Lecture 11

Stochastic Algorithms

11.1 Lecture Objectives

- Understand some of the interest in introducing randomness in optimization algorithms
- Develop a basic understanding of the types of stochastic algorithms used to train neural networks
- Understand the blurred lines between formulation and algorithm that result from certain algorithmic choices (eg: dropout in the training of neural networks)

11.2 Stochastic Gradient Descent

11.2.1 Problem formulation

Suppose we have an optimization problem $\min_{\mathbf{x}} f(\mathbf{x})$ where the cost function $f(\mathbf{x})$ is a sum of many individual components:

$$f(\mathbf{x}) = \sum_{m=1}^{M} g_m(\mathbf{x}) \tag{11.1}$$

where each individual component g_m will typically represent the fitting error for an individual data point. For instance, in least-squares fitting problems, g_m may take the form

$$g_m(\mathbf{x}) = |\phi(\mathbf{x}; \theta_m) - s_m|^2 \tag{11.2}$$

where we try to fit a set of acquired data points s_m using a signal model that depends on our unknown parameter vector $\mathbf{x}$ as well as on some design parameters θ_m, For example, θ_m could represent k-space location in an MRI reconstruction problem, and $\phi(\mathbf{x}; \theta_m)$ could represent the Fourier Transform of our unknown image $\mathbf{x}$ at k-space location θ_m. For such

a problem, we could simply apply a gradient descent algorithm, where the gradient can be expressed as:

$$\nabla f(\mathbf{x}) = \sum_{m=1}^{M} \nabla g_m(\mathbf{x}) \tag{11.3}$$

By evaluating this gradient at each iteration as part of an iterative algorithm, we can perform gradient-based optimization of this cost function. Note that we can express the gradient of the overall cost function as the sum of the gradients of the individual components. In this way, we could solve this problem using the gradient based techniques we have covered in previous lectures. However, if M is truly enormous, evaluating this gradient can be computationally very expensive (and, remember, we typically need to evaluate this gradient many times as part of an iterative optimization algorithm - at least once per iteration). For some imaging applications (including the training of machine learning models), M can be huge (on the order of millions, or even billions). Therefore, as datasets and optimization problems keep increasing in size, there is also enhanced interest in algorithms that do not require evaluation of the 'entire' gradient including a sum over M terms.

11.2.2 Randomizing the gradient components

Stochastic gradient descent (SGD) consists of approximating the gradient direction by using, at each iteration t in a descent algorithm, only a single randomly-chosen component of the gradient, ie: uses $-\nabla g_m(\mathbf{x})$ as the descent direction at each iteration. Note that this descent direction will be much faster to compute (by a factor of $\sim M$) compared to the 'full' gradient-based descent direction $-\nabla f(\mathbf{x})$. However, it is likely that this single-component direction will be a very coarse approximation to the gradient descent direction. Indeed, it is quite likely that $-\nabla g_m(\mathbf{x})$ may not be a descent direction at all for the overall cost function $f(\mathbf{x})$). For this reason, an intermediate approach between the full gradient and the single-component gradient is often used. In this intermediate approach, at each iteration t we pick a random subset (mini batch) of observations (let us call it Γ_t), where the size of Γ_t is much smaller than M. Then, we use this random subset of the M components in the evaluation of the gradient:

$$\nabla_{\Gamma_t} f(\mathbf{x}) = \sum_{m \in \Gamma_t} \nabla g_m(\mathbf{x}) \tag{11.4}$$

where a different random subset Γ_t is picked for each iteration t. In practice, this intermediate approach has been shown to often produce excellent results for the optimization of functions with very high components.

What is the optimum size of the subset Γ_t? To the best of the instructor's knowledge, there is no general answer to this question. In fact, mini batch size is one of the parameters that machine learning practicioners typically tune when attempting to make sure their model training converges.

11.3 What Else Can We Randomize?

11.3.1 Random initialization

What initial guess should we use to get our optimization algorithm started? Often times in this course, we have chosen to initialize with $\mathbf{x}^(0)$ equal to all zeroes. This is OK in many cases (eg: in convex optimization it should not matter too much), but there are cases when a random initialization helps get the algorithm up and running. Training of neural networks is one such case, where random initialization is a popular choice.

11.3.2 Dropout in the training of neural networks

Neural network training can be viewed as an optimization problem where we seek the set of network parameters $\mathbf{w}$ such that we minimize some measure of fit error relative to training data (inputs $\mathbf{x}$, outputs $\mathbf{y}$). In other words, the training can be viewed as solving the minimization problem:

$$\min_{\mathbf{w}} f(\mathbf{w}; \mathbf{x}, \mathbf{y}) \tag{11.5}$$

where our cost function reflects how well our model fits our training data.

Neural networks generally have an input layer, followed by possibly multiple hidden layers, followed by an output layer. Each of these layers contains a set of nodes that perform simple operations (eg: ReLU, convolutions, etc). These nodes have corresponding parameters ($\mathbf{w}$) that need to be trained. Oftentimes, neural networks are so large that they tend to overfit: we can pick the network parameters such that we fit the training data so well that we end up fitting the noise and spurious features in the training data, rather then just the meaningful features we are interested in. Overfitting is typically reflected as very small training errors, but large test errors (ie: poor generalizability).

A common approach to avoid overfitting is regularization, where we add a penalty term to our parameters during training, eg: we add $\lambda\|\mathbf{w}\|_p$ to our cost function. This penalty term will generally preclude the weights $\mathbf{w}$ from taking on very large values, therefore helping avoid overfitting. This approach is in line with the contents of our previous lectures, where the formulation (cost function) is neatly separated from the algorithm (that just seeks to minimize the cost function as efficiently as possible).

Another common approach to regularization, termed dropout, blurs this neat separation between formulation and algorithm[1]. In dropout-based algorithms[2], we do not alter the cost function, but instead we modify the optimization algorithm. At each iteration in our algorithm, we drop out (eliminate) a certain randomly chosen subset of the nodes in our network. Intuitively, this approach prevents the network from being too reliant on individual nodes, since they are likely to br dropped out at random steps of the optimization. This way, the neural network will build some redundancy in 'explaining' the

[1]Dropout method: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
[2]Patented by Google https://patents.google.com/patent/US9406017B2/en

training data, ideally avoiding overfitting. Indeed, this approach is related to standard regularization, as has been explored in several works[3]

[3]For an analysis of the connection between dropout training and explicit regularization, you can check out: https://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf