Deep Learning Study Lecture Note

Lecture 3 : Various Gradient Descent

Kim Tae Young, Lee Janggun

January 3, 2020

3.1 Stochastic Gradient Descent

As the same with gradient descent, we consider the following problem of minimizing a differenciable function $f(x)$, but this time we consider 'splitting' f into n sub-functions as $f(x) = \frac{1}{n}\sum_i^n f_i(x)$, where each f_i are also differential. For update, instead of taking the gradient of the entire function, we select an index i and only take the gradient of f_i. The stochastic gradient descent algorithm is as follows.

$$x_t = x_{t-1} - \mu_k \nabla_{x_{t-1}} f_i(x_{t-1})$$

Normally, the selection of index i is made uniformly from $[1, n]$. As with the standard assumption that ∇f is Lipshcitz, there are a few standard assumptions for stochastic process.

$$\mathbb{E}[||\nabla f_{i_k}(x)||_2^2] \leq \sigma^2 \tag{3.1}$$

$$\mathbb{E}[\nabla f_{i_k}(x) - \nabla f(x)] = 0 \tag{3.2}$$

To show convergence of x_k, it suffices to show convergence of $\nabla E[||f(x_k)||_2^2]$ to 0. Under assumption of convex f and Lipshcitz gradient, we have that:

$$f(y) \leq f(x) + \nabla f(x)^T(y - x) + \frac{L}{2}||y - x||_2^2$$

Inserting the update rule, we have

$$f(x_{k+1}) \leq f(x_k) + \nabla f(x_k)^T(-\mu_k f_{i_k}(x_k)) + \frac{L}{2}||\mu_k f_{i_k}(x_k)||_2^2$$

and taking expectation yields

$$f(x_{k+1}) \leq f(x_k) - \mu_k \mathbb{E}[||\nabla f_{i_k}(x)||_2^2] + \frac{L\mu_k^2}{2}\mathbb{E}[||\nabla f_{i_k}(x)||_2^2]$$

moving terms and telescoping, we have that

$$\mathbb{E}[||\nabla f_{i_k}(x_k)||_2^2] \leq \frac{f(x_0) - f(x^*)}{\sum_{i=1}^k \mu_k} + \frac{L\sigma^2}{2}\frac{\sum_{i=1}^k \mu_k^2}{\sum_{i=1}^k \mu_k}$$

where x^* is the optimal point of f. Thus for suitable choice of μ_k, the RHS goes to 0 and we have the desired conclusion. In practice, it is hard to determine if the gradient is Lipshitz and has bounded expectation, so a small step size is used.

Though gradient descent is a good algorithm to find the optimal value, there are many problems involved in actual experiments. The most evident problem is the issue of local minima and saddle points. As the gradient being zero only tells about f's critical points, we cannot know if we have reached the desired global minimum. For convex functions this is not an issue, but as many loss functions are non-convex, this is an issue.

The second problem is that of step size. As gradient only tells that of steepest decrease, the step size is selected to be a small value in order to ensure the the minimum is not passed. However, too small of a step size implies more iterations towards the minimum, which causes slow convergence. Thus the step size has a trade - off of convergence speed and safety.

The final problem is oscillation. This is not seen in 1 - dimensional functions, but found in functions such as the Rosenbrock function. While descending into the minimum, the path may resemble a zigzag pattern rather than a straight one, which leads to unnecessary iterations of the algorithm.

3.2 Momentum Method

The motivation for momentum is the momentum you knows. Consider optimizing following function, where we starting gradient on init point.

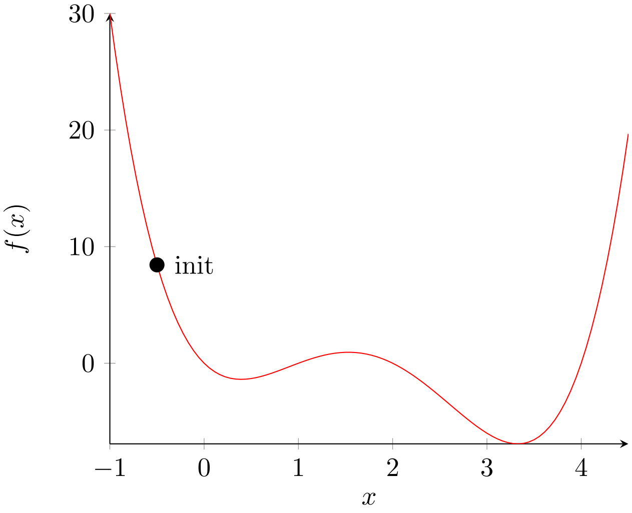

Figure 3.1: Momentum required optimization

Now think it as rolling a stone from initialization point with friction existing. Then as we expect, the stone will safely enters to global minima. The difference between vanilla gradient descent and this example is that stone remembers its step, by momentum. (yes in fact it is not momentum but energy conservation law.) Here motivation of momentum method comes, adding momentum to our gradient descent.

The momentum algorithm is defined as following.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \mu \nabla_x \mathcal{L}(x_{t-1}) && v_0 = 0 \\ x_t &= x_{t-1} - v_t \end{aligned}$$

Here, γ is decaying rate of momentum, which you can think as friction.

The main effects of momentum method is escaping local minimum, which is illustrated in example. Also, it increases rate of convergence, since stepping to same direction increases step size.

There is similar momentum like algorithm called NAG(Nesterov Accelerated Gradient), defined as following.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \mu \nabla_x \mathcal{L}(x_{t-1} - \gamma v_{t_1}) && v_0 = 0 \\ x_t &= x_{t-1} - v_t \end{aligned}$$

The difference between original momentum method and NAG is that we calculate gradient after applying momentum.

3.3 Adaptive Gradient Descent

The basic idea of adaptive gradient descent is the assumption that parameters that has moved a lot will have small step size, and that parameters that moved little, will have big step size to reach global minimum. So as the name states, we adaptively change learning rate for each parameter.

3.3.1 AdaGrad

AdaGrad is most basic adaptive gradient descent, defined as following.

$$\begin{aligned} G_t &= G_{t-1} + (\nabla_x \mathcal{L}(x_{t-1}))^2 && G_0 = 0 \\ x_t &= x_{t-1} - \mu \frac{1}{\sqrt{G_t + \epsilon}} \nabla_x \mathcal{L}(x_{t-1}) \end{aligned}$$

So G_t stores sum of gradient until time t. ϵ is added to prevent explosion of gradient.

AdaGrad has one critical issue, that after enough steps the step size converges to 0, since value of G_t will increases every step.

3.3.2 RMSProp

RMSProp resolves that issue, by replacing sum of gradient to weighted sum.

$$\begin{aligned} G_t &= \gamma G_{t-1} + (1-\gamma)(\nabla_x \mathcal{L}(x_{t-1}))^2 && G_0 = 0 \\ x &= x - \mu \frac{1}{\sqrt{G_t + \epsilon}} \nabla_x \mathcal{L}(x_{t-1}) \end{aligned}$$

3.3.3 AdaDelta

AdaDelta also resolves issue of AdaGrad, but also one another problem. Consider unit in gradient descent and newton's method, with loss's unit is 1 and x's unit is u.

$$\begin{aligned}\Delta_x^{(1)} &\propto \frac{\partial \mathcal{L}}{\partial x} \propto \frac{1}{u}\\ \Delta_x^{(2)} &\propto \frac{\partial \mathcal{L}/\partial x}{\partial^2 \mathcal{L}/\partial x^2} \propto u\end{aligned}$$

So as you can see, the unit does not match in gradient descent. AdaDelta solves this problem by approximating newton's method.

$$\begin{aligned}G_t &= \gamma G_{t-1} + (1-\gamma)(\nabla_x \mathcal{L}(x_{t-1}))^2 && G_0 = 0\\ v_t &= \frac{\sqrt{s_{t-1}+\epsilon}}{\sqrt{G+\epsilon}} \nabla_x \mathcal{L}(x_{t-1})\\ x_t &= x_{t-1} - v_t\\ s_t &= \gamma s_{t-1} + (1-\gamma) v_t^2 && s_0 = 0\end{aligned}$$

3.4 Adam

Adam(Adaptive momentum approximation) is algorithm that is combining RMSProp and momentum method. First, it stores weighted sum of both gradient and squared gradient.

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_x \mathcal{L}(x_{t-1}) && m_0 = 0\\ v_t &= \beta_2 v_{t-2} + (1-\beta_2)(\nabla_x \mathcal{L}(x_{t-1}))^2 && v_0 = 0\end{aligned}$$

Then since m_t, v_t are initialized to 0, during initial stages of training, it will be closed to 0. So we need to normalize it. By expanding m_t and v_t

$$\begin{aligned}m_t &= \sum_{i=0}^{t-1} \beta_1^i (1-\beta_1)\Delta_1 = (1-\beta_1^t)\Delta_1\\ v_t &= \sum_{i=0}^{t-1} \beta_2^i (1-\beta_2)\Delta_2 = (1-\beta_2^t)\Delta_2\end{aligned}$$

where Δ_1, Δ_2 is expectation value for gradient and squared gradient. With normalized values, we have

$$\begin{aligned}\hat{m}_t &= \frac{m_t}{1-\beta_1^t}\\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}\\ x_t &= x_{t-1} - \mu \frac{1}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t\end{aligned}$$

3.5 L-BFGS

L-BFGS(Limited memory Broyden–Fletcher–Goldfarb–Shanno) algorithm is quasi-newton algorithm that performs approximation of newton's method. The main problem of newton's method is that it requires computation of hessian matrix, which is inefficient in both terms of time and space. One main failure is that chain rule is not that fast in multi variable, degree 2 case. The rule is given as following.

$$\frac{\partial^2 y}{\partial x_i \partial x_j} = \sum_k \frac{\partial y}{\partial u_k} \frac{\partial^2 u_k}{\partial x_i \partial x_j} + \sum_{k,l} \frac{\partial^2 y}{\partial u_k \partial u_l} \frac{\partial u_k}{\partial x_i} \frac{\partial u_l}{x_j}$$

You can see we have sum for N^2 items, yielding $O(n^2)$ time complexity for each chain rule application.

Also, the other problem is storing hessian matrix and inverting it. Simply, it first requires $O(n^2)$ memory for storing hessian, and inverting it also requires $O(n^3)$ time complexity.

So we approximate hessian matrix in L-BFGS algorithm. First, we will see BFGS algorithm, which is basis of L-BFGS algorithm. Here, H is approximation of hessian matrix in BFGS algorithm.

$$\begin{aligned}
x_{t+1} &= x_t - \mu H_t \nabla \mathcal{L} \\
H_{t+1} &= (I - \frac{sy^T}{y^T s} H_t (I - \frac{ys^T}{y^T s}) + \frac{ss^T}{y^s}) \\
s_t &= x_{t+1} - x_t \\
y_t &= \nabla_x \mathcal{L}(x_{t+1}) - \nabla_x \mathcal{L}(x_t)
\end{aligned}$$

Then we have for every $q \in \mathbb{R}^n$

$$H_{t+1} q = H_t q + (\frac{s^T q}{y^T s} - \frac{y^T p}{y^T s}) s$$

Finally, L-BFGS is computed as following.

```
Result: $p_t = -H_t \nabla \mathcal{L}(x_t)$
$q := -\nabla \mathcal{L}(x_t)$;
for $i \leftarrow k - 1$ to $\min(k - m, 0)$ do
    $\alpha_i := \frac{s_i^T q}{y_i^T s_i}$;
    $q := q - \alpha y_i$
end
$p := H_{0,k} q$;
for $i \leftarrow \min(k - m, 0)$ to $k - 1$ do
    $\beta := \frac{y_i^T p}{y_i^T s_i}$;
    $p := p + (\alpha_i - \beta) s_i$
end
return $p$
```

3.6 Genetic Algorithm

Here, I will explain non gradient descent algorithm. Genetic Algorithm is simulation of natural evolution.

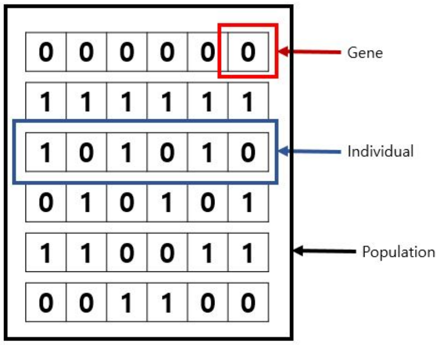

Figure 3.2: Initial Population

Each individual is characterized by set of parameters, called as genes. Then genes are joined into a string to form a chromosome. Then we start with initial population, where population is set of individuals.

Then the fitness function determines how fit an individual is. It gives score based on the function, and the probability that an individual selected for reproduction is determined by this score. You can think this function as loss function, however unlike loss function it does not need to be differentiable.

In selection phase, we select pairs of individuals based on their fitness score. Here, individual with higher fitness score is more likely to be selected. Suppose we selected i_1, i_2.

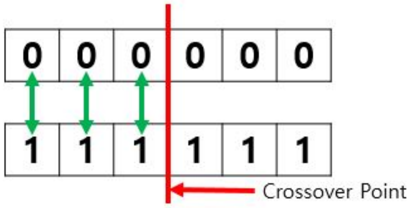

Figure 3.3: Crossover

In crossover phase we randomly choose crossover point, within the genes. After choosing crossover point, we create two new individuals that can be obtained by exchanging genes before crossover point from i_1, i_2. The crossover phase is repeated until we fills rest of population with crossovered offsprings.

Finally, mutation occurs. With a low random probability, some of the genes are flipped.

The algorithm terminates if the population has converged, i.e. does not produce offspring that is significantly different.