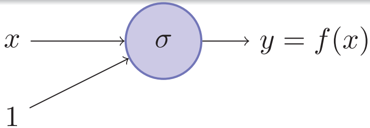
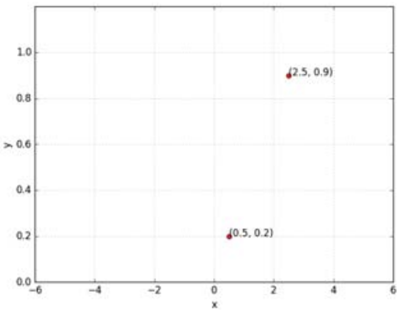
Gradient Descent (GD), Momentum Based GD, Nesterov Accelerated GD, Stochastic GD, AdaGrad, RMSProp, Adam

Learning Parameters : Infeasible (Guess Work)



$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$



Input for training

$\{x_i, y_i\}_{i=1}^{N} \rightarrow N$ pairs of (x, y)

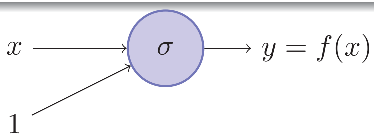
Training objective

Find w and b such that:

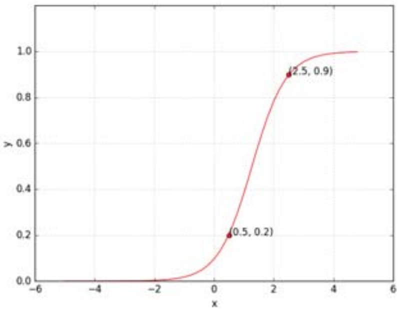
$$\underset{w,b}{\text{minimize}} \, \mathscr{L}(w, b) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

What does it mean to train the network?

- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w^*, b^* such that:
- $f(0.5) \rightarrow 0.2$ and $f(2.5) \rightarrow 0.9$



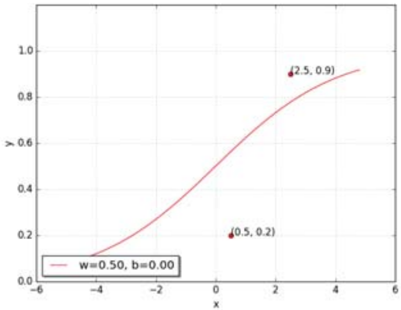
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



In other words...

- We hope to find a sigmoid function such that $(0.5, 0.2)$ and $(2.5, 0.9)$ lie on this sigmoid

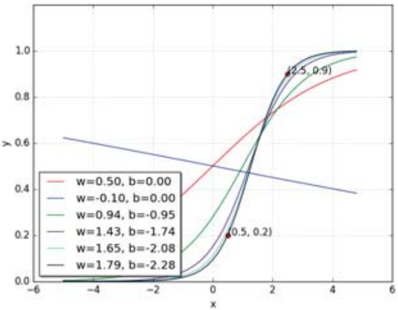
Let us see this in more detail....



- Can we try to find such a w^*, b^* manually
- Let us try a random guess.. (say, $w = 0.5, b = 0$)
- Clearly not good, but how bad is it ?
- Let us revisit $\mathscr{L}(w, b)$ to see how bad it is ...

$$\begin{aligned}\mathscr{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2 \\ &= \frac{1}{2} * ((y_1 - f(x_1))^2 + (y_2 - f(x_2))^2) \\ &= \frac{1}{2} * ((0.9 - f(2.5))^2 + (0.2 - f(0.5))^2) \\ &= 0.073\end{aligned}$$

We want $\mathscr{L}(w, b)$ to be as close to 0 as possible



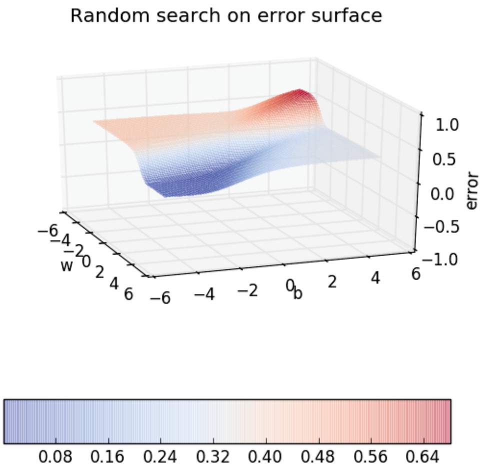
Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

Oops!! this made things even worse...

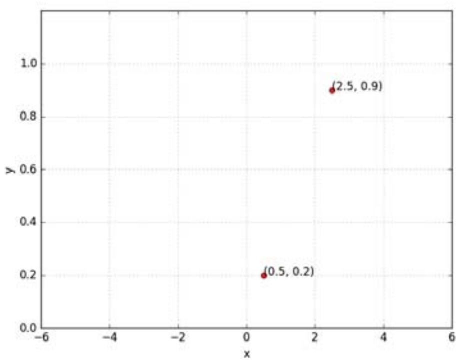
Perhaps it would help to push w and b in the other direction...

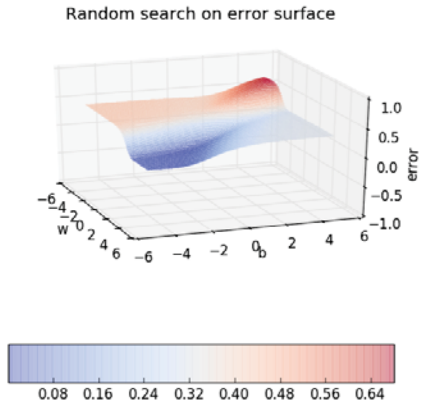
Let us look at something better than our "guess work" algorithm....

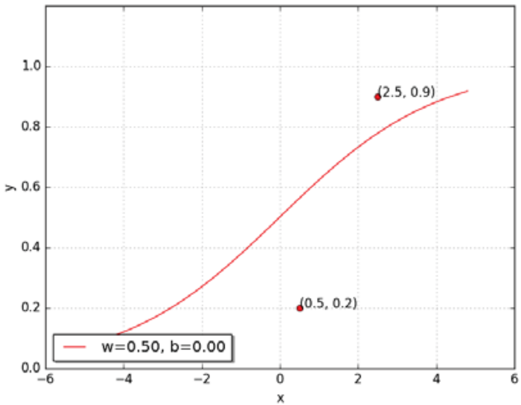


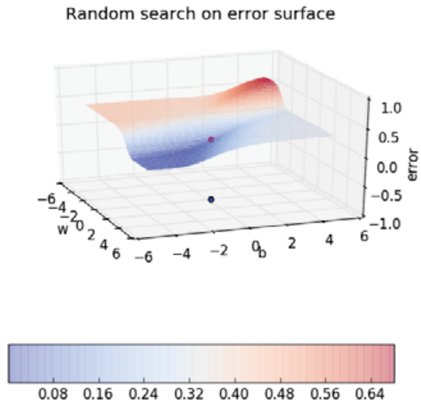
- Since we have only 2 points and 2 parameters $(w,\ b)$ we can easily plot $\mathscr{L}(w,b)$ for different values of $(w,\ b)$ and pick the one where $\mathscr{L}(w,b)$ is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!
- Further, even here we have plotted the error surface only for a small range of $(w,\ b)$ [from $(-6,6)$ and not from $(-\inf,\inf)$]

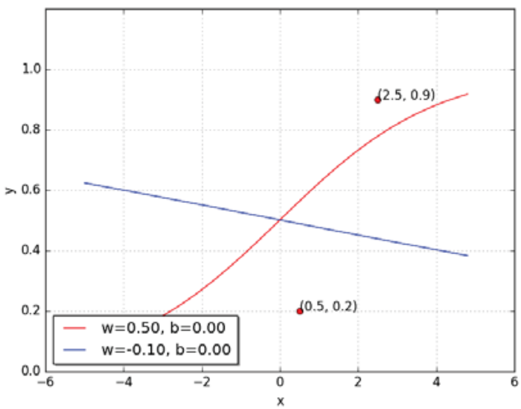
Let us look at the geometric interpretation of our "guess work" algorithm in terms of this error surface

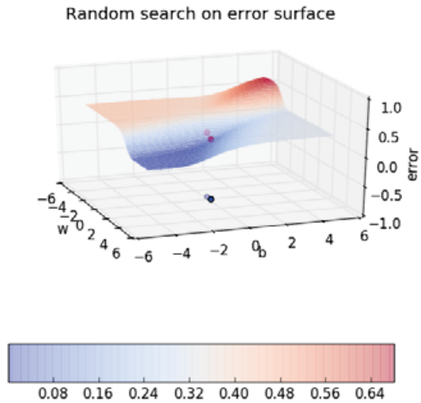

(2.5, 0.9)
(0.5, 0.2)
y
x

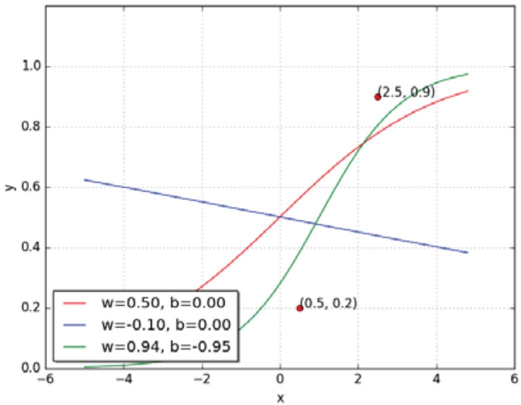

Random search on error surface
error
w
b


(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
x
y

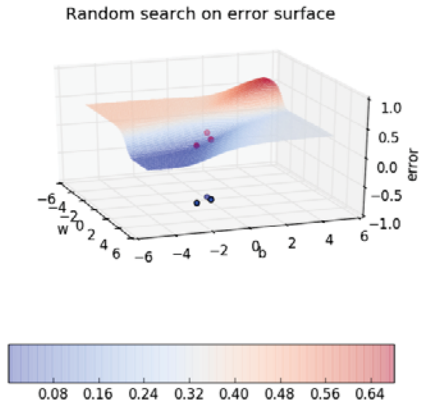

Random search on error surface
error
w
b
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64


(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
x
y

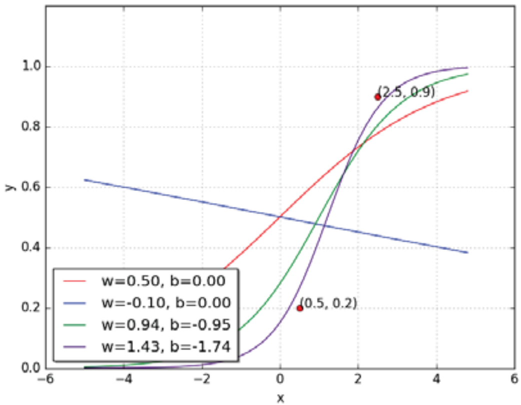

Random search on error surface
w
b
error
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64

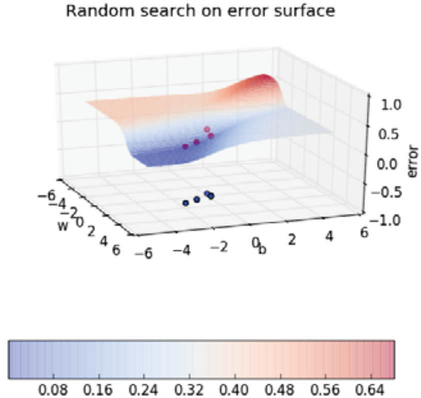


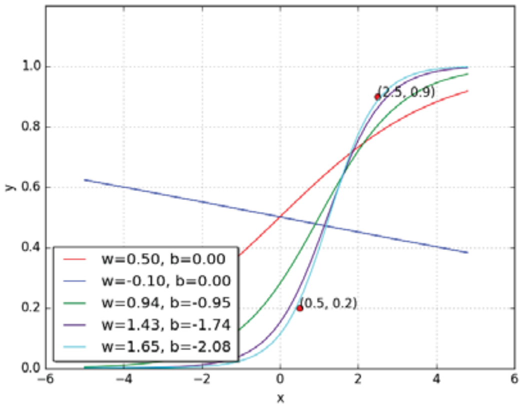
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
x
y



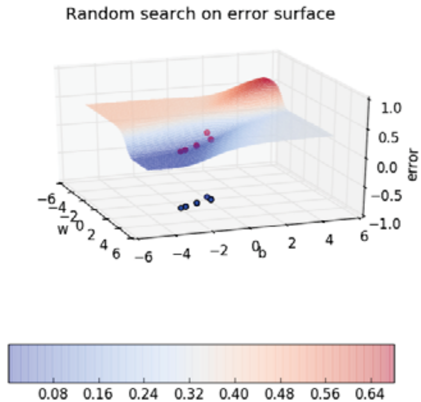
Random search on error surface
error
w
b
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64


w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
(2.5, 0.9)
(0.5, 0.2)
x
y

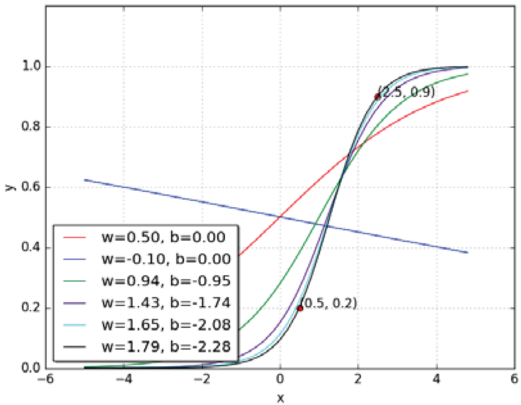

Random search on error surface
w
b
error
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64



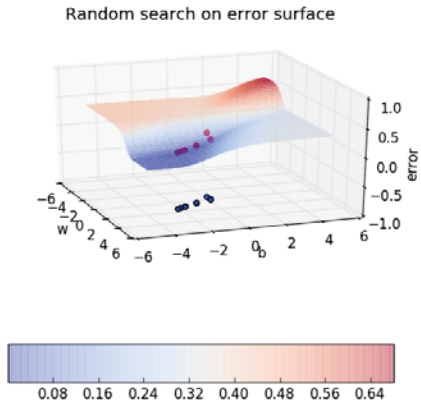
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
x
y



Random search on error surface
w
b
error
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64



(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
w=1.79, b=-2.28
x
y



Random search on error surface
w
b
error
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64

Learning Parameters : Gradient Descent

Now let's see if there is a more efficient and principled way of doing this

Goal

Find a better way of traversing the error surface so that we can reach the minimum value quickly without resorting to brute force search!

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$
$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w = w_t,\, b = b_t}, \nabla b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w = w_t,\, b = b_t}$$

So we now have a more principled way of moving in the w-b plane than our "guess work" algorithm

- Let's create an algorithm from this rule ...

Algorithm 1: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

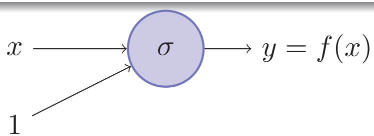
while $t < max\_iterations$ **do**

$\quad w_{t+1} \leftarrow w_t - \eta \nabla w_t$;

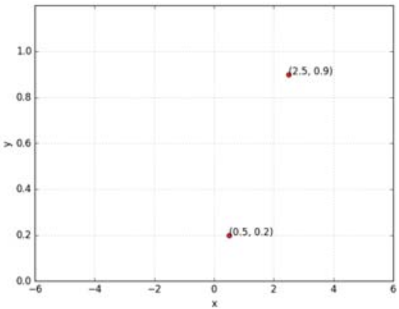
$\quad b_{t+1} \leftarrow b_t - \eta \nabla b_t$;

end

- To see this algorithm in practice let us first derive ∇w and ∇b for our toy neural network



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



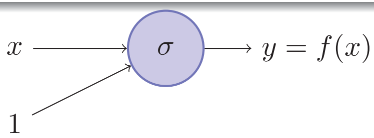
Let's assume there is only 1 point to fit (x, y)

$$\mathscr{L}(w, b) = \frac{1}{2} * (f(x) - y)^2$$

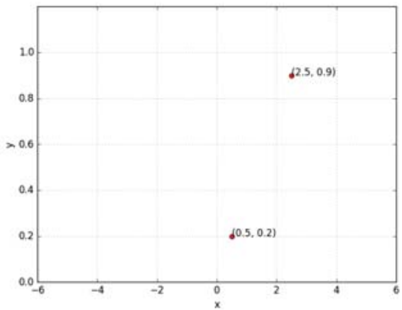
$$\nabla w = \frac{\partial \mathscr{L}(w, b)}{\partial w} = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$

$$
\begin{aligned}
\nabla w &= \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2] \\
&= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)] \\
&= (f(x) - y) * \frac{\partial}{\partial w}(f(x)) \\
&= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\
&= (f(x) - y) * f(x) * (1 - f(x)) * x
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)})) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b))) \\
&= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x) \\
&= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x) \\
&= f(x) * (1 - f(x)) * x
\end{aligned}
$$



$f(x) = \frac{1}{1+e^{-(w \cdot x+b)}}$



So if there is only 1 point (x, y), we have,

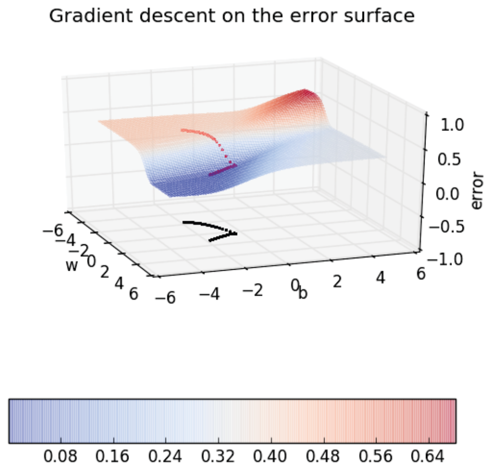
$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

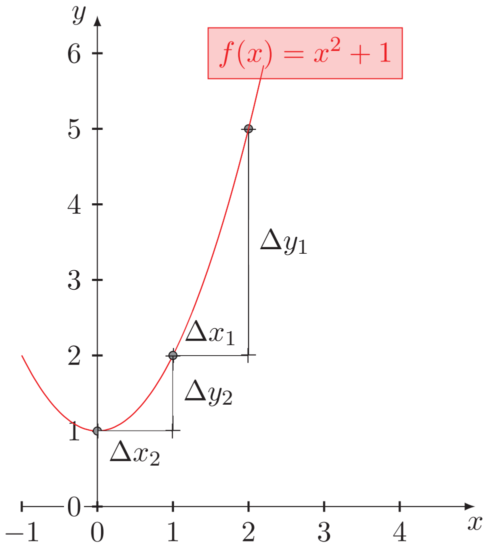
For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

```python
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
```

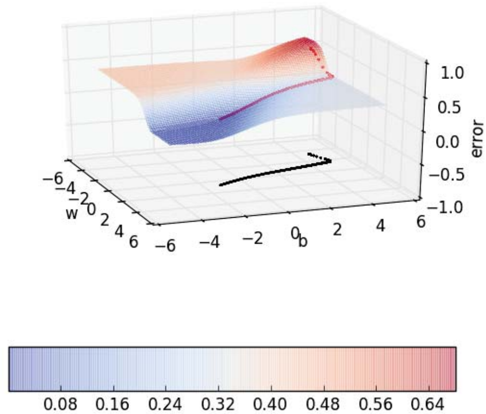




- When the curve is steep the gradient $(\frac{\Delta y_1}{\Delta x_1})$ is large
- When the curve is gentle the gradient $(\frac{\Delta y_2}{\Delta x_2})$ is small
- Recall that our weight updates are proportional to the gradient $w = w - \eta \nabla w$
- Hence in the areas where the curve is gentle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

- *Let's see what happens when we start from a different point*

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



Contours

- *Visualizing things in 3d can sometimes become a bit cumbersome*
- *Can we do a 2d visualization of this traversal along the error surface*
- *Yes, let's take a look at something known as contours*

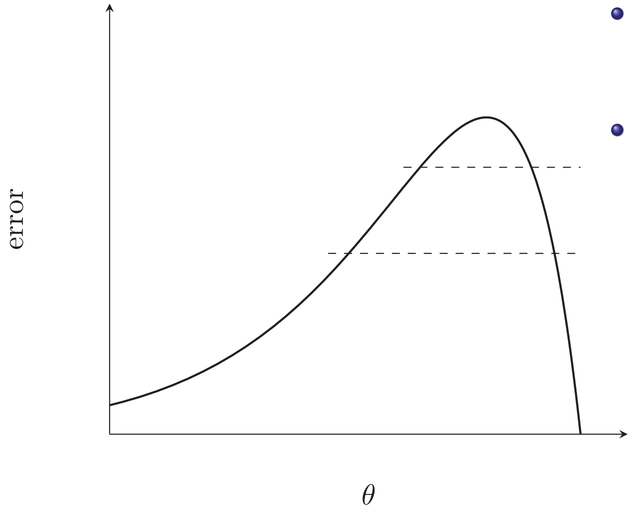
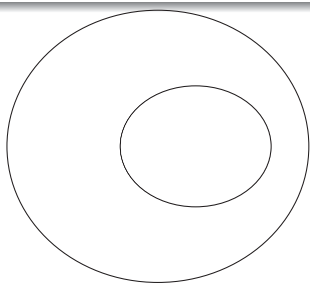


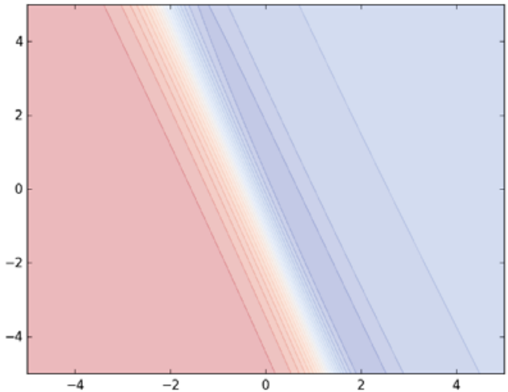
Figure: Front view of a 3d error surface

- Suppose I take horizontal slices of this error surface at regular intervals along the vertical axis
- How would this look from the top-view ?

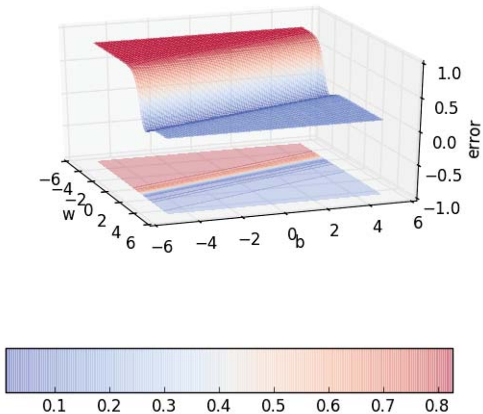


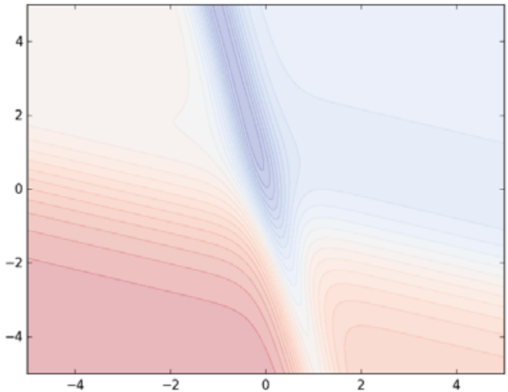
- A small distance between the contours indicates a steep slope along that direction
- A large distance between the contours indicates a gentle slope along that direction

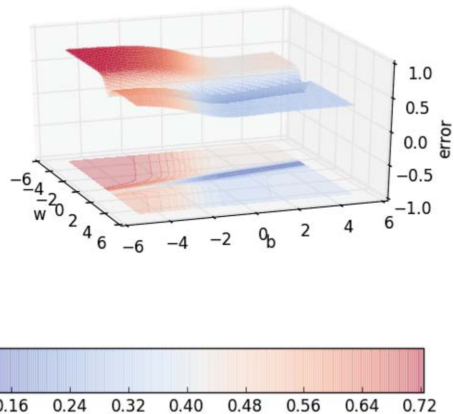
- *Just to ensure that we understand this properly let us do a few exercises ...*



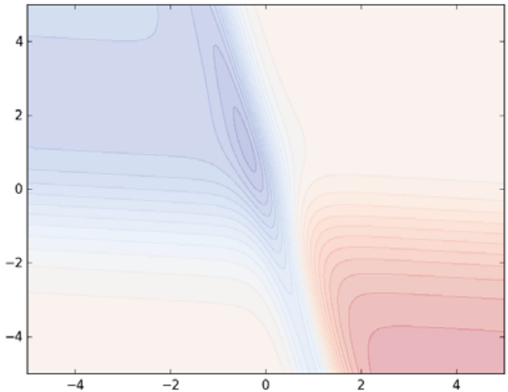
Guess the 3d surface

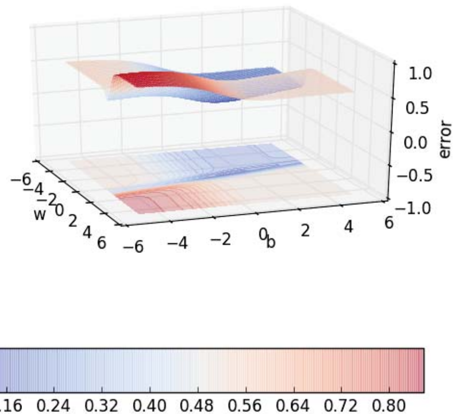






Guess the 3d surface





Guess the 3d surface

Momentum based Gradient Descent

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'

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

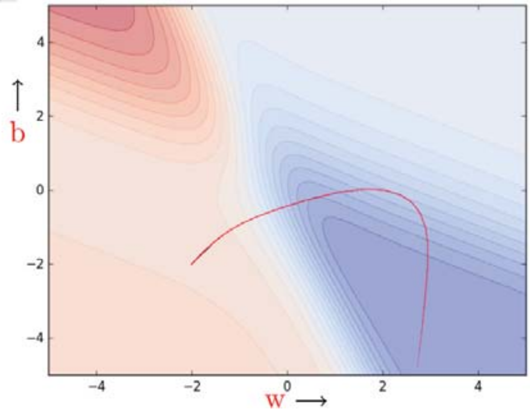
$$v_t = \gamma \cdot v_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - v_t$$

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

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

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= \gamma \cdot v_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
v_2 &= \gamma \cdot v_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
v_3 &= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
v_4 &= \gamma \cdot v_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
v_t &= \gamma \cdot v_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + ... + \eta \nabla w_t
\end{aligned}$$

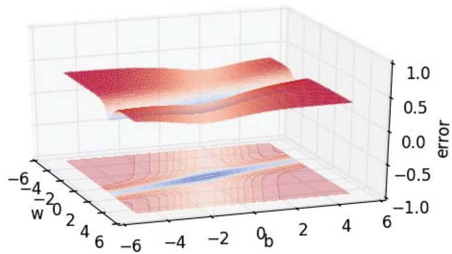
```
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        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
```

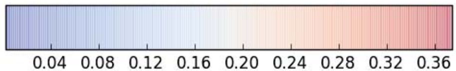


Some observations and questions

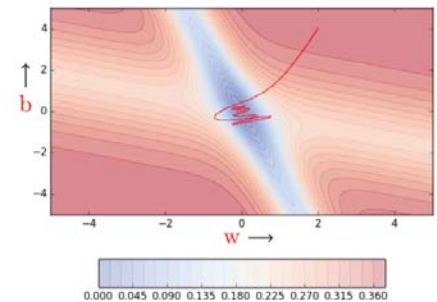
- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along
- Is moving fast always good? Would there be a situation where momentum would cause us to run pass our goal?
- Let us change our input data so that we end up with a different error surface and then see what happens ...

- In this case, the error is high on either side of the minima valley
- Could momentum be detrimental in such cases... let's see....

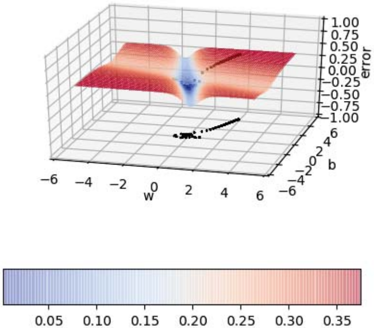




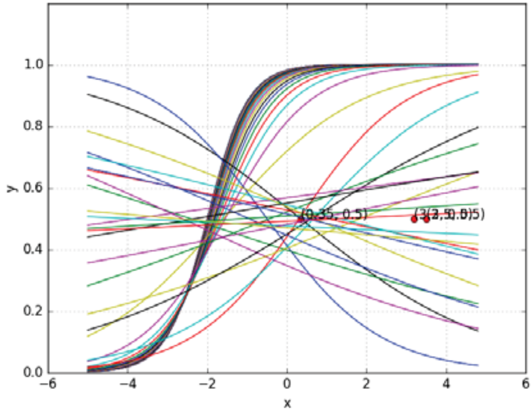
- Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley
- Takes a lot of u-turns before finally converging
- Despite these u-turns it still converges faster than vanilla gradient descent
- After 100 iterations momentum based method has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



Let's look at a 3d visualization and a different geometric perspective of the same thing...



error
w
b
0.05 0.10 0.15 0.20 0.25 0.30 0.35



y
x
(0.35, 0.5)

Nesterov Accelerated Gradient Descent

Question

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

Intuition

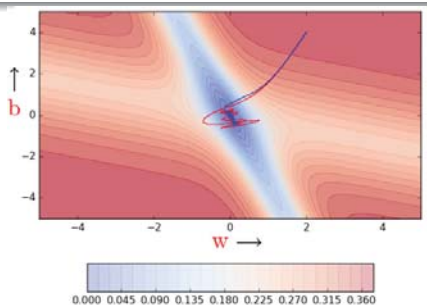
- Look before you leap
- Recall that $v_t = \gamma \cdot v_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot v_{t-1}$ and then a bit more by $\eta \nabla w_t$
- Why not calculate the gradient ($\nabla w_{look\ ahead}$) at this partially updated value of w ($w_{look\ ahead} = w_t - \gamma \cdot v_{t-1}$) instead of calculating it using the current value w_t

Update rule for NAG

$$w_{look\_ahead} = w_t - \gamma \cdot v_{t-1}$$
$$v_t = \gamma \cdot v_{t-1} + \eta \nabla w_{look\ ahead}$$
$$w_{t+1} = w_t - v_t$$

We will have similar update rule for b_t

```python
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

Stochastic And Mini-Batch Gradient Descent

Let's digress a bit and talk about the stochastic version of these algorithms...

```python
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

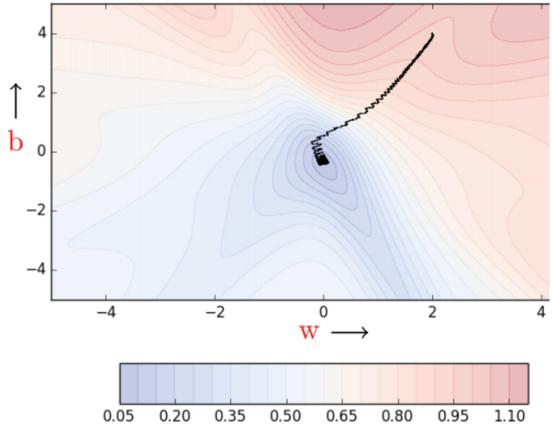
```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- 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).

```python
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 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 (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)
- Indeed we see that there is no guarantee that each local greedy move reduces the global error
- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)



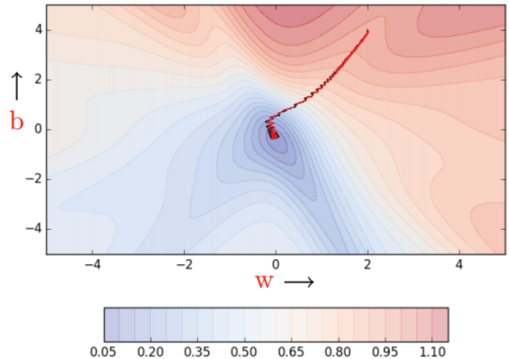
- Yes, let's look at mini-batch gradient descent

```python
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
```

```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(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 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 $k = 2$

- 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	$\frac{N}{B}$

Similarly, we can have stochastic versions of Momentum based gradient descent and Nesterov accelerated based gradient descent