# CSE-573
# Homework 4: Autoencoders for Image Classification

Name: Rajiv Ranjan
Person ID: 50249099
Ubname:rajivran

Write Up PDF

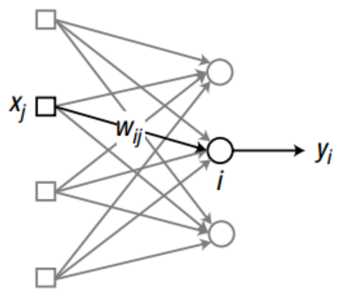
1.2 Questions
Q1) How does an autoencoder detect errors?
Soln) The learning process in an autoencoder is simply defined as minimizing a loss function.

$$L\ (x,\ g(f(x)))$$

where L is a loss function penalizing g(f(x)) for being dissimilar from x, such as the mean squared error.

$$E = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn} - \hat{x}_{kn}\right)^2}_{\text{mean squared error}} + \lambda * \underbrace{\underbrace{\Omega_{weights}}_{L_2}}_{\text{regularization}} + \beta * \underbrace{\underbrace{\Omega_{sparsity}}_{\text{sparsity}}}_{\text{regularization}},$$

So this error called the reconstruction error is detected during the training phase using the following logic, say for a single layer neural network,



let di be the known correct output and yi be he output obtained from one iteration of training phase for an input node j.
error is generally given as e(i)=d(i)-y(i) in the back propagation algorithm
here xj is the input node as we are considering it to be a single layer neural network, in case of multi layer it becomes the output from the last hidden layer to the output layer
wij = The weight between the output node i and input node j

So for training the neural network for say n iterations:
Error, E is given as the mean squared value of all the individual errors:

E= (e(i).^2)for n iterations/n

Q2) The network starts out with 28 × 28 = 784 inputs. Why do subsequent layers have fewer nodes?

Ans: An autoencoder whose code dimension is less than the input dimension is called undercomplete. The subsequent layers have fewer nodes because learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

In contrast, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data. Similarly, a similar problem occurs if the hidden code is allowed to have dimension
equal to the input, and in the overcomplete case in which the hidden code has dimension greater than the input. In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

Q3) Why are autoencoders trained one hidden layer at a time?
Ans: We train one layer at a time, so that we train a network with 1 hidden layer, and only after that is done, train a network with 2 hidden layers, and so on. At each step, we take the old network with $k-1$ hidden layers, and add an additional $k$-th hidden layer (that takes as input the previous hidden layer $k-1$ that we had just trained). The weights from training the layers individually are then used to initialize the weights in the final/overall deep network, and only then is the entire architecture "fine-tuned" (i.e., trained together to optimize the labeled training set error).
The success of greedy layer-wise training can be attributed to these two factors:

a) **Availability of data:** While labeled data can be expensive to obtain, unlabeled data is cheap and plentiful. The promise of self-taught learning is that by exploiting the massive amount of unlabeled data, we can learn much better models. By using unlabeled data to learn a good initial value for the weights in all the layers $W^{(l)}$ (except for the final classification layer that maps to the outputs/predictions), our algorithm is able to learn and discover patterns from massively more amounts of data than purely supervised approaches. This often results in much better classifiers being learned.
b) **Better local optima:** After having trained the network on the unlabeled data, the weights are now starting at a better location in parameter space than if they had been randomly initialized. We can then further fine-tune the weights starting from this location. Empirically, it turns out that gradient descent from this location is much more likely to lead to a good local minimum, because the unlabeled data has already provided a significant amount of "prior" information about what patterns there are in the input data.

**Q4)** 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?

Soln). Basically, how an autoencoder is working here is that an input image is there say 28*28, now we resize it into a 784*1 matrix so that it could be used as an input for the autoencoder, the first part of the autoencoder does the encoding part, which is taking some initial weights and biases and feeding it to the hidden layer. Every node in the hidden layer is basically an activation function, and based on the inputs given and the ‘SparsityProportion’ value a certain no of internal hidden nodes are activated and after that a reconstruction error value is also entered as ‘L2WeightRegularization’ along with a sparsity penalty as ‘SparsityRegularization’. So keeping these values in mind, a loss function is generated which is used by the encoder for learning.
Based on these values from every input node some feature is detected and is passed to the next

layer which is a hidden layer. Similarly, modifying these parameters can help tune the features that are extracted in each layer. So in figure 3 the output after the first layer is shown which is a set of features extracted from each image after running the encoder part of the autoencoder.

In HW1, the features were extracted using the Gaussian filters, which works on the concept of traversing a particular kind of filter throughout the image to extract features pertaining to that filter from the image.

**A few pros and cons of these methods are discussed in detail which goes as follows:**
Our first homework was based on extracting features from an image using the bag of visual words approach, with its spatial pyramid extension. There we simply built a dictionary out of features of the images by performing k-means algorithm where as an autoencoder is a neural network based feature extraction method achieves great success in generating abstract features of high dimensional data. Bag-of-words method is very popular due to it's simplicity and computational efficiency. Particularly for the task of action recognition, where state-of-the-art feature extraction approaches lead to a vast amount of features even for comparably small datasets, compact and efficient feature representations like bag-of-words are widely.

For most classification tasks, the application of a bag-of-words model can be subdivided into three major steps. First, a visual vocabulary is created by clustering the features, usually using k-Means model. In the second step, the input data is quantized and eventually represented by a histogram of the previously obtained visual words. Finally, the data is classified using a support vector machine. However, there are some significant drawbacks in this pipeline. Clustering algorithms like k-Means and Gaussian mixture models require the computation of distances of all input features to all cluster centers in each iteration. Due to the extensive amount of features generated by state-of-the-art feature extraction algorithms it is infeasible to run the algorithms on the complete data and a subset has to be sparsely sampled. Although it is possible to run k-Means several times and select the most compact clustering to ensure a good subsampling, k-Means is usually not sensitive to the subsampling of local descriptors. However, the visual vocabulary is created without supervision and optimized to be a good representation of the overall data, whereas for classification, the visual vocabulary should ideally be optimized to best separate the classes.
**Here are the limitations of the bag of words approach:**

- **Vocabulary**: The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
- **Sparsity**: Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
- **Meaning**: Discarding visual word order ignores the context, and in turn meaning of visual words.

**Advantages of using autoencoders are as follows**:
Since we know that autoencoders can be used for finding a low-dimensional representation of your input data. Some of these features may be redundant or correlated, resulting in wasted processing time and overfitting in our model (too many parameters). It is thus ideal to only include the features we need. If your "reconstruction" of inputs is very accurate, that means your low-dimensional representation is good. You can then use this transformation as input into another model.

**Disadvantages of Autoencoders are as follows**:
1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on.
2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs. This differs from lossless arithmetic compression.
3) They are rarely used in practical applications. I

Q5) What does the function plotconfusion do?
Ans: The plotconfusion is a matlab function having the following syntax:

```
plotconfusion(targets,outputs)
```

It returns a confusion matrix plot for the target and output data in targets and outputs respectively. So in our tutorial what we are doing is that:

```
plotconfusion(tTest,y);
```

it is ploting a confusion matrix between the target class and output class as stated above.
Where 'tTest' is the actual output and 'y' is the output from the deep neural networks.
The numbers in the bottom right-hand square of the matrix give the overall accuracy.

Q6) What activation function is used in the hidden layers of the MATLAB tutorial?
Ans: sigmoid transfer function in the hidden layer.

Q7) In training deep networks, the ReLU activation function is generally preferred to the sigmoid activation function. Why might this be the case?

Ans: In training deep networks, the ReLU activation function is generally preferred to the sigmoid activation function because of the vanishing gradient problem. It is similar to the error delta in the back propagation algorithm. The vanishing gradient in the training process with the back-propagation algorithm occurs when the output error is more likely to fail to reach the farther nodes. The back-propagation algorithm trains the neural network as it propagates the output error backward to the hidden layers. However, as the error hardly reaches the first hidden layer, the weight cannot be adjusted. Therefore, the hidden layers that are close to the input layer are not properly trained. There is no point of adding hidden layers if they cannot be trained.

The representative solution to the vanishing gradient is the use of the Rectified Linear Unit (ReLU) function as the activation function. It is known to better transmit the error than the sigmoid function. The ReLU function is defined as follows:

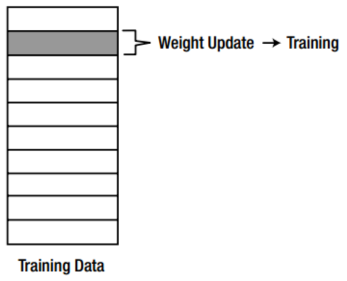
$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$
$$= \max(0,\ x)$$

It produces zero for negative inputs and conveys the input for positive inputs.
The sigmoid function limits the node's outputs to the unity regardless of the input's magnitude. In

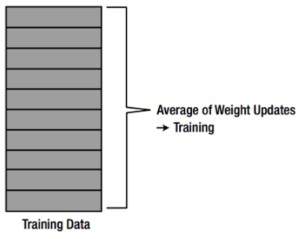
contrast, the ReLU function does not exert such limits.

Q8) 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?
Soln) It is not a good idea to initialize a network with all zeros because:
Firstly, neural networks tend to get stuck in local minima, so it's a good idea to give them many different starting values. This cannot happen if we use all zeros or same constant value as the starting weights. Secondly, if the neurons start with the same weights, then all the neurons will follow the same gradient, and will always end up doing the same thing as one another. This way the learning will be hindered in this case, for the neural networks.
If the initial value is zeros, then the network doesn't learn at all.

Q9) 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?
Soln: **Stochastic gradient descent**: The Stochastic Gradient Descent (SGD) calculates the error for each training data and adjusts the weights immediately. If we have 100 training data points, the SGD adjusts the weights 100 times. As the SGD adjusts the weight for each data point, the performance of the neural network is crooked while the undergoing the training process.
The below figure shows it:



**Batch gradient descent:** In the batch method, each weight update is calculated for all errors of the training data, and the average of the weight updates is used for adjusting the weights. This method uses all of the training data and updates only once. Figure below explains the weight update calculation and training process of the batch method.



The batch method calculates the weight update as:

$$\Delta w_{ij} = \frac{1}{N}\sum_{k=1}^{N}\Delta w_{ij}(k)$$

where, Δwij(k) is the weight update for the k-th training data and N is the total number of the training data.

Pros and Cons:

1) The SGD trains every data point immediately and does not require addition or averages of the weight updates. Therefore, the code for the SGD is simpler than that of the batch.
2) The batch method requires more time to train the neural network to yield a similar level of accuracy of that of the SGD method. In other words, the batch method learns slowly.
3) The SGD yields faster reduction of the learning error than the batch, the SGD learns faster.

**BGD has more number of training examples, so it would train faster with respect to number of epochs. SGD has just one of it and it would be faster in terms of no of iterations.**

**Q10) 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?**

Soln:) The various parameters used in the neural networks and the results associated with playing around with them has been presented below:

| Parameter Changed | MaxEpochs | hidden size1 | L2WeightRegularization | SparsityRegularization | SparsityProportion | hidden size2 | Confusion Matrix result of test images | OverAll Accuracy after applying backpropagation for fine tuning |
|---|---|---|---|---|---|---|---|---|
| Default | 400 | 100 | 0.004 | 4 | 0.15 | 50 | 59.7 | 98.4 |
| MaxEpochs halved | 200 | 100 | 0.004 | 4 | 0.15 | 50 | 48.4 | 98.4 |
| MaxEpochs doubled | 800 | 100 | 0.004 | 4 | 0.15 | 50 | 57.6 | 98.8 |
| L2WeightRegularization halved | 400 | 100 | 0.002 | 4 | 0.15 | 50 | 54.4 | 98.8 |
| L2WeightRegularization doubled | 400 | 100 | 0.008 | 4 | 0.15 | 50 | 51.2 | 98.6 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SparsityRegularization doubled | 400 | 100 | 0.004 | 8 | 0.15 | 50 | 50.7 | 98.7 |
| SparsityRegularization halved | 400 | 100 | 0.004 | 2 | 0.15 | 50 | 58.5 | 98.9 |
| SparsityProportion doubled | 400 | 100 | 0.004 | 4 | 0.3 | 50 | 27.1 | 99.1 |
| SparsityProportion halved | 400 | 100 | 0.004 | 4 | 0.075 | 50 | 23.7 | 95.2 |
| hiddensize1 halved | 400 | 50 | 0.004 | 4 | 0.15 | 50 | 37.6 | 97.4 |
| hiddensize1 doubled | 400 | 200 | 0.004 | 4 | 0.15 | 50 | 65.2 | 99.4 |
| hiddensize2 halved | 400 | 100 | 0.004 | 4 | 0.15 | 25 | 44.4 | 98.7 |
| hiddensize2 doubled | 400 | 100 | 0.004 | 4 | 0.15 | 100 | 69.6 | 98.7 |

Result: As we can see that there is not much change in the results of the confusion matrix for overall accuracy after applying backpropagation for fine tuning even after changing the values of the various parameters associated with not only the training of autoencoder neural network namely these MaxEpochs, L2WeightRegularization, SparsityRegularization and SparsityProportion. But also changing both the hiddenlayer sizes for encoder and decoder part of the autoencoder.

Here all the parameter values have been halved and doubled and it can be shown that there is not much improvement or degradation in the accuracy provided by the autoencoder which assumes a value between 95% to 99%.

Hence it can be said that the tutorial presented is robust to the impact of the slight modification the various parameters associated with the autoencoder.

**1.3 Extra Credit**

For the extra credit part, we have to download and process the MNIST data as described in the specified paper. The data has been downloaded using:
loadMNISTImages.m and loadMNISTLabels.m files.

Considering the training time, this example employs only 10,000 images with the training data and verification data in an 8:2 ratios. Therefore, we have 8,000 MNIST images for training and 2,000 images for validation of the performance of the neural network.

Here is the design of the neural network that will be used for the implementation of this part.

Let’s consider a Neural Network that recognizes the MNIST images. As the input is a 28´28 pixel black-and-white image, we allow 784(=28x28) input nodes. The feature extraction network contains a single convolution layer with 20 9*9 convolution filters. The output from the convolution layer passes through the ReLU function, followed by the pooling layer. The pooling layer employs the mean pooling process of two by two submatrices. The classification neural network consists of the single hidden layer and output layer. This hidden layer has 100 nodes that use the ReLU activation function. Since we have 10 classes to classify, the output layer is constructed with 10 nodes. We use the softmax activation function for the output nodes.

| **Layer** | **Remark** | **Activation Function** |
|---|---|---|
| Input | 28 by 28 nodes | Not there |
| Convolution | 20 convolution filters (9 * 9) | ReLU |
| Pooling | 1 mean pooling (2*2) | Not there |
| Hidden | 100 nodes | ReLU |
| Output | 10 nodes | Softmax |

**Architecture of the Neural Network:**
Although it has many layers, only three of them contain the weight matrices that require training; they are W1, W5, and Wo. W5 and Wo contain the connection weights of the classification neural network, while W1 is the convolution layer’s weight, which is used by the convolution filters for image processing. The input nodes between the pooling layer and the hidden layer are the transformations of the two-dimensional image into a vector.

I have written a function MnistConv.m, which trains the network using the back-propagation algorithm, takes the neural network’s weights and training data and returns the trained weights.

where W1, W5, and Wo are the convolution filter matrix, pooling-hidden layer weight matrix, and hidden-output layer weight matrix, respectively. X and D are the input and correct output from the training data, respectively.

**Explanation of the code and implementation:**
1) The function MnistConv trains the network via the minibatch method. The number of batches, bsize, is set to be 100. As we have a total 8,000 training data points, the weights are adjusted 80 (=8,000/100) times for every epoch. The variable blist contains the location of the first training data point to be brought into the minibatch. Starting from this location, the code brings in 100 data points and forms the training data for the minibatch. Once the starting point, begin, of the minibatch is found via blist, the weight update is calculated for every 100th data point. The 100 weight updates are summed and averaged, and the weights are adjusted. Repeating this process 80 times completes one epoch.

Another aspect of the function MnistConv is that it adjusts the weights using momentum. The variables momentum1, momentum5, and momentum0 are used here.

2) Now, let’s look at the learning rule, the most important part of the code. ConvNet also employs back-propagation training. The first thing that must be obtained is the output of the network. It can be intuitively understood from the architecture of the neural network. The variable y of this code is the final output of the network. Now that we have the output, the error can be calculated. As the network has 10 output nodes, the correct output should be in a 10*1 vector in order to calculate the error. However, the MNIST data gives the correct output as the respective digit. For example, if the input image indicates a 4, the correct output will be given as a 4. The last part of the process is the back-propagation of the error. As this implementation employs cross entropy
and softmax functions, the output node delta is the same as that of the network output error. The next hidden layer employs the ReLU activation function. There is nothing particular there. The connecting layer between the hidden and pooling layers is just a rearrangement of the signal.

3) We have two more layers to go: the pooling and convolution layers. The following listing shows the back-propagation that passes through the pooling layer-ReLU-convolution layer. This code performs the convolution operation using conv2, a built-in two-dimensional convolution function of MATLAB. The function MnistConv also calls the function Pool, which is implemented in the function Pool.m . This function takes the feature map and returns the image after the 2 *2  mean pooling process.

4) The driver function is TestMnistConv.m file, which tests the function MnistConv.m . This program calls the function MnistConv and trains thenetwork three times. It provides the 2,000 test data points to the trained network and displays its accuracy. The test run of this example yielded an accuracy of around 93.5 percent. The original dimension of the MNIST image is 28 * 28. Once the image is processed with the 9*9 convolution filter, it becomes a 20´20 feature map* As we have 20 convolution filters, the layer produces 20 feature maps. Through the 2*2  mean pooling process, the pooling layer shrinks each feature map to a 10 *10 map. The final result after passing the convolution and pooling layers is as many smaller images as the number of the convolution filters; ConvNet converts the input image into the many small feature maps

Here are the files attached along with this explanation pdf:

1) TestMnistConv.m ----main driver function. Run this to start the implementation. The accuracy of the implementation is shown here.
2) MnistConv.m---This contains the code of the neural network used for this implementation.
3) Pool.m—Contains the code for pooling layer.
4) Softmax.m--- Contains the code for the Softmax layer.
5) ReLU---- Contains the code for the ReLU activation function.
6) loadMNISTImages.m---It loads the images of the MNIST dataset.
7) loadMNISTLabels.m---- It loads the labels of the MNIST dataset.

The comparison between of the accuracy using the synthetic dataset given in the tutorial and the one used in this implementation taking the MNIST dataset is as follows:

| **Implementation** | **Accuracy** |
|---|---|
| Using synthetic dataset | 98-99% |
| Using MNIST dataset | 93% |

The implementation used in the tutorial is much more efficient.

References:
1) Book: Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville
2) Book: Deep Learning with Machine Learning, Neural Networks and Artificial Intelligence by Phil Kim Apress Publication
3) Andrew NG's videos on youtube on Sparse Encoders
4) http://hips.seas.harvard.edu/