Learning Parameters, Part 5: AdaGrad, RMSProp, and Adam

towardsdatascience.com/learning-parameters-part-5-65a2f3583f7d

December 9, 2019

Learning Parameters

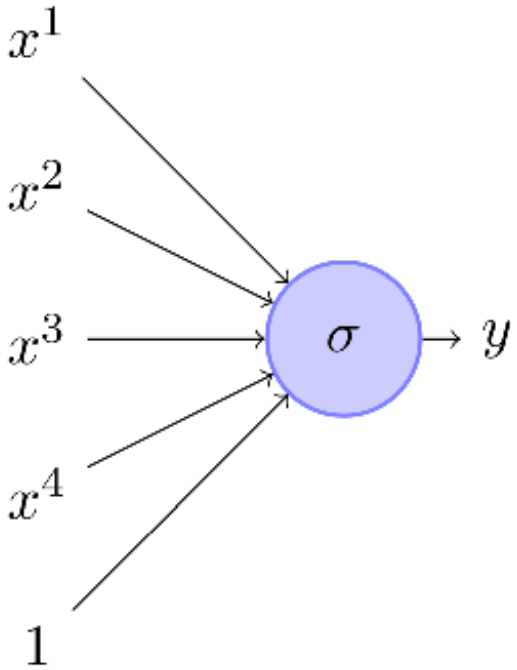
Let's look at gradient descent with an adaptive learning rate.

In part 4, we looked at some heuristics that can help us tune the learning rate and momentum better. In this final article of the series, let us look at a more *principled* way of adjusting the learning rate and give the learning rate a chance to adapt.

> 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.

Motivation for Adaptive Learning Rate

Consider the following simple perceptron network with sigmoid activation.



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

It should be easy to see that given a single point (**x**, *y*), gradients of **w** would be the following:

$$\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$$
$$\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2 \ \ldots \text{ so on}$$

See if this is unclear.

Gradient of w.r.t to a particular weight is clearly dependent on its corresponding input. If there are *n* points, we can just sum the gradients over all the *n* points to get the total gradient. This news is neither new nor special. But what would happen if the feature is very sparse (i.e., if its value is 0 for most inputs)? It is fair to assume that ∇ will be 0 for most inputs (see formula) and hence will not get enough updates.

Why should that bother us though? It is important to note that if at all is both sparse as well as important, we would want to take the updates to seriously. To make sure updates happen even when a particular input is sparse, can we have a different learning rate for each parameter which takes care of the frequency of the features? We sure can. I mean that is the whole point of this article.

AdaGrad — Adaptive Gradient Algorithm

Intuition

Decay the learning rate for parameters in proportion to their update history (more updates means more decay).

Update Rule for AdaGrad

It is clear from the update rule that history of the gradient is accumulated in . The smaller the gradient accumulated, the smaller the value will be, leading to a bigger learning rate (because divides *η*).

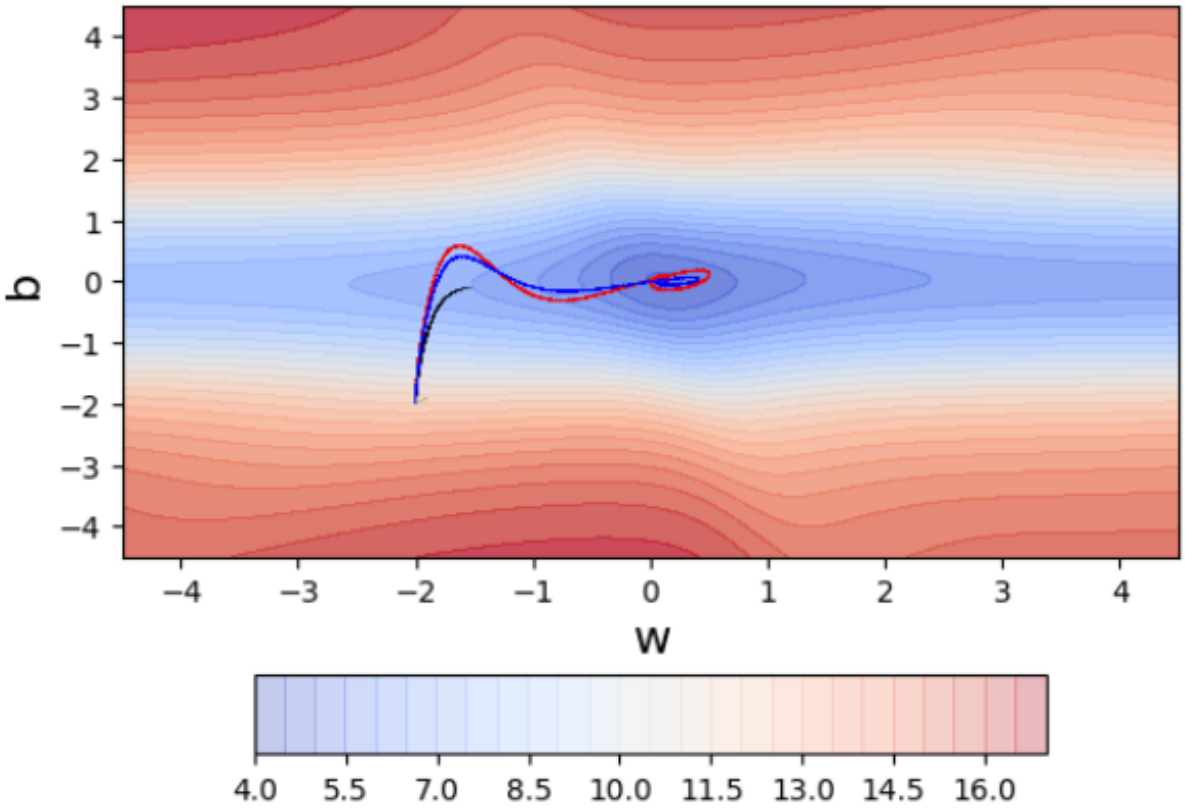
$$v_t^w = v_{t-1}^w + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = v_{t-1}^b + (\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

Python Code for AdaGrad

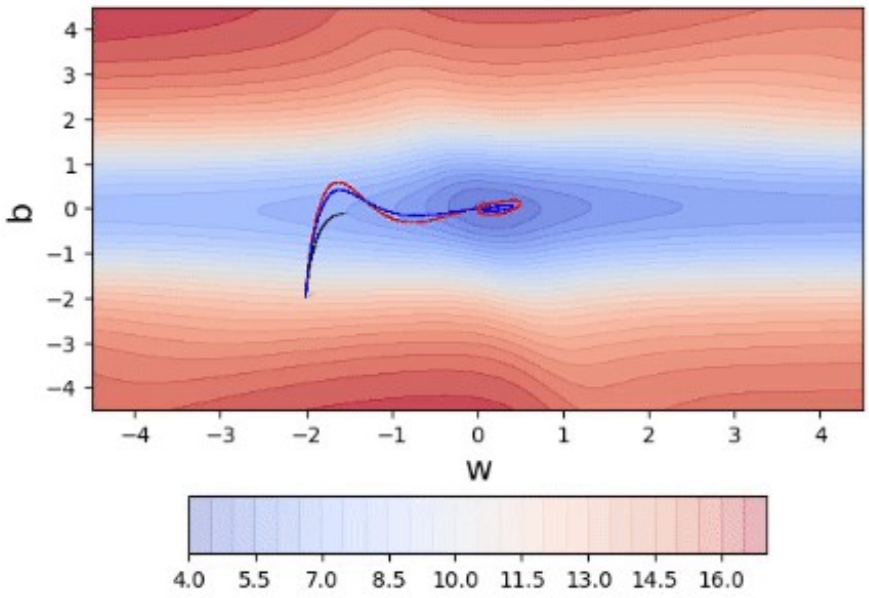
AdaGrad in Action

To see AdaGrad in action, we need to first create some data where one of the features is sparse. How would we do this to the toy network we used across all parts of the Learning Parameters series? Well, our network has just two parameters (and , see *Motivation* in part-1). Of these, the input/feature corresponding to b is always on, so we can't really make it sparse. So the only option is to make x sparse. Which is why we created 100 random *(x,y)* pairs and then roughly 80% of these pairs we set *x* to 0, making the feature for sparse.



Vanilla GD(black), momentum (red), NAG (blue)

Before we actually look at AdaGrad in action, please look at the other 3 optimizers above - vanilla GD(black), momentum (red), NAG (blue). There is something interesting that these 3 optimizers are doing for this dataset. Can you spot it? Feel free to pause and ponder.
Answer: Initially, all three algorithms are moving mainly along the vertical () axis and there is very little movement along the horizontal () axis. Why? Because in our data, the feature corresponding to is sparse and hence undergoes very few updates. On the other hand, is very dense and undergoes many updates. Such sparsity is very common in large neural networks containing 1000s of input features and hence we need to address it. Let us now look at AdaGrad in action.



Voila! By using a parameter specific learning rate AdaGrad ensures that despite sparsity gets a higher learning rate and hence larger updates. Furthermore, it also ensures that if undergoes a lot of updates, its effective learning rate decreases because of the growing denominator. In practice, this does not work so well if we remove the square root from the denominator (something to ponder about). What's the flipside? Over time the effective learning rate for will decay to an extent that there will be no further updates to . Can we avoid this? RMSProp can!

RMSProp — **Root Mean Square Propagation**

Intuition

AdaGrad decays the learning rate very aggressively (as the denominator grows). As a result, after a while, the frequent parameters will start receiving very small updates because of the decayed learning rate. To avoid this why not decay the denominator and prevent its rapid growth.

Update Rule for RMSProp

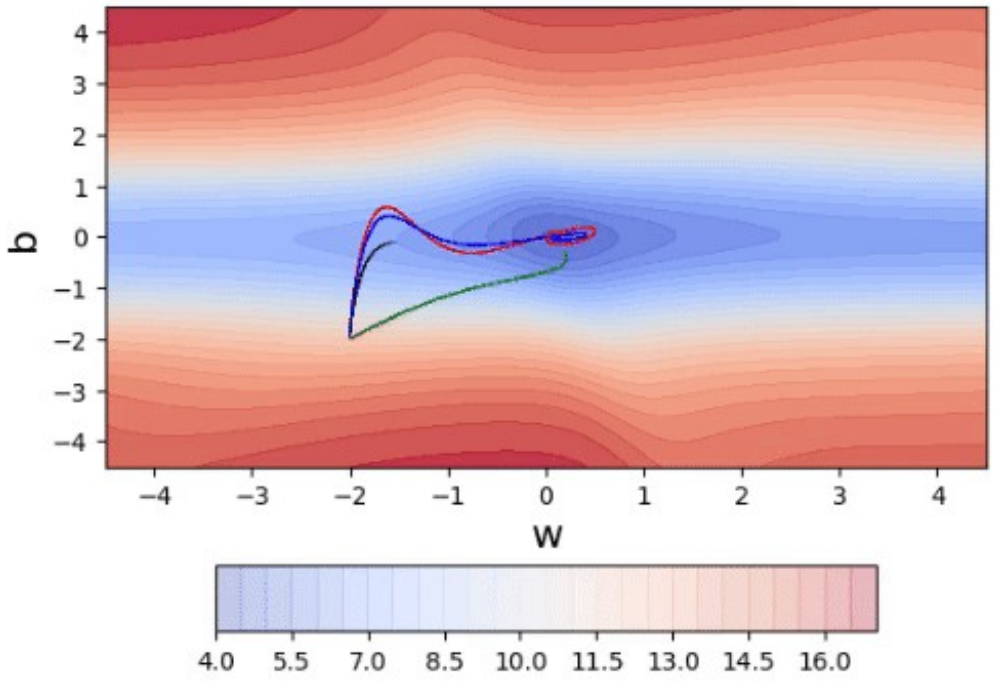
$$v_t^w = \beta * v_{t-1}^w + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = \beta * v_{t-1}^b + (1-\beta)(\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

Everything is very similar to AdaGrad, except now we decay the denominator as well.

Python Code for RMSProp

RMSProp in Action



Vanilla GD(black), momentum (red), NAG (blue), AdaGrad (magenta)

What do you see? How is RMSProp different from AdaGrad? Feel free to pause and ponder.
Answer: AdaGrad got stuck when it was close to convergence, it was no longer able to move in the vertical () direction

because of the decayed learning rate. RMSProp overcomes this problem by being less aggressive on the decay.

Adam — **Adaptive Moment Estimation**

Intuition

Do everything that RMSProp does to solve the denominator decay problem of AdaGrad. In addition to that, use a cumulative history of gradients.

Update Rule for Adam

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$

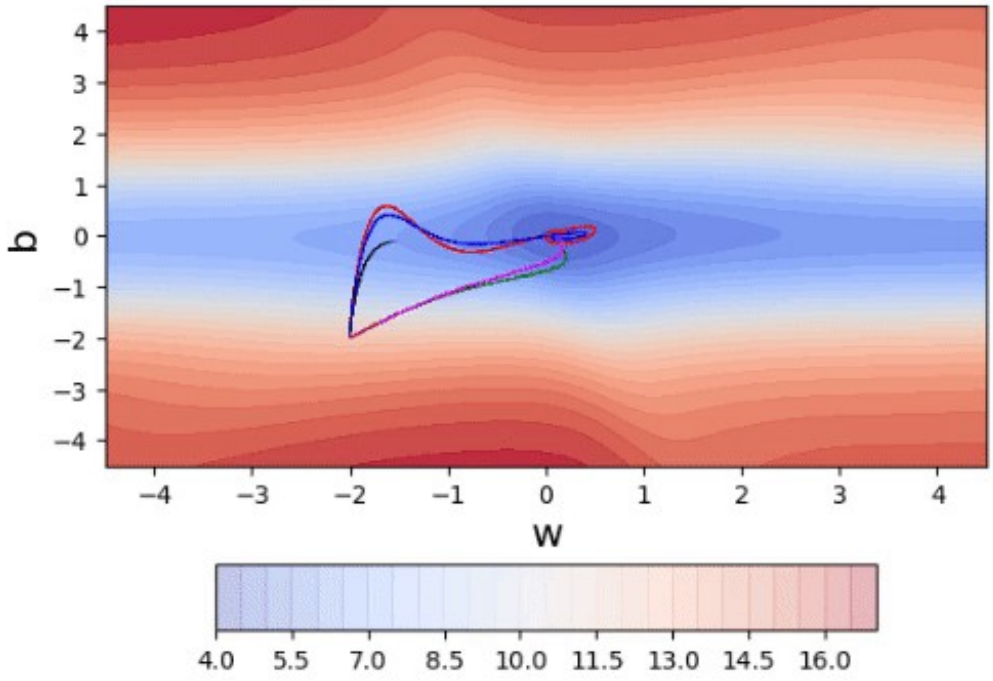
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

and a similar set of equations for _*t*. Notice that the update rule for Adam is very similar to RMSProp, except we look at the cumulative history of gradients as well (_*t*). Note that the third step in the update rule above is bias correction. Explanation by Prof. Mitesh M Khapra on why bias correction is necessary can be found here.

Python Code for Adam

Adam in Action



Quite clearly, taking a cumulative history of gradients speeds it up. For this toy dataset, it appears to be overshooting (a little) but even then it converges way faster than the other optimizers.

Million Dollar Question: Which algorithm to use?

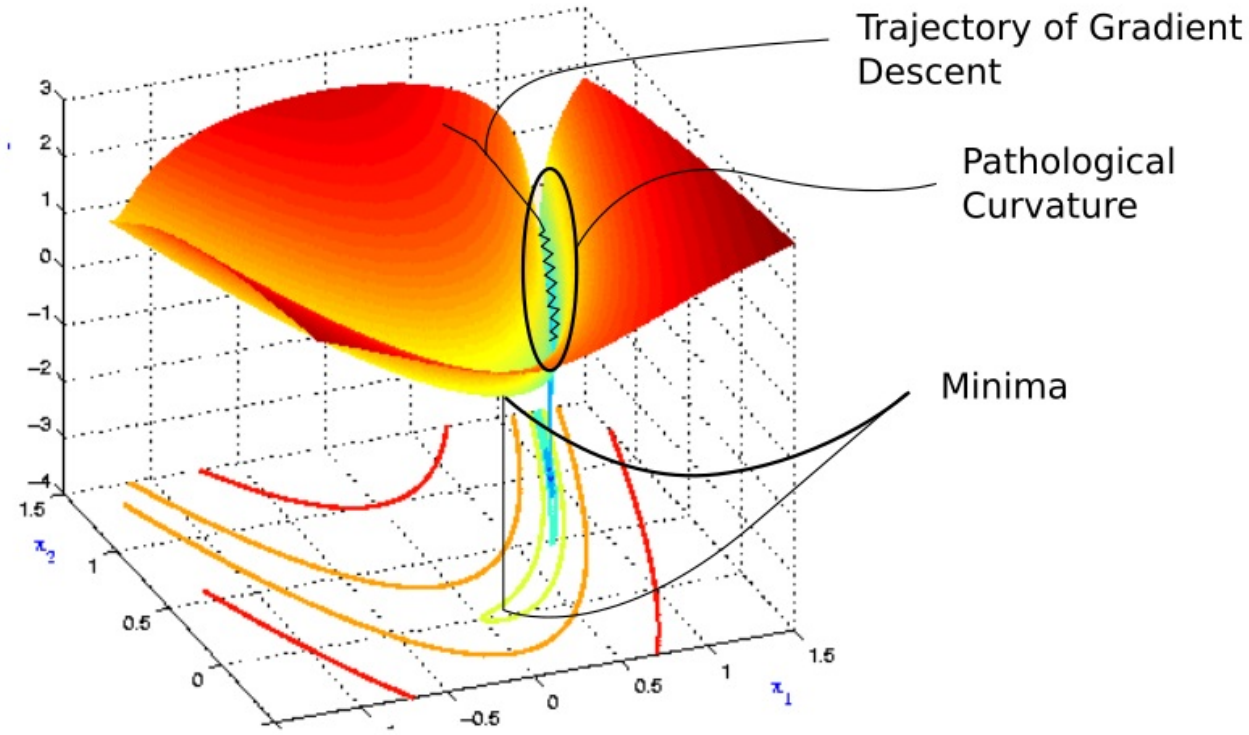
- Adam seems to be more or less the default choice now (= 0.9, = 0.999 and = 1e − 8).
- Although it is supposed to be robust to initial learning rates, we have observed that for sequence generation problems = 0.001, 0.0001 works best.
- Having said that, many papers report that SGD with momentum (Nesterov or classical) with a simple annealing learning rate schedule also works well in practice (typically, starting with = 0.001, 0.0001 for sequence generation problems).
- Adam might just be the best choice overall.
- Some recent work suggests that there is a problem with Adam and it will not converge in some cases.

Conclusion

In this final article of the series, we looked at how gradient descent with adaptive learning rate can help speed up convergence in neural networks. Intuition, python code and visual illustration of three widely used optimizers — AdaGrad, RMSProp, and Adam are covered in this article. Adam combines the best properties of RMSProp and AdaGrad to work well even with noisy or sparse datasets.

Acknowledgment

A lot of credit goes to and the TAs of course by IIT Madras for such rich content and creative visualizations. I merely just compiled the provided lecture notes and lecture videos concisely.



Paperspace article on RMSProp by .