PSYCH 734/CSE 734

Assignment 3

Sara
001143947
jamils2@mcmaster.ca
12/19/2016

Introduction

This assignment examines and implements the training of a 2-layer back-propagation neural network on the MNIST dataset. The data consists of 70,000 correctly labelled images of hand-drawn digits, of which 60,000 are training patterns and 10,000 are test patterns. Each pattern is a 28x28 pixel image that has been reshaped into a 784 input vector to the neural network. Some examples of input from the MNIST data set are shown below.

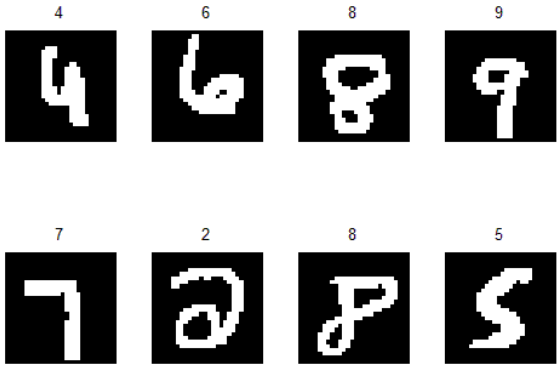

Figure 1: MNIST dataset examples with target values

In order to train the artificial neural network, backpropagation was used in conjunction with an optimization method such as gradient descent. The 2-layer network used in this implementation is represented in the figure below.

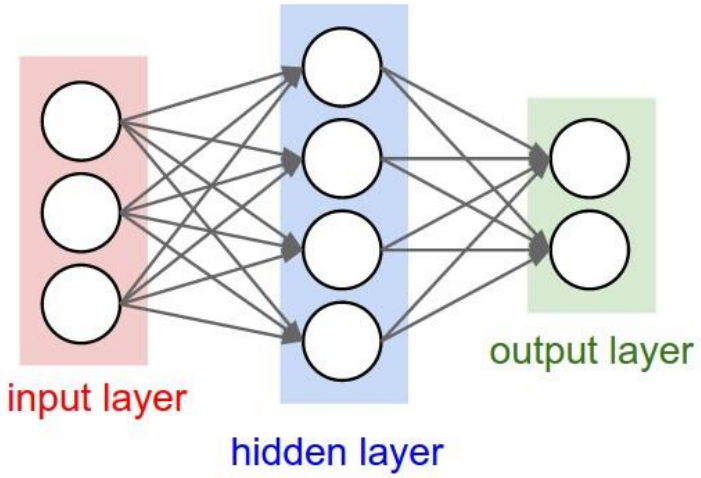

Figure 2: 2-layer neural network

In this example, the input layer consists of the 784 image input vector, the hidden layer consists of 20 units, and the output layer consists of 10 units corresponding to the predicted digits from 0 to 9. The input and hidden layers contain an additional bias unit.

The training algorithm consists of a forward and backward propagation phase followed by the weight update. An input pattern is presented to the network and propagates forward through the network to the output layer. A non-linear, differentiable sigmoid activation function was used at each unit in the network. The output layer is then compared to the desired output and an error value is calculated for each unit. The delta error values are then propagated backwards until each unit has an associated error value in order to compute the loss function. Backpropagation uses these error values to calculate the gradient of the loss function with respect to the weights of the network.

This paper will examine the results of using batch learning vs. online learning in the learning procedure. The performance of the two methods will be evaluated by calculating the accuracy of classification of the training and test sets. The weights learned by the hidden units in the network will also be examined and several different types of modifications to the network will be discussed.

Methods

The learning procedure in the backpropagation algorithm can be modified to use batch learning or online learning. With batch learning, the gradients are accumulated over the entire training set and the weight updates are made after each complete pass through the training set. With online learning (a.k.a. stochastic gradient descent), the weight update occurs after each individual training pattern, and thus the gradients are not accumulated over the entire training set.

To test the batch learning backpropagation procedure, several runs were performed in order to find an adequate learning rate parameter to use in the training process. The learning rate used in the batch learning example was 0.75, which was able to reach a low mean squared error rate within 200 runs of the training procedure or less. The following figures show the error rate and the gradient size as a function of the number of runs through the training set.

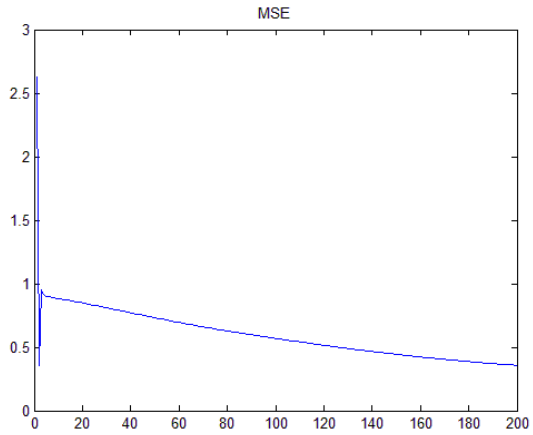

Figure 3: MSE - batch learning trial

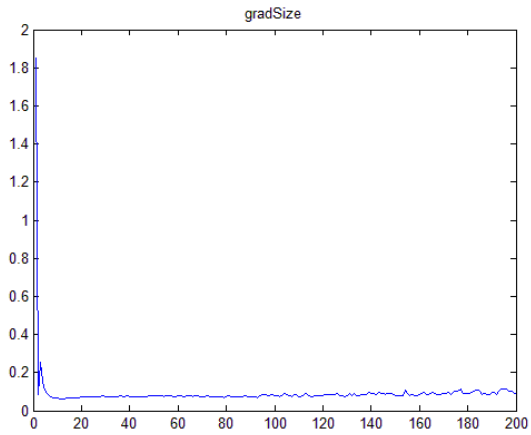

Figure 4: Gradient size - batch learning trial

Also, another indication that the network has converged is the change in error. If the change in the MSE error values decreases to zero and the (i.e. the error plateaus), this means that the error is unlikely to decrease further and would provide a good stopping criterion.

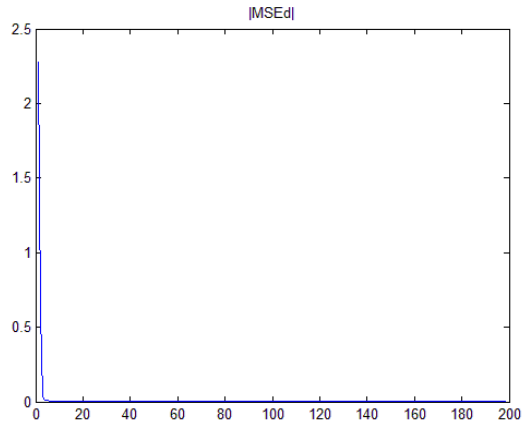

Figure 5: Change in MSE - batch learning

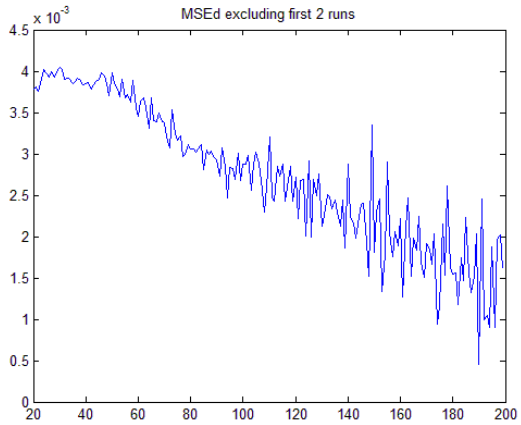

Figure 6: Zooming in on |MSEd| plot

Similarly, in order to test the online learning procedure, several runs were required to find an adequate learning rate that was able to converge to a low accuracy within 200 runs. After several tests, it was found that a learning rate less than 0.1 would be adequate, and generally smaller values of the learning parameter seemed to converge faster with low error. In the following example, the learning rate parameter was set to 0.01 and the error and gradient size are shown as follows.

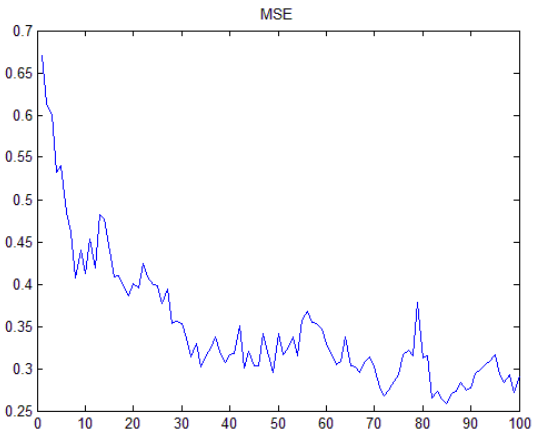

Figure 7: MSE – online learning trial

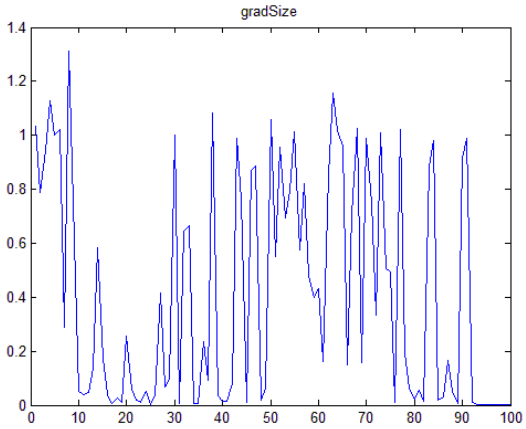

Figure 8: Gradient size – online learning trial

Using the stochastic gradient descent method, the mean squared error varies a lot more than the batch gradient descent method, as expected. Therefore, it is better to examine the average of 10 trials to get an estimate of the error. As shown above, the gradient size varies greatly and was therefore not used in the selection of the stopping criterion. The figures below show the averaged MSE as well as the averaged change in MSE for the online learning trial.

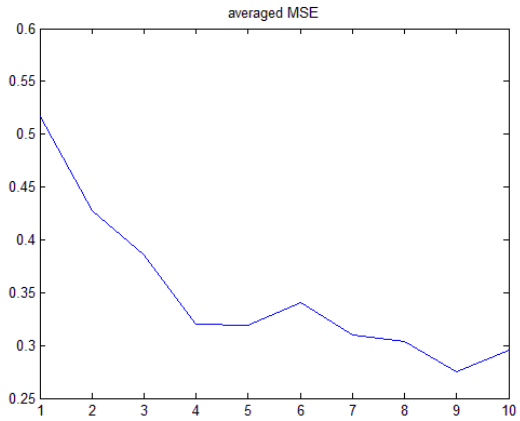

Figure 9: Averaged MSE - online learning

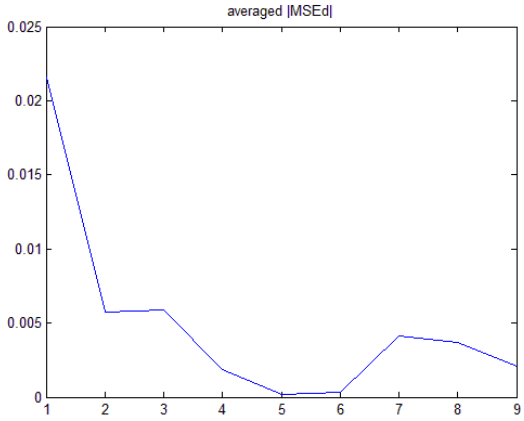

Figure 10: Averaged change in MSE - online learning

Based on these trials, an adequate stopping criterion was chosen for the training procedure. The stopping criterion was chosen to be based on the mean squared error value and the difference between the error values in successive training runs. For the batch learning case, the training would stop if the latest error value was less than a specified tolerance (i.e. 0.4) and the averaged change in error was small (i.e. less than 0.003). For the online learning case, the training would stop if the averaged error value was less than 0.3 and the averaged change in error was less than 0.005. Although the gradient size can give useful insight on the convergence of the network, it was not used because the change in gradient size may vary slightly despite the fact that the error and change in error values are decreasing.

In order to test the accuracy of the models generated after training was completed, a Matlab script testNetwork was created to calculate the percentage of correct classification on both the training and testing set. In order to calculate this accuracy, the input training and test patterns were put through a forward pass through the model and output states were calculated. The correctness criterion that was chosen for the final output was a winner-take-all approach where the largest valued unit in the output layer would be set to one and all others are zero. This was chosen to show how the model would be used to predict new input where the output can only be one value corresponding to the best guess of which number the hand drawn digit represents. The following code shows how the training set accuracy was calculated.

```
accVector = min((train_y == train_outputActivations),[],1);
train_acc = sum(accVector)/nTrainPats;
```

The accuracy vector, *accVector,* compares the targets of the training set *train*_y with the output activations obtained by the trained mode *train_outputActivations* and takes the minimum such that the output is a 0 when there is incorrect classification and 1 with correct classification. The accuracy was then obtained by summing all the correctly classified cases and dividing by the total number of patterns, whether it is the training or test patterns. The network was evaluated based on its performance across three trained models starting from different random initial weights.

It is important to note that in the batch and online learning trials above, the plots for batch learning show 200 runs while the plots for online learning show 100 runs. It was found that for certain values of the learning rate the networks produced very similar accuracy rates. Thus, the stopping criteria were chosen based on these results to try to produce similar performance using both methods so that they can be compared in terms of their performance.

In order to try to understand what the model has learned in terms of its representation the hand drawn digits, the weights learned by the hidden units of the network were examined. The weights can be visualized using the visualizeSample function.

Results

The batch learning and online learning procedures were used to train the network three times. The results of the mean squared error with respect to runs are shown for all runs of the network in the following figures.

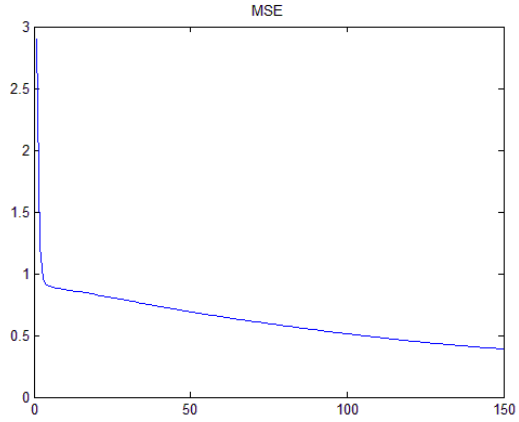

Figure 11: MSE batch learning trial 1

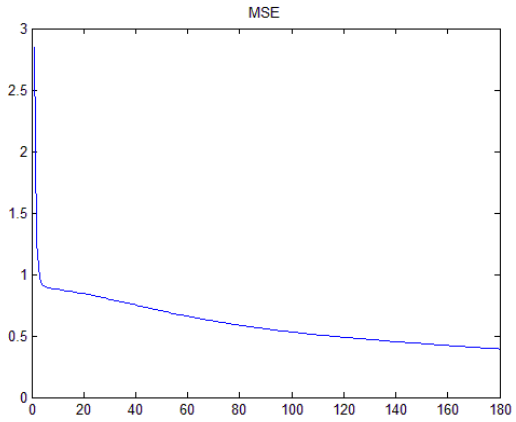

Figure 12: MSE batch learning trial 2

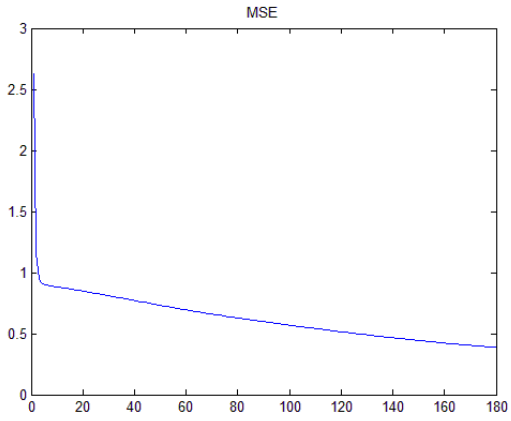

Figure 13: MSE batch learning trial 3

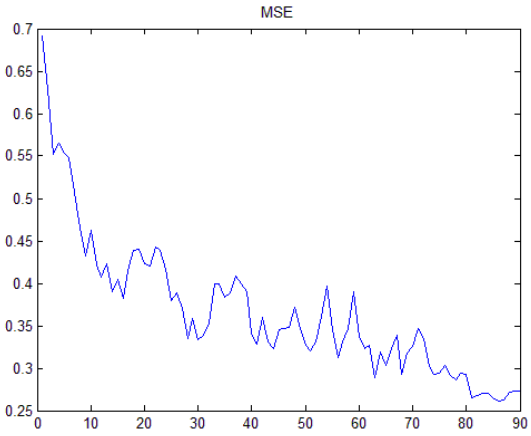

Figure 14: MSE online learning trial 1

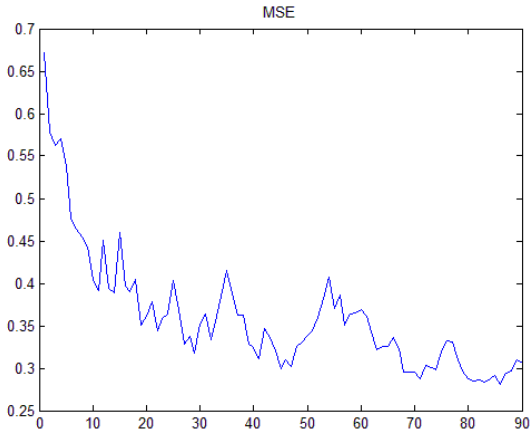

Figure 15: MSE online learning trial 2

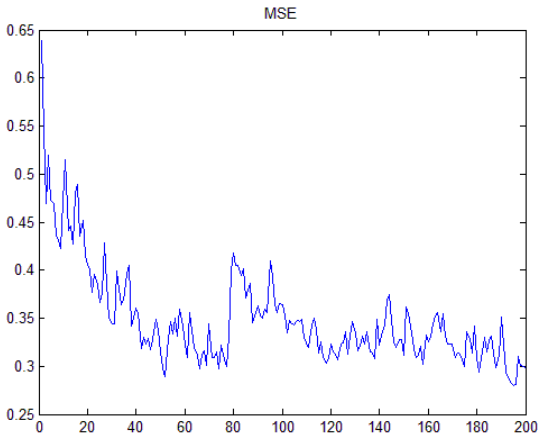

Figure 16: MSE online learning trial 3

The results of the training and test set accuracy for each of the trials in the batch and online learning cases are presented in the table below.

Trial	Batch learning	Online learning
1	training set accuracy = 81.9% test set accuracy = 83.2%	training set accuracy = 82.6% test set accuracy = 83.4%
2	training set accuracy = 78.8% test set accuracy = 78.9%	training set accuracy = 80.2% test set accuracy = 80.5%
3	training set accuracy = 81.5% test set accuracy = 82.1%	training set accuracy = 82.0% test set accuracy = 82.3%

Table 1: Training and test set accuracies

These results show that similar accuracies were achieved between the batch learning and online learning training cases. The slight discrepancy can be attributed to the chosen value of the learning rate and stopping criteria previously mentioned. While the accuracy could have been used to determine the stopping criteria, the error values are typically used where the error must meet some desired tolerance.

Due to the similarity of the accuracy performance, comparing the number of runs required to achieve these accuracies show that online learning is able to converge faster than batch learning on average. It is important to note that if the networks are allowed to run for more iterations, they can achieve higher accuracies with accuracy increasing nonlinearly with respect to the number of iterations.

In order to examine the weights learned by the network, the hidden layer weights were visualized as an image and the results are shown in the following figures.

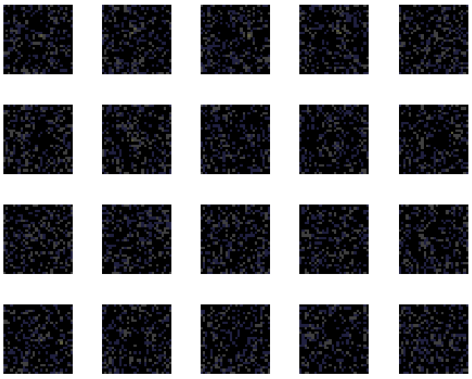

Figure 17: Weights batch learning trial 1

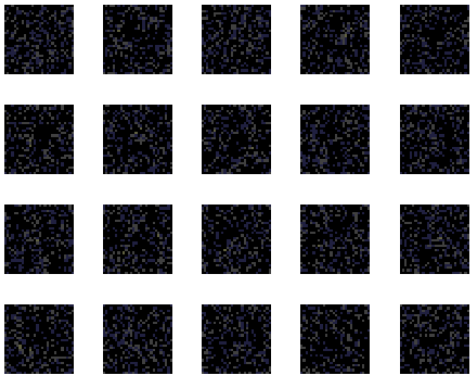

Figure 18: Weights batch learning trial 2

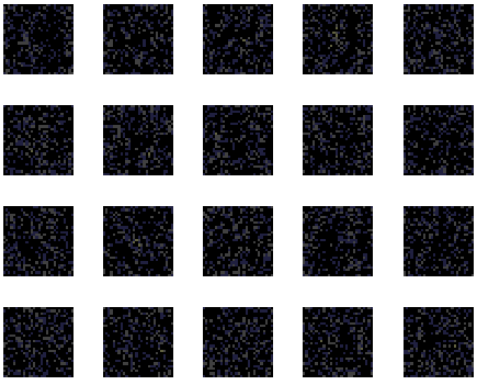

Figure 19: Weights batch learning trial 3

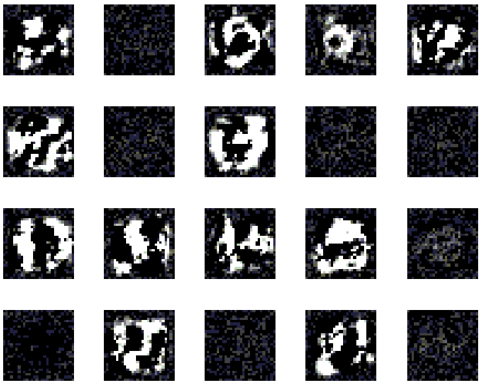

Figure 20: Weights online learning trial 1

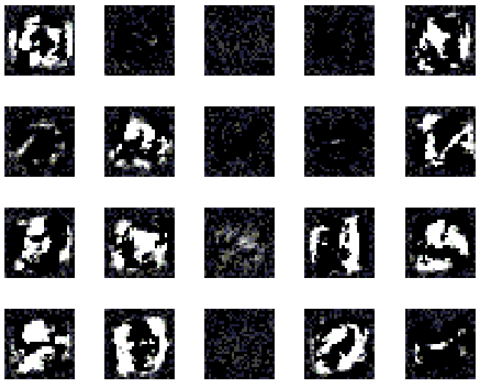

Figure 21: Weights online learning trial 2

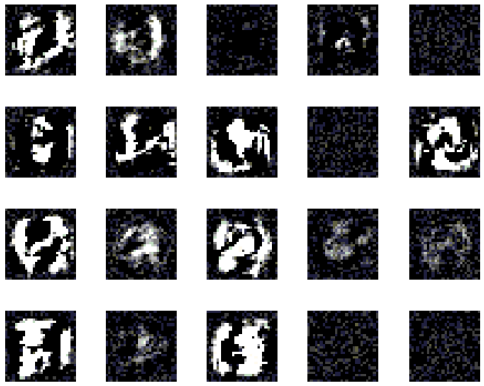

Figure 22: Weights online learning trial 3

These visualizations of the weights show what the internal representation of the network may be by the features they are able to recognize. It is interesting to note that the range of the weight values learned by stochastic gradient descent is much larger than the range of values learned by batch gradient descent, perhaps due to the larger number of weight updates in stochastic gradient descent.

Discussion

The results of the batch gradient descent algorithm and the stochastic gradient descent algorithm show some insight as to what the best method may be for this dataset. While batch learning may perform well for convex or relatively smooth error manifolds, stochastic gradient descent is preferred for error manifolds that have many local minima. The stochastic nature of online learning may aid to jerk the model out of a local minimum and into a more optimal global minimum. In the case of large datasets, using batch gradient descent can be costly since there is only one step taken over the entire training set. The larger the training set, the slower the updates to the weights will occur and the

longer it may take to converge. Thus, an advantage to using stochastic gradient descent is that it converges faster, and this was supported by the results shown above. Since this method computes the gradient and updates the weights after each individual sample, it is able to follow the path of steepest descent after each sample, and the average of all these steps tends to be a good approximation of the true input distribution.

As previously shown, the value of the accuracy for all the networks are roughly around 80%. Higher accuracy was easily achieved by changing the stopping criteria or simply allowing the network to run for more iterations. Generally, higher accuracy was achieved using online learning within a limited number of iterations. The values chosen allowed for similar performance by the two learning procedures for comparison purposes and allowed for convergence within a reasonable amount of computation time.

When examining the weights learned by the networks, many interesting features stand out. The networks seem to have a distributed representation of the digits. The weight matrices are not sparse and some seem to be quite stochastic. However, comparing the weights learnt by batch learning vs online learning show some stark differences. The weights learnt with batch gradient descent do not seem to have any discernable features that we are able to see. In contrast, the weights learnt with stochastic gradient descent seem to have a majority of units that show distinct features of different strokes. These units do not exactly represent any one particular digit or any one of the individual training patterns. While some of these units show similarity to parts of digits (like parts of a 0 or 8) these representations are not recognizable as any number.

Another was to train a multi-layer neural network is to use unsupervised pre-training. This involves using radial basis function units in the first layer and pre-training their weights using k-means clustering, which is essentially a k-winner-take-all competitive learning algorithm. These weights from the input to hidden layer can then be fixed and the weights from the hidden to output layer can be trained as a regular classification network. This two-phase RBF network has the advantage of being able to use unlabeled training data for the first training phase. Since the first phase involves unsupervised learning, the weights are expected to learn features that are inherent to the data set. Therefore, another advantage is that this allows the weights from the input to hidden layer to be based off of features that are learned from clustering the data and may give insights on the distribution of the hand drawn digits that could help in classification.

However, this scheme can me modified to include a third phase which involves backpropagation training of the RBF network. By applying backpropagation to the entire network, the parameters can be adapted simultaneously. A paper that applies three phase learning on hand drawn digits shows that applying the third phase improves the performance of the RBF classifiers: “The results suggest that unsupervised pre-training guides the learning towards basins of attraction of minima that support better generalization from the training data set; the evidence from these results supports a regularization explanation for the effect of pre-training.” [1] This allows the weights learnt using unsupervised learning to be adapted using supervised learning and therefore improved for the purpose of classification.

Another possible advantage is that using unsupervised learning in the initial phase can help find better initial weights for the supervised learning procedure, which may help in reducing convergence time.

There can be many different modifications that could be made to the neural network model to make it learn a multi-level hierarchical model of digits, similar to the human visual system. First, you would need to include more hidden layers to form a deep neural network. Aside from the typical multi-layer deep neural network structure, there can be many different configurations to the network that would allow it to learn a hierarchical model of digits. The use of expert systems trained on different types of distributions of 2D image data may be useful in classifying images; however, whether or not this is similar to the human visual system is debatable.

Another approach is to use a multi-layer neural network that is pre-trained using unsupervised learning, as in the previously mentioned example. Different types of unsupervised methods, such as stacked RBMs or stacked convolutional auto-encoders can be applied to the network. A paper discussing how stacked convolutional auto-encoders can be used for hierarchical feature extraction found that these pre-trained CNNs tend to outperform randomly initialized neural nets in the task of classifying the MNIST dataset [2]. They also support the idea that these convolutional filters are biologically plausible.

References

[1] Schwenker, F., Kestler, H. A., & Palm, G. (2001). Three learning phases for radial-basis-function networks. *Neural Networks,14*(4-5), 439-458. doi:10.1016/s0893-6080(01)00027-2

[2] Masci, J., Meier, U., Cireşan, D., & Schmidhuber, J. (2011). Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction. *Lecture Notes in Computer Science Artificial Neural Networks and Machine Learning – ICANN 2011,* 52-59. doi:10.1007/978-3-642-21735-7_7

Appendix A: testing123.m

```
% PSYCH 734/CSE 734
% Assignment 3
% Sara Jamil

%% Training bp with batch learning
clear; close all; clc;

bpinit

MSE = [];
gradSize = [];

bp %first run
for i = 1:49 %max 500 iterations total
    i %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % calculating the change in MSE
    % for last 10 iterations
    clear diff %inconvenient name
    MSEd = abs(sum(diff(MSE(1,(length(MSE)-9):end)))/9);

    % stopping criterion:
    % last MSE must be below 0.4 AND
    % change in MSE averaged over last 10 runs is less than 0.003
    if(MSE(1,end)<=0.4)&&(MSEd<=0.003)
        break;
    else
        % run training
        bp %nIters = 10 (each bp call)
    end

end

figure(1)
plot(MSE);
title('MSE');

figure(2)
plot(gradSize);
title('gradSize');

testNetwork


%% Training bp with online learning
clear; clc;

bpinit

% change the learning rate
epsilon = 0.01; %chosen after many trials

MSE = [];
```

```
gradSize = [];

bponline %first run
for i = 1:49 %max 500 iterations total
    i %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % calculating the change in MSE
    % and the averaged MSE
    % for last 10 iterations
    clear diff %inconvenient name
    MSEd = abs(sum(diff(MSE(1,(length(MSE)-9):end)))/9);
    MSEa = sum(MSE(1,(length(MSE)-9):end))/10;

    % stopping criterion:
    % MSE averaged over last 10 runs must be below 0.3 AND
    % change in MSE averaged over last 10 runs is less than 0.005
    if(MSEa(1,end)<=0.3)&&(MSEd<=0.005)
        break;
    else
        % run training
        bponline %nIters = 10 (each bp call)
    end

end

figure(1)
plot(MSE);
title('MSE');

figure(2)
plot(gradSize);
title('gradSize');

testNetwork
```

Appendix B: bponline.m

```
%%% bp modifier to use online learning instead of batch learning
% a.k.a. stochastic gradient descent

for i = 1:nIters
  sumSqrError = 0.0;
  outputWGrad = zeros(size(outputWeights));
  hiddenWGrad = zeros(size(hiddenWeights));

  for pat = 1:nTrainPats,
    %%%%% forward pass %%%%%
    inputStates = train_x(:,pat);
    inputStatesBias = [[inputStates]',[1]]';
    hiddenNetInputs = hiddenWeights * inputStatesBias;
    hiddenStates = sigmoidFunc(hiddenNetInputs);
    % Add a '1' to the hidden state vector representing the bias
    % unit's input to the output layer
    hidStatesBias = [[hiddenStates]',[1]]';
    target = train_y(:,pat);
    outputNetInputs = outputWeights * hidStatesBias;
    outputStates = sigmoidFunc(outputNetInputs);

    %%%%% backward pass %%%%%
    diff = outputStates - target;
    sumSqrError = sumSqrError + sum(diff' * diff);

    %%%%%%%%%%%% Calculate Deltas %%%%%%%%%%%%%%%%%%%%%%%%%
    % Delta is an intermediary term for computing the weight change.
    % For each unit i, delta_i is the derivative
    % of the error with respect to the ith unit's net input
    outputDel = outputDeltas(outputStates,target);
    hiddenDel = hiddenDeltas(outputDel,hidStatesBias,outputWeights);

    %%%%%%%%%%%% Calculate gradients %%%%%%%%%%%%%%%%%%%%%%%%%
    % The gradient is the vector of derivatives of the error with
    % respect to each weight.
    outputWGrad = outputDel * hidStatesBias';
    hiddenWGrad = hiddenDel(1:nHidden,:) * inputStatesBias';

    % There are 60,000 MNIST training patterns so for batch learning
    % it's important to divide the weight change by the number of
    % patterns. For online learning / stochastic gradient descent,
    % this division by nTrainPats would not be necessary.
    outputWeights = outputWeights - epsilon * outputWGrad;
    hiddenWeights = hiddenWeights - epsilon * hiddenWGrad;
  end

  % calculate MSE and gradSize averaged over pass thru training set
  MSE(1,size(MSE,2)+1) = sumSqrError / nTrainPats;
  gradSize(1,size(gradSize,2)+1) = norm([hiddenWGrad(:);outputWGrad(:)]);
  fprintf(1,'E=%f,
|G|=%f\n',MSE(1,size(MSE,2)),gradSize(1,size(gradSize,2)));
end
```

Appendix C: testNetwork.m

```
% Calculate accuracy of training and test set
% after training is done

train_outputActivations = zeros(nOutputs, nTrainPats);
test_outputActivations = zeros(nOutputs, nTestPats);

for pat = 1:nTrainPats,
    %%% forward pass
    inputStates = train_x(:,pat);
    inputStatesBias = [[inputStates]',[1]]';
    hiddenNetInputs = hiddenWeights * inputStatesBias;
    hiddenStates = sigmoidFunc(hiddenNetInputs);
    hidStatesBias = [[hiddenStates]',[1]]';
    outputNetInputs = outputWeights * hidStatesBias;
    outputStates = sigmoidFunc(outputNetInputs);

    %%% apply winner-take-all as correctness criterion
    [out, index] = max(outputStates);
    train_outputActivations(index,pat) = 1;
end

%%% calculating accuracy for training set
accVector = min((train_y == train_outputActivations),[],1);
train_acc = sum(accVector)/nTrainPats;
fprintf('training set accuracy = %3.1f%% \n',train_acc*100);

for pat = 1:nTestPats
    %%% forward pass
    inputStates = test_x(:,pat);
    inputStatesBias = [[inputStates]',[1]]';
    hiddenNetInputs = hiddenWeights * inputStatesBias;
    hiddenStates = sigmoidFunc(hiddenNetInputs);
    hidStatesBias = [[hiddenStates]',[1]]';
    outputNetInputs = outputWeights * hidStatesBias;
    outputStates = sigmoidFunc(outputNetInputs);

    %%% apply winner-take-all as correctness criterion
    [out, index] = max(outputStates);
    test_outputActivations(index,pat) = 1;
end

%%% calculating accuracy for test set
accVector = min((test_y == test_outputActivations),[],1);
test_acc = sum(accVector)/nTestPats;
fprintf('test set accuracy = %3.1f%% \n',test_acc*100);
```

Appendix D: visualizeSample2.m

```
function visualizeSample2(in_x,in_y,sample)
% SORRY I MODIFIED THIS FUNCTION
%
% Modified function can plot 8 samples
% from the MNIST dataset at the same time
%
% Example: visualizeSample(train_x,train_y,1:8)

for k = 1:length(sample)
    x = in_x(:,sample(k));
    x = x'; %modified because of transpose in bpinit

    [Nsamples,Npixels] = size(x);

    % Convert the sample into its original 2D form
    x = reshape(x,sqrt(Npixels),sqrt(Npixels));

    % for displaying the correct category
    y = in_y(:,sample(k));
    [n,i] = max(y);

    % Display the image with target as title
    figure(1)
    subplot(2,4,k);
    imshow(x'); %another transpose...
    title(num2str(i-1));
    hold on
end
```

Appendix E: visualizeSample3.m

```
function visualizeSample3(in_x)
% SORRY I MODIFIED IT AGAIN...
%
% This one plots all the hidden units' weights
% at the same time (as an image)
%
% Example: visualizeSample(hiddenWeights)

for k = 1:size(in_x,1)
    x = in_x(k,1:784);
    [Nsamples,Npixels] = size(x);

    % Convert the sample into its original 2D form
    x = reshape(x,sqrt(Npixels),sqrt(Npixels));

    % Display the image
    figure(1)
    subplot(4,5,k);
    imshow(x);
end
```