CS6301: Optimization in Machine Learning

Extra Lecture: Practical Aspects of Gradient Descent

Rishabh Iyer

Department of Computer Science
University of Texas, Dallas
https://sites.google.com/view/cs-6301-optml/home

March 9th, 2020

Gradient Descent in Practice: Fixed Learning Rate

```
def gd(funObj,w,maxEvals,alpha,X,y,lam, verbosity):
    [f,g] = funObj(w,X,y,lam)
    funEvals = 1
    funVals = []
    while(1):
        [f,g] = funObj(w,X,y,lam)
        optCond = LA.norm(g, np.inf)
        if (verbosity > 0):
            print(funEvals,alpha,f,optCond)
        w = w - alpha*g
        funEvals = funEvals+1
        if ((optCond < 1e-2) and (funEvals > maxEvals)
            break
        funVals.append(f)
    return funVals
```

Gradient Descent in Practice: Basic Version

- Run this by invoking:

```
funV = gd(LogisticLoss,w,200,1e-1,X,y,1,1,10)
```

- Try running this with different values of learning rates: $\alpha = 1e-1, 1e-3, 1e-5, ...$
- How do we find the optimal learning rate every time?
- Can there be better strategies to adapt the learning rates?
- Next, we shall see a few line search based strategies.

Armijo Backtracking Line-Search V1

- We don't want to tune α every time
- This is the idea behind line search
- Simple Line search strategy:
  - Start with a large value of α
  - Divide α by 1/2 if it doesn't satisfy Armijo's condition:

$$f(w - \alpha g) \leq f(w) - \gamma\alpha||g||^2$$

  - Basically find α such that there is a reduction in function value by atleast $\gamma\alpha||g||^2$
  - Idea: Choose α and γ such that this happens.

Armijo Backtracking Line-Search V1

Line Search:

- Start with a large value of α
- Divide α by 2 if it doesn't satisfy Armijo's condition: $f(w - \alpha g) \leq f(w) - \gamma\alpha||g||^2$.
- Basically find α such that there is a reduction in function value by atleast $\gamma\alpha||g||^2$
- Set gamma appropriately (for example, gamma = 1e-4 works generally). LS should not be much sensitive to gamma (try it empirically)

Armijo Backtracking Line-Search V1

Line Search:

- Start with a large value of α
- Divide α by 2 if it doesn't satisfy Armijo's condition: $f(w - \alpha g) \leq f(w) - \gamma\alpha ||g||^2$.
- Set gamma appropriately (for example, gamma = 1e-4 works generally).

```
wp = w - alpha*g
[fp,gp] = funObj(wp,X,y,lam)
while fp > f - gamma*alpha*np.dot(g.T, g):
    alpha = alpha/2
    wp = w - alpha*g
    [fp,gp] = funObj(wp,X,y,lam)
f = fp
g = gp
w = wp
```

Armijo Backtracking Line-Search V2

- Danger with the simple backtracking is that α may quickly become very small quickly
- Easy fix: Reset α every time!
- Issue with this: Too many function evaluations lost in repeated backtracking!
- Takeaway: V2 Armijo LS same as V1 except with an additional line in the beginning alpha $= 1$ before every line search.

Armijo Backtracking Line-Search V3

- Just halving the step size ignores the information collected during line search!
- Reduce the number of backtracks using a polynomial interpolation!
- Minimize a quadratic passing through $f(w), f'(w)$ and $f(w - \alpha g)$
- Choose α using a polynomial interpolation as follows:

$$\alpha = \frac{\alpha^2 g^T g}{2(fp + \alpha g^T g - f)}$$

- Here *fp* is the function evaluation with the current value of α and f is the function value before starting backtracking!
- Takeaway: V3 Armija same as V2 just changing the alpha reduction to the polynomial interpolation!

Armijo Backtracking Line-Search V4

- Final Issue to fix is better initialization of α.
- Initializing $\alpha = 1$ is too large in practice
- Wasted backtracks because of this.
- Use a hueristic like $\alpha = 1/||g||$
- On subsequent iterations again use a polynomial interpolation to reset alpha (after the line search)

$$\alpha = \min(1, 2(f_{old} - f)/g^T g)$$

- A lot of this is tried empirically and based on empirical knowledge..

Final Armijo v4 Line Search: Putting everything together

- Initialize *alpha* $= 1/||g||$ in the very beginning (instead of initializing with 1), i.e. in the zeroth iteration.
- Use the Polynomial interpolation instead of simply halving the alpha in the line search
- In subsequent iterations, reset alpha as

$$\alpha = \min(1, 2(f_{old} - f)/g^T g)$$

for the next iteration.

Accelerated Gradient Descent

Algorithm:

- Define $\lambda_0 = 0, \lambda_t = \frac{1+\sqrt{1+4\lambda_{t-1}^2}}{2}$ and $\gamma_t = \frac{\lambda_t - 1}{\lambda_{t+1}}$.
- Note $\gamma_t \leq 0$
- Initialize $x_1 = y_1$ as an arbitrary point
- Step 1: $y_{t+1} = x_t - \alpha \nabla f(x_t)$ (like normal GD)
- Step 2: $x_{t+1} = (1 + \gamma_t) y_{t+1} - \gamma_t y_t = y_{t+1} + \gamma_t (y_{t+1} - y_t)$ (slide a little bit further than y_{t+1} towards the previous point y_t!)
- Use the Armijo V4 Line Search to select alpha
- The final output is y_T.

Conjugate gradient Descent

- Conjugate Gradient Framework: (Initialize $d_0 = -g_0$)
  - Set $x_{k+1} = x_k + \alpha_k d_k$
  - Set $d_{k+1} = -g_{k+1} + \beta_k d_k$ where β_k is set based on the below schemes!
- Fletcher–Reeves CG algorithm: $\beta_k = \frac{||\nabla f(x_k)||^2}{||\nabla f(x_{k-1})||^2}$
- Polak–Ribière: $\beta_k = \frac{\nabla f(x_k)^T(\nabla f(x_k) - \nabla f(x_{k-1}))}{||\nabla f(x_{k-1})||^2}$
- Hestenes–Stiefel: $\beta_k = \frac{\nabla f(x_k)^T(\nabla f(x_k) - \nabla f(x_{k-1})}{d_k^T(\nabla f(x_k) - \nabla f(x_{k-1})}$
- You can use a line search algorithm similar to Gradient Descent!

Barzelia Bowrein Step Length

- Approach 1: $\alpha_k = \frac{s_{k-1}^T s_{k-1}}{s_{k-1}^T y_{k-1}}$
- Approach 2:$\alpha_k == \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$
- Again, use a Line Search on top of this to get the value of α
- The rest of the algorithm is exactly similar to Gradient Descent!
- Here $s_{k-1} = x_k - x_{k-1}$ and $y_{k-1} = \nabla f(x_k) - \nabla f(x_{k-1})$.