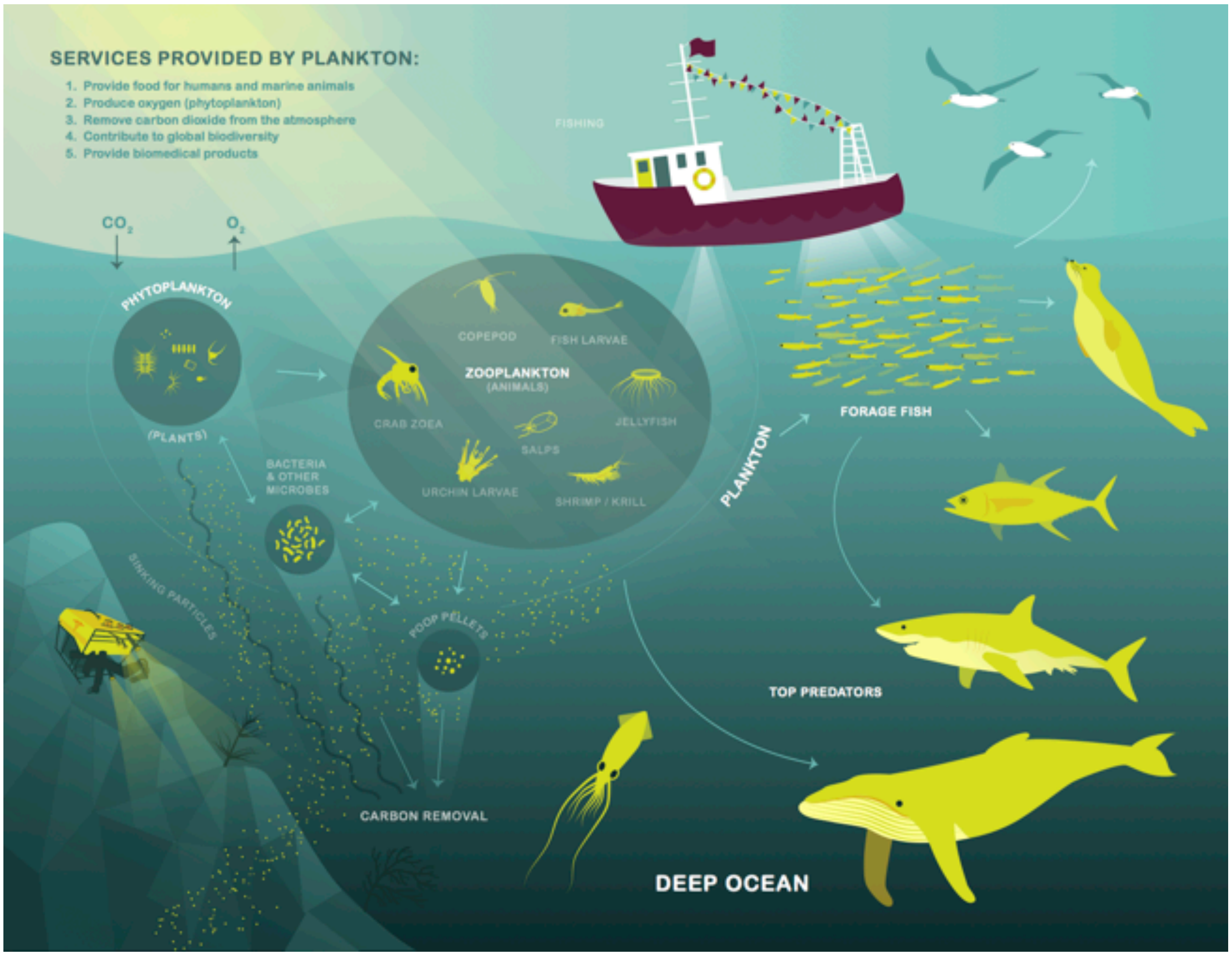
SERVICES PROVIDED BY PLANKTON:
1. Provide food for humans and marine animals
2. Produce oxygen (phytoplankton)
3. Remove carbon dioxide from the atmosphere
4. Contribute to global biodiversity
5. Provide biomedical products
FISHING
CO_2
O_2
PHYTOPLANKTON
(PLANTS)
COPEPOD
FISH LARVAE
ZOOPLANKTON (ANIMALS)
CRAB ZOEA
JELLYFISH
SALPS
URCHIN LARVAE
SHRIMP / KRILL
BACTERIA & OTHER MICROBES
PLANKTON
FORAGE FISH
SINKING PARTICLES
POOP PELLETS
TOP PREDATORS
CARBON REMOVAL
DEEP OCEAN

Milestone Report: Classification of Plankton Classes

By Tae Ho Kim and Saaid Haseeb Arshad

1. Problem and Data

Our project attempts to classify plankton images through supervised machine-learning algorithms. As mentioned in our proposal, an automated plankton classification system would help scientists gauge ocean and ecosystem health more efficiently and accurately. We mainly use convolutional neural networks (CNN), the learning algorithm that outperforms all other known image classification algorithms in the literature. Regarding the data we use, we use the MNIST dataset, which consists of a set of 60,000 handwritten digits from 0 to 9, for testing of our algorithm, as well as our plankton image dataset. One revelation since the time of the proposal has been that our initial training set of 30,000 images is the only labeled data, the other 70,000 images in our supposed test set are unlabeled, and would be impossible to label without a trained marine biologist. As such, we split the training set into 20,000 images for training and 10,000 for testing. Example images of our plankton training set are shown below:

2. Milestone Goals and Progress

In our proposal, we aimed to achieve the following goals by the milestone presentation date:

1. Size-normalize, center, and perform other processing of input images.
2. Research and evaluate the best networks for plankton recognition.
3. Make significant progress on writing a Matlab script that implements the convolutional neural networks and produces preliminary results.
4. Research the literature and other learning resources to gain better understanding of CNN and explore other algorithms that could complement CNN.

We successfully completed goals 1, 3, and 4, and we are on good track to achieve goal 2. The preprocessing of the data involved size-normalizing and centering, as well as preparation of the labels each image from the initial set provided. A basic script was written which can reproduce the dataset in a pseudo-randomized fashion, and can also adjust the ending size. At the moment, all images are resized into square images. Regarding goal number two, in the past few weeks, we examined an extensive array of resources in order to better understand CNN and implement one. Among many others, we relied on the Deep Learning Stanford Tutorial, LeCun et al. (1998), and the book by Michael Nielsen *Neural Networks and Deep Learning*. We first implemented our CNN for the handwritten digit classification problem using basic starter codes given by the Stanford tutorial. (Please refer to the Appendix for an example of our MATLAB implementation of CNN). After we successfully implemented our first CNN, we modified the codes to solve our main problem of plankton classficiation. Through reading the literature and discussing with Professor Torresani, we confirmed that CNN outperforms all other learning algorithms . Since our plankton classification problem is similar to the handwritten digit classification problem extensively researched by Lecun, we decided to follow his approach in implementing our CNN and hope to model our CNN after his implementation. We will continue to follow closely Lecun's approach and tweak our algorithm to suit our unique problem of classification for amorphously-shaped organisms.

3. Implementation of CNN

a. Forward Propagation

Our CNN has two hidden layers. The first layer is a convolutional layer followed by mean pooling of the convolved features. The convolutional layer applies the following sigmoid function to all valid points in the image $\sigma\left(\mathrm{W}\mathrm{x}_{\mathrm{r,c}} + \mathrm{b}\right)$, 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 map W(9 x 9 pixels). We have 20 different feature maps at the convolutional layer. After we convolved each image, we divided the convolved feature matrix using a pooling dimension of two-by-two pixels to create disjoint regions and took the mean of each of these regions to obtain the pooled convolved features.
We then implemented a densely connected layer into a standard softmax regression that outputs a probability matrix that estimates the 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:

$$-\frac{1}{\mathrm{n}}\sum_{x}\sum_{j}\left[y_j \ln a_j^L + (1-y_j)\ln\left(1-a_j^L\right)\right],$$

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:

$$-\frac{1}{\mathrm{n}}\sum_{x}\sum_{j}\left[y_j \ln a_j^L + (1-y_j)\ln\left(1-a_j^L\right)\right] + \frac{\lambda}{2n}\sum_{w} w^2$$

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 with a fixed momentum of 0.95 and a learning rate of 0.1, we optimized our CNN model.

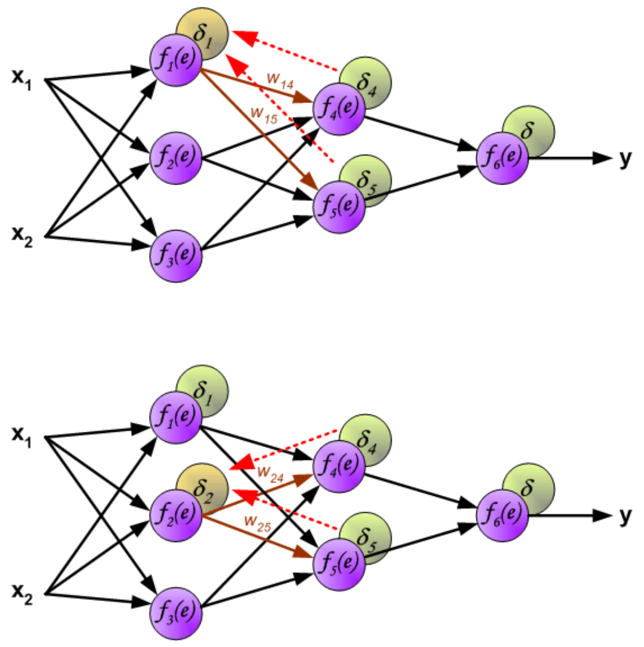

Diagram 1. Simple graphic illustration of forward and backward propagation

4. Results and Discussion

Our code was modularized according to the different layers and operations and can be found in Appendix A: Code. We tested our CNN first on the MNIST data and subsequently on our plankton data. With a basic version of the convolutional neural networks algorithm implemented, 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. Results of our training and testing of the algorithm are shown in Tables 1 and 2. The columns batch and weight decay are highlighted in maroon to indicate that they were the variables changed to test the algorithms performance. The input image size (ImageDim) was also varied for the plankton image data set. Figures 1 through 4 plot the trends in the changes in accuracy with respect to these variables.

Table 1

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 2

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

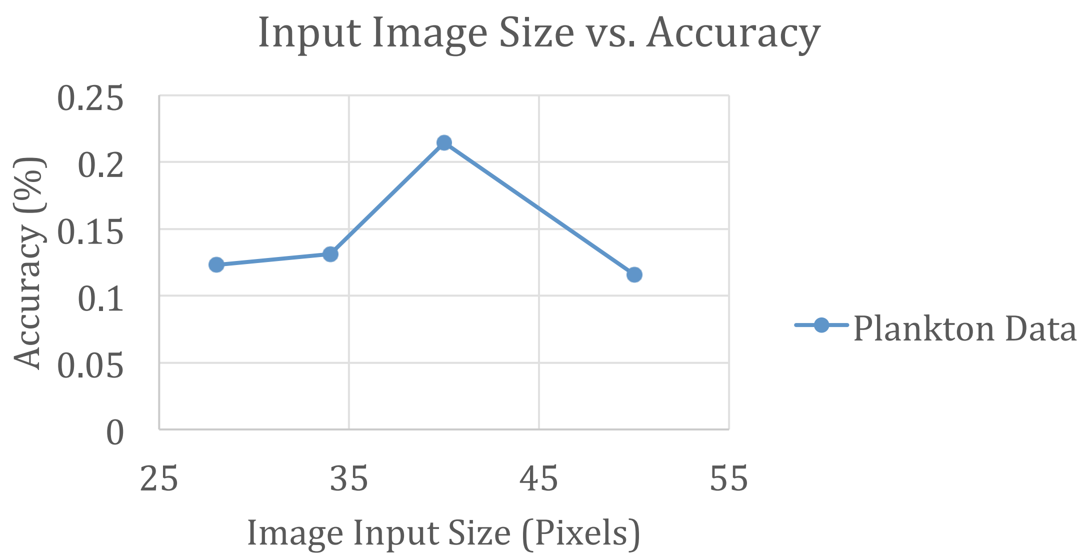

Figure 1

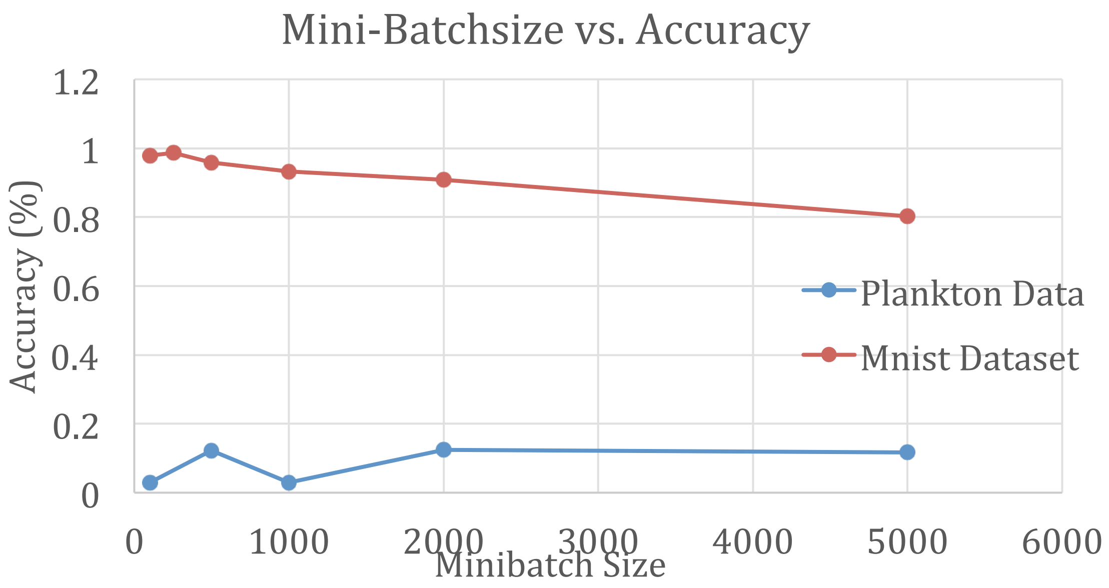

Figure 2

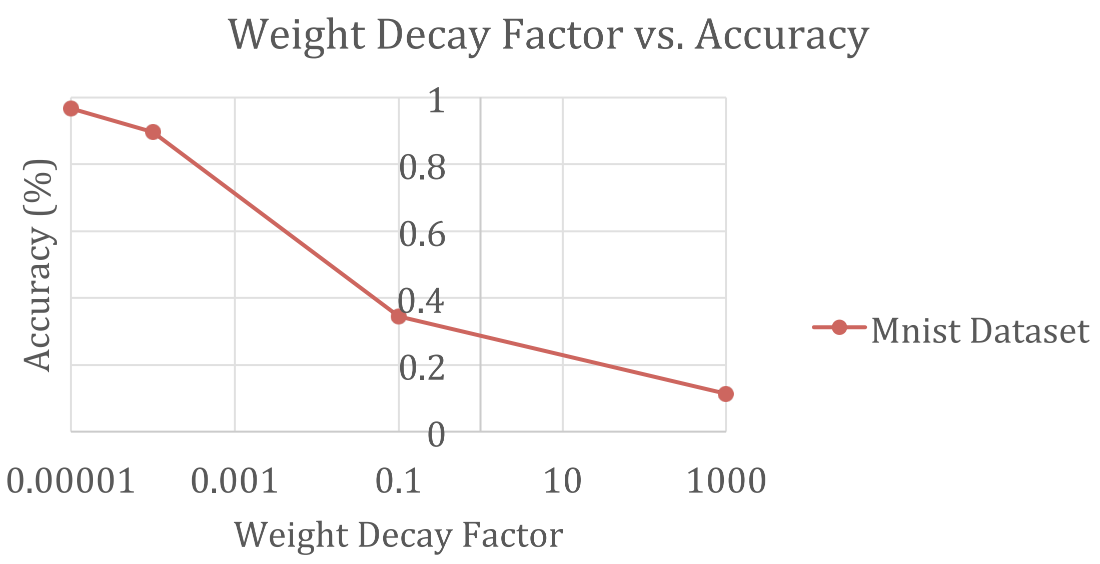

Figure 3

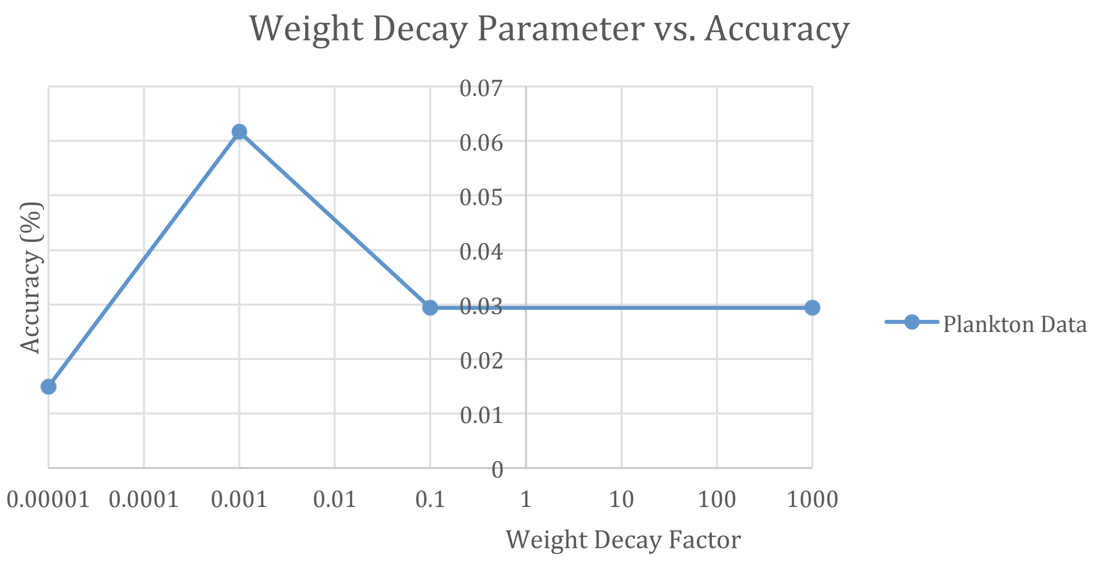

Figure 4

We were pleased to observe very high rates of accuracy on the MNIST data set, as reflected in Figure 2 and 3. 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 and Figure 1.

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.

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.

Finally, looking at Figures 3 and 4, the influence of the weight decay parameter lambda is as expected. Increasing lambda significantly diminished the complexity, and thus the accuracy of the model when trained on the MNIST dataset. A similar result is 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 is definitely too simple for the problem of plankton classification we are trying to solve.

The overall conclusion is that a significantly more robust and complex algorithm is needed if we are to significantly improve accuracy on the plankton data set. This will at a minimum involve the addition of more convolution, pooling, and densely connected hidden layers. Beyond this, some further preprocessing of the data, including mean-thresholding on the image to emphasize the slighter details in the images will help create more needed contrast in these images.

5. Future Work

We hope to work on the following tasks to improve our CNN and the performance of our model:

a. Add more densely connected layers and another set of convolutional/pooling layer.
b. Make training faster by implementing techniques such as vectorising matrix operations.
c. Experiment with different functional forms at each layer such as the rectified-linear function and the tanh hyperbolic function.
d. Experiment with different cost functions such as the minimum mean squared error.
e. Expand our training data by rotating and translating original training data images.

We will further modify our future goals based on feedback from ours peers in class, from TA Haris Baig, and from Professor Torresani.

References:

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/

Appendix A: Code

cnnTrain.m

```
%% Convolution Neural Network Exercise

%  Instructions
%  ------------
%
%  This file contains code that helps you get started in building a
single.
%  layer convolutional nerual network. In this exercise, you will only
%  need to modify cnnCost.m and cnnminFuncSGD.m. You will not need to
%  modify this file.

%%=======================================================================
=
%% STEP 0: Initialize Parameters and Load Data
%  Here we initialize some parameters used for the exercise.

% Configuration
imageDim = 40;
numClasses = 121;  % Number of classes (MNIST images fall into 10
classes)
filterDim = 9;    % Filter size for conv layer
numFilters = 20;   % Number of filters for conv layer
poolDim = 2;      % Pooling dimension, (should divide imageDim-
filterDim+1)

% Load MNIST Train
addpath ../common/;
% Mnist Data set
% images = loadMNISTImages('../common/train-images-idx3-ubyte');
% images = reshape(images,imageDim,imageDim,[]);
% labels = loadMNISTLabels('../common/train-labels-idx1-ubyte');
% labels(labels==0) = 10; % Remap 0 to 10

%Plankton Dataset
load('PlanktonTrain_imsize40');
images = trainsetX;
labels = trainsetY;

% Initialize Parameters
theta =
cnnInitParams(imageDim,filterDim,numFilters,poolDim,numClasses);

%%=======================================================================
=
%% STEP 1: Implement convNet Objective
%  Implement the function cnnCost.m.

%%=======================================================================
=
%% STEP 2: Gradient Check
%  Use the file computeNumericalGradient.m to check the gradient
%  calculation for your cnnCost.m function.  You may need to add the
%  appropriate path or copy the file to this directory.
```

```
DEBUG=false;  % set this to true to check gradient
if DEBUG
    % To speed up gradient checking, we will use a reduced network and
    % a debugging data set
    db_numFilters = 2;
    db_filterDim = 9;
    db_poolDim = 5;
    db_images = images(:,:,1:10);
    db_labels = labels(1:10);
    db_theta = cnnInitParams(imageDim,db_filterDim,db_numFilters,...
                db_poolDim,numClasses);

    [cost grad] = cnnCost(db_theta,db_images,db_labels,numClasses,...
                                db_filterDim,db_numFilters,db_poolDim);


    % Check gradients
    numGrad = computeNumericalGradient( @(x) cnnCost(x,db_images,...
                                db_labels,numClasses,db_filterDim,...
                                db_numFilters,db_poolDim), db_theta);

    % Use this to visually compare the gradients side by side
    disp([numGrad grad]);

    diff = norm(numGrad-grad)/norm(numGrad+grad);
    % Should be small. In our implementation, these values are usually
    % less than 1e-9.
    disp(diff);

    assert(diff < 1e-9,...
        'Difference too large. Check your gradient computation again');

end;

%%=====================================================================
=
%% STEP 3: Learn Parameters
%  Implement minFuncSGD.m, then train the model.

options.epochs = 3;
options.minibatch = 500;
options.alpha = 1e-1;
options.momentum = .95;
options.tol = 0.05;

opttheta = minFuncSGD(@(x,y,z) cnnCost(x,y,z,numClasses,filterDim,...
                      numFilters,poolDim),theta,images,labels,options);

%%=====================================================================
=
%% STEP 4: Test
%  Test the performance of the trained model using the MNIST test set.
Your
%  accuracy should be above 97% after 3 epochs of training
```

```
%Load Test Data for MNist
% testImages = loadMNISTImages('../common/t10k-images-idx3-ubyte');
% testImages = reshape(testImages,imageDim,imageDim,[]);
% testLabels = loadMNISTLabels('../common/t10k-labels-idx1-ubyte');
% testLabels(testLabels==0) = 10; % Remap 0 to 10

%Load/Generate Test Data for Plankton
load PlanktonTest_imsize40

testImages = testsetX;
testLabels = testsetY;

[~,cost,preds]=cnnCost(opttheta,testImages,testLabels,numClasses,...
                filterDim,numFilters,poolDim,true);

acc = sum(preds==testLabels)/length(preds);

% Accuracy should be around 97.4% after 3 epochs
fprintf('Accuracy is %f\n',acc);
```

cnnInitParams.m

```
function theta = cnnInitParams(imageDim,filterDim,numFilters,...
                                poolDim,numClasses)
% Initialize parameters for a single layer convolutional neural
% network followed by a softmax layer.
%
% Parameters:
%  imageDim   -  height/width of image
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  numClasses -  number of classes to predict
%
%
% Returns:
%  theta      -  unrolled parameter vector with initialized weights

%% Initialize parameters randomly based on layer sizes.
assert(filterDim < imageDim,'filterDim must be less that imageDim');

Wc = 1e-1*randn(filterDim,filterDim,numFilters);

outDim = imageDim - filterDim + 1; % dimension of convolved image

% assume outDim is multiple of poolDim
assert(mod(outDim,poolDim)==0,...
       'poolDim must divide imageDim - filterDim + 1');

outDim = outDim/poolDim;
hiddenSize = outDim^2*numFilters;

% we'll choose weights uniformly from the interval [-r, r]
r  = sqrt(6) / sqrt(numClasses+hiddenSize+1);
Wd = rand(numClasses, hiddenSize) * 2 * r - r;

bc = zeros(numFilters, 1);
bd = zeros(numClasses, 1);

% Convert weights and bias gradients to the vector form.
% This step will "unroll" (flatten and concatenate together) all
% your parameters into a vector, which can then be used with minFunc.
theta = [Wc(:) ; Wd(:) ; bc(:) ; bd(:)];

end
```

cnnCost.m

```
function [cost, grad, preds] =
cnnCost(theta,images,labels,numClasses,...
                                filterDim,numFilters,poolDim,pred)
% Calcualte cost and gradient for a single layer convolutional
% neural network followed by a softmax layer with cross entropy
% objective.
%
% Parameters:
%  theta      -  unrolled parameter vector
%  images     -  stores images in imageDim x imageDim x numImges
%                array
%  numClasses -  number of classes to predict
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  pred       -  boolean only forward propagate and return
%                predictions
%
%
% Returns:
%  cost       -  cross entropy cost
%  grad       -  gradient with respect to theta (if pred==False)
%  preds      -  list of predictions for each example (if pred==True)


if ~exist('pred','var')
    pred = false;
end;


imageDim = size(images,1); % height/width of image
numImages = size(images,3); % number of images


lambda = 1000;
USE_WEIGHT_DECAY = 1;

%% Reshape parameters and setup gradient matrices

% Wc is filterDim x filterDim x numFilters parameter matrix
% bc is the corresponding bias

% Wd is numClasses x hiddenSize parameter matrix where hiddenSize
% is the number of output units from the convolutional layer
% bd is corresponding bias
[Wc, Wd, bc, bd] =
cnnParamsToStack(theta,imageDim,filterDim,numFilters,...
                        poolDim,numClasses);

% Same sizes as Wc,Wd,bc,bd. Used to hold gradient w.r.t above params.
Wc_grad = zeros(size(Wc));
Wd_grad = zeros(size(Wd));
bc_grad = zeros(size(bc));
bd_grad = zeros(size(bd));
```

```
%%======================================================================
=
%% STEP 1a: Forward Propagation
%  In this step you will forward propagate the input through the
%  convolutional and subsampling (mean pooling) layers.  You will then
use
%  the responses from the convolution and pooling layer as the input to
a
%  standard softmax layer.

%% Convolutional Layer
%  For each image and each filter, convolve the image with the filter,
add
%  the bias and apply the sigmoid nonlinearity.  Then subsample the
%  convolved activations with mean pooling.  Store the results of the
%  convolution in activations and the results of the pooling in
%  activationsPooled.  You will need to save the convolved activations
for
%  backpropagation.
convDim = imageDim-filterDim+1; % dimension of convolved output
outputDim = (convDim)/poolDim; % dimension of subsampled output

% convDim x convDim x numFilters x numImages tensor for storing
activations
activations = zeros(convDim,convDim,numFilters,numImages);

% outputDim x outputDim x numFilters x numImages tensor for storing
% subsampled activations
activationsPooled = zeros(outputDim,outputDim,numFilters,numImages);
%Convolving Code written by Saaid H. Arshad
%Begun February 11, 2015, 11:22 pm
%%% YOUR CODE HERE %%%
activations = cnnConvolve(filterDim,numFilters,images,Wc,bc);
activationsPooled = cnnPool(poolDim,activations);



% Reshape activations into 2-d matrix, hiddenSize x numImages,
% for Softmax layer
activationsPooled = reshape(activationsPooled,[],numImages);

%% Softmax Layer
%  Forward propagate the pooled activations calculated above into a
%  standard softmax layer. For your convenience we have reshaped
%  activationPooled into a hiddenSize x numImages matrix.  Store the
%  results in probs.

% numClasses x numImages for storing probability that each image
belongs to
% each class.
probs = zeros(numClasses,numImages);

%Softmax Code written by Saaid H. Arshad and Tae Ho Kim
%Begun February 12, 2015, 11:22 am
%%% YOUR CODE HERE %%%
```

```
% z = (Wd * activationsPooled);
% z = bsxfun(@plus,z,bd);
% expz = exp(z);
% denoms = sum(expz);
% for i = 1:numImages
%
%     probs(:,i) = expz(:,i)/denoms(i);
%
% end

%------------------------------------------------------
activationsSoftmax = Wd * activationsPooled + repmat(bd, 1, numImages);
activationsSoftmax = bsxfun(@minus, activationsSoftmax,
max(activationsSoftmax));
activationsSoftmax = exp(activationsSoftmax);
probs = bsxfun(@rdivide, activationsSoftmax, sum(activationsSoftmax));

%%=======================================================================
=
%% STEP 1b: Calculate Cost
%  In this step you will use the labels given as input and the probs
%  calculate above to evaluate the cross entropy objective.  Store your
%  results in cost.

cost = 0; % save objective into cost

%Cost Calc. Code written by Saaid H. Arshad and Tae Ho Kim
%Begun February 12, 2015, 6:52 pm
%%% YOUR CODE HERE %%%
% groundTruth = full(sparse(labels, 1:numImages, 1));
% %
% cost = -mean(sum((groundTruth*log(probs') - (1-groundTruth)*log(1-
probs')),1));
onehotLabels = zeros(size(activationsSoftmax));
labelIndex = sub2ind(size(activationsSoftmax), labels', 1:numImages);
onehotLabels(labelIndex) = 1;
cost = -sum(sum(onehotLabels .* log(probs)));

if USE_WEIGHT_DECAY
    lambda = .5 * lambda * (sum(Wd(:) .^ 2) + sum(Wc(:) .^ 2));
else
    lambda = 0;
end

cost = cost / numImages + lambda;

% Makes predictions given probs and returns without backproagating
errors.
if pred
    [~,preds] = max(probs,[],1);
    preds = preds';
    grad = 0;
    return;
end;

%%=======================================================================
=
%% STEP 1c: Backpropagation
%  Backpropagate errors through the softmax and
convolutional/subsampling
%  layers.  Store the errors for the next step to calculate the
gradient.
%  Backpropagating the error w.r.t the softmax layer is as usual.  To
%  backpropagate through the pooling layer, you will need to upsample
the
%  error with respect to the pooling layer for each filter and each
image.
%  Use the kron function and a matrix of ones to do this upsampling
%  quickly.
% Code written by Saaid H. Arshad and Tae Ho Kim
%%% YOUR CODE HERE %%%
% delta_init = -sum(groundTruth - probs,2);
% delta_softmax = (Wd' * delta_init) * (1./z).*(1 - 1./z);
% delta_softmax = (Wd' * delta_init) * (1./(1 + exp(-(Wd *
activationsPooled + bd)))*(1 - 1./(1 + exp(-(Wd * activationsPooled +
bd)))));
%delta_pool = (1/poolDim^2) * kron(delta,ones(poolDim));

errorsSoftmax = probs - onehotLabels;
errorsSoftmax = errorsSoftmax / numImages;
errorsPooled = Wd' * errorsSoftmax;
errorsPooled = reshape(errorsPooled, [], outputDim, numFilters,
numImages);
errorsPooling = zeros(convDim, convDim, numFilters, numImages);
unpoolingFilter = ones(poolDim);
poolArea = poolDim ^ 2;
unpoolingFilter = unpoolingFilter / poolArea;

for imageNum = 1:numImages
    % for imageNum = 1:numImages
    for filterNum = 1:numFilters
        e = errorsPooled(:, :, filterNum, imageNum);
        errorsPooling(:, :, filterNum, imageNum) = kron(e,
unpoolingFilter);

        %         errorsPooling(:, :, filterNum, imageNum) =
kron(errorsPooled(:, :, filterNum, imageNum), unpoolingFilter);
    end
end

errorsConvolution = errorsPooling .* activations .* (1 - activations);
%%======================================================================
=
%% STEP 1d: Gradient Calculation
%  After backpropagating the errors above, we can use them to calculate
the
%  gradient with respect to all the parameters.  The gradient w.r.t the
%  softmax layer is calculated as usual.  To calculate the gradient
w.r.t.
%  a filter in the convolutional layer, convolve the backpropagated
error
%  for that filter with each image and aggregate over images.

%Code written by Saaid H. Arshad and Tae Ho Kim
```

```
%%% YOUR CODE HERE %%%
% Wd_grad = gradient;
Wd_grad = errorsSoftmax * activationsPooled';

if USE_WEIGHT_DECAY
    Wd_grad = Wd_grad + lambda * Wd;
end

bd_grad = sum(errorsSoftmax, 2);


for filterNum = 1 : numFilters
%     e = errorsConvolution(:, :, filterNum, :);
        e = errorsPooling(:, :, filterNum, :);
    bc_grad(filterNum) = sum(e(:));
end

for filterNum = 1 : numFilters
    % for filterNum = 1 : numFilters
    for imageNum = 1 : numImages
        e = errorsConvolution(:, :, filterNum, imageNum);
        %          e = errorsPooling(:, :, filterNum, imageNum);
        errorsConvolution(:, :, filterNum, imageNum) = rot90(e, 2);

        %          errorsPooling(:, :, filterNum, imageNum) =
rot90(errorsPooling(:, :, filterNum, imageNum), 2);
    end
end


for filterNum = 1 : numFilters
    Wc_gradFilter = zeros(size(Wc_grad, 1), size(Wc_grad, 2));
    %     parfor imageNum = 1 : numImages
    for imageNum = 1 : numImages
        %                  image = images(:, :, imageNum);
        %                  error = errorsPooling(:, :, filterNum,
imageNum);
        %          %          Wc_grad(:, :, filterNum) = Wc_grad(:, :,
filterNum) + conv2(image, error, 'valid');
        %          Wc_gradFilter(:, :, imageNum) = conv2(image, error,
'valid');
        %          Wc_gradFilter(:, :, imageNum) = conv2(images(:, :,
imageNum), errorsPooling(:, :, filterNum, imageNum), 'valid');

        Wc_gradFilter = Wc_gradFilter + conv2(images(:, :, imageNum),
errorsConvolution(:, :, filterNum, imageNum), 'valid');
        %          Wc_gradFilter = Wc_gradFilter + conv2(image, error,
'valid');
    end
    %     Wc_grad(:, :, filterNum) = sum(Wc_gradFilter, 3) / numImages
+ regularization;
    Wc_grad(:, :, filterNum) = Wc_gradFilter;
end

if USE_WEIGHT_DECAY
    Wc_grad = Wc_grad + lambda * Wc;
```

```
end


%% Unroll gradient into grad vector for minFunc
grad = [Wc_grad(:) ; Wd_grad(:) ; bc_grad(:) ; bd_grad(:)];

end
```

cnnConvolve.m

```
function convolvedFeatures = cnnConvolve(filterDim, numFilters, images,
W, b)
%cnnConvolve Returns the convolution of the features given by W and b
with
%the given images
%
% Parameters:
%  filterDim - filter (feature) dimension
%  numFilters - number of feature maps
%  images - large images to convolve with, matrix in the form
%           images(r, c, image number)
%  W, b - W, b for features from the sparse autoencoder
%         W is of shape (filterDim,filterDim,numFilters)
%         b is of shape (numFilters,1)
%
% Returns:
%  convolvedFeatures - matrix of convolved features in the form
%                      convolvedFeatures(imageRow, imageCol,
featureNum, imageNum)

numImages = size(images, 3);
imageDim = size(images, 1);
convDim = imageDim - filterDim + 1;

convolvedFeatures = zeros(convDim, convDim, numFilters, numImages);

% Instructions:
%   Convolve every filter with every image here to produce the
%   (imageDim - filterDim + 1) x (imageDim - filterDim + 1) x
numFeatures x numImages
%   matrix convolvedFeatures, such that
%   convolvedFeatures(imageRow, imageCol, featureNum, imageNum) is the
%   value of the convolved featureNum feature for the imageNum image
over
%   the region (imageRow, imageCol) to (imageRow + filterDim - 1,
imageCol + filterDim - 1)
%
% Expected running times:
%   Convolving with 100 images should take less than 30 seconds
%   Convolving with 5000 images should take around 2 minutes
%   (So to save time when testing, you should convolve with less
images, as
%   described earlier)


for imageNum = 1:numImages
```

```
  for filterNum = 1:numFilters

    % convolution of image with feature matrix
    convolvedImage = zeros(convDim, convDim);

    % Obtain the feature (filterDim x filterDim) needed during the
convolution

    filter = zeros(filterDim);
    filter = W(:,:,filterNum);

    % Flip the feature matrix because of the definition of convolution,
as explained later
    filter = rot90(squeeze(filter),2);

    % Obtain the image
    im = squeeze(images(:, :, imageNum));

    % Convolve "filter" with "im", adding the result to convolvedImage
    % be sure to do a 'valid' convolution

    %Convolving Code written by Saaid H. Arshad and Tae Ho Kim
    %Begun February 11, 2015, 5:40 pm

    convolvedImage = convolvedImage + conv2(im,filter,'valid');


    % Add the bias unit

    convolvedImage = convolvedImage + b(filterNum);

    % Then, apply the sigmoid function to get the hidden activation

    convolvedImage = 1./(1 + exp(-convolvedImage));


    convolvedFeatures(:, :, filterNum, imageNum) = convolvedImage;
  end
end


end
```

cnnParamsToStack.m

```
function [Wc, Wd, bc, bd] =
cnnParamsToStack(theta,imageDim,filterDim,...
                                 numFilters,poolDim,numClasses)
% Converts unrolled parameters for a single layer convolutional neural
% network followed by a softmax layer into structured weight
% tensors/matrices and corresponding biases
%
% Parameters:
%  theta      -  unrolled parameter vectore
%  imageDim   -  height/width of image
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  numClasses -  number of classes to predict
%
%
% Returns:
%  Wc      -  filterDim x filterDim x numFilters parameter matrix
%  Wd      -  numClasses x hiddenSize parameter matrix, hiddenSize is
%             calculated as numFilters*((imageDim-
filterDim+1)/poolDim)^2
%  bc      -  bias for convolution layer of size numFilters x 1
%  bd      -  bias for dense layer of size hiddenSize x 1

outDim = (imageDim - filterDim + 1)/poolDim;
hiddenSize = outDim^2*numFilters;

%% Reshape theta
indS = 1;
indE = filterDim^2*numFilters;
Wc = reshape(theta(indS:indE),filterDim,filterDim,numFilters);
indS = indE+1;
indE = indE+hiddenSize*numClasses;
Wd = reshape(theta(indS:indE),numClasses,hiddenSize);
indS = indE+1;
indE = indE+numFilters;
bc = theta(indS:indE);
bd = theta(indE+1:end);


end
```

sigmoid.m

```
function [ output ] = sigmoid( input )

output = 1./(1+exp(-input));
```

cnnPool.m

```
function pooledFeatures = cnnPool(poolDim, convolvedFeatures)
%cnnPool Pools the given convolved features
%
% Parameters:
%  poolDim - dimension of pooling region
%  convolvedFeatures - convolved features to pool (as given by
cnnConvolve)
%                      convolvedFeatures(imageRow, imageCol,
featureNum, imageNum)
%
% Returns:
%  pooledFeatures - matrix of pooled features in the form
%                   pooledFeatures(poolRow, poolCol, featureNum,
imageNum)
%

numImages = size(convolvedFeatures, 4);
numFilters = size(convolvedFeatures, 3);
convolvedDim = size(convolvedFeatures, 1);

pooledFeatures = zeros(convolvedDim / poolDim, ...
        convolvedDim / poolDim, numFilters, numImages);

% Instructions:
%   Now pool the convolved features in regions of poolDim x poolDim,
%   to obtain the
%   (convolvedDim/poolDim) x (convolvedDim/poolDim) x numFeatures x
numImages
%   matrix pooledFeatures, such that
%   pooledFeatures(poolRow, poolCol, featureNum, imageNum) is the
%   value of the featureNum feature for the imageNum image pooled over
the
%   corresponding (poolRow, poolCol) pooling region.
%
%   Use mean pooling here.

%Mean Pooling Code written by Saaid H. Arshad
%Begun February 11, 2015, 5:40 pm

for imageNum = 1:numImages
  for filterNum = 1:numFilters

      currFeat = convolvedFeatures(:,:,filterNum,imageNum);
      currFeat = conv2(currFeat,ones(poolDim),'valid')/poolDim^2;

%       for i = 0:size(pooledFeatures,1)-1
%           for j = 0:size(pooledFeatures,2)-1
%
%               pooledFeatures(i+1,j+1,filterNum,imageNum) =
currFeat(1+i*poolDim,1 + j*poolDim);
%
%
%           end
%       end
```

```
%%% YOUR CODE HERE %%%
%Written by Saaid and Tae Ho 2/15/2015

extractrow = (1:poolDim:poolDim * (size(pooledFeatures,1)));
extractcol = (1:poolDim:poolDim * (size(pooledFeatures,2)));
pooledFeatures(:,:,filterNum,imageNum) =
currFeat(extractrow,extractcol);

  end
end
end
```

minFuncSGD.m

```
function [opttheta] = minFuncSGD(funObj,theta,data,labels,...
                        options)
% Runs stochastic gradient descent with momentum to optimize the
% parameters for the given objective.
%
% Parameters:
%  funObj     -  function handle which accepts as input theta,
%                data, labels and returns cost and gradient w.r.t
%                to theta.
%  theta      -  unrolled parameter vector
%  data       -  stores data in m x n x numExamples tensor
%  labels     -  corresponding labels in numExamples x 1 vector
%  options    -  struct to store specific options for optimization
%
% Returns:
%  opttheta   -  optimized parameter vector
%
% Options (* required)
%  epochs*     - number of epochs through data
%  alpha*      - initial learning rate
%  minibatch*  - size of minibatch
%  momentum    - momentum constant, defualts to 0.9


%%=====================================================================
=
%% Setup
assert(all(isfield(options,{'epochs','alpha','minibatch'})),...
        'Some options not defined');
if ~isfield(options,'momentum')
    options.momentum = 0.9;
end;
tol = options.tol;
epochs = options.epochs;
alpha = options.alpha;
minibatch = options.minibatch;
m = length(labels); % training set size
% Setup for momentum
mom = 0.5;
momIncrease = 20;
velocity = zeros(size(theta));

costprev = 0;
thetaprev = 0;
```

```
%%=======================================================================
=
%% SGD loop
it = 0;
for e = 1:epochs

    % randomly permute indices of data for quick minibatch sampling
    rp = randperm(m);

    for s=1:minibatch:(m-minibatch+1)
        it = it + 1;

        % increase momentum after momIncrease iterations
        if it == momIncrease
            mom = options.momentum;
        end;

        % get next randomly selected minibatch
        mb_data = data(:,:,rp(s:s+minibatch-1));
        mb_labels = labels(rp(s:s+minibatch-1));

        % evaluate the objective function on the next minibatch
        [cost grad] = funObj(theta,mb_data,mb_labels);

        % Instructions: Add in the weighted velocity vector to the
        % gradient evaluated above scaled by the learning rate.
        % Then update the current weights theta according to the
        % sgd update rule

        %Code written by Saaid H. Arshad and Tae Ho Kim
        %%% YOUR CODE HERE %%%

        velocity = mom * velocity + alpha * grad; % add the velocity
vector to the gradient
        theta = theta - velocity; %update the unrolled parameter vector


%          abs(costprev - cost)
%          abs(norm(thetaprev) - norm(theta))
%          if abs(costprev-cost) < tol
%              break
%          end

%          if abs((norm(thetaprev) - norm(theta))) < tol
%              break
%          end
%



%          costprev = cost;
%          thetaprev = theta;



        fprintf('Epoch %d: Cost on iteration %d is %f\n',e,it,cost);
    end;

    % aneal learning rate by factor of two after each epoch
    alpha = alpha/2.0;

end;

opttheta = theta;

end
```

computeNumericalGradient.m

```
function numgrad = computeNumericalGradient(J, theta)
% numgrad = computeNumericalGradient(J, theta)
% theta: a vector of parameters
% J: a function that outputs a real-number. Calling y = J(theta) will
return the
% function value at theta.

% Initialize numgrad with zeros
numgrad = zeros(size(theta));

%% ---------- YOUR CODE HERE --------------------------------------
% Instructions:
% Implement numerical gradient checking, and return the result in
numgrad.
% (See Section 2.3 of the lecture notes.)
% You should write code so that numgrad(i) is (the numerical
approximation to) the
% partial derivative of J with respect to the i-th input argument,
evaluated at theta.
% I.e., numgrad(i) should be the (approximately) the partial derivative
of J with
% respect to theta(i).
%
% Hint: You will probably want to compute the elements of numgrad one
at a time.

epsilon = 1e-4;

for i =1:length(numgrad)
    oldT = theta(i);
    theta(i)=oldT+epsilon;
    pos = J(theta);
    theta(i)=oldT-epsilon;
    neg = J(theta);
    numgrad(i) = (pos-neg)/(2*epsilon);
    theta(i)=oldT;
    if mod(i,100)==0
       fprintf('Done with %d\n',i);
    end;
end;

% eps = 1e-4;
% n = size(numgrad);
% I = eye(n);
% for i = 1:size(numgrad)
%     eps_vec = I(:,i) * eps;
%     numgrad(i) = (J(theta + eps_vec) - J(theta - eps_vec)) / (2 *
eps);
% end






%% -----------------------------------------------------------------
end
```