

## Project 2: Linear and Non-linear Networks

### Introduction

In this project, we will explore linear and non-linear networks on the same data sets we used in project 1.

**Grading:** see note from P1.

You will need to hand in: (a) your analysis in a file called `writeup.pdf`, (b) your code for `GD.m`, `linearModel.m` and `TwoLayer.m`, (c) a file called `results.mat` that will be created for you.

I have provided quite a bit of skeleton code. More importantly, there is a `run.m` script. This is the central part of the project. Once you're confident that your learning code is doing (more or less) the right thing, you'll run `run.m`.

After running the tests, it will use your code to make predictions on some held-out test data. You don't have access to the true labels for the test data, so you'll just have to cross your fingers (you really need to *trust* the learning algorithm). The predictions of your model on the test data are stored in the `results.mat` file, which we will compare to the truth.

**Important note:** Every time you run `run.m` it will first load in `results.mat` so that it can resume where it left off (this way you don't have to keep rerunning the same tests over and over). However, this also means that if you need to "back up" and run some old experiments, you will need to delete the `results.mat` file so that it can't load it.

**Early warning:** getting through each of the full experiment cycles takes about 16 minutes with my implementation (most of the time with the machine thinking). Yours will likely take longer, especially if you are new to Matlab. (And my same code running in Octave takes over an hour!) So, start early and don't put off the "analysis" section until the end because your write-up will depend on the output that the `run` produces.

Remember, handins are at: <http://www.cs.utah.edu/~hal/handin.pl?course=cs5350>.

Finally, one of the datasets we're using has 10 classes (labeled zero through nine), the other has two (labeled 0 and 1)... your implementations should be robust to this.

### Task 1: Gradient Descent

(10%)

Your first task is to implement gradient descent in the file `GD.m`. This takes as input a current weight vector `w0`, some data `X` and `Y`, a learning rate `eta` a number of iterations (passes over the training set) to run `numIter` and a boolean flag for if it should do `stochastic` optimization or not.

It produces two outputs: the final weight vector and a trace of all weight vectors encountered during training.

The most important thing is that it is passed a function `grad` which takes parameters `(X, Y, w)` and produces the gradient of the function we're trying to optimize on that data. That way, this function is generic: we can plug in any gradient we want and it should (try to) find a minimum.

A simple way to test this function is to use it to try to find the minimum of some simple function, like  $f(w) = 3w^2 - 2w + 1$ . We can do this by passing it a single data point (which it will ignore) and then a gradient function that computes the gradient. Here, the gradient is just  $6w - 2$ . With a learning rate of 0.1 and 10 iterations, we can do this as:

```
>> [w,wtrace] = GD( @(X,Y,w) 6*w-2 , 0.1, 10, 0, [1], [1], 0)
w =
    0.3333
```

```
wtrace =
    0.2000
    0.2800
    0.3120
    0.3248
    0.3299
    0.3320
    0.3328
    0.3331
    0.3332
    0.3333
```

We can verify by hand that this is correct. Moreover, we can plot the results.

```
>> x = [-1:0.1:3];
>> wtrace = cell2mat(wtrace);
>> plot(x, 3*x.^2 - 2*x + 1, 'b-', wtrace, 3*wtrace.^2 - 2*wtrace + 1, 'ro')
```

Here, you should get an image that shows the points encountered in the trace. They're all close to the optimum. If we initialize  $w$  with 6 instead of 0, it takes a while longer to converge:

```
>> [w,wtrace] = GD( @(X,Y,w) 6*w-2 , 0.1, 10, 6, [1], [1], 0)
w =
    0.3339
```

```
wtrace =
    2.6000
    1.2400
    0.6960
    0.4784
    0.3914
    0.3565
    0.3426
    0.3370
    0.3348
    0.3339
```

```
>> plot(x, 3*x.^2 - 2*x + 1, 'b-', wtrace, 3*wtrace.^2 - 2*wtrace + 1, 'ro')
```

If you feel like playing around with other functions, you can do so to make sure your implementation works.

For instance, let's take a non-convex function:  $f(w) = w^4 - 6w^2 + 2w$ , with derivative  $4w^3 - 12w + 2$ . First, plot this function to see what it looks like:

```
>> x = [-3:0.1:3];
>> plot(x, x.^4 - 6*x.^2 + 2*x, 'b-');
```

Now, we can try optimizing starting at 0:

```
>> [w,wtrace] = GD( @(X,Y,w) 4*w.^3 - 12*w + 2 , 0.1, 10, 0, [1], [1], 0)
w =
```

```

-2.0075

wtrace =
-0.2000
-0.6368
-1.4977
-2.1512
-0.9508
-1.9479
-1.5290
-2.1340
-1.0076
-2.0075

>> wtrace = cell2mat(wtrace);
>> plot(x, x.^4 - 6*x.^2 + 2*x, 'b-', wtrace, wtrace.^4 - 6*wtrace.^2 + 2*wtrace, 'ro-');

```

Here we can see that (a) it found the “wrong” minimum and (b) that the learning rate was too large.

Play around with the learning rate to see if you can find a value that works well. Alternatively, maybe try with decreasing learning rates. Try to get it to find the global minimum.

**Please include in your writeup** a bit about what you did to make this work.

## Task 2: Linear Model

(35%)

Your first task is to implement a linear classifier that will work with an  $l_2$  regularizer and a selection of loss functions (perceptron, hinge, squared, logistic and exponential). There is shell code for this in `linearModel.m`. This makes use of your gradient descent implementation in `GD.m`.

**Note:** If you’re having a huge difficulty getting your `GD.m` to work, let us know by email and we can give you ours. Of course, in this case, you’re forfeiting the 10% points from Task 1.

As usual, you’ll have to implement both a `train` function and a `predict` function. The `predict` should be a trivial one-liner. The `train` function is also trivial: it’s provided to you. Essentially, it just shells out to the gradient descent implementation. *Your real job* is to implement the gradient computations in `computeGradient()`.

My suggested implementation (which is primarily for ease of reasoning, not for efficiency) is sketched in the skeleton code. If you want to do stuff at a matrix level rather than a vector level to show off your matlab chops, more power to you. Essentially, we’re computing the *gradient* of a function that looks like  $\lambda \|\mathbf{w}\| + \sum_n \ell(y_n, \mathbf{w}^\top \mathbf{x}_n)$ , where  $\lambda$  is the regularization constant. Unless you’re trying for the bonus (see the end of this document), the norm is  $\|\mathbf{w}\| = \sum_d w_d^2$  the  $l_2$  norm. The regularization constant  $\lambda$  is stored in `settings.regConst`.

You have to implement the gradient computation for five loss functions: squared loss, perceptron loss, log loss, hinge loss and exponential loss. See the course notes for definitions of each of these. I suggest you start off with squared or perceptron because those are the ones that are hardest to mess up. Once you have those working, you can work toward getting log loss, hinge loss and exponential loss together.

We can test our computation on a simple classification example: the “or” problem. I set the regularization constant to 0 to mean no regularization. I’m also not doing stochastic optimization:

```

>> model = linearModel('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.1, 100, 0, 'perceptron', 'l2', 0)
model =
  weights: [0.2000 0.2000]
  bias: -0.1000

```

```
>> linearModel('predict', model, [0 0;0 1;1 0;1 1])
ans =
    -1
     1
     1
     1
```

We can also try the `or` function with stochastic optimization. I get exactly the same solution. Interestingly, squared loss gives a fairly different response:

```
>> model = linearModel('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.1, 100, 0, 'squared', 'l2', 0)
model =
    weights: [1.0000 1.0000]
    bias: -0.5000
```

As do all the others:

```
>> model = linearModel('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.1, 100, 0, 'hinge', 'l2', 0)
model =
    weights: [2.1000 2.1000]
    bias: -1.1000
```

```
>> model = linearModel('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.1, 100, 0, 'log', 'l2', 0)
model =
    weights: [2.7987 2.7987]
    bias: -0.8527
```

```
>> model = linearModel('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.1, 100, 0, 'exp', 'l2', 0)
model =
    weights: [3.0716 3.0716]
    bias: -1.0605
```

**Please include in your writeup** a brief description of why these values are different, even though they all achieve perfect classification. (Hint: it's *not* because gradient descent hasn't converged or because of local minima or anything like that.)

### Task 3: Two Layer Neural Network

(25%)

Your second task is to do the same thing, but for a two layer network. Skeleton code is in `TwoLayer.m`. **Big Warning:** this is *hard*: it took me quite a few tries to get it right, mostly because of sign errors and annoyances like that. There are some debugging hints below that were really helpful for getting my solution working.

The network representation, etc., are all described in comments in the matlab code. Essentially, if we have  $D$  input dimensions, plus one for a bias, we have a “hidden” weight matrix of size  $(D + 1) \times K$ , where  $K$  is the number of hidden units. We then also have an “output” weight vector of length  $(K + 1)$ . The first component is the output bias. Since the gradient descent code expects everything in one component, there's some code in the skeleton to pack all of these into a giant matrix and then unpack it as needed. You shouldn't need to worry about the packing/unpacking.

**Hints to debug:** You might want to first check whether your output bias is training correctly. Then check whether your output weights are training correctly. Then check whether your input weights are training

correctly. I do this by using  $K = 1$  hidden units and forcing the initial weights to be  $\mathbf{w}_o(2) = 1$  and  $\mathbf{w}_h(1,:) = [-5 \ 10 \ 10]$ . Then, I have it just set  $\mathbf{g}_o(1)$  to something other than zero. In this way, I know it should be able to learn the correct function. Once I've done that, I add in the gradient for the rest of  $\mathbf{g}_o$  and then finally the gradient for  $\mathbf{g}_h$ .

Here's how my output looks, though note that due to randomness it varies. This is with only learning the output bias:

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.01, 100, 0, 0, 1)
model =
```

```
    wh: [-5 10 10]
    wo: [-0.0267 1]
 settings: [1x1 struct]
```

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.01, 100, 0, 0, 1)
model =
```

```
    wh: [-5 10 10]
    wo: [-0.1228 1]
 settings: [1x1 struct]
```

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.01, 100, 0, 0, 1)
model =
```

```
    wh: [-5 10 10]
    wo: [-0.1707 1]
 settings: [1x1 struct]
```

In all cases, I checked the predictions to make sure they were right. Basically, the first component of  $\mathbf{w}_o$  had better be negative.

Now I add in the gradient computation for the rest of  $\mathbf{g}_o$  and try again. I tend to get similar results, but where the second component of  $\mathbf{w}_o$  is not exactly one, but pretty close.

At this point, I remove the initialization of  $\mathbf{w}_o$  entirely, to see how well it fares (though I still initialize  $\mathbf{w}_h$ ). I continue to get the same thing.

Next, I add in the computation of  $\mathbf{g}_h$ . It seems not to move the weights very far from their initial points, so I randomly initialize them as well. I found that the learning rate/number of iterations I was using before don't quite work, so I changed to  $\eta = 0.05$  and 1000 iterations, and everything seems good: it learns a perfect classifier:

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.05, 1000, 0, 0, 1)
model =
```

```
    wh: [-0.5737 3.2634 3.2630]
    wo: [-2.1541 3.2881]
 settings: [1x1 struct]
```

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.05, 1000, 0, 0, 1)
model =
```

```
    wh: [-0.6635 3.0430 3.0433]
    wo: [-2.0364 3.2082]
 settings: [1x1 struct]
```

```
>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.05, 1000, 0, 0, 1)
model =
```

```

        wh: [-0.5793 3.2730 3.2730]
        wo: [-2.1476 3.2808]
    settings: [1x1 struct]

>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.05, 1000, 0, 0, 1)
model =
        wh: [-0.6258 3.1740 3.1735]
        wo: [-2.0888 3.2386]
    settings: [1x1 struct]

>> model = TwoLayer('train', [0 0;0 1;1 0;1 1], [-1 1 1 1]', 0.05, 1000, 0, 0, 1)
model =
        wh: [-0.5675 3.2912 3.2911]
        wo: [-2.1622 3.2924]
    settings: [1x1 struct]

```

Note, of course, that these are somewhat different values, but they all make the same predictions.

Finally, I tried increasing the number of hidden units to make sure everything still worked, and it did. Yay!

#### Task 4: Analysis (30%)

If all the above tests passed okay, it's probably not time to try the `run` script. This will produce a *large* amount of output.

This works just like before. Here are the tests we run through. The tests are identical for the two data sets. The times given are first for 20ng and then for MNIST.

- Try a linear model with log loss and no regularization parameter, plotting performance (train and test) as a function of iteration. *[10 seconds / 15 seconds]*
- Try the same thing but with stochastic optimization. *[40 seconds / 1 minute]*
- Try different learning rates. *[30 seconds / 30 seconds]*
- Do the same thing, but by varying the regularization parameter and keep the number of iterations fixed. *[12 minutes / 5 minutes]*
- Using a fixed regularization parameter, try different loss functions. *[4 minutes / 2 minutes]*
- Now, with the two layer model, run with no regularization and either 2, 5 or 10 hidden nodes. Plot performance as a function of iteration. *[30 minutes / 20 minutes]*
- Try the same thing but with stochastic optimization. *[30 minutes / 20 minutes]*

For each of these outputs, include the graph and describe in a short paragraph the results that you see and whether they make sense from the perspective of what we expect things like regularization and loss functions to do. Which loss function would you prefer? Would you prefer to regularize or just stop optimizing early? Which seems more stable? Is the two-layer network doing better than the single-layer network? Include answers to **all** these questions in your writeup.

#### Bonus Task: Do $\ell_1$ regularization (up to 10%)

Title speaks for itself. Modify the linear model to do  $\ell_1$  regularization. Check on the two data sets if you are indeed learning a sparser solution. Include the code in `linearModel.m` and include in a writeup a description of how you implemented in, plus a brief analysis of the outcome.