Machine Learning Engineer Nanodegree

Capstone Project

Creating a Custom Dataset and Classifying Grocery Item Images using Deep Learning

Cassandra Durkee

January 2020

1 Definition

1.1 Project Overview

Automation is one of the largest growing applications in AI, with the goal of replacing tedious jobs and reducing time and energy devoted to mundane daily tasks. Spam filters for email, warehouse sorting robots, and even more ambitious projects such as data analytics or self-driving cars are all examples of how we are increasingly relying more heavily on automation in our personal lives and as a society. Using machine learning makes it possible for these tools to navigate through dynamic and ambiguous situations by identifying patterns among each unique scenario that is encountered.

One area that is deeply entrenched in an average person's daily life is grocery shopping and food preparation. While there have been advances in making this process easier and more efficient – online food shopping, food delivery, and millions of recipes available online – there is a lot of room for making the process easier. For example, using computer vision techniques in AI could automate item inventory and shelfing through image recognition of grocery items. Another possibility is using consumer purchase history to make food recommendations to buyers, to help them find desired items more easily.

1.2 Problem Statement

There is a growing number of publicly available datasets containing images of grocery items and produce items. The datasets vary widely across grocery item and country of origin. The goal of this particular project is to combine existing datasets that contain images of grocery items to create a customized larger and more comprehensive grocery dataset, and then find a pre-trained model that will classify the combined images at an accuracy of at least 70%. Considering that each grocery store has thousands of unique items which vary by grocery chain and region, it is out of the scope of this project to design a fully comprehensive dataset of images of grocery items. This project is designed to be a first stepping stone to see whether existing data can be combined to make a larger customized dataset.

My personal motivation for this project stems from my interest in finding practical ways to use more environmentally friendly alternatives in our routines. One particular area is from our increasing reliance of take-out food instead of cooking food at home. Take-out food comes in disposable packaging and generates a lot of waste. Additionally, a large portion of the food purchased at grocery stores ends up not being used and discarded. By introducing ways to make grocery shopping and cooking easier, and ways to help consumers make smarter purchases when grocery shopping, this may incentivize people to cook at home more often, and to use all of the food they purchase so there is less waste.

1.3 Metrics

The datasets (discussed in 2.1 – Data Exploration and Visualization) contain multiple classes and roughly the same number of images per class. A good accuracy metric for this type of the dataset is the Classification Accuracy metric, which is the number of correct predictions divided by the total number of predictions:

$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

Image found here

This metric simply compares the ratio of correct classifications to the overall classifications that the model made.

2 Data Exploration and Visualization

2.1 Freiberg Dataset

The Freiberg Grocery dataset contains 4,947 images for 25 classes of grocery items. Each image is taken with four different smartphones across different grocery stores or apartments in Freiberg, Germany. Since the images were taken in real-world settings, they have different lighting, angles, and settings. Classes are generic food items (e.g., water, oil), so packaging and size is varied in each class. The images were downscaled to 256 x 256 pixels and padded with gray borders. I chose this dataset because of its variability of images in each class. The example of images of jam below demonstrate that the model should be able to identify jam in any region of the world, because the images vary between angle, brand, and lighting.

Some example images of the Freiberg Grocery dataset from the jam class:

These images, taken from the jam class, show the angle, lighting, and brand variance within each class.

Full dataset can be found here

2.2 Fruits 360 Dataset

The Fruits 360 dataset contains 55,244 images for 81 classes of produce. Each image is taken in a controlled setting by placing them on the shaft of a low speed motor rotating at 3 rpm, and a 20 second movie was recorded using a Logitech C920 camera. The images share the same lighting and white background due to an algorithm designed replace any background noise with a standard white color. The images were resized to 100 x 100 pixels.

Some example images of the Fruits 360 dataset from the lychee class:

These images, taken from the lychee class, show that the images within each class vary only by the angle in which they were taken. Lighting, size, and background remain the same.

Full dataset can be found here.

3 Method

Resources

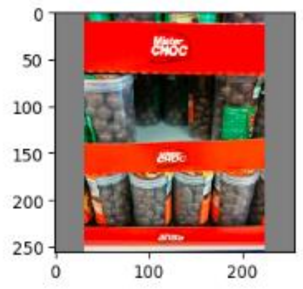
- Python v 3.7.3
- PyTorch v 0.4.1
- Torchvision v 0.2.1
- Numpy v 1.16.4
- Matplotlib v 3.1.1
- Imgaug 0.3.0
- Jupyter Labs
- Windows 10 OS

3.1 Data Augmentation

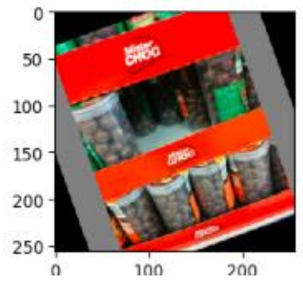
I originally trained the benchmark and comparison model using the Freiberg Grocery dataset as is. The result was that many of the classes in the Freiberg dataset were under 10% accuracy, especially with classes that contained than 100 images.

Since the Freiberg Grocery dataset contains substantially less images than the Fruits 360 dataset, I used image augmentation to increase the dataset size. The library imaug provides easy syntax to transform pre-existing images in a variety of ways ranging from simple augmentation techniques (e.g., rotation, flip, crop) to complex augmentation techniques (e.g., implementing weather conditions, changing lighting and color gradients). Using these techniques, it is possible to increase a dataset size to several times larger its original size, or add in features that are not easy to collect real data on. Data augmentation is applied only to the training set, the test set should contain only original images.

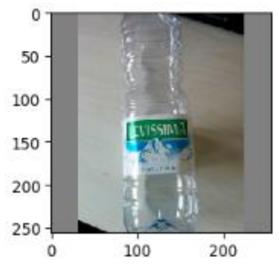
An example of image augmentation using only 25 degree rotation:



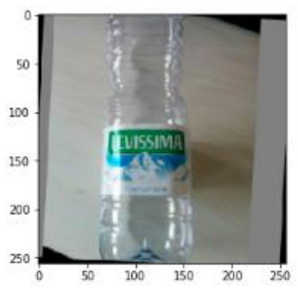
```
Augmented:
```



An example of image augmentation using rotation, image flipping, and cropping:



```
]: <matplotlib.image.AxesImage at 0x1b34ebe58d0>
```



The code to crop, rotate, and flip this image is fairly straightforward:

```
nimg = imageio.imread('C:\\put image path here')

ia.imshow(nimg)

ia.seed(3)

seq = iaa.Sequential([
    iaa.Fliplr(0.5),
    iaa.Crop(percent=(0, 0.1)),
    iaa.Affine(rotate=(-25, 25))
], random_order=True)

nimg_aug = seq.augment_image(nimg)

plt.imshow(nimg_aug)
```

Every image in the training folder Freiberg Grocery dataset was randomly flipped left to right, cropped, and rotated 25 degrees. This doubled the dataset size for every class in the Freiberg Grocery dataset. Some classes still had a smaller number of images after augmentation, so another round of augmentation was applied to those specific classes, to double their size again.

3.2 Data Cleaning and Preparation

Images from both datasets were merged into a 'train' set and a 'test' set. Images for each class was separated in to different folders, and the format was changed to uniformly match. An example directory path for an image of a banana is: train/Banana/banana_36.png.

3.3 Data Transformation

Creating a custom dataset in PyTorch is possible by overriding the __init___, __len__, and __getitem__ subclass functions of the Dataset class in PyTorch.

The __init__ function is used for data preparation, transformation, reading files and images, and setting up data to be read by index. The __getitem__ function is used to return data and labels, apply initiated transformations, set up GPU, and return data as tensors. The __len__ function returns the number of data points in the new dataset.

3.4 Transforming Data

The images need to be altered into having shared characteristics that are compatible with the type of algorithm that will be used to process them.

To transform the training images, I used:

- Resize: resized all images to 224 pixels. 224 is the standard size used for ResNet models.
- RandomRotation: some images were randomly flipped 90 degrees.
- RandomResizedCrop: some images were randomly cropped, so that there was more variance among the images.

- RandomHorizontalFlip: some images were randomly flipped, which gives more variance among images and potentially prevents overfitting.

For the test images, I used:

- Resize: Images were resized to 224, same as train images.
- CenterCrop: Images were cropped to 224.

3.5 Transfer Learning from Pre-Trained Network

3.5.1 ResNet18

ResNet is a pre-trained convolutional neural network designed for image classification tasks. ResNet18 has 18 layers, and was pre-trained on 1000 image classes.

When a model is pre-trained, it means that the weights and parameters have previously been trained on a different dataset, which saves time and energy of not having to start from scratch every time you want to train a new dataset. What sets ResNet models apart from other pre-trained CNNs is that its architecture has built-in features that diminish the vanishing gradient problem. The vanishing gradient problem occurs when a network has so many layers that the gradients from the loss function diminish to zero from becoming too saturated. Subsequently, the weights are never adjusted to updated values, causing learning to stagnate. The deep architecture of ResNet models works because it has a feature called skip connection built into it:

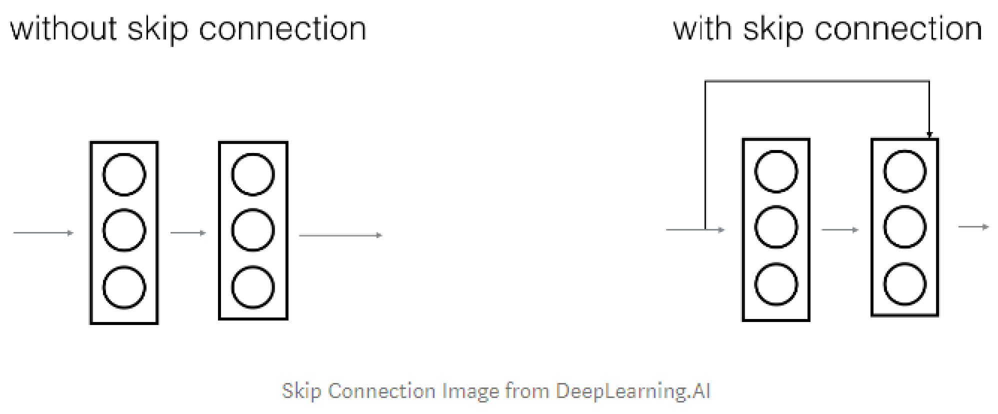
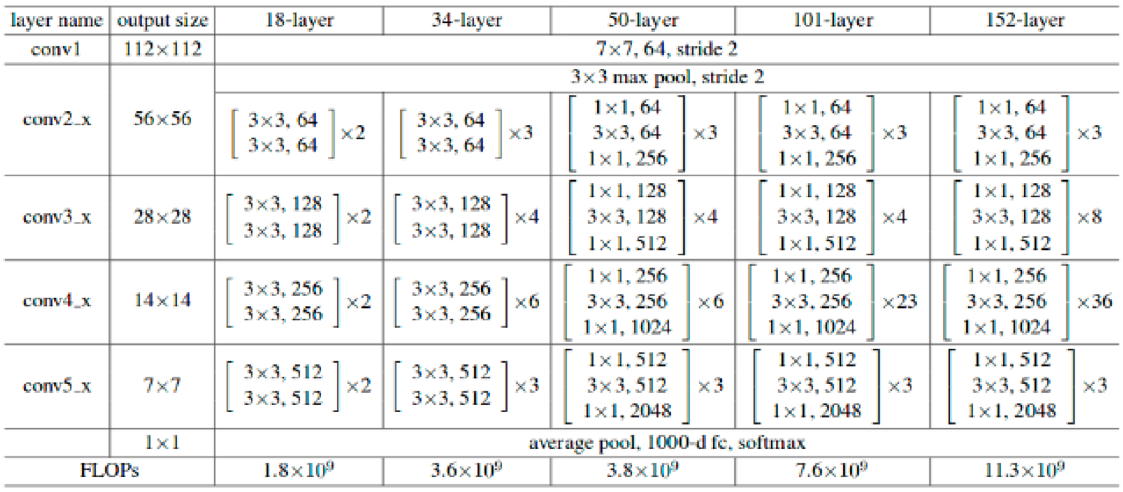


Image found here

With skip connection, the previous layer is updating the next layer, but it is also updating the layers following the next layer. This prevents the weights from diminishing to zero.

The following diagram shows ResNet architectures:



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

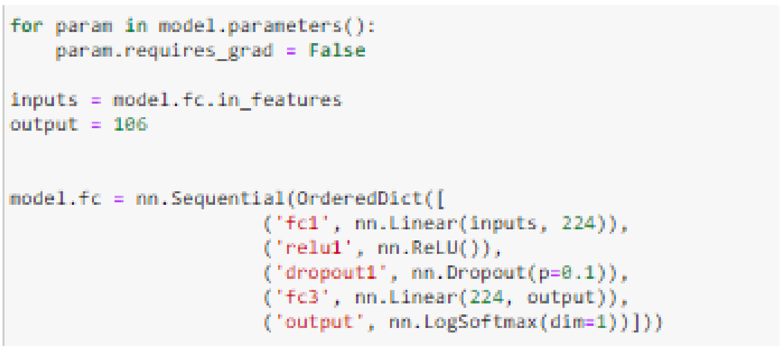
ResNet Architectures

Image found here

3.5.2 Freezing Layers for ResNet18

The benefit of using a pre-trained network such as ResNet is that the weights and parameters have already been initialized and trained on a different dataset. Since the model is pre-trained, it has already learned certain patterns from the previous dataset, making feature extraction more efficient for the new dataset. This method of training is called transfer learning, because features detected from the previous training session can be transferred to new images.

When using a pre-trained model, there is the option to freeze all, or particular layers, meaning that the parameters that are frozen remain fixed. To train on this dataset, I froze the final layer, and added a new layer:



```
for param in model.parameters():
    param.requires_grad = False

inputs = model.fc.in_features
output = 106


model.fc = nn.Sequential(OrderedDict([
                      ('fc1', nn.Linear(inputs, 224)),
                      ('relu1', nn.ReLU()),
                      ('dropout1', nn.Dropout(p=0.1)),
                      ('fc3', nn.Linear(224, output)),
                      ('output', nn.LogSoftmax(dim=1))]))
```

3.5.3 Hyperparameter Tuning for ResNet18

I used the Adam optimization algorithm for the optimization method. Adam, (an abbreviation of adaptive moment estimation), was created specifically for deep learning tasks. The Adam algorithm determines learning rates for different parameters by using estimations of the first and second moments of gradient for each weight in the neural network. I set the learning rate at .01.

I used cross-entropy loss for the loss function. Cross-entropy loss is used for classification tasks. Cross entropy gives greater loss to wrong predictions, particularly if they are done in higher confidence, meaning that an additional weight is placed on the confidence of each incorrect prediction.

I used step LR scheduler with a step size of 5, and a gamma of 0.1. The learning rate scheduler uses step decay to prevent the parameter vector from having too high of kinetic energy, and overshooting the local minima, or from having too low of kinetic energy, and settling in a false minima.

3.5.4 DenseNet121

Another pre-trained model I used for testing was DenseNet121. I tried this model after noticing that my accuracy for classes in the Freiberg Grocery Dataset were getting much lower accuracy than classes from the Fruits 360 dataset. DenseNet models differ from ResNet models in that DensetNet models use a technique called “feature re-use”. With ResNet models, every layer has a set of weights that need to be trained. With DenseNet models, every layer is directly connected with every other layer. By making the path between the input layer and output layer more direct, information and gradients do not get lost between inputs and outputs.

A visual of the connections of layers in DenseNet models:

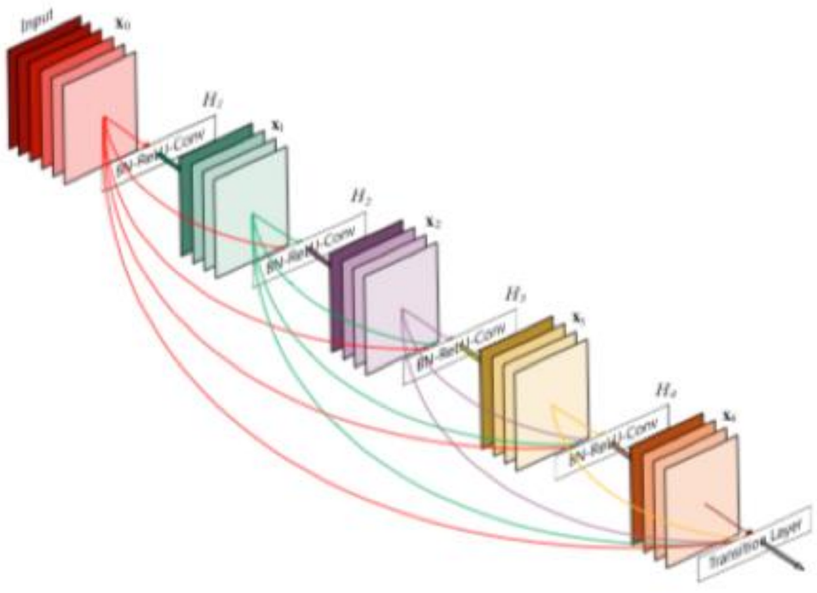
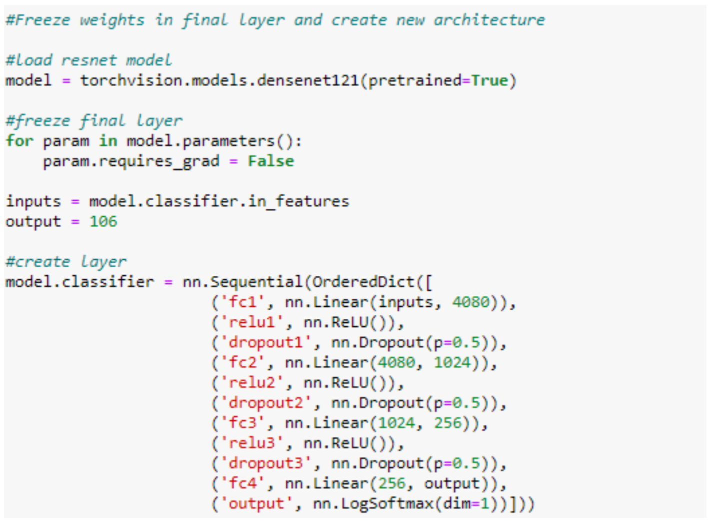


Figure 1. DenseNet with 5 layers with expansion of 4. [1]

Image found here

3.5.5 Freezing Layers for DensetNet121

For this model, I added more layers to the final frozen layer, and increased dropout from 0.1 to 0.5:



```
#Freeze weights in final layer and create new architecture

#load resnet model
model = torchvision.models.densenet121(pretrained=True)

#freeze final layer
for param in model.parameters():
    param.requires_grad = False

inputs = model.classifier.in_features
output = 106

#create layer
model.classifier = nn.Sequential(OrderedDict([
                      ('fc1', nn.Linear(inputs, 4080)),
                      ('relu1', nn.ReLU()),
                      ('dropout1', nn.Dropout(p=0.5)),
                      ('fc2', nn.Linear(4080, 1024)),
                      ('relu2', nn.ReLU()),
                      ('dropout2', nn.Dropout(p=0.5)),
                      ('fc3', nn.Linear(1024, 256)),
                      ('relu3', nn.ReLU()),
                      ('dropout3', nn.Dropout(p=0.5)),
                      ('fc4', nn.Linear(256, output)),
                      ('output', nn.LogSoftmax(dim=1))]))
```

3.5.6 Hyperparameter Tuning for DenseNet121

After some trial and error, I found that DenseNet121 was successful when I changed the optimizer from Adam to Stochastic Gradient Descent (SGD). The optimal learning was 0.03, instead of 0.01 as it was for ResNet18.

4 Benchmark

4.1 Benchmark Model ResNet50

My benchmark model was the ResNet50 model with the SGD optimization method. ResNet50 is from the same model architecture as ResNet18, however it has 50 layers. Additionally, ResNet50 has 3 layers, whereas ResNet18 has 2 layers:

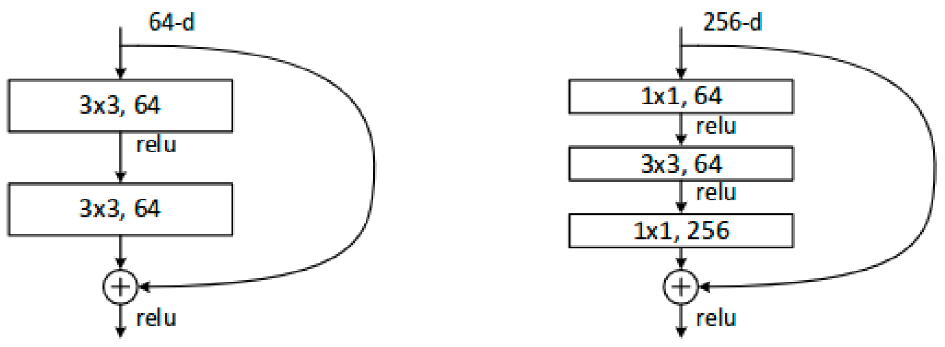


Image found here

The final layer and the hyperparameters were the same as the ResNet18 model.

4.2 Benchmark Results

4.2.1 Original Freiberg Grocery Dataset

Using this model, ResNet50 could only get accuracy up to 52% when running the training data for 15 epochs. I attempted to add another Dropout layer and another Linear layer to ResNet50, however that caused the model to stop learning. When I switched the optimization method from stochastic gradient method to Adam, the model's accuracy went up to 58%. The lowest accuracy was in the Freiberg Grocery Dataset, with some classes having 0% accuracy.

4.2.2 Modified Freiberg Grocery Dataset

I added enough augmented images to ensure that each class had at least 300 images, and trained the same model again. This time around, the accuracy was only 1% overall. Classes from both datasets failed to train. The class with the highest accuracy was "pineapple mini" from the Fruits 360 dataset, at 41% accuracy. All but one other class had 0% accuracy (apple_granny_smith class had 5% accuracy).

5 Results

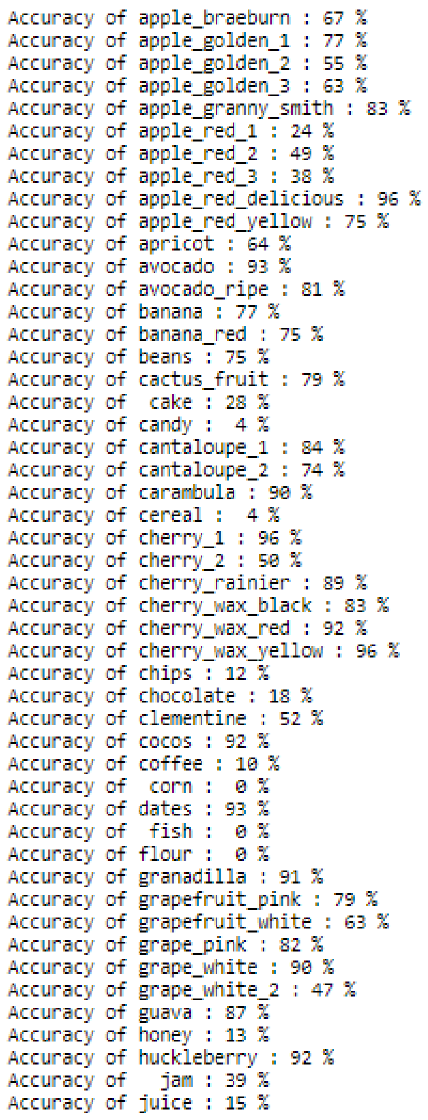
I initially trained the models using the original images from the Freiberg and Fruits datasets. After receiving feedback to increase the size of the Freiberg dataset, I augmented some of the original images to make the dataset size on par with the size of the Fruits dataset. The results below show how increasing my dataset size required me to make changes in my model choice, and my hyperparameter tuning.

5.1 Benchmark ResNet50 vs ResNet18

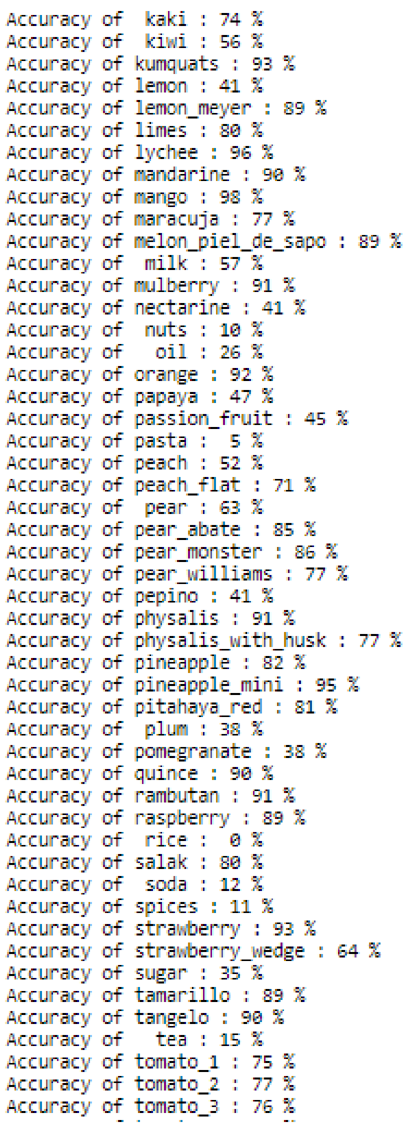
5.1.1 Original Freiberg Grocery Dataset

The overall accuracy after 12 epochs was 72%. Switching from ResNet50 to ResNet18 caused the accuracy to go up 14% to 72%. The increase was noticeably for classes from the Freiberg Grocery dataset. With the ResNet50 model, many classes from the Freiberg dataset were below 10%. While there are still some classifications that are very low in the Freiberg set, there is an improvement. ResNet18 is a clear improvement over ResNet50.

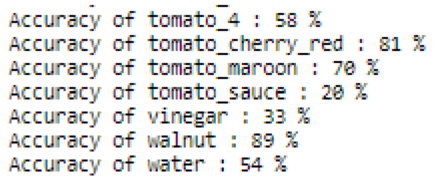
Here is the breakdown of accuracy for each individual class using ResNet18:



```
Accuracy of apple_braeburn : 67 %
Accuracy of apple_golden_1 : 77 %
Accuracy of apple_golden_2 : 55 %
Accuracy of apple_golden_3 : 63 %
Accuracy of apple_granny_smith : 83 %
Accuracy of apple_red_1 : 24 %
Accuracy of apple_red_2 : 49 %
Accuracy of apple_red_3 : 38 %
Accuracy of apple_red_delicious : 96 %
Accuracy of apple_red_yellow : 75 %
Accuracy of apricot : 64 %
Accuracy of avocado : 93 %
Accuracy of avocado_ripe : 81 %
Accuracy of banana : 77 %
Accuracy of banana_red : 75 %
Accuracy of beans : 75 %
Accuracy of cactus_fruit : 79 %
Accuracy of  cake : 28 %
Accuracy of candy :  4 %
Accuracy of cantaloupe_1 : 84 %
Accuracy of cantaloupe_2 : 74 %
Accuracy of carambula : 90 %
Accuracy of cereal :  4 %
Accuracy of cherry_1 : 96 %
Accuracy of cherry_2 : 50 %
Accuracy of cherry_rainier : 89 %
Accuracy of cherry_wax_black : 83 %
Accuracy of cherry_wax_red : 92 %
Accuracy of cherry_wax_yellow : 96 %
Accuracy of chips : 12 %
Accuracy of chocolate : 18 %
Accuracy of clementine : 52 %
Accuracy of cocos : 92 %
Accuracy of coffee : 10 %
Accuracy of  corn :  0 %
Accuracy of dates : 93 %
Accuracy of  fish :  0 %
Accuracy of flour :  0 %
Accuracy of granadilla : 91 %
Accuracy of grapefruit_pink : 79 %
Accuracy of grapefruit_white : 63 %
Accuracy of grape_pink : 82 %
Accuracy of grape_white : 90 %
Accuracy of grape_white_2 : 47 %
Accuracy of guava : 87 %
Accuracy of honey : 13 %
Accuracy of huckleberry : 92 %
Accuracy of   jam : 39 %
Accuracy of juice : 15 %
```



```
Accuracy of  kaki : 74 %
Accuracy of  kiwi : 56 %
Accuracy of kumquats : 93 %
Accuracy of lemon : 41 %
Accuracy of lemon_meyer : 89 %
Accuracy of limes : 80 %
Accuracy of lychee : 96 %
Accuracy of mandarine : 90 %
Accuracy of mango : 98 %
Accuracy of maracuja : 77 %
Accuracy of melon_piel_de_sapo : 89 %
Accuracy of  milk : 57 %
Accuracy of mulberry : 91 %
Accuracy of nectarine : 41 %
Accuracy of  nuts : 10 %
Accuracy of   oil : 26 %
Accuracy of orange : 92 %
Accuracy of papaya : 47 %
Accuracy of passion_fruit : 45 %
Accuracy of pasta :  5 %
Accuracy of peach : 52 %
Accuracy of peach_flat : 71 %
Accuracy of  pear : 63 %
Accuracy of pear_abate : 85 %
Accuracy of pear_monster : 86 %
Accuracy of pear_williams : 77 %
Accuracy of pepino : 41 %
Accuracy of physalis : 91 %
Accuracy of physalis_with_husk : 77 %
Accuracy of pineapple : 82 %
Accuracy of pineapple_mini : 95 %
Accuracy of pitahaya_red : 81 %
Accuracy of  plum : 38 %
Accuracy of pomegranate : 38 %
Accuracy of quince : 90 %
Accuracy of rambutan : 91 %
Accuracy of raspberry : 89 %
Accuracy of  rice :  0 %
Accuracy of salak : 80 %
Accuracy of  soda : 12 %
Accuracy of spices : 11 %
Accuracy of strawberry : 93 %
Accuracy of strawberry_wedge : 64 %
Accuracy of sugar : 35 %
Accuracy of tamarillo : 89 %
Accuracy of tangelo : 90 %
Accuracy of   tea : 15 %
Accuracy of tomato_1 : 75 %
Accuracy of tomato_2 : 77 %
Accuracy of tomato_3 : 76 %
```



```
Accuracy of tomato_4 : 58 %
Accuracy of tomato_cherry_red : 81 %
Accuracy of tomato_maroon : 70 %
Accuracy of tomato_sauce : 20 %
Accuracy of vinegar : 33 %
Accuracy of walnut : 89 %
Accuracy of water : 54 %
```

The most common misclassifications were from the Freiberg Grocery Dataset. The creators of the Freiberg Grocery Dataset achieved an accuracy of 78.9% when they tested their dataset, however, for this algorithm, the overall accuracy for this portion of the dataset was much lower.

The Fruits 360 dataset images had a much higher overall accuracy than the Freiberg Grocery Dataset images. On Kaggle, there are some models that have achieved accuracy over 90% for the Fruits 360 dataset using the ResNet50 model.

5.1.2 Modified Freiberg Grocery Dataset

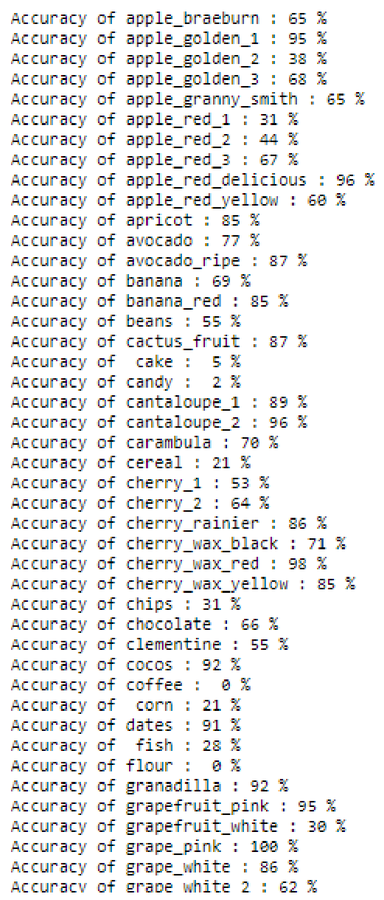
When adding augmented images from the Freiberg Grocery dataset, the ResNet18 and 50 models did not train. Almost every image was predicted as one class. These models no longer worked with the increased dataset size. I tried changing some of the hyperparameters; for example, changing the learning rate, adding new layers to the final layer, and changing dropout, but I still ended up with the same results.

5.2 Benchmark ResNet50 vs DenseNet121

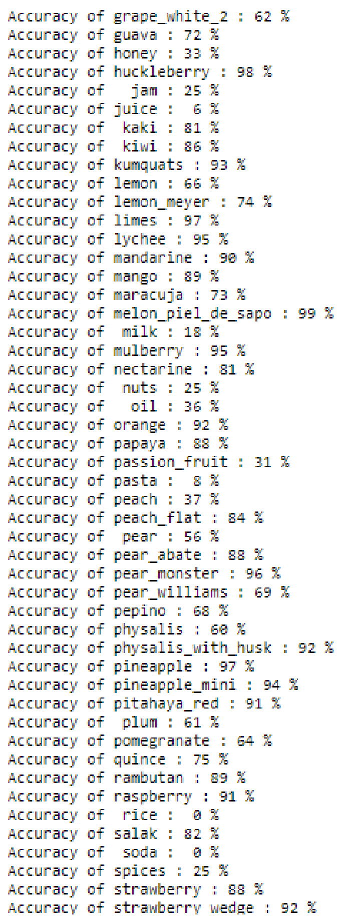
The DenseNet121 model was only trained on the modified dataset, after ResNet18 and ResNet50 were no longer working.

Since ResNet models were no longer training on my new data, I tried using DenseNet121. As mentioned previously, some of the hyperparameters had to be changed before the model would train. I trained DensetNet for a total of 7 epochs, and achieved 73% overall accuracy. The Freiberg Grocery dataset classes were still lower than the classes from the Fruits360 dataset, but there was an increase in accuracy overall.

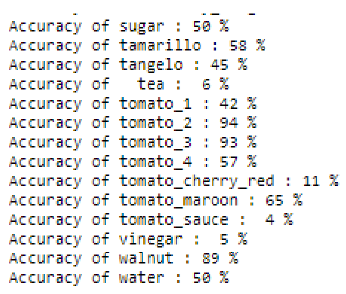
Here is the breakdown of accuracy for each individual class using DenseNet121:



```
Accuracy of apple_braeburn : 65 %
Accuracy of apple_golden_1 : 95 %
Accuracy of apple_golden_2 : 38 %
Accuracy of apple_golden_3 : 68 %
Accuracy of apple_granny_smith : 65 %
Accuracy of apple_red_1 : 31 %
Accuracy of apple_red_2 : 44 %
Accuracy of apple_red_3 : 67 %
Accuracy of apple_red_delicious : 96 %
Accuracy of apple_red_yellow : 60 %
Accuracy of apricot : 85 %
Accuracy of avocado : 77 %
Accuracy of avocado_ripe : 87 %
Accuracy of banana : 69 %
Accuracy of banana_red : 85 %
Accuracy of beans : 55 %
Accuracy of cactus_fruit : 87 %
Accuracy of  cake :  5 %
Accuracy of candy :  2 %
Accuracy of cantaloupe_1 : 89 %
Accuracy of cantaloupe_2 : 96 %
Accuracy of carambula : 70 %
Accuracy of cereal : 21 %
Accuracy of cherry_1 : 53 %
Accuracy of cherry_2 : 64 %
Accuracy of cherry_rainier : 86 %
Accuracy of cherry_wax_black : 71 %
Accuracy of cherry_wax_red : 98 %
Accuracy of cherry_wax_yellow : 85 %
Accuracy of chips : 31 %
Accuracy of chocolate : 66 %
Accuracy of clementine : 55 %
Accuracy of cocos : 92 %
Accuracy of coffee :  0 %
Accuracy of  corn : 21 %
Accuracy of dates : 91 %
Accuracy of  fish : 28 %
Accuracy of flour :  0 %
Accuracy of granadilla : 92 %
Accuracy of grapefruit_pink : 95 %
Accuracy of grapefruit_white : 30 %
Accuracy of grape_pink : 100 %
Accuracy of grape_white : 86 %
Accuracy of grape_white_2 : 62 %
```



```
Accuracy of grape_white_2 : 62 %
Accuracy of guava : 72 %
Accuracy of honey : 33 %
Accuracy of huckleberry : 98 %
Accuracy of   jam : 25 %
Accuracy of juice :  6 %
Accuracy of  kaki : 81 %
Accuracy of  kiwi : 86 %
Accuracy of kumquats : 93 %
Accuracy of lemon : 66 %
Accuracy of lemon_meyer : 74 %
Accuracy of limes : 97 %
Accuracy of lychee : 95 %
Accuracy of mandarine : 90 %
Accuracy of mango : 89 %
Accuracy of maracuja : 73 %
Accuracy of melon_piel_de_sapo : 99 %
Accuracy of  milk : 18 %
Accuracy of mulberry : 95 %
Accuracy of nectarine : 81 %
Accuracy of  nuts : 25 %
Accuracy of   oil : 36 %
Accuracy of orange : 92 %
Accuracy of papaya : 88 %
Accuracy of passion_fruit : 31 %
Accuracy of pasta :  8 %
Accuracy of peach : 37 %
Accuracy of peach_flat : 84 %
Accuracy of  pear : 56 %
Accuracy of pear_abate : 88 %
Accuracy of pear_monster : 96 %
Accuracy of pear_williams : 69 %
Accuracy of pepino : 68 %
Accuracy of physalis : 60 %
Accuracy of physalis_with_husk : 92 %
Accuracy of pineapple : 97 %
Accuracy of pineapple_mini : 94 %
Accuracy of pitahaya_red : 91 %
Accuracy of  plum : 61 %
Accuracy of pomegranate : 64 %
Accuracy of quince : 75 %
Accuracy of rambutan : 89 %
Accuracy of raspberry : 91 %
Accuracy of  rice :  0 %
Accuracy of salak : 82 %
Accuracy of  soda :  0 %
Accuracy of spices : 25 %
Accuracy of strawberry : 88 %
Accuracy of strawberry_wedge : 92 %
```



```
Accuracy of sugar : 50 %
Accuracy of tamarillo : 58 %
Accuracy of tangelo : 45 %
Accuracy of   tea :  6 %
Accuracy of tomato_1 : 42 %
Accuracy of tomato_2 : 94 %
Accuracy of tomato_3 : 93 %
Accuracy of tomato_4 : 57 %
Accuracy of tomato_cherry_red : 11 %
Accuracy of tomato_maroon : 65 %
Accuracy of tomato_sauce :  4 %
Accuracy of vinegar :  5 %
Accuracy of walnut : 89 %
Accuracy of water : 50 %
```

6 Conclusion

6.1 Summary

This project covered multiple different stages of data preparation and image recognition:

- Cleaning and organizing images
- Augmenting images to increase dataset size
- Changing ___getitem__ and __len__ subclass to prep for custom data
- Transform images by resizing and re-cropping
- Load Data
- Try different pre-trained models
- Try different hyperparameters
- Train
- Test

While creating a dataset and an algorithm that can classify different images of grocery items at scale is far beyond the scope of this Capstone project, I was able to cover many steps of the machine learning workflow, including: cleaning data, preparing data, transforming data, developing and training a model, and evaluating the model.

The two datasets I chose were very different in their content, and by the way that images were obtained and formatted. This makes it difficult to preprocess them and find a model that can identify both sets at a high rate. Regardless, this was an interesting exploration of different datasets, as well as the steps needed to create a custom dataset.

My goal was to combine two large publicly available datasets containing images of grocery items, and find a model that could predict their classification at a minimum of 70%. This project only encompasses 106 different classes of grocery items, and most of them are produce items. The task of grocery item classification is still far off, but this project is still a step in the direction of an area that could greatly benefit from AI technology.

6.2 Reflection

The biggest takeaway I got from this project is the importance of exploring different models and parameters. I find it so interesting that my model would no longer train on ResNet when I increased the dataset size. Something about my dataset size and content required me to change my model and change my hyperparameters. To my knowledge, our understanding exactly why these changes occur is still far away. It's possible to make educated guesses, for example, using cross validation and regularization to reduce overfitting. But the only real way to test is to make these educated guesses, check the result, and try again.

On my own time, I would like to make a few additional changes to this project:

- As of now, I am still trying to improve accuracy with the DenseNet121 model. Unfortunately, I do not the computer power to run more tests before the deadline of this project, so I am doing this on my own time. For this scope of this project, the current results will suffice, as I achieved an accuracy above 70%.
- Turning most of the processes into functions. This part will require some thought, because a lot of the data cleaning and merging for this project are specific to the datasets I worked with, but I think this would be a fun side-project to practice my python. I would also like to try to clean up my python code a bit and see if I can make things run more efficiently or with less lines of code.
- Creating a validation set. A validation set is important to detect over-fitting and to help fine tune hyper-parameters. Having the validation set would have made it easier when I was trying different hyper-parameters, and trying different epoch ranges. As of now, my only indicator of training performance is training loss. I will always add a validation set in the future when I create projects.

Some additional changes to consider:

- Utilizing helpful features from PyTorch as well as other libraries. PyTorch has an "early stopping" method that monitors validation accuracy and stops when it starts to increase. This helps prevent overfitting. Additionally, there are many cool features from scikit-learn, such as easy methods to split data into training, validation, and test sets. Now that I'm more comfortable with building my own project, I can consider these shortcuts for future projects.
- If I had unlimited GPU and more time, I would try re-testing with more hyperparameters, and also try different pre-trained models. I mentioned three models (ResNet18, 50, and DenseNet121), but I also tried a few other models (VGG16, 19, and ResNet101). There are a lot more models out there, and a lot of ways to tune these models, but I am satisfied with DenseNet121 for this project.