Gradient Descent in Functional Space

Gradient Descent

Remember gradient descent? We used it to minimize a loss function $\mathcal{L}$ with respect to some parameters $\mathbf{w}$ of the model (for example, to learn a logistic regression classifier). Each iteration we updated the parameters $\mathbf{w}$ with a gradient update:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$$

Here, $\alpha > 0$ was the step size and $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ the gradient of the loss function $\mathcal{L}$ with respect to the parameters $\mathbf{w}$.

☆ Key Points

Gradient boosting (as the name suggests) is highly related to gradient descent.

In each iteration of boosting, we add a weak learner that adjusts the training predictions to be closer to the true labels.

Boosting is a very general concept with many instantiations, some of which have specialized names such as GBRT, LogitBoost, and AdaBoost. All other than GBRT are out-of-scope for this course.

Gradient Boosting and Gradient Descent

Gradient boosting is quite similar to gradient descent. We are minimizing a loss function $\mathcal{L}$, not with respect to a parameter vector, however, but with respect to a classifier H. We assume that this classifier is an ensemble:

$$H(\mathbf{x}) = \sum_{j=1}^{m} \alpha h_j(\mathbf{x})$$

Boosting is an iterative algorithm, and each iteration we learn a new function h_j that we add to the ensemble. In other words, the iterative update per iteration is

$$H \leftarrow H + \alpha h_j$$

If you compare this equation with the update from gradient descent, you should realize an undeniable similarity. So what should h_j be in each iteration? Similar to the argument we made to derive gradient descent, h_j should resemble the negative gradient. That is,

$$h_j \approx -\frac{\partial \mathcal{L}}{\partial H}$$

Gradient With Respect to a Function

It may seem strange to think of the gradient of a loss function with respect to a function H. How exactly are we supposed to do this? Well, as it turns out, we don't have to. At the end of the day, we are only evaluating H on the training data points, so we can **pretend** that a function is really just an n-dimensional vector of the predictions on the n training points, i.e., $H(\mathbf{x}_1), \ldots, H(\mathbf{x}_n)$. To illustrate this, let us take a step back and reconsider the loss function $\mathcal{L}$. A loss function is essentially a measure of how well a classifier performs on the inputs in a given data set. For example, we can define the squared loss as

$$\mathcal{L}(H) = \frac{1}{2} \sum_{i=1}^{n} (y_i - H(\mathbf{x}_i))^2$$

Note that it evaluates the function H only on the n training inputs. So, as far as the loss $\mathcal{L}$ is concerned, we can view H as an n-dimensional vector. Suddenly it is no longer so weird if we take the derivative $\frac{\partial \mathcal{L}}{\partial H}$; it is nothing else but the derivative with respect to a vector — similar to taking the derivative with respect to the parameter vector $\mathbf{w}$ in gradient descent.

Gradient Approximation

To summarize, we regard the ensemble classifier as an n-dimensional vector $H \in \mathcal{R}^n$, where the i^{th} dimension corresponds to the prediction of H on input $\mathbf{x_i}$. We can now compute the gradient $G = \frac{\partial \mathcal{L}}{\partial H} \in \mathcal{R}^n$, where $G_i = \frac{\partial \mathcal{L}}{\partial H(\mathbf{x_i})}$. As an example, let us consider $\mathcal{L}$ to be the squared loss as defined above. Then we obtain

$$G_i = H(x_i) - y_i$$

This quantity is easy to compute for each training input.

But there is a catch!

If functions really were just n-dimensional vectors, we could simply perform an update and subtract G from H. Unfortunately, that's not true.

H is still a function, and we can only add functions together (not functions and vectors). Consequently, So what we need is a function h_j that behaves just like the negative gradient $-G$. In other words, $H \leftarrow H + \alpha h_j$ should have a similar effect on the evaluations to taking a gradient descent step in the n-dimensional vector space $H \leftarrow H - \alpha G$.

Treating H as a function in one place and as a vector in another can be confusing. The important idea here to grasp is that H is and will always remain a function, but one that we want to evaluate to $H(\mathbf{x}_1), \ldots, H(\mathbf{x}_n)$. G could help H move in the direction towards the evaluation goal. But, since G is a vector of evaluations and H a function, we cannot perform a mathematical operation between them.

To this end, we try to find a function h_j that mimics $-G$ on all training points. Formally, we are trying to find

$$h_j = \arg\min_{h \in \mathbb{H}} \sum_{i=1}^{n} \left(h(\mathbf{x}_i) - t_i\right)^2$$

where $t_i = -G_i$, the negative gradient with respect to $H(\mathbf{x}_i)$; i.e., $t_i = -\frac{\partial \mathcal{L}}{\partial H(\mathbf{x}_i)}$. The resulting function h_J is similar to the negative gradient on all training points; thus, if we add it h_j to the ensemble H, we are essentially taking a gradient step to bring the training predictions closer to the true labels.

Gradient Boosted Regression Trees

In GBRT, we use the squared loss. The gradient then becomes $G_i = H(\mathbf{x}_i) - y_i$ and $t_i = y_i - H(\mathbf{x}_i)$. This term has a very nice interpretation: The negative gradient is the "residual" of the current classifier H. In other words, t_i measures the difference between the prediction of the current ensemble classifier and the label for each training data point. The weak learner h_j in the next iteration will learn to predict that difference, and if we add it to the ensemble, we are taking a small step towards the vector of true labels. One beautiful aspect of GBRT is that the search for h_j becomes a simple call of the CART algorithm, with inputs $(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_n, t_n)$.

Other Boosting Algorithms

There are many variations of boosting. For example, if the loss function $\mathcal{L} = \sum_{i=1}^{n} \log\left(1 + e^{-y_i H(\mathbf{x}_i)}\right)$ is the logistic loss function, then we obtain **LogitBoost**. Or if the loss function is the exponential loss $\mathcal{L} = \sum_{i=1}^{n} e^{-y_i H(\mathbf{x}_i)}$, we obtain **AdaBoost** (where the step size α is found via a closed-form line search). In AdaBoost the weak learner is typically found by optimizing $h = \mathbf{argmax}_{h \in \mathcal{H}} \sum_{i=1}^{n} h(\mathbf{x}_i) t_i$. The negative gradient takes on the form $t_i = -y_i e^{-y_i H(\mathbf{x}_i)}$. If the labels and the outputs of the weak learners are restricted to +1,-1, this can be interpreted as minimizing the weighted training error — another intuitive interpretation of boosting (although the details are slightly beyond the scope of this module).

Remark About Loss Functions

We must remind you that there are **two** loss functions being considered in gradient boosting. There is the global loss function $\mathcal{L}$ that the ensemble classifier H is minimizing, and there is the loss that the weak learner h_j is minimizing. We calculate the global loss function on the predictions-label $(H(\mathbf{x}_i), y_i)$ pairs, while the weak learning algorithm calculates its loss on $(h_j(\mathbf{x}_i), t_i)$ pairs. Often both of them are the squared loss, but they don't have to be. For example, you could choose the logistic loss for $\mathcal{L}$ and then use standard regression trees that minimize the squared loss to find the weak learner h_j. This allows you to minimize a *classification loss* using *regression trees*.