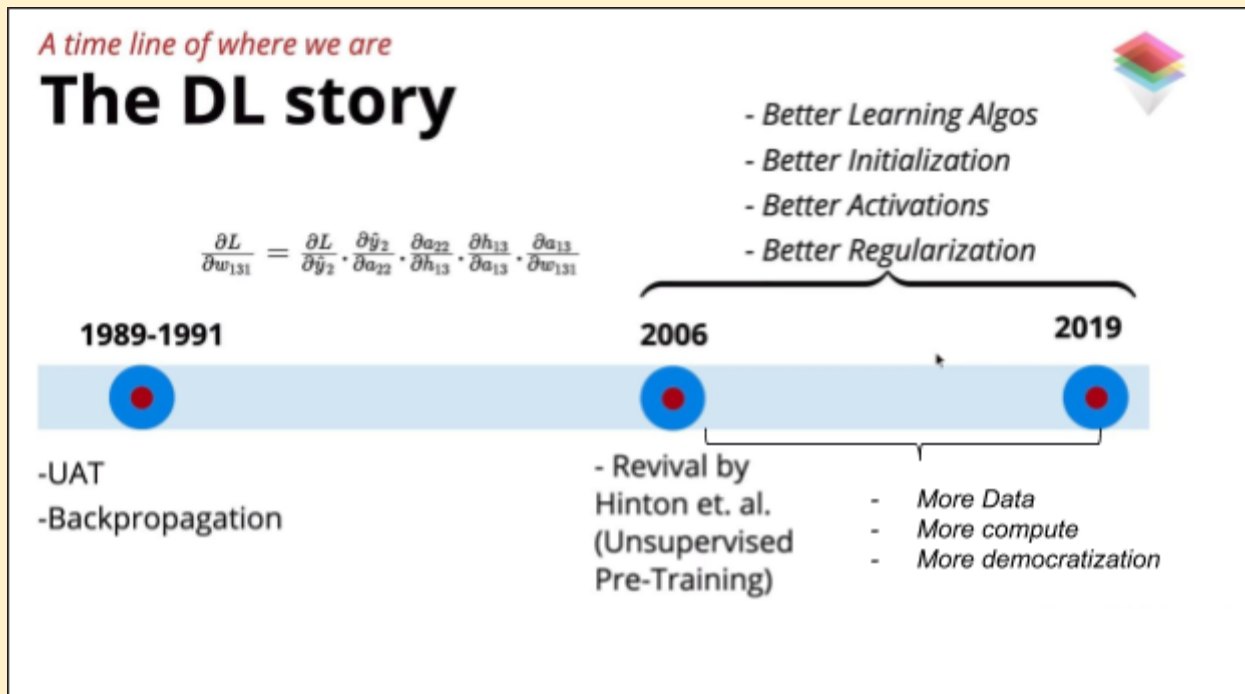


Optimization Algorithms

A quick history of DL to set the context

1. The following illustration shows the progress of Deep Learning over the last 3 decades



2. Some of the salient points in the DL-timeline are as follows
 - a. **1989-1991**
 - i. Universal Approximation Theorem: we will be able to approximate any kind of function with our Neural Network
 - ii. Backpropagation: Derivative calculation happens backwards from the output layer to the input, ie back propagation. It is nothing but Gradient Descent(1847) applied with the chain rule
 - b. **1993-1994**
 - i. A lot of work was done on Recurrent Neural Networks
 - c. **1998**
 - i. LSTMs (Long Short-Term Memory) were proposed
 - ii. Work done on Convolutional Neural Networks
 - d. **2006**
 - i. Revival of DL by Hinton et. al. with the proposal of Unsupervised Pre-training
 - ii. People's interest in DL started increasing.
 - e. **2019**
 - i. Better Learning Algorithms, Initializations, Activation and Regularization
 - ii. More Data, compute and democratization.

Highlighting a limitation of Gradient Descent

Let's look at better learning algorithms

1. The gradient descent update rule is as follows

- $\omega = \omega - \eta \frac{\partial L(\omega)}{\partial \omega}$
- The questions we should be asking are: How do we compute the gradients? What data should we use for computing the gradients?
- To follow up on those questions: How do we use the gradients? Can we come up with a better update rule?
- Here is the Python implementation of Gradient Descent similar to what we have seen earlier

```
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)))

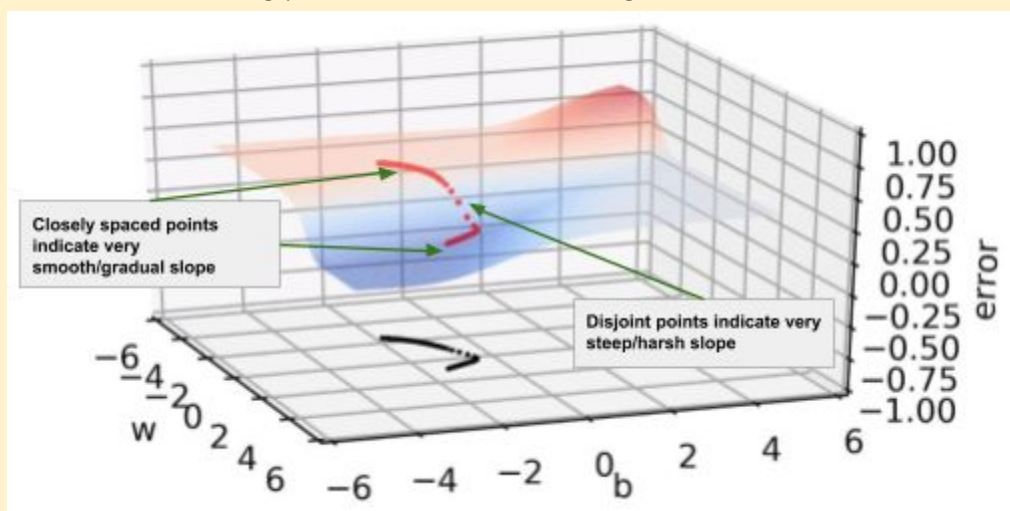
def error(w, b):
    err = 0.0
    for x, y in zip(X, Y):
        fx = f(w, b, x)
        err += 0.5 * (fx - y) ** 2
    return err

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 = -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
```

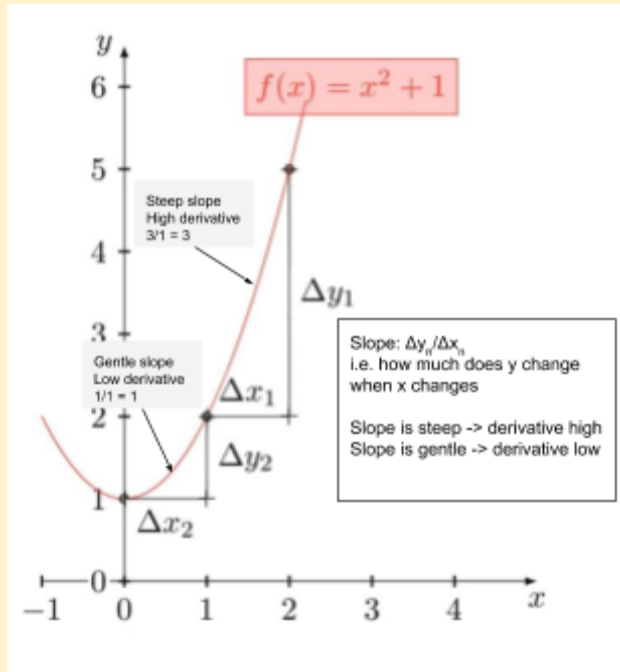
2. Look at the following plot of w, b and error using Gradient descent



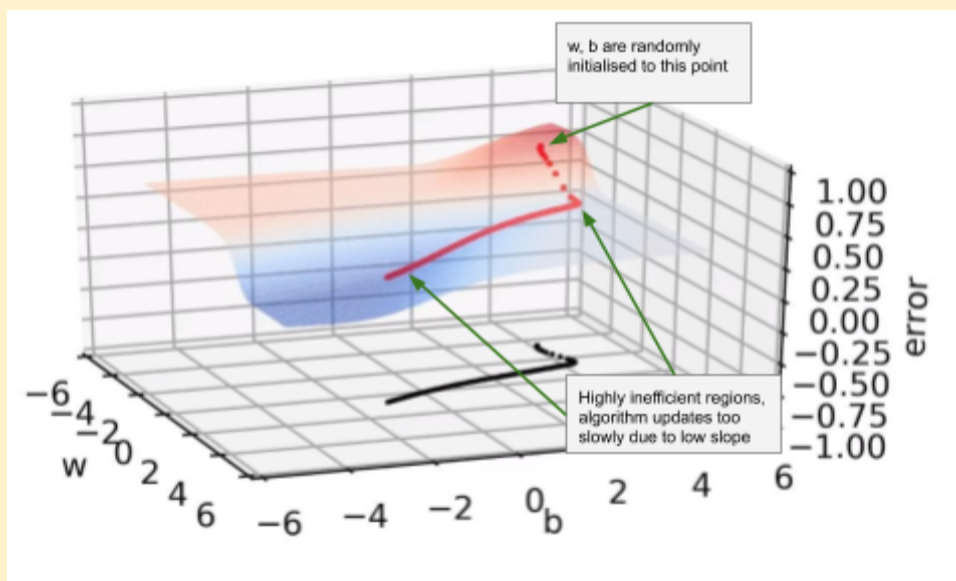
A deeper look into the limitation of Gradient Descent

Why is the behaviour different on different surfaces

1. First, let us look at function to illustrate how the slope behaves



2. So if the derivative is small, the amount by which w or b will be updated by is also small and vice versa if the derivative is large.
3. This could become a problem as in the low-slope/flatter regions, the algorithm does not move fast enough. Here is an example of Gradient descent running inefficiently, moving too slowly.

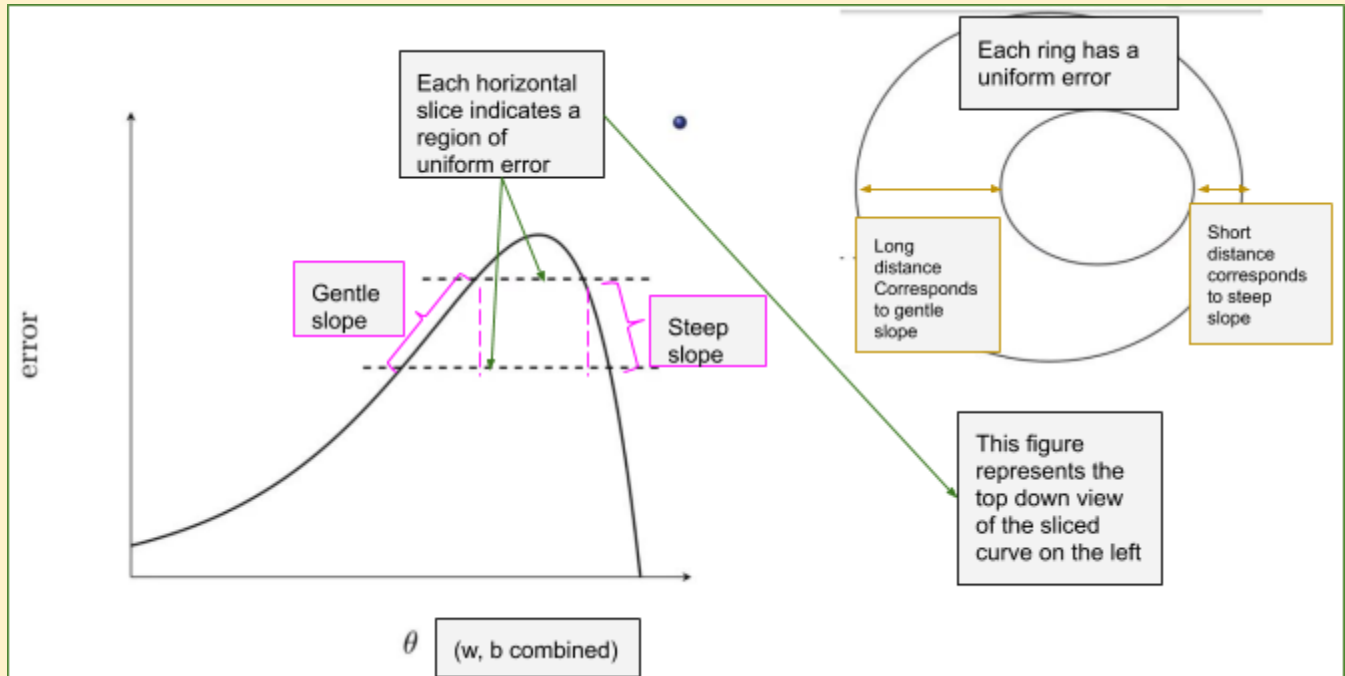


4. If it so happens that our random initialisation of w and b start at a flat region, then the algorithm would need to run many epochs to get out of the plateau. We need to find a solution for dealing with low slope regions.

Introducing contour maps

Can we visualize things in 2D instead of 3D?

1. Look at the following image to understand how contour maps help visualise 3D data in 2D



2. Some interesting points to note
 - a. The rings/contour in the top-down plot each indicate a uniform loss along the contour boundary
 - b. A small distance between contours indicates a steep slope along that direction
 - c. A large distance between contours indicates a gentle slope along that line
3. These are the main points to remember when reading contour plots

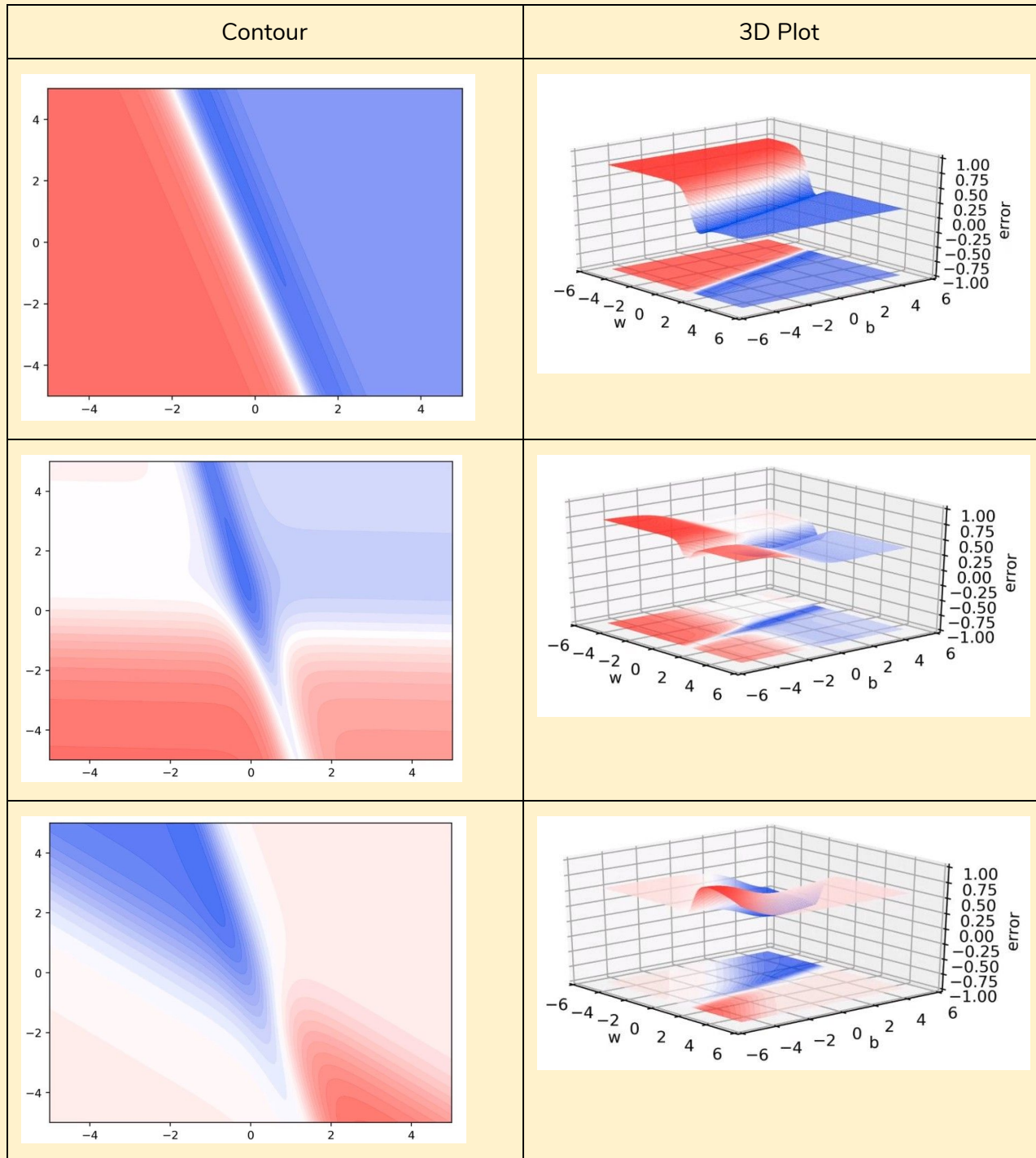
PadhAI: Variants of Gradient Descent

One Fourth Labs

Exercise: Guess the 3D surface

Can we do a few exercises?

1. Look at the following Contour plots and guess their 3D counterparts

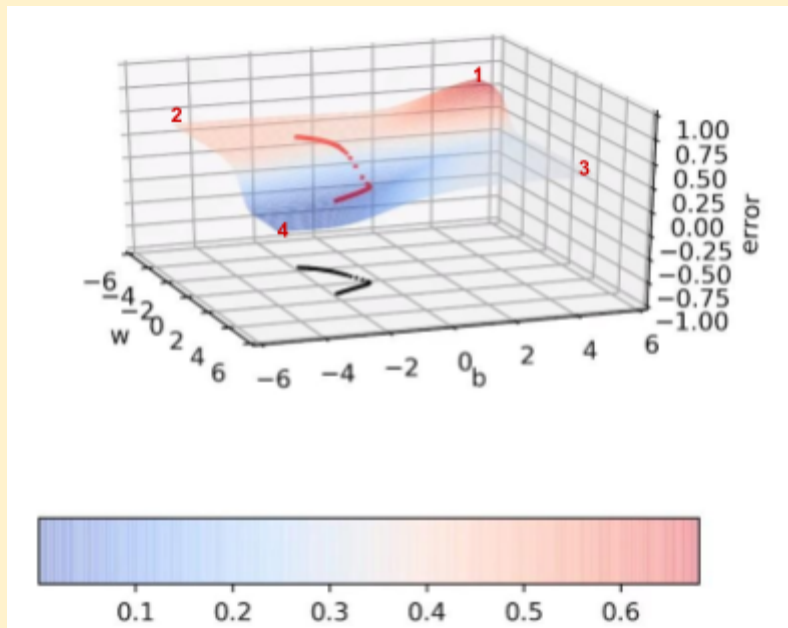


2. Henceforth, we will be showing the gradient descent movement on a 2D contour map.

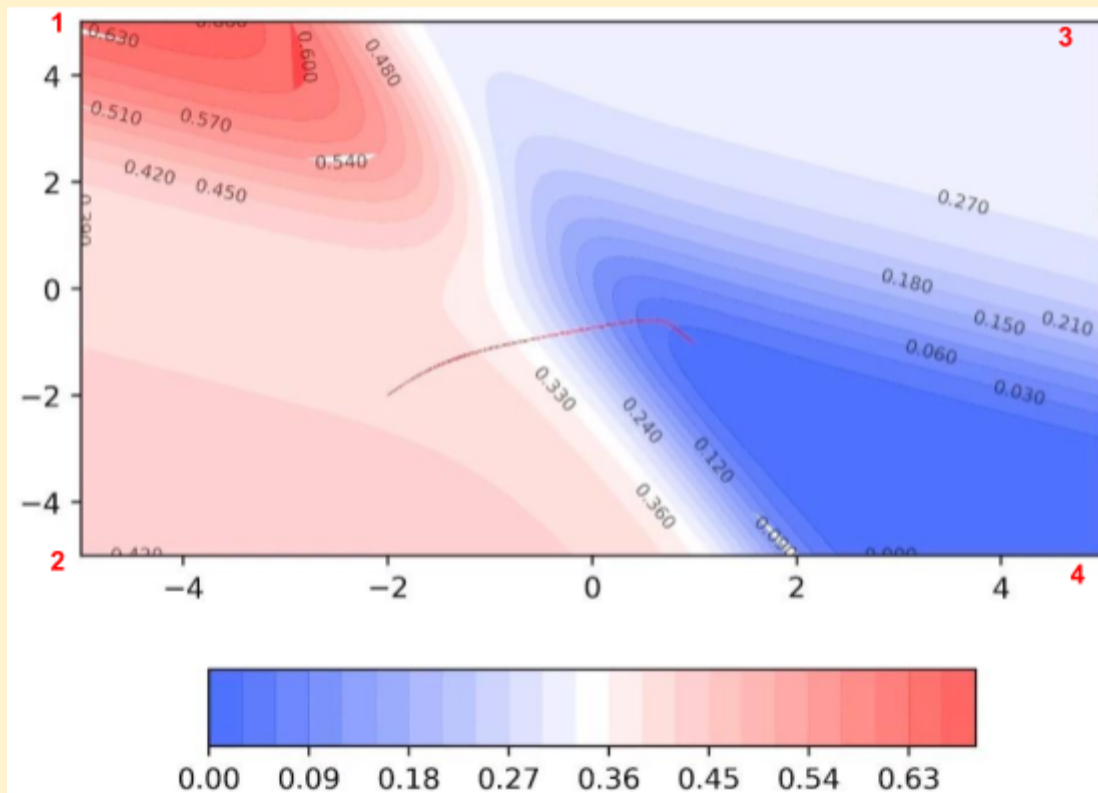
Visualising gradient descent on a 2D contour map

Can we visualise Gradient Descent on a 2D error surface?

1. Now we will visualise Gradient descent on a 3D map and then its corresponding 2D map
2. Here is the 3D plot that we have seen many times before



3. Now, let's look at the same Gradient Descent plotted in a 2D contour map



4. In both of the plots, the GD line eventually reaches the centre of the dark-blue region.

Intuition for momentum based gradient descent

Why do we need a better algorithm?

1. **One of the main issues** with Gradient Descent is that it takes a lot of time to navigate regions with gentle slopes, because the gradient is very small in these regions.
2. **An intuitive solution** would be that if the algorithm is repeatedly being asked to go in the same direction, then it should probably gain some confidence and start taking bigger steps in that direction.
3. Now, we have to convert this intuition into a set of mathematical equations
4. Consider the following equations
5. Gradient Descent Update Rule
 - a. $\omega_{t+1} = \omega_t + \eta \nabla \omega_t$
6. Momentum based Gradient Descent Update Rule
 - a. $v_t = \gamma * v_{t-1} + \eta \nabla \omega_t$
 - b. $\omega_{t+1} = \omega_t - v_t$
 - c. $\omega_{t+1} = \omega_t - \gamma * v_{t-1} - \eta \nabla \omega_t$
 - d. If $\gamma * v_{t-1} = 0$ then it is the same as the regular Gradient Descent update rule
 - e. To put it briefly v_{t-1} is the history of movement in a direction and γ ranges from 0-1

Dissecting the update rule for momentum based gradient descent

Can we dissect the equations in more detail?

1. Let us further dissect the momentum based Gradient Descent
2. $v_t = \gamma * v_{t-1} + \eta \nabla \omega_t$ this variable is called the history.
3. $\omega_{t+1} = \omega_t - v_t$ this variable represents the current movement to be made
4. Consider every instance in time denoted by the subscript, ranging from 0 to t
5. $v_0 = 0$
6. $v_1 = \gamma * v_0 + \eta \nabla \omega_1 = \eta \nabla \omega_1$
7. $v_2 = \gamma * v_1 + \eta \nabla \omega_2 = \gamma \cdot \eta \nabla \omega_1 + \eta \nabla \omega_2$
8. $v_3 = \gamma * v_2 + \eta \nabla \omega_3 = \gamma(\gamma \cdot \eta \nabla \omega_1 + \eta \nabla \omega_2) + \eta \nabla \omega_3$
 - a. $v_3 = \gamma^2 \cdot \eta \nabla \omega_1 + \gamma \cdot \eta \nabla \omega_2 + \eta \nabla \omega_3$
9. $v_4 = \gamma * v_3 + \eta \nabla \omega_4 = \gamma^3 \cdot \eta \nabla \omega_1 + \gamma^2 \cdot \eta \nabla \omega_2 + \gamma^1 \cdot \eta \nabla \omega_3 + \eta \nabla \omega_4$

.

.

.
10. $v_t = \gamma * v_{t-1} + \eta \nabla \omega_t = \gamma^{t-1} \cdot \eta \nabla \omega_1 + \gamma^{t-2} \cdot \eta \nabla \omega_2 + \dots + \eta \nabla \omega_t$
11. Here, we take an Exponentially Decaying Weighted Sum, whereby as we move further and further into the series, the weight decays more.
12. The intuition behind this is as we progress further and further down a series/direction, we can place lesser and lesser importance to the later gradients as we move along the same direction.

PadhAI: Variants of Gradient Descent

One Fourth Labs

Running and visualising momentum based gradient descent

Let's look at the Python implementation of Momentum based Gradient Descent

- Here is the Python code for Momentum Based Gradient Descent

```
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

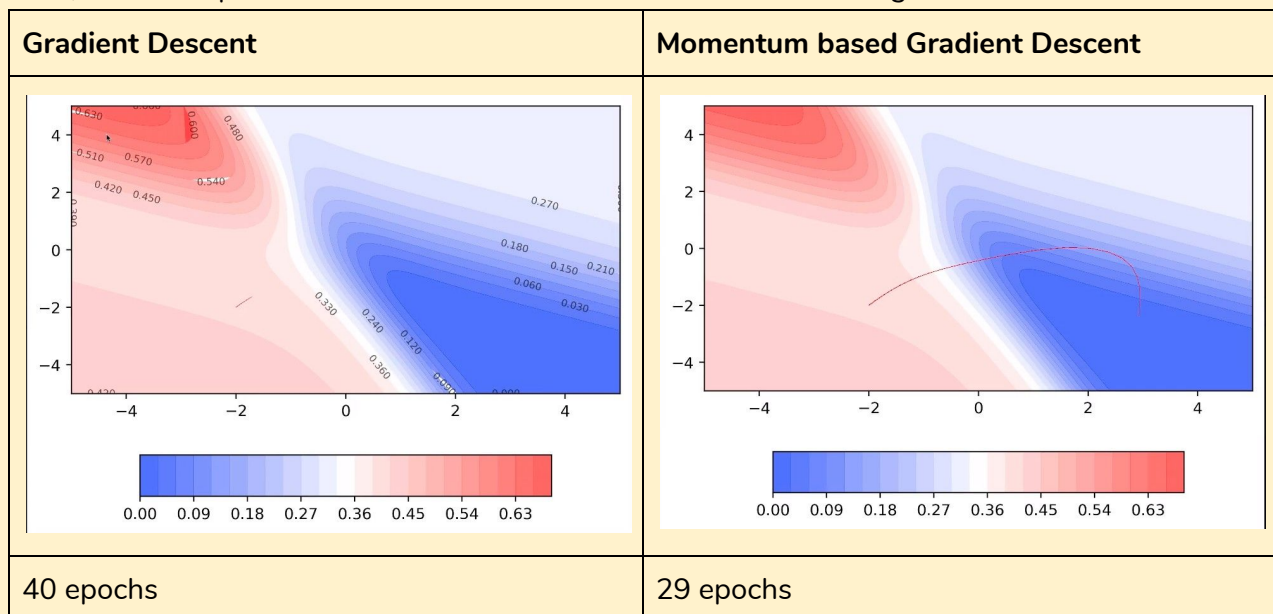
def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

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

def do_momentum_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    gamma = 0.7
    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
```

- Now, let us compare the movement of Momentum based GD and regular GD



- However, there are still some issues with Momentum based GD that we will address in the next section

PadhAI: Variants of Gradient Descent

One Fourth Labs

A disadvantage of momentum based gradient descent

Let us make a few observations and ask some questions.

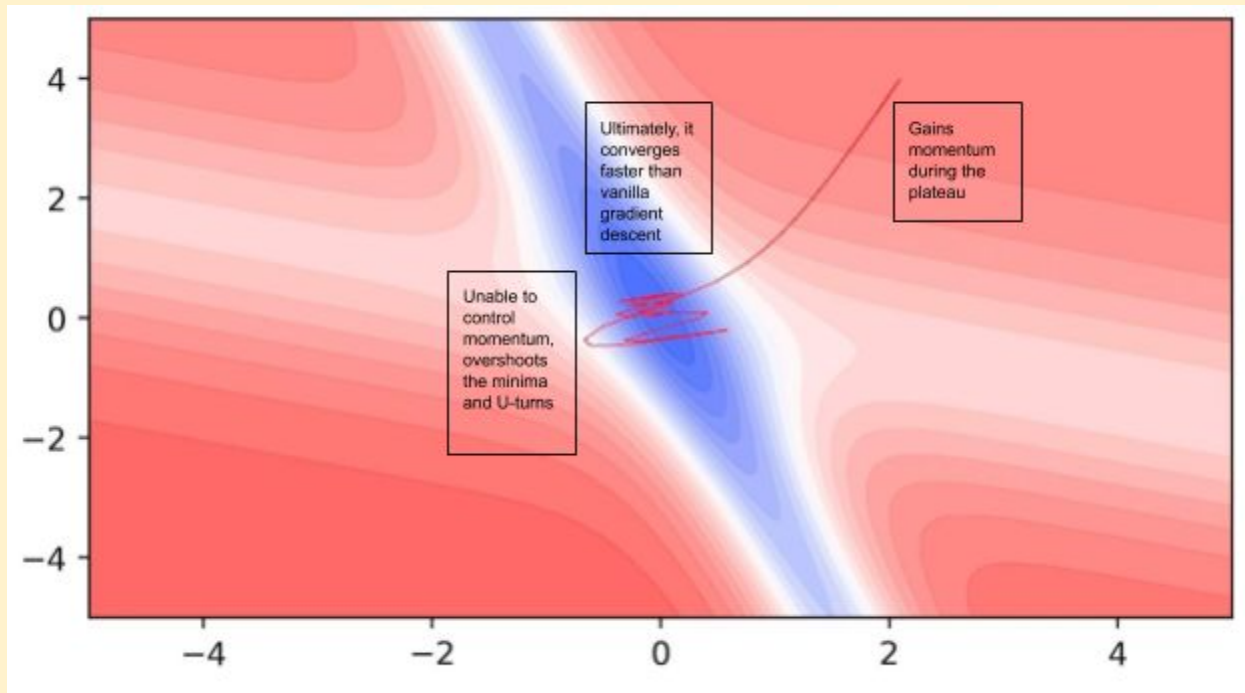
1. Observations

- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along

2. Questions

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

3. Let us look at an implementation of Momentum based GD



4. A few points to note

- Momentum based gradient descent oscillates in and out of the minima valley (u-turns)
- Despite these u-turns it still converges faster than vanilla gradient descent

5. Now, we will look at reducing the oscillations in Momentum based GD

Intuition behind nesterov accelerated gradient descent

Can we do something to reduce the oscillation in Momentum based GD

1. Let us consider the Momentum based Gradient Descent Update Rule
 - a. $v_t = \gamma * v_{t-1} + \eta \nabla \omega_t$
 - b. $\omega_{t+1} = \omega_t - v_t$
 - c. $\omega_{t+1} = \omega_t - \gamma * v_{t-1} - \eta \nabla \omega_t$
 - d. Here, we can see that the movement occurs in two steps
 - i. The first is with the history-term $\gamma * v_{t-1}$
 - ii. The second is with the weight term $\eta \nabla \omega_t$
 - iii. When moving both steps each time, it is possible to overshoot the minima between the two steps
 - iv. So we can consider first moving with the history term, then calculate the second step from where we were located after the first step (ω_{temp}).
2. Using the above intuition, the Nesterov Accelerated Gradient Descent solves the problem of overshooting and multiple oscillations
 - a. $\omega_{temp} = \omega_t - \gamma * v_{t-1}$ compute ω_{temp} based on movement with history
 - b. $\omega_{t+1} = \omega_{temp} - \eta \nabla \omega_{temp}$ move further in the direction of the derivative of ω_{temp}
 - c. $v_t = \gamma * v_{t-1} + \eta \nabla \omega_{temp}$ update history with movement due to derivative of ω_{temp}

PadhAI: Variants of Gradient Descent

One Fourth Labs

Running and visualising nesterov accelerated gradient descent

Let's execute the code for this

1. Here is the Python code for NAG, it is an improvement on the MGD

```
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_nag_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    gamma = 0.7
    for i in range(max_epochs):
        dw, db = 0, 0

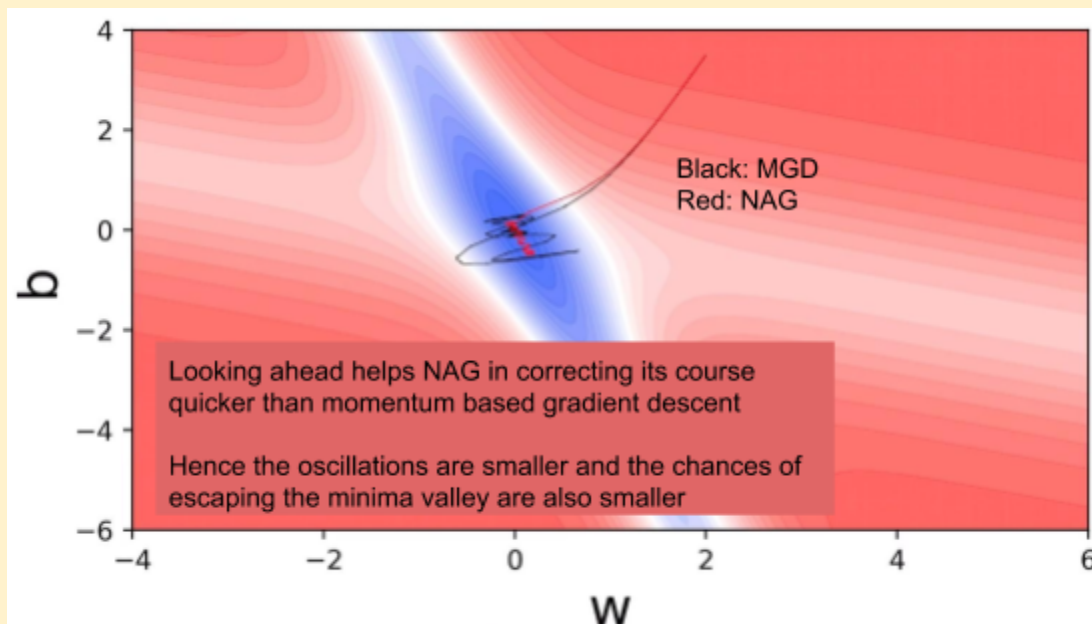
        #Compute the lookahead value
        w = w - gamma*v_w # this is w_temp
        b = b - gamma*v_b # this is b_temp

        for x, y in zip(X, Y):
            #Compute the derivatives using the lookahead value
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        #Now move further in the direction of that gradient
        w = w - eta*dw
        b = b - eta*db

        v_w = gamma * v_w + eta * dw
        v_b = gamma * v_b + eta * db
```

2. Here we have a comparison between NAG and MGD



Summary and what next

What have we learned this chapter

1. We have seen two new update rules, namely Momentum based gradient descent and Nesterov Accelerated Gradient Descent
2. These each mitigate some of the shortcomings of vanilla gradient descent
3. MGD allows for faster movement at plateau regions, thereby saving a lot of time/epochs
4. However, MGD can be a bit wasteful as it approaches the minima valley and oscillates till it stops
5. This flaw was remedied using NAG, whereby the oscillations near the minima valley are drastically reduced
6. NAG offers a good improvement to MGD

PadhAI: Variants of Gradient Descent

One Fourth Labs

The idea of stochastic and mini-batch gradient descent

How many updates are we making?

1. Let us consider vanilla gradient descent

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

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 # Updates to w are made only after all data-points are covered
        b = b - eta * db # Updates to b are made only after all data-points are covered
```

2. From the above image, we can see that we make one update(w,b) for one pass/epoch over the data
3. It can be exemplified as follows
 - a. Consider a training set with 1 million data points
 - b. With Gradient Descent, we calculate the derivatives for each of these points
 - c. Once we're done, we update the parameters
 - d. Thus, we pass over all 1 million points to make a single update to w & b
 - e. It can also be called **batch gradient descent**, as the entire dataset is used as a single batch
4. However, we can choose to make an approximation based on looking at a smaller portion(batch) of the data points instead of analysing the whole dataset each time.
5. This is called **mini-batch gradient descent** and can be described as follows
 - a. Consider a training set of 1 million data points
 - b. Select a batch size of 100 data points
 - c. What this means is that every batch, the algorithm calculates all of the 100 derivatives and updates the parameters
 - d. Thus, passing over all 1 million data points results in 10000 updates to w & b.
6. **Stochastic gradient descent** is when the batch size is 1, i.e. an update to the parameters after each single data point
7. One key thing to note is that both stochastic and mini-batch gradient descent are approximations of the true derivative obtained by batch gradient descent.
8. However it is advantageous as it allows is to make updates faster and achieve quicker progress.

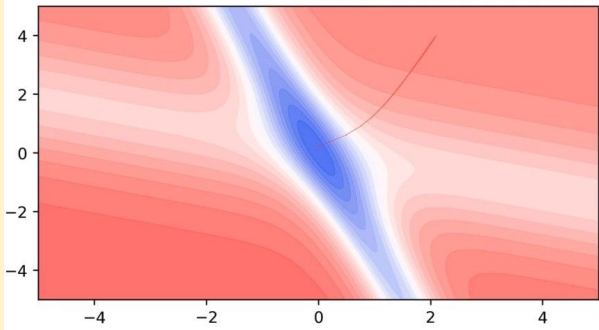
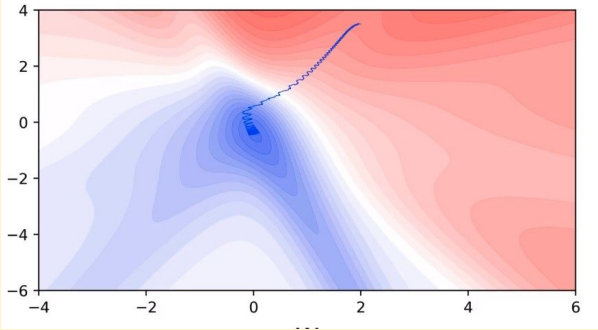
PadhAI: Variants of Gradient Descent

One Fourth Labs

Running stochastic gradient descent

Can we make stochastic updates

1. Let's do a side by side comparison of batch GD and stochastic GD

Batch GD	Stochastic GD
<pre>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</pre>	<pre>def do_stochastic_gradient_descent(): w, b, eta, max_epochs = -2, -2, 1.0, 1000 for i in range(max_epochs): dw, db = 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</pre>
	

2. Some of the advantages of Stochastic GD are
 - a. Quicker updates
 - b. Many updates in one pass of the data
3. Some of the disadvantages of Stochastic GD are
 - a. Approximate(stochastic) gradient
 - b. Almost like tossing a coin once and computing $P(\text{heads})$
4. From the Gradient descent visualization, we can see that it oscillates during movement. However, this oscillation is different from Momentum GD or NAG.
5. In stochastic GD, the oscillations are due to redirection after every point, as every point behaves as an individual greedy entity influencing w & b , thus leading to fluctuations right from the start.
6. In MGD or NAG, the oscillations appear the value approaches the minima as a result of overshooting the intended destination.

PadhAI: Variants of Gradient Descent

One Fourth Labs

Running mini-batch gradient descent

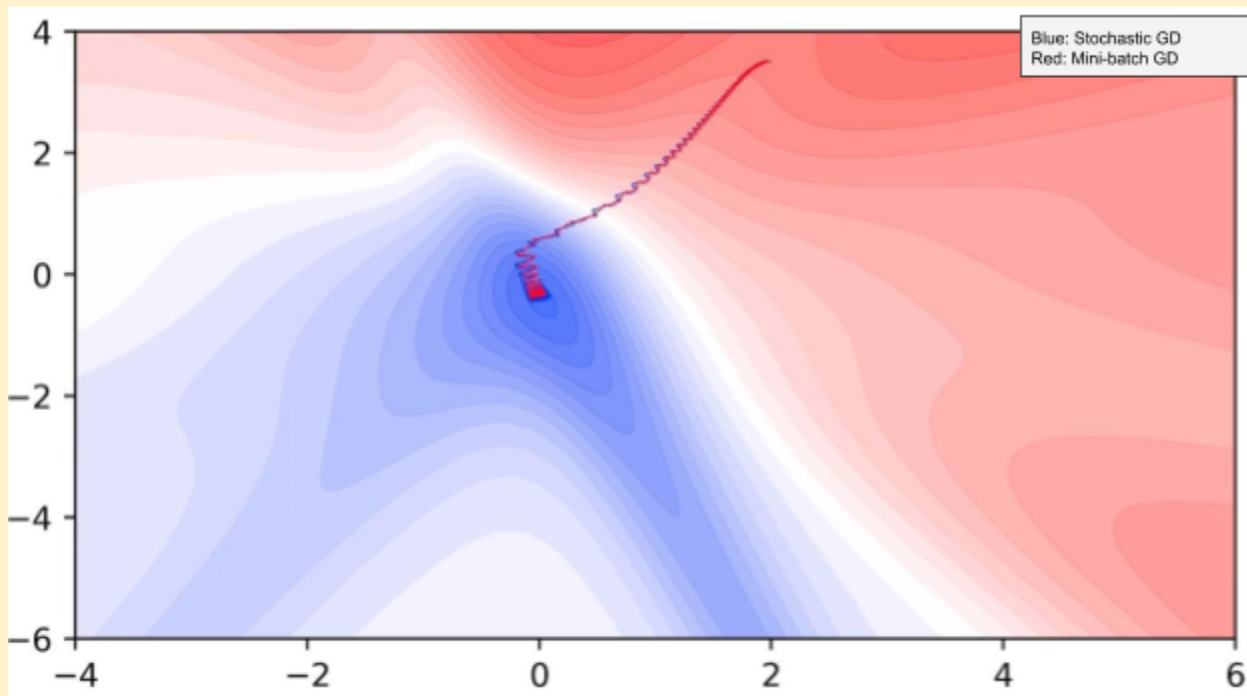
Doesn't it make sense to use more than one point or a mini-batch of points?

1. Let's look at the python implementation of mini-batch GD

```
def do_mini_batch_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    mini_batch_size = 10
    num_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_points_seen += 1

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

2. Now let us look at the 2D visualisation of mini-batch superimposed over stochastic GD



3. Here, we can observe that even though the plot oscillates for mini-batch GD, it is still considerably less than with stochastic GD, evidenced by the red plot lying entirely within the blue plot.
4. As we increase the batch size, the stability of the curve also improves, resulting in better estimates of the gradient
5. Recommended batch size is 32, 64, 128 etc.
6. The higher the batch size (k), the more accurate the estimates.

PadhAI: Variants of Gradient Descent

One Fourth Labs

Epochs and Steps

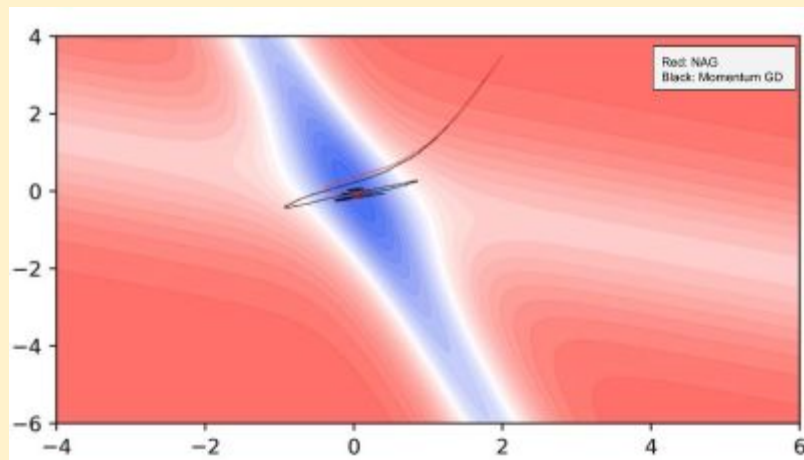
What is an epoch and what is a step?

- Let us go over the definitions of an epoch and a step
 - 1 epoch = one pass over the entire data
 - 1 step = one update of the parameters
 - N = number of data points
 - B = mini-batch size
- Let's analyse the algorithms using epochs and steps

Algorithm	Number of steps in one epoch
Batch Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	N/B

- Let's look at stochastic version of NAG and Momentum based GD

Stochastic Momentum GD	Stochastic NAG
<pre>def do_stochastic_momentum_gradient_descent(): w, b, eta, max_epochs = -2, -2, 1.0, 1000 v_w, v_b = 0.0, 0.0 gamma = 0.7 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</pre>	<pre>def do_stochastic_nag_gradient_descent(): w, b, eta, max_epochs = -2, -2, 1.0, 1000 v_w, v_b = 0, 0 gamma = 0.9 for i in range(max_epochs): dw, db = 0, 0 #Compute the lookahead value w = w - gamma*v_w b = b - gamma*v_b for x, y in zip(X, Y): #Compute the derivatives using the lookahead value dw += grad_w(w, b, x, y) db += grad_b(w, b, x, y) #Now move further in the direction of that gradient w = w - eta*dw b = b - eta*db #Now update the history v_w = gamma * v_w + eta * dw v_b = gamma * v_b + eta * db</pre>

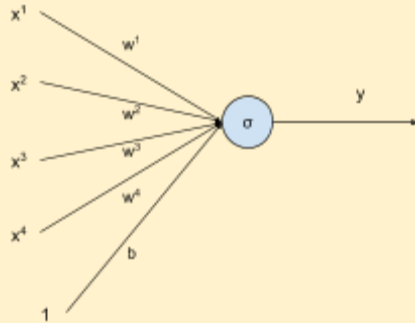


- Since there is a history component, NAG and Momentum GD have slightly smoother oscillations.

Why do we need an adaptive learning rate?

Why do we need an adaptive learning rate for every feature?

1. Consider input data with 4 features being processed through a sigmoid neuron



2. Here $y = f(x) = \frac{1}{1 + e^{-(w \cdot x + b)}}$
 - a. $x = \{x^1, x^2, x^3, x^4\}$
 - b. $w = \{w^1, w^2, w^3, w^4\}$
3. From our gradient formula, we know that the value of the input feature plays a role in the gradient calculation i.e. $\nabla w^n = (f(x) - y) * f(x) * (1 - f(x)) * x^n$
4. In real world scenarios, many features in the data are **sparse**, i.e. they take on a 0 value for most of the training inputs. Therefore, the derivatives corresponding to these 0 valued points are also 0, and the weight update is going to be 0.
5. To aid these sparse features, a larger learning rate can be applied to the **non-zero valued points** of these sparse features.
6. Example:
 - a. Consider a subject at college that you are taught for only 5 minutes a day
 - b. For those 5 minutes, maximising your attention span would allow for maximum knowledge retention
 - c. In this case, the 5-minute subject would be a sparse feature i.e. a feature that does not occur very often in the training data
 - d. And the attention span would be our learning rate. A high learning rate for the sparse features allows us to maximise the learning (weight updation) we get from it.
7. Conversely, **dense** features are those with non-zero values for most of the data points. They must be dealt with by using a lower learning rate.
8. Can we have a different learning rate for each parameter(weights) which takes care of the frequency(sparsity/density) of features?

Introducing Adagrad

How do we convert the adaptive learning rate intuition into an equation?

1. **Intuition:** Decay the learning rate for parameters in proportion to their update history (fewer updates, lesser decay)
2. The Adagrad (Adaptive Gradient) is an algorithm which satisfies the above intuition
3. Adagrad
 - a. $v_t = v_{t-1} + (\nabla \omega_t)^2$
 - i. This value increments based on the gradient of that particular iteration, i.e. the value of the feature is non-zero.
 - ii. In the case of dense features, it increments for most iterations, resulting in a larger v_t value
 - iii. For sparse features, does not increment much as the gradient value is often 0, leading to a lower v_t value.
 - b. $\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla \omega_t$
 - i. The denominator term $\sqrt{(v_t)}$ serves to regulate the learning rate η
 - ii. For dense features, v_t is larger, $\sqrt{(v_t)}$ becomes larger thereby lowering η
 - iii. For sparse features, v_t is smaller, $\sqrt{(v_t)}$ becomes smaller and lowers η to a smaller extent.
 - iv. The ϵ term is added to the denominator $\sqrt{(v_t)} + \epsilon$ to **prevent a divide-by-zero error** from occurring in the case of very sparse features i.e. where all the data points yield zero up till the measured instance.

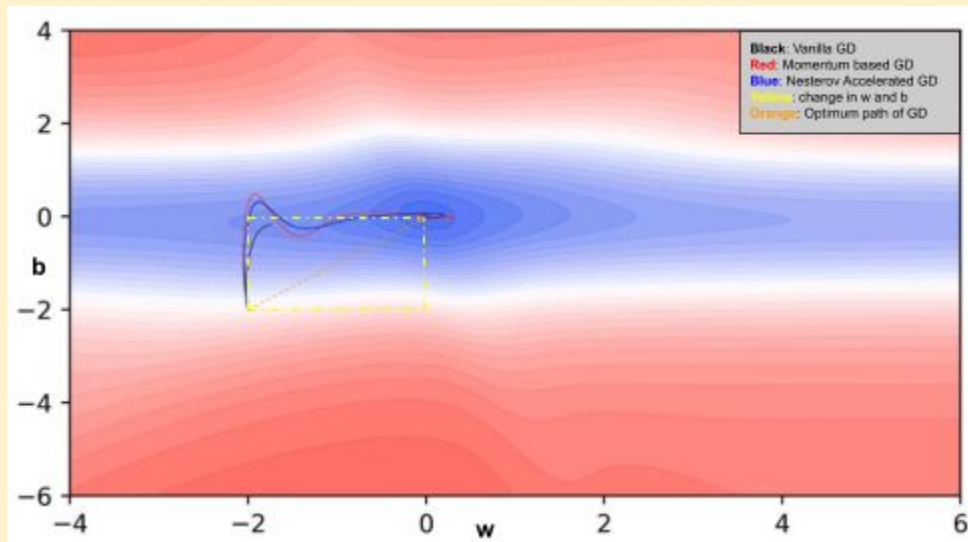
PadhAI: Variants of Gradient Descent

One Fourth Labs

Running and Visualizing Adagrad

Let's compare this to vanilla, momentum based, NAG gradient descent

1. Let's plot the 2D visualisation of vanilla, momentum based, NAG gradient descent



2. Here, w & b behave as two features of the input (x_0, x_1). b is a dense feature and is always a non-zero value. w is deliberately chosen as a sparse feature with 80% of the values as 0.
3. Thus, we would need a higher learning rate for w and a lower learning rate for b, if not, we will end up with sub-optimal paths as shown by the previous 3 types of GD from the figure.
4. Let's look at a visualisation of Adagrad

