Adagrad – eliminating learning rates in stochastic gradient descent

Posted on January 23, 2014 by xcorr

Earlier, I discussed how I had no luck using second-order optimization methods on a convolutional neural net fitting problem, and some of the reasons why stochastic gradient descent works well on this class of problems.

Stochastic gradient descent is not a plug-and-play optimization algorithm; it requires messing around with the step size hyperparameter, forcing you to expend a lot of energy getting the optimization to work properly, time probably better spent considering different model forms or novel analyses. Since deep neural nets have become very popular, a lot of research has goneinto eliminating learning rates within the context of stochastic gradient descent.

Adagrad is a theoretically sound method for learning rate adaptation which has has the advantage of being particularly simple to implement. The learning rate is adapted component-wise, and is given by the square root of sum of squares of the historical, component-wise gradient. Pseudo-code:

```
master_stepsize = 1e-2 #for example
fudge_factor = 1e-6 #for numerical stability
historical_grad = 0
w = randn #initialize w
while not converged:
E,grad = computeGrad(w)
historical_grad += g^2
adjusted_grad = grad / (fudge_factor + sqrt(historical_grad))
w = w - master_stepsize*adjusted_grad
```

Simple enough, right? Empirically, adagrad works much better – i.e. convergence is faster and more reliable – than simple SGD when the scaling of the weights is unequal. It is also not very sensitive to the master step size; just find some value which converges in a reasonable amount of time and leave it as is.

Adagrad has the natural effect of decreasing the effective step size as a function of time. Perhaps you have good reason to use your own step-size decrease schedule. In this case, you can use a running average of the historical gradient instead of a sum. Pseudocode:

```
autocorr = .95 #for example
master_stepsize = 1e-2 #for example
fudge_factor = 1e-6 #for numerical stability
historical_grad = 0
w = randn #initialize w
while not converged:
E,grad = computeGrad(w)
if historical_grad == 0:
historical_grad = g^2
else:
historical_grad = autocorr*historical_grad + (1-autocorr)*g^2
adjusted_grad = grad / (fudge_factor + sqrt(historical_grad))
w = w - master_stepsize*adjusted_grad
```

Here’s an interesting practical guide to learn more.