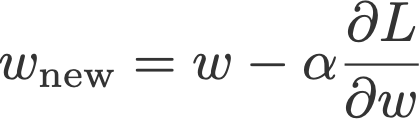


### What do gradient descent optimizers do?

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

  (1) modifying the learning rate component, **α**, or  
  (2) modifying the gradient component, **∂L/∂w**, or  
  (3) both.

See the last term in Eqn. 1 below:



Eqn. 1: The terms in stochastic gradient descent

**Learning rate schedulers vs. Gradient descent optimizers**The main difference between these two is that gradient descent optimizers adapt the learning rate component by multiplying the learning rate with a factor that is a function of the gradients, whereas learning rate schedulers multiply the learning rate by a factor which is a constant or a function of the time step.

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

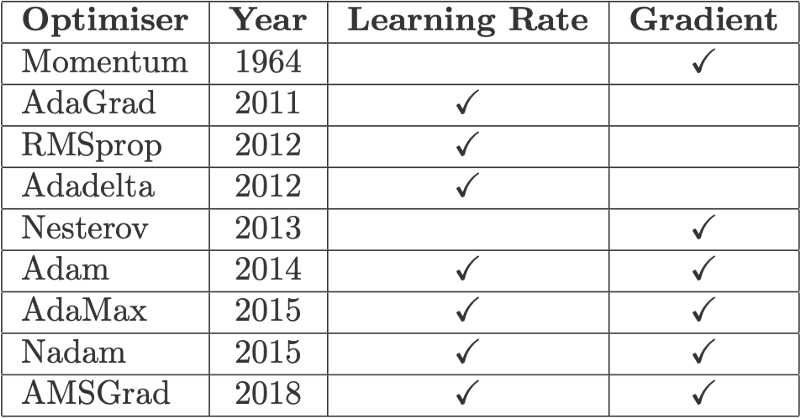


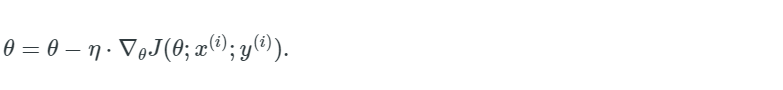
Fig. 2: Gradient descent optimizers, the year in which the papers were published, and the components they act upon

**Gradient descent variants**

## Batch gradient descent



## Stochastic gradient descent



SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image 1.

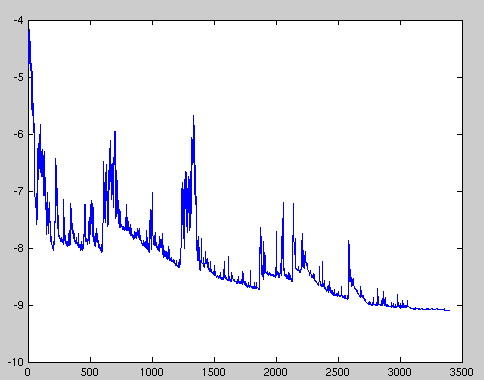
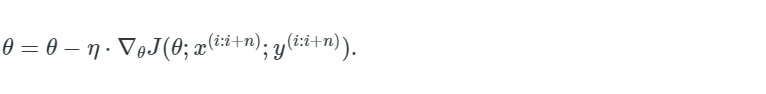


Image 1: SGD fluctuation (Source: [Wikipedia](https://upload.wikimedia.org/wikipedia/commons/f/f3/Stogra.png))

## Mini-batch gradient descent



# Challenges

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

* Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
* Learning rate schedules [[1]](https://ruder.io/optimizing-gradient-descent/index.html#fn1) try to adjust the learning rate during training by e.g., annealing, i.e., reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics [[2]](https://ruder.io/optimizing-gradient-descent/index.html#fn2).
* Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
* Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al. [[3]](https://ruder.io/optimizing-gradient-descent/index.html#fn3) argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

# Gradient descent optimization algorithms

# Momentum

Momentum [[5]](https://ruder.io/optimizing-gradient-descent/index.html#fn5) is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

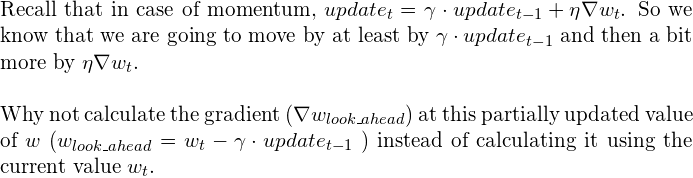


Note: Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. γ<1). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

# Nesterov accelerated gradient

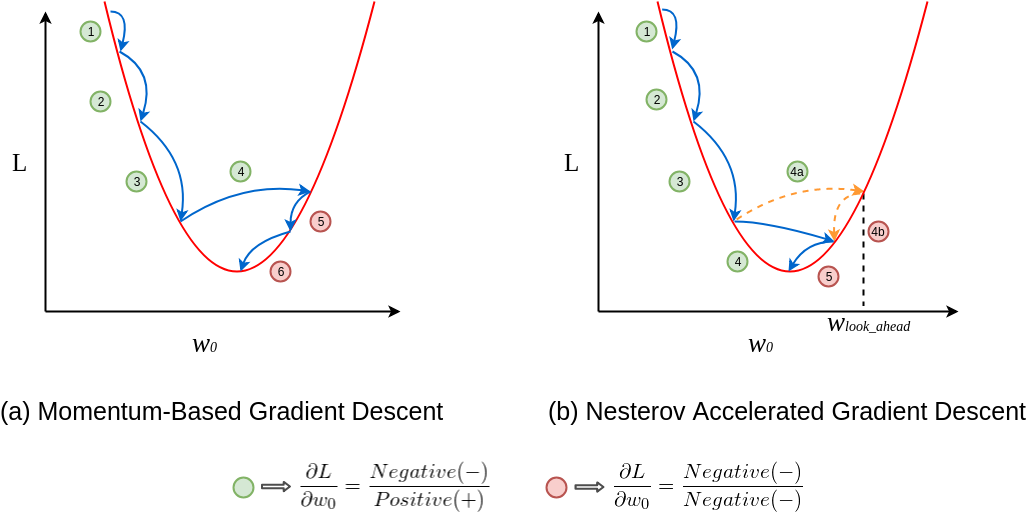
*Look ahead before you leap!*



## **The NAG Update Rule**

## 

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.



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.

# Adagrad

Adagrad [[9]](https://ruder.io/optimizing-gradient-descent/index.html#fn9) is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.