Using Deep Learning Techniques to Classify Photographs

# Problem and Client

The problem I have chosen for this capstone project, is to use deep learning techniques to assign photographs to one of four categories, i.e. landscape, animals, people, plants. One potential application for this type of model is applicable for a stock photography company. By allowing a trained model to classify photographs, the company could either supplement human-classified photos or, if photos have no classification attached to them, provide those classifications. This could reduce the workload for employees as well as ensure that photographs received from disparate sources are all classified similarly. As another application, this type of model could be used by individual photographers. With the rise of digital photography, photographers can take and store large numbers of photos. The ability to tag, or classify these photos becomes more and more important so that when a photographer needs to find a photo they can. A model like this could help individual photographers automatically tag large numbers of photos. A task that could take hours, days, or weeks if done by hand could be accomplished automatically.

# Data Wrangling and Cleaning

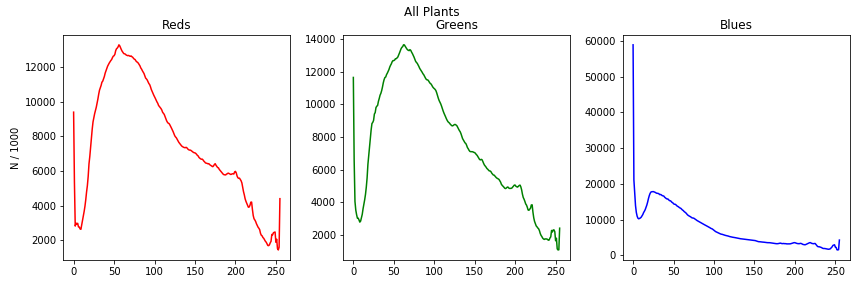
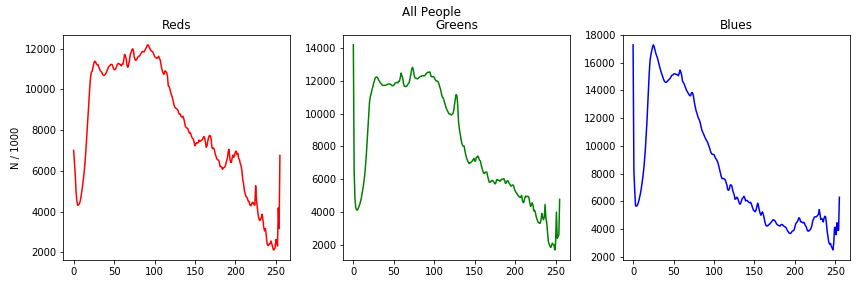
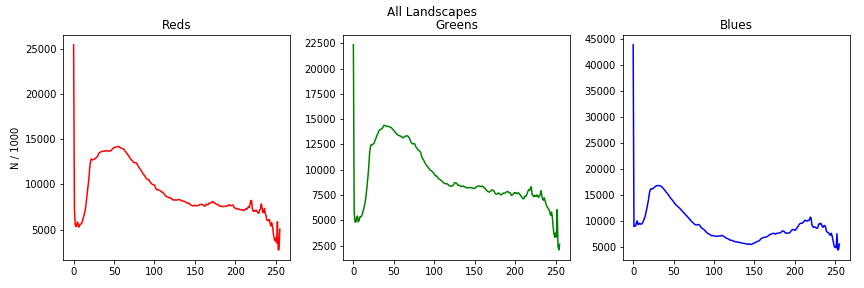
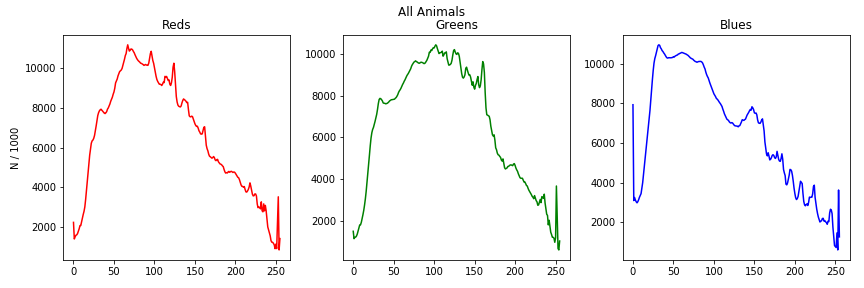
The photographs that I used to train my model came from my personal collection. I selected 200 photos in each of the four categories for a total of 800 photographs. These photos span at least 10 years of photography and consist of images taken on multiple cameras including digital and film cameras. The images taken on film have been previously printed and then scanned into digital format. I tried not to select too many photographs from the same shooting sessions. I have chosen both black and white as well as color photographs in each of the four categories. I did not do any resizing or cropping of the images during the selection process. All photographs are JPGs and are in RGB color format.

I randomly split my photos into training and validation sets. I originally copied all photos to their respective category folders then randomly selected 20% from each category to move to a validation folder. This left me with 640 training photos (160 photos x 4 categories) and 160 validation photos (40 x 4). The code I used to randomly select these photos can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/Randomly_Sort_Images.py).

# Initial Data Exploration

Since photographs don’t lend themselves to traditional summary statistics, I decided to take a slightly different tack. I first wanted to look at the frequency of color values for each of the color channels. I thought that this might be a way to determine how similar the color patterns are for each of the different photo categories. Each pixel in and RGB image is represented by three values (channels), one each representing red, green, and blue. Each channel is represented by an integer value between 0 and 256. For example, a pixel with values (0, 0, 0) would display black, (256, 0, 0) red, (0, 256, 0) green, and (0, 0, 256) blue. I looped over each photograph in a category (i.e. landscape, animals, people, plants) and for each photo I then looped through each color channel and pixel. I summed the number of times a color value appeared and stored those sums in a dictionary before plotting them. The results are shown in Figure 1.

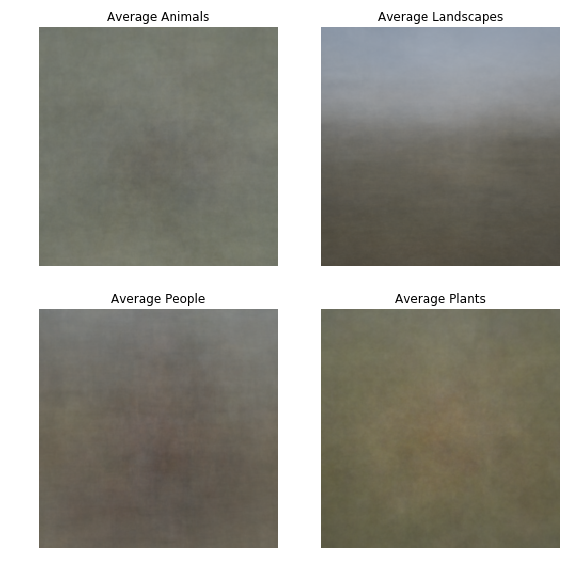
Figure 1. Category Color Channel Values



It looks to me like each of the four categories are somewhat unique. Animals and people are probably the most similar while the landscapes and plants seem to have somewhat unique color profiles. The code I used to generate these plots, as well as plots for the training and validation sets individually can be found in the Jupyter Notebook [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/Jupyter_Notebooks/Color_Channel_Plots.ipynb).

Another way to visualize the differences, or similarities, in the categories would be to plot an *average* image. This can be done by taking the average RGB color value for each pixel across every image in a category. The images in my dataset are not of a standardized aspect ratio, so first I needed to get all images into the same format. To do this, I first resized each image so that the smaller edge was 256 pixels. I then took a 224 x 224 center crop of the image. I then looped through each image and added 1/200th of the pixel color values of each image together. The resulting images are displayed in Figure 2.

Figure 2. Average Images



While you can’t see any distinct features, except maybe the sky in the landscape category, the colors of each category are slightly different. It seems to me that the animals tend to have a darker spot in the center of the frame, surrounded by more earth tones, i.e. greens and browns. The people photographs also tend to have a darker mass in the center of the image, but that mass is warmer, consisting of more reds potentially from varying skin tones. Finally, the plants tend to be more green and yellow overall which upon seems appropriate. The code that I used to generate these images can be found in the Jupyter Notebook [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/Jupyter_Notebooks/Average_Images.ipynb).

# Models Tested

I wanted to explore different model types to determine which were the best at accurately classifying images. Before I jumped immediately to training a deep learning model, I wanted to see if a traditional logistic regression model could accurately provide the classification that I wanted. After exploring logistic regression, I explored a fully connected feed-forward neural network (FNN) and a simple convolutional neural network (CNN). Finally, I used transfer learning to re-train various versions of residual networks (ResNet) designed by He et al. (2015)1 for the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2015 and CIFAR-10 image recognition challenges. These pretrained ResNet models are available in Python from Torchvision’s models module. I built all models using PyTorch.

## Image Transformations

All images went through transformations prior to being fed into the models for both training and validation. For the logistic regression, FNN, and CNN models, each image was resized so that the smallest edge was 256 pixels. I then took a square, center crop of that image with the dimensions 224 x 224. For the ResNet models I used Torchvision’s RandomResizedCrop to randomly select a portion of the image and then crop to 224 x224. This is the method used in PyTorch’s Transfer Learning Tutorial2. I tried this method for the other models but did not get results as accurate as when I used the center crop method described above. Each image was also normalized using mean = [0.485, 0.456, 0.406] and standard deviation = [0.229, 0.224, 0.225]. This normalization is expected by all pre-trained models in PyTorch so I determined that it was most appropriate to apply the normalization for all models. Finally, for the training set I applied a random horizontal flip to the images. This was not done for validation as I wanted the models to validate on the true version of the image.

## Logistic Regression Model

I used a standard logistic regression model with input dimensions of 150,528 (3 color channels x 224 x 224 image dimensions) and output dimension of 4 for the four image categories. I tested various batch sizes, learning rates, and epochs for training. The final model used a batch size of 20 trained for 5 epochs and an initial learning rate of 0.01. I used PyTorch’s lr\_scheduler function to decrease the learning rate by a factor of 10 after every other epoch. I evaluated the model’s accuracy after every 20 batches of training data to determine the progression of the model’s accuracy. This helped me fine-tune the images-per-batch, learning rate, and learning rate scheduler. The code used to create and implement this model can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/Log_Reg.py).

## Feed-forward Neural Network

This model is a simple two-layer, 100 hidden-dimension, fully-connected neural network. Each layer of the network was followed by a non-linear activation function. I tested ReLU, Tanh, and Sigmoid activation functions and received the best results with the Sigmoid function. As with the logistic regression model above, the model took an input dimension of 105,528 and had an output dimension of 4. I used batches of 20 images for training, trained for 5 epochs, and evaluated the model’s performance after every 20 batches. The initial learning rate was set to 0.01 and decreased by a factor of 10 after the third epoch. The code I used to create and implement this model can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/FF_Neural_Net.py).

## Simple Convolutional Neural Network

For this model I used two convolutional layers separated by ReLU functions. The first layer took an input of 3 channels (one for each of the image color channels) and output 64 channels, representing 64 different kernels of size 7x7. I used a stride of one and “same” padding so that the output dimension was the same as the input dimension (224x224). The second convolutional layer used a kernel size of 5x5, stride of one, “same” padding, and output 128 channels. After both convolutional layers I used max pooling. I tested the model with average pooling, but the results were not as accurate. The final, linear readout layer took dimensions of 128x56x56 (128 output channels, and 56x56 images after pooling) and 4 (four photo categories). I tested this model with and without batch normalization and got the best results without batch normalization. I trained this model with batches of 20 images for 10 epochs. I started with a learning rate of 0.01 and decreased the learning rate by a factor of 10 after the fifth epoch. I evaluated the model after every 20 batches. I saved the model weights for the most accurate validation run and used those for the final exported model. The code I used to create and implement this model can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/Simple_CNN.py).

## ResNet Models

He et al.1 created residual network models as an attempt to build deeper models that don’t lose accuracy as more layers are added. They were successful in creating models that are much deeper than other models but solved the problem of accuracy loss while also significantly decreasing the number of parameters in their model. They developed models with depth ranging from 18 to 152 layers and won the 2015 ILSVRC competition by using an ensemble of these methods. These models were trained on a dataset of 1.28 million training images and evaluated on 50,000 validation images. The final test set consisted of 100,000 images. Their final ensemble method had a top-5 error rate of 3.57%.

I tested four different ResNet models available through Torchvision3. I chose to load these models with pre-trained weights, but to re-train all parameters as opposed to just the final fully-connected layer. I trained ResNet18, ResNet34, ResNet101, and ResNet152 and got the best results using ResNet101. I did attempt to retrain just the final layer of ResNet101, but the results were not as accurate as retraining the whole model. I tested these models with batch sizes ranging from four to 25 and initial learning rates from 0.001 to 0.1. My final model utilized batches of 20 images and an initial learning rate of 0.001. I trained for 25 epochs and decreased the learning rate by a factor of 10 every seven epochs. I evaluated the model using the validation images after every epoch and saved the model weights from the most accurate epoch to use in my final exported model. The code I used to create and implement this model can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/ResNet_TL.py).

# Model Performance

The ResNet101 model performed the best with an accuracy of 98.1% on the validation data set. This was followed by the simple CNN (62.5%), FNN (53.1%), and the logistic regression model (51.9%). I wanted a way to visualize model performance, so I decided to randomly select 10 photos from the my whole dataset (but ensure that at least two photos from each category were represented) and the run them all through each of my models to see how the models each classified the same photos (Figures 3-6). Not surprisingly, the ResNet101 model predicted all 10 images correctly. What was surprising is that the logistic regression model predicted the second most correct labels (7), followed by the FNN (6) and the CNN (2). The code I used to select the 10 random photos can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/Select_Final_Test_Set.py), and the code used to evaluate each of my models on those 10 photos can be found [HERE](https://github.com/Liptoni/Springboard/blob/master/Photo_Classification_Capstone/python/Visualize_Models.py).

Figure 3. Predictions from Logistic Regression Model

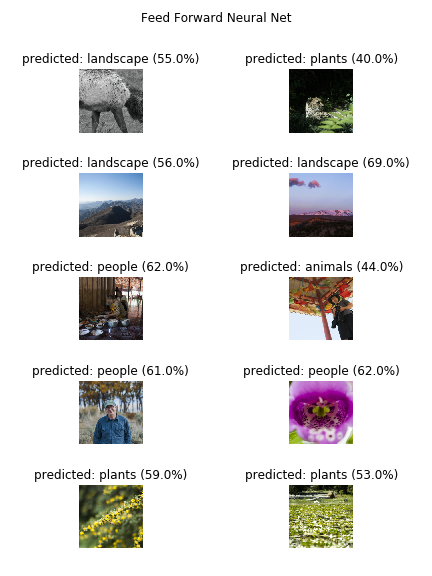
Figure 4. Predictions for Feed-forward Neural Net

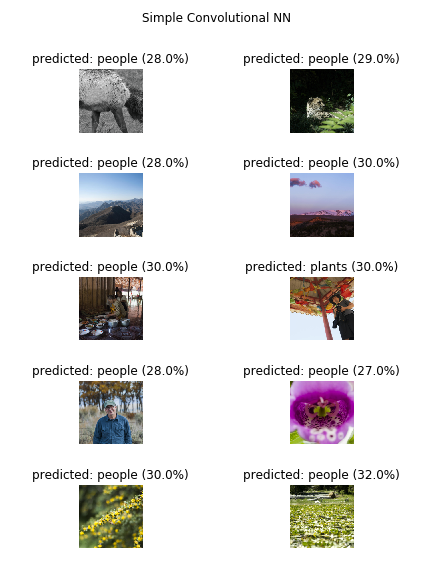
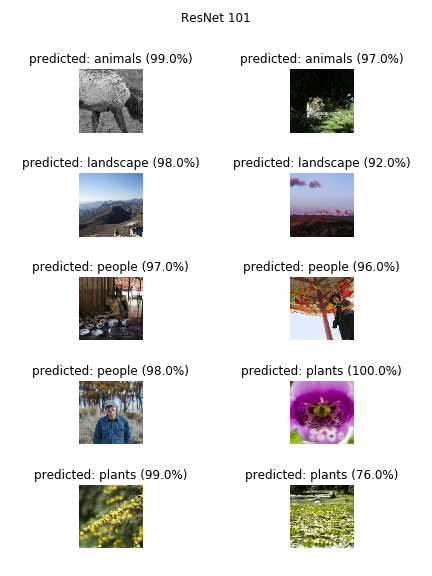
Figure 5. Predictions from Simple Convolutional Neural Net

Figure 6. Predictions from ResNet101 Model



# Conclusion

While it is not surprising that the ResNet model had the highest accuracy, I was surprised at just how accurate it was. With only 640 training images and 160 validation images, and in roughly 45 minutes, I was able to get a model that predicted those validation images with over 98% accuracy. This shows the true power of using transfer learning and adapting a model that has been previously trained with over a million images over the course of days or weeks.

The implications of this are quite promising in the context of both the stock photography site as well as for classifying photographs for individual photographers. With more training photos and more categories, we could easily adapt this model to predict more image categories. Also, my model only predicts the top-1 category. Given enough categories and images we could adapt this model to give the top n-categories. This could allow for images to be assigned multiple classifiers i.e. tags, which would be useful in both potential client contexts. And, while we would need to obtain more photos for training, we would not need nearly as many images as the initial model required reducing the amount of human labor required to get a very accurate image classifier.

# 7. References

1. He, Kaiming et al. 2015. Deep Residual Learning for Image Recognition. Available: <https://arxiv.org/pdf/1512.03385.pdf>

2. PyTorch. Transfer Learning Tutorial. Accessed June 2018. Available: <https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html>

3. PyTorch. Source code for torchvision.models.resnet. Accessed June 2018. Available: <https://pytorch.org/docs/stable/_modules/torchvision/models/resnet.html#resnet18>