Paper Summaries

Paper Summaries

By James Camacho

Adam: A Method for Stochastic Optimization

The thing that makes Adam unique is confidence. Other gradient descent optimizers decide which direction to go, but don't know when to stop going. Simple gradient descent can take forever to converge, but adding momentum makes it go right past the optimum and keep going. Adam, on the other hand, calculates how confident it is in its current gradient.

The authors called it the "signal-to-noise ratio" (SNR), basically mean divided by variance of the gradient's historical values. If a parameter is near its optimal value, then the gradient for the parameter should fluctuate around zero, so the SNR will be near zero. Obviously the gradient will be pretty consistently large and with the same sign when the parameter is just starting out, leading to a large SNR (approximately one). When multiplied with the learning rate, it achieves learning with confidence.

Computing the true mean and variance would require storing the gradient each optimization step, which is incredibly memory intensive, so they suggested a weighted mean (with exponential decay on the weights) instead, much like a q-table. It provides a decent approximation, plus has the benefit of giving newer data more relevance. Unfortunately, they initialized it to zero, which only adds a bias they spend half the paper correcting. They should have initialized it to the first gradient, and adjusted the update formula appropriately, which would result in a cleaner algorithm.

Anyway, that's mostly what the paper is about. They briefly mentioned using different norms for the SNR (such as the infinity norm, giving AdaMax). I find it quite amusing how they got lost in the math notation and forgot what the infinity norm is actually doing. It's just a maximization on the (absolute-valued) gradient, they could've completely skipped equations (8)–(11). Oh well.

Note: To be fair, it's a lot easier to see everything they could've done better in hindsight. But I still think they should've solved these issues before publishing the paper!

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Dropout is a form of noise to prevent nodes from relying on each other. To paraphrase, if no node can rely on any other node, then each node must get good at what it does. The particular noise most explored in this paper is ”dropping out” neurons (who could've guessed?!), i.e. removing them from the neural net randomly. The most accurate way to use the net would be to average the results from several hundred runs of dropout, but a good approximation in production is to just scale down the weights to the dropin rate.

The authors demonstrated state-of-the-art results almost everywhere. The only thing dropout could not beat was Bayesian neural nets, but that's just because those are a generaliztion of dropout--they use completely uncorrelated neural nets and priors rather than correlated ”thinned” nets and averages.

One interesting generalization they briefly mentioned was other kinds of noise. Instead of multiplying weights by a Bernoulli random variable (zero or one, dropped out or still in), you could multiply it by any random variable! In fact, using a normal distribution with mean one seemed to work even better.

Okay, time for my rant about dropout. It doesn't work like biology! The authors mentioned how genes are randomly selected in sexual reproduction. What they failed to mention was linked genes. Neucleotides aren't chosen completely arbitrarily, they're usually linked together forming groups. Dropout is great at preventing nodes from becoming too reliant on each other, but it fails to let them work together. They observed dropout rates of 0.5 as near-optimal. How can neurons form strong linkages when their links are being severed half the time?

I would be interested to see how linking neurons aids dropout. One possibility is to build an asymmetric covariance matrix for the noise, rather than giving each node its own noise. The covariance matrix can be updated based on how well the network performs when one node is dropped but not the other by using gradient descent. Ideally you would hold the weights/biases constant, but I don't think it would be necessary. The covariance would probably go to 1, so a regularizer should be added.