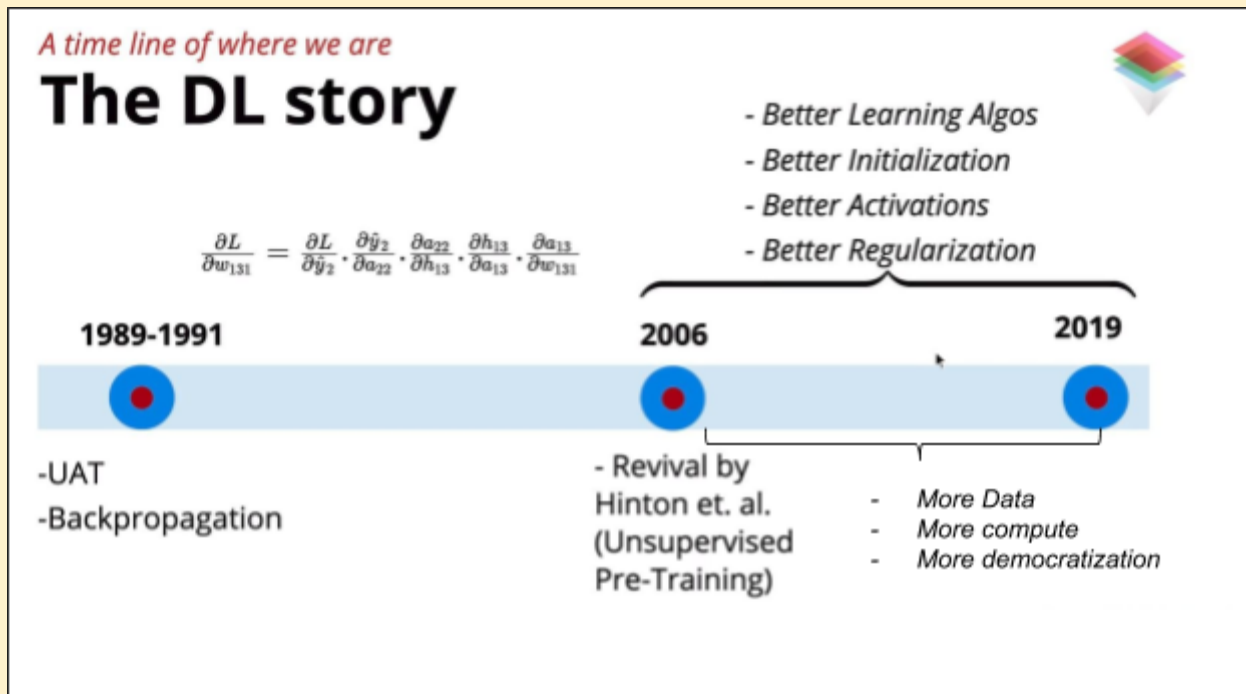


## Optimization Algorithms

### 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)))

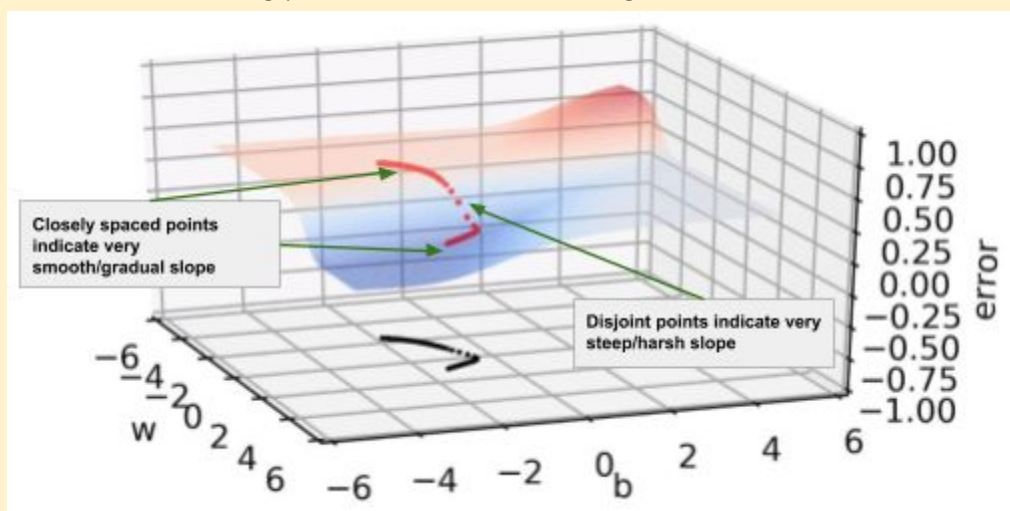
def error(w, b):
    err = 0.0
    for x, y in zip(X, Y):
        fx = f(w, b, x)
        err += 0.5 * (fx - y) ** 2
    return err

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



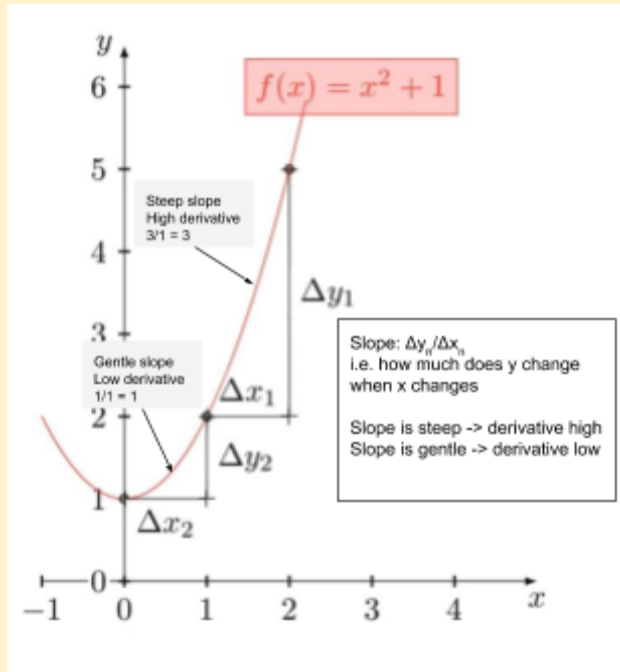
# PadhAI: Variants of Gradient Descent

## One Fourth Labs

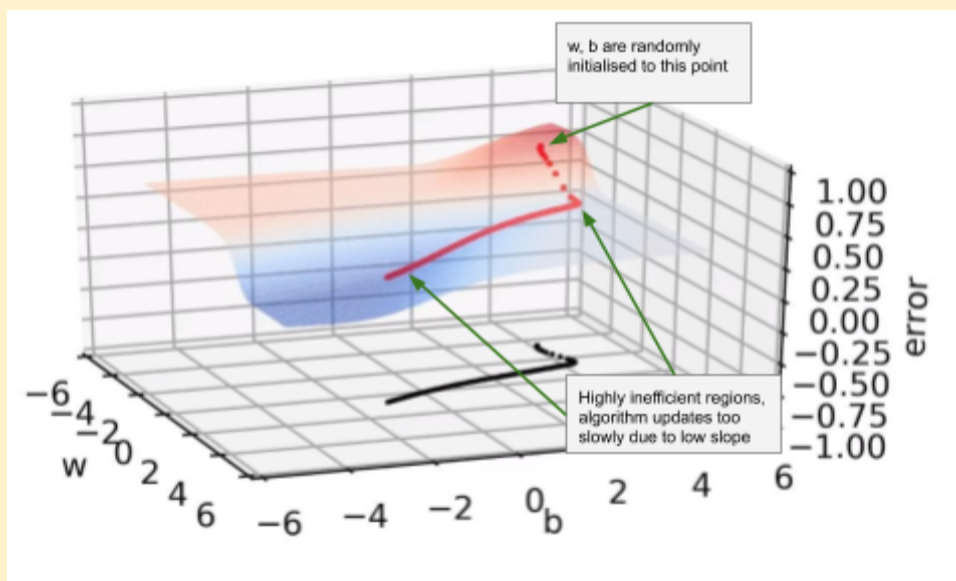
### A deeper look into the limitation of Gradient Descent

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.

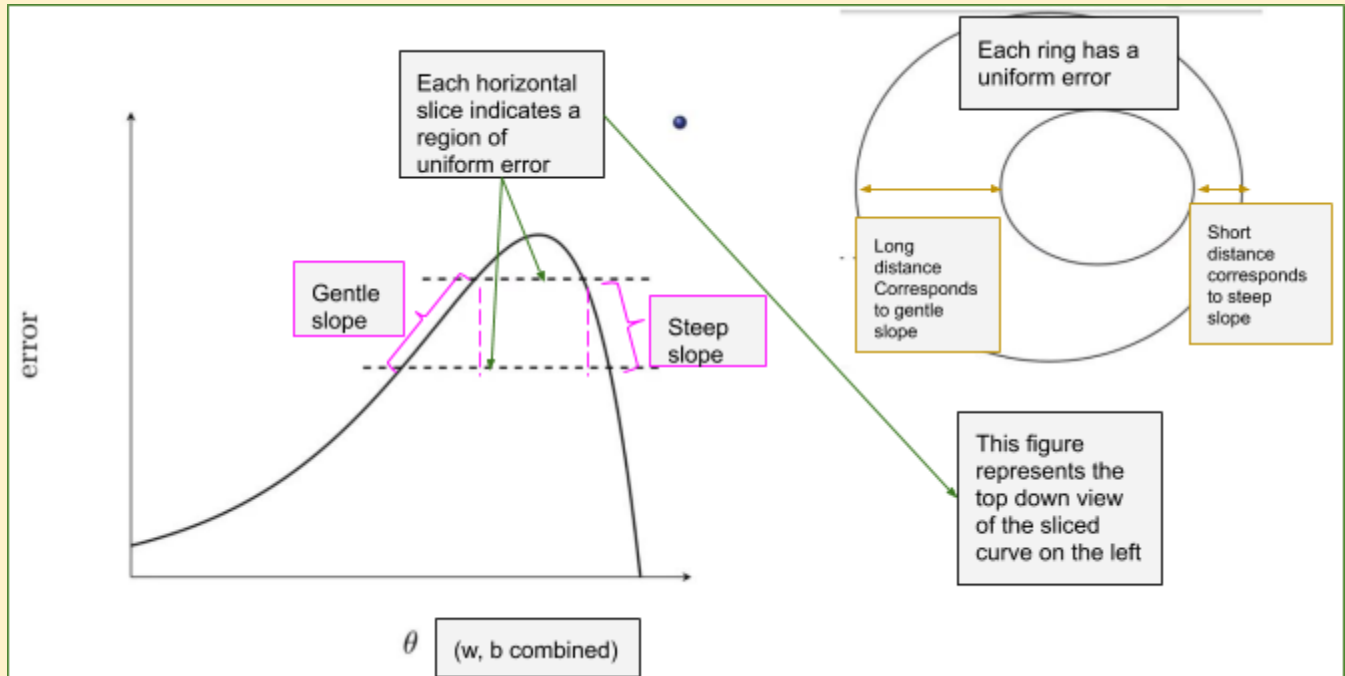


4. If it so happens that our random initialisation of  $w$  and  $b$  start at a flat region, then the algorithm would need to run many epochs to get out of the plateau. We need to find a solution for dealing with low slope regions.

### 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/conour 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

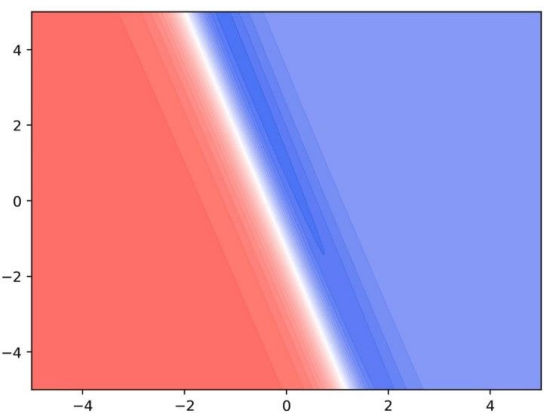
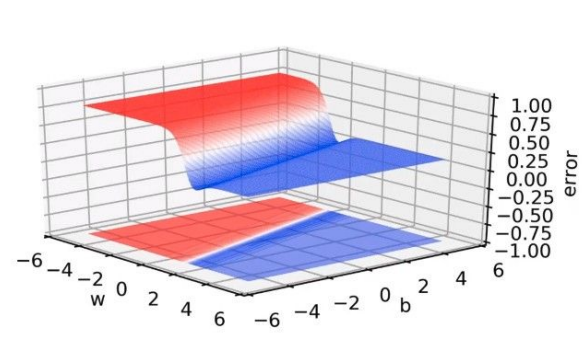
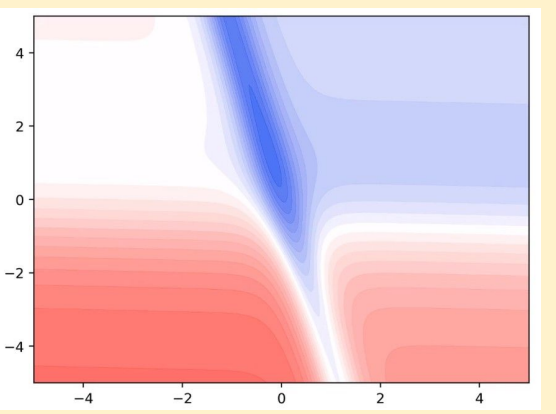
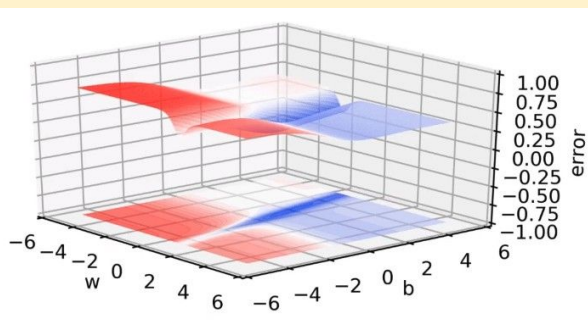
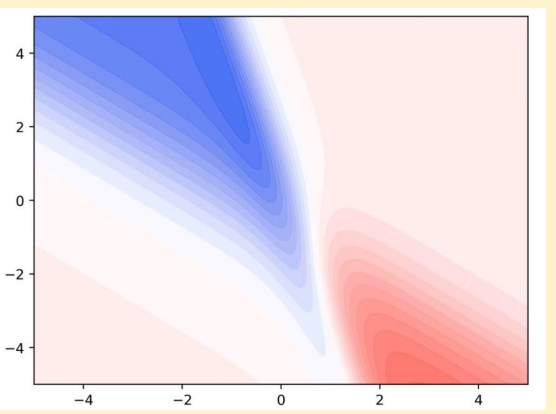
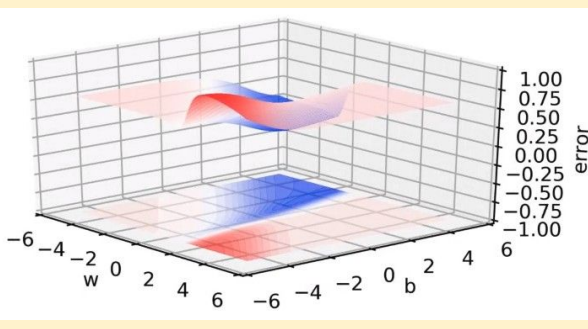
# PadhAI: Variants of Gradient Descent

## 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

Contour	3D Plot
	
	
	

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

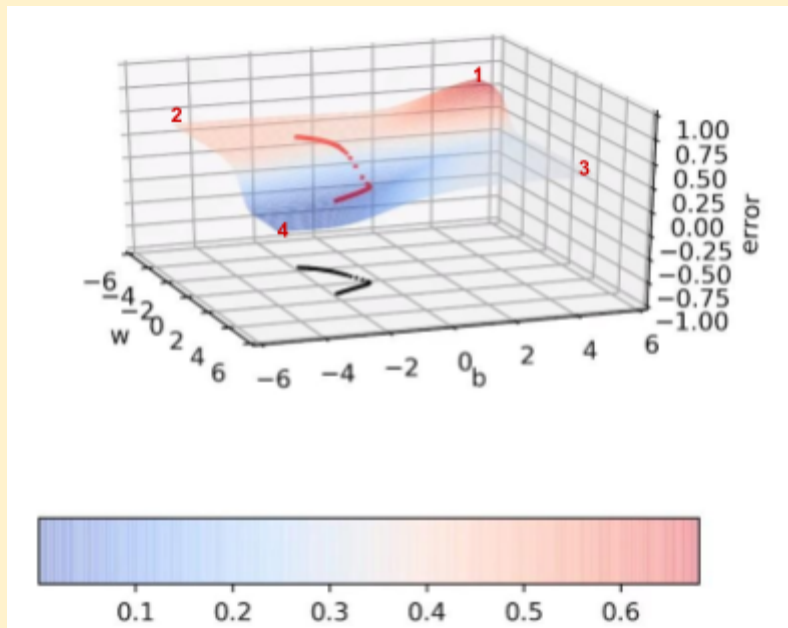
# PadhAI: Variants of Gradient Descent

## One Fourth Labs

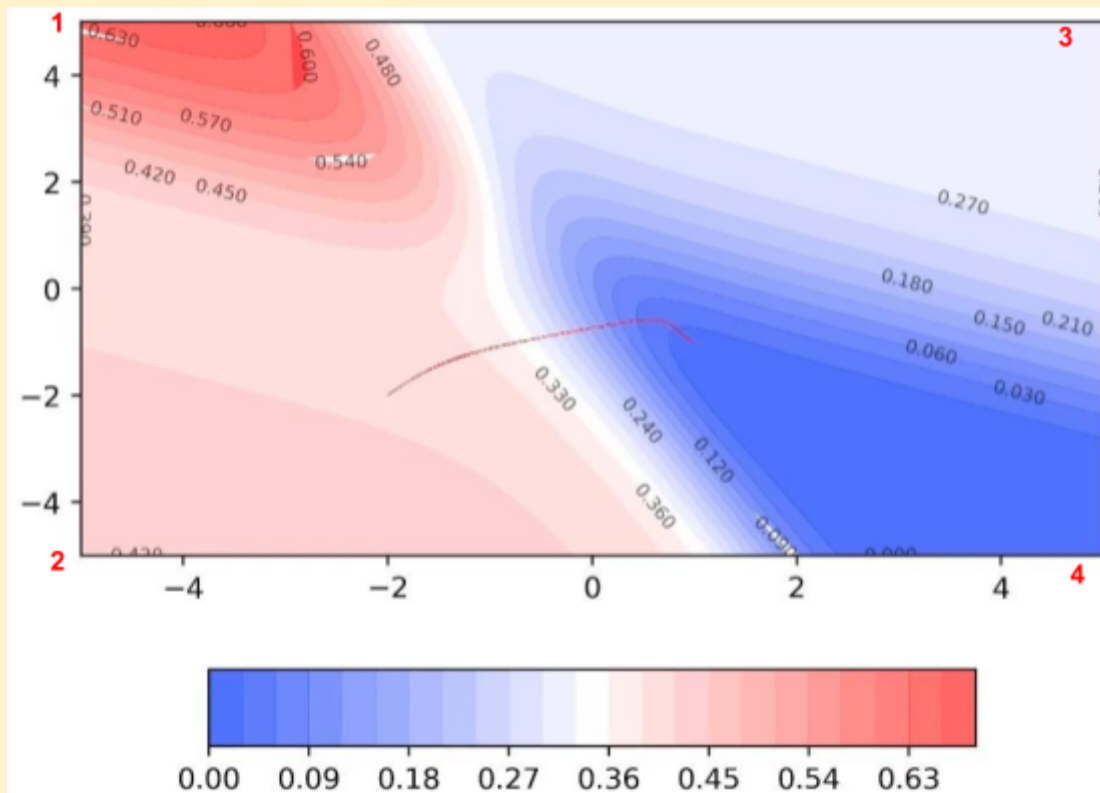
### Visualising gradient descent on a 2D contour map

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  $\gamma$  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

1. 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

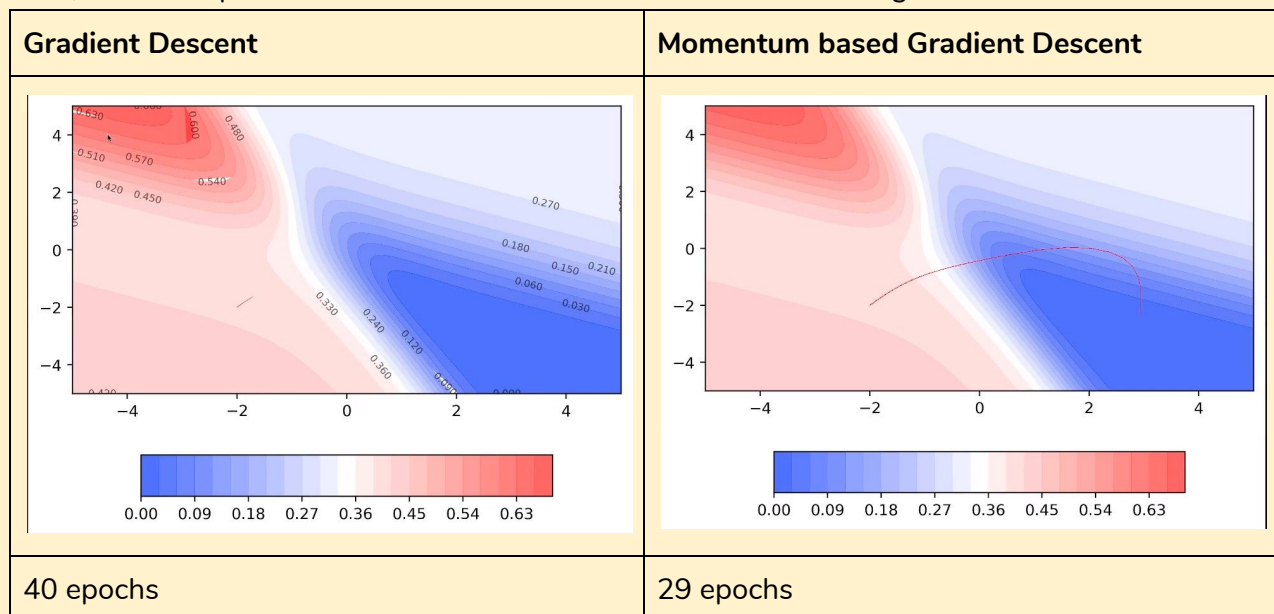
def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

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
```

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



3. 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.

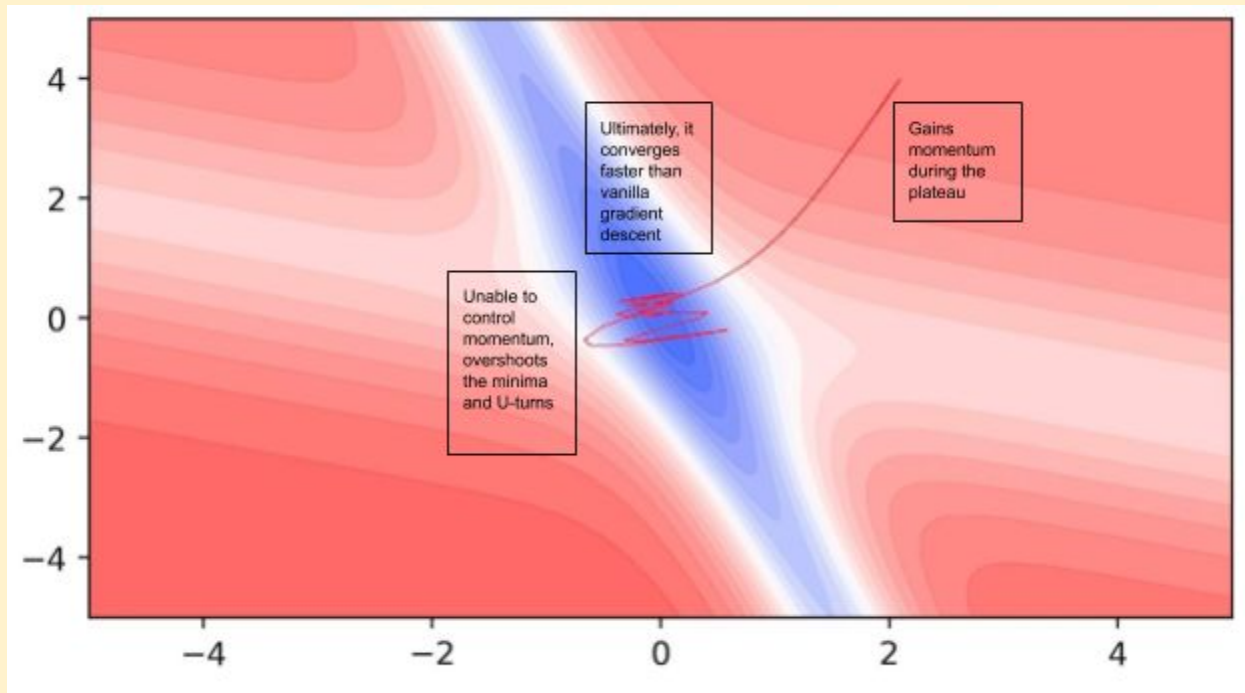
#### 1. Observations

- 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 ( $\omega_{temp}$ ).
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  $\omega_{temp}$  based on movement with history
  - b.  $\omega_{t+1} = \omega_{temp} - \eta \nabla \omega_{temp}$  move further in the direction of the derivative of  $\omega_{temp}$
  - c.  $v_t = \gamma * v_{t-1} + \eta \nabla \omega_{temp}$  update history with movement due to derivative of  $\omega_{temp}$

# 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

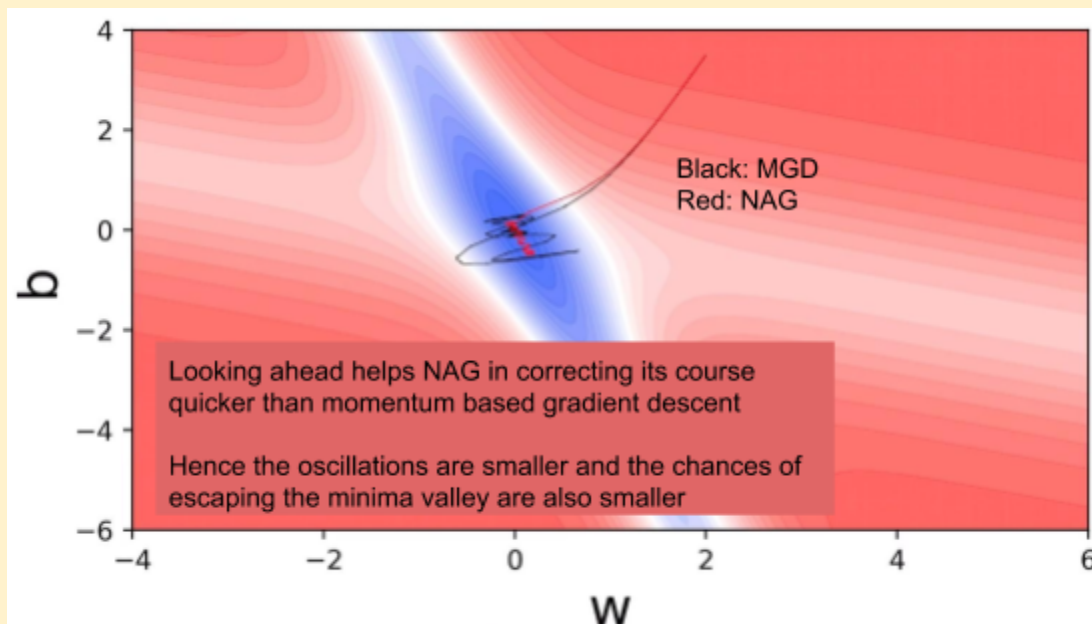
        #Compute the lookahead value
        w = w - gamma*v_w # this is w_temp
        b = b - gamma*v_b # this is b_temp

        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