Programming Exercise 4: Multi-class Classification and Neural Networks

Files included in this exercise

File	Description
`data/ex3data1.mat`	Training set of hand-written digits
`data/ex3weights.mat`	Initial weights for the neural network exercise
`utils.py`	Functions to display data
[⋆] `multi_class.py`	Functions to compute multi-class classification

[⋆] indicates files you will need to complete

Part A: Multi-class Classification

Problem Statement

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits[1]. The `.mat` format means that the data has been saved in a binary 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 `scipy.io.loadmat` function, as shown below.

```
    data = loadmat('data/ex3data1.mat', squeeze_me=True)

```

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

```
    X = data['X']
    y = data['y']
```

There are 5000 training examples in `ex3data1.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. The second part of the training set is a 5000-dimensional vector y that contains labels for the training set labeled as “0” to “9” in their natural order.

Training data is shown in Figure 1.1 which has been generated with the help of function `utils.plot_data` applied on a random selection of 100 X rows:

```
    rand_indices = np.random.choice(X.shape[0], 100, replace=False)
    utils.displayData(X[rand_indices, :])
```

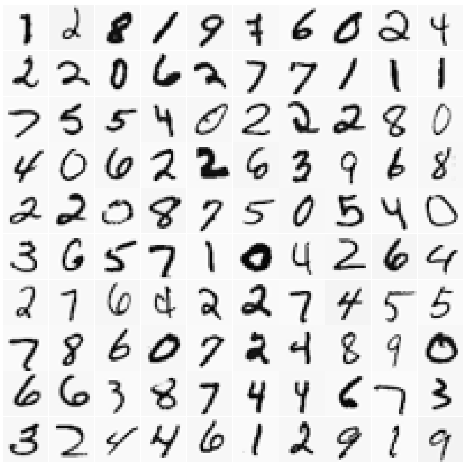

Figure 1.1: Dataset sample

Vectorizing Logistic Regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized.

Implement a vectorized version of logistic regression that does not employ any for loops. Use your code in the previous exercise as a starting point to implement vectorized versions of `logistic_reg.compute_cost_reg` and `logistic_reg.compute_gradient_reg` and check if your implementation is correct by running and passing those same tests.

One-vs-all Classification

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset. In the handwritten digits dataset, $K = 10$, but your code should work for any value of K.

Complete the code for the function `multi_class.oneVsAll`, to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\theta \in \mathbb{R}^{K\times(N+1)}$, where each row of θ corresponds to the learned logistic regression parameters for one class ($\theta_0 = b, \theta_1 = w_1, \ldots, \theta_n = w_n$). You can do this with a "for"-loop, training each classifier independently. Hint: You can use $y == c$ to obtain a vector of 1's and 0's that has 1's in the posititions where y is equals to c and 0's in the rest.

```python
def oneVsAll(X, y, n_labels, lambda_):
    """
     Trains n_labels logistic regression classifiers and returns
     each of these classifiers in a matrix all_theta, where the i-th
     row of all_theta corresponds to the classifier for label i.

     Parameters
     ----------
     X : array_like
         The input dataset of shape (m x n). m is the number of
         data points, and n is the number of features.

     y : array_like
         The data labels. A vector of shape (m, ).

     n_labels : int
         Number of possible labels.

     lambda_ : float
         The logistic regularization parameter.

     Returns
     -------
     all_theta : array_like
         The trained parameters for logistic regression for each class.
         This is a matrix of shape (K x n+1) where K is number of classes
         (ie. `n_labels`) and n is number of features without the bias.
     """

    return all_theta
```

One-vs-all Prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the "probability" that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label $(0, 1, \ldots, K-1)$ as the prediction for the input example. Hint: this code can be vectorized using the numpy `argmax` function.

```
def predictOneVsAll(all_theta, X):
    """
    Return a vector of predictions for each example in the matrix X.
    Note that X contains the examples in rows. all_theta is a matrix where
    the i-th row is a trained logistic regression theta vector for the
    i-th class. You should set p to a vector of values from 0..K-1
    (e.g., p = [0, 2, 0, 1] predicts classes 0, 2, 0, 1 for 4 examples) .

    Parameters
    ----------
    all_theta : array_like
        The trained parameters for logistic regression for each class.
        This is a matrix of shape (K x n+1) where K is number of classes
        and n is number of features without the bias.

    X : array_like
        Data points to predict their labels. This is a matrix of shape
        (m x n) where m is number of data points to predict, and n is number
        of features without the bias term. Note we add the bias term for X in
        this function.

    Returns
    -------
    p : array_like
        The predictions for each data point in X.
        This is a vector of shape (m, ).
    """

    return p
```

Once you are done, call your `predictOneVsAll` function using the learned value of θ returned by `oneVsAll`. You should see that the training set accuracy is about 95% (i.e., it classifies 95% of the examples in the training set correctly).

Part B: Neural Networks

Model representacion

In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. You will be using parameters from a neural network that have already been trained. Your goal is to implement the feedforward propagation algorithm to use those weights for prediction.

Our neural network is shown in Figure 1.2. 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 20x20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1).

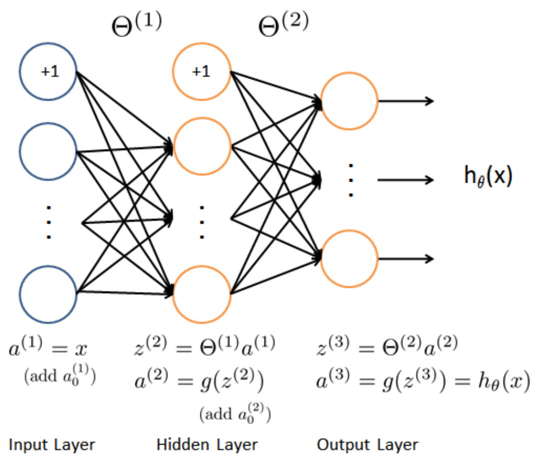

Figure 1.2: Neural network model

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.

```
    weights = loadmat('data/ex3weights.mat')
    theta1, theta2 = weights['Theta1'], weights['Theta2']
```

Feedforward Propagation and Prediction

Complete the code in the function `multi_class.predict` to return the neural network's prediction. You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and returns the associated predictions. Similar to the one-vs-all classification strategy, the prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$.

Hint: It will help to obtain a vectorized solution to the problem if a column of 1's is added to the matrix X before computing the feedforward propagation:

```
    m = X.shape[0]
    X1s = np.hstack([np.ones((m, 1)), X])
```

```
def predict(theta1, theta2, X):
    """
    Predict the label of an input given a trained 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)

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

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

    Return
    ------
    p : array_like
        Predictions vector containing the predicted label for each example.
        It has a length equal to the number of examples.
    """

    return p
```

Applied on the provided parameters, $Theta1$ and $Theta2$, and the training examples in X you should see that the accuracy is about 97.5%.