CSE 573
HW4 Report
Enze Qian
50091378

Q1 How does an autoencoder detect errors?

It uses cost function to measure the error between the input x and its reconstruction at the output ^x. The cost function for training a sparse autoencoder is an adjusted mean squared error function as follows:

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

Reference: https://www.mathworks.com/help/nnet/ref/trainautoencoder.html

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

Because it makes autoencoder to reduce the dimension of data. Making subsequent layers have fewer nodes is a common way to add constraints in order to make this non-trivial. Otherwise, we will get a perfect reconstruction, that is not what we want.

Q3 Why are autoencoders trained on hidden layer at a time?

Because we need to use the result(features) generated from first hidden layer as the training data when we train the next autoencoder (next hidden layer).

Q4 How were the feature in Figure 3 obtained? Compare the method of identifying features here with the method of HW1. What are few pros and cons of these methods?

The feature in Figure 3 is the result from running autoencoder on first layer. At beginning, weights are randomly initialized. The first autoencoder takes initial weight and bias to compute new weights, biases and activation values as follows:

$$a^{(l)} = f(z^{(l)})$$

$$z^{(l+1)} = W^{(l,1)}a^{(l)} + b^{(l,1)}$$

Here a stands for activation value, f stands for activation function, W stands for weight and b stands for bias.

After the first autoencoder is trained, this tutorial calls function plotWeights on the trained autoencoder to get Figure 3. plotWeights plots a visualization of the weights for the encoder of an autoencoder.

The method that we used in HW1 to get features is Spatial Pyramid Matching (Bag of words). The advantage of autoencoder is that we can use unlabeled data to do pre-training since it is unsupervised learning; However, it requires many parameters and overfitting may occur. For Bag of words model, it works great when I was doing HW1 if two scenes have obvious differences since it counts the occurrence of visual words and generates histogram. However, the performance is poor if two images are similar. For example, mountain and garden.

Reference:
http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
https://www.mathworks.com/help/nnet/ref/autoencoder.plotweights.html

Q5 What does the function plotconfusion do?

Plotconfusion returns a confusion matrix plot for the target and output data in targets and outputs, respectively.

Reference:
https://www.mathworks.com/help/nnet/ref/plotconfusion.html

Q6 What activation function is used in the hidden layers of the MATLAB tutorial?

It uses Logistic sigmoid function as default, which is

$$f(z) = \frac{1}{1 + e^{-z}}$$

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

Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
ReLU: $f(x) = \max(0, x)$

First, it is obvious that there is more complex computation when we use Sigmoid since exponential operation is involved.

Second, consider the function max (0, x), it makes the output of some nodes become zero. It will sparse the activation.

Third, using Sigmoid function may cause gradient vanishing/exploding problem(see reference, page 12).

Reference: https://cs.stanford.edu/~quocle/tutorial2.pdf

Q8 The MATLAB demo uses a random number generator to initialize the network. Why is not a good idea to initialize a network with all zeros? How about all one, or some other constant value?

Initializing random value is used for symmetry breaking. If a network is initialized with all zeros, we cannot propagate the gradients. Consider following equations:

One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$
$$b_i^{(l)} := b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$

We cannot compute partial derivatives on zero.

If it is initialized with all ones or other identical values, then all nodes will learn the same thing because all of W will be the same so that a will be the same for all inputs x.

Reference: https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf

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?

Batch gradient descent

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

Reference: http://cs229.stanford.edu/notes/cs229-notes1.pdf

Pros: Consider the function above, it is obvious that it will give us a global optimal solution(global minimum).

Cons: As addressed in the HW description, this method needs all the examples in the training set. Training time will be slow if training set is large.

Stochastic gradient descent

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

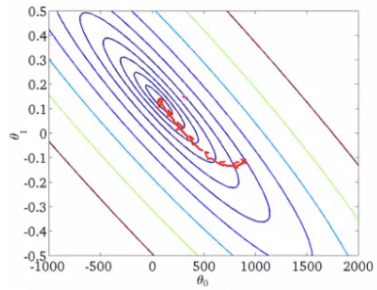
}

Reference: http://cs229.stanford.edu/notes/cs229-notes1.pdf

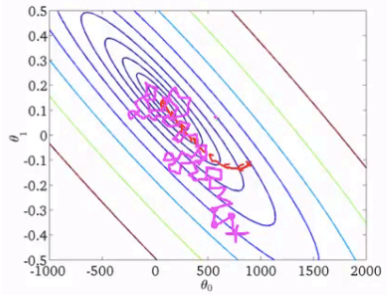
Pros: SGD is used for solving the timing problem in batch gradient descent. The gradient is computed with one sample, which gives us a faster training time.

Cons: As its name stochastic, it has lower accuracy and it will not give us a global optimal solution because it may not converge to the minimum.

- As we saw, batch gradient descent does something like this to get to a global minimum



- With stochastic gradient descent every iteration is much faster, but every iteration is flitting a single example

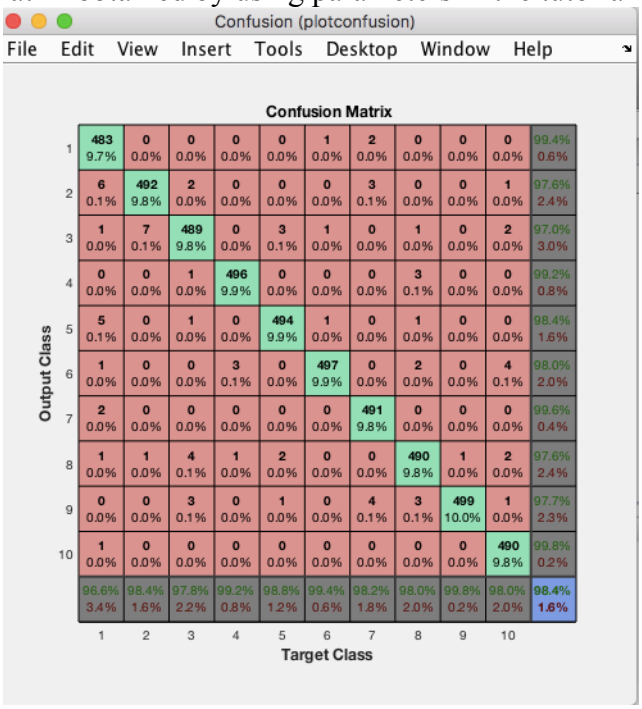


Reference for plots:
https://doc.plob.org/machine_learning/17_Large_Scale_Machine_Learning.html

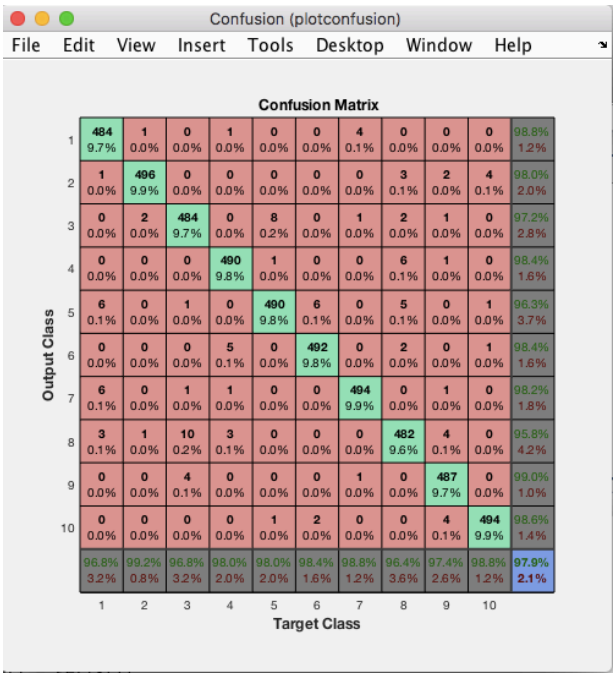
As we can see in above two plots, batch gradient descent takes few iterations and converge to the minimum, stochastic gradient descent takes more iterations and can only get very close to the minimum. Hence, in terms of number of iterations, batch gradient descent runs faster. In terms of number of epoch, stochastic gradient descent runs faster because one epochs equals to one forward and backward pass of all training data and SGD randomly chooses sample to train.

Q10 Report the impact of slightly modifying the parameters.

The confusion matrix obtained by using parameters in the tutorial is given below:

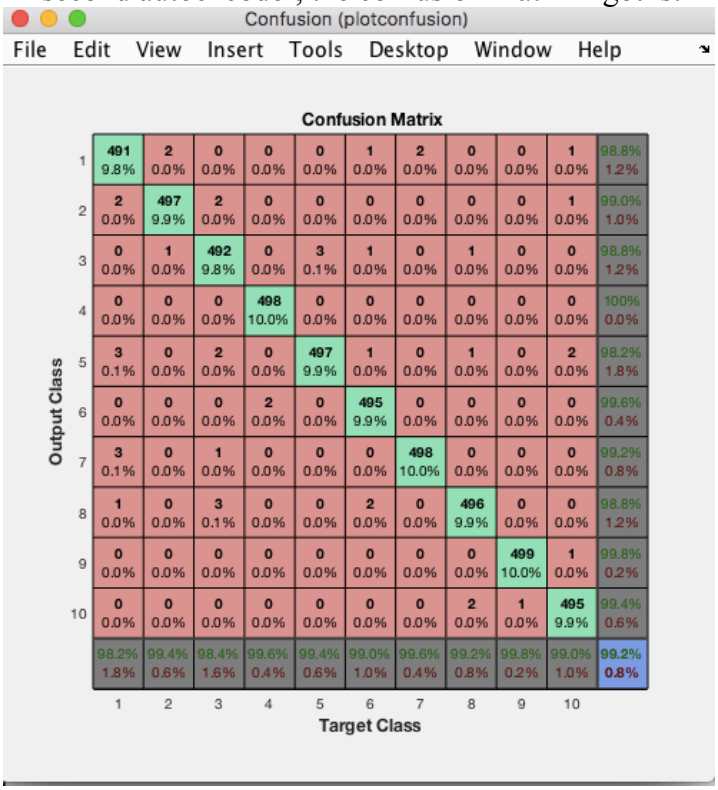


Now I want to change the size of two hidden layers and see if it impacts the confusion matrix. The original hiddensize1=100, hiddensize2=50, I try hiddensize1=50, hiddensize2=25 and keep all other parameters remain the same.Then the confusion matrix I got is:



The original accuracy is 98.4%, the new accuracy is 97.9%. The changing of hidden layer size doesn't impact the result too much.

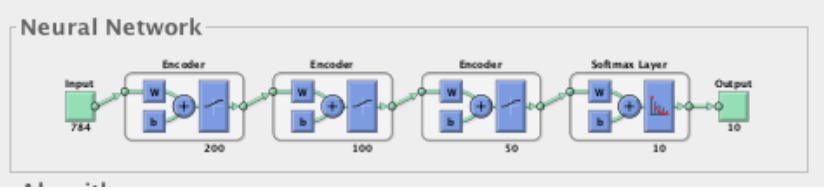
Next, I want to change the SparsityProportion parameter, I set 0.4 in first antoencoder and 0.2 in second autoencoder, the confusion matrix I got is:



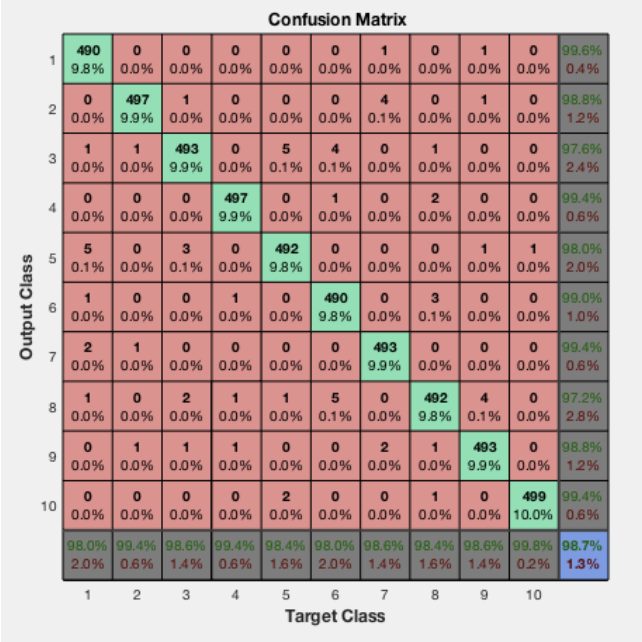
With new value of SparsityProportion, the autoencoder even gives me better accuracy 99.2%.

Finally, I want to add one more layer in my network. To do so, I need adjust the size for both three hidden layer because if I keep using hiddensize1=100 and hiddensize2=50, it will achieve the minimum in the second hidden layer so that the third layer is not needed. So I choose hiddensize1=200, hiddensize2=100 and hiddensize3=50. Then I construct my autoencoder3 (also adjust softmax layer training and stacked neural network formation) as follows:

```
hiddenSize3 = 50;
autoenc3 = trainAutoencoder(feat2,hiddenSize3, ...
    'MaxEpochs',100, ...
    'L2WeightRegularization',0.002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);

feat3 = encode(autoenc3,feat2);

softnet = trainSoftmaxLayer(feat3,tTrain,'MaxEpochs',400);

deepnet = stack(autoenc1,autoenc2,autoenc3,softnet);
```



The result is: Accuracy 98.7%



In conclusion, I tried to change some parameters in the tutorial and it didn't impact the result. I believe the tutorial presentation is robust with respect to parameter settings.