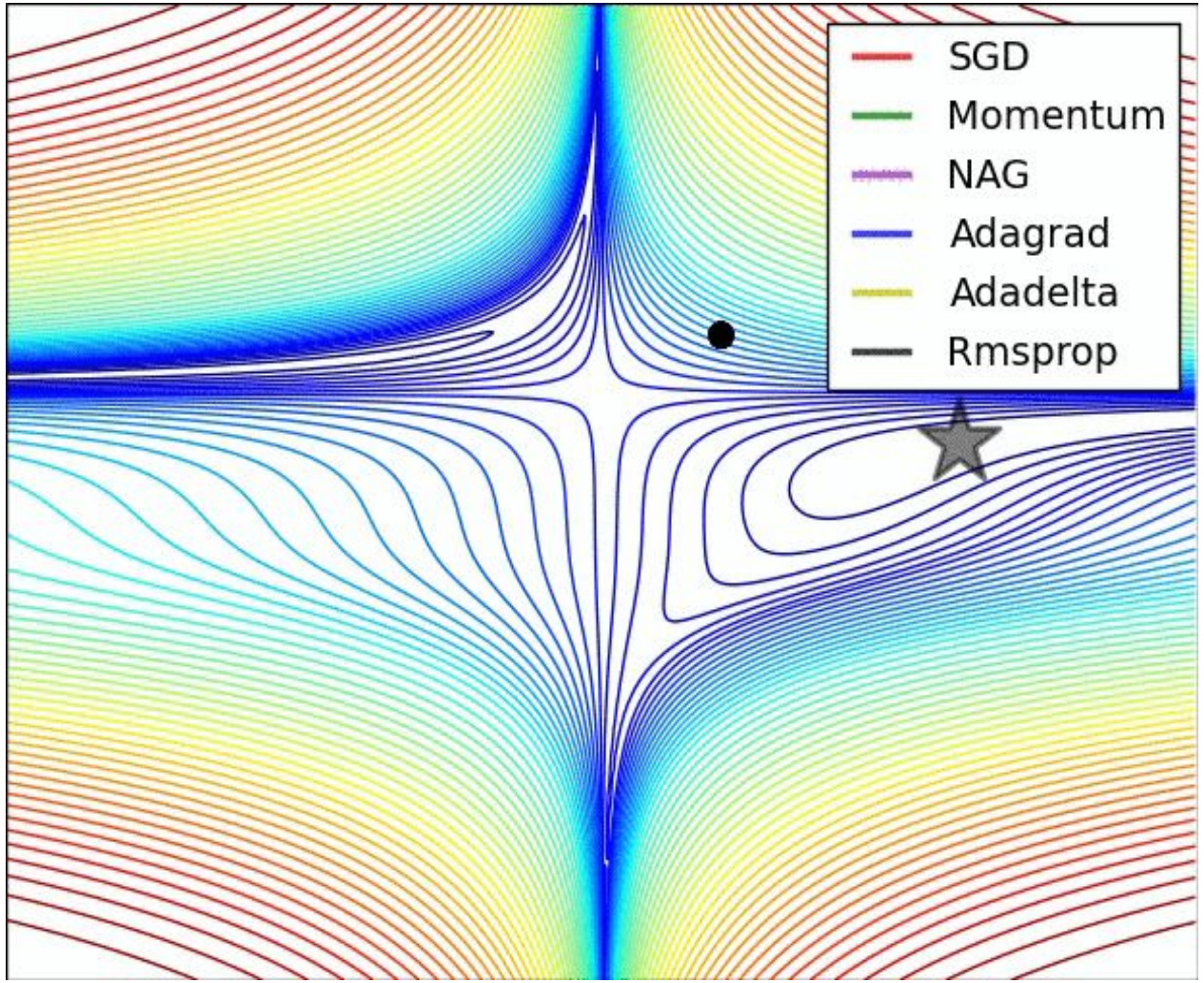
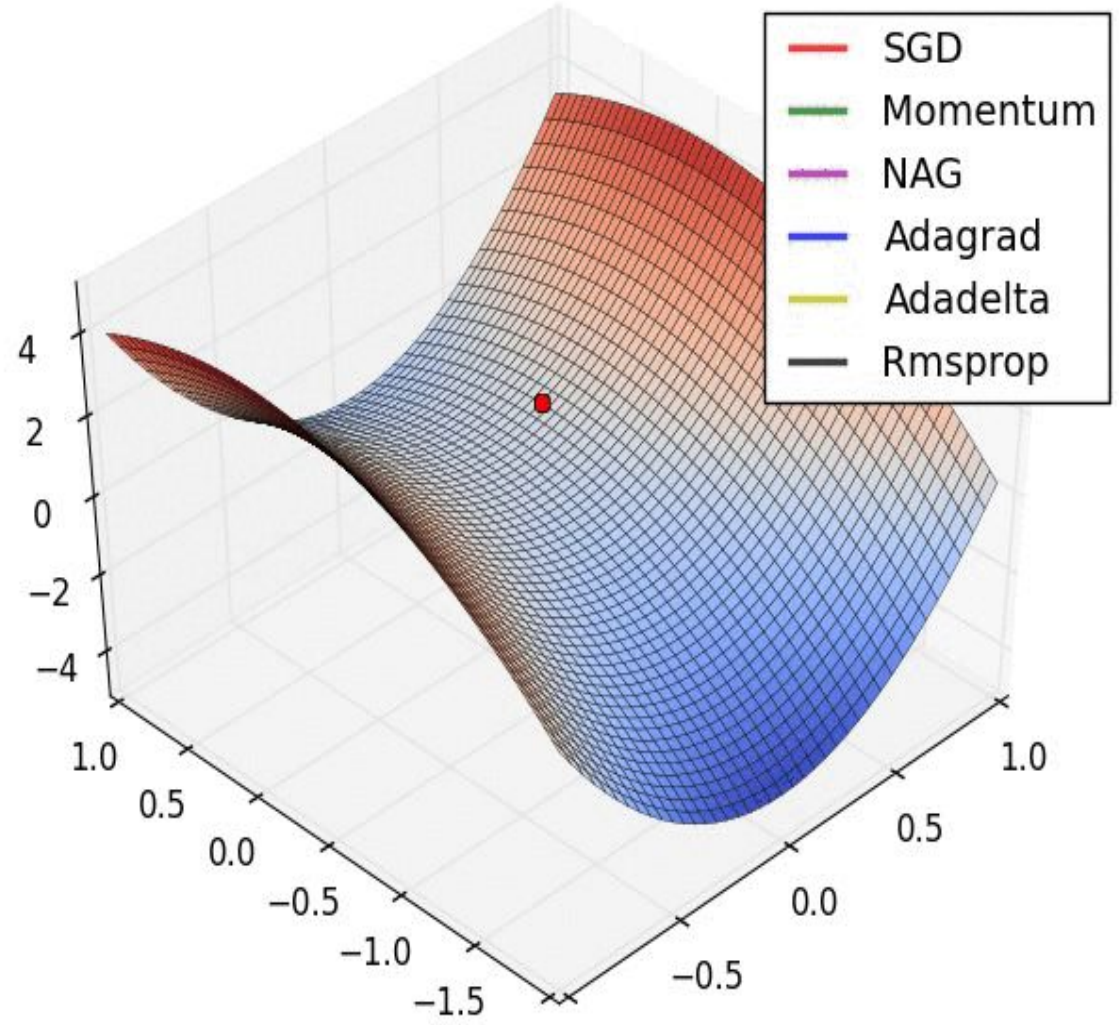
Optimizers


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

Optimizers

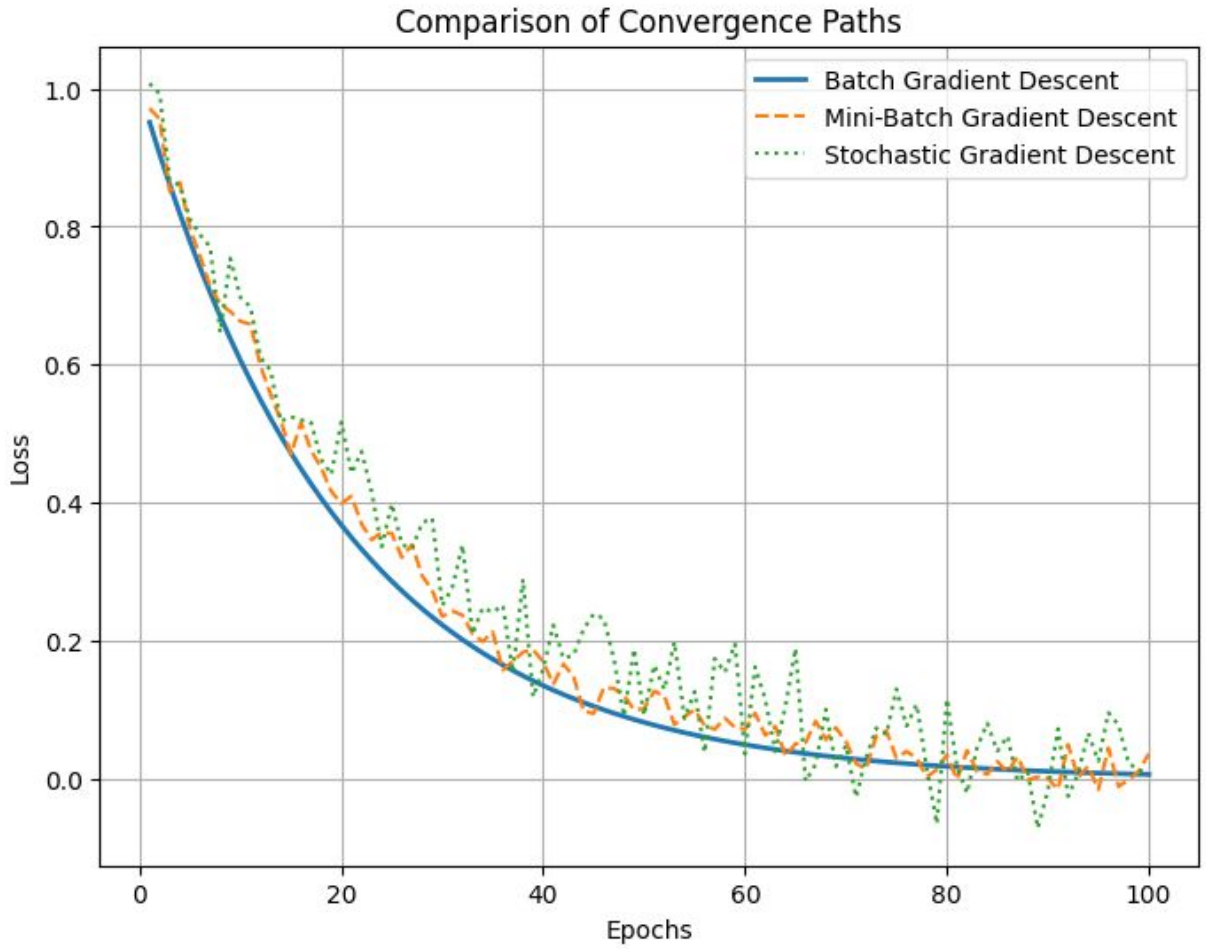
- **Batch Gradient Descent:** Uses the **entire dataset** to compute the gradient, which is more stable but can be slow for large datasets.
- **Stochastic Gradient Descent:** Updates parameters for **each data point**, which can be faster but noisier.
- **Mini-Batch Gradient Descent:** Uses **small batches** of data, balancing the speed and stability of the updates.

Optimizers

1. Stochastic Gradient Descent

The vanilla gradient descent updates the current weight w_t using the current gradient $\partial L/\partial w_t$ multiplied by some factor called the learning rate, α.

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$



Optimizers

What do gradient descent optimisers do?

There are 3 main ways how these optimisers can act upon gradient descent:

(1) modifying the learning rate component, $\boldsymbol{\alpha}$, or

(2) modifying the gradient component, $\boldsymbol{\partial L/\partial w}$, or

(3) both.

See the last term in Eqn. 1 below:

$$w_{\text{new}} = w - \alpha \frac{\partial L}{\partial w}$$

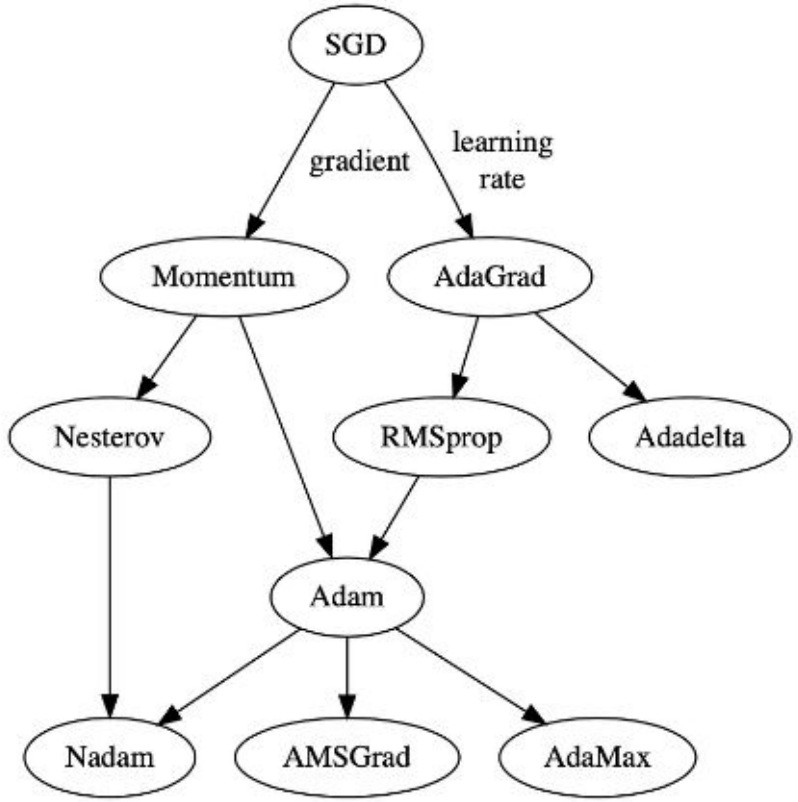
Eqn. 1: The terms in stochastic gradient descent

For (1), these optimisers multiply a positive factor to the learning rate, such that they become smaller (e.g. RMSprop). For (2), optimisers usually make use of the moving averages of the gradient (momentum), instead of just taking one value like in vanilla gradient descent. Optimisers that act on both (3) are like Adam and AMSGrad.

Optimizers

Optimiser	Year	Learning Rate	Gradient
Momentum	1964		✓
AdaGrad	2011	✓	
RMSprop	2012	✓	
Adadelta	2012	✓	
Nesterov	2013		✓
Adam	2014	✓	✓
AdaMax	2015	✓	✓
Nadam	2015	✓	✓
AMSGrad	2018	✓	✓

Fig. 2: Gradient descent optimisers, the year in which the papers were published, and the components they act upon



Optimizers

2. Momentum

Instead of depending only on the current gradient to update the weight, gradient descent with momentum (Polyak, 1964) replaces the current gradient with V_t (which stands for velocity), the exponential moving average of current and past gradients (i.e. up to time t). Later in this post, you will see that this momentum update becomes the standard update for the gradient component.

$$w_{t+1} = w_t - \alpha V_t$$

where

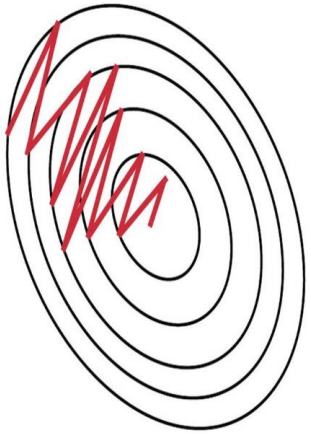
$$V_t = \beta V_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$$

and V initialised to 0.

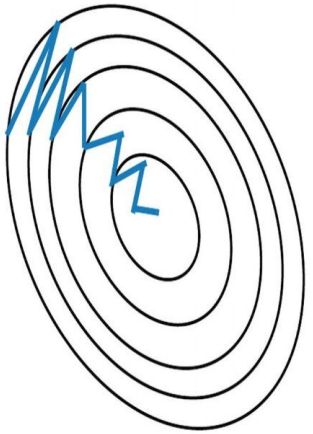
Common default value:

- $\beta = 0.9$

> Note that many articles reference the momentum method to the publication by Ning Qian, 1999. However, the paper titled Sutskever et al. attributed the classical momentum to a much earlier publication by Polyak in 1964, as cited above. (Thank you to James for pointing this out.)



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Optimizers

3. Nesterov Accelerated Gradient (NAG)

After Polyak had gained his momentum (pun intended ????), a similar update was implemented using Nesterov Accelerated Gradient (Sutskever et al., 2013). This update utilises V, the exponential moving average of what I would call *projected gradients*.

$$w_{t+1} = w_t - \alpha V_t$$

where

$$V_t = \beta V_{t-1} + (1-\beta)\frac{\partial L}{\partial w^*}$$

and V initialised to 0.

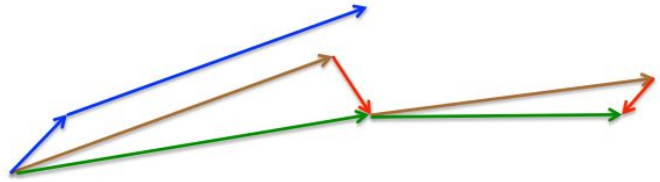
The last term in the second equation is a *projected gradient*. This value can be obtained by going 'one step ahead' using the previous velocity (Eqn. 4). This means that for this time step t, we have to carry out another forward propagation before we can finally execute the backpropagation. Here's how it goes:

1. Update the current weight w_t to a *projected weight* w^* using the previous velocity.

$$w^* = w_t - \alpha V_{t-1}$$

2. Carry out forward propagation, but using this *projected weight*.
3. Obtain the *projected gradient* $\partial L/\partial w^*$.
4. Compute V_t and w_{t+1} accordingly.

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Common default value:

- $\beta = 0.9$

> Note that the original Nesterov Accelerated Gradient paper (Nesterov, 1983) was not about *stochastic* gradient descent and did not explicitly use the gradient descent equation. Hence, a more appropriate reference is the above-mentioned publication by Sutskever et al. in 2013, which described NAG's application in stochastic gradient descent. (Again, I'd like to thank James's comment on Hacker News for pointing this out.)

Optimizers

4. AdaGrad

Adaptive gradient, or AdaGrad (Duchi et al., 2011), works on the learning rate component by dividing the learning rate by the square root of *S*, which is the cumulative sum of current and past squared gradients (i.e. up to time *t*). Note that the gradient component remains unchanged like in SGD.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$S_t = S_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

and *S* initialised to 0.

Notice that ε is added to the denominator. Keras calls this the *fuzz factor*, a small floating point value to ensure that we will never have to come across division by zero.

Default values (from Keras):

- $\alpha = 0.01$
- $\varepsilon = 10^{-7}$

Optimizers

5. RMSprop

Root mean square prop or RMSprop (Hinton et al., 2012) is another adaptive learning rate that is an improvement of AdaGrad. Instead of taking cumulative sum of squared gradients like in AdaGrad, we take the exponential moving average of these gradients.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$S_t = \beta S_{t-1} + (1 - \beta) \left[\frac{\partial L}{\partial w_t} \right]^2$$

and S initialised to 0.

Default values (from Keras):

- $\alpha = 0.001$
- $\beta = 0.9$ (recommended by the authors of the paper)
- $\varepsilon = 10^{-6}$

Optimizers

6. Adadelta

Like RMSprop, Adadelta (Zeiler, 2012) is also another improvement from AdaGrad, focusing on the learning rate component. Adadelta is probably short for 'adaptive delta', where *delta* here refers to the difference between the current weight and the newly updated weight.

The difference between Adadelta and RMSprop is that Adadelta removes the use of the learning rate parameter completely by replacing it with *D*, the exponential moving average of squared *deltas*.

$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1} + \epsilon}}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$D_t = \beta D_{t-1} + (1 - \beta)[\Delta w_t]^2$$

$$S_t = \beta S_{t-1} + (1 - \beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

with *D* and *S* initialised to 0, and

$$\Delta w_t = w_t - w_{t-1}$$

Default values (from Keras):

- $\beta = 0.95$
- $\varepsilon = 10^{-6}$

Optimizers

7. Adam

Adaptive moment estimation, or Adam (Kingma & Ba, 2014), is a combination of momentum and RMSprop. It acts upon

(i) the gradient component by using *V*, the exponential moving average of gradients (like in momentum), and

(ii) the learning rate component by dividing the learning rate α by square root of *S*, the exponential moving average of squared gradients (like in RMSprop).

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{S}_t} + \epsilon} \cdot \hat{V}_t$$

where

$$\hat{V}_t = \frac{V_t}{1 - \beta_1^t}$$

$$\hat{S}_t = \frac{S_t}{1 - \beta_2^t}$$

are the bias corrections, and

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2) \left[\frac{\partial L}{\partial w_t} \right]^2$$

with *V* and *S* initialised to 0.

Proposed default values by the authors:

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\varepsilon = 10^{-8}$

Optimizers

8. AdaMax

AdaMax (Kingma & Ba, 2015) is an adaptation of the Adam optimiser by the same authors using infinity norms (hence 'max'). *V* is the exponential moving average of gradients, and *S* is the exponential moving average of past *p*-norm of gradients, approximated to the max function as seen below (see paper for convergence proof).

$$w_{t+1} = w_t - \frac{\alpha}{S_t} \cdot \hat{V}_t$$

where

$$\hat{V}_t = \frac{V_t}{1 - \beta_1^t}$$

is the bias correction for *V* and

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1)\frac{\partial L}{\partial w_t}$$

$$S_t = \max(\beta_2 S_t, \left|\frac{\partial L}{\partial w_t}\right|)$$

with *V* and *S* initialised to 0.

Proposed default values by the authors:

- $\alpha = 0.002$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$

Optimizers

Why take *exponential moving average* of gradients?

We need to update the weight, and to do so we need to make use of some value. The only value we have is the current gradient, so let's utilise this to update the weight.

But taking only the current gradient value is not enough. We want our updates to be 'better guided'. So let's include previous gradients too.

One way to 'combine' the current gradient value and information of past gradients is that we could take a simple average of all the past and current gradients. But this means each of these gradients are equally weighted. This would not be intuitive because spatially, if we are approaching the minimum, the most recent gradient values might provide more information that the previous ones.

Hence the safest bet is that we can take the exponential moving average, where recent gradient values are given higher weights (importance) than the previous ones.

Why divide learning rate by *root mean square* of gradients?

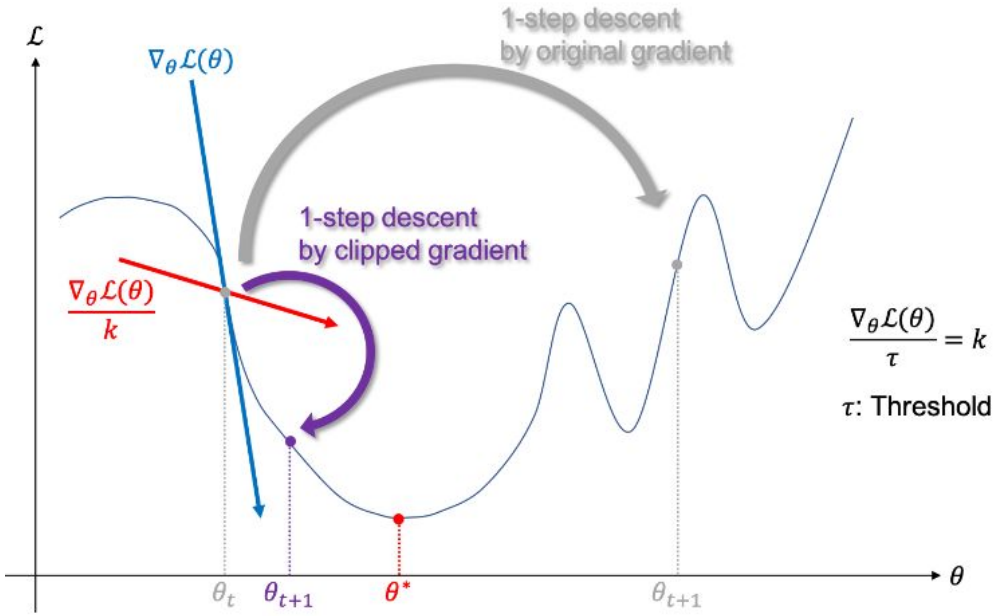
The goal is to adapt the learning rate component. Adapt to what? The gradient. All we need to ensure is that when the gradient is large, we want the update to be small (otherwise, a huge value will be subtracted from the current weight!).

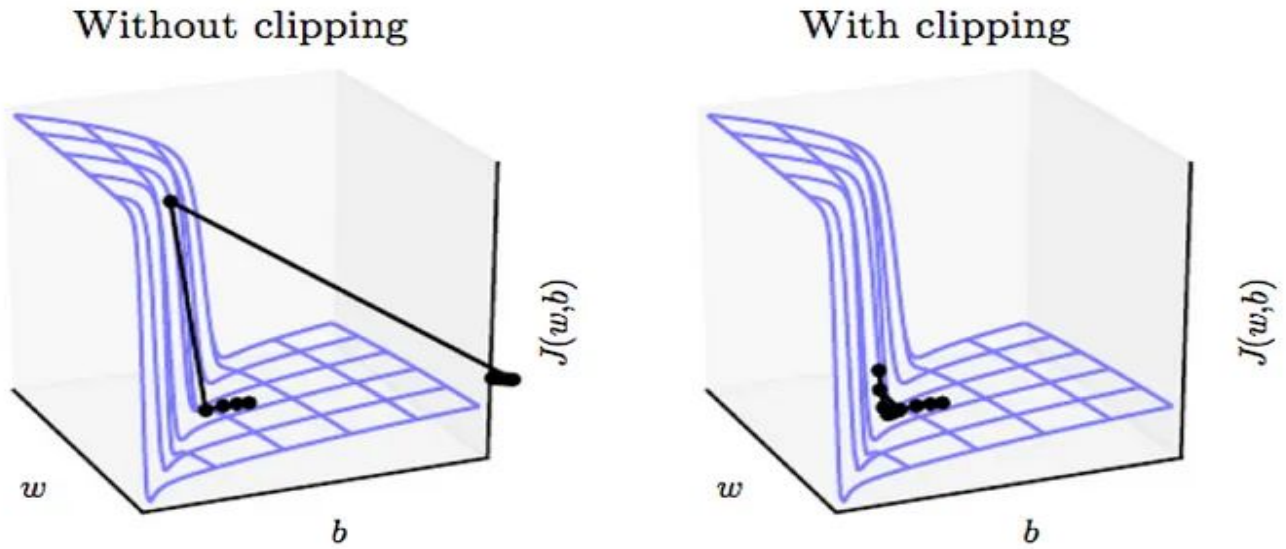
In order to create this effect, let's *divide* the learning rate α by the current gradient to get an adapted learning rate.

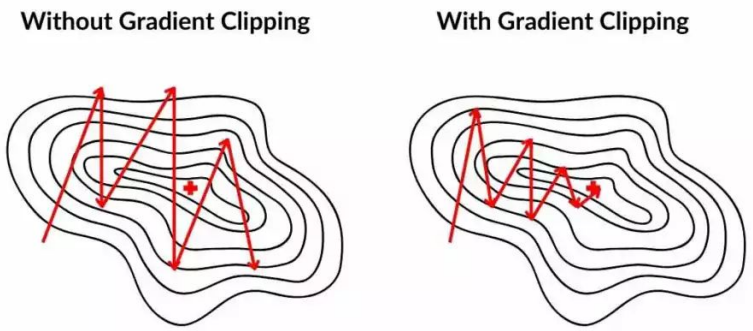
Bear in mind that the learning rate component must always be positive (because the learning rate component, when multiplied with the gradient component, should have the same sign as the latter). To ensure it's always positive, we can take its absolute value or its square. Let's take the square of the current gradient and 'cancel' back this square by taking its square root.

But like momentum, taking only the current gradient value is not enough. We want our updates to be 'better guided'. So let's make use of previous gradients too. And, as discussed above, we'll take the exponential moving average of past gradients ('mean square'), then taking its square root ('root'), hence 'root mean square'. All optimisers in this post which act on the learning rate component does this, except for AdaGrad (which takes cumulative sum of squared gradients).

Gradient Clipping







$$\frac{\partial \epsilon}{\partial \theta} \leftarrow \begin{cases} \frac{\text{threshold}}{\|\hat{g}\|}\hat{g} & \text{if } \|\hat{g}\| \geq \text{threshold} \\ \hat{g} & \text{otherwise} \end{cases}$$

$$\text{where } \hat{g} = \frac{\partial \epsilon}{\partial \theta}.$$