Gradient-Based Optimization Notes

Piyush Patil

September 19, 2017

1 Introduction

Gradient descent is a general, iterative optimization algorithm for computing minima of differentiable functions. The idea is simple - let's say we have a differentiable (and therefore continuous) function mapping n-dimensional space into the reals. We can visualize this graphically as an $(n+1)$-dimensional surface, with hills, valleys, saddles, and other such features giving the function the terrain it has. The gradient descent algorithm exploits the definition of the gradient (the generalization of the derivative to more than one dimension in the domain) of the function - since the gradient at a point is in the direction of steepest ascent, by going the exact opposite direction, we'll end up at a point lower on the function than the one we started at. We can then repeat this at the new point to continue finding lower and lower points; visually, on the terrain of the function's surface, we're just following the downward direction until we get to a valley. To avoid overshooting valleys and going to far, though we follow the direction of the negative gradient, we attentuate the magnitude of the step with a parameter called the learning rate. By iteratively taking such steps and updating the point we're at, eventually we should converge to a valley. The update rule for gradient descent can hence be written

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla f(\theta^{(t)})$$

where f is the function to be optimized, $\theta^{(t)}$ is the point at iteration n, and η is the learning rate. What follows below are various optimizations and modifications of the gradient descent algorithm which build on the original idea but try to further mitigate two main problems with the vanilla technique: (1) in very high dimensional space, the algorithm might converge slowly, and it's difficult to find a good learning rate that balances quick convergence with the avoidance of overshooting a minimum; (2) unless the function is convex (in which case the terrain has only a single valley), gradient descent will usually end up in a local minimum, not a global one, and this depends entirely on the starting point we use, $\theta^{(0)}$.

In practice, we have a loss function $L_\theta : X \times Y \to \mathbb{R}^+$ which depends on parameters θ, which we're optimizing over, and maps input-output combinations, from input space X and output space Y respectively, to positive real loss values. We want to find parameters θ which minimize this L the entire space $X \times Y$. Of course, in practice we don't have access to the full input space, and sometimes not the full output space either, so we're forced to approximate these spaces using finite training data. Hence we are unable to compute the gradient of L with respect to θ, but given training data $\{(x_i, y_i), 1 \leq i \leq N\}$ we can approximate it as

$$\nabla_\theta L_\theta \approx \frac{1}{N} \sum_{i=1}^{n} \nabla_\theta L_\theta(x_i, y_i)$$

An issue with this approach is that it requires computing the gradient over the entire training dataset, which is computationally difficult. We might be willing to sacrifice some accuracy in our approximation of the gradient if it gives us a much faster way of actually computing the gradient approximation. This is the idea behind *batch gradient descent* - partition the training data into small batches, and iteratively perform gradient descent on randomly chosen batches. Computing the gradient over a batch is much faster than computing it for the entire dataset, simply because batches have only a fraction of the data points, and even though using less data points adds random noise into the gradient approximation, if the training data is indeed a random sample of the true input space X and the batches are indeed randomly selected, in expecatation the random noise should disappear anyways. *Stochastic gradient descent* takes this idea to the extreme, computing gradients on single data points (i.e. a batch size of 1).

The random noise introduced by batch gradient descent, and especially in stochastic gradient descent, can actually be helpful. As we stated above, a major problem with gradient-based optimization methods is their tendency to get stuck in local minima. This is the result of the algorithm blindly following the steepest, "most downhill" direction at every point. If we sprinkle in some random noise to that process, we enable the algorithm to spontaneously "tunnel" through local minima, potentially finding slopes beyond local minima that take it to deeper local minima or even a global minimum.

Let's now turn to algorithms that try to optimize gradient descent. Most the methods below are not mathematically rigorous or theoretically sound, but rather based on intuitive, heuristic ideas about the terrain of the loss function, and have been shown to work well in practice.

2 Momentum

This update method is based on the inuition that the steeper the loss landscape is, the faster we should try to converge. Traditional SGD has trouble with parts of the loss landscape in which steepness has a lot of variance in different dimensions, and zigzags around while only reluctantly forging a path towards the local minimum. Momentum tries to ameliorate this by appealing to the intuition of a ball rolling down the loss landscape; as the landscape gets steeper, the ball gains momentum and rolls downwards even faster, so that the rate of convergence compounds. If the ball overshoots a local minimum, it'll start rolling upwards and its momentum rapidly dissipates. We can implement this idea by having our update rule account not only for the gradient but also for the previous "velocity" of convergence.

$$v^{(t+1)} \leftarrow \gamma v^{(t)} + \eta \nabla_\theta L_{\theta^{(t)}}$$
$$\theta^{(t+1)} \leftarrow \theta^{(t)} - v^{(t+1)}$$

where γ is another factor similar to the learning rate which controls the degree to which we want to rely on the momentum.

3 Nesterov accelerated gradient

This is a slight optimization to momentum. As our ball rolls down the loss landscape, we want it to have some intelligent notion of not just where it currently is, but where it will be in the future, so it "knows" to slow down if it sees an upward slope approaching. In momentum, recall that our update rule can be written in the condensed version

$$\theta_{t+1} \leftarrow \theta^{(t)} - \gamma v^{(t)} - \eta \nabla_\theta L_{\theta^{(t)}}$$

This is simply the original gradient descent update rule, but we shift our point using the gradient by starting not at θ_t, the previous location, but at the location $\theta_t - \gamma v_t$, the location we reach from the previous location after accounting for our "existing" momentum. To make our ball more intelligent, we can have it compute the steepness of the loss function not at its previous location, but at the location it's going to be at, accounting for momentum. Hence, the update rule becomes

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma v^{(t)} - \eta \nabla_\theta L_{\theta^{(t)} - \gamma v^{(t)}}$$

4 Adagrad

This is an optimization of gradient descent which works on a more fine-grained, per-parameter basis. Whereas gradient descent by default computes one single update that's distributed to every parameter, Adagrad avoids computing such a one-size-fits-all update. Instead, it tries to personalize updates with the following intuition: sure, we can compute the same update vector using the gradient for every parameter in the parameter vector θ, but the degree to which we attentuate this update using the learning rate should depend on how frequently we update a parameter. The parameter vector θ consists of many parameters, and parameters that aren't frequently updated and have remained more or less the same over time require larger updates than parameters which have been rapidly changing. Thus, Adagrad implements gradient descent using a local learning rate for each parameter, instead of one global η that's used for the entire parameter vector. Let's now look at how we can mathematically frame this intuition. First, let's say our parameter vector looks like

$$\theta = \left[\theta_i\right]_{1 \le i \le n}$$

so that we have

$$\nabla_\theta L_\theta = \left[\tfrac{\partial L}{\partial \theta_i}\right]_{1 \le i \le n}$$

The per-parameter update rule for gradient descent can be written

$$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} - \eta \nabla_{\theta_i} L_{\theta^{(t)}}$$

We modify this by looking at the past gradients that have been computed, with respect to θ_i and modifying η so that the larger the past gradients have been, the smaller it is, and vice versa. Note here that because we're looking at per-parameter updates, the gradients here, are scalars, not vectors. One simple, convex, and differentiable way of aggregating past gradients is inspired by using the Pythagorean theorem to compute the total "length" of the vector induced by all the past gradients:

$$g_i^{(t)} := \sum_{t=0}^{t-1} \left(\nabla \theta_t L_{\theta^{(t)}}\right)^2$$

so that $\sqrt{g_i^{(t)}}$ is the length we want. Hence, our per-parameter update rule becomes

$$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)}}} \nabla_{\theta_i} L_{\theta^{(t)}}$$

We can now vectorize this into a full parameter update using the following trick: consider the sum of outer products of the gradients:

$$G^{(t)} := \sum_{t=0}^{t-1} \left(\nabla_\theta L_{\theta^{(t)}}\right) \left(\nabla_\theta L_{\theta^{(t)}}\right)^\intercal$$

Clearly, $G^{(t)}$ is diagonal (after all, it's the sum of outer products, each of which is diagonal by definition). Further, notice that we have $G_{i,i}^{(t)} = g_i^{(t)}$. Thus, we can write the full parameter update as

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \frac{\eta}{\sqrt{G^{(t)}}} \odot \nabla_\theta L_{\theta^{(t)}}$$

where $\odot$ is the Hadamard (i.e. element-wise) product.

5 Adadelta

The main problem with Adagrad is that over time, the elements of the diagonal matrix $G^{(t)}$ grow larger and larger without bound, and hence the per-parameter learning rates all asymptotically approach zero. Hence, though learning is more optimized than having a single global learning rate, learning will inevitably slow down over time. Adadelta tries to address this problem by using an exponentially decaying moving average over past gradients, relative to some fixed window size, rather than a straight sum of squares. Ideally, one would use the past several gradients, for some window size, but using an exponentially decaying moving average instead of a strict window keeps the update rule differentiable. Furthermore, it allows us to compute the window without having to actually inefficiently store all the past gradients. TODO

6 RMSprop

7 Adam

8 AdaMax

9 Nadam