

Final Report: Classification of Plankton Classes

By Tae Ho Kim and Saaid Haseeb Arshad

Table of Contents

1. Project Overview
 - a. Problem Statement
 - b. Data
 - c. Overview of the Two Stages of Implementation
2. The CNN Algorithm
3. Model Optimization
4. Implementation Details
5. Conclusion

1. Project Overview

a. Problem Statement

Our project develops convolutional neural networks (CNN) to automatically classify plankton images. Given as input images of a single organism, the algorithm would predict one of 121 plankton classes that it belongs to. Using our algorithm we have achieved accuracy of 52%.

Our algorithm has the potential to help scientists gauge ocean and ecosystem health more efficiently and accurately. As planktons are a crucial part of the earth's ecosystem, scientists have manually measured and monitored plankton populations. However, the traditional method was time-consuming and lacked the scope necessary for large-scale studies. An alternative solution is an automated image classification system based on machine-learning tools like our algorithm. After training on large data images, the system can read in many images of planktons and output their species easily and reasonably accurately.

b. Data

Regarding the data of the project, we use the MNIST dataset, which consists of a set of 60,000 handwritten digits from 0 to 9, for testing of our algorithm in the first stage of implementation, as well as our plankton image dataset. The data used to classify planktons come from the Kaggle website. We have a total of about 30,000 labelled plankton images that we split into 20,000 images for training and 10,000 for testing. Example images of our plankton training set are shown below:



c. Two Stages of Implementation

We ended up having two main implementations of our convolutional neural network. The first implementation was based off of the Stanford UFLDL tutorial on CNN¹. This first implementation constructs CNN with one convolutional layer. Unfortunately, our initial implementation was limited in that it was a learning exercise, very well designed to teach us the basics of the architecture of CNNs, but could not easily extend additional convolutional and pooling layers. Adding more layers ended up being extremely difficult in this implementation because the selection of layers and their parameters was not

¹ <http://ufldl.stanford.edu/tutorial/supervised/ExerciseConvolutionalNeuralNetwork/>

modularized, rather, the whole system was hard-coded assuming only a single hidden layer. A better means of trying a more robust and complex model were needed.

The second implementation utilized the DeepLearnToolbox developed by Rasmus Berg Palm for MATLAB. We specifically used the commented version and guide provided by Chris McCormick². In this second implementation, we were able to design a more complete CNN with 2 convolutional layers and perform many experiments to find the final model structure. Our best result comes from this implementation.

2. The CNN Algorithm

a. Forward Propagation

A CNN begins with a certain number of hidden layers, where a single layer is defined as a convolution layer followed by a pooling or subsampling layer. The first layer is a convolutional layer followed by mean pooling of the convolved features. The convolutional layer applies some mapping function (like a sigmoid or rectified linear unit) to all valid points in the image $f(Wx_{r,c} + b)$ to increase the non-linear properties of the decision function and the overall network, where W and b are the learned weights from the input layer to the convolutional layer and $x_{(r,c)}$ is a subset of the image with the upper left corner at (r, c) . The size of the subset corresponds to that of the feature W . The images obtained from convolution are summarized in the subsequent pooling stage. The images are divided in disjoint regions to which we apply the mean or max operation get the pooled convolved features. In our complete model, we add another convolution/pooling layer that repeats the operation above.

Finally, we pass the images to a single densely connected layer that outputs a probability matrix consisting of estimated probability for each class given an input example image.

b. Back Propagation and Learning Parameters

For the cost of the network, we used the standard cross entropy between the predicted probability distribution over the classes for each image and the ground truth distribution:

$$-\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)],$$

where j indicates an output neuron, y is the desired value at the output neuron, n is the total number of example inputs, and a^L is the actual output value. We also tested using a second version of the cost function with a weight decay parameter λ , which penalizes weights that are too large. This effect is amplified as the value of λ increases:

² <https://chrisjmcormick.wordpress.com/2015/01/10/understanding-the-deeplearntoolbox-cnn-example/>

$$-\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^l + (1 - y_j) \ln(1 - a_j^l)] + \frac{\lambda}{2n} \sum_w w^2$$

After deriving the error for the output of the CNN using the cross entropy function, we propagated the error through all our previous layers and calculated the gradient of the weights and biases at each layers. Using stochastic gradient descent we optimized our CNN model.

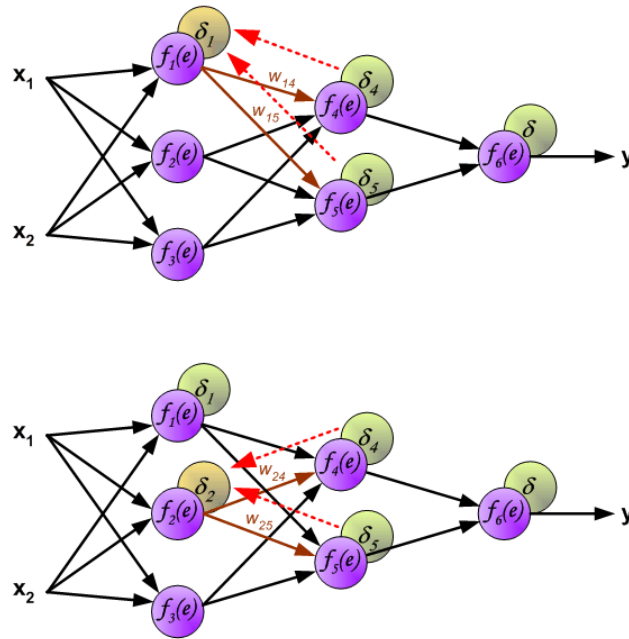


Diagram 1. Simple graphic illustration of forward and backward propagation

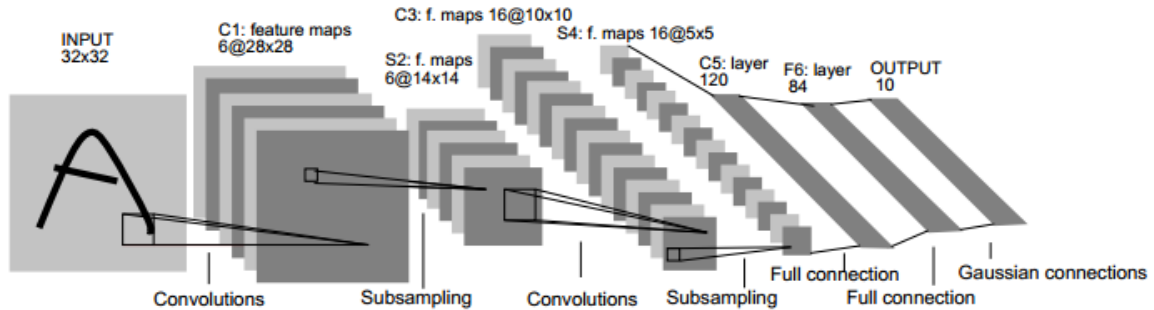


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

In summary, the algorithm can be described as follows:

1. Given an input image or set of images, convolve each one using x filters to get x feature maps for a single image.
2. Subsample each feature map using pooling (mean or max); repeat steps 1 and 2 a desired number of times.

3. Using some non-linear function on the resulting activations from step 2.
4. Implement a standard feed-forward neural network and forward propagate to get results and back-propagate using the errors calculated from the results and the expected labeled values.
5. Repeat forward and back-propagation through all layers until best results are obtained.

3. Model Optimization

Our process began with our first implementation CNN that has one convolutional layer using the UFLDL tutorial. We present the results using the MNIST data and the plankton data. In our second implementation, we use DeepLearn Toolbox to have a more complete CNN with two convolutional layers. We discuss experiments we ran to optimize our models of both implementations in turn.

As we mentioned in our milestone, a basic metric of accuracy was determined to gauge the performance of the algorithm as three main factors that determine how the algorithm runs were varied:

$$Accuracy = \frac{\sum_{i=1}^n 1\{y_i == \hat{y}_i\}}{n}$$

where n is the total number of examples, i is the index of the i -th example, y_i is the “real” test value and \hat{y}_i is the predicted value that is output from our algorithm.

Implementation 1 (1 Convolutional Layer: UFLDL Tutorial)

We were pleased to observe very high rates of accuracy on the MNIST data set, as reflected in Table 1. Since we were using starter code specifically catered towards this dataset, this was a good test to ensure our initial understanding and implementation of our convolutional neural network was sound. Training and testing on the plankton data set yielded much lower accuracy, peaking at 21% as shown in Table 2. Tables 1 and 2 show various accuracy results on the MNIST and the plankton data that we obtained as we changed model parameters. Below we illustrate our experiments in more detail with figures.

***MNIST
Dataset***

Epochs	Batch	Alpha	Mom	Weight Dec	ImageDim	numClasses	FilterDim	numFilters	poolDim	Accuracy
3	100	0.1	0.95	0	28	121	9	20	2	0.9789
3	256	0.1	0.95	0	28	121	9	20	2	0.987
3	256	0.1	0.95	0.00001	28	121	9	20	2	0.9668
3	256	0.1	0.95	0.0001	28	121	9	20	2	0.8959
3	256	0.1	0.95	0.1	28	121	9	20	2	0.3446
3	256	0.1	0.95	1000	28	121	9	20	2	0.1135
3	500	0.1	0.95	0	28	121	9	20	2	0.9577
3	1000	0.1	0.95	0	28	121	9	20	2	0.9331
3	2000	0.1	0.95	0	28	121	9	20	2	0.9077
3	5000	0.1	0.95	0	28	121	9	20	2	0.8017

Table 1. Results for the MNIST data

***Plankton
Dataset***

Epochs	Batch	Alpha	Mom	Weight Dec	ImageDim	numClasses	FilterDim	numFilters	poolDim	Accuracy
3	100	0.1	0.95	0	28	121	9	20	2	0.029371
3	500	0.1	0.95	0	28	121	9	20	2	0.123022
3	500	0.1	0.95	0	34	121	9	20	2	0.131207
3	500	0.1	0.95	0	40	121	9	20	2	0.214205
3	500	0.1	0.95	0.00001	40	121	9	20	2	0.014969
3	500	0.1	0.95	0.001	40	121	9	20	2	0.061629
3	500	0.1	0.95	0.1	40	121	9	20	2	0.029421
3	500	0.1	0.95	1000	40	121	9	20	2	0.029421
3	500	0.1	0.95	0	50	121	9	20	2	0.115929
3	1000	0.1	0.95	0	28	121	9	20	2	0.029074
3	2000	0.1	0.95	0	28	121	9	20	2	0.125
3	5000	0.1	0.95	0	28	121	9	20	2	0.117781
3	6000	0.1	0.95	0	28	121	9	20	2	0.044403
3	20000	0.1	0.95	0	28	121	9	20	2	0.034612

Table 2. Results for the plankton data

We experimented with different model parameters to find the optimal structure in our first implementation:

- 1) We first varied the input image sizes and tested for the accuracy. Looking at Figure 1, the Input Image sizes of 28, 34, 40, and 50 were tested out, yielding accuracies of 12%, 13%, 21%, and 12%, showing an apparent local optimal size of 40. Since the total image size determines the total number of input neurons, the image size can significantly impact model complexity, while also influencing how

easily the convolution and pooling layer can extract features from the images. Smaller images will make image feature extraction more difficult (less pixels per unit area to work with) and yield a simpler model (less parameters meaning a shorter length for our theta vector). The opposite is true for a larger image, so it seems we would tend for a larger image. However, given that the parameter size increases with the square of the image dimensions (30 pixels means 30^2 input neurons while 40 pixels mean 40^2 neurons, a difference of 700) increasing image dimension size too much can quickly lead to an overly complex model and thus over fitting. From the values we tested, the optimum falls somewhere around 40, though more values will be tested to see if a better optimum exists.

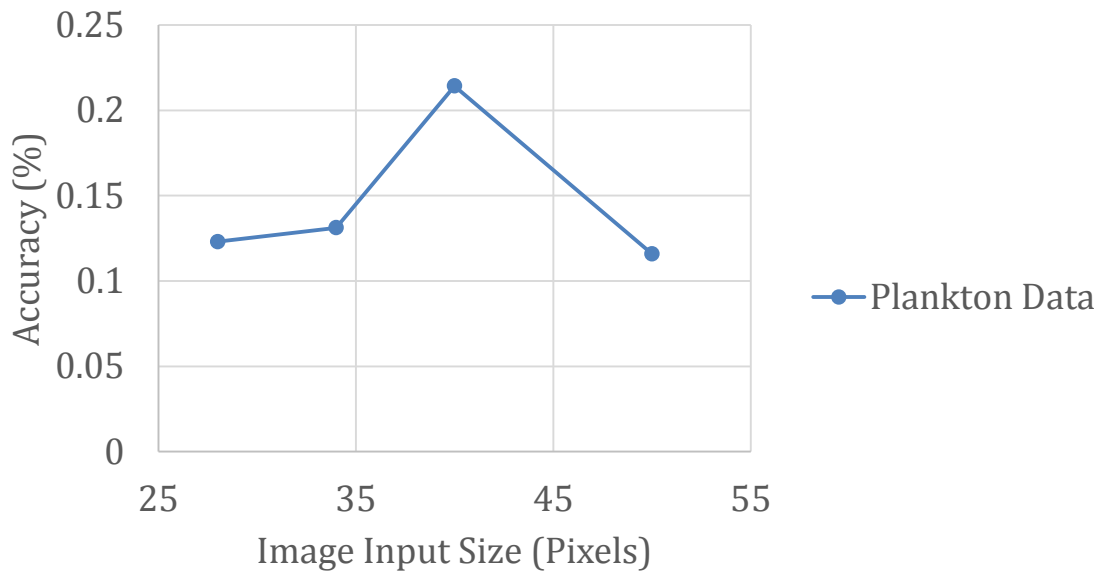


Figure 1

- 2) Minibatch determines the size of the subsamples taken from the entire training set for every iteration of the optimization's calculation of new parameter values. For example, when we set the minibatch size to be 256, the main optimization function calls on 256 random values from the training set many times until every example in the training set has been used in some combination. Interestingly, significantly increasing the minibatch size seems to significantly decrease overall accuracy for both the MNIST and plankton datasets.

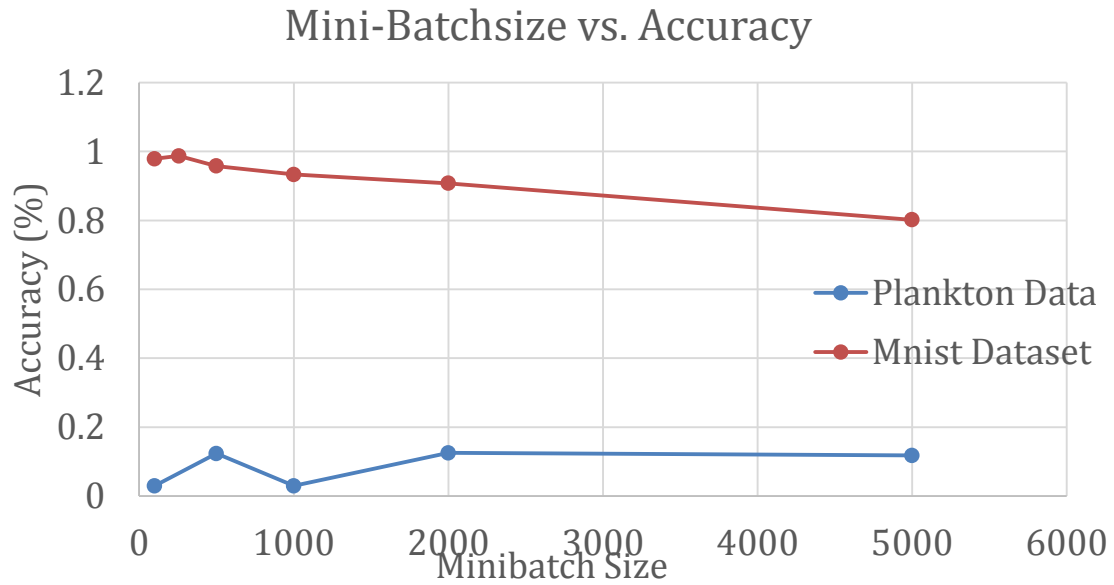


Figure 2

- 3) Finally, looking at Figures 3 and 4, the influence of the weight decay parameter lambda was as expected. Increasing lambda significantly diminished the complexity, and thus the accuracy of the model when trained on the MNIST dataset. A similar result was possible for the Plankton Image, but given that the addition of the weight decay parameter was causing it to plateau at a very low accuracy also shows the model was definitely too simple for the problem of plankton classification we are trying to solve.

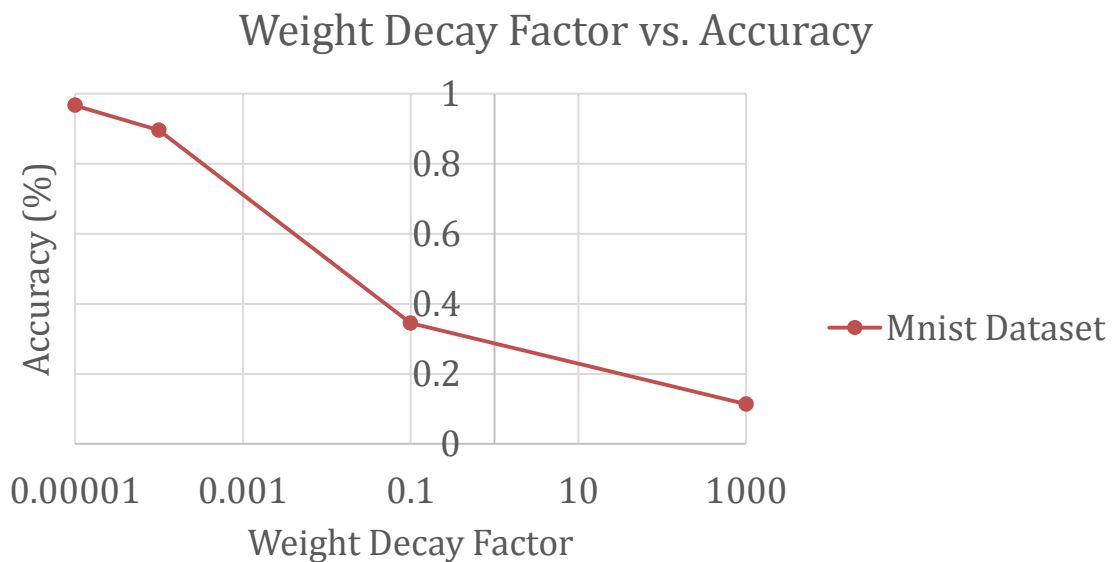


Figure 3

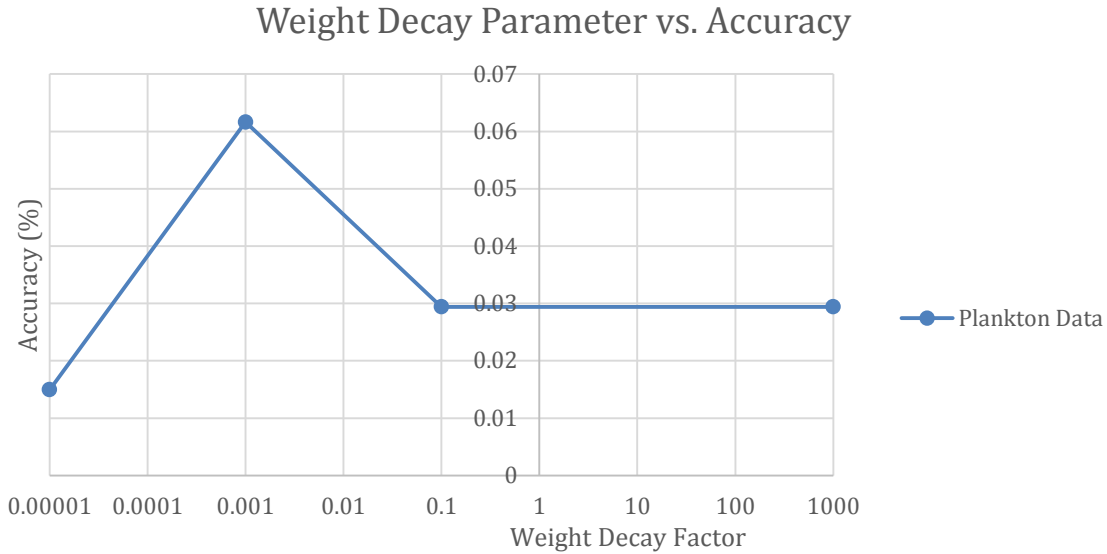


Figure 4

The overall conclusion was that a significantly more robust and complex algorithm was needed if we were to significantly improve accuracy on the plankton data set.

In this implementation, we tried several methods to improve performance. For example, a preprocessing step was added that boosted the best performance from 21 to 27% accuracy in our initial implementation. However, it was found that histogram equalization had no tangible effect in later tests with Implementation 2 and was thus scrapped as a preprocessing step. Mean thresholds and edge detection were tried as well but to no avail.

Implementation 2 (2 convolutional layers: DeepLearnToolbox)

We augmented the first model by adding another convolutional layer. In total, this model contains two sets of convolutional/mean pooling layers and one fully connected layer that classify the outputs.

First and foremost, we had to verify that the toolbox and our implementation yielded the same results with the same parameters, to ensure that the assumption that our implementation worked as well as another toolbox. We were pleased to see that our implementation got exactly the average accuracy that the DeepLearnToolbox got, averaged over 3 runs:

**Plankton Dataset:
Implementation 1**

Epochs	Batch	Alpha	Mom	ImageDim	numClasses	FilterDim	numFilters	poolDim	Accuracy
3	50	0.1	0.95	40	121	9	20	2	0.0614

Plankton Dataset:
Implementation 2

Epochs	Batch	Alpha	Mom	ImageDim	numClasses	FilterDim	numFilters	poolDim	Accuracy
3	50	0.1	0.95	40	121	9	20	2	0.0614

Table 3. Comparison of Implementation 1 and 2

Thus we assumed that all experiments run with the toolbox are reflective of how our initial implementation, Implementation 1 would perform if we could add another hidden layer and vary parameters as we do in following.

We performed the following experiments using the DeepLearn toolbox to find the optimal model structure.

- 1) We varied the number of feature maps in each of the convolutional layer keeping other parameters fixed. Table 4 shows the results from this experiment. We found that the optimum number of feature maps is 6 for the first layer and 8 for the second layer. We ran each combination of the parameters above using three different initializations and obtained the accuracy score by averaging the accuracy outcomes. The other parameters were held in the following way: number of epochs: 3, filter size: 5, mean pooling dimension: 2, and batch size: 50. Figure 5 illustrates the typical trend of accuracy score as we vary the number of feature maps in the second layer keeping the feature map count in the first layer fixed at 6. The optimum usually occurs when the feature map counts are similar for both layers.

# Feature Maps -Layer 1	# Feature Maps -Layer 2	Accuracy Score(%)
2	2	8.66
2	4	8.02
2	6	10.39
2	8	11.15
2	10	7.24
2	12	5.89
4	4	10.09
4	6	10.40
4	8	7.73
4	10	9.70
4	12	7.00
4	14	7.91
6	6	11.72
6	8	12.05
6	10	7.49
6	12	5.31
6	14	8.15
6	16	6.37
8	8	10.44
8	10	8.03
8	12	6.37
8	14	7.60
8	16	5.04
8	18	8.36

Table 4. Number of Feature Maps VS Accuracy

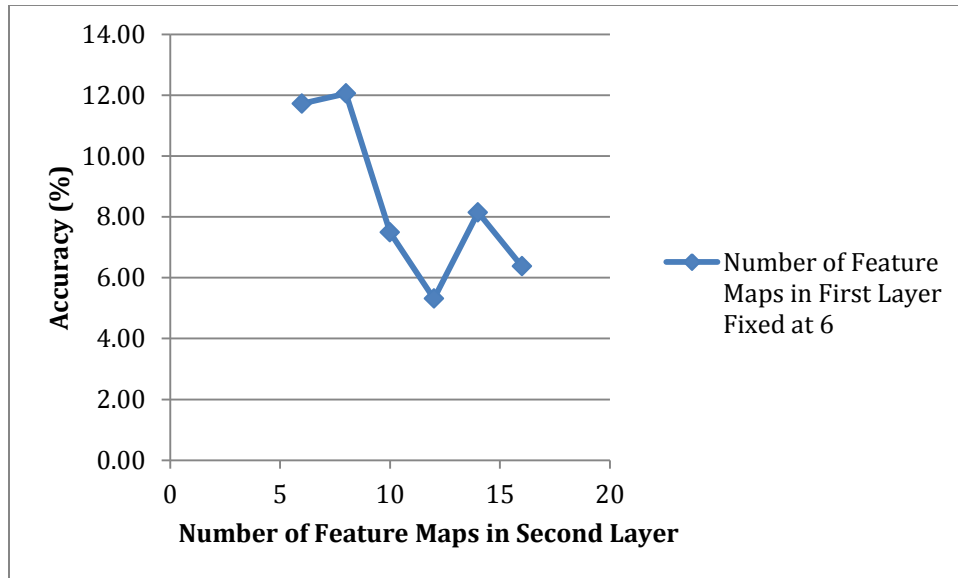


Figure 5. Number of Feature Maps VS Accuracy

- 2) We found that the optimal filter size is 5 for each of the convolutional layers (Table 5). We used three different initializations and obtained the accuracy score by averaging the accuracy outcomes. The other parameters were held in the following way: number of epochs: 3, first layer filter dimension: 6, second layer filter dimension: 8, mean pooling dimension: 2, and batch size: 50.

Filter Size - Layer 1	Filter Size - Layer 2	Accuracy Score (%)
5	5	11.99
5	9	11.94
5	13	9.36
9	5	10.20
9	9	9.02
9	13	6.48
13	5	6.48
13	9	8.98
13	13	6.59

Table 5. Filter Size VS Accuracy

- 3) We determined the optimal mean gradient step (batch size) of the model to be 50. Figure 6 shows how our model's performance substantially worsened with higher values of batch size. We ran the model for each values of the parameter using three different initializations and obtained the accuracy score by averaging the

accuracy outcomes. The other parameters were held in the following way: number of epochs: 3, filter size: 5, first layer filter dimension: 6, second layer filter dimension: 8, mean pooling dimension: 2, and batch size: 50.

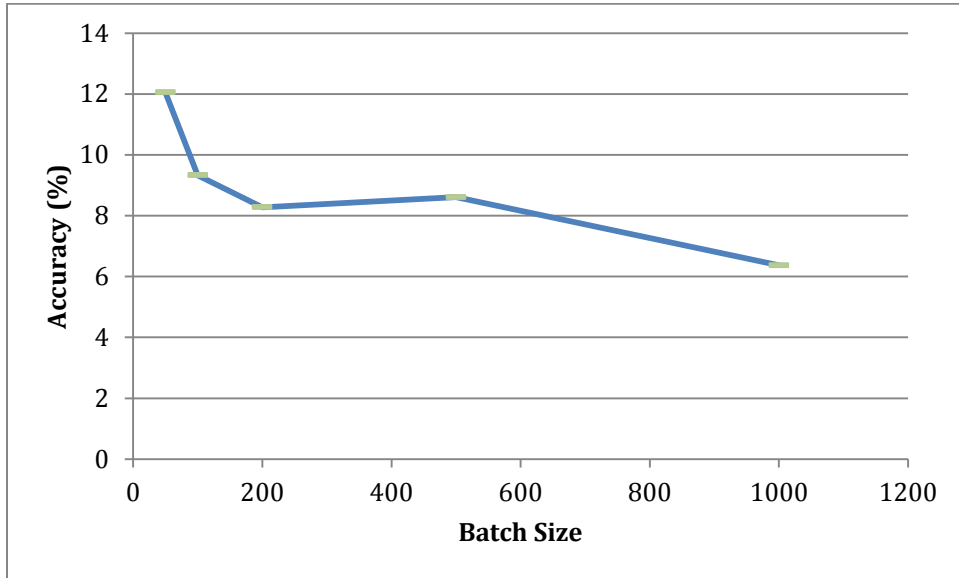


Figure 6. Batch Size VS Accuracy

- 4) Figure 7 shows our optimized model tested over various time periods. Obviously, the longer we train, the better our accuracy, but we see that after 50 epochs, the rate of improvement for every epoch decreases. The best accuracy we have achieved after 300 epochs so far has been 52%. We are currently running experiments for 500 and 1000 epochs but those will not be done till tomorrow night!

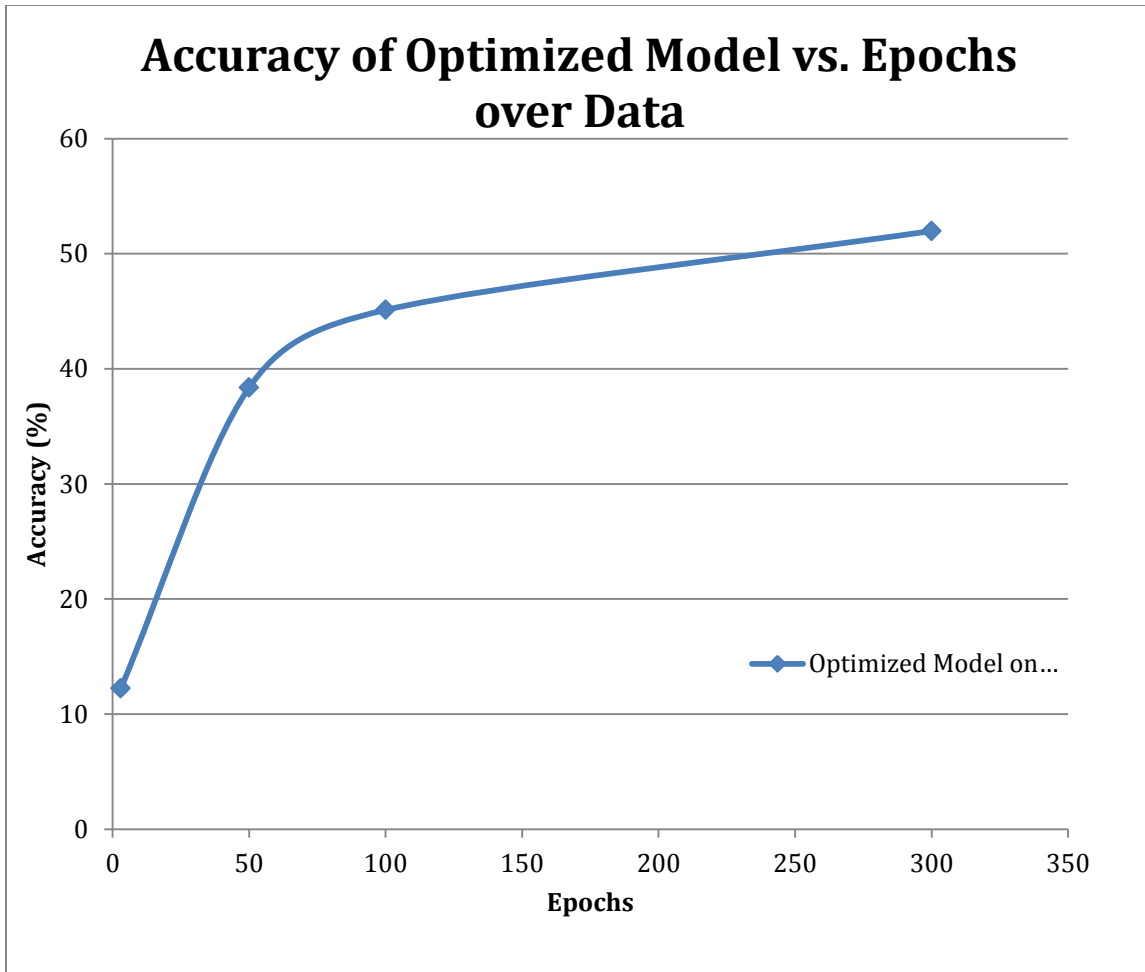


Figure 7. Number of Epochs VS Accuracy

- 5) An important factor to account for was the difference between classes in terms of the labeled examples available for each class. For example, the class with the smallest number of examples has only 9 images compared to the one with the most with 1979. It was suggested to us by Professor Torresani and TA Haris Baig that we account for this by multiplying by a weight factor $W = \frac{1}{size(class\ of\ y^{(l)})}$ in our cost function. However, in trying to implement this in our first implementation, results dropped to 0.06 % accuracy at best and implementation 2 yielded at best 1.34% accuracy.

4. Implementation Details

The Stanford tutorial of the first implementation came with a starter code where we have to implement the convolution and pooling layers, the forward and backward propagation, and the calculation of the gradients. The README file found in our uploaded code will make it apparent, but the code written by us is indicated by comment tags in capital

commented tags %SAAID AND TAE HO CODE STARTS HERE and %SAAID AND TAE HO CODE ENDS HERE. What we wrote can be assumed to be in between these two lines and everything else is externally sourced. There can be multiple sections where our code ends and begins in one source code file. Our codes can be found in the following files:

- Final_Submission
 - Code_Final_Impl1
 - cnnTrain.m
 - cnnCost.m
 - computeNumericalGradient.m
 - minFuncSGD.m
 - cnnConvolve.m
 - cnnPool.m

We had an original script to process the data from the original jpeg images into usable .mat files. This file also allowed us to do some basic processing of the data, primarily resizing and mean-thresholding, though after optimization, the only preprocessing done in this script was resizing of the images:

- Final_Submission
 - ImageHandler

However the original training images folder of jpegs has not been included because it would take up too much space and be too time consuming to move.

For the second implementation, the set of code can be found in the Code_Final_Impl2 folder in our Final Submission file. The only file we modified here was test_example_CNN, which can be found through the following path, and we have tagged the code we have written as we did in the first implementation:

- Final_Submission
 - Code_Final_Impl2
 - tests
 - test_example_CNN

5. Conclusion

In this project, we set out to implement an automated image processing algorithm that classifies plankton species given plankton images. We achieved accuracy of 52% so far. Such high accuracy attests to the strength of CNN in image classification. We plan on increasing the number of epochs and increase the accuracy for better performance.

REFERENCES

<https://chrisjmcormick.wordpress.com/2015/01/10/understanding-the-deeplearntoolbox-cnn-example/>

<http://www.kaggle.com/c/datasciencebowl/details/tutorial>

Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, *Proceedings of the IEEE*, 86(11):2278-2324, November 1998, \cite{lecun-98}.

<http://neuralnetworksanddeeplearning.com/index.html>

<http://ufldl.stanford.edu/tutorial/>