Computer Vision and Image Processing CSE573
Homework 4: Autoencoders for Image Classification

Charushi Nanwani
50248736

1.2

- **Q1 (8 points) How does an autoencoder detect errors?**
  The error function used is mean squared error with L2 and Sparsity Regularizer.

- **Q2 (8 points) The network starts out with 28 × 28 = 784 inputs. Why do subsequent layers have fewer nodes?**
  An autoencoder is an artificial feedforward neural network for unsupervised learning algorithm. Autoencoder consists of an input layer, an output layer and one or more hidden layers. An autoencoder is trained to attempt to copy its input to its output. Internally, it has a hidden layer that describes a code used to represent the input.
  We need to apply some constraint on the subsequent layers otherwise, it would be an approximation to identity function so as the output is similar to input.
  However, having fewer nodes in the subsequent layers forces the autoencoder to represent the input data in fewer numbers. This allows for only important features to be passed to the next layer. Thus, having fewer nodes in the hidden layers helps in dimensionality reduction and is useful for feature extraction.

- **Q3 (8 points) Why are autoencoders trained one hidden layer at a time?**
  An autoencoder is trained using a variant of backpropagation. However, when we have many hidden layers, backpropagation faces an issue. Once errors are back propagated for the first few layers, they become small and insignificant. This implies that the network is able to easily reconstruct the input data. This problem can be solved by using some initial weights to get a better final solution. This process is referred to as pre-training which was developed by Geoffrey Hinton for training 'deep-autoencoders'.
  In his paper, he mentioned that for an autoencoder having multiple hidden layers, it is difficult to optimize the weights, because if the weights are large, local minima obtained would not be best and if the weights are small, the gradients obtained would become insignificant making it difficult to train the encoder. If the initial weights are close to a good solution, gradient descent works well, but finding such initial weights requires a very different type of algorithm that learns one layer of features at a time. Thus, the autoencoders are trained one hidden layer at a time to obtain the optimal initial weights.

- **Q4 (8 points) How were the features in Figure 3 obtained? Compare the method of identifying features here with the method of HW1. What are a few pros and cons of these methods?**
  We are plotting the weights using the plotweights() function. The weights are learned by the 784 by 100 encoder which is trained on the data provided.
  In HW1, we are extracting filter response of the images, and creating a dictionary using k-means clustering. In k-means clustering we are selecting alpha random pixels, whereas

in autoencoders, features extracted are relevant. We were using 20 filters to extract filter responses and select random pixels, and used k-means to cluster to form visual words dictionary. Here, we are using neural network on continuous variables and so the features extracted at each level will give a better representation. So, autoencoders gives better accuracy compared to k-means clustering approach for classification. However, k-means performs faster than the autoencoder.

- **Q5 (8 points) What does the function plotconfusion do?**
  plotconfusion(targets,outputs) returns a confusion matrix plot for the target and output data in targets and outputs, respectively.
  On the confusion matrix plot, the rows correspond to the predicted class (Output Class), and the columns show the true class (Target Class). The diagonal cells show for how many (and what percentage) of the examples the trained network correctly estimates the classes of observations. That is, it shows what percentage of the true and predicted classes match. The off diagonal cells show where the classifier has made mistakes. The column on the far right of the plot shows the accuracy for each predicted class, while the row at the bottom of the plot shows the accuracy for each true class. The cell in the bottom right of the plot shows the overall accuracy.

1.2.1 Activation Functions

- **Q6 (8 points) What activation function is used in the hidden layers of the MATLAB tutorial?**
  The activation function used in the hidden layers is the logistic sigmoid function.

- **Q7 (8 points) In training deep networks, the ReLU activation function is generally preferred to the sigmoid activation function. Why might this be the case?**
  Rectified linear units, compared to sigmoid function or similar activation functions, allow for faster and effective training of deep neural architectures on large and complex datasets.
  A rectifier linear unit (ReLU) function can be written as:

$$\boldsymbol{f(x) = x^{+} = max(0, x)}$$

  ReLU provides the benefit of sparsity and reduced likelihood of vanishing gradient. In ReLU, when x>0, the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning. The other benefit of ReLUs is sparsity. Sparsity arises when x<=0. These values add to the sparsity of the layer, as more such values, the sparse the layer will be. Sparse representations seem to be more beneficial than dense representations.
  In addition, ReLU allows for efficient computation and are scale invariant.

$$max(0, ax) = a\ max(0, x)$$

1.2.2 Initialization

- **Q8 (8 points) The MATLAB demo uses a random number generator to initialize the network. Why is it not a good idea to initialize a network with all zeros? How about all ones, or some other constant value? (Hint: Consider what the gradients from backpropagation will look like.)**
    - A neural network sometimes tends to get stuck in local minima. If we initialize all the weights starting with zeros, it increases the chances of being stuck. To reduce this, you want to give many different starting values.
    - Also, if the neurons in network start with the same weight, they will obtain a similar gradient. If the input weights are same, all units in the hidden layer will be same too. Even though the neurons are changing the values, they will end up with the same value. So after each update, the parameters corresponding to the inputs going into each of the two hidden units are identical. It implies that the similar feature is being computed repeatedly. All the hidden units are computing the same output based on the input which would be redundant.

  Thus, we need to initialize the weights randomly to avoid symmetry.

1.2.3 Training Loop

- **Q9 (8 points) Give pros and cons for both stochastic and batch gradient descent. In general, which one is faster to train in terms of number of epochs? Which one is faster in terms of number of iterations?**

  Batch and Stochastic gradient descent differ in terms of the amount of data being used to calculate the gradient of the function.

  Batch Gradient descent-

  In batch gradient descent, the entire training dataset is passed and the gradient of the cost function is evaluated. The weights are updated incrementally after each epoch.

  $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

  Pros:
    - If we are using plain batch gradient descent, we get a stable convergence compared to other variants.
    - It gives a good minimum, if the data is small and is not to be updated.
    - Gradient descent assures that the updates are done in the 'right direction'- choosing the one that minimizes the cost function.

  Cons:
    - We need to calculate the gradient of the complete dataset to perform just one update. This leads to slow process and also can lead to memory problems, as if the data is large, it won't be able to fit in memory
    - It doesn't allow to update the models online (adding new examples later on)

Stochastic Gradient descent-

Stochastic Gradient descent provides a parameter update.

$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x(i); y(i))$

Pros:

- It is computationally faster than batch gradient descent
- SGD can converge faster than batch training because it performs updates more frequently
- SGD can improve generalization performance for large data set problems

Cons:

- SGD keeps overshooting and can be hard to converge. This problem can be solved by decreasing learning rate.
- Moving along the direction of a stochastic gradient does not always guarantee a decrease of the entire training loss. Under large stochastic gradient variance, the estimated parameters often drastically bounce around the global optimal solution

Batch gradient descent is faster in terms of epochs, while in terms of number of iterations, stochastic gradient descent performs better.

In batch, since we are passing the entire dataset in each epoch, it will train faster and provide better results.

In terms of iterations, stochastic gradient descent would give better results, as for each iteration, a random sample is chosen and gradient is calculated. So, this would be faster than going through the entire data.

- **Q10 (28 points) Try playing around with some of the parameters specified in the tutorial. Perhaps the sparsity parameters or the number of nodes; or number of layers. Report the impact of slightly modifying the parameters. Is the tutorial presentation robust or fragile with respect to parameter settings?**

Changing the following parameters: L2 regularization, sparsity regularization, sparsity proportion, number of nodes in hidden layer, number of hidden layers.

Changing the sparsity Regularization

Sparsity Regularization	Accuracy (with fine tuning)
0.5	98.8%
1	99.0%
7	98.7%

Changing the sparsity proportion:

Sparsity Proportion	Accuracy (with fine tuning)
0.001	10%
0.5	99.3%
0.8	99.2%

Sparsity proportion varies between 0 and 1. For a very low value of sparsity proportion (0.001), the accuracy is reduced to 10%. And if the value is 0.5, we are getting a high accuracy of 99.3%. Increasing the proportion varies the accuracy, however only slightly.

Changing L2 regularization:

L2 regularization	Accuracy (with fine tuning)
0.1	10%
0.01	97.8%
0.001	98.9%

Changing number of nodes in hidden layer:

1st hidden layer	2nd hidden layer	Accuracy (with fine tuning)
200	100	99.2%
800	500	98.9%
100	100	99.0%

Changing number of hidden layers:

Number of hidden layers	Accuracy (with fine tuning)
3(100,50,30)	98.4%
3(200,100,50)	99.5%
1(100)	98.4%

I tried changing the activation function of the first layer to 'poslin' which is ReLU function, while using the sparsity proportion as 0.5.

$$deepnet.layers\{1\}.transferFcn = 'poslin'$$

The accuracy obtained is 99.2%.
Changing different parameters in combination i.e.
Using three layers 250,100,and 50 and setting sparsity proportion to 0.5, the highest accuracy obtained is 99.5%.

The accuracy is improved slightly. Thus, the tutorial presentation seems to be robust to parameter settings, as we are getting similar accuracy without affecting it drastically.

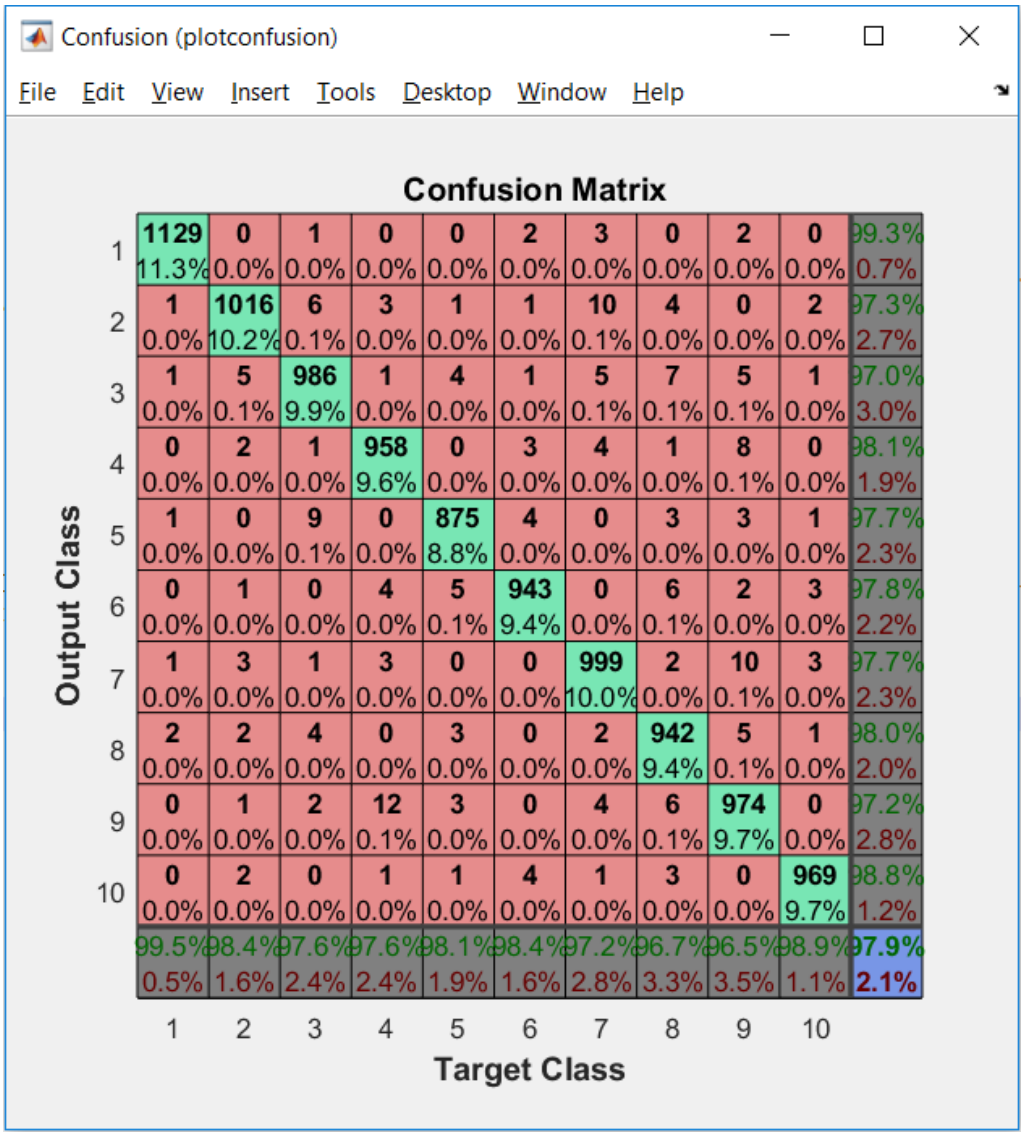
Extra Credit

- The code for loading MNIST datasets and running the autoencoder example on this dataset is given in '50248736' directory. The files in this folder are:

- ***'readTrainingImages.m'***- for reading the training images and training labels
- ***'readTestData.m'***- for reading the test data set and test labels
- ***'main.m'***- for performing the autoencoder example on MNIST dataset using autoencoders

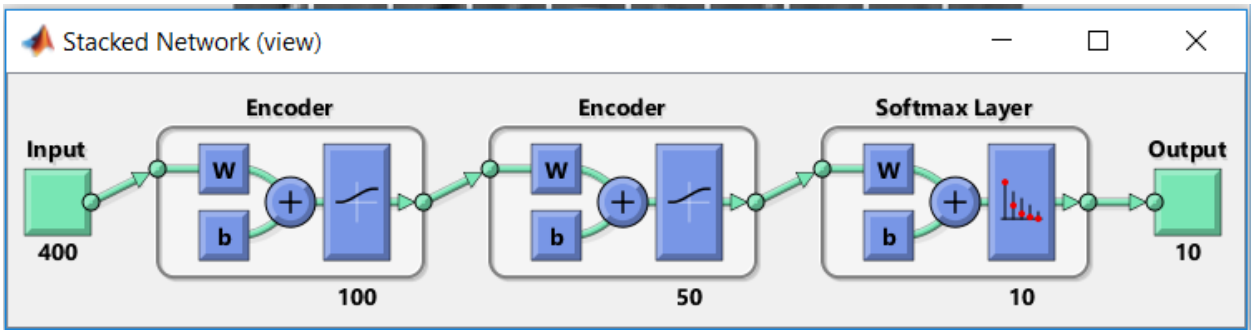
➢ For the MNIST dataset, using the example given in the MATLAB tutorial without changing any parameters, the accuracy obtained is 97.9%. The features are different for real dataset when compared with synthetic dataset. Performance is slightly better when using synthetic dataset (98.7%). However, since we are training and testing on the same datasets, the features extracted are properly identified in both the cases. So, the accuracy obtained is good.

➢ The confusion matrix for the final classifier is shown below:



➢

- Network architecture



- Parameter settings

For first autoencoder:

- Hidden layer size-100
- Max Epochs-400
- L2 Weight Regularization-0.004
- Sparsity Regularization-4
- Sparsity Proportion-0.15

For second autoencoder:

- Hidden layer size-50
- Max Epochs-100
- L2 Weight Regularization-0.002
- Sparsity Regularization-4
- Sparsity Proportion-0.1

- Number of training Images-60000
- Number of testing Images-10000

References:

- https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf
- https://www.coursera.org/learn/machine-learning/lecture/ND5G5/random-initialization
- http://mccormickml.com/2014/05/30/deep-learning-tutorial-sparse-autoencoder/
- http://ruder.io/optimizing-gradient-descent/
- https://en.wikipedia.org/wiki/Autoencoder
- https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
- https://towardsdatascience.com/difference-between-batch-gradient-descent-and-stochastic-gradient-descent-1187f1291aa1
- https://arxiv.org/pdf/1506.08350.pdf
- https://cilvr.cs.nyu.edu/diglib/lsml/bottou-sgd-tricks-2012.pdf
- https://www.mathworks.com/help/nnet/ref/plotconfusion.html