Optimization for ML (4)

CS771: Introduction to Machine Learning

Piyush Rai

Some Practical Aspects: Initialization

- Iterative opt. algos like GD, SGD, etc need to be initialized to "good" values
  - Bad initialization can result on bad local optima

But still be careful with learning rate

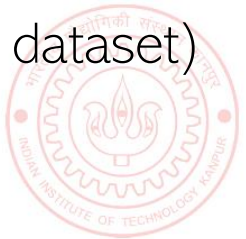
- Mainly a concern for non-convex loss functions, not so much for convex loss functions

If the goal is to learn the same model but for a different training set

- Transfer Learning: Initialize using params of a model trained on a related dataset

- Initialize using solution of a simpler but related model
  - E.g., for multitask regression (say T coupled regression problems), initialize using the solutions of the T independently trained regression problems

- For deep learning models, initialization is very important
  - Transfer learning approach is often used (initialize using "pre-trained" model from another dataset)
  - Bad initialization can make the model be stuck at saddle points. Need more care.
  - Random restarts: Running with several random initializations can often help



Some Practical Aspects: Assessing Convergence

- Various ways to assess convergence, e.g. consider converged if
  - The objective's value (on train set) ceases to change much across iterations

$$L(\boldsymbol{w}^{(t+1)}) - L(\boldsymbol{w}^{(t)}) < \epsilon$$ (for some small pre-defined ϵ)

  - The parameter values cease to change much across iterations

$$\| \boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)} \| < \tau$$ (for some small pre-defined τ)

  - Above condition is also equivalent to saying that the gradients are close to zero

$$\| \boldsymbol{g}^{(t)} \| \rightarrow 0$$

Caution: May not yet be at the optima. Use at your own risk!

  - The objective's value has become small enough that we are happy with ☺
  - Use a validation set to assess if the model's performance is acceptable (early stopping)

Some Practical Aspects: Learning Rate (Step Size)

- Some guidelines to select good learning rate (a.k.a. step size) η_t

C is a hyperparameter

- For convex functions, setting η_t something like C/t or $C/\sqrt{t}$ often works well
  - These step-sizes are actually theoretically optimal in some settings
  - In general, we want the learning rates to satisfy the following conditions
    - $\eta_t \to 0$ as t becomes very very large
    - $\sum \eta_t = \infty$ (needed to ensure that we can potentially reach anywhere in the parameter space)
  - Sometimes carefully chosen constant learning rates (usually small, or initially large and later small) also work well in practice
- Can also search for the "best" step-size by solving an opt. problem in each step

Also called "line search"

$$\eta_t = \arg\min_{\eta \geq 0} f\left(\mathbf{w}^{(t)} - \eta \cdot \mathbf{g}^{(t)}\right)$$

A one-dim optimization problem (note that $\mathbf{w}^{(t)}$ and $\mathbf{g}^{(t)}$ are fixed)

- An faster alternative to line search is the Armijo-Goldstein rule
  - Starting with current (or some large) learning rate (from prev. iter), and try a few values in decreasing order until the objective's value has a sufficient reduction

Some Practical Aspects: Adaptive Gradient Methods

- Can also use different learning rate in different dimensions

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{e}^{(t)} \odot \boldsymbol{g}^{(t)} \qquad e_d^{(t)} = \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^{t} \left(g_d^{(t)}\right)^2}}$$

Vector of learning rates along each dimension

Element-wise product of two vectors

If some dimension had big updates recently (marked by large gradient values), show down along those directions by using smaller learning rates - AdaGrad (Duchi et al, 2011)

- Can use a momentum term to stabilize gradients by reusing info from past grads
  - Move faster along directions that were previously good
  - Slow down along directions where gradient has changed abruptly

$$\boldsymbol{m}^{(t)} = \beta \boldsymbol{m}^{(t-1)} + \eta_t \boldsymbol{g}^{(t)}$$
$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \boldsymbol{m}^{(t)}$$

β usually set as 0.9

The "momentum" term. Set to 0 at initialization

In an even faster version of this, $\boldsymbol{g}^{(t)}$ is replaced by the gradient computed at the next step if previous direction were used, i.e., $\nabla L(\boldsymbol{w}^{(t)} - \beta \boldsymbol{m}^{(t-1)})$. Called Nesterov's Accelerated Gradient (NAG) method

- Also exists several more advanced methods that combine the above methods
  - RMS-Prop: AdaGrad + Momentum, Adam: NAG + RMS-Prop
  - These methods are part of packages such as PyTorch, Tensorflow, etc

Optimization for ML: Some Final Comments

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods also often use gradient methods
- Backpropagation algo used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization can use Lagrangian or projected/proximal GD
- Second order methods such as Newton's method faster but computationally expensive
- But computing all this gradient related stuff by hand looks scary to me. Any help?
  - Don't worry. Automatic Differentiation (AD) methods available now (will see them later)
  - AD only requires specifying the loss function (especially useful for deep neural nets)
  - Many packages such as Tensorflow, PyTorch, etc. provide AD support
  - But having a good understanding of optimization is still helpful