Problem Set 4: Backpropagation

CMSC 422, Fall 2019

Assigned 9/25, Due 10/9 at 2pm

Submission Instructions
You will submit a zip file named YourID_project1.zip containing a directory named YourID. So if your login ID is steveholt, you will submit steveholt_project1.zip containing the folder steveholt. Inside the directory should be your version of the file ps4.py, along with a pdf named YourID_project1.pdf with the necessary answers to written questions. Failure to follow the format guidelines will result in points being taken off.
Grading:
We will test your code on withheld data to make sure they work. Do not modify the provided names and interfaces, or the autograder might fail.

In this problem set you will implement backpropagation for a set of different neural network architectures. Your implementations should be written in the file `ps4.py`. There is some code provided for you, and you will write your implementations in the places marked with `#TODO: Your Code Here`.

1. We'll begin with the simplest possible network. It has a single input feature that we call x. This is the activation of the single node of the input layer. This is connected to a single output node, which has a weight, w. We also have a bias term, so the activation of the output unit is $a = wx + b$. This network will be used to solve a linear regression problem. So, if we are given an input pair of (x, y), we want to minimize a loss of $L(x, y) = (a - y)^2 = (wx + b - y)^2$. To do this, you will need to randomly initialize the weight and bias and then perform gradient descent.

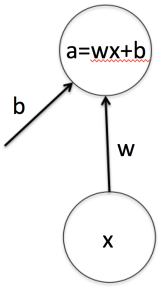

Figure 1: Network for Problem 1

The gradient of the loss is computed using a training set containing pairs, $(x_1, y_1), (x_2, y_2), ...(x_n, y_n)$. We have:

$$\nabla L = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\partial L}{\partial w}(x_i, y_i), \frac{\partial L}{\partial b}(x_i, y_i) \right)$$

If we denote $\theta = (w, b)$ as a vector containing all the parameters of the network, we perform gradient descent with the update:

$$\theta^k = \theta^{k-1} - \eta \nabla L$$

Here η is the learning rate, and θ^k denotes a vector of (w, b) after the k'th iteration of gradient descent.

We provide you with a routine to generate training data. This has the form:

```
simplest_training_data(n)
```

This just generates n random training points on the line $y = 3x + 2$, with a little Gaussian noise added to the points.

You need to write a routine with the form:

```
simplest_training(n, k, eta)
```

Here n indicates the number of points in the training set (you can call simplest_training_data to get the training data), k indicates the total number of iterations that you will use in training, and η is the learning rate. To initialize the weights in your network, we suggest that you initialize w with a Gaussian random variable with mean 0 and variance of 1, and that you initialize $b = 0$.

You also need to write a routine of the form:

```
simplest_testing(theta, x)
```

This routine applies the network, using the parameters in theta, to the input values in the vector x, and returns a vector of results in y.

After training, the network should learn w and b values that are similar to those used to train the network. So you can test your network by looking at the learned w and b values. Or you can use the testing algorithm to see if the network computes appropriate y values. In testing, you may find that if you use too big a value for η the network will not converge to anything meaningful. If you use a value of k that is too small, it won't have time to converge to a good solution.

We run our algorithm with $n = 30, k = 10,000, \eta = .02$. When we test using x values of 0, 1, ..., 9 we get the result:

```
    1.9504    4.9666    7.9828   10.9990   14.0152   17.0314   20.0476
23.0638   26.0800   29.0962
```

These points fit the line $y = 3x + 2$.

Test your algorithm using this or any other test you choose. Hand in the code you have written and a written description of your test and the results.

2. You will now create a network that is a little more complicated. It still contains just an input and an output layer, with no hidden layers. But it now has a nonlinearity along with a cross-entropy loss, so that we can use it for classification.

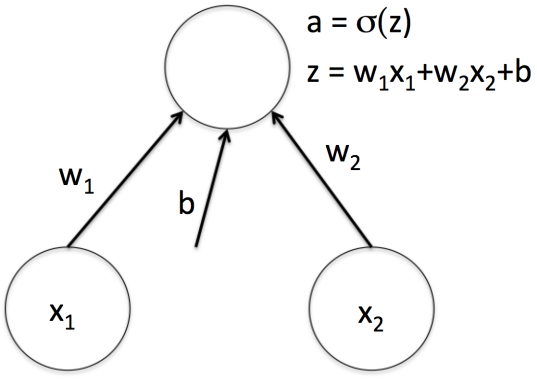

Figure 2: Network for Problem 2

The network has two inputs, x_1 and x_2. These are connected with two weights to a single output unit. If we let $z = w_1x_1 + w_2x_2 + b$, the output unit will have an activation of $a = \sigma(z)$, where $\sigma(z)$ represents the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

We can interpret the output as giving the probability that the input belongs to class 1. If the probability is low, then the input probably belongs to class 0. Hint: the derivative of the sigmoid is given by:

$$\frac{d\sigma}{dz} = \sigma(x)(1 - \sigma(x))$$

In training the network, you will use the cross-entropy loss. In this case, the cross entropy loss will be:

$$-(y \log a + (1 - y) \log (1 - a)).$$

If $y = 1$, this is just the negative log of the probability that the network predicts for the probability that the input belongs to class 1. If $y = 0$, it is the negative log of the probability that the input belongs to class 0.

We provide you with a routine to generate training data. This has the form:

```
single_layer_training_data(trainset)
```

which returns X and y. This provides two different training sets. When the input, trainset, is 1, the function produces a simple, linearly separable training set. Half the points are near $(0, 0)$ and half are near $(10, 10)$. X is a matrix in which each row contains one of these points, so it is $n \times 2$, where n is the number of points. y is a vector of class labels, which have the value 1 for the points near $(0, 0)$ and 0 for the points near $(10, 10)$.

When trainset is 2, we generate a different training set that is not linearly separable, but that corresponds to the Xor problem. Points from class 1 are either near $(0, 0)$ or $(10, 10)$, while points in class 0 are near either $(10, 0)$ or $(0, 10)$.

You will need to implement two routines. The first is:

```
single_layer_training(k, eta, trainset)
```

As before, k will indicate the number of iterations of gradient descent and eta gives the learning rate. trainset indicates which training set to use, 1 or 2. You will train the network using the same gradient descent approach as in the previous problem. As before, we suggest that you initialize weights using random values chosen from a Gaussian distribution with zero mean, and that you initialize bias at 0.

You will also implement a test routine:

```
single_layer_testing(theta, X)
```

This takes in the network parameters and a matrix, X, of the form returned by single_layer_training_data. It returns a vector of the output values the network computes.

Hand in the code for these two routines, and a description of tests of your code using the two trainset values. You will need to figure out how many iterations of gradient descent to perform and the appropriate learning rate to get good results. Do not use the data you used to train the network, call single_layer_training_data again to get fresh data for testing. When trainset = 1, your network should assign a high probability of belonging to class 1 for points near $(0, 0)$, and a low probability for points near $(10, 10)$. When trainset = 2, the data is not linearly separable, so you may find that your network has problems being able to separate. Describe what happens in both cases.

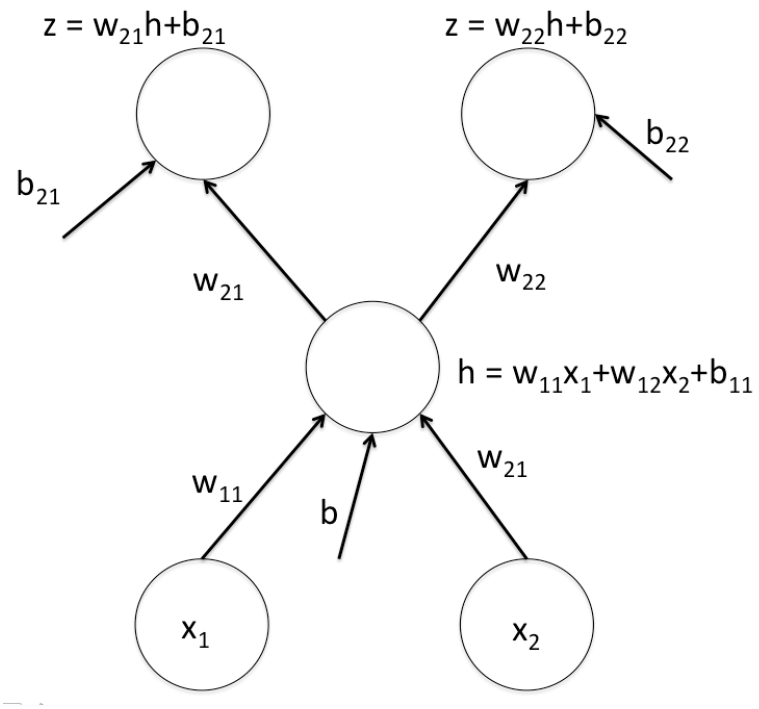

Figure 3: Network for Problem 3

3. Now you will implement a multi-layer network that has a hidden layer. To start with a relatively simple case, we will do this without any non-linearities. The network has two input units, x_1 and x_2. These are connected to a single hidden unit. We'll call the activation of this hidden unit h, so $h = w_{11}x_1 + w_{12}x_2 + b_{11}$. This hidden unit is connected to two output units. We'll call their activation z_1 and z_2, so we have:

$$z_1 = w_{21}h + b_{21} \qquad z_2 = w_{22}h + b_{22}$$

To train this network, we use a loss function that says that we want the output to be close to the input. So the loss function is:

$$L(x_1, x_2) = (z_1 - x_1)^2 + (z_2 - x_2)^2$$

That is, the input is also acting as the label. This kind of network is called an *auto-encoder*. You may be wondering what the point of this is. Because the hidden layer is smaller than the input and output layers, the network is forced to learn low-dimensional representation of the data. In this case, the network learns to map the input points onto a line in the hidden layer, and then compute the 2D coordinates of the points on this line for the output layer. This process is called Principal Component Analysis (PCA), and we will learn more about it later in the semester.

We will provide a routine to generate training data:

```
pca_training_data(n, sigma)
```

The input parameter n indicates the number of points in the training set. As in the last problem, X contains a $n \times 2$ matrix in which each row contains the coordinates of a 2D point. These points are generated to lie along the line $y = x + 1$. Then Gaussian noise is added to the points, with zero mean and a standard deviation of sigma.

Once again, you will implement training and testing routines.

```
pca_training(k, eta, n, sigma)
```

The input k gives the number of iterations of gradient descent to use, while *eta* gives the learning rate. The input value n indicates the number of points in the training set, while *sigma* indicates the amount of noise added to these points. Use these as parameters to pca_training_data. The routine returns theta, a representation of all the weights and biases in the network.

Also implement a test routine:

```
pca_test(theta, X)
```

X will contain test data in the form returned by pca_training_data. Z provides the results the network produces given this input; Z has the same format as X.

To test this, try training the network with $n = 10$ and $sigma = .1$. Then test, using the input:

```
pca_test(theta, [[1,2], [4,5], [10, 3]])
```

When I run my code with this test I get:

```
[[0.9418    2.0653]
[3.9543    5.0511]
[6.1780    7.2551]]
```

as the result. It may take a little work to find good values for k and η. Turn in your code, along with a description of your experimental results. Can you explain why the network produces the point $(6.1780, 7.2551)$ with an input of $(10, 3)$?

Do another test with $sigma = 0$ instead of $sigma = .1$. Run your network with the same test data. How have the results changed? Can you explain this change?

4. **Challenge Problem (optional, for extra credit):**
Ok, now you are ready to create a complete, fully connected neural net with a hidden layer and non-linearities. You will use this network to solve the Xor problem, using the same training data as in Problem 2. Your network architecture should have the following components:

   - Two input units, with activations x_1 and x_2. These are just the coordinates of 2D points.
   - A variable number of hidden units, H. Write your code so that you can select the number of hidden units as a hyperparameter. Let's call the activation of the i'th hidden unit, a_i^1. Let's call the weights of these units w_{ij}^1. This is the weight from input unit j to hidden unit i.
   - Use a RELU non-linearity for the hidden units. So to determine the activation of a hidden unit we have: $z_i^1 = w_{i1}^1 x_1 + w_{i2}^1 x_2 + b_i^1$, and $a_i^1 = max(0, z_i^1)$.
   - There is then a single output unit. Call its activation a^2. We compute this as: $z^2 = \left(\sum_{i=1}^{H} w_{1i}^2 a_i^1\right) + b^2$, and $a^2 = \sigma(z^2)$, where σ is the sigmoid nonlinearity. This last part is just the same as in Problem 2. And, like Problem 2, you can train your network using the cross-entropy loss.

   Implement test and training functions with the templates:

   ```
   nn_training(k, eta, trainset, H)
   ```

   ```
   nn_testing(theta, X)
   ```

   The parameters to the training routine are similar to those in Problem 2, with H indicating the number of hidden units. The testing routine has the same form as in Problem 2.

   Run experiments to demonstrate that your network can solve the Xor problem. How do you find the results vary as you vary the number of hidden units? Show the results of your experiments and turn in the code.