

# An overview of gradient descent optimization algorithms

## 2. Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function.

[根据计算objective function的梯度时使用的样本数量不同, gradient descent有三种不同的变形。]

2.1 Batch gradient descent

2.2 Stochastic gradient descent

2.3 Mini-batch gradient descent

## 2.1 Batch gradient descent

Batch gradient descent computes the gradient of the cost function w.r.t. to the parameter  $\theta$  **for the entire training dataset**.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

According to this equation, we need to calculate the gradient for the whole dataset to perform just one update, batch gradient descent can be very slow.

Batch gradient descent is intractable for dataset that do not fit in memory.

Batch gradient descent also does not allow us to update our model online, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):  
    params_grad = evaluate_gradient(loss_function, data, params)  
    params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector **params\_grad** of the loss function for the whole dataset w.r.t. our parameter vector **params**.

We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform.

*BGD is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.*

## 2.2 Stochastic gradient descent

Stochastic gradient descent (SGD) performs a parameter update for each training example  $x(i)$  and label  $y(i)$ :

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

*Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. (不懂)*

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

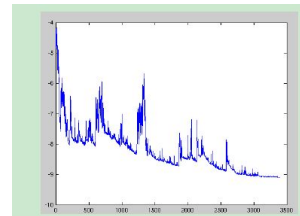


Figure 1: SGD fluctuation (Source: Wikipedia)

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting.

However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

```
for i in range(nb_epochs):  
    np.random.shuffle(data)  
    for example in data:  
        params_grad = evaluate_gradient(loss_function, example, params)  
        params = params - learning_rate * params_grad
```

## 2.3 Mini-batch gradient descent

Mini-batch gradient descent performs an update for every mini-batch of  $n$ -training examples.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- a) It reduces the variance of the parameter updates, which can lead to more stable convergence.
- b) It can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that making computing the gradient w.r.t. A mini-batch very efficient.
- c) Common mini-batch sizes range between 50 and 256, but can vary for different applications.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):  
    np.random.shuffle(data)  
    for batch in get_batches(data, batch_size=50):  
        params_grad = evaluate_gradient(loss_function, batch, params)  
        params = params - learning_rate * params_grad
```



# 3. Challenges

Mini-batch gradient descent does not guarantee good convergence, and offers a few challenges that need to be addressed.

1. Choosing a proper learning rate can be difficult;
2. Learning rate schedules try to adjust the learning rate during training by e.g. annealing.
3. Additionally, the same learning rate applies to all parameter updates.
4. Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.

## 4. Gradient descent optimization algorithms

We will outline some algorithms that are widely used by the Deep Learning community to deal with aforementioned challenges.

## 4.1 Momentum

SGD has trouble navigating ravines which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.



(a) SGD without momentum



(b) SGD with momentum

Figure 2: Source: [Genevieve B. Orr](#)

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Figure 2.

It does this by adding a fraction  $\gamma$  of the update vector of the past time step to the current update vector.

$$\begin{aligned}v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \\ \theta &= \theta - v_t\end{aligned}$$

The momentum term  $\gamma$  is usually set to 0.9 or a similar value.

The momentum 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.

**Batch gradient descent:**  $\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$

Compare with momentum, the update vector of the past time step should be

$$v_t = \eta \nabla_{\theta} J(\theta)$$

## 4.2 Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory.

Nesterov accelerated gradient (NAG) is a way to give our momentum term this kind of prescience(预知).

In momentum term  $\gamma v_{t-1}$  to move the parameter  $\theta$ . Computing  $\theta - \gamma v_{t-1}$  thus gives us an approximation of the next position of the parameters, a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameter  $\theta$  but w.r.t. the approximate future position of our parameters  $\theta - \gamma v_{t-1}$

$$\begin{aligned}v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t\end{aligned}$$

Again, we set the momentum term  $\gamma$  to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Figure 3) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks.



$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Figure 3: Nesterov update (Source: [G. Hinton's lecture 6c](#))

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.