Lab#6: Neural Networks

Prepared by: Teeradaj Racharak (r.teeradaj@gmail.com)

Feedforward propagation for Prediction[1]

In this exercise, you will implement a neural network to recognize handwritten digits using the training examples from Scenario#2 of Lab#3 *i.e.* 5,000 images of handwritten digits. Figure 1 shows examples from the dataset, in which each example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating number indicating the grayscale intensity at that location.

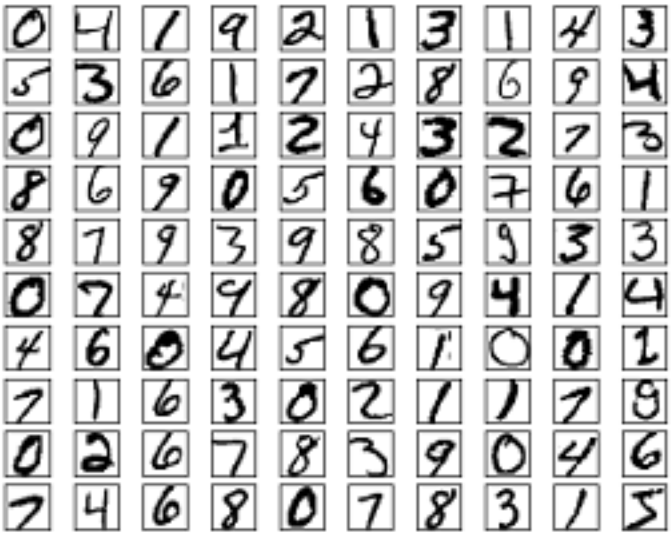

Figure 1 Examples from the dataset

The 20×20 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 5000×400 matrix X where every row is a training example for a handwritten digit image as follows:

$$X = \begin{bmatrix} --(x^{(1)})^T-- \\ --(x^{(2)})^T-- \\ \vdots \\ --(x^{(m)})^T-- \end{bmatrix}_{500\times(400+1)}$$

[1] This exercise is taken from and revised from Andrew Ng's machine learning assignments (Coursera).

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. Since this example (as in Scenario#2 of Lab#3) stores the images in .mat file, then we make them compatible with MATLAB indexing *i.e.* a '0' digit is labeled as '10' whereas the digits '1' to '9' are labeled as '1' to '9', respectively. The following illustrate 5000×1 vector y:

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}_{10\times 1}$$

Problem 1.1 [Python from scratch]. Let's practice using a set of provided weights as the parameters $\Theta^{(1)}, \Theta^{(2)}$ to implement a neural network classifier. Since you are provided with the parameters, your goals in this problem are only:

- implement the feedforward propagation algorithm, and
- make predictions *i.e.* recognizing handwritten digits.

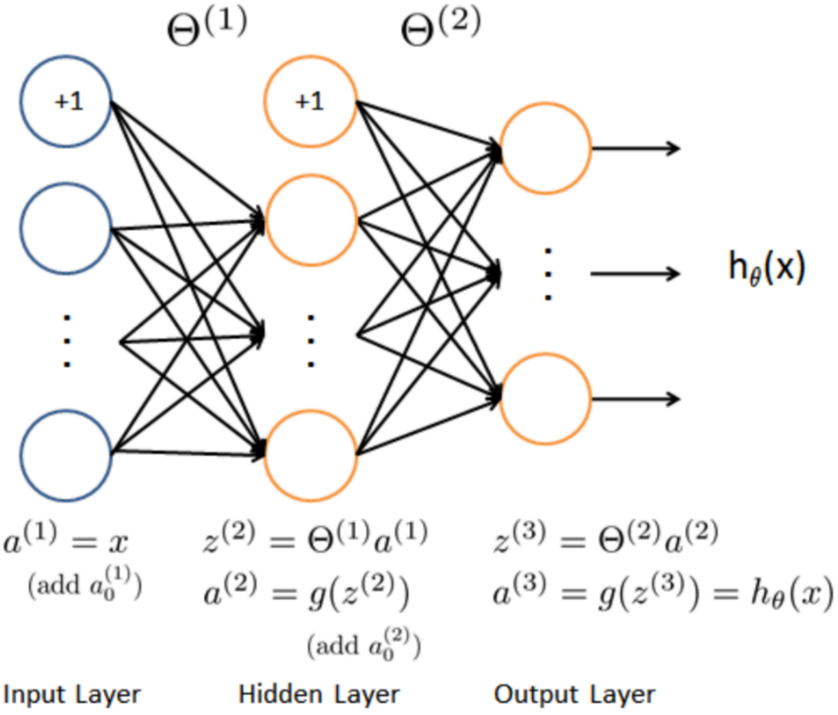

Figure 2 Neural network model

In order to make prediction using neural networks, we model an architecture by consisting of 3 layers as follows (*cf.* Figure 2) *viz.* an input layer, a hidden layer, and an output layer. As before, the training examples will be loaded into (X, y).

Recall that our input units are pixel values of digit images. Since the images are of size 20×20, this gives us 400 input units (not counting the extra bias unit which always outputs +1 yet). These units will forward propagate through the network with a set of parameters $\Theta^{(1)}_{25\times401}$ and $\Theta^{(2)}_{10\times26}$. These parameters are stored in ex4weights.mat and you are asked to write a Python code for loading those parameters into variables Theta1 and Theta2. These parameters also implies that:

- the hidden layer contains 25 units, and
- the output layer contains 10 units.

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 method, the prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$. Once you are done, you should see that the accuracy is about 97.5%.

Backpropagation for Parameters Learning

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

Recall that the original labels are ‘1’, ‘2’, …, ’10’. To learn the parameters of a neural network, we need to recode each label in y as a one-hot vector containing only values 0 or 1 as follows:

$$1 \Longrightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{10\times1}, 2 \Longrightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{10\times1}, \ldots, \quad \text{and } 10 \Longrightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}_{10\times1}$$

Problem 2.1 [Python from scratch]. Now, you are asked to implement the cost function for the network. Let’s try to complete the function costfunction to compute the cost function without regularization. Recall that the cost function is defined as:

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

where $h_\theta(x^{(i)})$ is computed as shown in Figure 2 and $K = 10$ *i.e.* the total number of all labels. Noted that $h_\theta(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k-th output unit. Your computation should compute $h_\theta(x^{(i)})$ for every example i and sum the cost over all examples. After you have completed the implementation, you should see that the cost is around 0.287629.

Problem 2.2 [Python from scratch]. The cost function for neural network with regularization is defined as:

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

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

Noted that you should NOT regularize the terms that correspond to biases. For the matrices `Theta1` and `Theta2`, this corresponds to the first column of each matrix. You may reuse the code from Problem 2.2 by writing the code to compute the regularization term separately and adding up with the unregularized cost function. Once you are done, setting $\lambda = 1$, you should see that the cost is around 0.383770.

Problem 2.3 [Python from scratch]. Before implementing the backpropagation algorithm to compute the gradient for the neural network cost function, we will implement the sigmoid gradient function in this problem. The gradient for the sigmoid function can be computed as:

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

where $\text{sigmoid}(x) = g(z) = \dfrac{1}{1+e^{-z}}$

When you are done, try testing with a few values on the implemented function. 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.

Problem 2.4 [Python from scratch]. Now, we are ready to implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows.

Given a training example $(x^{(i)}, y^{(i)})$, we will first run a 'feedforward' to compute all 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 hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

Here is the backpropagation algorithm in detail (*cf.* Figure 4). You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop to enumerate a training example at a time and place steps 1-4 inside that loop. That is, at iteration t^{th}, we perform the calculation on the t^{th} training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

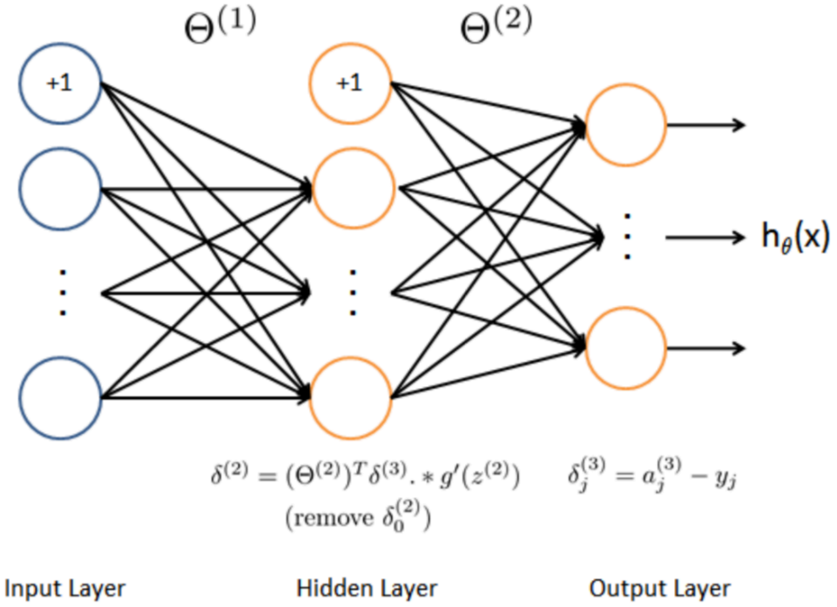
$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)}.*g'(z^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$
Input Layer
Hidden Layer
Output Layer

Figure 4 Backpropagation updates

1. Set the input units *i.e.* $a^{(1)}$ to the t^{th} training example $x^{(t)}$. Perform a feedforward pass (*cf.* Figure 2) to compute the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$ for layers 2 and 3. Noted 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.

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 $(y_k = 0)$.

3. For the hidden layer $l = 2$, set $\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)}.*g'(z^{(2)})$.

4. Accumulate the gradient using the formula: $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$. Noted that you ignore $\delta_0^{(2)}$ at this step.

5. Obtain the unregularized gradient for the neural network cost function by dividing the accumulated gradients by $1/m$:

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

Problem 2.5 [Python from scratch]. 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. That is, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization as follows:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \qquad \text{for}\, j = 0$$

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

Noted that you should NOT regularize 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,

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

Now, it’s time to modify your code to compute the gradients in the function `costfunction` to account for the regularization.

Gradient Descent for Optimal Parameters

In the exercise, we will put everything together for learning a good set of parameters. You may use an off-the-shelf optimizer (such as the one from SciPy that we used last time) for this purpose. However, this is the first time we practice on neural networks. So, let’s try to continuously make it from scratch !

Problem 3.1 [Python from scratch]. Write a batch gradient descent to train the neural network for 1,000 iterations with $\alpha = 0.8$. If your implementation is correct, the training accuracy should be about 95.3% (this may vary by about 1% due to the random initialization and a permutation). We also encourage to try training the neural network for more iterations and different values of parameter λ.

Problem 3.2 [Python from scratch]. scikit-learn has provided methods to load and fetch popular datasets. This helps us to study and conduct experiments on machine learning techniques (see **https://scikit-learn.org/stable/datasets/index.html#datasets**). In this exercise, we will try to load the breast cancer dataset using the methods provided in **https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html**. This is a binary classification dataset and your tasks are to:

- construct a 4-layer neural network with sigmoid activation functions, in which the first hidden layer has 6 units and the second hidden layer has 5 units;
- initialize the value of parameters to close to 0 *i.e.* $\Theta_{ij} \sim \mathcal{N}(0,0.1)$
- write a stochastic gradient descent to train for 1,000 iterations with $\alpha = 0.01$

If your implementation is correct, you should see that the training cost is 0.400422. This value may vary depending on the random initialization.

Verifying Your Gradient Computation

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 θ. By doing so, we can think of the cost function as $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that computes $\frac{\partial}{\partial \theta_i} J(\theta)$. You’d like to check if function f_i is outputting the correct derivative values. To do this, we let:

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

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:

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

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

Problem 4.1 [Python from scratch]. In this exercise, you are asked to extend Problem 3.2 to encapsulate the above numerical gradient method, which can be triggered to execute via a flag (*i.e.* if-condition). If your implementation is correct you should see that the difference between the actual gradients and the theoretical gradients are relatively small.

Tip 1: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units because the network will have a relatively small number of parameters. Since each dimension of θ requires two evaluations of the cost function, this kind of checking can be expensive. After you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

Tip 2: Gradient checking works for any function that you are computing the cost and the gradient. That is, you can use this technique to check if your gradient implementations for other exercises are correctly implemented, too (such as logistic regression's cost function).

Implementation with TensorFlow

It's time to practice an implementation of neural networks using tf.keras, which is high-level API to build and train models in TensorFlow.

In this part of the exercise, you will practice another optimizer provided by TensorFlow called 'AdamOptimizer'.

What is the Adam optimization algorithm? Adam was first introduced by Diederik Kingma (from OpenAI) and Jimmy Ba (from the University of Toronto) in a 2015 ICLR paper titled 'Adam: A Method for Stochastic Optimization'. As stated in the paper, Adam is not an acronym and is not written as 'ADAM'; but is rather derived from 'adaptive moment estimation'.

How does Adam work? Adam is different to classical stochastic gradient descent as follows. Stochastic gradient descent maintains a single learning rate (*i.e.* α) for all weight updates and the learning rate does not change during training.

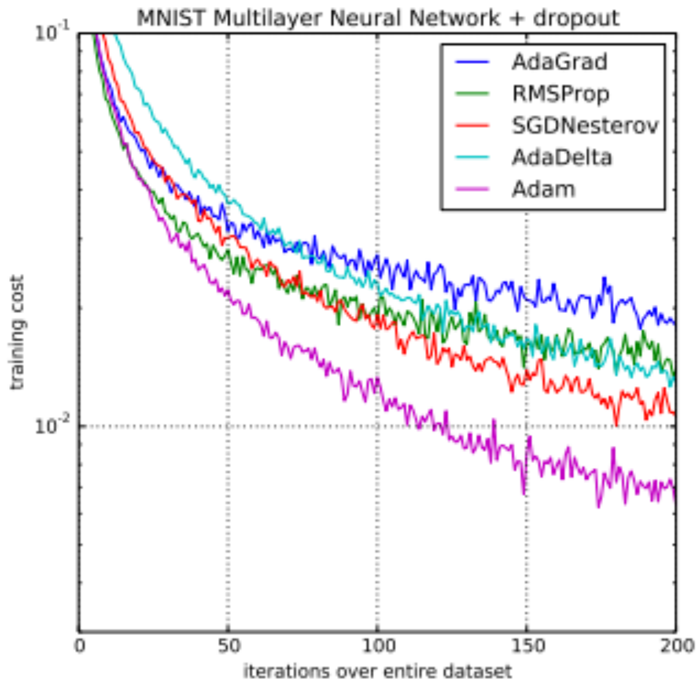

Figure 5 Comparison of Adam to other optimization algorithms[2]

However, Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Adam is a popular algorithm in the field of deep learning because it achieves good results. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods (*e.g.* AdaGrad, RMSProp). In the original paper, Adam was shown to solve practical deep learning problems (*cf.* Figure 5).

Problem 5.1 [Python with TensorFlow]. Now, you will use the Fashion MNIST dataset which contains 70,000 grayscale images in 10 categories. The images are 28x28 NumPy arrays, with pixel values ranging between 0 and 255. Note that Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset — often used as the 'Hello, World' of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, *etc.*) in an identical format to the clothing dataset we use here.

[2] Taken from Adam: A Method for Stochastic Optimization, 2015