Why Gradient descent isn't enough: A comprehensive introduction to optimization algorithms in neural networks

vikashraj luhaniwal Follow
May 7, 2019 · 14 min read ★

The goal of neural networks is to minimize the ***loss,*** for producing better and accurate results. In order to minimize the loss, we need to update the internal learning parameters(especially ***weights*** and ***biases***). These parameters are updated based on some *update rule/function*. Generally, we think about ***Gradient descent*** as an update rule. Now two types of questions arise w.r.t parameters update.

- **How much/what data should be used for an update?**
- **What update rule should be used?**

This post revolves around these two questions and answers in the simplest way in the context of better optimization. In this post, I will present an intuitive vision of optimization algorithms, their different types, and variants.

> ***Additional NOTE***
>
> *This article assumes that the reader has basic knowledge about the concept of the neural network, forward and backward propagation, weight initalization, activation functions, etc. In case you are not familiar then I would recommend you to follow* **my other articles** *on these topics.*
>
> *Forward propagation in neural networks — Simplified math and code version*
>
> *Why better weight initialization is important in neural networks?*

> *Analyzing different types of activation functions in neural networks — which one to prefer?*

Optimization algorithms

Optimization algorithm tries to minimize the **loss(cost)** by following some update rule. The loss is a *mathematical* function denoting the difference between the *predicted value* and *actual value*. Loss is dependent on the actual value which is derived with the help of ***learning parameters(weights*** and ***biases)*** and *inputs*. Therefore learning *parameters* are very important for better training and producing accurate results. To find out the optimal value of these parameters we need to continuously *update* them. There should be some update rule for this purpose. So we use various ***optimization algorithms*** to follow some update rule and each *optimization algorithm* has a different approach to calculate, update and find out the optimal value of model parameters.

Types of optimization algorithms

Based on our first question **“How much data should be used for an update”** optimization algorithms can be classified as ***Gradient Descent***, ***Mini batch Gradient Descent***, and ***Stochastic Gradient Descent.***

In fact, the basic algorithm is *Gradient Descent. Mini-batch Gradient descent* and *Stochastic Gradient Descent* are two different strategies based on the amount of the data taken. These two are also known as the variants of *Gradient Descent.*

Gradient Descent

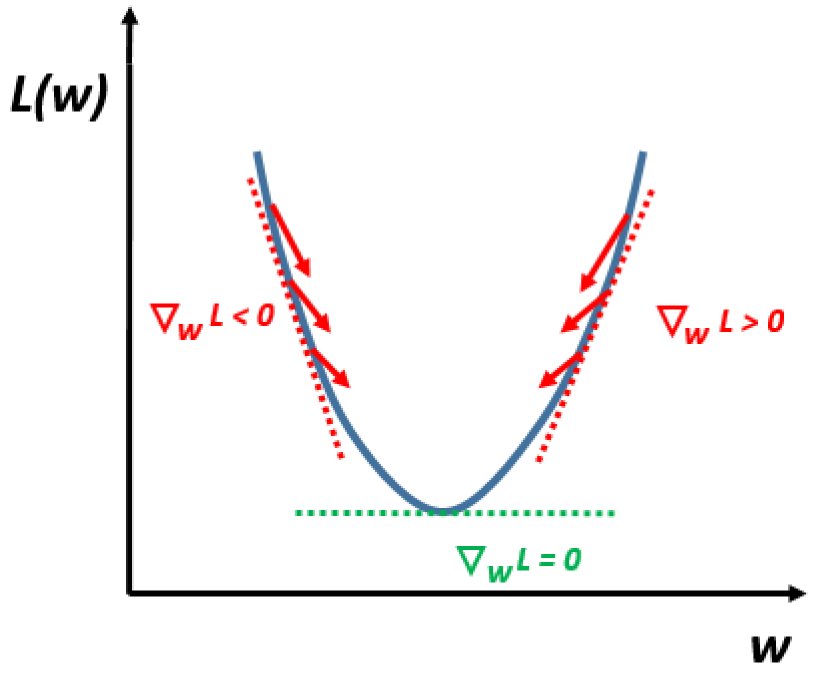
Gradient descent is most commonly used and popular ***iterative*** machine learning algorithm. It is also the foundation for other optimization algorithms. *Gradient descent* has the following *update rule* for *weight* parameter

$$\boldsymbol{w = w - \eta \nabla w \,;\; b = b - \eta \nabla b}$$

$$\text{where } \nabla w = \frac{\partial L(w)}{\partial w} \text{ and } \nabla b = \frac{\partial L(b)}{\partial b}$$

Since during *backpropagation* for updating the parameters, the *derivative of loss* w.r.t. a parameter is calculated. This *derivative* can be dependent on more than one variable so for its calculation ***multiplication chain rule*** is used. For this purpose, a ***Gradient*** is required. A ***gradient*** is a vector indicating the direction of increase.

For gradient calculation, we need to calculate *derivatives of loss* w.r.t the *parameters* and update the *parameters* in the opposite direction of the *gradient*.



The above ideal convex curve image displays the *weight update* in the opposite direction of the *gradient*. As we can notice for too large and small values of *weights* the *loss* is maximum and our goal is to *minimize* the *loss* so the *weights* are updated. If the *gradient* is negative then ***descent***(dive) towards the positive side and if the *gradient* is positive then descent towards the negative side until the minimal value of *gradient* is found.

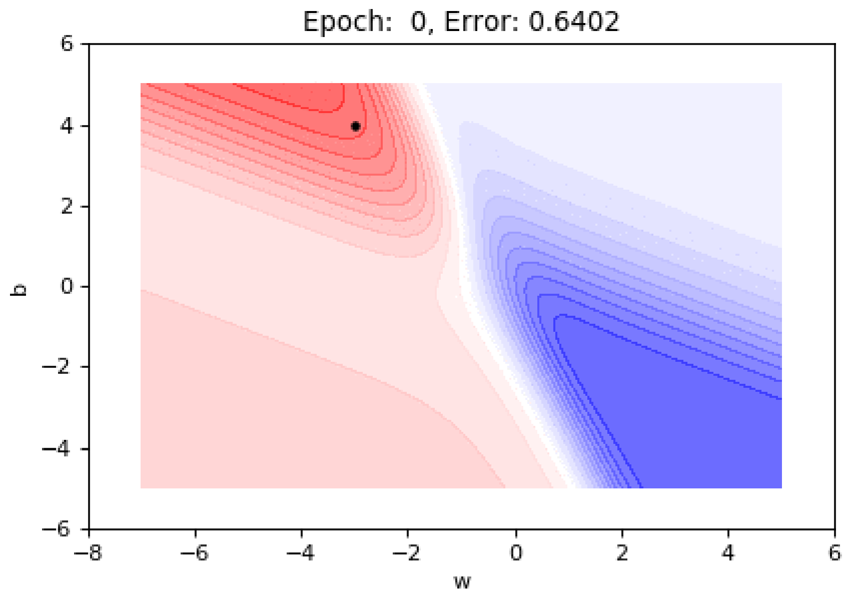
Algorithm for ***Gradient descent*** using a single neuron with *sigmoid* activation function in Python

```
def sigmoid(w,b,x):
    return 1.0 / (1.0 + np.exp(-w*x + b))

def grad_w(w,b,x,y):
    fx = sigmoid(w,b,x)
    return (fx - y) * fx * (1-fx) * x
```

```
def grad_b(w,b,x,y):
    fx = sigmoid(w,b,x)
    return (fx − y) ∗ fx ∗ (1−fx)

def do_gradient_descent():
    w,b,eta = −2, −2, 1.0
    max_epochs = 1000
    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)
        w = w − eta ∗ dw
        b = b − eta ∗ db
```



The above *animation* represents how the algorithm converges after 1000 *epochs*. The error surface used in this *animation* is as per the input. This error surface is animated in 2D space. For 2D, a *contour map* is used where the contours represent the third dimension i.e. ***error***. The red regions represent the high values of error, more the intensity of the red region more the error. Similarly, the blue regions represent the low values of error, less the intensity of the blue region less the error.

Standard Gradient descent updates the *parameters* only after each epoch i.e. after calculating the *derivatives* for all the observations it updates the *parameters*. This phenomenon may lead to the following ***caveats***.

- It can be very slow for very large datasets because only one-time *update* for each *epoch* so large number of ***epochs*** is required to have a substantial number of *updates*.
- For large datasets, the vectorization of data doesn't fit into ***memory***.
- For *non-convex* surfaces, it may only find the ***local minimums.***

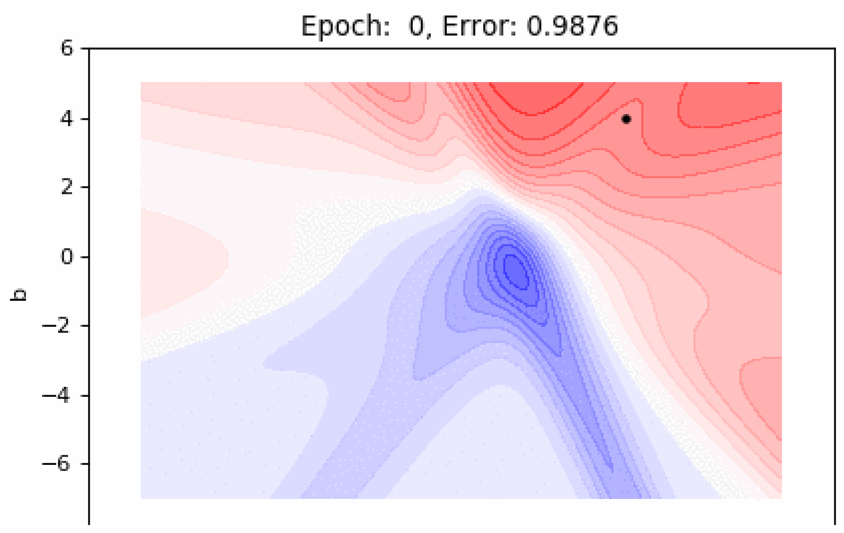
Now let see how ***different variations of gradient descent*** can address these challenges.

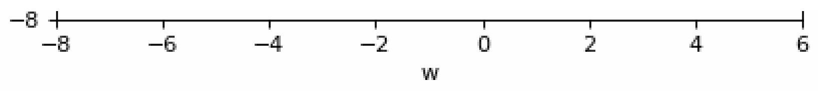
Stochastic gradient descent

Stochastic gradient descent updates the *parameters* for *each observation* which leads to more number of updates. So it is a faster approach which helps in quicker decision making.

Algorithm for *Stochastic Gradient descent* using a single neuron with *sigmoid* activation function in Python

```
def do_stochastic_gradient_descent():
    w,b,eta = -2, -2, 1.0
    max_epochs = 1000
    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)
            w = w - eta * dw
            b = b - eta * db
```





Quicker updates in different directions can be noticed in this animation. Here, lots of oscillations take place which causes the *updates* with ***higher variance*** i.e. ***noisy updates***. These noisy updates help in finding ***new*** and ***better local minima***.

Disadvantages of SGD

- Because of the ***greedy approach***, it only ***approximates (stochastics)*** the gradient.
- Due to ***frequent fluctuations***, it will keep ***overshooting*** near to the desired ***exact minima***.

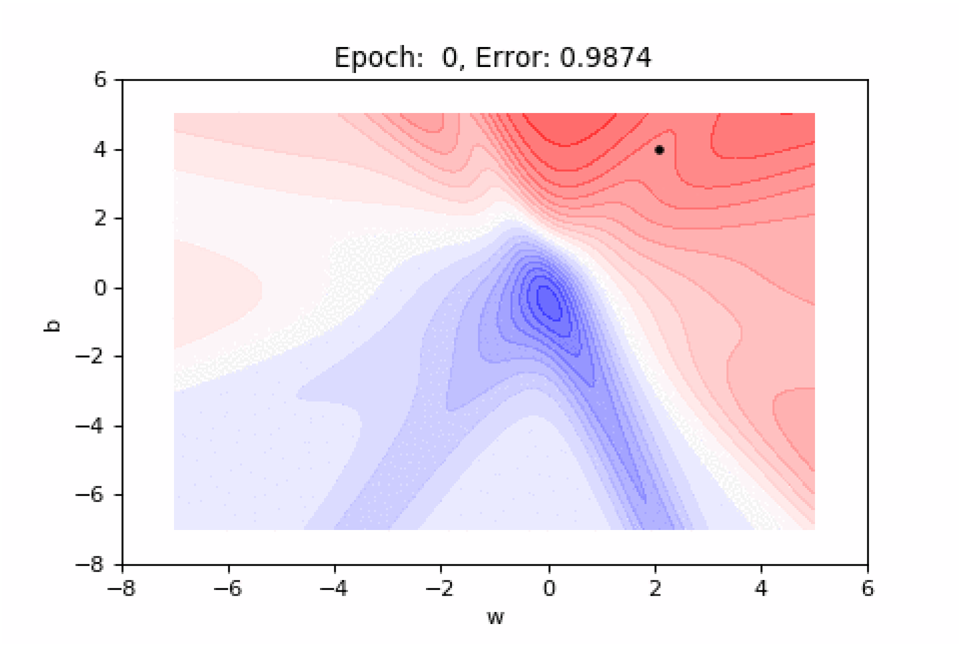
Now let see how another ***variant of gradient descent*** can address these challenges.

Mini Batch Gradient Descent

Another variant of ***GD*** to address the problems of ***SGD,*** it lies in between ***GD*** and ***SGD. Mini-batch Gradient descent*** updates the parameters for a finite number of observations. These observations together are referred to a **batch** with some fixed size. **Batch size** is chosen as a multiple of 64 e.g. 64, 128, 256, etc. Many more updates take place in one epoch through *Mini-batch GD*.

Algorithm for *Mini-batch Gradient descent* using a single neuron with *sigmoid* activation function in Python

```
def do_mini_batch_gradient_descent():
    w,b,eta = -2, -2, 1.0
    max_epochs = 1000
    mini_batch_size = 3
    num_of_points_seen = 0
    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)
            num_of_points_seen += 1
        if num_of_points_seen % mini_batch_size == 0:
            w = w - eta * dw
            b = b - eta * db
```



As we can see there are fewer oscillations in ***Mini-batch*** in contrast to ***SGD***.

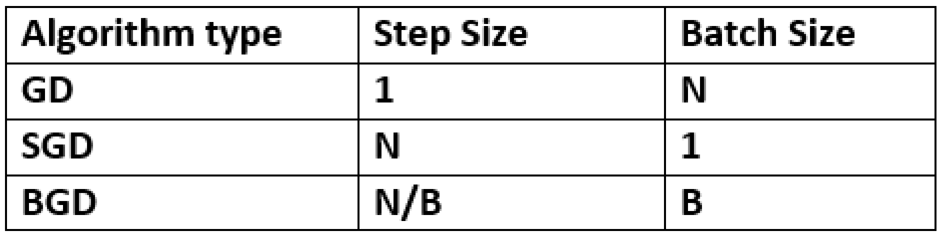
Basic notations

1 epoch = one pass over the entire data

1 step = one update for parameters

N = number of data points

B = Mini-batch size



Algorithm type	Step Size	Batch Size
GD	1	N
SGD	N	1
BGD	N/B	B

Advantages of Mini-batch GD

- Updates are less noisy compared to SGD which leads to better convergence.
- A high number of updates in a single epoch compared to GD so less number of epochs are required for large datasets.

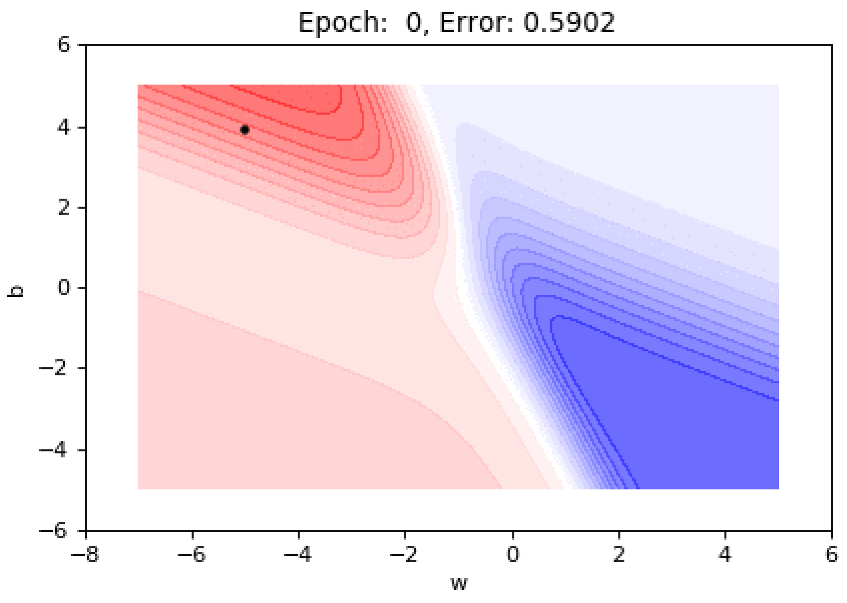
- Fits very well to the processor memory which makes computing faster.

Better optimization w.r.t. Gradient Descent

The ***error surface*** contains *more sloppy* as well *less sloppy* areas. During *back propagation*, there will be more update in parameters for the regions with *more slope* whereas less update in parameters for the regions with a *gentle slope*. More change in parameters leads to more change in loss, similarly less change in parameters leads to less change in the loss.

If the parameter initialization lands in a *gentle slope* area then it requires a large number of *epochs* to navigate through these areas. It happens so because the *gradient* will be very small in *gentle slope* regions. So it moves with ***small baby steps*** in *gentle* regions.

Consider a case with initialization in a flat surface as shown below where ***GD*** is used and the ***error*** is not reducing when the ***gradient*** is in the ***flat surface***.



Even after a large number of *epochs* for e.g. 10000 the algorithm is ***not converging***.

Due to this issue, the *convergence* is not achieved so easily and the learning takes too much time.

To overcome this problem ***Momentum based gradient descent*** is used.

Momentum-based gradient descent

Consider a case where in order to reach to your desired destination you are continuously being asked to follow the same direction and once you become confident that you are following the right direction then you start taking ***bigger steps*** and you keep getting ***momentum*** in that same direction.

Similar to this if the ***gradient*** is in a ***flat surface*** for long term then rather than taking constant steps it should take ***bigger steps*** and keep the ***momentum*** continue. This approach is known as ***momentum based gradient descent***.

Momentum-based gradient descent update rule for weight parameter

$$w_{t+1} = w_t - v_t$$

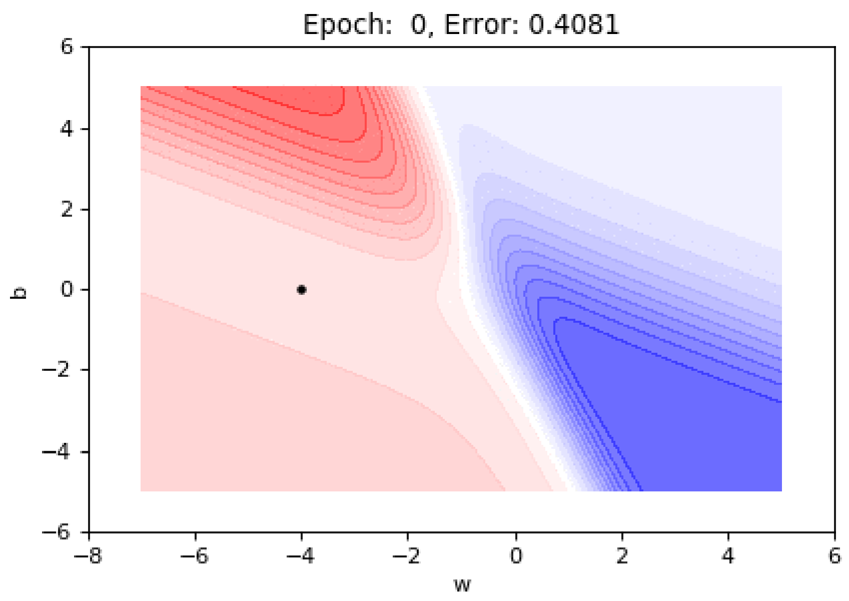
$$\text{where, } v_t = \gamma . v_{t-1} + \eta \nabla w_t$$

Gamma parameter*(γ)** is the momentum term which indicates how much acceleration you want. Here along with the ***current gradient* (η∇w(t)),** the movement is also done according to history **(γV(t−1))** so the ***update becomes larger which leads to faster movement and faster ***convergence***.

v(t) is ***exponentially decaying weighted sum***, as **t** increases **γV(t−1)** becomes smaller and smaller i.e. this equation holds the farther updates by a small magnitude and recent updates by a large magnitude.

Momentum-based gradient descent in Python for sigmoid neuron

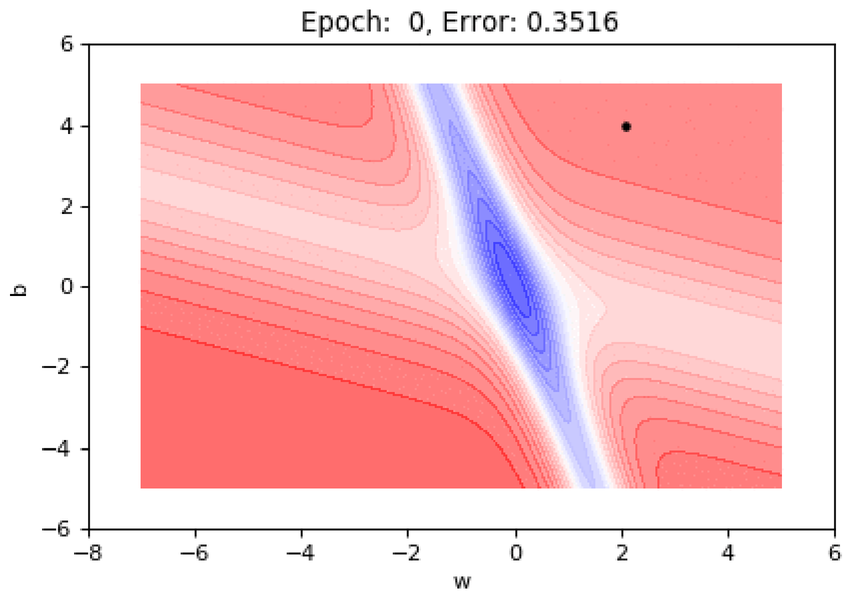
```
def do_momentum_based_gradient_descent():
    w,b,eta,max_epochs = −2, −2, 1.0, 1000
    v_w, v_b = 0, 0
    for i in range(max_epochs):
        dw,db = 0,0
        for x,y in zip(X,Y):
            dw += grad_w(w,b,x,y)
            db += grad_b(w,b,x,y)
        v_w = gamma ∗ v_w + eta ∗ dw
        v_b = gamma ∗ v_b + eta ∗ db
        w = w − v_w
        b = b − v_b
```



This algorithm ***adds momentum*** in the direction of ***consistent gradients*** and ***cancels the momentum*** if the ***gradients*** are in ***different directions***.

Issues with momentum based Gradient descent

In the ***valley*** that leads to exact ***desired minima***, there are a large number of *oscillations* using *momentum-based GD*. Because it ***overshoots*** the minima with ***larger steps*** and takes a ***U-turn*** but again overshoots so this process repeats. Which means moving with larger steps is not always good.



- ***Momentum-based GD*** *oscillates* for a ***large number of times*** in and out of the ***minima***.

To overcome this issue ***Nesterov accelerated Gradient Descent*** is used.

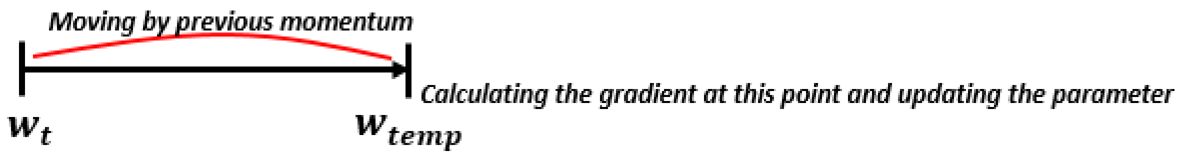
Nesterov accelerated Gradient Descent

In *momentum based GD* as the *gradient* heads to the valley(minima region), it makes a lot of ***U-turns(oscillations)*** before it *converges*. This problem was initially identified and responded by a researcher named Yurii Nesterov.

He suggested, make the *movement* first by ***history amount(previous momentum)*** then calculate the ***temporary gradient*** at this point and then *update* the *parameters*. In other words, before making an *update* directly first it looks ahead by moving with the *previous momentum* then it finds what the *gradient* should be.

This *looking ahead* helps ***NAG*** in finishing its job(finding the minima) quicker than ***momentum-based GD.*** Hence the ***oscillations*** are ***less*** compared to ***momentum based GD*** and also there are fewer chances of missing the ***minima.***

NAG update rule for *weight* parameter


Moving by previous momentum
Calculating the gradient at this point and updating the parameter
w_t
w_{temp}

$$w_{temp} = w_t - \gamma . v_{t-1}$$
$$w_{t+1} = w_{temp} - \eta \nabla w_{temp}$$
$$v_t = \gamma . v_{t-1} + \eta \nabla w_{temp}$$

NAG algorithm in Python for sigmoid neuron

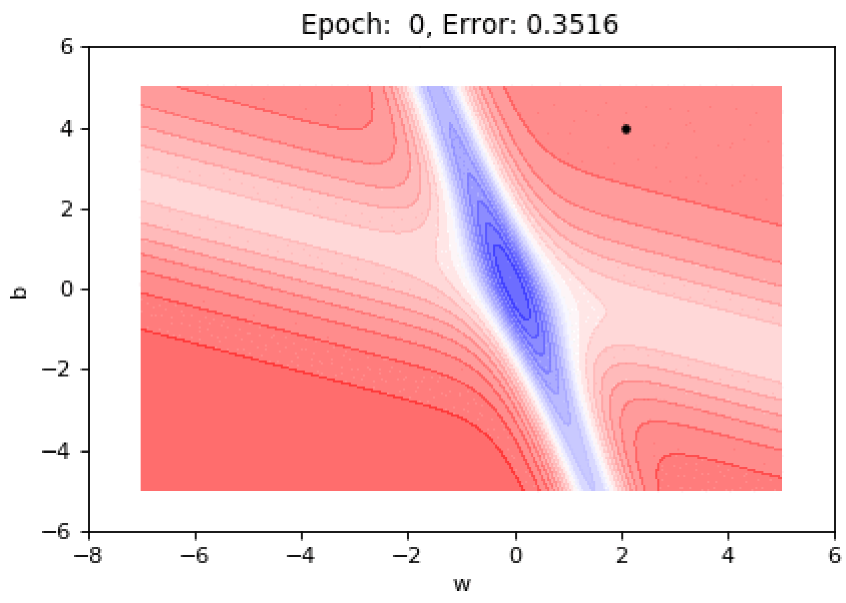
```
def do_nag_gradient_descent():
    w,b,eta,max_epochs = -2, -2, 1.0, 1000
    v_w, v_b, gamma = 0, 0, 0.9
```

```
      for i in range(max_epochs):
          dw,db = 0,0
          #compute the look ahead value
          w = w – gamma * v_w
          b = b – gamma * v_b

         for x,y in zip(X,Y):
             #compute the derivatives using look ahead value
             dw += grad_w(w,b,x,y)
             db += grad_b(w,b,x,y)
         #Now move further in the opposite direction of that gradient
         w = w – eta * dw
         b = b – eta * db

        #Now update the previous momentum
        v_w = gamma * v_w + eta * dw
        v_b = gamma * v_b + eta * db
```

Here v_w and v_b refer to **v(t)** and **v(b)** respectively.



Concept of Adaptive learning rate

As per the update rule

$$\boldsymbol{w' = w - \eta \nabla w}$$

$$\text{where, } \boldsymbol{\nabla w = (f(x) - y) * f(x) * \big(1 - f(x)\big) * x}$$

The *update* is directly proportional to the ***gradient***(**∇w**)**.** Smaller the *gradient* smaller the *update* and the *gradient* is directly proportional to the ***input***. Therefore the *update* is dependent on the *input* also.

Need for an adaptive learning rate

For the real-time datasets, most of the features are ***sparse*** i.e. having zero values. Due to this for most of the cases, the corresponding *gradient* is zero and therefore the parameters *update* is also zero. To resonate this problem, these update should be boosted i.e. a ***high learning rate*** for *sparse* features. Therefore the *learning rate* should be ***adaptive*** for fairly *sparse* data.

In other words, if we are dealing with ***sparse features*** then *learning rate* should be *high* whereas for ***dense features*** *learning rate* should be *low*.

Adagrad, RMSProp, Adam algorithms are based on the concept of *adaptive learning rate*.

Adagrad

It adopts the *learning rate(**η**)* based on the ***sparsity*** of features. So the parameters with **small *updates(sparse features)*** have high *learning rate* whereas the parameters with **large *updates(dense features)*** have low *learning rate*. Therefore **adagrad** uses a different *learning rate* for each *parameter.*

Adagrad update rule for *weight* parameter

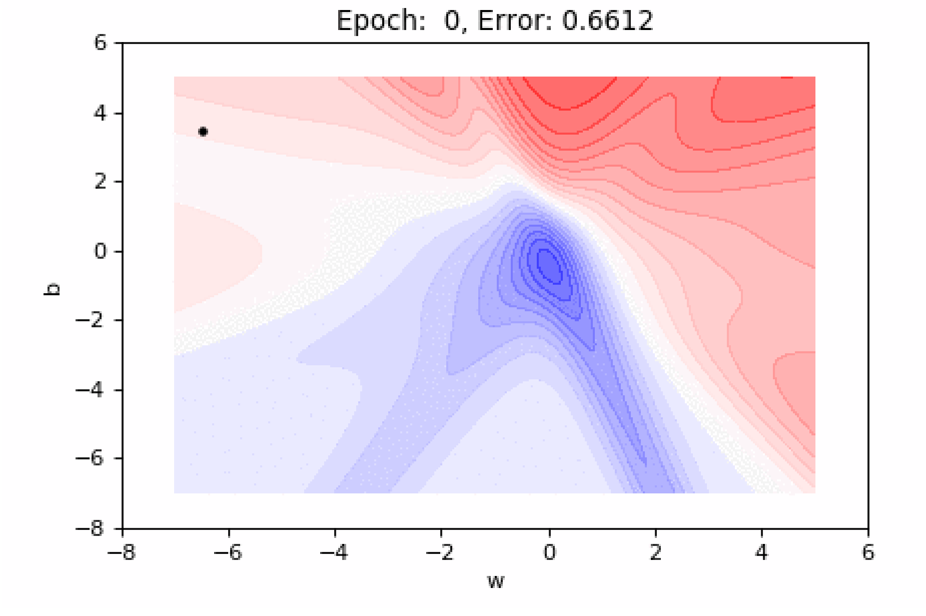
$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \nabla w_t$$

v(t) accumulates the running sum of square of the gradients. Square of **∇w(t)** neglects the sign of gradients. **v(t)** indicates accumulated gradient up to time **t**. ***Epsilon*** in the denominator avoids the chances of ***divide by zero error***.

So if **v(t)** is **low** (due to less *update* up to time **t**) for a *parameter* then the effective *learning rate* will be **high** and if **v(t)** is high for a *parameter* then effective *learning rate* will be **less**.

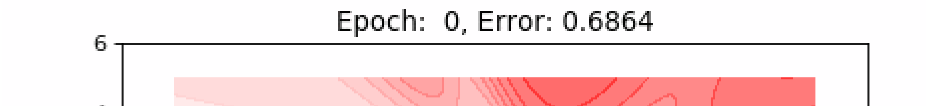
Adagrad algorithm in Python for sigmoid neuron

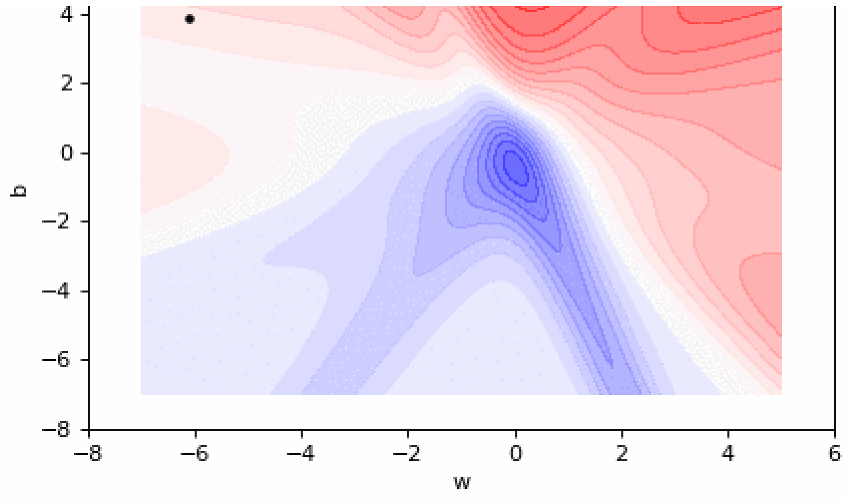
```
def do_adagrad():
    w,b,eta,max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    for i in range(max_epochs):
        dw,db = 0,0
        for x,y in zip(X,Y):
            dw += grad_w(w,b,x,y)
            db += grad_b(w,b,x,y)
        v_w += dw**2
        v_b += db**2
        self.w -= (eta / np.sqrt(v_w) + eps) * dw
        self.b -= (eta / np.sqrt(v_b) + eps) * db
```



Disadvantage with Adagrad

- The learning rate decays very aggressively





For *parameters*(especially *bias*) corresponding to *dense features*, after a few *updates*, the *learning rate* **decays** rapidly as the denominator grows rapidly due to the accumulation of *squared gradients*. So after a finite number of *updates*, the algorithm refuses to learn and *converges slowly* even if we run it for a large number of *epochs*. The *gradient* reaches to a bad minimum (close to desired *minima)* but not at exact *minima*. So **adagrad** results in decaying and decreasing learning rate for *bias* parameters.

RMSProp

RMSProp overcomes the ***decaying learning rate*** problem of ***adagrad*** and prevents the rapid growth in **v(t).**

Instead of accumulating *squared gradients* from the beginning, it accumulates the *previous gradients* in some portion(weight) which prevents rapid growth of **v(t)** and due to this the algorithm keeps learning and tries to *converge*.

RMSProp update rule for *weight* parameter

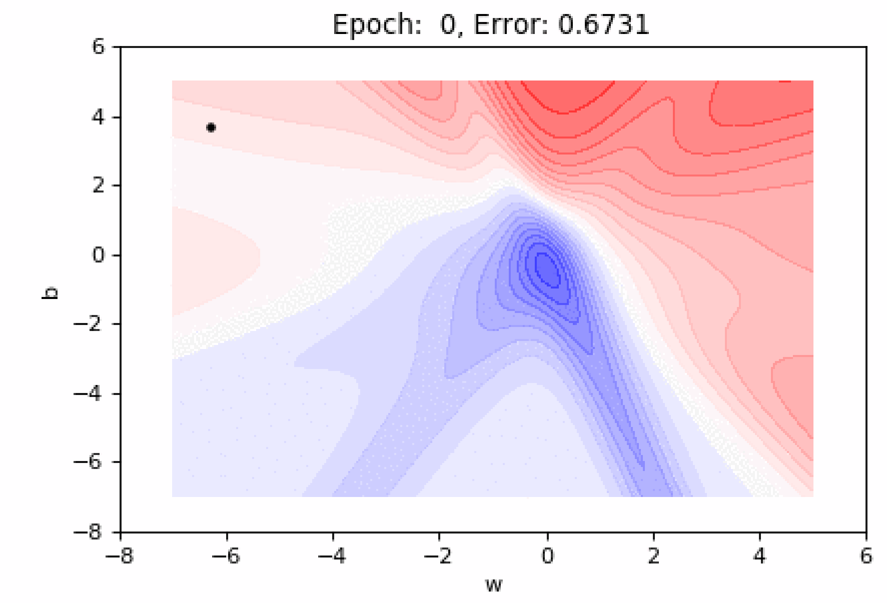
$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \nabla w_t$$

Here **v(t)** is ***exponentially decaying average*** of all the previous squared gradients. The ***beta*** parameter value is set to a similar value as the momentum term. The running average **v(t)** up to time t is dependent on ***weighted previous average gradients*** and ***current gradient***. **v(t)** maintains **$(\nabla w(t))^2$** for a fixed window time.

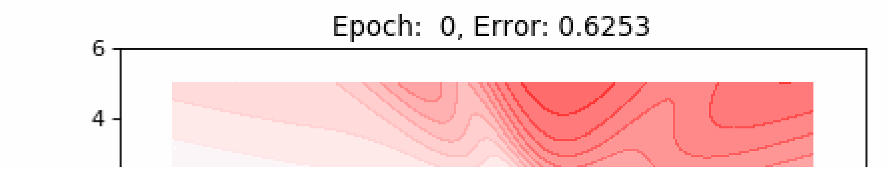
Adagrad algorithm in Python for sigmoid neuron

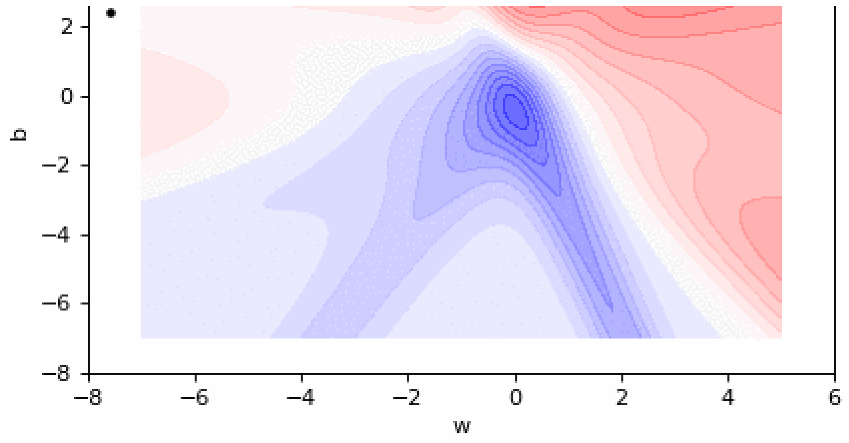
```
def do_RMSProp():
    w,b,eta,max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    for i in range(max_epochs):
        dw,db = 0,0
        for x,y in zip(X,Y):
            dw += grad_w(w,b,x,y)
            db += grad_b(w,b,x,y)
        v_w = beta * v_w + (1 - beta) * dw**2
        v_b = beta * v_b + (1 - beta) * db**2
        self.w -= (eta / np.sqrt(v_w) + eps) * dw
        self.b -= (eta / np.sqrt(v_b) + eps) * db
```



Issues with RMSProp

- A large number of oscillations with high learning rate or large gradient





So far in ***Adagrad, RMSProp*** we were calculating different ***learning rates*** for different parameters, can we have different ***momentums*** for different parameters. ***Adam*** algorithm introduces the concept of ***adaptive momentum*** along with ***adaptive learning rate***.

Adam

Adaptive Moment Estimation (Adam) computes the *exponentially decaying average of previous gradients m****(t)*** along with an ***adaptive learning rate. Adam*** is a combined form of ***Momentum-based GD*** and ***RMSProp.***

In ***Momentum-based GD,*** *previous gradients(history)* are used to compute the current gradient whereas, in RMSProp *previous gradients(history)* are used to adjust the *learning rate* based on the features. *Therefore* **Adam** deals with *adaptive learning rate* and *adaptive momentum* where ***RMSProp*** ensures **v(t)** does not grow rapidly to avoid the chances of decaying learning rate and **m(t)** from ***Momentum-based GD*** ensures it calculates the *exponentially decaying average of previous gradients,* not the *current gradient.*

Adam update rule for weight parameter

$$\boldsymbol{m_t = \beta_1 * v_{t-1} + (1-\beta_1)(\nabla w_t)}$$
$$\boldsymbol{v_t = \beta_2 * v_{t-1} + (1-\beta_2)(\nabla w_t)^2}$$

Here ***m(t)*** and ***v(t)*** are values of the ***mean*** obtained from the first moment.

Adam uses **bias corrected values (*uncentered variance)*** of gradients for *update rule* and these values are obtained through the second moment**.**

$$\hat{m}_t = \frac{m_t}{1 - {\beta_1}^t}$$

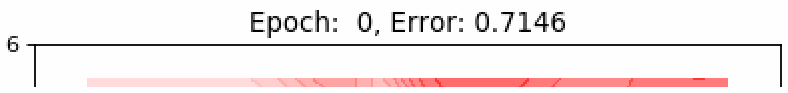
$$\hat{v}_t = \frac{v_t}{1 - {\beta_2}^t}$$

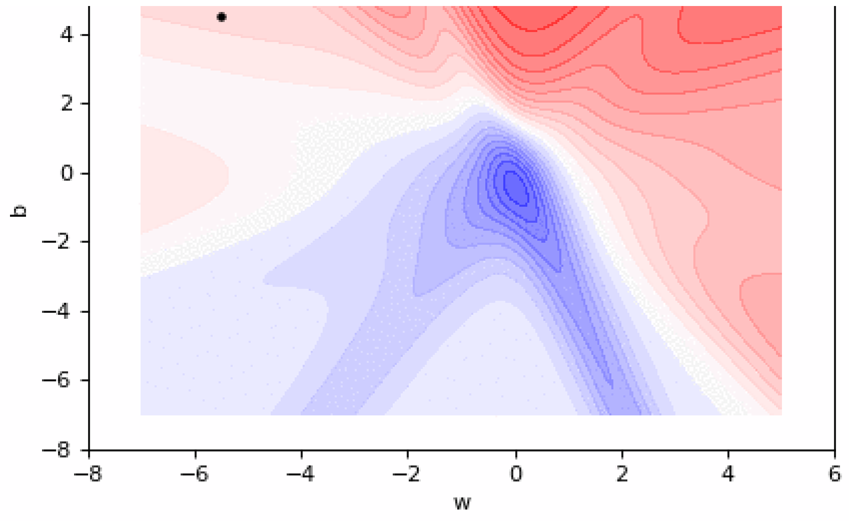
The final *update rule* is given as

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t$$

Adam algorithm in Python for sigmoid neuron

```
def do_Adam():
    w,b,eta,max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    m_w, m_b = 0, 0
    num_updates = 0
    for i in range(epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = self.grad_w(x, y)
            db = self.grad_b(x, y)
        num_updates += 1
        m_w = beta1 * m_w + (1-beta1) * dw
        m_b = beta1 * m_b + (1-beta1) * db
        v_w = beta2 * v_w + (1-beta2) * dw**2
        v_b = beta2 * v_b + (1-beta2) * db**2
        #m_w_c, m_b_c, v_w_c and v_b_c for bias correction
        m_w_c = m_w / (1 - np.power(beta1, num_updates))
        m_b_c = m_b / (1 - np.power(beta1, num_updates))
        v_w_c = v_w / (1 - np.power(beta2, num_updates))
        v_b_c = v_b / (1 - np.power(beta2, num_updates))
        self.w -= (eta / np.sqrt(v_w_c) + eps) * m_w_c
        self.b -= (eta / np.sqrt(v_b_c) + eps) * m_b_c
```





So in ***Adam*** unlike ***RMSProp*** fewer *oscillations* and it moves more deterministically in the right direction which leads to ***faster convergence*** and ***better optimization***.

End Notes

In this article, I have discussed the different types of optimization algorithms and the common issues one might encounter while using each of them. Generally, **Adam** with **mini-batch** is preferred for the training of deep neural networks.

Optimization Algorithms · Gradient Descent · Stochastic Gradient · Adam · Adagrad