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

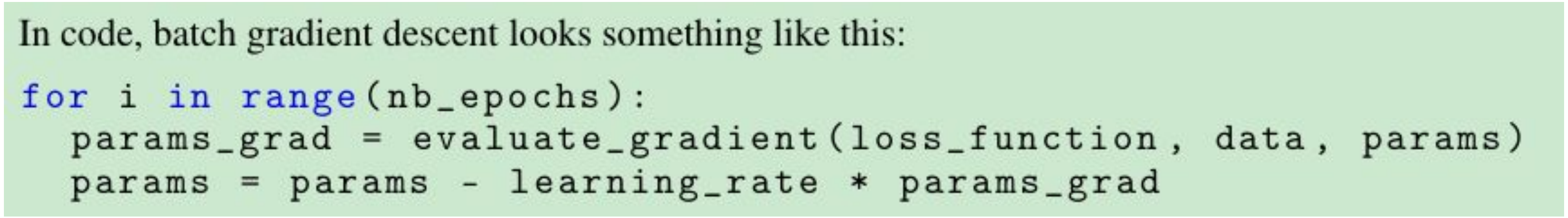
Batch gradient descent computes the gradient of the cost function w.r.t. to the parameter θ **for the entire training dataset**.

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

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.

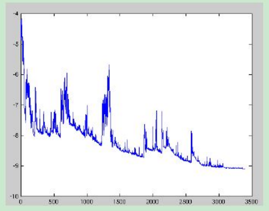


Figure 1: SGD fluctuation (Source: Wikipedia)

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 aforementiond challenges.

4.1 Momentum

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

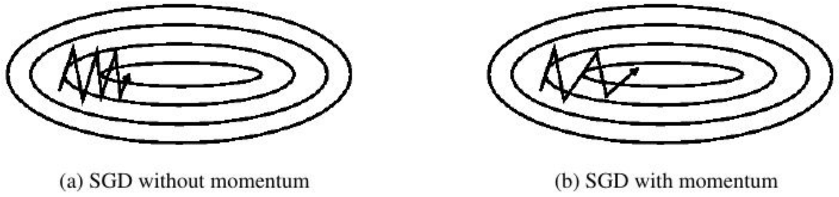


Figure 2: Source: Genevieve B. Orr

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 γ of the update vector of the past time step to the current update vector.

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

The momentum term γ 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 termγv_{t-1} to move the parameter θ. Computing θ- γv_{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 out current parameter θ but w.r.t. the approximate future position of out parameters θ- γv_{t-1}

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

Again, we set the momentum term γ 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.

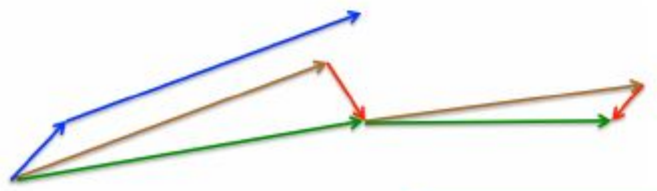


Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

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

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.

4.3 Adagrad

It adapts the learning rate to the parameters, performing larger updates for infrequent updates and smaller updates for frequent parameters. ⇒ For this reason, it is well-suited for dealing with sparse data.

Dean et al. have found that Adagrad greatly improved the **robustness** of SGD.

Previously, we performed an update for all parameters θ at once as every parameter θi used the same learning rate η.

As Adagrad uses a different learning rate for every parameter θi at every time step t, we first show Adagrad's per-parameter update, which we then vectorize.

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θi based on the past gradients that have been computed for θi:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t[11], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

4.4 Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.

Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients.

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

4.5 RMSprop

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates.

RMSprop in fact is identical to the first update vector of Adadelta.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

4.6 Adam

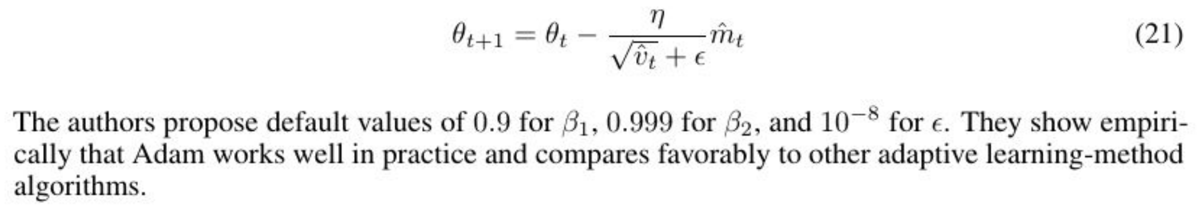
Adaptive Moment Estimation (Adam) is another method **that computes adaptive learning rates for each parameter.**

In addition to storing an exponentially decay decaying average of past square gradients vt like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients mt, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

The Adam update rule:



$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{21}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

4.7 AdaMax

The v_t factor in the Adam update rule scales the gradient inversely proportionally to the ℓ_2 norm of the past gradients (via the v_{t-1} term) and current gradient $|g_t|^2$:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)|g_t|^2 \tag{22}$$

We can generalize this update to the ℓ_p norm. Note that Kingma and Ba also parameterize β_2 as β_2^p:

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p)|g_t|^p \tag{23}$$

Norms for large p values generally become numerically unstable, which is shy l1 and l2 norms are most common in practice. However, l_{infinity} also generally exhibits stable behavior. ⇒ For this reason, the authors propose AdaMax and show that vt with l_{infinity} converges to the following more stable value.

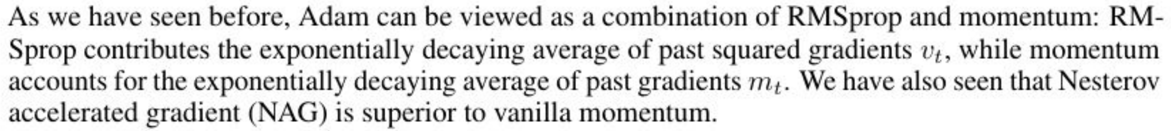
Good default values are η=0.002

beta1=0.9, beta2=0.999

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty)|g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t$$

4.8 Nadam (complex)


As we have seen before, Adam can be viewed as a combination of RMSprop and momentum: RMSprop contributes the exponentially decaying average of past squared gradients v_t, while momentum accounts for the exponentially decaying average of past gradients m_t. We have also seen that Nesterov accelerated gradient (NAG) is superior to vanilla momentum.

Nadam (Nesterov-accelerated Adaptive Moment Estimation) thus combines Adam and NAG. In order to incorporate NAG into Adam, we need to modify its momentum term mt.

NAG then allows us to perform a more accurate step in the gradient direction by updating the parameters with the momentum step before computing the gradient.