

Implementing gradient descent

Okay, now we know how to update our weights:

$$\Delta w_{ij} = \eta * \delta_j * x_i$$

You've seen how to implement that for a single update, but how do we translate that code to calculate many weight updates so our network will learn?

As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at <http://www.ats.ucla.edu/stat/data/binary.csv>). This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (numbered 1 through 4). Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.

Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.



The goal here is to predict if a student will be admitted to a graduate program based on these features. For this, we'll use a network with one output layer with one unit. We'll use a sigmoid function for the output unit activation.

Data cleanup

You might think there will be three input units, but we actually need to transform the data first. The rank feature is categorical, the numbers don't encode any sort of relative values. Rank 2 is not twice as much as rank 1, rank 3 is not 1.5 more than rank 2. Instead, we need to use [dummy variables](#) to encode rank, splitting the data into four new columns encoded with ones or zeros. Rows with rank 1 have one in the rank 1 dummy column, and zeros in all other columns. Rows with rank 2 have one in the rank 2 dummy column, and zeros in all other columns. And so on.

We'll also need to standardize the GRE and GPA data, which means to scale the values such that they have zero mean and a standard deviation of 1. This is necessary because the sigmoid function squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. Since the GRE and GPA values are fairly large, we have to be really careful about how we initialize the weights or the gradient descent steps will die off and the network won't train. Instead, if we standardize the data, we can initialize the weights easily and everyone is happy.

This is just a brief run-through, you'll learn more about preparing data later. If you're interested in how I did this, check out the `data_prep.py` file in the programming exercise below.

| | admit | gre | gpa | rank_1 | rank_2 | rank_3 | rank_4 |
|------------|--------------|------------|------------|---------------|---------------|---------------|---------------|
| 15 | 0 | -0.932334 | 0.131646 | 0 | 0 | 1 | 0 |
| 115 | 0 | 0.279614 | 1.576859 | 0 | 0 | 1 | 0 |
| 55 | 1 | 1.318426 | 1.603135 | 0 | 0 | 1 | 0 |
| 175 | 1 | 0.279614 | -0.052290 | 0 | 1 | 0 | 0 |
| 63 | 1 | 0.799020 | 1.208986 | 0 | 0 | 1 | 0 |
| 67 | 0 | 0.279614 | -0.236227 | 1 | 0 | 0 | 0 |
| 216 | 0 | -2.144282 | -1.287291 | 1 | 0 | 0 | 0 |
| 145 | 0 | -1.798011 | 0.105369 | 0 | 0 | 1 | 0 |
| 286 | 1 | 1.837832 | -0.446439 | 1 | 0 | 0 | 0 |
| 339 | 1 | 0.625884 | 0.210476 | 0 | 0 | 1 | 0 |

Ten rows of the data after transformations.

Now that the data is ready, we see that there are six input features: gre, gpa, and the four rank dummy variables.

Mean Square Error

We're going to make a small change to how we calculate the error here. Instead of the SSE, we're going to use the **mean** of the square errors (MSE). Now that we're using a lot of data, summing up all the weight steps can lead to really large updates that make the gradient descent diverge. To compensate for this, you'd need to use a quite small learning rate. Instead, we can just divide by the number of records in our data, m to take the average. This way, no matter how much data we use, our learning rates will typically be in the range of 0.01 to 0.001. Then, we can use the MSE (shown below) to calculate the gradient and the result is the same as before, just averaged instead of summed.

$$E = \frac{1}{2m} \sum_{\mu} (y^{\mu} - \hat{y}^{\mu})^2$$

Here's the general algorithm for updating the weights with gradient descent:

- Set the weight step to zero: $\Delta w_i = 0$
- For each record in the training data:
 - Make a forward pass through the network, calculating the output $\hat{y} = f(\sum_i x_i w_i)$
 - Calculate the error term for the output unit, $\delta = (y - \hat{y}) * f'(\sum_i x_i w_i)$
 - Update the weight step $\Delta w_i = \Delta w_i + \delta x_i$
- Update the weights $w_i = w_i + \eta \Delta w_i / m$ where η is the learning rate and m is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
- Repeat for e epochs.

You can also update the weights on each record instead of averaging the weight steps after going through all the records.

Remember that we're using the sigmoid for the activation function, $f(h) = 1/(1 + e^{-h})$

And the gradient of the sigmoid is $f'(h) = f(h)(1 - f(h))$

where h is the input to the output unit, $h = \sum_i x_i w_i$

Implementing with NumPy

For the most part, this is pretty straightforward with NumPy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is $1/\sqrt{n}$ where n is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

```
weights = np.random.normal(scale=1/n_features**.5, size=n_features)
```

NumPy provides a function `np.dot()` that calculates the dot product of two arrays, which conveniently calculates h for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

```
# input to the output layer
```

```
output_in = np.dot(weights, inputs)
```

And finally, we can update Δw_i and w_i by incrementing them with `weights += ...` which is shorthand for `weights = weights + ...`

Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, $f'(h) = f(h)(1 - f(h))$. That means that once you calculate $f(h)$, the activation of the output unit, you can use it to calculate the gradient for the error gradient.

Programming exercise

Below, you'll implement gradient descent and train the network on the admissions data. Your goal here is to train the network until you reach a minimum in the mean square error (MSE) on the training set.

You need to implement:

- The network output: `output`.
- The output error: `error`.
- The error term: `error_term`.
- Update the weight step: `del_w +=`.
- Update the weights: `weights +=`.

After you've written these parts, run the training by pressing "Test Run". The MSE will print out, as well as the accuracy on a test set, the fraction of correctly predicted admissions.

Feel free to play with the hyperparameters and see how it changes the MSE.

The code in code section