

Exercise: Implement deep networks for digit classification

From Ufldl


Contents

- 1 Overview
- 2 Dependencies
- 3 Step 0: Initialize constants and parameters
- 4 Step 1: Train the data on the first stacked autoencoder
- 5 Step 2: Train the data on the second stacked autoencoder
- 6 Step 3: Train the softmax classifier on the L2 features
- 7 Step 4: Implement fine-tuning
- 8 Step 5: Test the model

Overview


In this exercise, you will use a stacked autoencoder for digit classification. This exercise is very similar to the self-taught learning exercise, in which we trained a digit classifier using an autoencoder layer followed by a softmax layer. The only difference in this exercise is that we will be using two autoencoder layers instead of one and further finetune the two layers.

The code you have already implemented will allow you to stack various layers and perform layer-wise training. However, to perform fine-tuning, you will need to implement backpropagation through both layers. We will see that fine-tuning significantly improves the model's performance.

In the file `stackedae_exercise.zip` (http://ufldl.stanford.edu/wiki/resources/stackedae_exercise.zip) , we have provided some starter code. You will need to complete the code in `stackedAECost.m`, `stackedAEPredict.m` and `stackedAEEExercise.m`. We have also provided `params2stack.m` and `stack2params.m` which you might find helpful in constructing deep networks.

Dependencies

The following additional files are required for this exercise:

- MNIST Dataset (<http://yann.lecun.com/exdb/mnist/>)
- Support functions for loading MNIST in Matlab
- Starter Code (`stackedae_exercise.zip`) (http://ufldl.stanford.edu/wiki/resources/stackedae_exercise.zip) 

You will also need your code from the following exercises:

- Exercise: Sparse Autoencoder
- Exercise: Vectorization
- Exercise: Softmax Regression
- Exercise: Self-Taught Learning

If you have not completed the exercises listed above, we strongly suggest you complete them first.

Step 0: Initialize constants and parameters

Open `stackedAEEExercise.m`. In this step, we set meta-parameters to the same values that were used in previous exercise, which should produce reasonable results. You may to modify the meta-parameters if you wish.

Step 1: Train the data on the first stacked autoencoder

Train the first autoencoder on the training images to obtain its parameters. This step is identical to the corresponding step in the sparse autoencoder and STL assignments, complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

Step 2: Train the data on the second stacked autoencoder

We first forward propagate the training set through the first autoencoder (using `feedForwardAutoencoder.m` that you completed in Exercise: Self-Taught_Learning) to obtain hidden unit activations. These activations are then used to train the second sparse autoencoder. Since this is just an adapted application of a standard autoencoder, it should run similarly with the first. Complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

This part of the exercise demonstrates the idea of greedy layerwise training with the *same* learning algorithm reapplied multiple times.

Step 3: Train the softmax classifier on the L2 features

Next, continue to forward propagate the L1 features through the second autoencoder (using `feedForwardAutoencoder.m`) to obtain the L2 hidden unit activations. These activations are then used to train the softmax classifier. You can either use `softmaxTrain.m` or directly use `softmaxCost.m` that you completed in Exercise: Softmax Regression to complete this part of the assignment.

Step 4: Implement fine-tuning

To implement fine tuning, we need to consider all three layers as a single model. Implement `stackedAECost.m` to return the cost and gradient of the model. The cost function should be as defined as the log likelihood and a gradient decay term. The gradient should be computed using back-propagation as discussed earlier. The predictions should consist of the activations of the output layer of the softmax model.

To help you check that your implementation is correct, you should also check your gradients on a synthetic small

dataset. We have implemented `checkStackedAECost.m` to help you check your gradients. If this checks passes, you will have implemented fine-tuning correctly.

Note: When adding the weight decay term to the cost, you should regularize only the softmax weights (do not regularize the weights that compute the hidden layer activations).

Implementation Tip: It is always a good idea to implement the code modularly and check (the gradient of) each part of the code before writing the more complicated parts.

Step 5: Test the model

Finally, you will need to classify with this model; complete the code in `stackedAEPredict.m` to classify using the stacked autoencoder with a classification layer.

After completing these steps, running the entire script in `stackedAETrain.m` will perform layer-wise training of the stacked autoencoder, finetune the model, and measure its performance on the test set. If you've done all the steps correctly, you should get an accuracy of about 87.7% before finetuning and 97.6% after finetuning (for the 10-way classification problem).

From Self-Taught Learning to Deep Networks | Deep Networks: Overview | Stacked Autoencoders | Fine-tuning Stacked AEs |
Exercise: Implement deep networks for digit classification

Retrieved from

"http://ufddl.stanford.edu/wiki/index.php/Exercise:_Implement_deep_networks_for_digit_classification"

- This page was last modified on 26 May 2011, at 11:04.