2EL1730: MACHINE LEARNING

CENTRALESUPÉLEC

Lab 6: Neural Networks for Hand-written Digit Recognition

Instructors: Nora Ouzir and Maria Vakalopoulou
TAs: Theodore Aouad, Hakim Benkirane, Aaron Mamann, Lily Monnier

January 6, 2022

1 Description

The goal of the lab is to implement a general purpose Neural Network and to apply it to the task of hand-written digit recognition. Nowadays, automated handwritten digit recognition is widely used – from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. For the lab needs, we will use the MNIST database[1] which consists of digits written by high school students and employees of the United States Census Bureau. Actually, the MNIST dataset consists of handwritten digit images ($0 - 9$) and it is divided in 60.000 examples for the training set and 10.000 examples for testing. Figure 1 represents a randomly selected instance of the dataset.

Figure 1: Example of one hand-written digit of the training set.

All digit images have been size-normalized and centered in a fixed size image of 28×28 pixels. Each pixel of the image is represented by a value in the range of $[0, 255]$, where 0 corresponds to black, 255 to white and anything in between is a different shade of grey. In our case, the pixels are normalized and constist the features of our dataset; therefore, each image (instance) has 784 features. That way, the training set has dimensions $60,000 \times 784$ and the test set $10,000 \times 784$. Regarding the class labels, each figure (digit) belongs to the category that this digit represents (e.g., digit 2 belongs to category 2).

[1]The MNIST database: `http://yann.lecun.com/exdb/mnist/`.

2 Neural Networks - Pipeline

The main goal of the lab is to give us the opportunity to examine and analyze the behavior of different Neural Networks structures. In general terms, you should examine and compare the performance of Neural Networks with different number of *hidden layers* and *units* per layer, in the problem of hand-written digit recognition.

For notation simplicity, let's assume a neural network with only 3 layers (an input layer, a hidden layer and an output layer) of the following structure:

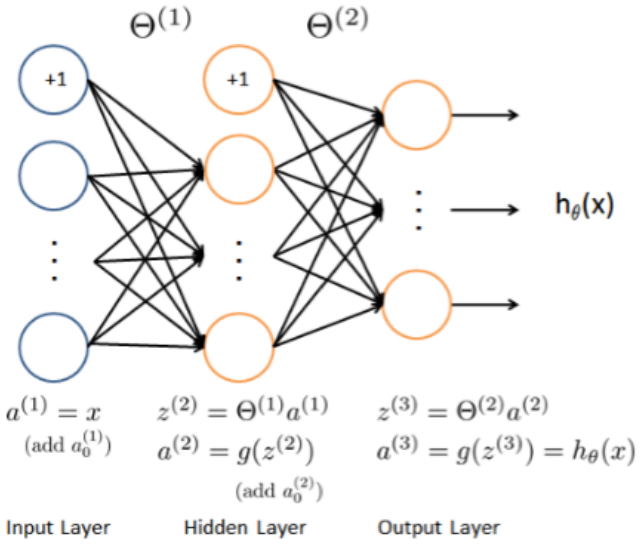

Figure 2: Neural Network with one hidden layer

where x is the input and $\theta = (\Theta_1, \Theta_2)$ is a set of network parameters.

Note: Please take into account that your code should work for any number of hidden layers and any number of units per layer.

2.1 Loading and Visualizing the MNIST Dataset

In the first part, the code will load the MNIST dataset, calling the function `read_dataset.py`. This returns the variables `images_training` and `labels_training` which contain the training instances and their class labels, respectively. In a similar way, the test data and their class are loaded in the variables `images_test` and `labels_test`, respectively. Recall that there are $60,000$ examples for the training set and $10,000$ examples for testing, where each instance is a 28×28 pixels grayscale image of the digit. Each pixel of the image is represented by a floating point number in the range of $[0, 1]$ indicating the grayscale intensity at that location. The pixels of each image are "unrolled" into a 784-d vector. In this way, each one of these examples becomes a single row in our data matrices. Finally, we display 100 randomly selected instances-digits on a 2-d plot (Figure 3) by calling the script `displayData`.

Note: In order to keep the complexity reasonable, use a lower number of instances for training. This can be achieved by changing the value of the variable `size_training` which is located in the main.py script.

2.2 Setting up the Neural Network Structure

In the second part, you are asked to set up the structure of the Neural Network under which you intend to train for the hand-digit recognition task. Actually, you will be asked to select both the number of hidden layers as well as the number of nodes for each of them.

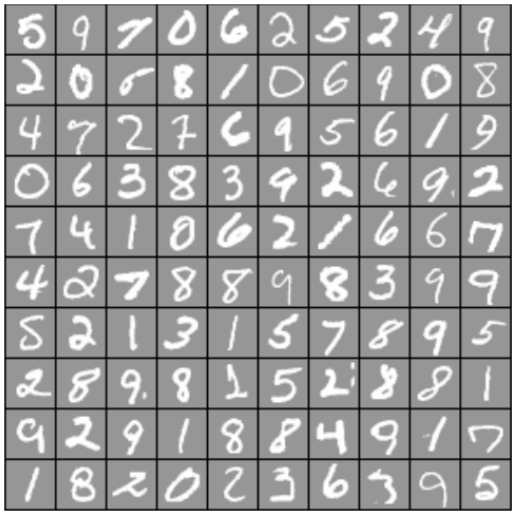

Figure 3: Example from the MNIST dataset.

Then, you should randomly initialize the neural network's parameters. An effective way is to select random values for the parameters uniformly in the range $[-\epsilon, \epsilon]$, where ϵ is equal to a small value (e.g., $\epsilon = 0.12$).

Tasks to be Performed

- Fill in the `randInitializeWeights.py` in order to initialize the neural network weights.
- Examine why it is important to randomly initialize the neural network's parameters. What happens in the case where we initialize to zero all neural network's parameters?

2.3 Sigmoid Function

In the third part, you should implement the sigmoid function which can be computed as follows:

$$\text{sigmoid}(z) = g(z) = \frac{1.0}{1.0 + \exp(-z)} \tag{1}$$

Tasks to be Performed

- Fill in the `sigmoid.py` function in order to implement the sigmoid function.

Note: When $z = 0$, the sigmoid should exactly be equal to 0.5.

2.4 Sigmoid Gradient Function

In the fourth part, you should implement the sigmoid gradient function which can be computed as:

$$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z)), \tag{2}$$

where g(z) is given by Eq. 1.

Tasks to be Performed

- Fill the `sigmoidGradient.py` function in order to implement the sigmoid gradient function.

Note: For large absolute values of $|z|$, the gradient should be close to 0. When $z = 0$, the gradient should exactly be equal to 0.25.

2.5 FeedForward and Cost Function

In the fifth part, you should implement the cost function of the neural network, which calculate and return the cost. The cost function of a neural network is calculated according to the following formula:

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right], \tag{3}$$

where $K = 10$ is the number of possible labels and m is the total number of training instances. Note that $h_\theta(x^{(i)})_k$ represents the activation of the k^{th} output unit.

Tasks to be Performed

- Fill in the `costFunction.py` function that return the cost of the neural network. Roughly speaking, implement the feedforward computation that computes $h_\theta(x^{(i)})_k$ for every instance i and sum the cost over all examples.

Note: You need to recode the image labels as vectors containing only values 0 or 1. For example, if the corresponding label of instance $x^{(i)}$ is equal to 3, then the $y^{(i)}$ should be a 10-dimensional vector with $y_3^{(i)} = 1$, and all the other elements equal to 0. Moreover, after implementing the cost function, the `main.py` script will execute the `checkNNCost.py` script. Thus, you will have the opportunity to examine if your code computing the cost correctly.

2.6 [Optional] FeedForward and Cost Function with Regularization

In the sixth part, you should implement the cost function with regularization of the neural network. The cost function with regularization of a neural network is calculated according to the following formula:

$$\begin{aligned} J(\Theta) = &\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] \\ &+ \frac{\lambda}{2m}\left[\sum_{j=1}^{40}\sum_{k=1}^{784}\left(\Theta_{j,k}^{(1)}\right)^2\sum_{j=1}^{10}\sum_{k=1}^{40}\left(\Theta_{j,k}^{(2)}\right)^2\right], \end{aligned} \tag{4}$$

where $K = 10$ is the number of possible labels and m is the total number of training instances. In the above formula, we assume that the hidden layer composed by 40 nodes.

Tasks to be Performed

- Fill in the `costFunction.py` function by adding the cost for the regularized terms.

Note: If no regularization is used, set variable `lambd` to zero. You should not be regularizing the terms that correspond to the bias. Moreover, after implementing the cost function, the `main.py` script will execute the `checkNNCost.py` script. Thus, you will have the opportunity to examine if your code computes the cost correctly.

2.7 Backpropagation

In this part, you should implement the backpropagation algorithm which calculates the gradient for the cost function of the neural network. Once you have computed the gradient, the gradient is fed to an advanced optimization method which in turn uses it to update the weights, in an attempt to train the neural network by minimizing the cost function. In the following, we present the basic steps of the backpropagation algorithm. You should implement steps 1 to 4 in a loop that processes one example per time.

1. Set the input layer's values $a^{(1)}$ to the the t^{th} training example $x^{(t)}$. Perform a feedforward pass by computing the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$

2. For each output unit k in the outer layer (layer 3), we set

$$\delta_k^{(3)} = \left(a_k^{(3)} - y_k\right) \tag{5}$$

3. For the hidden layer, set

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^\top \delta^{(3)} .* g'(z^{(2)}) \tag{6}$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^\top \tag{7}$$

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

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} \tag{8}$$

Tasks to be Performed

- Fill the `backwards.py` script by implementing the backpropagation algorithm.

Note: After implementing the backpropagation algorithm, the `main.py` script will execute the `checkNNGradient.py` script. Thus, you will have the opportunity to examine if your code computing the gradients correctly.

2.8 [Optional] Backpropagation with Regularization

Now, you should add reguralization to the gradient computing by the backpropagation algorithm. Actually, you can add this as an additional term after computing the gradients using backpropagation, according to the following formulas:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{for } j = 0 \tag{9}$$

and

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \quad \text{for } j \geq 1 \tag{10}$$

Tasks to be Performed

- Fill the `backwards.py` script by adding the reguralization term to the gradient computed at the previous step.

Note that you must not regularize the first column of $\Theta^{(l)}$ which is used for the bias term. After implementing the backpropagation algorithm, the `main.py` script will execute the `checkNNGradient.py` script, with the regularization parameter set to 3. Thus, you will have the opportunity to examine if your code computes the gradients correctly.

2.9 Training Neural Network

After you have successfully implemented the neural network cost function and gradient computation, the next step is to use an advanced optimization for training the neural network in order to discover a good set of parameters. For simplicity purposes, we consider the optimization process as a "black box" (the L-BFGS-B algorithm used for optimization).

Tasks to be Performed

- Fill in the `predict.py` script which takes as input a number of testing instances (`images_test`) and returns a vector of their predicted labels.

- You have to train and examine a number of different neural network schemes (different number of hidden layers and units per layer).
- You have to train and examine the above neural network schemes for more iterations (e.g., set `MaxFun` to 400) and also vary the regularization parameter λ.