ECE472 – Quiz 1

Jonathan Lam

September 16, 2020

1. *Explain the key differences between Adam and the basic gradient descent algorithm.*

   The basic stochastic gradient descent algorithm takes the gradient of the loss function based on a batch of samples, and it changes all of the variables in the direction of the negative gradient. The basic idea of the gradient descent doesn't mention much about step size or convergence, which could easily become problematic when the gradient is not smooth or somewhat irregular ("pathological curvature"). The easy thing to do (and this basic approach works for the first two homework assignments) is to assign a small value to the learning rate α and assume that the gradient of the objective (loss) function is smooth enough that there are no problems with the descent. When there are problems with convergence, you can manually make the step size smaller (this was my naive approach to nonconvergence for homework 2, in which I decreased α by a factor of 2 every 1000 epochs).

   The main difference for Adam descent is that it stores a weighted gradient average (very similar to momentum) and a weighted gradient-squared average (which can be loosely interpreted as an (uncentered) "variance"). These averages are calculated much like momentum is, with the current weight calculated from the previous weight (multiplied by a multiplicative decay factor) combined with the current gradient (or gradient-squared) – this in itself offers the benefit of momentum, similar to RMSProp. (It's important to note that this operations is not very expensive, as it is linear in time and space w.r.t. the number of coefficients, rather than the size of the input, and only involves the first-order gradient like regular stochastic gradient descent.) (These weighted averages are scaled by a factor to correct their initialization bias.)

   Using these calculated values, Adam descent has a different update rule: instead of subtracting the gradients (multiplied by some learning factor) from the weights, the update rule is:

   $$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

   where θ_t is the vector of weights at the current iteration, θ_{t-1} is the vector of weights at the previous iteration, α is the (roughly) upper limit on the learning rate, $\hat{m}$ is the vector of initialization-bias-corrected weighted gradient averages, $\hat{v}$ is the vector of initialization-bias-corrected weighted gradient "variances", and ϵ is a small nonzero constant to avoid division by zero.

   The reason we have this more complex update rule is that it is "automatically annealing." If we treat $\hat{m}$ as roughly the gradient (with some momentum), and treat $\hat{v}$ as some sort of variance/noise/uncertainty factor, then $\hat{m}$ denotes the (negative of) the direction we should

be moving towards to minimize the objective, and $\hat{v}$ indicates how sure we are of moving in the correct direction. The greater the gradient $\hat{m}$, the further we should move; the larger the uncertainty $\hat{v}$, the less we should move. This "intelligently" sets the step size, and it does so for each weight coefficient by considering its gradient and gradient-squared.

Other benefits of Adam is that it converges well and works well on sparse matrices (similar to AdaGrad). The cost is that each iteration is slightly more complex than that of a basic stochastic gradient descent scheme, but this cost is likely greatly outweighed by Adam's benefits.

2. *Why does momentum really work?*

We have the same kind of convergence problem that was mentioned in the above answer when choosing the learning rate α. We want the largest learning rate possible without causing issues with convergence.

The article "Why Momentum Really Works?" illustrates the point mathematically by making an eigenvalue decomposition of two problems. These problems are simple enough that we can essentially solve the problem in closed form by considering each parameter separately, and the article shows how this can be used to find the general update rule for the model variables. This turns out to limit the possible range of α by the minimum and maximum eigenvalues of the feature matrix. When the analysis is repeated with the momentum update rule rather than the plain update rule:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \nabla f(\theta_{t-1}) \qquad \text{(Ordinary GD update rule)}$$

$$\begin{cases} z_t \leftarrow \beta z_{t-1} + \nabla f(\theta_{t-1}), \\ \theta_t \leftarrow \theta_{t-1} - \alpha z_t \end{cases} \qquad \text{(Momentum GD update rules)}$$

where z_t is the weighted average after iteration t, then α can be increased further without divergence, leading to faster divergence. When the learning rate α and the momentum coefficient β have optimal values, then the condition number κ (which indicates convergence rate, lower is better) is roughly the square root of the κ without momentum (which means a much faster convergence).

That approach is purely mathematical, but intuitive understanding is given by the layman's analogy in physics: a damped spring. α multiplied by the eigenvalues λ_i of the problem matrix can be interpreted as an driving force, and β is analogous to the damping coefficient. As with damped harmonic oscillator problems, there is an optimal β, which is the critically-damped scenario. Without damping (and without manually decreasing α as t increases), it's likely that there will be large oscillations around complicated gradients such as "ravines" (leading to slow convergence) and may settle in local minimas (leading to a nonoptimal solutions). With momentum, however, we have some information about gradients from past iterations, so we have a less-local view of the overall gradient topology and is less likely to have these problems. These intuitive robustness of momentum translates to the mathematical advantage of being able to increase the learning rate without having convergence issues.