CS 438/638

Programming Assignment 5

In this programming assignment, we will implement a **feed-forward neural network**.

Before you start programming:

- Make sure that you have Octave on your system. If not, see https://www.gnu.org/software/octave/download to learn how you can install it.
- Download and extract the assignment zip from Blackboard. Make sure you have the files “test.m”, “run.m”, “data-test.txt”, “digit-data.mat”, and several other .m files in the assignment directory. Do not change the contents of those files unless you are instructed to do so.
- Open a terminal and navigate to the assignment directory using the “cd” command.
- Using the same terminal, run Octave with the “octave” command.
- The “test” and “run” scripts are ready to use the code that you will implement in this assignment. Simply run them by typing the name of the script on the Octave console. The “test” script will initially give errors, but those errors will get resolved as you add the required functions. The “test” script will test your functions on some preset values and will print the expected and calculated values. If those values are significantly different, then please go back to your implementation and fix the error. After you pass the tests, execute the “run” script to apply your code to digit classification.

Load data

The given function “loadData” will get you an X matrix for input training features and a y vector for output classes, but this time the X matrix will not have the added 1s at its first column. Therefore it is not a design matrix. We can call it just “input matrix”. Similar to the design matrix, our input matrix keeps individual input vectors at its rows.

Reformat the output values

Implement the function “prepareY” that will transform the current classification values into the expected outputs of the neural network. This method will take the y vector as input and will return a Y matrix. You can assume the y vector will have values from 1 to K, indicating the class of that specific training example with a single positive integer. The neural network, however, will represent a single output with multiple numbers. If there are K classes in the problem, then there will be K units in the output layer of the neural network, therefore each output is a vector of length K. For instance, assume K = 5 and the given class of the first training example is 2. This means there will be the value “2” at the beginning of the y vector (y(1) is 2) and the maximum value that we will find in the y vector is 5 (max(y) is 5). On the neural network side, the expected output for class 2 is:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

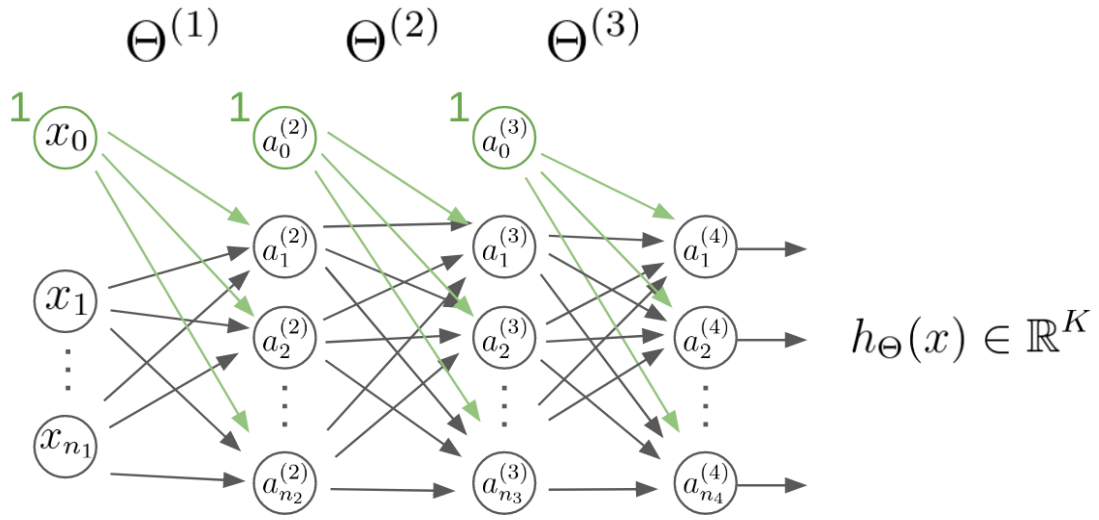
Your task is to make that conversion, considering all the training examples at once. In other words, the y vector will turn into a Y matrix where each expected output is stored on its columns. For example the below y vector should generate the below Y matrix.

$$y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

While implementing this, you may find it useful that comparison operators in octave can work on vectors and matrices. For instance the operation "`1:5 == 2`" will generate "`[0 1 0 0 0]`". You can implement this function anyway you like, but try not to use loops just for fun.

Initialize the Θ matrices

Implement the function "initTheta" that will take the n vector as input and produce the Θ matrices in a cell array. The n vector indicates the number of perceptrons on each layer of the neural network, not counting the bias units. Remember: n_1 has to be equal to the training features and n_L has to be the number of classes K where L is the total number of layers on the network. n = [400 600 10] indicates a 3-layer network with 400 training features, 10 output classes and a single hidden layer with 600 units. For your reference, below is a general depiction of a feed-forward neural network with 4 layers.



You first need to initialize a cell array for the Θ matrices. Make sure it is a single-column cell array. For instance the command "`cell(5, 1)`" generates a single-column cell array with 5 elements. The length of the n vector will guide you in finding the number of Θ matrices. Then you can access individual matrices by giving their index in curly brackets. For instance, if the name of the variable is `Theta`, then `Theta{1}` will address $\Theta^{(1)}$ and so on.

In a loop, initialize each $\Theta^{(l)}$ matrix and assign to its proper location on the cell array. Use the n vector to identify the required dimensions of the matrix. For the level l, $\Theta^{(l)}$ will have n_{l+1} rows and $n_l + 1$ columns.

You will assign random values to the elements in Θ matrices, between -0.1 and 0.1. The built-in function "rand(a, b)" will generate an a×b matrix that has random values in its elements between 0 and 1. You will need to rescale that to make the values spread between -0.1 and 0.1.

Implement the forward propagation

Implement the function "forwardPropagate" that will take the Θ matrices and the input matrix X, and return a cell array of activation matrices.

For a single training example x, the forward propagation generates the $a^{(l)}$ vectors for each layer l on the network. We need to initialize $a^{(1)}$ by assigning x vector to it. Other layers can be calculated in a loop, using the formula $a^{(l+1)} = g(\Theta^{(l)}a^{(l)})$. Our activation vectors will not include the bias units. So we will add those bias units when they are required for a calculation. In the previous equation, the operation is possible only if $a^{(l+1)}$ does not have a bias unit but $a^{(l)}$ has a bias unit. For this equation to work in Octave, we need to use $[1; a^{(l)}]$ instead of $a^{(l)}$.

Since we have more than one training example, instead of the activation vectors $a^{(l)}$, we will calculate the activation matrices $A^{(l)}$ that will have an activation vector for each training example in its columns. $A^{(1)}$ should be initialized to X^T (examples in columns). Then each activation matrix

can be calculated as $A^{(l+1)} = g(\Theta^{(l)}A^{(l)})$. Again, for this equation to work, the matrix $A^{(l)}$ needs to have 1s on its first row.

We will keep $A^{(l)}$ matrices in a vertical cell array of length L. Generate this cell array at the start of the function, in a similar manner as you generated the cell array for Θ matrices.

Implement the regularized cost function

The function “J” will take Θ matrices, an estimates matrix, Y matrix, and the regularization constant λ as input, and it will give us the cost of the current set of Θ matrices with respect to the training set.

Here, the estimates matrix is the activation values in the output layer. This function could take X instead of the estimates and execute the forward propagation to find the estimates, but this would be inefficient as we will already have the A matrices before calling this function. When we need to call this function, we will simply use `A{end}` to send the estimates matrix.

Remember the regularized cost function of a neural network:

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} -\log(h_\Theta(x^{(i)})_k)y_k^{(i)} - \log(1-h_\Theta(x^{(i)})_k)(1-y_k^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l+1}}(\Theta_{ji}^{(l)})^2$$

The first part is the unregularized cost and the second part is the sum of squared $\Theta^{(l)}_{ji}$ values scaled with λ/2m for regularization. Notice that regularization does not include $\Theta^{(l)}_{j0}$ values (i starts from 1).

Also notice that both the $x^{(i)}$ and $y^{(i)}$ are vectors (with different sizes) belonging to the i^{th} training example. $h_\Theta(x^{(i)})$ indicates the estimates, which is a vector of the same size with $y^{(i)}$. When we are dealing with vectors, using inner products does the inner summation for us. But when we have multiple training examples, and the estimates and Y are matrices with multiple columns, we cannot simply convert the inner product into a regular matrix multiplication. That’s because we only need to multiply the corresponding elements of log(estimates) and Y. Therefore, we need dot products, followed by the “sum” function applied twice.

Implement the back propagation

Implement the function “backPropagate” that will take Θ matrices, A matrices, Y matrix, and the regularization constant λ as input, and will return the cell array D which has the partial derivatives of the cost function w.r.t. each $\Theta^{(l)}_{ji}$. The number of matrices in D and their sizes should be identical to the matrices in Θ.

For a single training example, the backpropagation algorithm calculates the $\delta^{(l)}$ vectors, starting from the last layer, and finding values for the previous layers iteratively. We will have

$\delta^{(L)} = a^{(L)} - y^{(t)}$, where $y^{(t)}$ is the reformatted output vector of that specific training example. Then we can calculate each $\delta^{(l)}$ by $\delta^{(l)} = \Theta^{(l)T}\delta^{(l+1)} \cdot (a^{(l)} \cdot (1 - a^{(l)}))$. In this equation, $\Theta^{(l)}$ does not have the weights of the bias values, therefore you need to exclude its first column from the calculation.

Since we have multiple training examples, instead of $\delta^{(l)}$ vectors, we will calculate $\Delta^{(l)}$ matrices where $\delta^{(l)}$ of each training example will be on its columns. We will first find $\Delta^{(L)}$ by $\Delta^{(L)} = A^{(L)} - Y$. Then in a loop, we will find the previous layer matrices by $\Delta^{(l)} = \Theta^{(l)T}\Delta^{(l+1)} \cdot (A^{(l)} \cdot (1 - A^{(l)}))$. We will not find $\Delta^{(1)}$ as we don't need it. The note about the $\Theta^{(l)}$ in the previous paragraph also applies here.

After finding the Δ matrices, we will calculate the D matrices by $D^{(l)} := \Delta^{(l+1)}A^{(l)T}$. This is a precursor of the partial derivative that is not regularized and also not averaged. Also note that the A matrix in this equation needs to have the bias units.

We will finish this function by first adding $\lambda\Theta^{(l)}_{ji}$ to the corresponding $D^{(l)}_{ji}$ for regularization (only if $i \neq 0$), then dividing all elements by m (number of training examples) for averaging.

See how it works

If you have resolved all the problems indicated by the "test" script, you can continue executing the "run" script. The "run" script will load 5000 images of digits for training and display some of them. Then it will take a random 80% for training, call the given gradient descent function that will subsequently call your implementations to train a neural network, and calculate classification accuracy both on the training and the test sets. If your implementations are correct, both numbers should be over 0.9. Finally, it will display the misclassified digits in the test set.

In this example, each digit is 20 by 20, meaning we have 400 training features. We have 10 output classes, each class k corresponds to digit k, with the exception that the digit 0 is represented with class 10. For this example, the run script uses a single hidden layer of size 30. We chose this relatively small size for the hidden layer because it saves us some computation time and also it works fine for our problem. Try adding a second hidden layer by modifying the n vector (line 23 of run.m). You will witness that the accuracy on the training set will increase, but the accuracy on the test set will decrease, which indicates overfitting.

Submit your solutions

Collect every file that you created during this assignment (prepareY.m, initTheta.m, forwardPropagate.m, J.m, and backPropagate.m) in a zip file and submit the zip file to Gradescope. Do not zip their parent folder, but zip only the files. The name of the zip file does not matter.

Gradescope will give feedback on your assignment. If you don’t like your score, you can re-submit as long as the due date is in the future.