**Gradient Descent**

Guess I’ll put this here, as it kind of ties into the next topic: Regression. So say we have a bunch of points, **x**i, in some D-dimensional space, and associated values yi. I’ll take the yi to be a scalar I guess. And say we have a function f(**x**,**v**) which takes **x** as an argument (and contains a bunch of fitting parameters **v** not necessarily of the same dimensionality as **x**) and outputs a prediction for y. We want to work out the **v** which will make f a best fit hypercurve over the x data, in the least square error sense. That is to say, we want to find the **v** which minimizes,



One way to do this is the method of gradient descent. If we visualize E(**v**) as a surface in the **v** coordinate system. Then we can see that if we just follow the gradient dE/d**v**, we will come to the minimum. In case of 1 variable linear regression, looks like (MSE ~ cost ~ E):

Chart, surface chart

Description automatically generated

Well, follow the opposite of the gradient rather. So our algorithm would be:



where α is some step size (also called the learning rate), basically telling us how far along the gradient we’re going. Of course we don’t *know* the derivative ∂E/∂**v**. We have to approximate ∂E/∂v ≈ [E(**v**+δ**v**)-E(**v**)]/δ**v**. Not sure what we’d use for δ**v**, but something small. Note that the step size, α|∂E/∂**v**|, is getting smaller as we approach the minimum, since the slope is concomittantly getting smaller. This actually helps (doesn’t guarantee) us to not miss the minimum. In practice, we’d run this algorithm until the step size, α|∂E/∂**v**|, is less than 0.001, say. And/or we’d impose a maximum number of iterations.

**Example**

So say we’re just doing a linear regression. Then f = **v**1∙**x** + v0,



and,



and,



So then given some initial guess, and step size, we’d say:



Let’s look at another example,

**Example**

Say we have data:



And let’s try fitting function:



Loss function is:



And we’d start with some guess, say:



So then,



and,



So yeah.

**Stochastic Gradient Descent**

Reviewing those **v** formulas for the linear interpolation, can see that if we have N = 106 data points, for instance,

Chart, scatter chart

Description automatically generated

then this could involve a lot of calculations. Stochastic Gradient Descent tries to surmount this problem by just sampling n << N data points. So I guess it would update the formulas to, for instance:



Note the multiplication N/n is necessary so that batches of different n sizes can be handled together. One could formally absorb N/n into the α, and so we could just write:



Anyway, this modification seems to work pretty well. I guess when we group data into batches, like in Tensorflow or PyTorch, we are computing gradient descent each batch at a time. So it’s kind of like we’re implicitly using stochastic gradient descent?

**Scaling Data**

In practice, these algorithms run into a problem. Consider the recursion relations in a prior example,



So for instance Σi=1n xi2[fn(xi) – yi] can be huge. And it’d be nearly impossible to find a step size which is small enough to avoid jumping from one side of the minimum to the other, and large enough to not require a million iterations to converge. I guess this is why we need to scale the data. So another example. Say we have a simple linear regression problem,



We can map our data:



Then,



Solving for v, we have:



So we have a new problem,



We minimize E(v´), and once v´0 and v´1 are found, we can solve for v0 and v1. We can generalize this to a vector linear regression,



We just make the mappings,



We can write this in vector notation as:



where the funky multiplication symbol is defined as,



which is how numpy multiplies vectors. And then we have:



So then, again, we have a new problem to minimize,



and once the **v**´ are found via minimization, we can solve for the **v**. Note the /σx expression is to be interpreted as,



Well what if we have a general matrix relation, as in a neural net? Consider,

Diagram

Description automatically generated

Then f will be a complex function of x1,2,3. But dependence on x1,2,3 is through x(1)1,2,3,4. Can say,



Note we could write this in matrix notation as:



where,



So rows of W are enumerated by x(1) and columns of W by x. And rows of B are enumerated by columns of x(1), while columns are just 1. And we can say f = f(X(1)). And E(W(1),B(1);W(2),B(2);W(3),B(3)) will depend on W(1) and B(1) only through f(X(1)). In so far as we’re only concerned with E’s dependence on W(1) and B(1), we can say, E = E(W(1)X(0)+B(1)). So question is, how do W(1) and B(1) best fit parameters for X(0) relate to the best fit parameters W´(1) and B´(1) for Z(0), i.e., scaled X(0)? So on one hand, we have:



And on the other:



Remember the x1,2,3,..,n guys are drawn from columns 1, 2, 3, …,, n. So they all have different means and stds. So we have,



which must be true for all x’s. So (Einstein summation notation),



and therefore, switching out of Einstein summation notation,



These equations say that to get the unscaled W(1), we take the scaled one, W´(1) and divide all the numbers in column n by the std of the elements in column n of the training data. And we do similarly for Bm(1).

But what if we have a non-linear regression? Nothing changes if we have non-linear dependence on **x**i, as long as we still have linear dependence on the fitting parameters. For instance, if our f(x) is:



We could try to scale by σx and μx still. Maybe that’d be good enough. So it’d work like this:



We can map our data:



Then,



Yeah I don’t know. In fact, this approach doesn’t seem to work that great. Maybe better to redefine our independent variable x2 as χ, and write:



and then repeat the process above, scaling by σχ and μχ instead. What if we had something like,



Then I think we’d have to make a choice about which variable to normalize, either the x, or the x2. What if we’re doing softmax regression? Then our f is:



I think it’d still be worthwhile to do the scaling on x, and write:



Then,



Then can do the logistic regression in terms of the v´ variables. And once found, convert back. So it seems that gradient descent works best when the fitting function is an arbitrary function of a **linear combination** of the independent variables. What about when applying gradient descent to something like,



Well, we’d do:



and when minimizing,



The last two say, letting F(z) be ln(1+ev´\_0 + v´\_1·z),

 

And our solution is, from Cramer’s rule:



Last comment: seems really important for numerical convergence that we don’t have redundant d.o.f., like w­1­f(w2x+b2) + b1 + w3f(w4x+b4) + b3, where b1 and b3 are redundant.