParams.update({'font.size':22})
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.rcParams.update({'font.size':32})
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=90)
    plt.yticks(tick_marks, classes)


    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [ ]:
```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test_returned, y_pred_test_model7);
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure(figsize=(20,10))
plot_confusion_matrix(cnf_matrix, classes=['daisy','dandelion','rose','sunflow
er','tulip'],
                      title='Confusion matrix, with normalization')
plt.show()
```

Section 13: Create the classifcation chart

In [ ]:
```
print(classification_report(y_test_returned, y_pred_test_model7,target_names=[
'daisy','dandelion','rose','sunflower','tulip']))
```

Section 14: This was my way of finding the misclassified images and then plotting them to try to better understand what was going on.

In [ ]:
```
print(y_pred_test_model7[10:20])
print(y_test_returned[10:20])
```

In [ ]:
```
#plots different images of the flowers from the test data to be able
#to try to understand what is going on with the data
#Tulip to rose 4 to 2
plt.rcParams.update({'font.size':8})
first=plt.subplot(1,4,1)
first.title.set_text('Predicted:Tulip True Label:Rose')
plt.imshow(X_test[19])
plt.axis('off')
plt.subplot(1,4,2)
plt.imshow(X_test[20])
plt.axis('off')

plt.subplot(1,4,3)
plt.imshow(X_test[205])
plt.axis('off')
plt.subplot(1,4,4)
plt.imshow(X_test[242])
plt.axis('off')
```

In [ ]:
```
#opposite Rose to tulip 2 to 4

second=plt.subplot(1,4,1)
second.title.set_text('Predicted:Rose vs. True Label:Tulip')
plt.imshow(X_test[27])
plt.axis('off')
plt.subplot(1,4,2)
plt.imshow(X_test[114])
plt.axis('off')
plt.subplot(1,4,3)
plt.imshow(X_test[221])
plt.axis('off')
plt.subplot(1,4,4)
plt.imshow(X_test[276])
plt.axis('off')
```

In [ ]:
```
#sunflower to rose 3 to 2
c=plt.subplot(2,4,5)
c.title.set_text('P: Sunflower T: Rose')
plt.imshow(X_test[120])
plt.axis('off')
plt.subplot(2,4,6)
plt.imshow(X_test[201])
plt.axis('off')
#sunflower to tulip 3 to 4
d=plt.subplot(2,4,7)
d.title.set_text('P: Sunflower T: Tulip')
plt.imshow(X_test[25])
plt.axis('off')
plt.subplot(2,4,8)
plt.imshow(X_test[15])
plt.axis('off')
```