Capstone 2 Project Milestone Report 2

Contents

Problem Statement

The purpose of this project is to determine how well a pretrained transfer learning convolutional neural network (TL) can detect stylized images despite having been trained on non stylized images. In particular, this project will attempt to determine how well a TL model pretrained on the Imagenet dataset, which has a large variety of images (over 14 million at the moment) ranging from plants to animals to structures to vehicles, can detect flowers in stylized images that will be generated through an arbitrary style transfer model (AST).

The usefulness of this project comes from the fact that image detection and image generating models are becoming increasingly popular and increasingly sophisticated. With regard to image detection, a great deal of work is being done in facial recognition and automated driving. While in image generation, Generative Adversarial Networks (GAN) are used to create art. This project attempts to bring together both image recognition and image generation.

The clients for this project include anyone who is interested in developing better image recognition models, whether they are individuals or companies. While the work done here is not cutting edge, it can offer potential insight into how image recognition models “see” art. It goes beyond the usual recognition of everyday images to those that are more abstract and obscure but still retain traces of actual objects. In this sense, this project also becomes useful for artists as it may allow them some understanding of how image recognition models process their work. It's true that art is still made for human consumption, but it is increasingly being made by artificial intelligence.

As we continue to develop more human-like AI, perhaps we may wish for them to understand some of our more subjective practices such as art. Humans can in large part identify objects in various types of artwork, except perhaps in more extreme abstract art creations. Of course, art is open to interpretation, but that is well beyond the scope of this project.

Yet, in art, beauty(or in this case, aesthetics) is very much in the eye of the beholder. So, maybe it won't matter how AI interprets art because even if its interpretation is "wrong", it may very well still add to the larger conversation of engaging with art in which a wide range of voices are welcomed. And someday those voices may not necessarily only need to be human.

Project Summary

Our project involves two deep learning models. The first is a transfer learning model, which will be developed by testing three well known CNN models - VGG16, ResNet50, and DenseNet169. The second is Magenta's arbitrary style transfer model. We'll be training our TL models on over 2000 flower images that are divided among five species. Using our AST model, we'll develop 2000 images by stylizing 100 flower images using five different art styles and four different interpolation weights.

Once we determine our most accurate TL model by testing it on a subset of the original flower dataset, we'll test how well it predicts the species of stylized images. From there, we'll determine how accurate the model is with regard to certain subsets of the data, including species, art style, and interpolation weight. This latter testing of subsets of the stylized images may help us better determine whether there are certain types of images that are easier or harder for our model to predict. For instance, is it clearly easier to accurately identify flower images stylized with fractal art than that with abstract expressionism? Or are images created with 0.4 interpolation weight easier to identify than those made with 0.6 interpolation weight?

Exploratory Analysis

Flower Dataset

Our dataset has been acquired from Kaggle and was previously used in an image recognition contest. It can be found here: https://www.kaggle.com/alxmamaev/flowers-recognition

The dataset is relatively small, at 4242 images that are divided into 5 species - daisy, dandelion, rose, sunflower, and tulip. The size of each group ranges from 1055 to 734, with dandelions having the highest total and sunflowers the lowest. All of the images are in color. Below (Figure 1) is a sample image from each species:

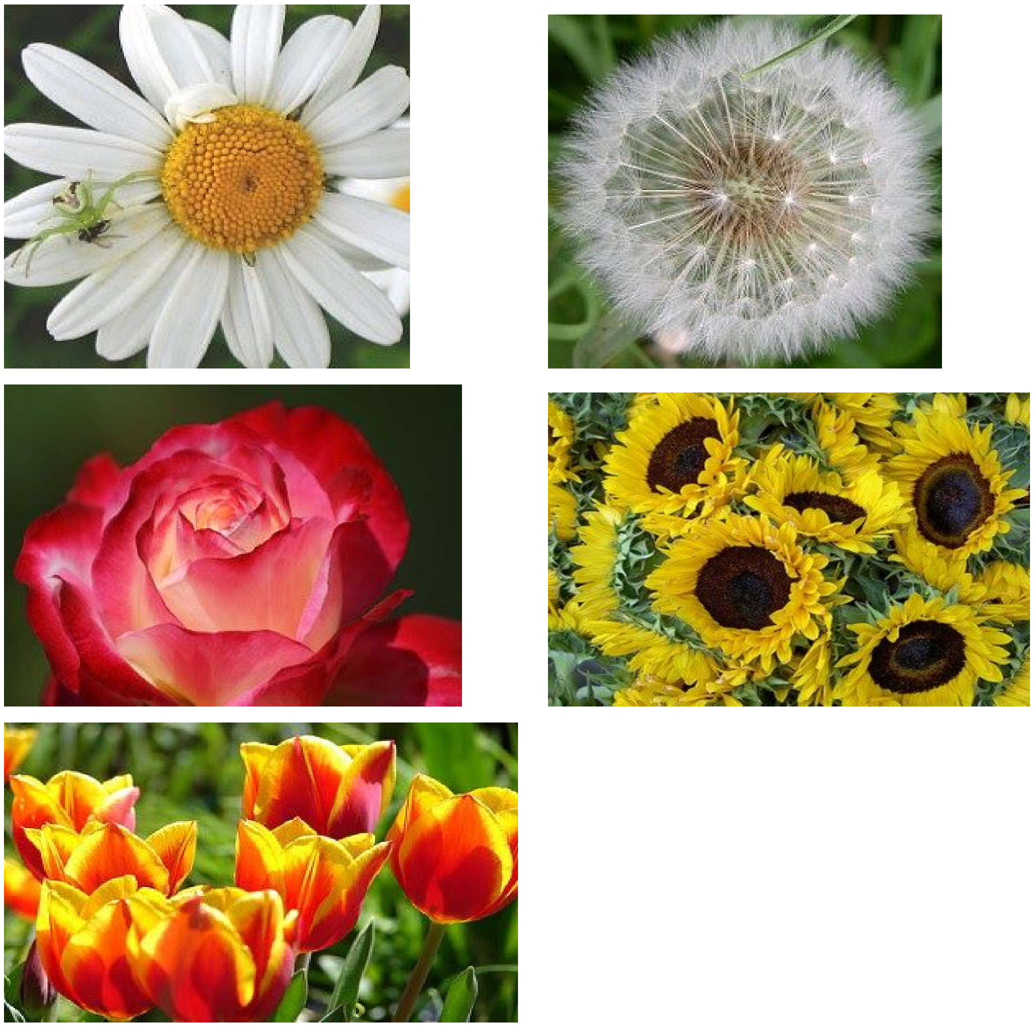


Figure 1: From top to bottom and left to right - daisy, dandelion, rose, sunflower, tulip.

While our dataset doesn't allow for an in depth statistical analysis, because they're images and not datatables, they do allow for a descriptive analysis that may allow us to form some sense of what to expect once we run the images through both of our models.

A sampling of the images reveals that they are not all presented in the same manner. For example, some images include people; some images are close ups of a single flower, while others show a field full of a single species. Comparing species, dandelions have a large amount of variety. They can have a white feathery look, but they can also have long, thin yellow petals, which can at times resemble those of sunflowers. Roses and tulips share some colors, including pink and yellow, and in some photos of closed roses, they do somewhat resemble tulips. This variety has its pros and cons. On the upside, it does allow the model to get familiar with a particular species through several different aspects, including shape and color variation and near and far perspectives. However, this variation also results in an intersection of qualities in which the model may get confused between a dandelion and a sunflower or a rose and a tulip. As such, it's hard to predict how both of the positive and negative aspects of the original dataset will affect the overall accuracy of the stylized dataset.

We'll be dividing our dataset into train, validation, and test groups. All five species will be present in each group, and there will be a 60-20-20 split for each species among train, validation, and test groups. We'll be using the training and validation datasets to train the model and the test model to determine the test accuracy of the final model.

We'll be assuming that the recognition model will not be able to do as well on the stylized images for several reasons - stylization may cause image distortion such as blurring of object borders or border destruction due the creation of new shapes and lines; it may also cause color changes - a rose that was largely pink may end up being composed of several different colors after stylization, making it more difficult for the recognition model.

Art Styles

We'll be using five different art styles that allow for a range in aesthetics - abstract expressionism, cyberpunk, fractal, pop art, and post-impressionism. All pieces are taken from Creative Commons as to avoid any conflict arising from its use, and in many cases, the artist and name of the piece are not known. However, that did place a limitation on the selection and quality of the style images.

A brief summary of each image and general style:

Abstract expressionism is often characterized by its perceived spontaneity and arbitrary appearance. Paint drops and splatters appear to be haphazardly strewn across a canvas, despite most works actually having involved careful planning. The piece we'll be using contains colors from across the color spectrum with lines going in all directions. It's also hard to discern what exactly the piece is portraying, made all the more difficult without a title.

Cyberpunk imagery displays futuristic settings and characters, typically dense cityscapes and cybernetically enhanced beings. Many pieces also rely on neon colors, particularly pink, blue, and purple against dark backdrops. Such is the case with our piece, which displays tall buildings outlined in bright purple, pink, and blue and a black background.

Fractal art uses repeating patterns, typically in the form of spirals and swirls or iterations of the Julia and Mandelbrot sets. Such designs can often be very elaborate and colorful, but the image we'll be using is fairly devoid of color. That's in contrast to some of our other images. That lack of color, as well as its intricate arched pattern, adds variety to our style set..

Pop art often appears in advertising and is known to use pop culture icons as its subject matter. Warhol’s Campbell Soup image is one such example. Our image is a close up of DC Comics icon Batgirl (likely), stylized as if lifted from a comic book.

Post-Impressionism stands out for its reaction against Impressionism, namely by rejecting the latter’s naturalistic depiction of color and light. In contrast, the former often uses unnatural, arbitrary colors, while emphasising distorted geometric forms. Our piece is Van Gogh’s Women Picking Olives, which depicts a field of green, distorted trees rooted in brown dirt, against a faded sky.

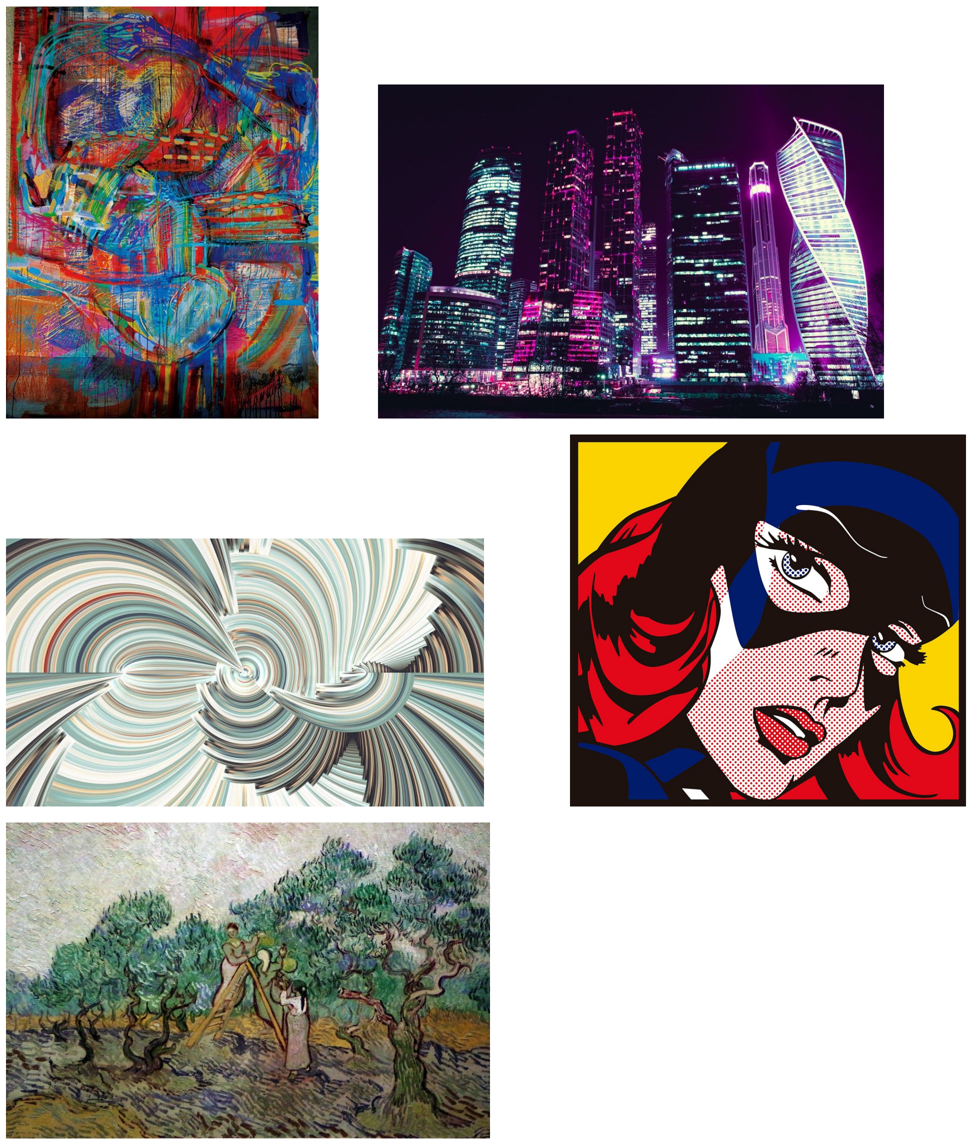


Figure 2: From top to bottom, left to right our styles for: abstract expressionism, cyberpunk, fractal art, pop art, and post-impressionism.

Magenta Neural Style Transfer Model

The neural style transfer model that we're using has been developed as a part of the Magenta project, which was begun by members of the Google Brain team and is an effort to use deep learning and reinforcement learning to generate images, drawings, and even music. We'll be using a form of style transfer referred to as arbitrary style transfer. It is a method that allows for fast style transfer for any arbitrary style.

Another advantage of this approach is that it allows us to determine how heavily we want the content image and style image to factor into the resulting stylized image. Using a scale ranging from 0.0 to 1.0, where 0.0 includes no influence from the style image and 1.0 accepts no influence from the content image, we can vary the influence of either image to varying degrees. This variation in influence is referred to as the interpolation weight.

The model works by mapping each content and style image to a 100 dimensional content and style vector, respectively. The weighted average of each vector is then used as the input for the stylized image. This is also how it's able to control the weight applied by each image for a given stylized image.

Running the model invokes only a small amount of code, but requires a decent amount of setup. If we're running it on Google Colab, as we did, we first need to download the pretrained model to Google Drive. Additionally, we also need to include content image and style image folders as well as an output folder. From there a short series of commands identifies the paths to each of the aforementioned files and folders as well as the interpolation weights to be considered and the content and style images sizes. When images are kept rather small, and a GPU is used to generate the images, more than 100 stylized images can be created in a matter of seconds.

It's worth noting that the output folder will include not only the stylized images but the original content and style images used as well. Thus, if you plan to run the images through an image recognition folder, it would be appropriate to delete the original images to achieve better accuracy.

Stylized Images

For our stylized images, we took 20 random images from each species and crossed them with our five styles, all for four interpolation weights - 0.2, 0.4, 0.6, 0.8 - to give us a total of 2000 stylized images. Below (Figure 3) is an example of a rose image styled with our fractal art style for four interpolation weights. Here, we have a gradual transformation from 0.2 to 0.8:

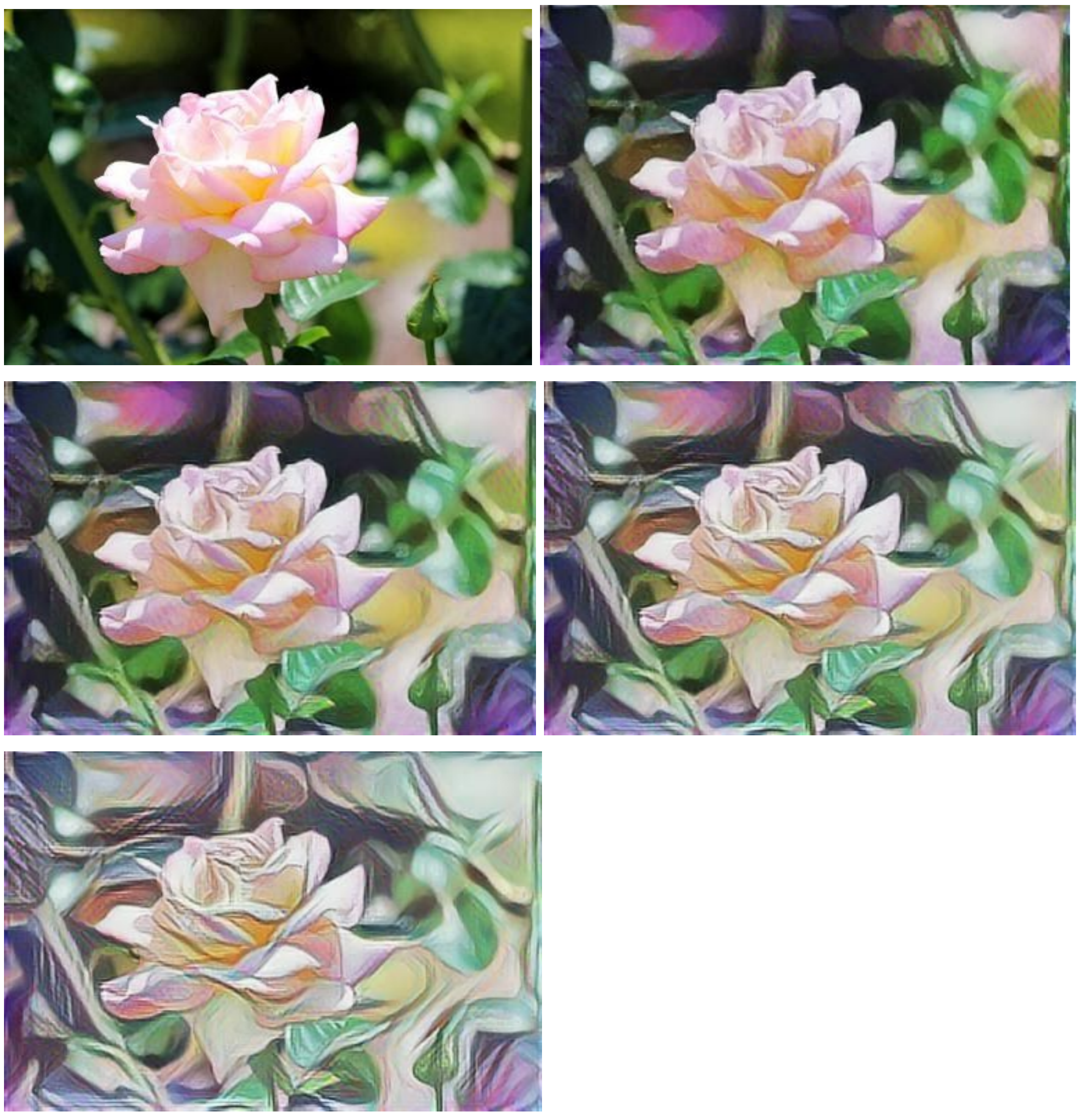


Figure 3: From left to right and top to bottom: The original rose image, the fractal art stylized images with interpolation weights 0.2, 0.4, 0.6, and 0.8.

In some cases, the 0.8 interpolation weights changes the original image to such a large extent that it's difficult even for humans to recognize whether a flower is present, much less identify the species. It may be an even more difficult task for a TL model. The following is a good example in which the original image of a field of sunflowers is completely unrecognizable in the pop art 0.8 stylized image on the right. Without having the original as a reference, we wouldn't be able to tell that there are flowers in the stylized image.

On the left, the original image of sunflowers. On the right, the same image pop art stylized image at 0.8 weight.

Transfer Learning CNN Model

The pretrained TL model that we'll be using is based on the model created by Github user hey-simone. The model can be found here:
https://github.com/hey-simone/flowers-classifier/blob/master/Keras_Flowers_Classifier-V1-2.ipynb
The VGG16 and DenseNet169 models were originally chosen by the author but without any real justification for their inclusion. Thus, any choices made with the original model will only be justified if the original author provided any such justification. Otherwise, we'll simply describe the process. Additionally, we'll be adding ResNet50 due to its reputation with multiclassification image problems.

Since there are five different species to identify, this will be a multiclass classification problem. In a binary classification problem, we would simply be trying to determine whether the model can detect if a flower is present. However, in our case, the model is working under the assumption that at least one flower is present in every image, and that the task is to correctly identify the correct species. In this case, there are two main factors working against high accuracy. First, a multiclass problem usually has a lower accuracy than a binary problem. Second, as we saw above, in some images, it's very difficult even for humans to determine whether a flower is present, mch less the correct species. The TL model is going to struggle with those images as well. As such, we are expecting the stylized test images accuracy to be considerably lower than the accuracy of the original test images.

Before we start, we need to set a random seed to ensure that our results are repeatable. Neural network algorithms are stochastic, meaning they rely on randomness to operate (such as initializing random weights). However, that randomness can lead to inconsistent results. By setting a random seed, we'll be using the same set of random values, allowing our model results to be repeated.

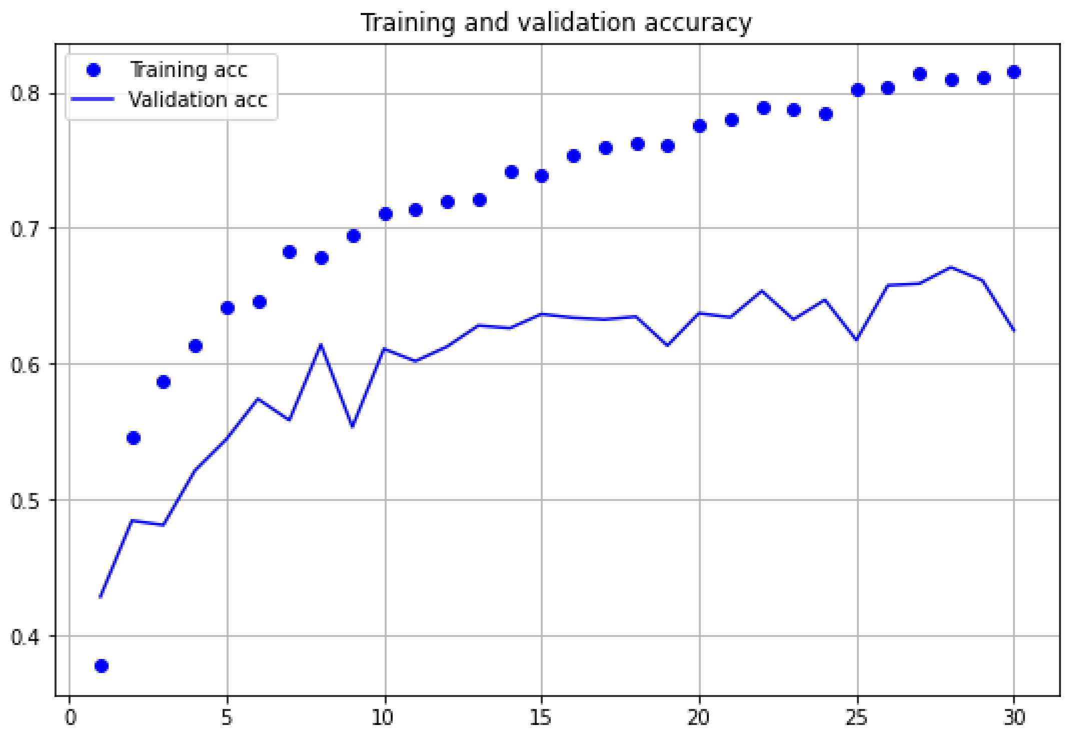
Since both Keras and Tensorflow backend have their own random seeds, we need to set both of them.

We begin by loading the image dataset, which has already been divided into training, validation, and testing groups, as detailed above. The images are then transformed into array data using ImageDataGenerator. We also normalized the images by figuring all of their pixel values by 255. Most images' pixels range between 0 and 255. The normalization process allows us to range the values between 0 and 1 instead.

We tried several different models, all pretrained on the Imagenet dataset (except for our baseline model) - VGG16, ResNet50, and DenseNet169. With all of the pretrained models, we froze most of the layers to preserve the pretrained weights. For the baseline model, we start with a series of convolutional layers and max pooling layers. The convolutional layers slide across images to create activation maps, while max pooling reduces volume and computation costs. For the convolutional layers we'll be using Rectified Linear Units, or ReLU's. Since ReLU's are linear for all positive values and zero for negative values, they are computationally cheap. And since their slope doesn't plateau, they don't suffer from vanishing gradients, which can decrease the chances of a signal propagating to the input layers, resulting in weights not adjusting and earlier layers failing to learn. Our final layer has a softmax activation with five units that correspond to the five classes or species of our dataset. We'll also be using the Adam optimizer. It can adjust the learning rate separately for each layer, including reducing the learning rate as the model gets closer to convergence. We'll also be using fit_generator because we are inputting our data through ImageDataGenerator. For most of our models, including baseline, we'll be running for 30 epochs. All of the above layers were chosen by the original author.

An initial run yields validation accuracy in the high 50's and almost perfect training accuracy, suggesting overfitting. We make some adjustments by invoking data augmentation on the training data inside ImageDataGenerator. This will allow the model to view the data from different randomized perspectives, hence increasing its generalizability. We'll also add dropout in our layers. With dropout, some layers are randomly ignored during training, giving the model a different view of the configured layer. As a result, the model will be less prone to overfitting.

The second run with our adjusted model yields validation accuracy in the mid 60's and training accuracy in the high 70's. So, it's an improvement in accuracy and overfitting. The following graphs (Figure 4) display the disparity between loss and accuracy of the training and validation data. Running the model on our test data leads to 69.1% accuracy.



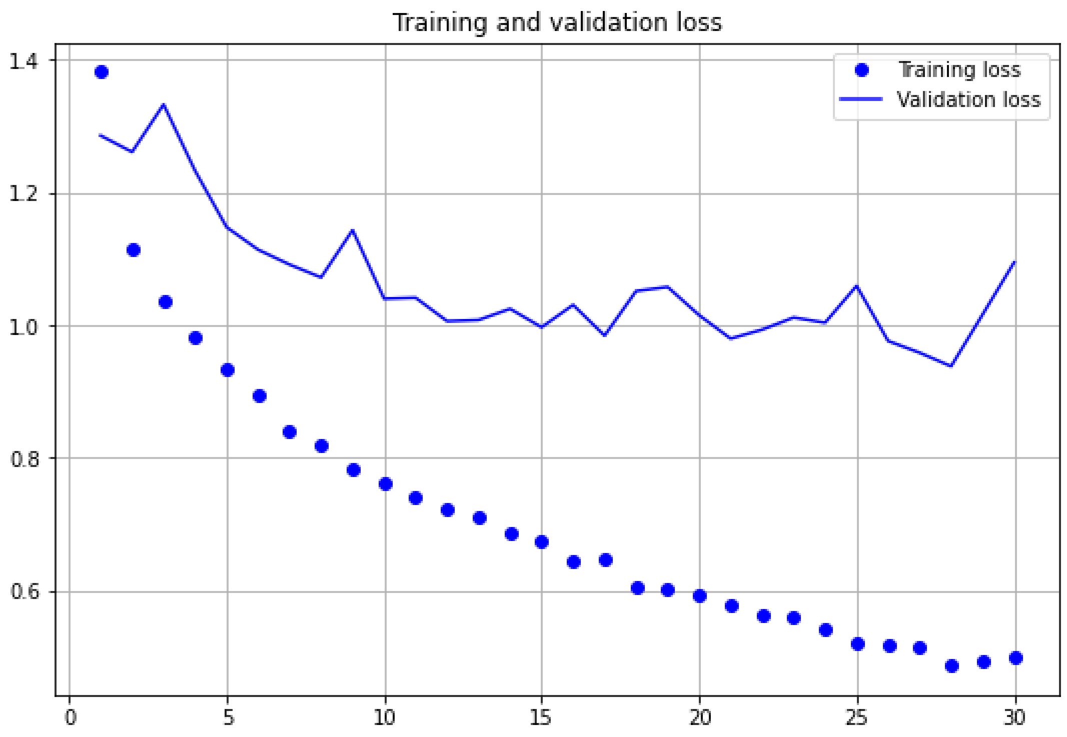
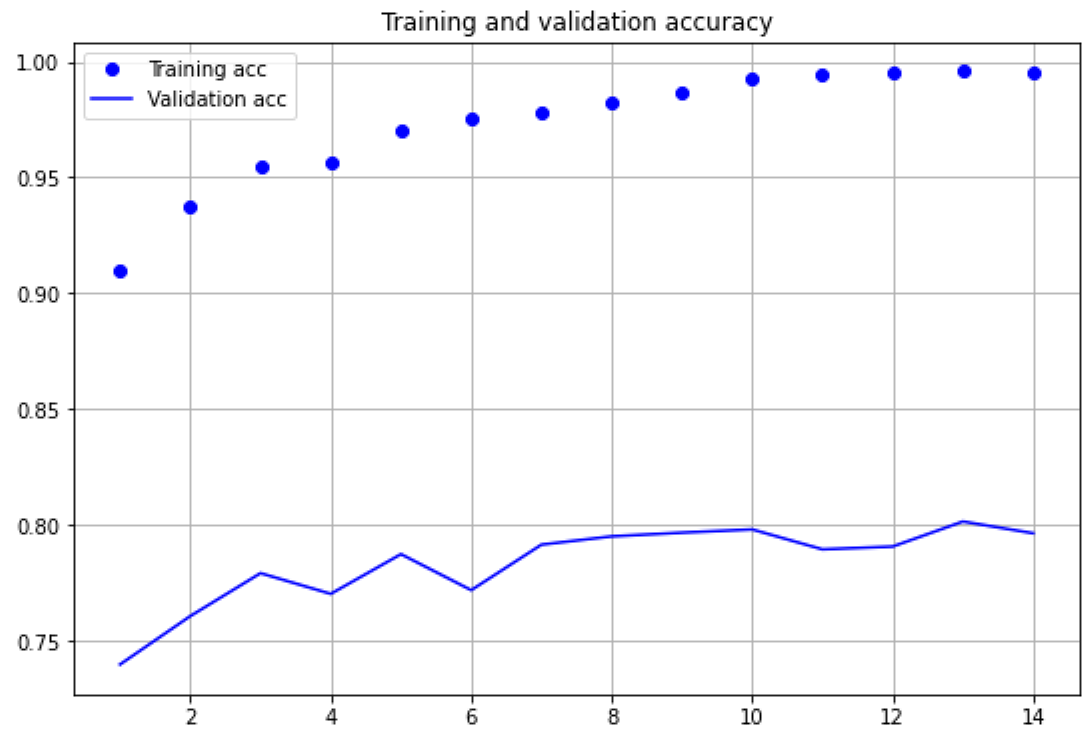


Figure 4: The training and validation accuracy and loss for our baseline model.

While we saw a modest improvement to our baseline model, it's probably wiser to use a pretrained model. For all, except the base model, we'll invoke several callbacks: ReduceLROnPlateau, which reduces the learning rate if the learning begins to stagnate; EarlyStopping, which stops the process if there isn't any improvement in the validation loss after a set number of epochs; and ModelCheckpoint, which saves the model and/or weights after the the most improved epochs.

For each model, we'll begin by only unfreezing the last layer. We'll then make adjustments in hopes of possibly improving the accuracy. We'll start with VGG16. VGG16 has 12 convolutional layers interspersed with max pooling and 4 fully connected layers for a total of 16 layers. It also has a 1000-way softmax classifier. In our initial run, without making any adjustments and using an Adam optimizer, we only achieve validation accuracy in the mid 60's. We then decide to freeze several of the layers in the convolutional base, only allowing four trainable weights. That allows us to increase our accuracy to the mid 70's, but the model is now overfitting.

We then unfreeze several of the layers from the previous run, which increases our accuracy to the high 70's , while still overfitting, and results in a test accuracy of 84%, a solid improvement over our base model. The improvement is also evident when comparing the VGG16 training and validation graphs (Figure 5) to the baseline model:



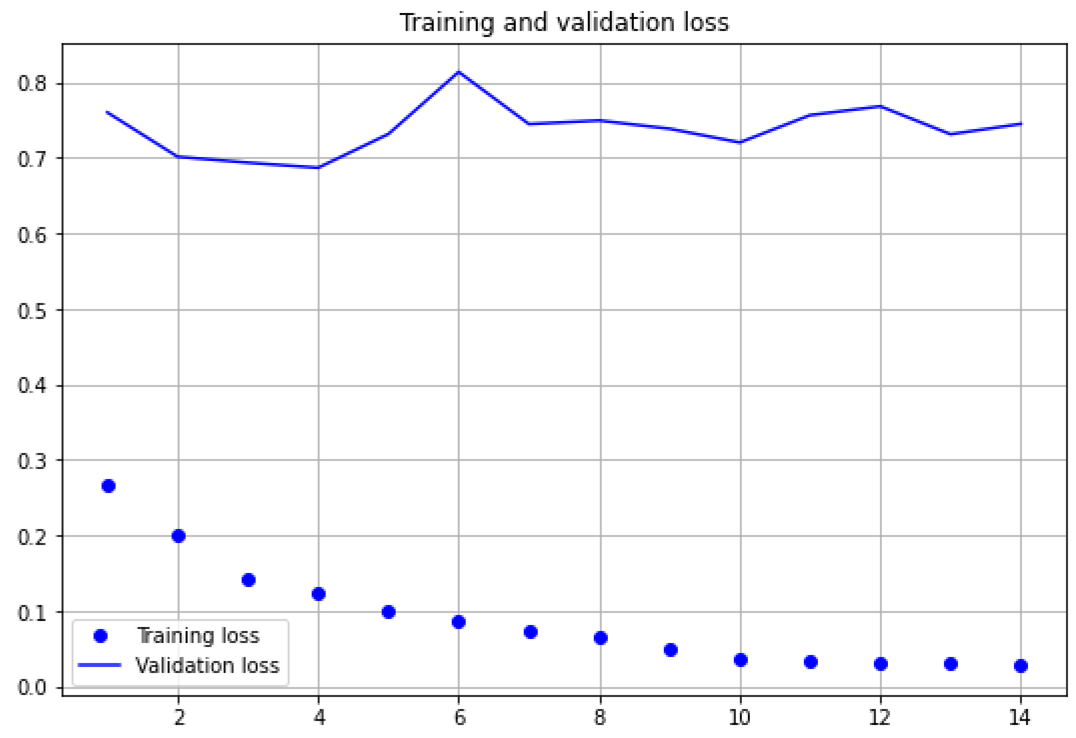
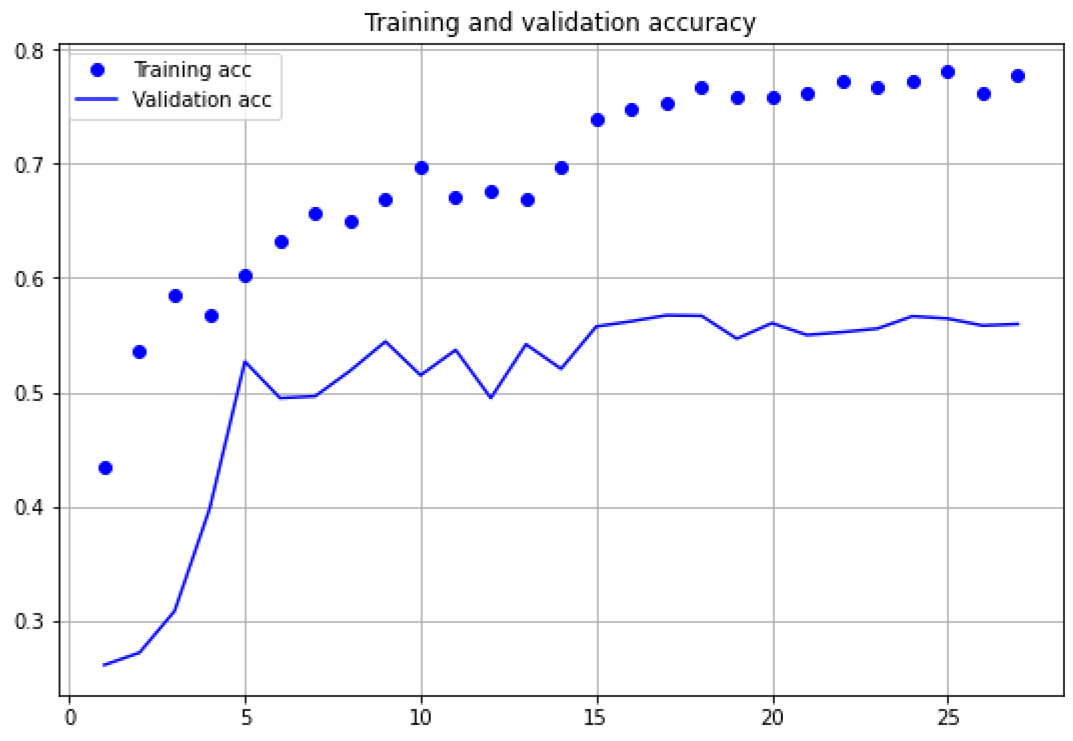


Figure 5: The training and validation accuracy and loss for our VGG16 model.

Next, we evaluate a ResNet50 model. ResNets are often regarded as improvements over VGG models. A unique feature that they have is skip connections, which allow original input to be added to the output of the convolutional block. This can help with the vanishing gradient problem that many CNN models face. We'll be using a similar process to that of the VGG model.

For our ResNet50 model, we begin by freezing all but four layers. Using an Adam optimizer, we find that the validation accuracy is in the low 50's and overfitting is quite high. We then decide to unfreeze many of the layers that we initially froze, allowing for 44 trainable weights. However, there fails to be any improvement in accuracy or overfitting. In fact, our test accuracy is only 63%, which is both disappointing and surprising given that ResNets are often touted for their strong results. It's not immediately clear why this model performed so poorly. It may be that we didn't train the correct weights. Perhaps at a later time, when we aren't working against a deadline, we may revisit this model and try other modifications.



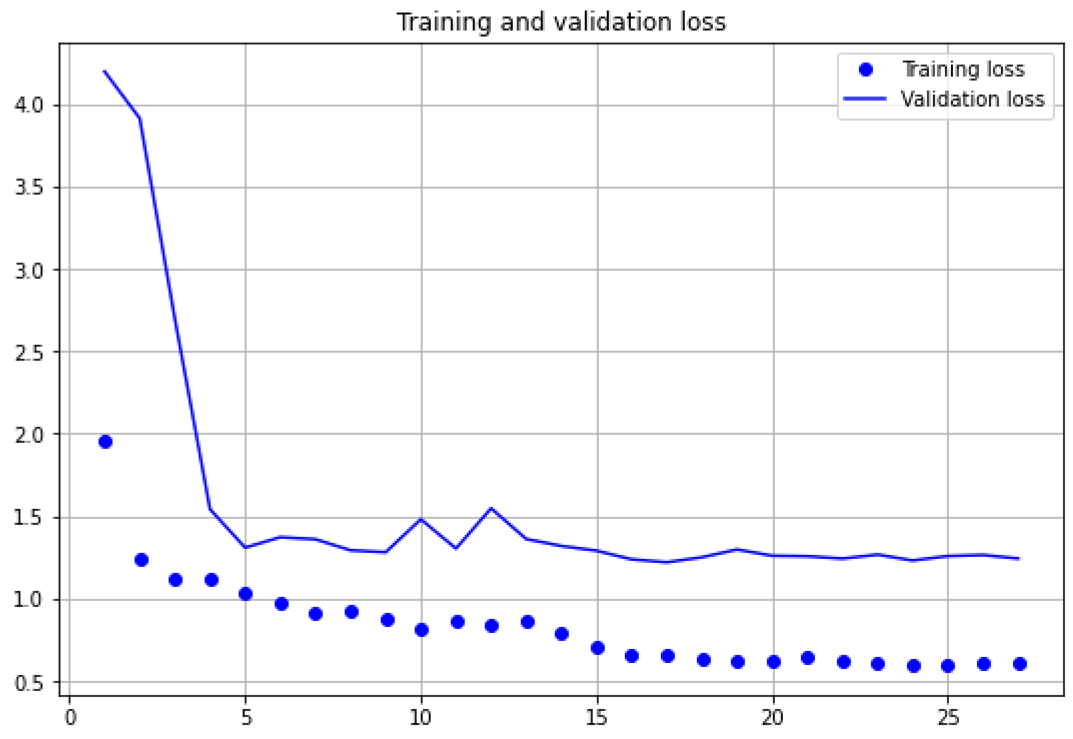
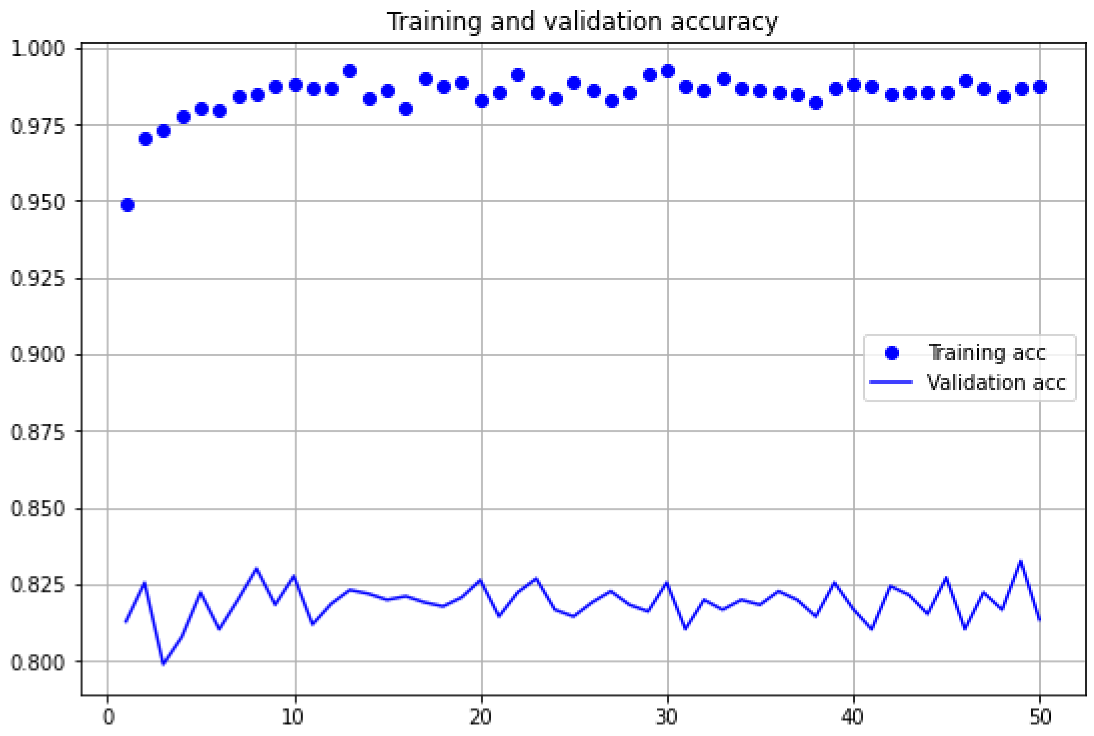


Figure 6: The training and validation accuracy and loss for our ResNet50 model.

Finally, we try a DenseNet169 model. DenseNets typically require fewer parameters than traditional CNNs. Additionally, each layer in a DenseNet has access to the original image and the gradients from the loss function. They also concatenate output and input feature maps rather than summing them.

As with our previous two models, we'll start by freezing some of the layers of the model. After freezing, we have four trainable layers. Our validation accuracy is in the low 80s and overfitting is rather high. Next, we unfreeze some of the previous frozen layers and run the model again. The accuracy and overfitting don't seem to be affected much. Running the test set on the DenseNet model yields 90% accuracy, and that makes it our best performing model.



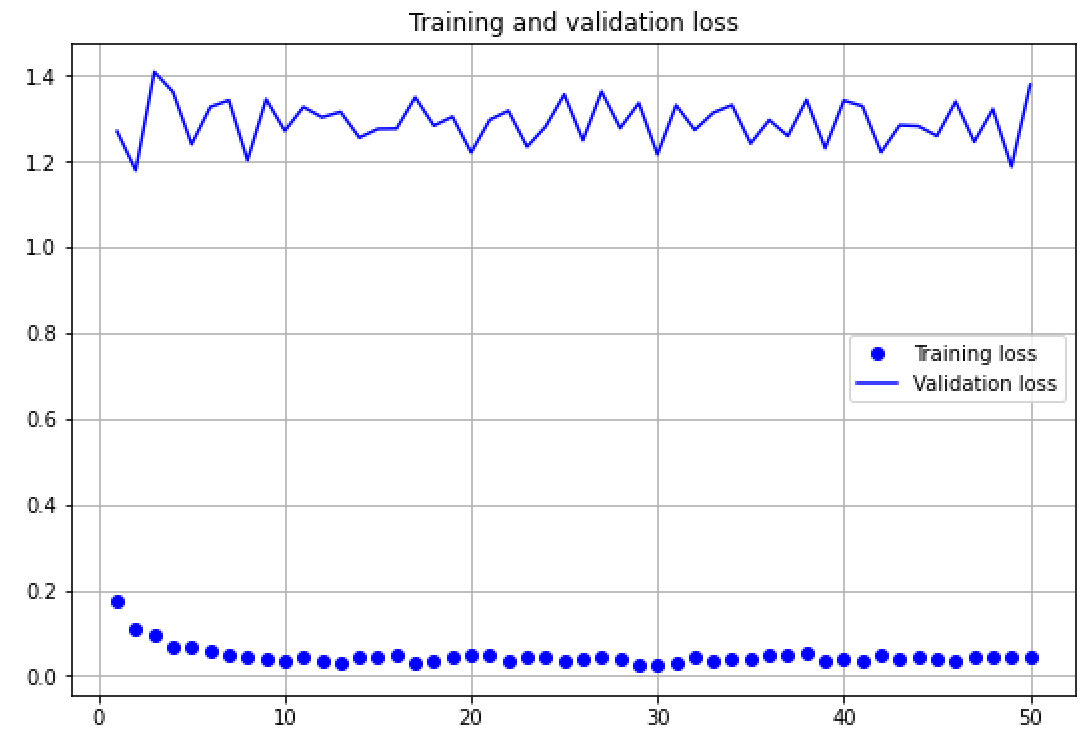


Figure 7: The training and validation accuracy and loss for our DenseNet169 model.

The following is a table (Table 1) of the test results from all four of our models:

Test Loss and Accuracy for Four CNN Models

	Test Loss	Test Accuracy
Baseline	0.87	69.1%
VGG16	0.50	84.1%
ResNet50	1.10	69%
DenseNet169	0.59	90%

Table 1: A summary of the results on our testing dataset.

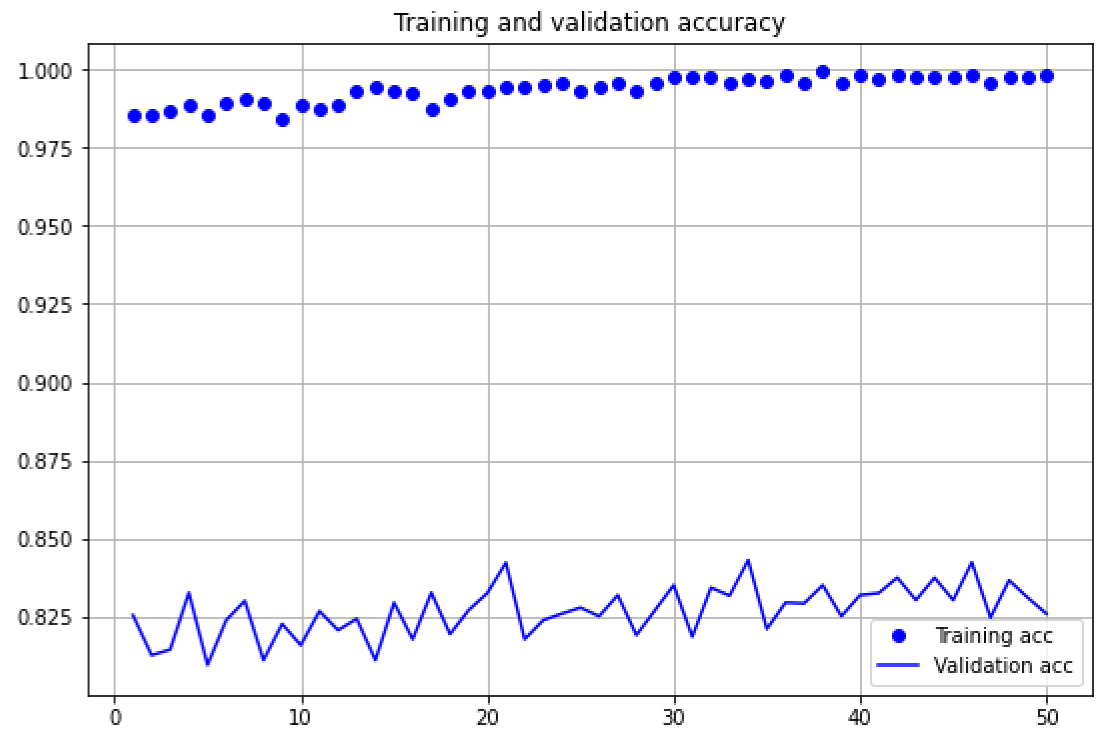
Since DenseNet169 is our best performing model, we'll use it to test our stylized image set. As expected, due to the reasons mentioned earlier, the model does quite poorly with the dataset, achieving a test loss value of 0.313 and a test accuracy value of 61%.

Cyclical Learning Rate

As we saw in the previous section, with some adjustments to some of the DenseNet model layers, we were able to achieve a satisfactory 90% accuracy for our test images but had a terrible 61 % accuracy of our stylized images. In an effort to increase both of these accuracies, we're going to implement a cyclical learning rate (CLR), which is part of Tensorflow's optimizer library. The learning rate hyperparameter is a vital part of the gradient descent process and controls how much our model is changed in response to the estimated errors. A learning rate that's too small may cause the model to get stuck, while a learning rate that's too large may result in a model skipping past the optimal minimum.

We could have used one of several approaches for increasing our accuracy, but what makes CLR appealing is that rather than sticking with a single learning rate for an entire training/test run, CLR allows for a change of the learning rate during the run. This 'resetting' of the learning rate may allow the model to find other local minima if the loss rate appears to have stalled for a number of epochs. As a result, the model may find a more optimal minimum than it would have settled on otherwise. This would potentially allow the model to generalize better and, in return, increase its accuracy.

We'll implement CLR through the Adam optimizer. Rather than having a constant learning rate, we'll set a range starting at 1e^-11 and having a maximum rate of 1e^-5. As with the previous DenseNet model, we'll run this for 50 epochs.



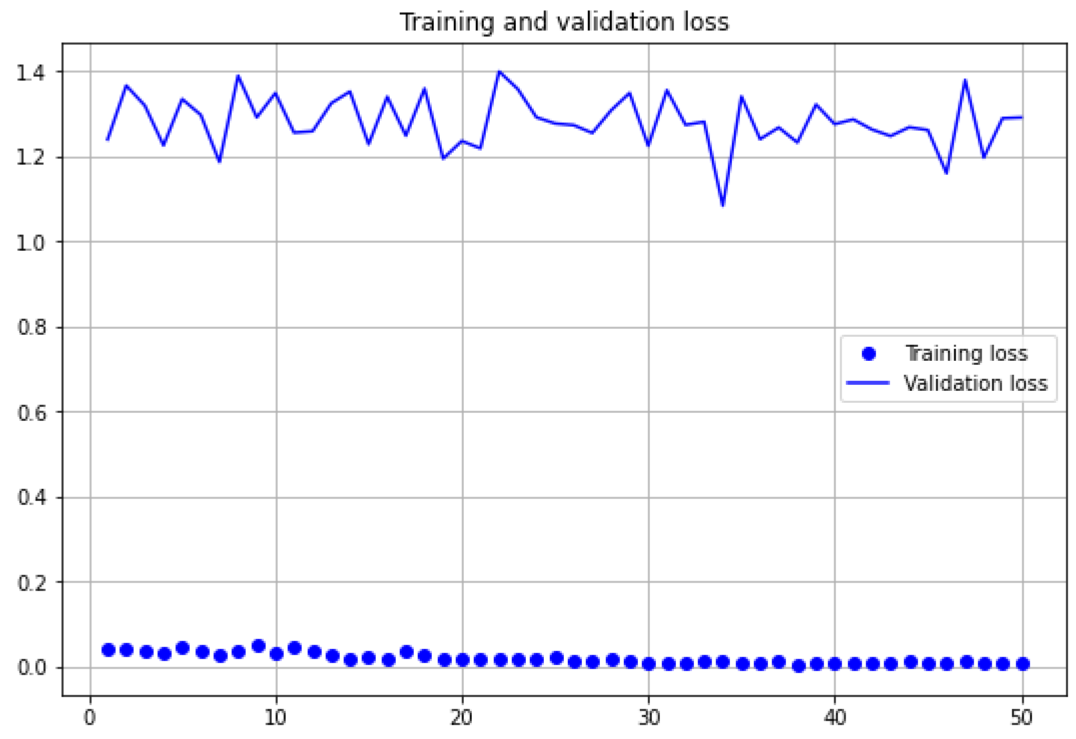


Figure 8: The training and validation accuracy and loss for our CLR DenseNet169 model.

We see only a slight increase in accuracy for the test images - 91.6%. The training loss has been reduced to 0.551. The stylized images only showed a small increase as well - 62% with a reduction in the training loss to 3.12. It appears that our original model was having issues with an optimal learning rate, particularly with the stylized images. While we had used ReduceLROnPlateau for our original model, and reduced by a factor of 0.01 if there wasn't a loss reduction within 5 epochs, CLR doesn't appear to have done a better job at finding a better minimum. It may have been that we set our patience too low, causing our model to converge too quickly.

Test Time Augmentation

As a final method to increase our accuracy, we'll be applying test time augmentation (TTA) to our CLR model. We had already augmented our data by using ImageDataGenerator, but we had made more significant augmentations to the training data than any of the other datasets. As we saw with our baseline model, augmenting the training data helped increase our accuracy. With TTA, we are able to apply similar augmentations during our test runs. Hopefully, it will allow for similar gains in accuracy.

For TTA, we'll be applying a modified version of the code written by Jason Brownlee. More information on the specifics of the code can be found here:
https://machinelearningmastery.com/how-to-use-test-time-augmentation-to-improve-model-performance-for-image-classification/

TTA allows for creating multiple augmented copies of every image in the test set, creating, in a sense, more data for the model to analyze and, potentially, reducing generalization error. The model will make a prediction on each image and return the ensemble. For this particular implementation, we'll be making seven additional copies of each image. After some trial and error, we find a large increase in accuracy with the following augments:
horizontal flip = True, width_shift_range=0.1, height_shift_range=0.1, shear_range=0.15, zoom_range=0.1. We run the model 10 times on each both the test and stylized data and average the accuracies to get a final accuracy for each dataset.

We see a significant increase in accuracy with a simple flip augmentation. For our test set, the accuracy increases to 97%, while the stylized set accuracy jumps to 75.3%. Just by creating augmented data and allowing the model to see the images from different perspectives, we were able to achieve an accuracy gain. In the table below (Table 2), we're able to see a nice evolution in the accuracies of our datasets as we further tuned our model.

Accuracies for all DenseNet Models for Test and Stylized Image Sets

Model Version	Test Images Accuracy	Stylized Images Accuracy
Original DenseNet169	90%	61%
DenseNet169 + CLR	91.6%	62%
DenseNet169 + CLR + TTA	97%	75.3%

Table 2: A summary of all DenseNet model accuracies for both the test and stylized datasets.

Neither the test set nor the stylized set saw much of a gain in accuracy from CLR. However, the accuracy gains for both datasets are remarkable through TTA, particularly for the stylized set which increased by 14% percentage points . We now have a model that can predict flower species with near 97% accuracy. But, more importantly for the purposes of this project, our model can predict the species of the stylized images with over 75% accuracy. While this still seems too low to be considered a significant achievement, consider that it started at around 61% accuracy. Plus, as we mentioned earlier, many of those stylized images are difficult to decifier, even for humans. In a good portion of those images not only is it difficult to tell the species, it's hard to even make out whether flowers are present at all. In fact, most humans might not be able to achieve 75% accuracy on the stylized dataset. As such, our achievement is significant because part of the aim of every image recognition model is that it should be able to perform better than humans, and we may have come close to achieving that here.

For the final part of the project, we'll be looking at the accuracies among species, styles, and interpolation weights to get a better sense of where the model succeeded and where it faltered. Doing so might allow us to determine possible trends or patterns and give us further insight into how our model works.

References

Transfer Learning Model Analysis:

https://github.com/kjd999/Springboard-files/blob/master/Capstone%20Project%202/Transfer_Learning_Initial_Models.ipynb

https://github.com/kjd999/Springboard-files/blob/master/Capstone%20Project%202/Transfer_Learning_Tuning_Models.ipynb

Magenta Arbitrary Style Transfer:

https://github.com/kjd999/Springboard-files/blob/master/Capstone%20Project%202/Magenta_Arbitrary_Style_Transfer.ipynb