Programming Exercise 5: Neural Networks Learning

Files included in this exercise

File	Description
`data/ex3data1.mat`	Training set of hand-written digits
`data/ex3weights.mat`	Weights for the trained neural network
`utils.py`	Functions to display data and test gradient computation
[$\star$] `nn.py`	Functions to compute neural networks cost and gradient

[$\star$] indicates files you will need to complete

Problem statement and training set

In the previous exercise, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the provided weights. In this exercise, you will implement the backpropagation algorithm to learn the parameters for the neural network.

Our neural network, as shown in Figure 1.1, 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 20x20, this gives us 400 input layer units. By design we have decided to have 25 units in the hidden layer.

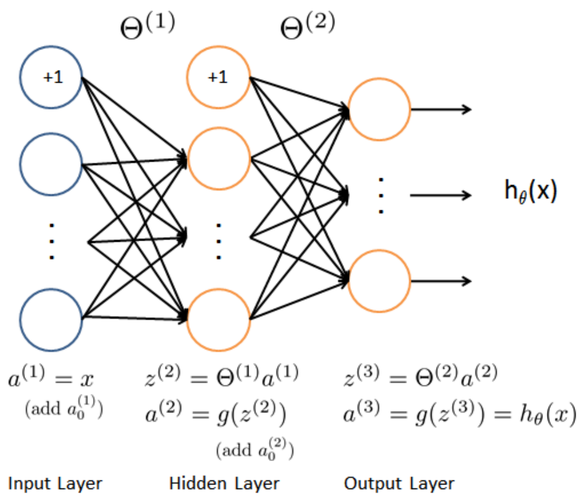

Figure 1.1: Neural network model

The output layer has 10 units corresponding to the 10 digit classes. Whereas the original labels (in the variable y) are $0, 1, \ldots, 9$, for the purpose of training a neural network, we need to encode the labels as vectors containing only values 0 or 1 ("one-hot" encoding), so that

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

For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ that you should use with the neural network will be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0.

Therefore, the first step after reading the data

```
data = loadmat('data/ex3data1.mat', squeeze_me=True)
y = data['y']
X = data['X']
```

is to build the one-hot encoding of the y readed from the file that will be used in the rest of the exercise.

Compute cost

The equation for the cost function for neural networks is:

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

where $h_\theta\left(x^{(i)}\right)$ is computed as shown in Figure 1.1, and $K = 10$ is the total number of possible labels. Note that $h_\theta(x^{(i)})_k$ is the activation (output value) of the k^{th} output unit.

You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$ already trained. 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). These parameters can be read by using the `scipy.io.loadmat` function, as shown below.

Complete the `nn.cost` function to implement the computation of the cost. You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ 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).

```python
def cost(theta1, theta2, X, y, lambda_):
    """
    Compute cost for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
```

```
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    """

    return J
```

You can check if your implementation is correct by computing the cost with the parameters provided in the file `data/ex3weights.mat`. The expected output of the cost function with those parameters is about 0.287629.

Next you have to extend the implementation of `nn.cost` to return the computation of regularized cost for neural networks, adding the regularization term to the previous function:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{j,i}^{(l)}\right)^2\right]$$

Although your code should work for any number of input units, hidden units and outputs units, assuming that the neural network will only have 3 layers, in this particular case the regularization term is:

$$\frac{\lambda}{2m}\left[\sum_{i=1}^{400}\sum_{j=1}^{25}\left(\Theta_{j,i}^{(1)}\right)^2 + \sum_{i=1}^{25}\sum_{j=1}^{10}\left(\Theta_{j,i}^{(2)}\right)^2\right]$$

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 ($i = 0$ in the equation above).

You can check if your implementation is correct by computing the cost with the parameters provided in the file `data/ex3weights.mat`. The expected output of the regularized cost function with those parameters and $\lambda = 1$ is about 0.383770.

Backpropagation

You will implement the backpropagation algorithm to compute the gradient for the neural network cost function. 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.

Complete the `nn.backprop` function to implement the computation of the gradient. Since the backpropagation requieres a first step of forward propagation, the function will also return the cost value with no additional computational cost:

```
def backprop(theta1, theta2, X, y, lambda_):
    """
    Compute cost and gradient for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    grad1 : array_like
        Gradient of the cost function with respect to weights
        for the first layer in the neural network, theta1.
        It has shape (2nd hidden layer size x input size + 1)

    grad2 : array_like
        Gradient of the cost function with respect to weights
        for the second layer in the neural network, theta2.
        It has shape (output layer size x 2nd hidden layer size + 1)

    """

    return J, grad1, grad2
```

You can check if your implementation is correct by calling `utils.checkNNGradients` with your backpropagation function as first parameter. `utils.checkNNGradients` creates a small neural network to check the backpropagation gradients by comparing the gradient returned by your backpropagation function with a numerical approximation that uses the cost also returned by your backpropagation function. These two gradient computations should result in very similar values.

Once the unregularized version is correct, extend `nn.backprop` to return regularized cost and gradient and check again that it is correct using `utils.checkNNGradients`.

Learning parameters

After you have successfully implemented the neural network cost function and gradient computation, the next step is to use gradient descent to learn a good set parameters:

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\qquad \Theta^{(1)} = \Theta^{(1)} - \alpha \frac{\partial}{\partial \Theta^{(1)}} J(\Theta) \\ &\qquad \Theta^{(2)} = \Theta^{(2)} - \alpha \frac{\partial}{\partial \Theta^{(2)}} J(\Theta) \\ &\} \end{aligned}$$

Before training you will need to randomly initilize the parameters. 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 $\epsilon_{init} = 0.12$. This range of values ensures that the parameters are kept small and makes the learning more efficient.

You can check your implementation by computing the training accuracy of your classifier, i.e, the percentage of training examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95% (this may vary by about 1% due to the random initialization) after running 1000 iterations of gradient descent with $\lambda = 1$ and $\alpha = 1$.

Learning parameters using `scipy.optimize.minimize`

Use now the `scipy.optimize.minimize` to learn the parameters for the neural network. `scipy.optimize.minimize` minimize a scalar function of one or more variables. In this case we will use it to find the parameters that minimize the cost of the neural network. Using the 'TNC' method, $\lambda = 1$ and just 100 iterations you should get again a training accuracy of about 95%.

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to "overfit" a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the `maxiter` parameter to a higher number of iterations to see this for yourself.