Momentum Based Gradient Descent

It takes a lot of time to navigate regions having a gentle slope. This is because the gradient is very small

Can we do something better?

Yes, let's look at Momentum Based Gradient Descent.

Intuition

If I am repeatedly being asked to move in the same direction, then I should probably gain some confidence & start taking bigger steps in that direction!

Just as a ball gains momentum while rolling down a slope.

Update Rule

$$\boxed{w_t = w_{t-1} - \eta u_t}$$

$$u_t = \beta u_{t-1} + \nabla w_t$$

$$u_0 = \nabla w_0 \quad \because u_{-1} = 0, \; 0 \le \beta \le 1$$

$$u_1 = \beta u_0 + \nabla w_1 = \beta \nabla w_0 + \nabla w_1$$

$$u_2 = \beta u_1 + \nabla w_2 = \beta(\beta \nabla w_0 + \nabla w_1) + \nabla w_2 = \beta^2 \nabla w_0 + \beta \nabla w_1 + \nabla w_2$$

$$\boxed{\therefore u_t = \sum_{\tau=0}^{t} \beta^{t-\tau} \nabla w_\tau}$$

```python
def do_mgd(max_epochs):
    w,b,eta = -2,-2,1.0
    prev_uw,prev_ub,beta = 0,0,0.9

    for i in range(max_epochs):
        dw,db = 0,0
        for x,y in zip(X,Y):
            dw += grad_w(w,b,x,y)
            db += grad_b(w,b,x,y)

        uw = beta*prev_uw+eta*dw
        ub = beta*prev_ub+eta*db
        w = w - vw
        b = b - vb
        prev_uw = uw
        prev_ub = ub
```

Observations

Even in regions having gentler slopes, mgd is able to take large steps because the momentum carries it along.

Is moving fast always good? Would there be a situation where momentum would cause us to run past it's goal?

Nesterov Accelerated Gradient Descent

Can we do something to reduce the oscillations of MGD?

Intuition

Look before you leap

Recall that $u_t = \beta u_{t-1} + \nabla w_t$

We are going to move at least by βu_{t-1} & then a bit more by ∇w_t

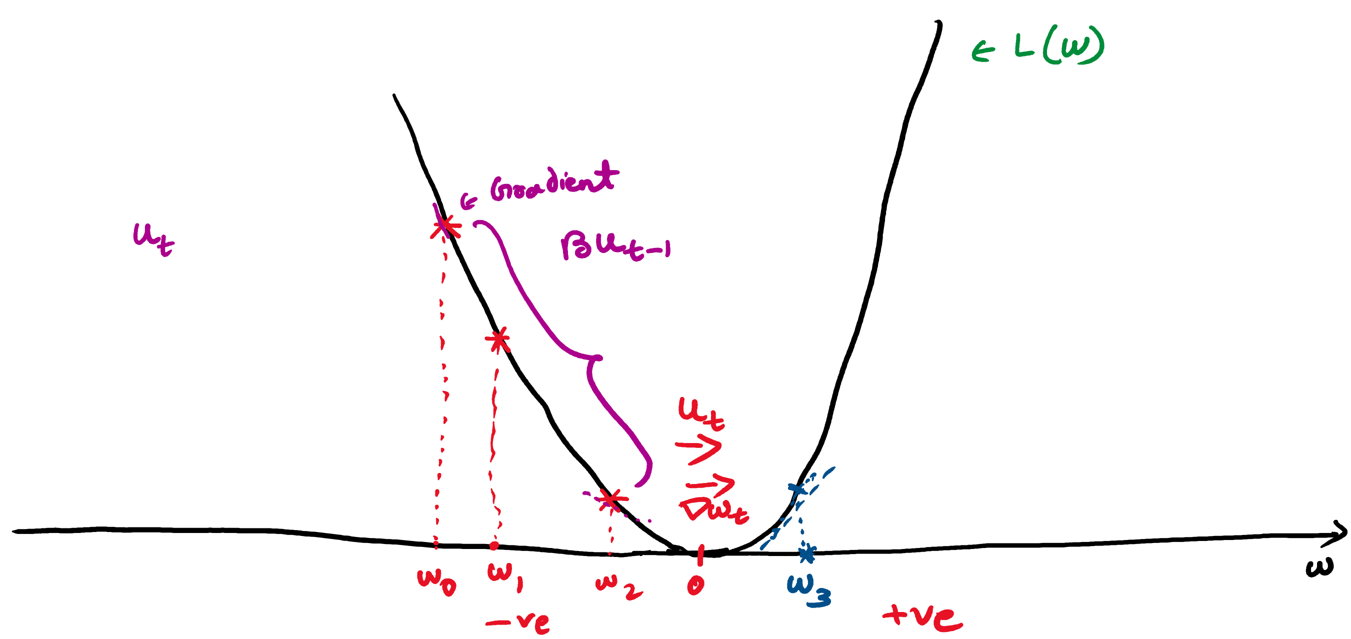
Why not make use of this to look ahead instead of relying only on the current ∇w_t

Update Rule

$$u_t = \beta u_{t-1} + \nabla(w_t - \beta u_{t-1})$$

$$w_{t+1} = w_t - \eta u_t$$

$$u_{-1} = 0, \quad 0 \leq \beta \leq 1$$



As both u_t & ∇w_t are asking you to move in the right direction, you might end up taking a large step.

What NAG is saying that you are currently at w_2, move by βu_{t-1} amount & then compute the derivative not before at w_3

```python
def do_nag(max_epochs):
    w,b,eta = -2,-2,1.0
    prev_vw,prev_vb,beta = 0,0,0.9

    for i in range(max_epochs):
        dw,db = 0,0
        # do partial updates
        v_w = beta*prev_vw
        v_b = beta*prev_vb
        for x,y in zip(X,Y):
            # Look ahead
            dw += grad_w(w-v_w,b-v_b,x,y)
            db += grad_b(w-v_w,b-v_b,x,y)
        vw = beta*prev_vw+eta*dw
        vb = beta*prev_vb+eta*db
        w = w - vw
        b = b - vb
        prev_vw = vw
        prev_vb = vb
```

Observations

Looking ahead makes NAG in correcting its course quicker than momentum based gradient descent.

Hence the oscillations (see video on the sides) are smaller. & the chance of escaping minima valley are also smaller

Stochastic vs Batch Gradient

```python
import numpy as np
X = [0.5,2.5]
Y = [0.2,0.9]



def do_gradient_descent():

  w,b,eta,max_epochs = -2,-2,1.0,1000

  for i in range(max_epochs):
    dw.db = 0.0
    for x,y in zip(X,Y):
      dw += grad_w(x,w,b,y)
      db += grad_b(x,w,b,y)

    w = w - eta*dw
    b = b - eta*db
```

The gradient descent algo goes through the entire data before updating the parameters.

This is not feasible if we have a million data points & to make 1 update in w,b, we need to go through so much data.

Alternative: Stochastic Gradient Descent

Stochastic Gradient Descent

```python

def do_stochastic_gradient_descent():

  w,b,eta,max_epochs = -2,-2,1.0,1000

  for i in range(max_epochs):
    dw,db = 0,0
    for x,y in zip(X,Y):
      dw += grad_w(x,w,b,y)
      db += grad_b(x,w,b,y)
      w = w - eta*dw
      b = b - eta*db
```

Now the updates happen for every single data point

Now if we have a million data points we will make a million updates in each epoch

This is an approximate gradient

Stochastic because we are estimating the total gradient based on a single data point

When we run this algorithm, we will see many oscillations. That is because every point gives their own direction to go as oppossed to GD which takes the sum of all points

That is because every point gives their own direction to go as oppossed to GD which takes the avg of all points.

A parameter update rule which is favourable locally to one point, may harm other points. As if the points are competing with each other.

Can we remove the oscillations by improving our stochastic estimates of the gradient?

Yes, let's look at mini batch gradient descent

Say 25 points instead of each.

Batch Gradient Descent

```python
def do_minibatch_stochastic_gradient_descent():

  w,b,eta,max_epochs = -2,-2,1.0,500
  mini_batch_size = 25

  for i in range(max_epochs):
    dw,db,num_points_seen = 0
    for x,y in zip(X,Y):
      dw += grad_w(x,w,b,y)
      db += grad_b(x,w,b,y)
      num_points_seen += 1

      if num_points_seen%minibatch_size == 0:
        w = w - eta*dw
        b = b - eta*db
        dw,db = 0,0
```

The stochastic estimates are better now!

The higher the value of batch size, the more accurate the estimates are.

Points to rember

1 epoch = 1 pass over the data

1 steps = 1 update of the parameters

N = no. of data points

B = mini batch size

Algorithm | # steps in 1 epoch

Algorithm	# steps in 1 epoch
Vanilla Gradient Descent	1
Stochastic Gradient Descent	N
Mini Batch Gradient Descent	N/B

Scheduling Learning Rate

Tips for initial learning rate

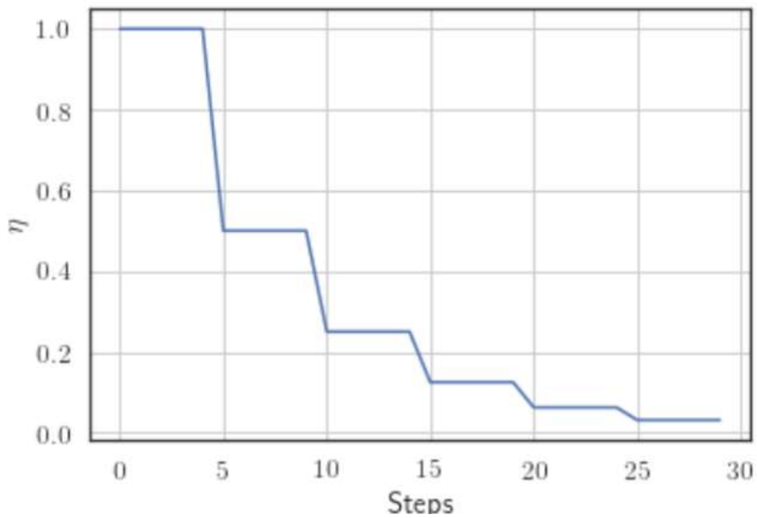
- Tune learning rate - Try different values on log scale
  0.0001, 0.001, 0.01, 0.1, 1.0
- Run for a few epochs with each of these & see which works the best
- Now do a finer search around this value
  eg- if best learning rate was 0.1 then try values around 0.1 such as 0.05, 0.2, 0.3

Tips of annealing (keep reducing) the learning rate

Step Decay:

Halve the learning rate after every 5 epochs or

Halve the learning rate after every an epoch if the validation error is more than what it was at the end of previous epoch.

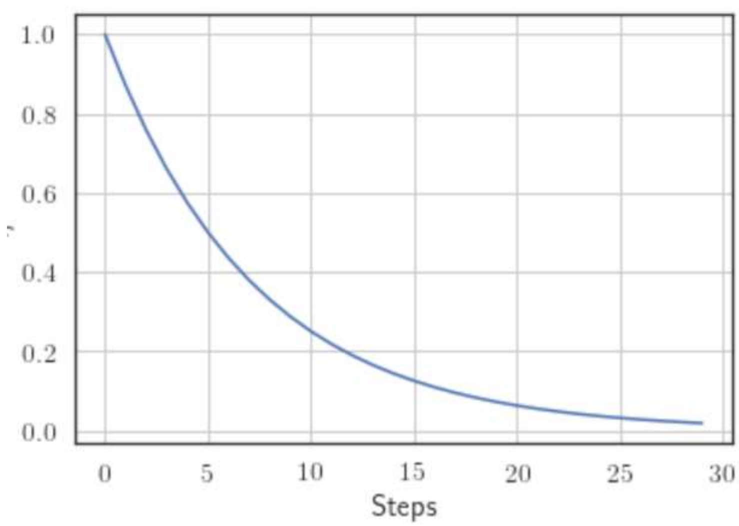


Exponential Decay:

$$\eta = \eta_0^{-kt}$$

$\eta_0, k \rightarrow$ hyperparameters

$t \rightarrow$ step number

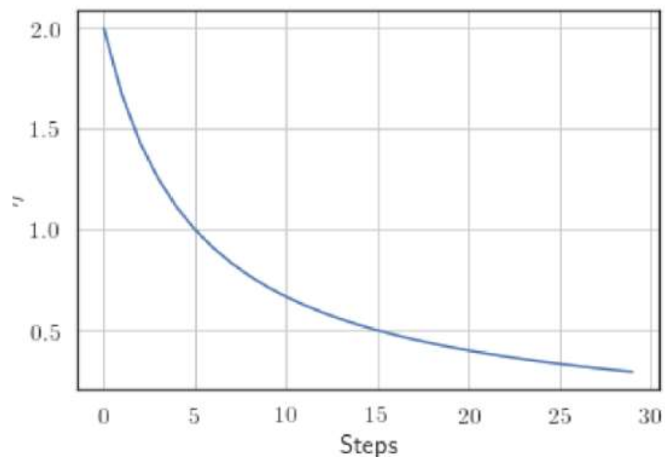


1/t Decay:

$$\eta = \frac{\eta_0}{1+kt}$$

$\eta_0, k \rightarrow$ hyperparameters

$t \rightarrow$ step number



The following schedule was suggested by Sutskever

$$\beta_t = \min\left(1 - 2^{-1-\log_2\left(\lfloor \frac{t}{250} \rfloor + 1\right)}, \beta_{max}\right)$$

when β_{max} is chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$

Line Search

```python
def do_line_search_gradient_descent(max_epochs):
    w,b,etas = -2,-2,[0.1,0.5,1,2,10]

    for i in range(max_epochs):

        dw,db = 0,0
        for x,y in zip(X,Y):
            dw += grad_w(w,b,x,y)
            db += grad_b(w,b,x,y)

        min_error = 10000  # some large value
        for eta in etas:
          temp_w = w - eta * dw
          temp_b = b - eta * db
          if error(temp_w,temp_b) < min_error:
            best_w = temp_w
            best_b = temp_b
            min_error = error(best_w,best_b)
        w = best_w
        b = best_b
```

A line search can be done to find a relatively better value of η.

Update w using different values of η

Keep the keep the updated value of w which gives lowest loss

Cons ?

More computations.