Prometheus Vision NN Final Report

Yuxuan Tian

Supervised by Joseph Vybihal

yuxuan.tian@mail.mcgill.ca

Abstract:

The ability to identify and categorize scenes are central to understanding the environment. This project consists of a convolutional neural network that takes in 1200 images as input. The images are collected from various sources, and they can have uneven qualities. A CNN model with customizable kernels is then trained based on the collected data to categorize the images into the following four categories: open, hall, stairs, and room. Various hyperparameters have been tested throughout the process, and despite all the results having relatively high variances, the model constantly gives much better results than the one from the previous year. The best prediction accuracy it obtained so far is 69%. Some potential causes of high variances will be discussed in the result session.

Introduction:

The Prometheus project's motivation is to develop a group of robots capable of exploring unknown areas, conducting rescue missions, and maneuvering freely in familiar regions. Those missions are all highly dependent on an intelligent vision system that helps the robot understand its surroundings.

In the Vision project, we write Convolutional neural networks and PCAs using different mathematical approaches to learn what the robot is seeing. Each method takes a guess on the input scene with a weight and then passes it to the democratic algorithm, making a final decision on the robot's location.

I'm responsible for writing a CNN that uses customizable kernels for feature extraction, allowing users to combine basic kernels (like horizontal and vertical detection, Etc.) to form new kernels that detect meaningful characteristics of an image. The main benefit of this design is that the program can keep only the important kernels that have a great impact on the later training process while maintaining relatively high accuracy. It greatly reduces the kernels needed compared to the method in which randomly generated kernels are used for feature extraction and are trained in the backpropagation process. Besides, being able to limit backpropagation to the fully connected layers means fewer computations are required. The above two factors contribute significantly to cutting down the time for trial and error. However, it is nearly impossible for the client to find the most optimized combinations

of kernels, since it has $1! + 2! + \cdots + n!$ possible combinations, assuming he created n basic kernels in total. The user has to keep trying various combinations until he gets a satisfying prediction accuracy.

Project Design

The entire project is built from scratch in Java, except it uses two libraries: Json-Simple that reads JSON file, and JFreeChart which helps to plot the cost in a graph to give the user a better intuition on the choice of parameters.

1. Data

The repository contained around 150 images for both training and testing purposes at the beginning, while many of them came with ambiguous classification and false labeling. I expanded and updated the dataset for each category multiple times and tested each version's prediction accuracy. Now there are 1998 images for training and 693 images for testing. All images, in addition to the ones provided in the repository, are collected by crawling Google and Naver using this[2] image crawler. It supports crawling both thumbnails and full images. In my case, I downloaded full images and resized them to 400x400.

hall *room* *stairs* *open*

Figure 1: A demo of images in all categories.

2. Image Processing

1) Color Mode

The program supports both RGB and grayscale images as input. Setting the color Option.grayscale will allow the program to convert images to grayscale while processing them. Throughout the first few trials, it turned out that using gray images generally produces a better result. Therefore, it is mainly used in the later testings.

2) Convolution

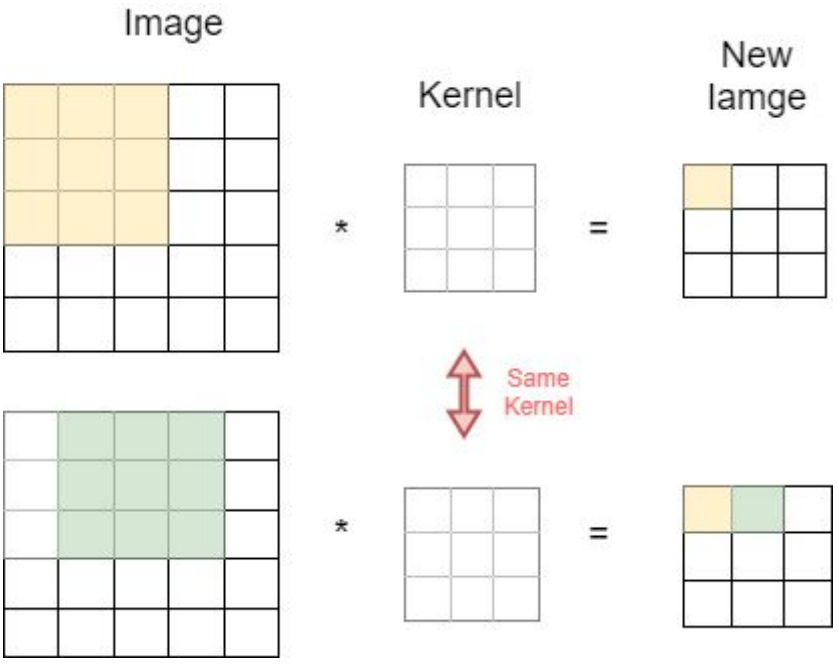


Figure 2: Steps of applying convolution to extract features

The objective of convolution is to extract features such as lines and edges of an image. A kernel is an (n x n) square matrix defined by the user. By applying convolution, the kernel slides across the original image while doing dot product with the area that it hops over. Each dot product produces a value, and it maps to a corresponding position of the new image, as shown in *Figure 2* above. The program also adds padding to ensure that the images have the same size before and after convolution. Most of the kernels used in the program can be found on this[1] page.

Figure 3 below is a demonstration of why certain kernels are capable of capturing features. {{1,0,-1},{1,0,-1},{1,0,-1}} is a classic kernel for vertical line detection. By convolving it with an image that has a vertical line, we get a new matrix that also has a darker vertical line (notice that in real life it is opposite, higher RGB value results in a brighter color, here it says darker just for demonstration). Despite the two lines having slightly different colors and thickness, the vertical line is still preserved.

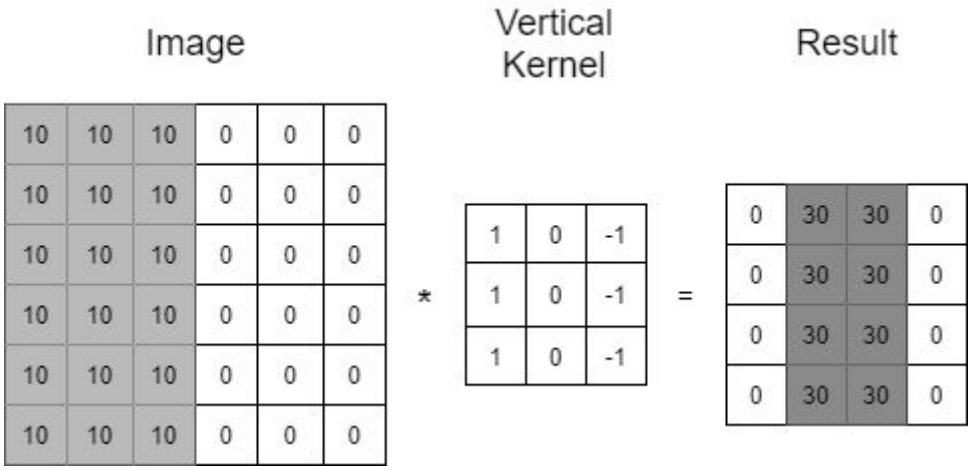


Figure 3: A demo showing why vertical kernel can extract vertical lines

3) Feature Extraction

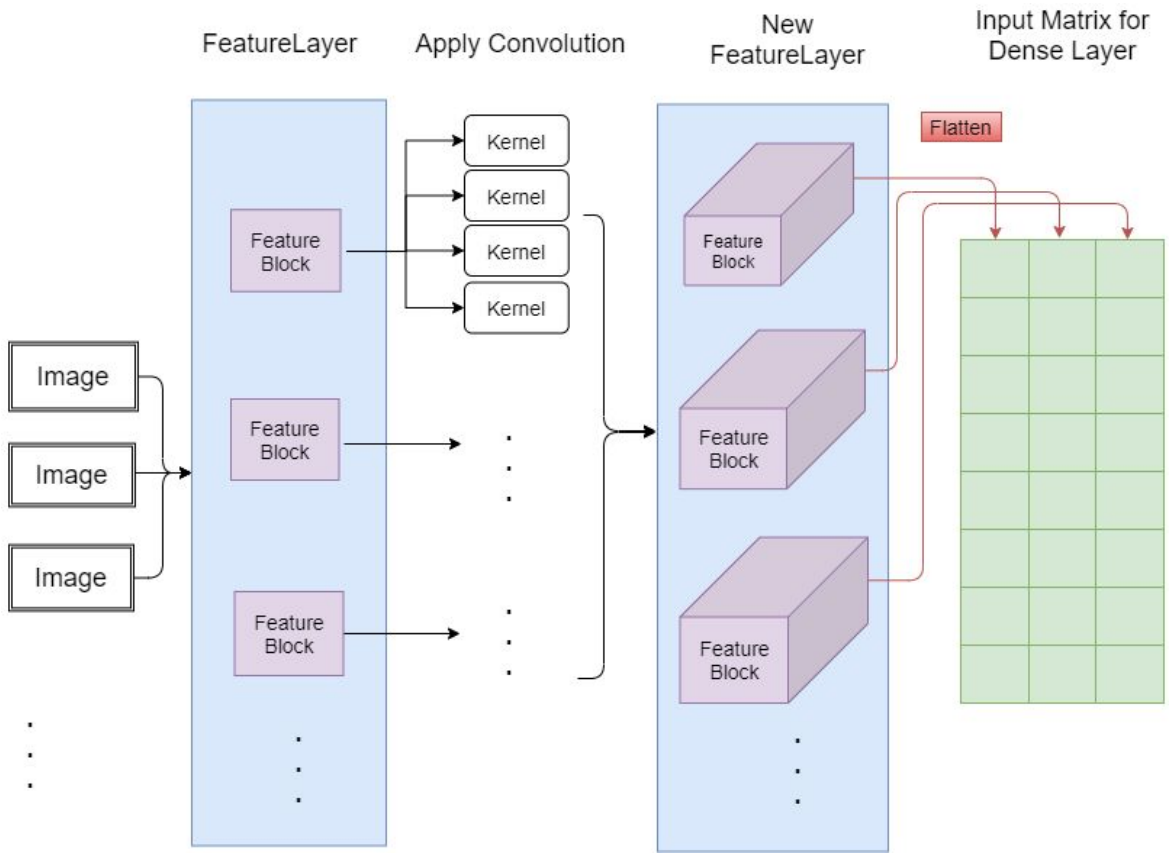


Figure 3: A flow chart of how the program processes images and feeds them into the dense layers.

To avoid confusion, I will start this section by briefly defining some keywords and essential building blocks of the project.

1. *Feature image*: A product of feature extraction through convolution. For example, the user applies convolution using a "Horizontal" kernel with an image A. Then, a feature image A(1) is formed, containing all horizontal lines of the original image A.
2. *FeatureBlock*: A structure that is built on feature images. Abstractly speaking, a FeatureBlock consists of the original image and all its feature images of a specific picture.
3. *FeatureLayer*: A wrapper that consists of FeatureBlocks.
4. *Flatten*: Reshape a FeatureLayer into a two-dimensional array.

FeatureBlock is the heart of the program in terms of feature extraction. It uses a HashMap<String, double[][]> to keep track of both the name and the actual RGB values of feature images. When the program processes an image from the database, it extracts its RGB value in the form of a two-dimensional array and puts it into the HashMap with the name "Original". In short, the HashMap inside the FeatureBlock only contains one image at the beginning. Later, when convolutions are applied to the "Original" image, the program then appends the additional feature images in the HashMap.

When convolving a new kernel with the original image or a specific feature image, the name of the new kernel is appended to the name of the feature image. For example, If an “EdgeDetection” kernel convolves with the “Original” image, a new feature image named “Original_EdgeDetection” will be produced and appended on the HashMap.

As illustrated in the introduction section, users can combine kernels to detect new features. By combining kernels, it really means applying one kernel after another. A good example will be "Original_GaussianBlur_Vertical". Since all images are resized and have different degrees of aliasing, applying "GaussianBlur" on the image helps to smoothen the picture and reduce aliasing. Therefore, extracting vertical lines after Gaussian blur results in a feature image with slightly less noise. However, there is one restriction when combining kernels: one kernel cannot be applied to the same image more than once, meaning that applying feature extraction to form images like "Original_sharpen_sharpen" or "Original_sharpen_vertical_sharpen" is prohibited.

The program uses polymorphism to ensure that the identical feature extractions are applied to all images. It wraps all feature blocks into one feature layer. When users call the "extract" method in the FeatureLayer class with a set of kernels, every feature block will call

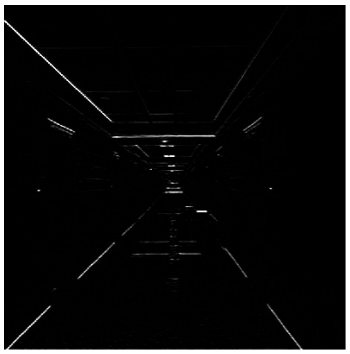
its feature extraction based on the input kernels. The "extract" method in the FeatureBlock class convolves every image contained in the feature block with all kernels provided in the parameter. Here is the pseudocode for the method:

```
extract ( list of Kernels){
        for every image within the featureBlock:
                for every kernel within the list of kernels:
                        if  the kernel has not been applied:
                                apply convolution using the current image and current kernel;
                        end if
                end for
        end for
end function
```

After feature extraction completes, all the feature blocks will be flattened and fed into the fully connected layers.



Original *EdgeDetection* *GaussianBlur + EdgeDetection*


Vertical *Horizontal* *Cross*

4) Fully Connected Layers

The fully connected layers is a classic multiplayer perceptron with vectorized implementation. Unfortunately, I was not able to find a library that uses multithreading to conduct matrix operations when I started the project, which deprived the meaning of

vectorization. But hopefully it can be used as a guideline on implementing vectorization for those who will be working on this project in the future, if they are not familiar with neural network implementations.

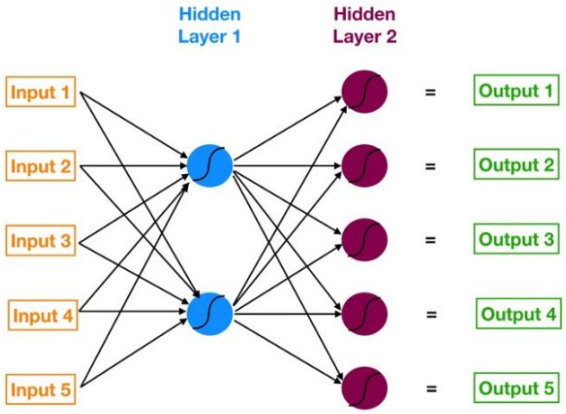


Figure 5: A flow chart explaining how neural networks function. It's found on this[2] website.

We will start by defining some mathematical notations, notice that all capital symbols represent a matrix:

1) *W*: weights
2) *B* : bias
3) m: number of images
4) g( ): activation function
5) *Z* : layer number activated
6) *A* : Z after activation. A = g(Z).
7) $W^{[L]}$ represents the weight of the x^{th} layer.

1. Forward Propagation

The input layer comes with label $A^{[0]}$, and the program will compute $A^{[1]}$ through $A^{[L]}$ through the following equations:

$$Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + B^{[L]}$$

$$A^{[L]} = g\,(Z^{[L]})$$

Note that since the program is designed to output probabilities for different categories, the activation function for the output layers $A^{[L]}$ has to be a softmax function.

2. Computing the cost:

Cost function measures the error of the model. In a sense, it tells the user how far away the algorithm is from the local minimum. Let $\hat{Y}$ denotes the output probability of each category after softmax activations are applied, let Y denote the hard-coded label of the image, the lost function is computed by

$$L(y, \hat{y}) = -\sum_{i=1}^{4} y_i \times log(\hat{y}_i)$$

and the cost function is $$J = \frac{1}{m} \sum_{a=1}^{m} L(\hat{y}^{(a)}, y^{(a)})$$

3. Backward Propagation:

Backward propagation calculates the gradient of the cost function *J* with respect to the weights and biases.

To initialize the process, the gradient of the previous activation layer with respect to the softmax function is *dA[l-1] = Ŷ - Y*. The rest of the gradients are calculated using the formulas below:

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$
$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$
$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$
$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

To update the parameters W and B after each epoch:

$$W^{[l]} = W^{[l]} - learningRate \times dW^{[l]}$$

$$B^{[l]} = B^{[l]} - learningRate \times dB^{[l]}$$

4. Regularization:

One common factor that causes high variance in the neural network is overfitting. When the program runs too many epochs with a small training set, it starts to learn the unimportant details of the training images, thus failing to generalize the features to make good predictions on the test set.

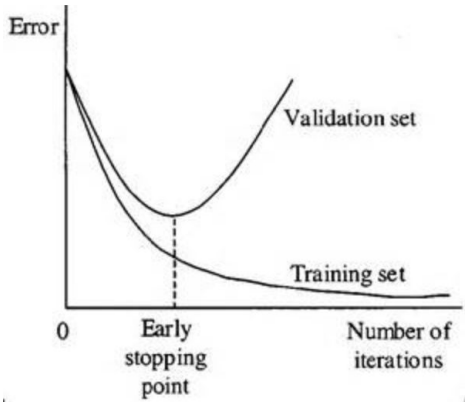


Figure 6: A graph that demonstrates how overfitting increases the error on the validation set. The image is obtained from this[3] website

L2 regularization is commonly used to prevent overfitting. When computing the cost J with regularization, we add an extra L2 regularization cost on top of the original cost function.

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

In the backward propagation with L2 regularization:

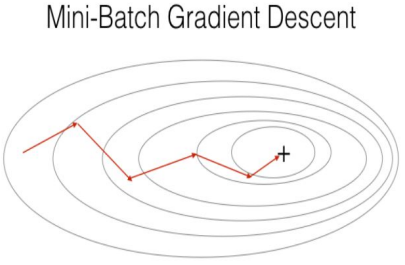
$$dW^{[l]} = \frac{1}{m}dZ^{[l]}A^{[l-1]T} + \frac{\lambda}{m}W^{[l]}$$

while dB, dZ, and dA remain unchanged.

5. Mini Batch:

Since matrix multiplication runs in $O(N^3)$, when the dataset gets bigger, the time it takes to train the model increases dramatically. Mini batch is a trade-off method that splits the training set into multiple batches, each batch is denoted as X{m}, and trains each mini-batch as if they are the entire dataset. Note that all mini-batches share the same weights and bias. The pseudocode for mini-batch gradient descent is as follow:

```
for (number of epochs)
        for (minibatch : mini batches)
                forward_propagation(minibatch);
                compute_cost(minibatch);
                backward_propagation(minibatch);
                update parameters;
        end for
```



end for *Figure 7: error oscillation with mini batch*

Using mini-batch gradient descent leads to oscillations of error since the gradient for each mini-batch is different. Therefore, it also acts as an implicit regularizer that prevents the cost from being too low, but it might also cause the training to diverge.

6. Momentum

The method is an application of exponentially weighted averages. It is an important algorithm that grants more control to the direction of gradient descent. It averages the gradient of the current batch with the gradients of the previous batches, allowing the model's error to reduce more smoothly. To implement momentum, the program takes in a new constant parameter β. In the backward propagation process, after dW and dB are computed, the program computes the velocity, VdW, and VdB, of these two gradients:

$$VdW := \beta \cdot VdW + (1-\beta) \cdot dW$$

$$VdB := \beta \cdot VdB + (1-\beta) \cdot dB$$

VdW and VdB are initialized to 0 at the beginning. The program updates the parameters W and B using the following formulas:

$$W := W - learningRate \cdot VdW$$

$$B := B - learningRate \cdot VdB$$

Result:

Unfortunately, some of the earlier models demonstrated below were not saved by serialization, and the program will not be able to reproduce the same result.

1. Accuracy

The weights and bias of the multilayer perceptron are generated through a Random object with a seed. The tables below show the accuracy of the model each time the dataset was updated.

1) The model trained using the original dataset with 100 training images and 54 testing images:

Layer Sizes	(10, 4)	
	Without Normalization	With normalization

Accuracy	Training	Testing	Training	Testing
Open(%)	0	0	100	85.7
Stairs(%)	0	0	100	35.3
Room(%)	0	0	100	57.1
Hall(%)	100	100	100	87.5
Overall(%)	**33**	**29.6**	**100**	**66.7**

2) Dataset Updated to 472 training images and 168 testing images: (only testing sets)

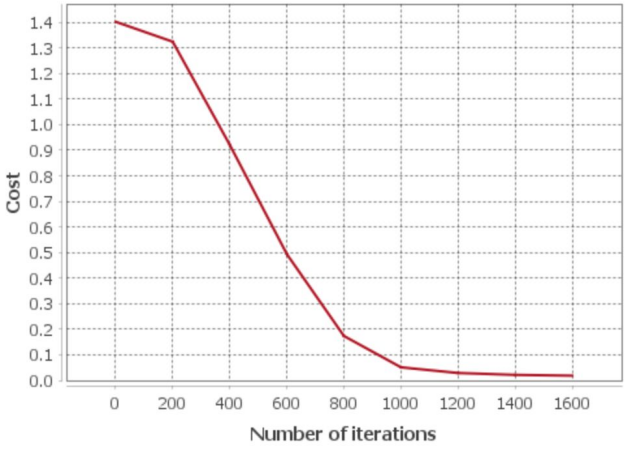
Layer Sizes	(10, 6, 4)			
Seed:	1	5	1	1
Regularization λ	No	No	0.7	1.5
Open(%)	80	70.7	78.0	80.4
Stairs(%)	65	71.1	65.4	63.5
Room(%)	65	63.4	70.7	70.7
Hall(%)	64	58.8	64.7	64.7
Overall(%)	**69**	**66.7**	**69.6**	**69.6**

3) Updated Room Dataset:

Layer Size	Open(%)	Stairs(%)	Room(%)	Hall(%)	Overall(%)
(10, 6, 4)	41.5	21.2	96.2	41.2	**70.6**

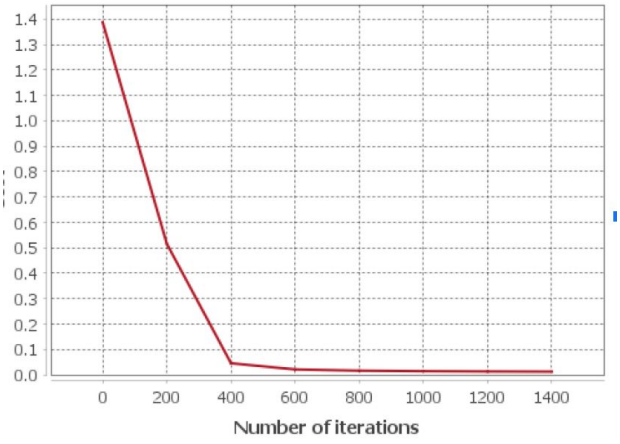
4) Updated to 1997 training images:

Layer Size	Momentum □	Open(%)	Stairs(%)	Room(%)	Hall(%)	Overall(%)
(10, 6, 4)	0.9	65.0	41.5	87.6	36.9	**56.3**



5) Reduced the size of dataset to 1200 training images and 382 testing images:

Layer Size	Momentum	Open(%)	Stairs(%)	Room(%)	Hall(%)	Overall(%)
(10, 6, 4)	0.9	91.5	44.0	71.0	40.0	**60.2**



2. Results Analysis and Drawbacks of the Design:

As demonstrated by the results above, expanding the training dataset is helpful to the accuracy in a particular range. When the number of training images increases from 100 to 472, the model's accuracy raised from 66.7 to 69.4 with regularization. However, expanding the dataset to above 500 only lowered the accuracy. One potential reason is that the image quality gets much lower as the image crawler downloads more pictures from Google and Naver, and training the model with high-quality images while making predictions on blurry and scaled images can lead to inaccurate results. To get more training data, I also tried

flipping the images horizontally and adding them to the training set. Unfortunately, it did not contribute much to raising accuracy.

The program is designed so that it loads all images into the memory and does training on them. The main benefit of the design is time-efficient since the program does not have to communicate with the hard disk every time it needs images. However, due to the heap space limitation, the program had to do more poolings to shrink the image down so that it does not run out of heap space. In most of the models demonstrated above, I set the pooling scale to 5 and applied it twice on 400 x 400 images, which means they are shrunken to 16 x 16 for training. Those small images might have lost most of the features needed for categorization, and it's suspected to be the main cause of high variances.

In terms of overfitting, L2 regularization generally raises the prediction accuracy by 0.6% to 3%.

The program is currently using 33 kernels to extract features. Adding more kernels is impossible under the current design since the heap will run out of space.

Future Work

In order to add more kernels and keep the training images larger, the design of the program must be changed to loading the images from the hard disk whenever mini batch gradient descent needs them. It might be time consuming, but it is also the only approach to raise the accuracy significantly. Besides, to make up the time needed for communicating with the hard disk, finding a library that uses multithreading on matrix operations could improve the computation efficiency with the current vectorized implementation.

Most models were trained using the grayscale mode. Users should also try training with RGB images, since as the dataset gets large, the program might be able to make better distinctions using colored images.

Last but not least, deep learning is a highly iterative process; it requires multiple trials to find out a specific set of hyperparameters that works well. Moreover, since the model is designed to be customizable, it is encouraged to keep trying more kernel combinations and sizes of hidden layers to see which work best.

References:

[1]:Kernel (image processing). (2020, June 23). Retrieved August 15, 2020, from https://en.wikipedia.org/wiki/Kernel_(image_processing)

[2]: Yiu, T. (2019, August 04). Understanding Neural Networks. Retrieved August 15, 2020, from https://towardsdatascience.com/understanding-neural-networks-19020b758230

[3]: Overfitting in Machine Learning: What It Is and How to Prevent It. (2020, May 23). Retrieved August 15, 2020, from https://elitedatascience.com/overfitting-in-machine-learning

[4]:Softmax Regression - Hyperparameter tuning, Batch Normalization and Programming Frameworks. (n.d.). Retrieved August 15, 2020, from https://www.coursera.org/lecture/deep-neural-network/softmax-regression-HRy7y

[5]: Training a softmax classifier - Hyperparameter tuning, Batch Normalization and Programming Frameworks. (n.d.). Retrieved August 15, 2020, from https://www.coursera.org/lecture/deep-neural-network/training-a-softmax-classifier-LCsCH