Exercise 9: Multilayer Neural Network

Lecture Information Processing and Communication

Jörn Anemüller, June 2022

Submit solutions until Tuesday 2022-06-21, 23:59h, by uploading to your group's exercise folder on cs.uol.de. You may submit your solutions in groups of at most two students. You are free to write your code in matlab or in python (but we provide the example functions in matlab only).

Summary of exercise 9

The goal of this exercise is to implement a three-layer feedfoward neural network for the task of digit recognition. The suggested structure of the code follows that of the previous assignments on logistic regression. You are free to build your code on the provided templates for this exercise 9 or extend your previous code from previous assignments to implement the neural network.

List of functions to be completed

ex09_script.m: main script for completion of exercise 9

nn_forward.m: run forward-propagation through network and return the vector with class-indices of highest output activation for each test sample

nn_gradient.m: compute loss-function and gradient of 3-layer neural network

nn_gradient_ls.m: compute least-squares loss-function and gradient of 3-layer neural network

nn_gradient_reg.m: compute regularized loss-function and gradient of 3-layer neural network

List of provided helper functions

checkgrad.m: check gradient computation for correctness

digit_data.mat: handwritten digits dataset, rescaled to the interval 0....1

labelvec2mat.m: convert numeric labels in vec_y and vec_y_train to binary output neuron representation

minimize.m: function minimizer based on conjugate gradient descent method, replaces the "simple" gradient we used before

paramvec2mat.m: convert the single parameter vector vec_w to two matrices connecting the layers

sigmoid.m: non-linear sigmoid activation function

1. Main script for this exercise session

Edit the script `ex09_script.m`, it contains template code for the main steps of this exercise.

Note 1: The self-implemented gradient descent iteration from the previous exercise session has been replaced by the use of an optimization function, minimize.m, that performs the conjugate gradient variant of gradient descent with an adaptive step size and a line-search in the gradient direction. Check out the provided template code to familiarize yourself with the way the opmizer is set-up and how it accesses your self-written loss- and gradient-function.

Note 2: Correctness of your implementation of the gradient-function can be ascertained through the checkgrad.m function, that is also provided. It computes a numeric approximation to the gradient based on your loss-function and compares results obtained from the implemented gradient function with this approximation. The difference is returned in measure *d* that should generally be rather small, e.g., *10e-6* or smaller.

Note 3: The MNIST data from the previous exercise have been rescaled to the range [0,1] and the large number of constant-zero-amplitude pixels has been removed by adding a small constant. Problems with NaN values during the computation should (hopefully) not occur with this scaling.

2. Classification with a neural network with given (fixed) weights

Edit the file `nn_forward.m` to predict the best matching labels for a given set of (fixed) weights. Notation and implemented equations are in corrspondence to the lecture notes.

Function definition:

```
function vec_bestclass = nn_forward(vec_w, mat_X, NInput, NHidden, NOutput)
```

3. Learning in a neural network with the cross-entropy cost function

Edit the file `nn_gradient.m` to define cost function and gradient for backpropagation gradient descent based on the cross-entropy loss function.

The cross-entropy loss function and its gradient are given (following the same steps as done for the least-squares cost in the lecture notes) by

$$L(w_{ij}^{(1)}, w_{ij}^{(2)}) = \frac{1}{d^{(2)}} \sum_{i=1}^{d^{(2)}} E\left[-y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)\right] \qquad \text{with} \quad \hat{y}_i = a_i^{(2)}$$

and

$$\begin{aligned} \frac{\partial L}{\partial w_{ij}^{(2)}} &= E\left[\delta_i^{(2)} a_j^{(1)}\right] \\ \frac{\partial L}{\partial w_{ij}^{(1)}} &= E\left[\delta_i^{(1)} a_j^{(0)}\right] \\ \delta_i^{(2)} &= \hat{y}_i - y_i \\ \delta_j^{(1)} &= \sum_{i=1}^{d^{(1)}} w_{ij}^{(2)} \delta_i^{(2)} f'(z_j^{(1)}) \end{aligned}$$

where $d^{(l)}$ denotes number of outputs in layer l, E denotes expectation, y_i denotes target (true) class label, and $\hat{y}_i = a_i^{(2)}$ denotes activity of the i-th output neuron.

Function definition:

```
function [L, vGrad] = nn_gradient(vec_w, mat_X, mat_y, NInput, NHidden, NOutput)
```

4. Learning in a neural network with the least-squares cost function

Edit the file `nn_gradient_ls.m` to define cost function and gradient for backpropagation gradient descent based on the least-squares loss function.

Train the classifier on the handwritten digits problem and compare its performance to the cross-entropy classifier.

Function definition:

```
function [L, vGrad] = nn_gradient_ls(vec_w, mat_X, mat_y, NInput, NHidden, NOutput)
```

5. Learning in a neural network with the cross-entropy cost function and an additive regularization term

Edit the file `nn_gradient_reg.m` to define cost function and gradient for backpropagation gradient descent based on the cross-entropy loss function with an additive regularization term.

Note that the term with the l-2-norm regularization contributes to value of the (total) loss function and to its (easily computed) gradient.

Compare the classification accuracy (on test data) for several values of $\lambda \geq 0$.

Function definition:

```
function [L, vGrad] = nn_gradient_reg(vec_w, mat_X, mat_y, NInput, NHidden, NOutput, Lambda)
```