Akshay L Chandra Follow

May 15, 2019 · 6 min read · Listen

LEARNING PARAMETERS

Learning Parameters, Part 2: Momentum-Based & Nesterov Accelerated Gradient Descent

Let's look at two simple, yet very useful variants of gradient descent.

In this post, we look at how the gentle-surface limitation of Gradient Descent can be overcome using the concept of momentum to some extent. Make sure you check out my blog post — Learning Parameters, Part-1: Gradient Descent, if you are unclear of what this is about. Throughout the blog post, we work with the same toy problem introduced in part-1. You can check out all the posts in the ***Learning Parameters*** series by clicking on the kicker tag at the top of this post.

In part-1, we saw a clear illustration of a curve where the gradient can be small in gentle regions of the error surface, and this could slow things down. Let's look at what momentum has to offer overcome this drawback.

> *Citation Note: Most of the content and figures in this blog are directly taken from Lecture 5 of CS7015: Deep Learning course offered by Prof. Mitesh Khapra at IIT-Madras.*

Momentum-Based Gradient Descent

The intuition behind MBGD from the mountaineer's perspective (yes the same trite metaphor we used in part-1) is

> *If I am repeatedly being asked to move in the same direction then I should probably gain*

The Momentum Update Rule

We accommodate the momentum concept in the gradient update rule as follows:

Without momentum:

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$

With momentum:

$$update_t^w = \gamma \cdot update_{t-1}^w + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t^w$$
$$update_t^b = \gamma \cdot update_{t-1}^b + \eta \nabla b_t$$
$$b_{t+1} = b_t - update_t^b$$

In addition to the current update, we also look at the history of updates. I encourage you to take your time to process the new update rule and try and put it on paper how the *update* term changes in every step. Or keep reading. Breaking it down we get

$$update_0 = 0$$
$$update_1 = \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1$$
$$update_2 = \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2$$
$$update_3 = \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3$$
$$= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3$$
$$update_4 = \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4$$
$$\vdots$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t$$

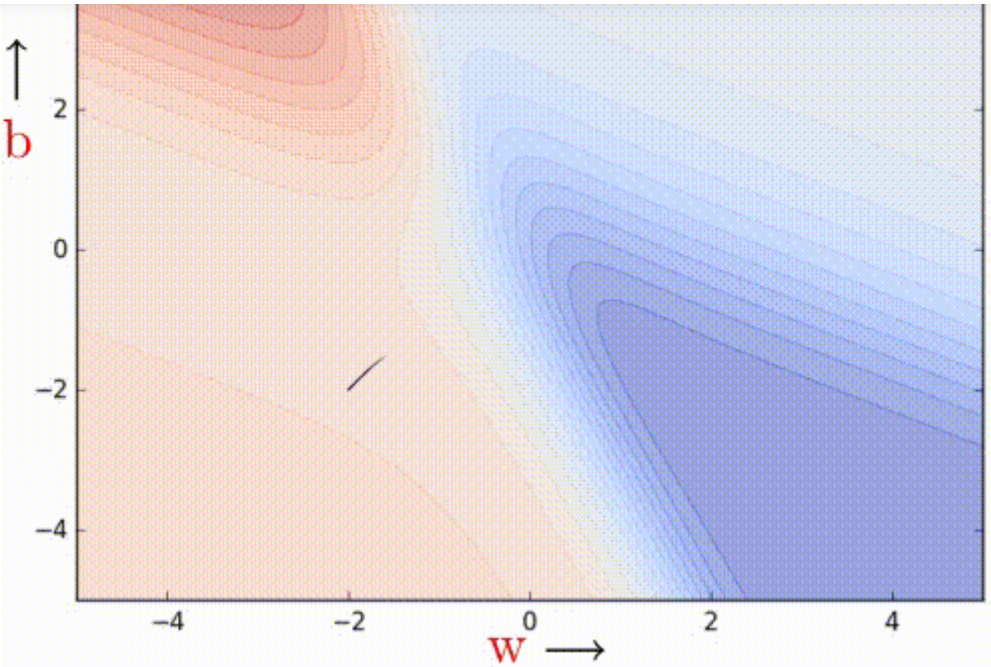
You can see that the current update is proportional to not just the present gradient but also gradients of previous steps, although their contribution reduces every time step by γ(gamma) times. And that is how we boost the magnitude of the update at gentle regions.

```python
def do_momentum_gradient_descent():
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs):
        dw, db = 0, 0
        for x,y in zip(X,Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        v_w = gamma * prev_v_w + eta*dw
        v_b = gamma * prev_v_b + eta*db
        w = w - v_w
        b = b - v-b
        prev_v_w = v_w
        prev_v_b = v_b
```

momentum-gd.py hosted with ❤️ by GitHub view raw

From now on, we will only work with contour maps. Visualizing things in 3-D can be cumbersome, so contour maps come in as a handy alternative for representing functions with 2-D input and 1-D output. If you are unaware of/uncomfortable with them, please go through **section 5** of my basic stuff blog post — *Learning Parameters, Part-0: Basic Stuff* (there is even a self-test you can take to get better at interpreting them). Let's see how effective MBGD is using the same toy neural network we introduced in part-1.

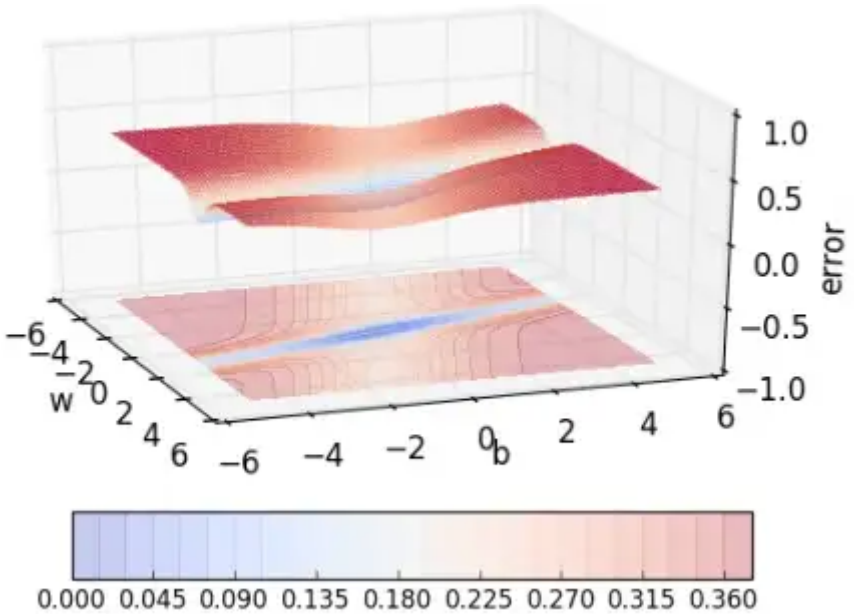


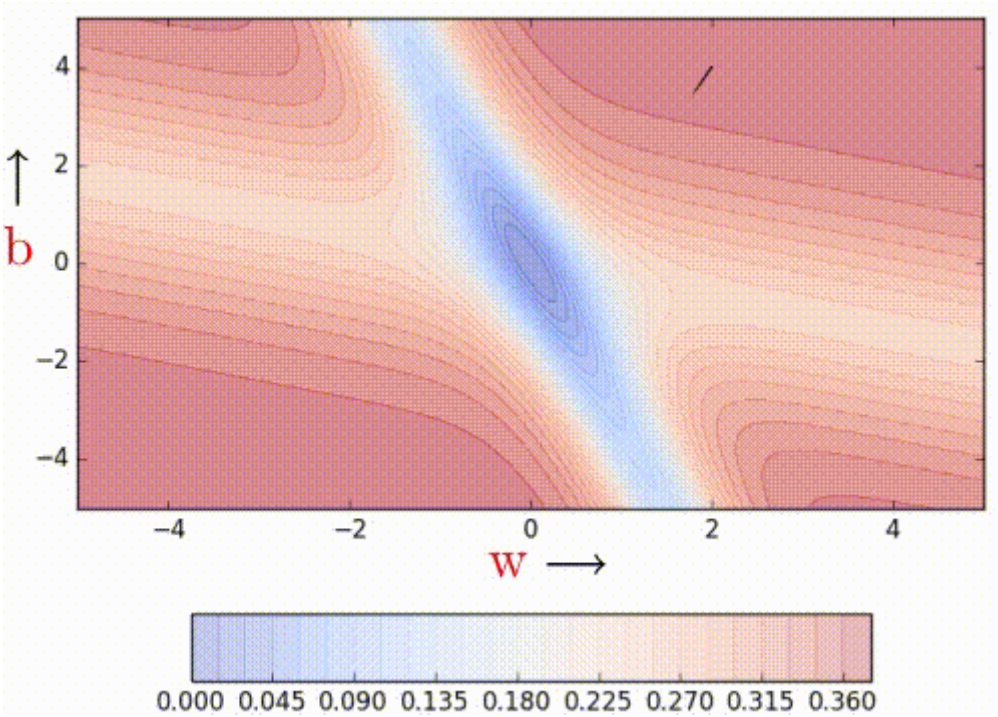
Momentum-Based Gradient Descent. 100 iterations of vanilla gradient descent make the black patch.

It works. 100 iterations of vanilla gradient descent make the black patch, and it is evident that even in the regions having gentle slopes, momentum-based gradient descent can take substantial steps because the momentum carries it along.

On a critical note, is moving fast always good? Would there be a situation where momentum would cause us to overshoot and run past our goal? Let test the MBGD by changing our input data so that we end up with a different error surface.



instead? Let's find out.



100 iterations of vanilla gradient descent make the black patch.

We can observe that momentum-based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley. This makes us take a lot of U-turns before finally converging. Despite these U-turns, it still converges faster than vanilla gradient descent. After 100 iterations momentum-based method has reached an error of 0.00001 whereas vanilla gradient descent is still stuck at an error of 0.36.

Can we do something to reduce the oscillations/U-turns? Yes, Nesterov Accelerated Gradient Descent helps us do just that.

Nesterov Accelerated Gradient Descent

The intuition behind NAG can be put into a single phrase:

> *Look ahead before you leap!*

more by $\eta \nabla w_t$.

Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w ($w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$) instead of calculating it using the current value w_t.

The NAG Update Rule

With momentum:

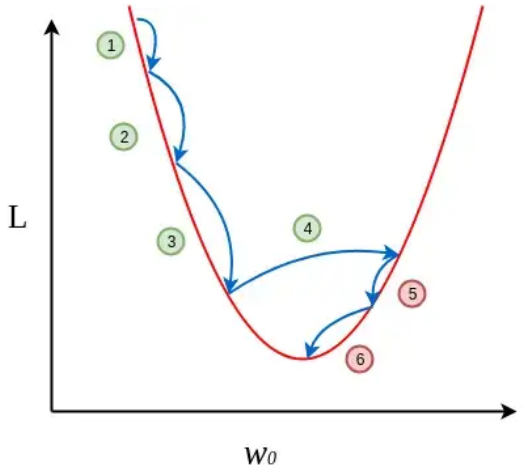
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

With NAG:

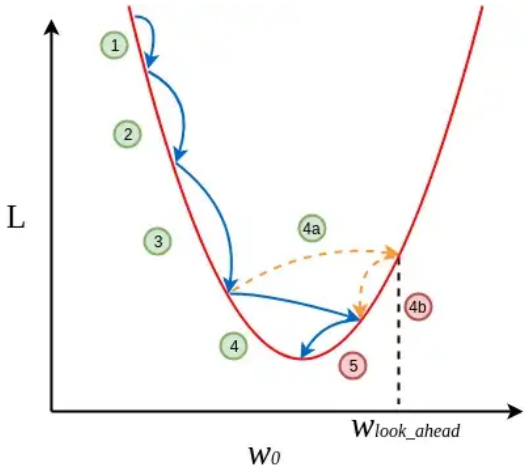
$$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead}$$
$$w_{t+1} = w_t - update_t$$

We follow similar update rule for b_t.

But why would looking ahead help us in avoiding overshoots? I urge you to pause and ponder. If it's not clear, I am sure it will be clear in the next few minutes. Take a look at this figure for a moment.



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

In figure (a), update 1 is positive i.e., the gradient is negative because as ***w_0*** increases ***L*** decreases. Even update 2 is positive as well and you can see that the update is slightly larger than update 1, thanks to momentum. By now, you should be convinced that update 3 will be bigger than both update 1 and 2 simply because of momentum and the positive update history. Update 4 is where things get interesting. In vanilla momentum case, due to the positive history, the update overshoots and the descent recovers by doing negative updates.

But in NAG's case, every update happens in two steps — first, a partial update, where we get to the *look_ahead* point and then the final update (see the NAG update rule), see figure (b). First 3 updates of NAG are pretty similar to the momentum-based method as both the updates (partial and final) are positive in those cases. But the real difference becomes apparent during update 4. As usual, each update happens in two stages, the partial update (4a) is positive, but the final update (4b) would be negative as the calculated gradient at ***w_lookahead*** would be negative (convince yourself by observing the graph). This negative final update slightly reduces the overall magnitude of the update, still resulting in an overshoot but a smaller one when compared to the vanilla momentum-based gradient descent. And that my friend, is how NAG helps us in reducing the overshoots, i.e. making us take shorter U-turns.

NAG In Action

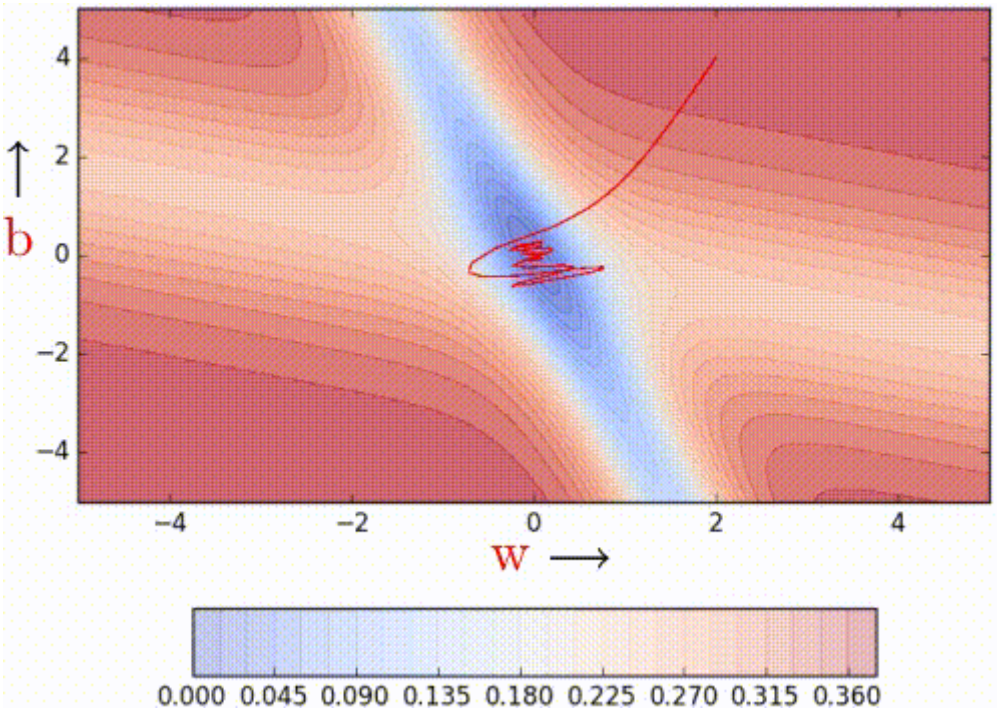
By updating the momentum code slightly to do both the partial update and the full update, we get the code for NAG.

```
def do_nesterov_accelerated_gradient_descent():
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs):
        dw, db = 0, 0
        # do partial update
        v_w = gamma * prev_v_w
        v_b = gamma * prev_v_b
        for x,y in zip(X,Y):
          # calculate gradients after partial update
            dw += grad_w(w - v_w, b - v_b, x, y)
```

```
17          b = b - v_b
18          prev_v_w = v_w
19          prev_v_b = v_b
```

nesterov-accelerated.py hosted with ❤️ by GitHub view raw

Let's compare the convergence of momentum-based method with NAG using the same toy example and the same error surface we used while illustrating the momentum-based method.



You can see that (I hope you can) NAG (blue) is taking smaller U-turns compared to vanilla momentum (red).

NAG is certainly making smaller oscillations/taking shorter U-turns even when approaching the minima valleys on the error surface. Looking ahead helps NAG in correcting its course quicker than momentum-based gradient descent. Hence the oscillations are smaller and the chances of escaping the minima valley also smaller. Earlier, we proved that looking back helps **ahem momentum ahem** and we now proved that looking ahead also helps.

Conclusion

In this blog post, we looked at two simple, yet hybrid versions of gradient descent that help us converge faster — *Momentum-Based Gradient Descent* and *Nesterov Accelerated Gradient Descent (NAG)* and also discussed why and where NAG beats

about stochastic versions of these algorithms.

Read all about it in the next post of this series at:

- Learning Parameters, Part 3: Stochastic & Mini-Batch Gradient Descent

Acknowledgment

A lot of credit goes to **Prof. Mitesh M Khapra** and the TAs of **CS7015: Deep Learning** course by IIT Madras for such rich content and creative visualizations. I merely just compiled the provided lecture notes and lecture videos concisely.