Assignment Set 5 - Machine Learning

Instructor: Dr. Enrique Hortal

December 3, 2020

Group assignment

These assignments should be handed in and defended individually.

Handing in

Upload your code for Artificial Neural Networks to the Canvas' assignment in a single zip archive. In addition, upload a very simple report in the form of a PDF. This document must include only an explanation (a brief explanation of the experiments performed, if applicable) of the concepts proposed in Section 2 Evaluation of this document. Submissions by email or other types of archives are not accepted. Thank you for your understanding. During the following assignment session, you will explain your implementation to the examiners.

1 Graded Assignment: Artificial Neural Networks

In this lab you will implement feedforward prediction and backpropagation for Artificial Neural Networks. The implementation is expected to be made in Python. To do so, a start code is provided. To get started with the exercise, download the starter code and unzip (using WinRar, WinZip or a similar software) its contents to the directory where you wish to complete the exercise.

Files included in the starter code: (* indicates files you will need to complete)

ex_nn - Script that will help you step through the exercises
digitdata.mat - Training set of hand-written digits
debugweights.mat - Previously learned weights to debug feedforward
displayData - Function to help visualize the dataset
checkNNGradients - Function to help check your gradients
fmincg - Function minimization routine
computeNumericalGradient - Numerically compute gradients
debugInitializeWeights - Function for initializing weights
sigmoid - Sigmoid function

* **predict** - Neural network prediction function
* **sigmoidGradient** - Compute the gradient of the sigmoid function
* **randInitializeWeights** - Randomly initialize weights
* **nnCostFunction** - Neural network cost function

Throughout the exercise, you will be using the script **ex_nn**. This script sets up the dataset and makes calls to functions that you will write. You do not need to modify this script (except to see extra output). You are only required to modify functions in other files, by following the instructions in this assignment.

The data

You are given a data set in **digitdata.mat** that contains 5000 training examples of handwritten digits[1]. The .mat format means that the data has been saved in a native Octave/Matlab matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the `load` command (Matlab) or using the `loadmat` function from the scipy.io library (Python). After loading, matrices of the correct dimensions and values will appear in your programs memory. When using Matlab, the matrix will already be named, so you do not need to assign names to them. There are 5000 training examples in **digitdata.mat**, where each training example is a 20 pixel by 20 pixel grayscale image of a digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is unrolled into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix *X*. This gives us a 5000 by 400 matrix *X* where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} -(x_1)^T- \\ -(x_2)^T- \\ \dots \\ -(x_m)^T- \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector *y* that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, the digit zero is mapped to the value 10. Therefore, a 0 digit is labeled as 10, while the digits 1 to 9 are labeled as 1 to 9.

First you can visualize a subset of the training set. In Part 1 of **ex_nn**, the code randomly selects 100 rows from *X* and passes those rows to the **displayData** function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. The **displayData** function has been provided for you in the starter code. After you run this step, you should see an image like Figure 1.

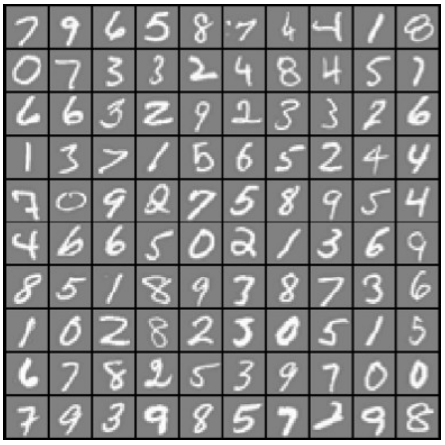

Figure 1: Examples from the digits dataset

[1] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/)

The network structure

The neural network to be used is shown in Figure 2. It has three layers — an input layer, a hidden layer and an output layer. Recall that the inputs are pixel values of digit images. Since the images are of size 20x20, this results in 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the data will be loaded into the variables *X* and *y*.

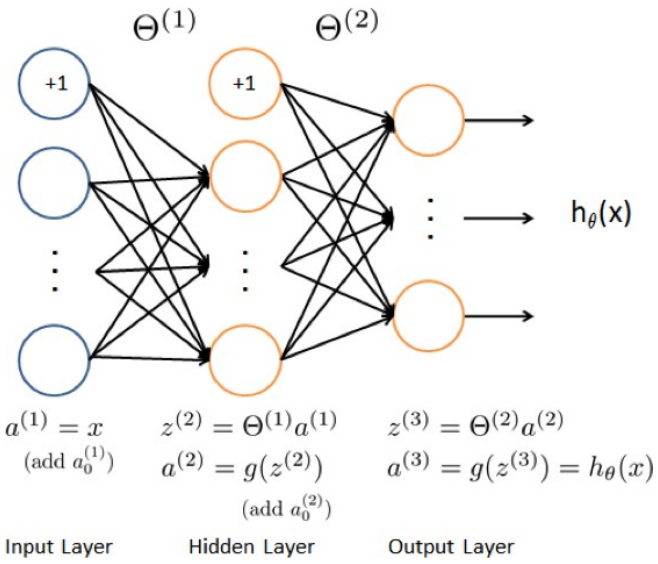

Figure 2: The neural network

You have been provided with a set of network parameters ($\Theta^{(1)}$, $\Theta^{(2)}$). These are stored in **`debugweights.mat`** and will be loaded by **`ex_nn`** into Theta1 and Theta2. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

Feedforward propagation — Prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in **`predict`** to return the neural networks predictions. You should implement the feedforward computation that computes $h_\Theta(x^{(i)})$ for every example $x^{(i)}$ in X and returns the associated prediction, i.e. the label of the node on the output layer that has the highest activation.

Implementation Note: The matrix X contains the examples in **rows**. When you complete the code in **`predict`**, you will need to add a column of 1s to the matrix. The matrices Theta1 and Theta2 contain the parameters for each unit in **rows**. Specifically, the first row of Theta1 corresponds to the first hidden unit in the second layer. When you compute $z^{(2)} = \Theta^{(1)} a^{(1)}$, be sure that you index (and if necessary, transpose) X correctly so that you get $a^{(l)}$ as a column vector.

Once you are done, **`ex_nn`** will call your predict function using the loaded set of parameters for Theta1 and Theta2. You should see that the accuracy is about 97.5%. After that, you can choose to run an interactive sequence displaying images from the training set one at a time, while the console prints out the predicted label for the displayed image. To stop the image sequence, press Ctrl-C (or press enter 5000 times, your choice ;)).

Backpropagation — Learning

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the **`nnCostFunction`** so that it returns an appropriate value for grad. Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as **`fmincg`** (i.e., an advanced gradient descent algorithm that uses "Conjugate Gradients" to speed up learning).

Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as

$$\frac{d}{dx} g(x) = g(x)(1 - g(x))$$

where

$$g(x) = \frac{1}{1 + e^{-\theta^T}}$$

When you are done, try testing a few values by calling **`sigmoidGradient(z)`** at the command line. For large values (both positive and negative) of z, the gradient should be close to 0. When z = 0, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$.

You should use $\epsilon_{init} = 0.12$. (See footnote[2] for details.) This range of values ensures that the parameters are kept small and makes the learning more efficient. Your job is to complete **`randInitializeWeights.m`** to initialize the weights. If needed, ask MatLab/Octave for help on the build-in rand function or check numpy documentation for Python implementation.

Computing the gradient

Now, you will implement the backpropagation algorithm, i.e. the computation of the δ's for each non-input node of the network. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, a forward pass computes all the activations throughout the network, including the output value of the hypothesis $h\Theta(x)$. Then, for each node j in layer l, an error term $\delta_j^{(l)}$ is computed that measures how much that node was responsible for any errors in the output. For an output node, the difference between the networks activation and the true target value directly defines $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

[2] One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of $\epsilon_{\text{init}} = \frac{\sqrt{6}}{\sqrt{(L_{in} + L_{out})}}$ where L_{in} and L_{out} are the number of units in the layers adjacent to $\Theta^{(l)}$.

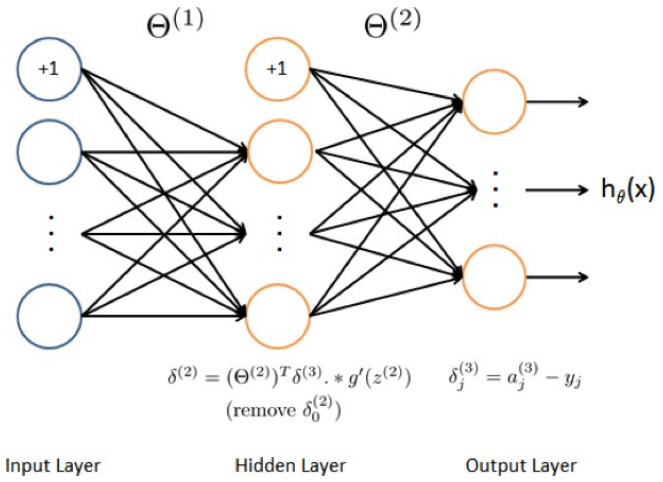

Figure 3: Backpropagation

In detail, here is the backpropagation algorithm (also depicted in Figure 3). In the file **`nnCostFunction.m`**, you should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop for **t**=1 to t=m and place steps 1–4 below inside the for-loop, with the t^{th} iteration performing the calculation on the t^{th} training example ($x^{(t)}$, $y^{(t)}$). Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layers values ($a^{(1)}$) to the t^{th} training example $x^{(t)}$. Perform a feedforward pass (Figure 2), computing the activations ($z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)}$) for layers 2 and 3. Note that you need to add a +1 term to ensure that the vectors of activations for layers $a^{(1)}$ and $a^{(2)}$ also include the bias unit. In MatLab/Octave, if a_1 is a column vector, adding one corresponds to $a_1 = [1; a_1]$. In Python, you can use $a_1 = \text{numpy.append}([1], a_1)$.
2. For each output unit k in layer 3 (the output layer), set $\delta^{(3)} = (a^{(3)} - y_k)$, where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class k ($y_k = 1$), or if it belongs to a different class ($y_k = 0$).
3. For the hidden layer $l = 2$, set $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta^{(2)}$. In Matlab/Octave, removing $\delta^{(2)}$ corresponds to delta2=delta2(2:end). For Python, it will be delta2=delta2[1:]

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

After you have implemented the backpropagation algorithm, the script **ex_nn** will proceed to run gradient checking on your implementation. The gradient check — which is based on a numerical approximation of the gradient — will allow you to increase your confidence that your code is computing the gradients correctly.

Adding regularization

In file **`nnCostFunction`** you can also see the regularization to the gradient. To account for regularization, it turns out that we can add this as an additional term after computing the gradients using backpropagation.

Specifically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you can add regularization using:

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \qquad \text{for } j = 0$$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\theta_{ij}^{(l)} \qquad \text{for } j \geq 1$$

Note that we don't regularize the first column of $\theta^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus,

$$\theta^{(l)} = \begin{bmatrix} \theta_{1,0}^{(l)} & \theta_{1,1}^{(l)} & \dots \\ \theta_{2,0}^{(l)} & \theta_{2,1}^{(l)} & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

Somewhat confusingly, indexing in Matlab/Octave starts from 1 (for both i and j), thus Theta1(2,1) actually corresponds to $\theta_{2,0}^{(l)}$ (i.e., the entry in the second row, first column of the matrix $\theta^{(l)}$ shown above).

If your code is correct, you should expect to see a relative difference that is less than 1e-9.

Learning the parameters

When you have successfully implemented the neural network gradient computation, you can run the rest of **`ex_nn`** which will use **`fmincg`** to learn a good set of parameters. After training completes, **`ex_nn`** reports the **training** accuracy of your classifier.

The script will also show the features captured by the hidden layer by showing the 400 weights connecting the input nodes to each of the 25 hidden nodes as 20x20 images. In your trained network, you should find that the hidden units corresponds roughly to detectors that look for strokes and other patterns in the input.

Feel free to experiment with different values of the regularization parameter *λ* and the number of iterations to see if you can get the network to overfit the learning data.

2 Evaluation

During the discussion with the examiners, the following concepts could be evaluated:

- Description of the network (e.g. number of layers, number of neurons per layer, etc.)
- Impact of specific parameters such as λ, number of iterations, weight initialization, etc.
- How does the regularization affect the training of your ANN?
- Did you manage to improve the initial results (using values in *debugweights.mat*)? Which was your best result? How did you configure the system? How could you improve them even more?
- Imagine that you want to use a similar solution to classify 50x50 pixel grayscale images containing letters (consider an alphabet with 26 letters). Which changes would you need in the current code in order to implement this classification task?
- Change the value of the variable `show_examples` in `ex_nn`, which information is provided? Did you get the expected information? Is anything unexpected there?
- How does your sigmoidGradient function work? Which is the return value for different values of z? How does it works with the input is a vector and with it is a matrix?