Group Homework 02

Neural Networks Learning

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition.

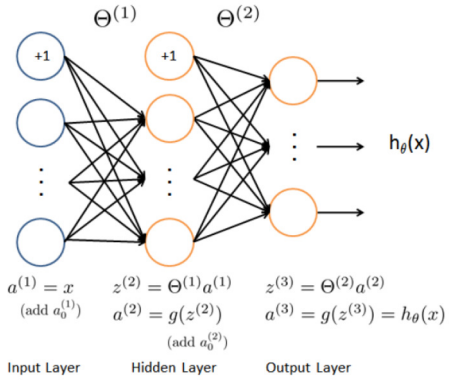
Part 1: Visualizing the data (0 points)

In the first part of **ghw2.m**, the code will load the data and display it on a 2-dimensional plot by calling the function **displayData**. This is the same dataset that you used in the previous exercise. There are 5000 training examples in **ghw2data.mat**, where each training example is a 20 pixel by 20 pixel grayscale image of the 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.

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

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, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

Part 2: Model representation (0 points)



Our neural network is shown in the above Figure. It has 3 layers: an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 x 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs +1). The training data will be loaded into the variables X and y by the **ghw2.m** script.

Group Homework 02

You have been provided with a set of network parameters ($\theta^{(1)}, \theta^{(2)}$) already trained by us. These are stored in **ghw2weights.mat** and will be loaded by **ghw2.m** 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).

Part 3: Feedforward and cost function (10 points)

Now you will implement the cost function and gradient for the neural network. First, complete the code in **nnCostFunction.m** to return the cost. Recall that the cost function for the neural network (without regularization) is

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

where $h_\theta\left(x^{(i)}\right)$ is computed as shown in the neural network Figure in Part 2 and K = 10 is the total number of possible labels. Note that $h_\theta\left(x^{(i)}\right)_k = a_k^{(3)}$ is the activation (output value) of the k-th output unit. Also, recall that whereas the original labels (in the variable y) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

$$y = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}, \quad \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}, \quad \ldots \quad \text{or} \quad \begin{bmatrix}0\\0\\0\\\vdots\\1\end{bmatrix}$$

For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0. You should implement the feedforward computation that computes $h_\theta\left(x^{(i)}\right)$ for every example i and sum the cost over all examples. Your code should also work for a dataset of any size, with any number of labels (you can assume that there are always at least $K \geq 3$ labels). Once you are done, **hw5.m** will call your **nnCostFunction** using the loaded set of parameters for **Theta1** and **Theta2**. You should see that the cost is about **0.287629**.

Part 4: Regularized cost function (10 points)

The cost function for neural networks with regularization is given by

Group Homework 02

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

You can assume that the neural network will only have 3 layers: an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\theta^{(1)}, \theta^{(2)}$ for clarity, do note that your code should in general work with $\theta^{(1)}, \theta^{(2)}$ of any size. Note that you should not be regularizing the terms that correspond to the bias. For the matrices **Theta1** and **Theta2**, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J using your existing **nnCostFunction.m** and then later add the cost for the regularization terms. Once you are done, **ghw2.m** will call your **nnCostFunction** using the loaded set of parameters for **Theta1** and **Theta2**, and $\lambda = 1$. You should see that the cost is about **0.383770**.

Part 5: Sigmoid gradient (5 points)

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.m** 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 using an advanced optimizer such as **fmincg**. You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

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

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

where

$$\text{sigmoid}(z) = g(z) = \frac{1}{1+e^{-z}}.$$

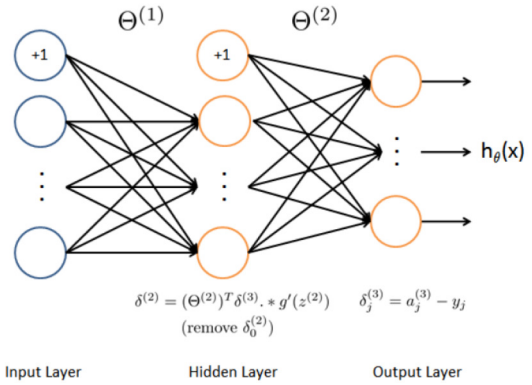
When you are done, try testing a few values by calling **sigmoidGradient(z)** at the Octave command line. For large values (both positive and negative) of z, the gradient should be close to 0. When z = 0, the gradient should be exactly **0.25**. Your code should also work with vectors and matrices. For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

Group Homework 02

Part 6: Random initialization (5 points)

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 $[-\varepsilon_{init}, \varepsilon_{init}]$. You should use $\varepsilon_{init} = 0.12$. 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 for $\theta^{(l)}$.

Part 7: Backpropagation (10 points)



Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example (x(t); y(t)), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_\theta(x)$. Then, for each node j in layer l, we would like to compute an "error term" $\delta_j^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (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).

In detail, here is the backpropagation algorithm (also depicted in Figure above), 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: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 $\left(x^{(t)}, y^{(t)}\right)$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

Group Homework 02

Step 1. Set the input layer's values ($a^{(1)}$) to the t-th training example $x^{(t)}$. Perform a feedforward pass, 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.

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

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

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 ($y_k = 0$).You may find logical arrays helpful for this task (explained in the previous programming exercise).

Step 3. For the hidden layer l = 2, set

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$

Step 4. Accumulate the gradient from this example using the following formula.

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

Note that you should skip or remove $\delta_0^{(2)}$ In Octave, removing $\delta_0^{(2)}$ corresponds to

```
delta_2 = delta_2(2:end)
```

Step 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 hw5.m will proceed to run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

Part 8: Gradient checking (0 points)

In your neural network, you are minimizing the cost function $J(\theta)$. To perform gradient checking on your parameters, you can imagine "unrolling" the parameters $\theta^{(1)}, \theta^{(2)}$ into a long vector. By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure. Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if $f_i(\theta)$ is outputting correct derivative values.

Group Homework 02

$$\text{Let} \quad \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \qquad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ε. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ε. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\varepsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

We have implemented the function to compute the numerical gradient for you in **computeNumericalGradient.m**. In the next step of **ghw2.m**, it will run the provided function **checkNNGradients.m** which will create a small neural network and dataset that will be used for checking your gradients. If your backpropagation implementation is correct, you should see a relative difference that is less than **1e-9**.

Part 9: Regularized Neural Networks (10 points)

After you have successfully implemented the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term after computing the gradients using backpropagation. Specifically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should 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 you should not be regularizing 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,

Group Homework 02

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(i)} & \Theta_{1,1}^{(l)} & \cdots \\ \Theta_{2,0}^{(i)} & \Theta_{2,1}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}$$

Somewhat confusingly, indexing in Octave starts from 1 (for both i and j), thus **Theta1(2, 1)** actually corresponds to $\theta_{20}^{(l)}$ (i.e., the entry in the second row, first column of the matrix shown above) Now modify your code that computes grad in **nnCostFunction** to account for regularization. After you are done, the **ghw2.m** script will proceed to run gradient checking on your implementation. If your code is correct, you should expect to see a relative difference that is less than **1e-9**.

Part 10: Learning parameters using fmincg (0 points)

After you have successfully implemented the neural network cost function and gradient computation, the next step of the **ghw2.m** script will use fmincg to learn a good set parameters. After the training completes, the **ghw2.m** script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about **95.3%** (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the neural network for more iterations. We encourage you to try training the neural network for more iterations (e.g., set **MaxIter** to 400) and also vary the regularization parameter λ. With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

Submission:

To submit, turn in the following files on Canvas:

- **nnCostFunction.m**
- **sigmoidGradient.m**
- **randInitializeWeights.m**