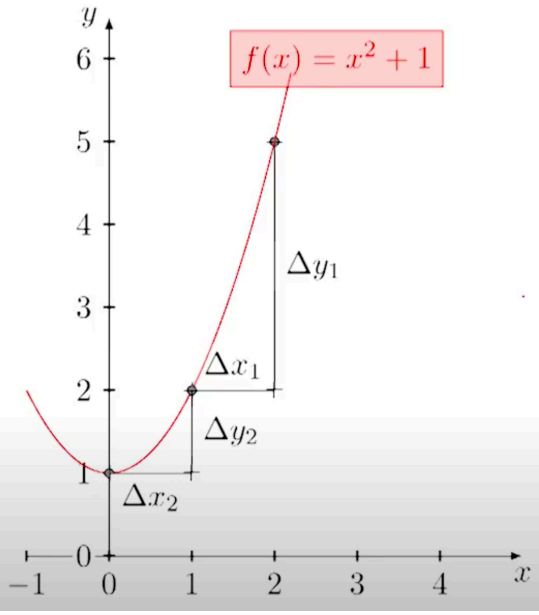
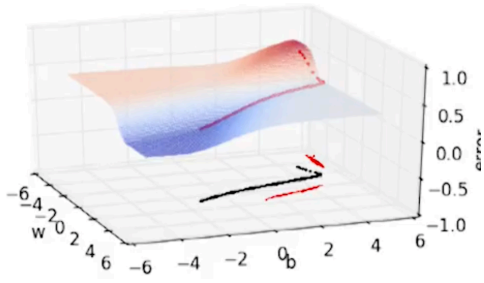
Dl bASICS3

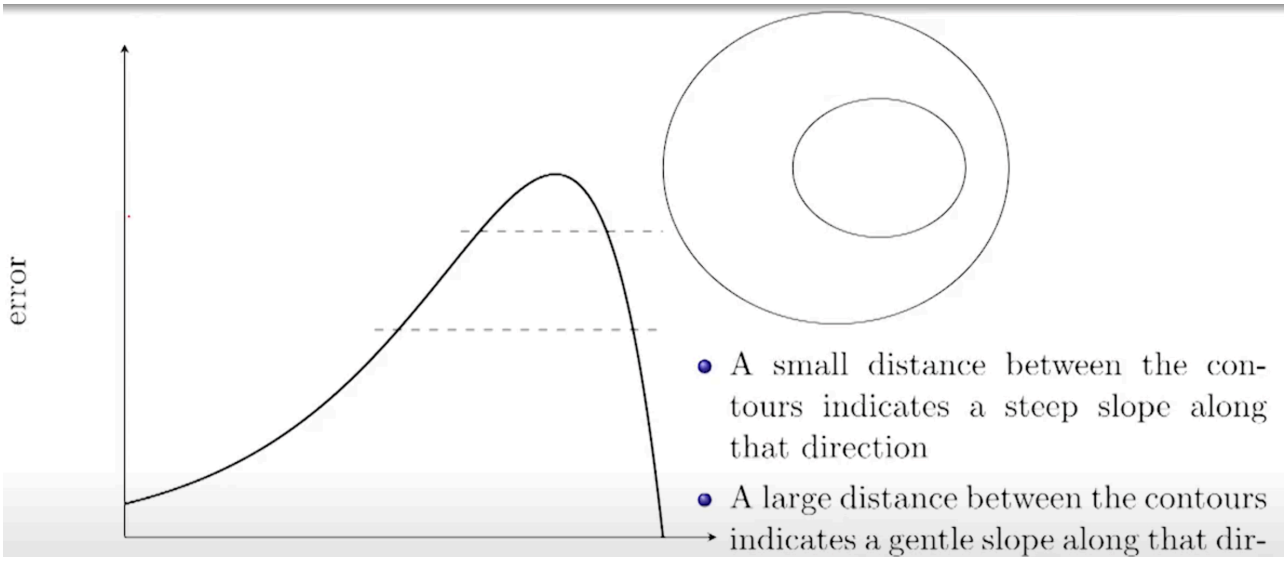
Wednesday, 20 March 2024 10:31 AM



- When the curve is steep the gradient $(\frac{\Delta y_1}{\Delta x_1})$ is large
- When the curve is gentle the gradient $(\frac{\Delta y_2}{\Delta x_2})$ is small
- Recall that our weight updates are proportional to the gradient $w = w - \eta \nabla w$
- Hence in the areas where the curve is gentle the updates are small whereas in the areas where the curve is steep the updates are large

- Irrespective of where we start from once we hit a surface which has a gentle slope, the progress slows down





- A small distance between the contours indicates a steep slope along that direction
- A large distance between the contours indicates a gentle slope along that dir-

ection

θ

the error is the same along that circle or ellipse, whatever you boundary that you see,

Some observations about gradient descent

- It takes a lot of time to navigate regions having a gentle slope
- This is because the gradient in these regions is very small
- Can we do something better ?
- Yes, let's take a look at 'Momentum based gradient descent'

Momentum based GD:

Intuition

- If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction
- Just as a ball gains momentum while rolling down a slope

Update rule for momentum based gradient descent

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

- In addition to the current update, also look at the history of updates.

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

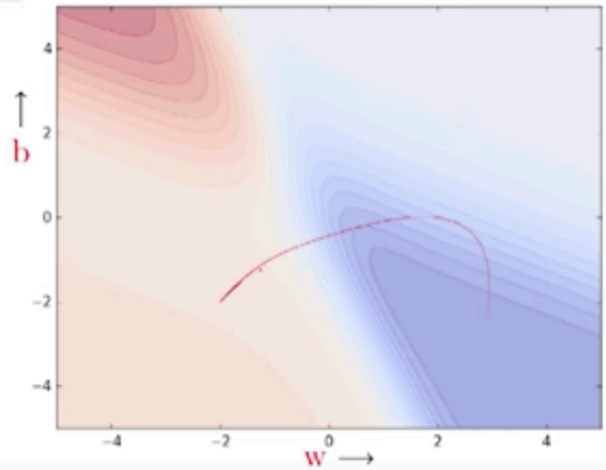
$$update_0 = 0$$
$$update_1 = \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1$$
$$update_2 = \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2$$
$$update_3 = \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3$$
$$= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3$$
$$update_4 = \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4$$

$$\vdots$$

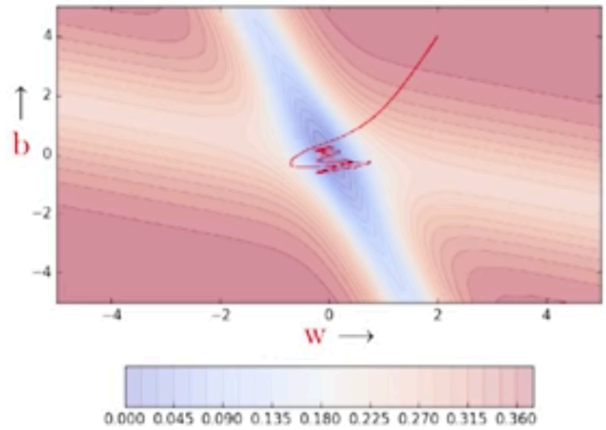
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + ... + \eta \nabla w_t$$

```
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 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)

        v_w = gamma * prev_v_w + eta* dw
        v_b = gamma * prev_v_b + eta* db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```



- Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley
- Takes a lot of u-turns before finally converging
- Despite these u-turns it still converges faster than vanilla gradient descent
- After 100 iterations momentum based method has reached an error of 0.00001 whereas vanilla gradient descent is still stuck at an error of 0.36



NAG GD::

Question

- Can we do something to reduce these oscillations ?
- Yes, let's look at Nesterov accelerated gradient

Intuition

- Look before you leap
- Recall that $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ and then a

bit more by $\eta \nabla w_t$

- Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w ($w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$) instead of calculating it using the current valuc w_t

Update rule for NAG

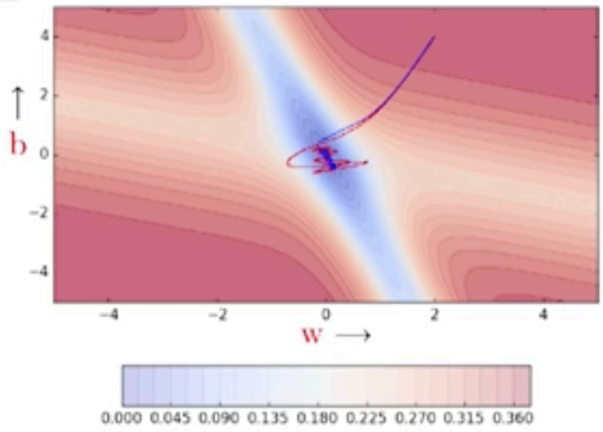
$$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead}$$
$$w_{t+1} = w_t - update_t$$

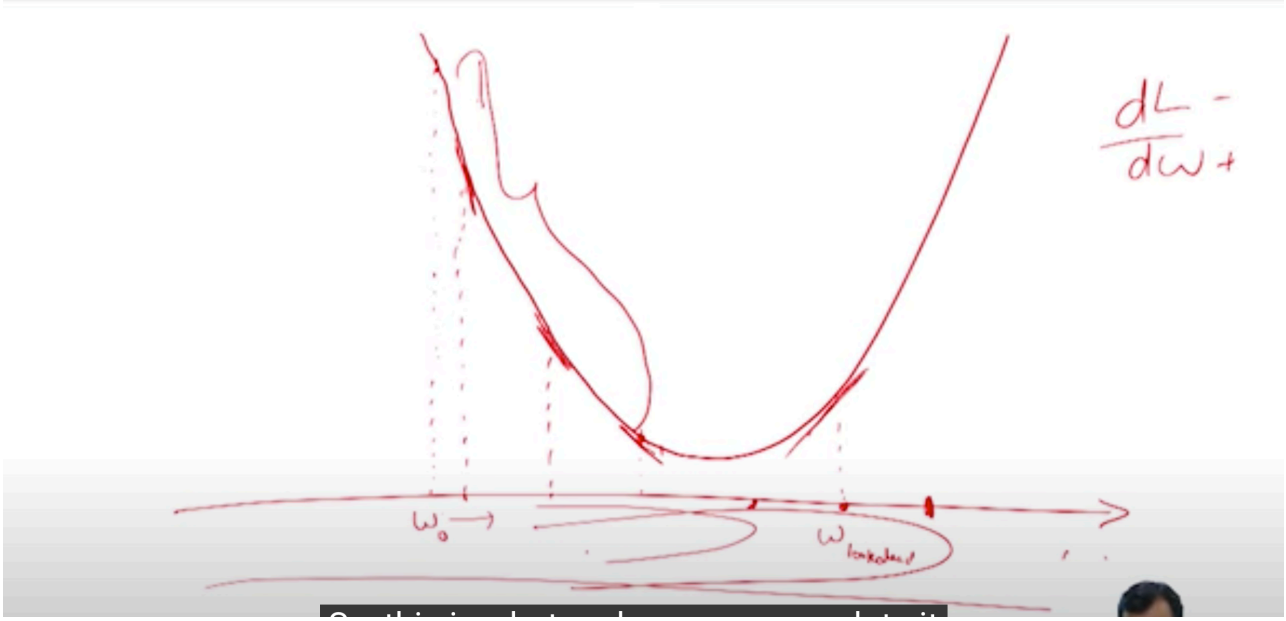
We will have similar update rule for b_t

```
def do_nesterov_accelerated_gradient_descent() :

    w, b, eta = init_w, init_b  , 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        #do partial updates
        v_w = gamma * prev_v_w
        v_b = gamma * prev_v_b
        for x,y in zip(X, Y) :
            #calculate gradients after partial update
            dw += grad_w(w - v_w, b - v_b, x, y)
            db += grad_b(w - v_w, b - v_b, x, y)

        #now do the full update
        v_w = gamma * prev_v_w + eta * dw
        v_b = gamma * prev_v_b + eta * db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```





Observations about NAG

- Looking ahead helps NAG in correcting its course quicker than momentum based gradient descent

- Hence the oscillations are smaller and the chances of escaping the minima valley also smaller

SGD:::

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

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

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(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to w, b the algorithm makes a million calculations. Obviously very slow!!
- Can we do something better ? Yes, let's look at stochastic gradient descent

```
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(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)

```
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(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

```
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(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

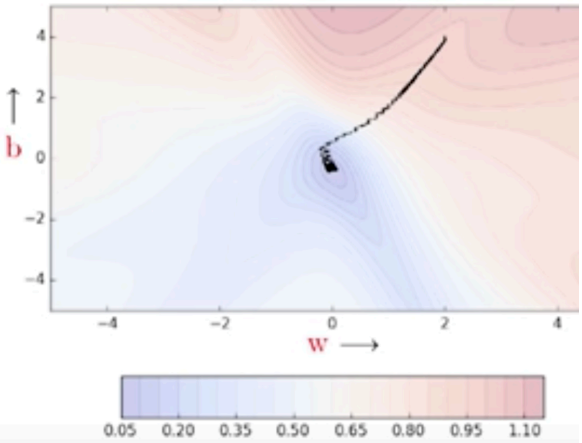
- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate

- Stochastic because we are estimating the total gradient based on a single data point. Almost like tossing a coin only once and estimating P(heads).

(rather stochastic) gradient

- No guarantee that each step will decrease the loss
- Let's see this algorithm in action when we have a few data points

- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)
- Yes, let's look at mini-batch gradient descent



Mini batch sgd::

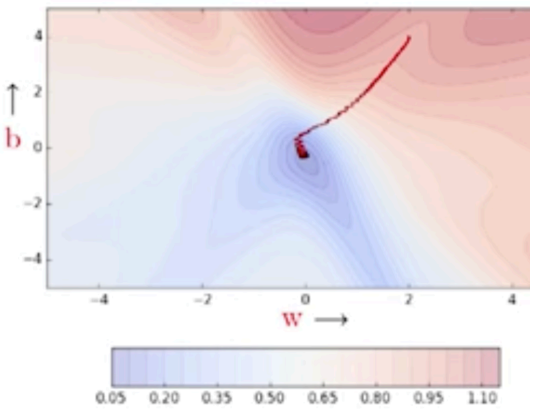
```
def do_mini_batch_gradient_descent() :
    w, b, eta = -2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```

- Notice that the algorithm updates the parameters after it sees *mini_batch_size* number of data points
- The stochastic estimates are now slightly better
- Let's see this algorithm in action when we have k = 2

```
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(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]
- The higher the value of k the more accurate are the estimates
- In practice, typical values of k are 16, 32, 64
- Of course, there are still oscillations and they will always be there as long as we are using an approximate gradient as opposed to the true gradient



Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$