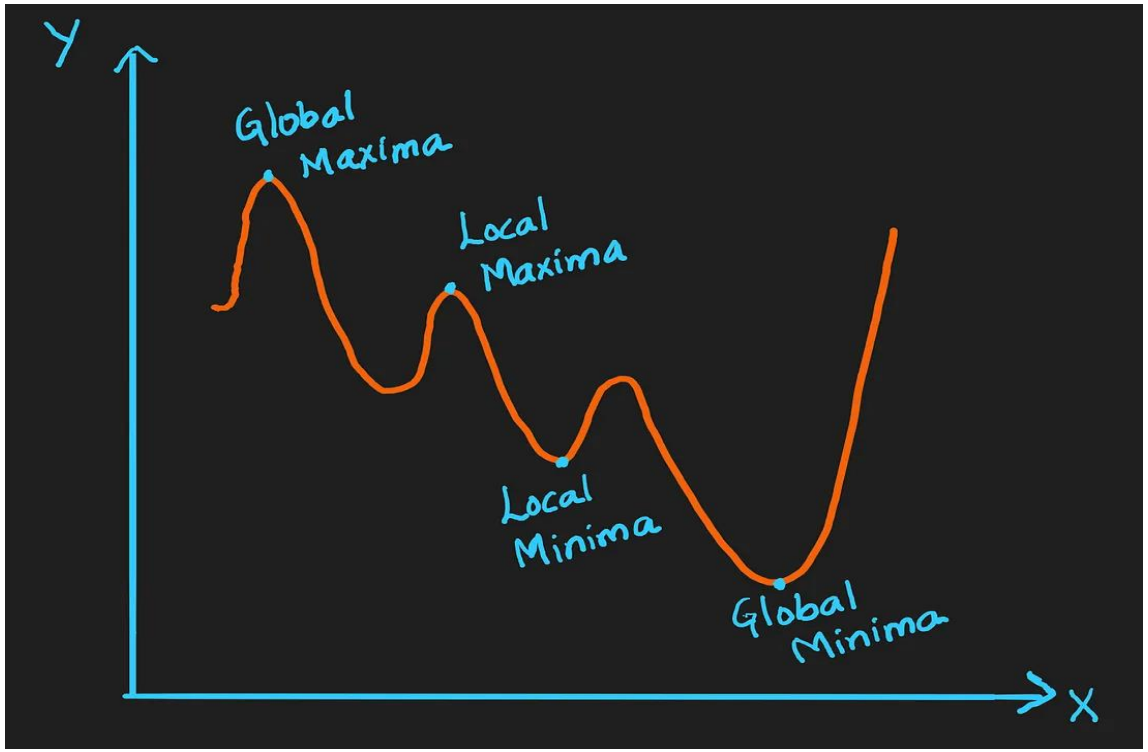
Understanding Optimization Algorithms in Machine Learning

- Optimization is the process where we train the model iteratively that results in a maximum and minimum function evaluation. It is one of the most important phenomena in Machine Learning to get better results.
- Why do we optimize our machine learning models? We compare the results in every iteration by changing the hyperparameters in each step until we reach the optimum results.
- We create an accurate model with less error rate. There are different ways using which we can optimize a model.

MAXIMA AND MINIMA

Maxima is the largest and Minima is the smallest value of a function within a given range. We represent them as below:



- *Global Maxima and Minima*: It is the maximum value and minimum value respectively on the entire domain of the function
- *Local Maxima and Minima*: It is the maximum value and minimum value respectively of the function within a given range.

- Convex optimization is a powerful tool used to solve optimization problems in various fields such as finance, engineering, and machine learning.
- In a convex optimization problem, the goal is to find a point that **maximizes or minimizes the objective function.**
- This is achieved through iterative computations involving convex functions, which are functions that always lie above their chords.

- **Convex optimization** plays a critical role in training machine learning models, which involves **finding the optimal parameters that minimize a given loss function**.
- In machine learning, convex optimization is used to solve a wide range of problems such as linear regression, logistic regression, support vector machines, and neural networks.

What are different techniques that are used for convex optimization?

- **Gradient methods using derivatives**: One of the approaches to convex optimization is via first computing the convex function's gradient, then using this derivative for performing convex optimization.
- If the convex problem has an easily computable convex objective function, then this gradient method of convex minimization can be used.
- This is because you are using information about the convex function to construct each step in the optimization. This convex optimization technique is widely used for solving convex problems.

- **Projected gradient methods**: This convex optimization approach was first introduced for solving convex problems with convex objective functions and convex constraints.
- It has an advantage over gradient methods using derivatives, if the system of convex constraints is not satisfied at the current iterate, it simply projects onto the subspace defined by these convex constraints.

- **Quasi-Newton methods**: This convex optimization approach is based on approximating the Hessian matrix of second derivatives by a quadratic approximation (where any convex function such as a linear one, can be used).
- The convex problem with convex constraints is solved using the quasi-Newton method to converge faster.
- It employs both first and second derivatives of convex objective functions to iteratively compute convex function minima using only function evaluations. The algorithm uses quadratic approximations around convex minimization iterations, thus requiring only function evaluations. An initial point is required to start convex optimization.

- **GRADIENT DESCENT**
- Gradient Descent is an optimization algorithm and it finds out the local minima of a differentiable function. It is a minimization algorithm that minimizes a given function.

$$y = 2x + 3$$

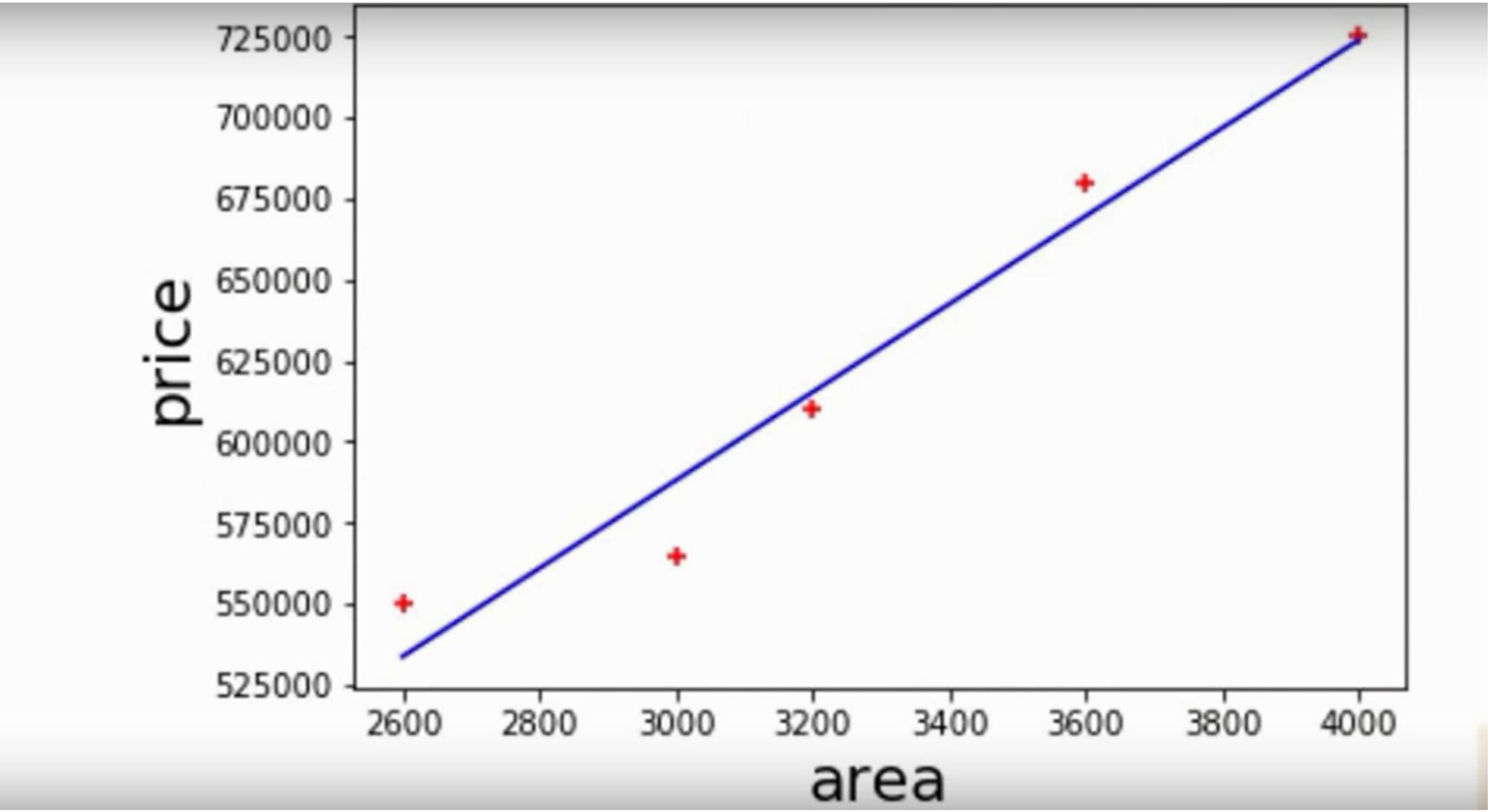
$$x = [1,2,3,4,5]$$

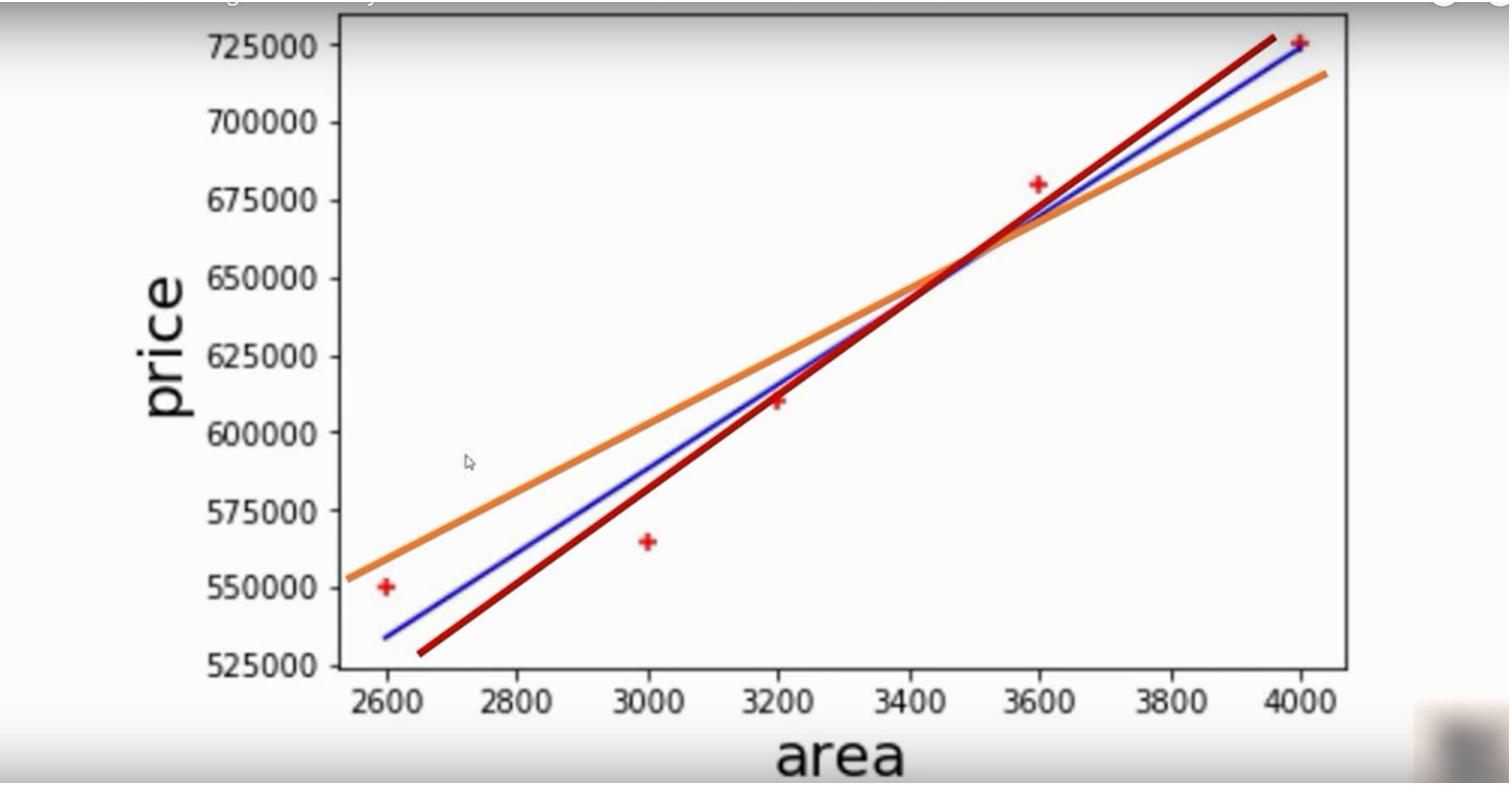
$$y = [5,7,9,11,13]$$

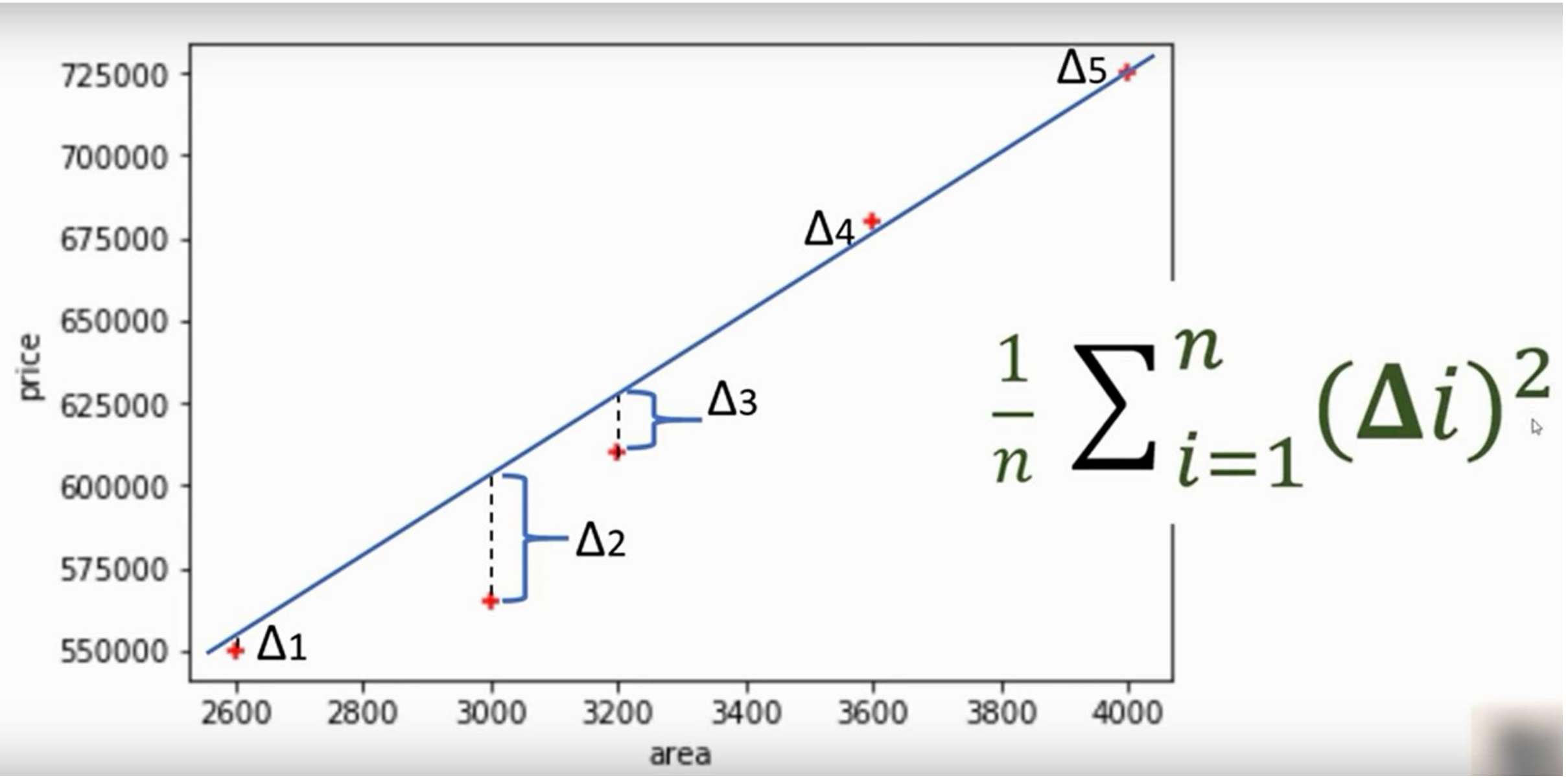
area = [2600,3000,3200,3600,4000]

price = [550k,565k,610k,680k,725k]

price = 135.78 * area + 180616.43


725000
700000
675000
650000
625000
600000
575000
550000
525000
price
2600
2800
3000
3200
3400
3600
3800
4000
area


725000
700000
675000
650000
625000
600000
575000
550000
525000
price
2600
2800
3000
3200
3400
3600
3800
4000
area



$$\frac{1}{n}\sum_{i=1}^{n}(\Delta i)^2$$

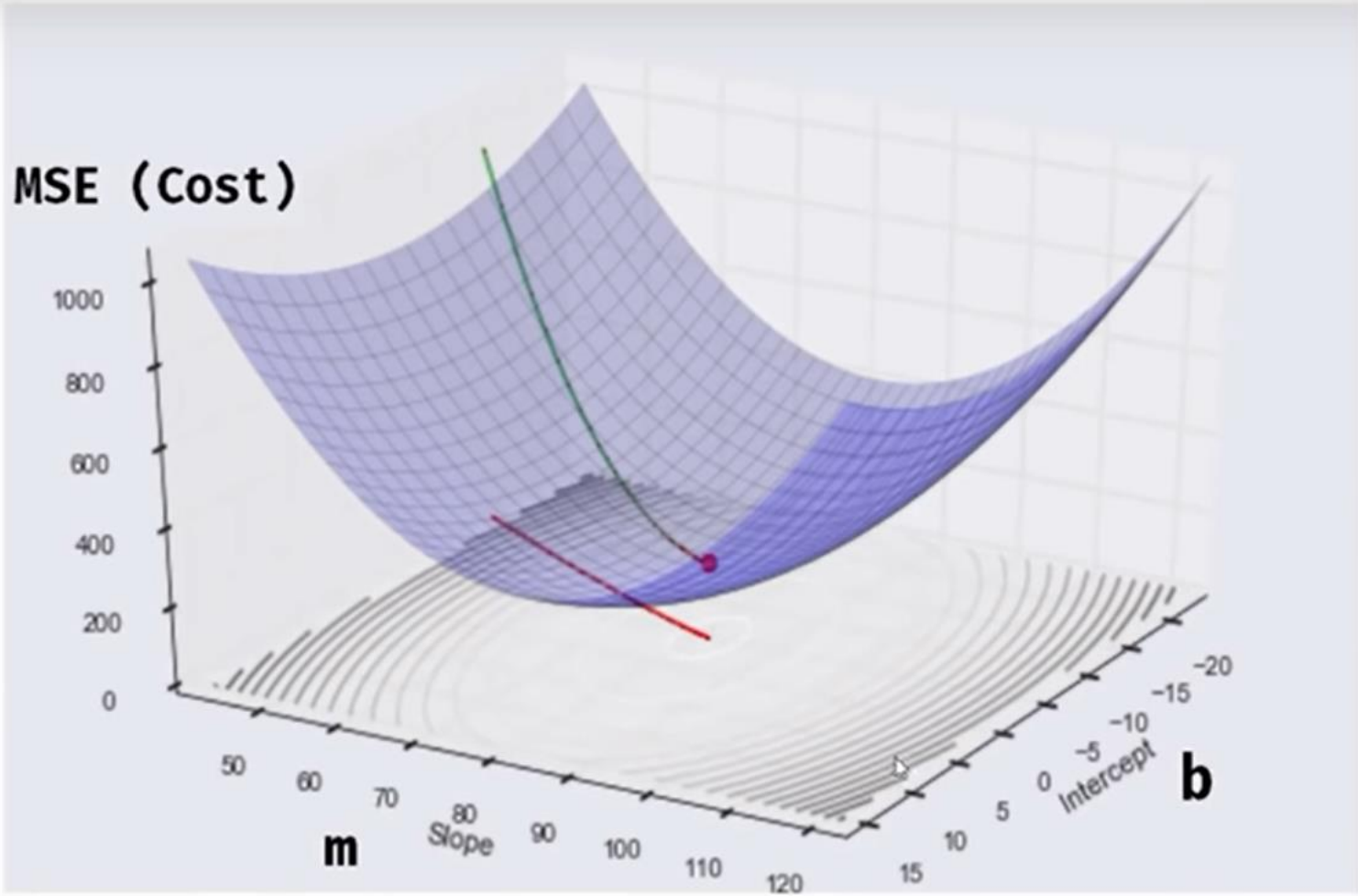
Mean Squared Error

$$ms\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - y_{predicted}\right)^2$$

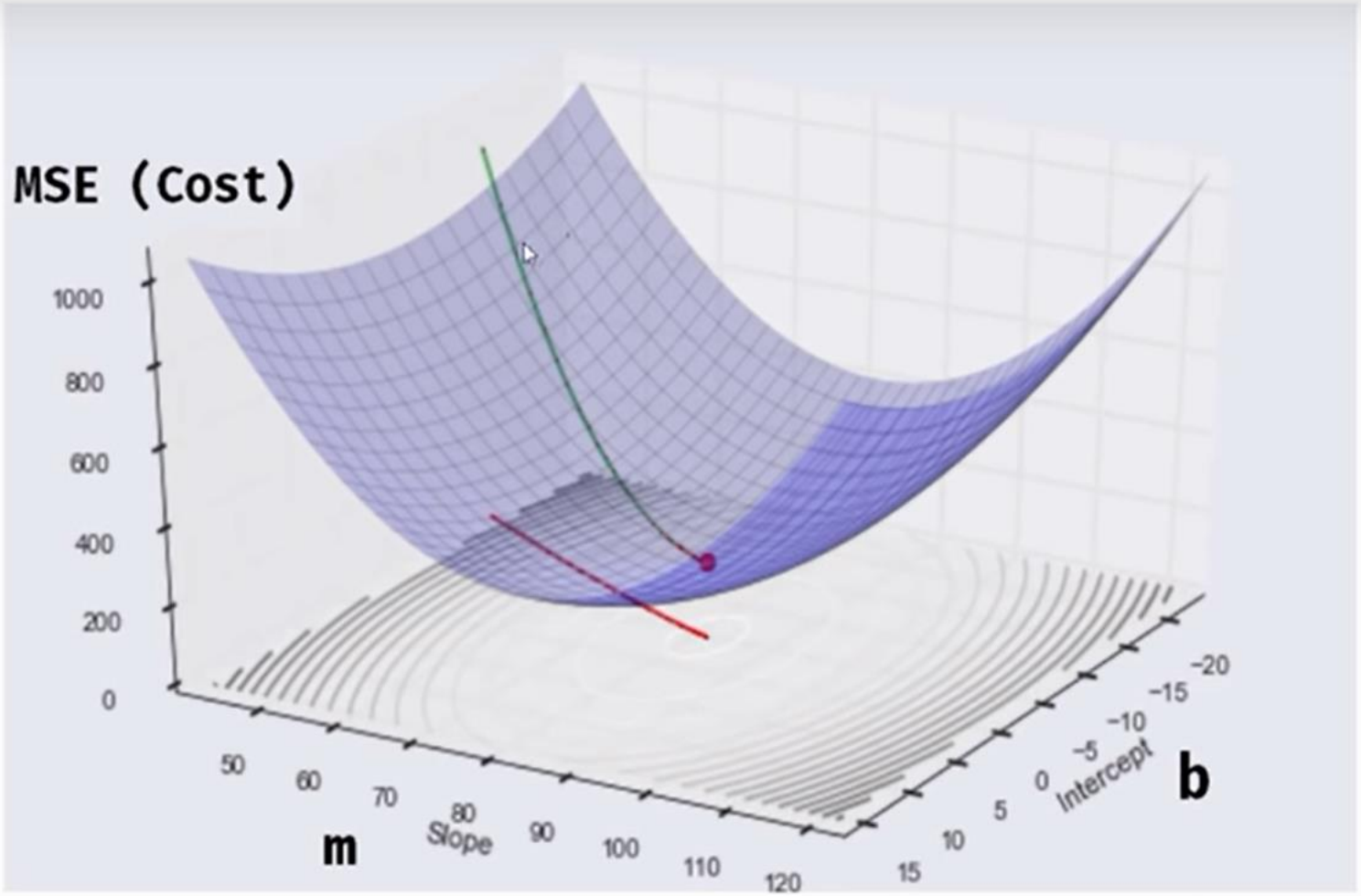
Mean Squared Error

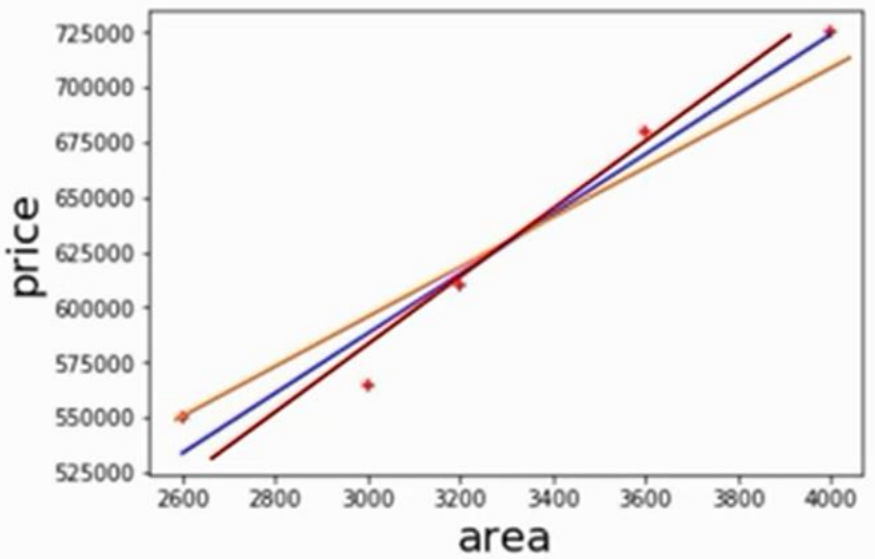
$$ms\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2$$

Cost Function

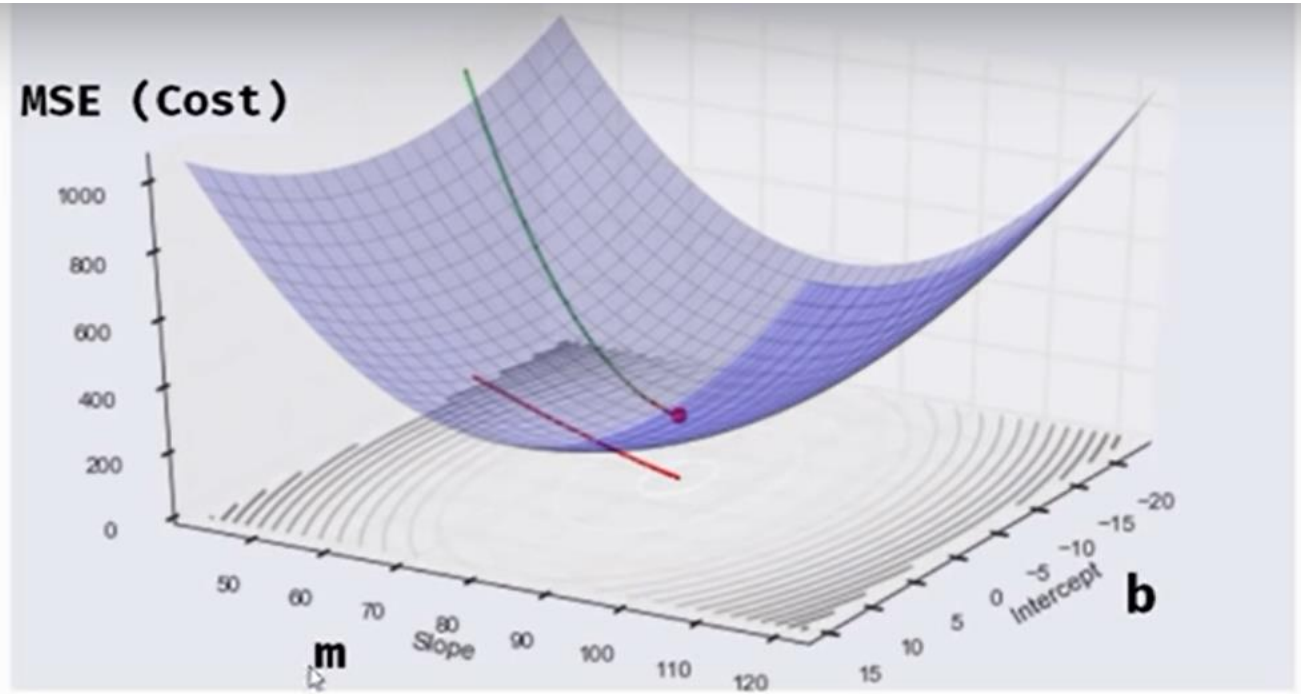


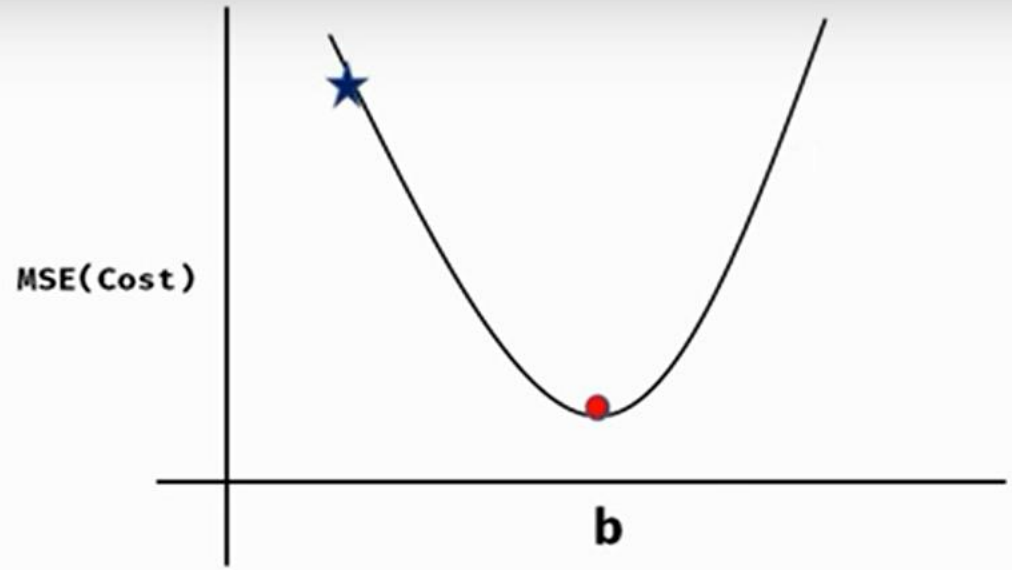
Reference: https://am207.github.io/2017/wiki/gradientdescent.html

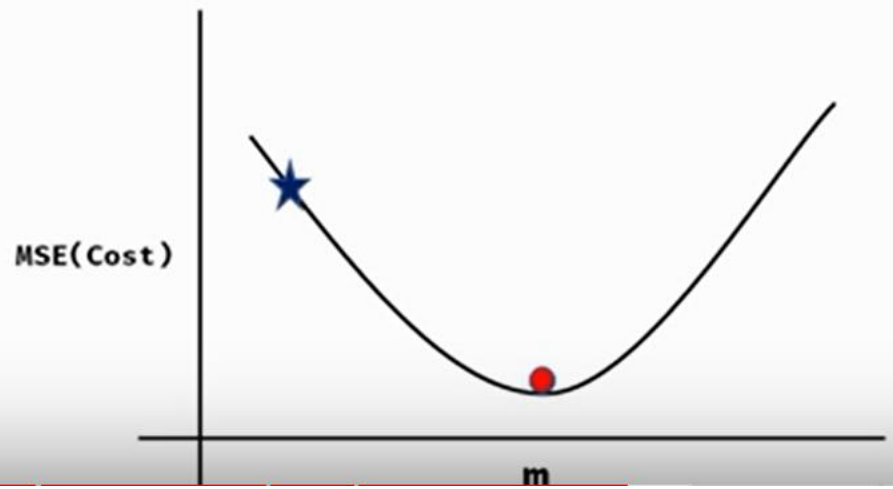




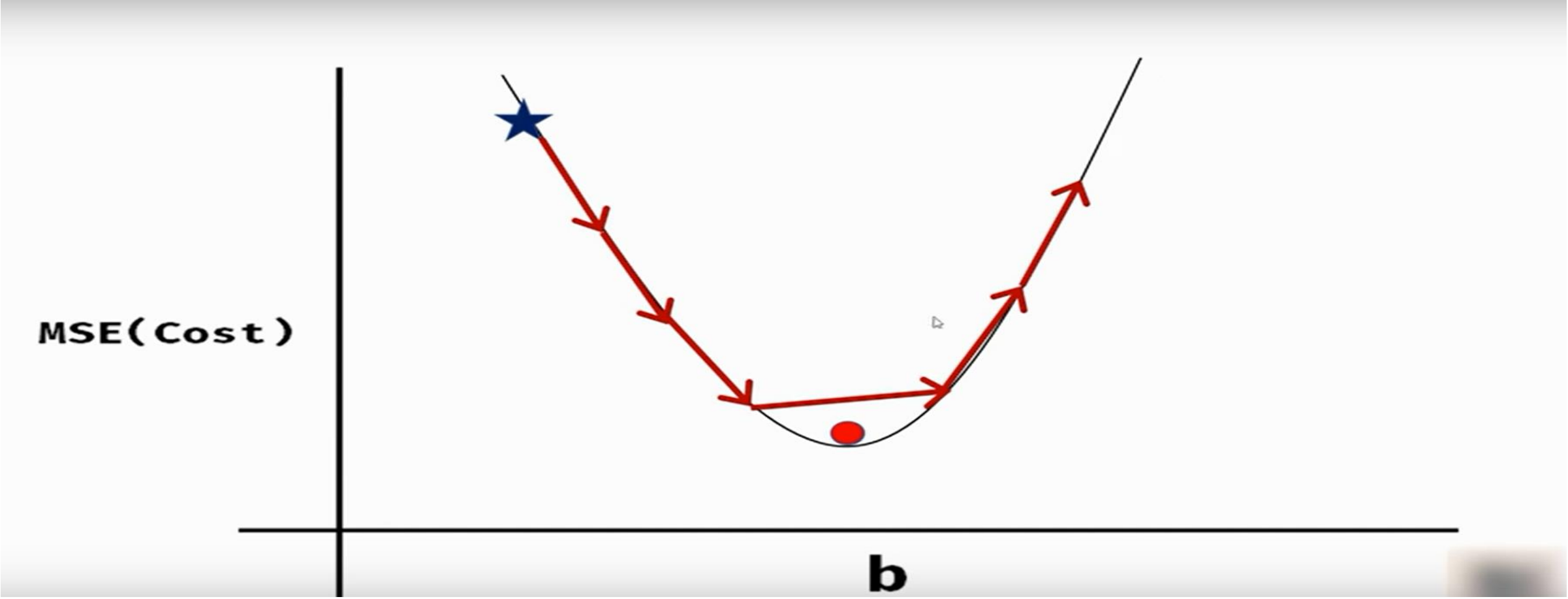
Reference: https://am207.github.io/2017/wiki/gradientdescent.html


MSE (Cost)
1000
800
600
400
200
0
50
60
70
80
90
100
110
120
m
Slope
15
10
5
0
−5
−10
−15
−20
Intercept
b

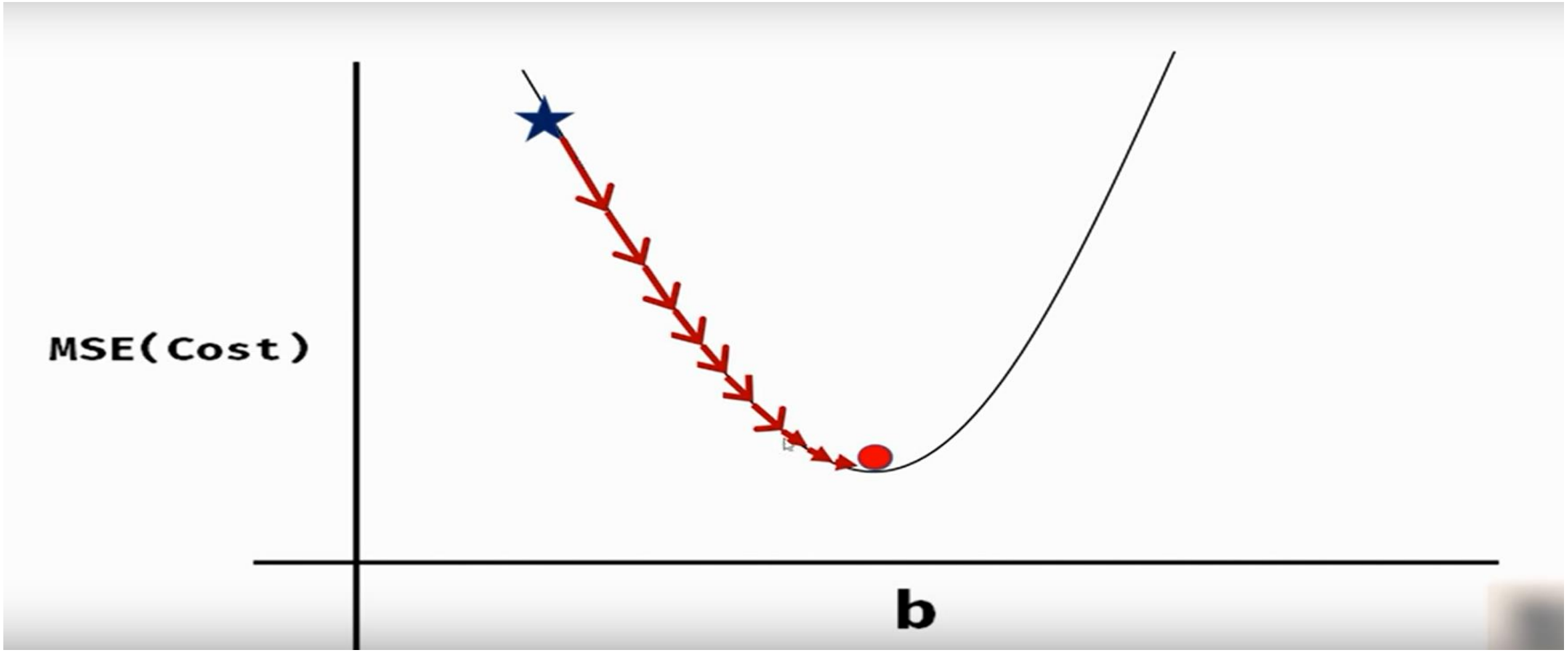

MSE(Cost)
b


MSE(Cost)
m

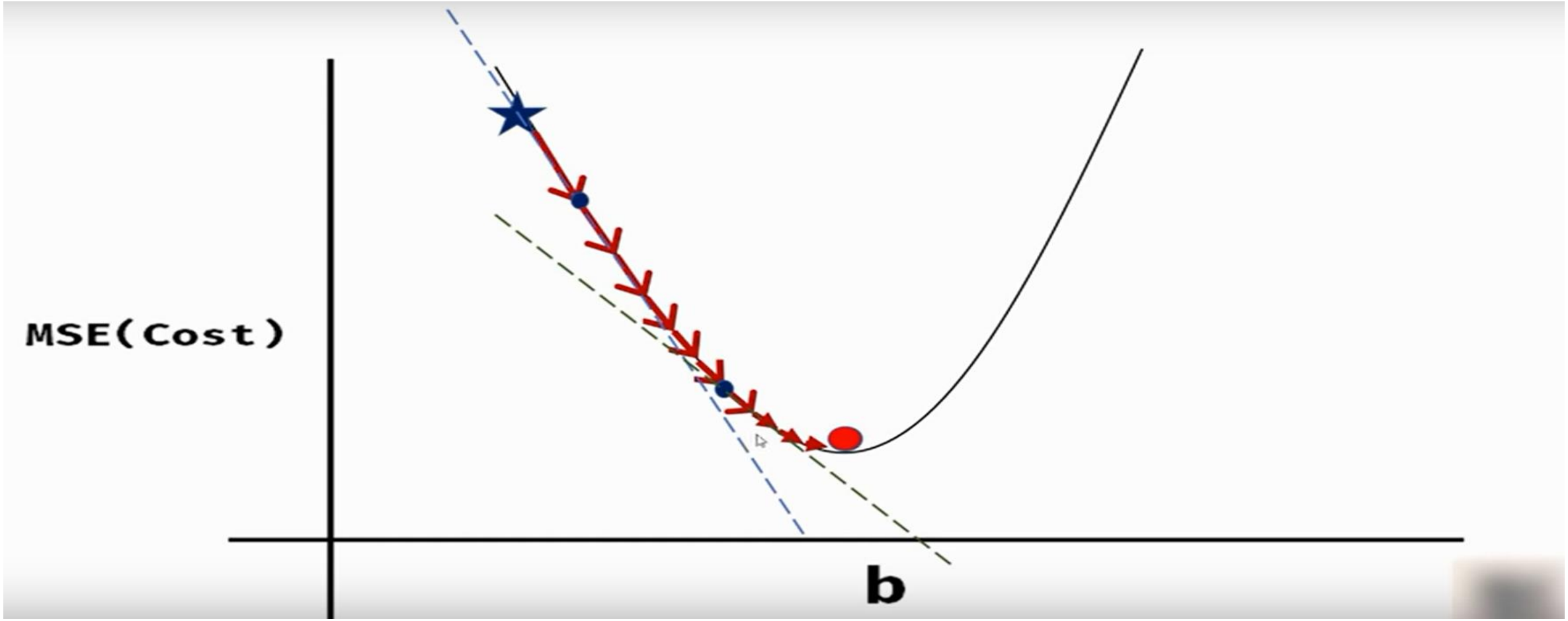
If equal size steps are taken,then it will never reach to global minima.

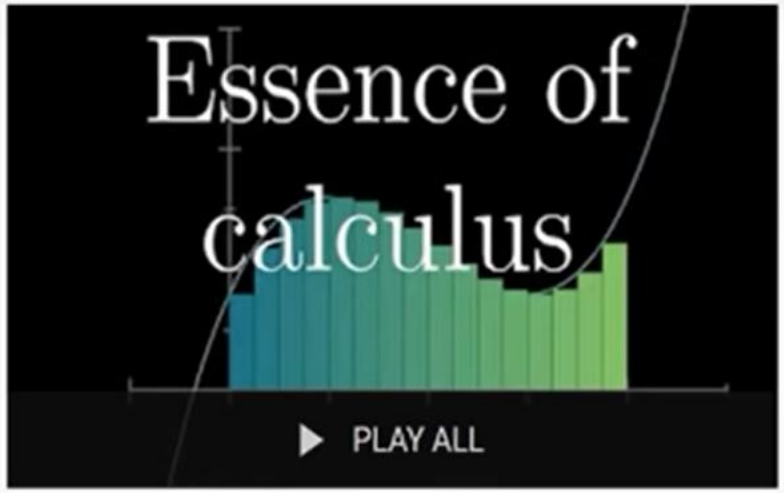


Variable size of steps are taken



Calculate slope at each point,which will guide the step size



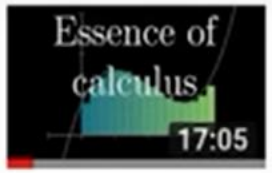


Essence of calculus

12 videos • 956,800 views • Last updated on May 23, 2019

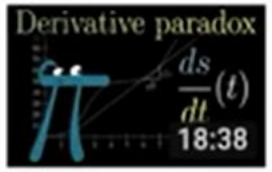
3Blue1Brown

SUBSCRIBE 1M

1 

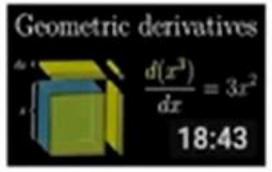
Essence of calculus, chapter 1

3Blue1Brown

2 

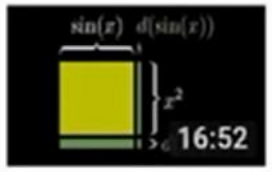
The paradox of the derivative | Chapter 2, Essence of calculus

3Blue1Brown

3 

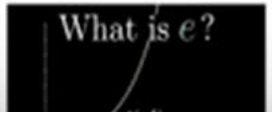
Derivative formulas through geometry | Chapter 3, Essence of calculus

3Blue1Brown

4 

Visualizing the chain rule and product rule | Chapter 4, Essence of calculus

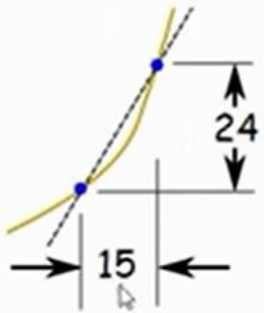
3Blue1Brown

5 

Derivatives of exponentials | Chapter 5, Essence of calculus

3Blue1Brown

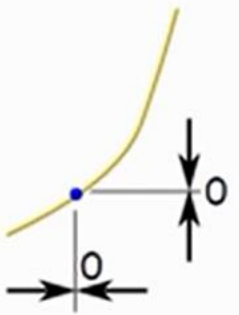
We can find an **average** slope between two points.



average slope = $\frac{24}{15}$

But how do we find the slope **at a point**?

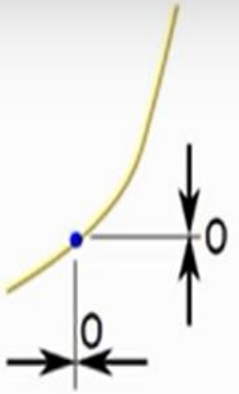
There is nothing to measure!



slope = $\frac{0}{0}$ = ???

But how do we find the slope **at a point**?

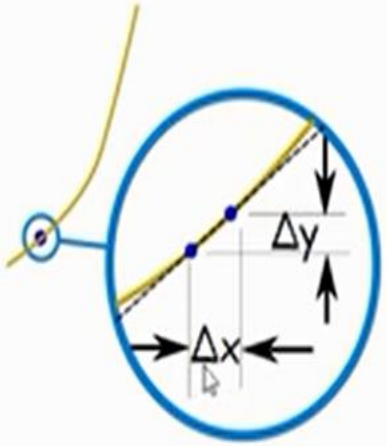
There is nothing to measure!



$$\text{slope} = \frac{0}{0} = ???$$

But with derivatives we use a small difference ...

... then have it **shrink towards zero**.



We write **dx** instead of **"Δx heads towards 0"**.

And "the derivative of" is commonly written $\frac{d}{dx}$:

$$\frac{d}{dx}x^2 = 2x$$

"The derivative of x^2 *equals* $2x$*"*

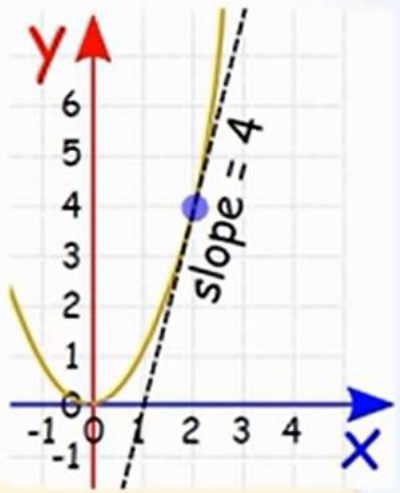
or simply *"d dx of* x^2 *equals* $2x$*"*

What does $\frac{d}{dx}x^2 = 2x$ mean?

It means that, for the function x^2, the slope or "rate of change" at any point is **2x**.

So when **x=2** the slope is **2x = 4**, as shown here:

Or when **x=5** the slope is **2x = 10**, and so on.



Note: sometimes $f'(x)$ is also used for "the derivative of":

$$f'(x) = 2x$$

"The derivative of f(x) equals 2x"

But what about a function of **two variables** (x and y):

$$f(x,y) = x^2 + y^3$$

To find its **partial** derivative **with respect to x** we treat **y as a constant** (imagine y is a number like 7 or something):

$$f'_x = 2x + 0 = 2x$$

Explanation:

- *the derivative of x^2 (with respect to x) is 2x*
- *we **treat y as a constant**, so y^3 is also a constant (imagine y=7, then 7^3=343 also a constant), and the derivative of a constant is 0*

$$f(x,y) = x^2 + y^3$$

To find its **partial** derivative **with respect to x** we treat **y as a constant** (imagine y is a number like 7 or something):

$$f'_x = 2x + 0 = 2x$$

Explanation:

- *the derivative of x^2 (with respect to x) is 2x*
- *we **treat y as a constant**, so y^3 is also a constant (imagine y=7, then 7^3=343 also a constant), and the derivative of a constant is 0*

To find the partial derivative **with respect to y**, we treat **x as a constant**:

$$f'_y = 0 + 3y^2 = 3y^2$$

- *the derivative of x^2 (with respect to x) is 2x*
- *we **treat y as a constant**, so y^3 is also a constant (imagine y=7, then 7^3=343 also a constant), and the derivative of a constant is 0*

To find the partial derivative **with respect to y**, we treat **x as a constant**:

$$f'_y = 0 + 3y^2 = 3y^2$$

Explanation:

- *we now **treat x as a constant**, so x^2 is also a constant, and the derivative of a constant is 0*
- *the derivative of y^3 (with respect to y) is $3y^2$*

That is all there is to it. Just remember to treat **all other variables as if they are constants**.

$$f(x,y) = x^3 + y^2$$

$$\partial f / \partial x = 3x^2 + 0 = 3x^2$$

$$\partial f / \partial y = 0 + 2y = 2y$$

$$ms\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2$$

$$\partial/_{\partial m} = \frac{2}{\mathrm{n}}\sum_{i=1}^{n} -x_i\left(y_i - (mx_i + b)\right)$$

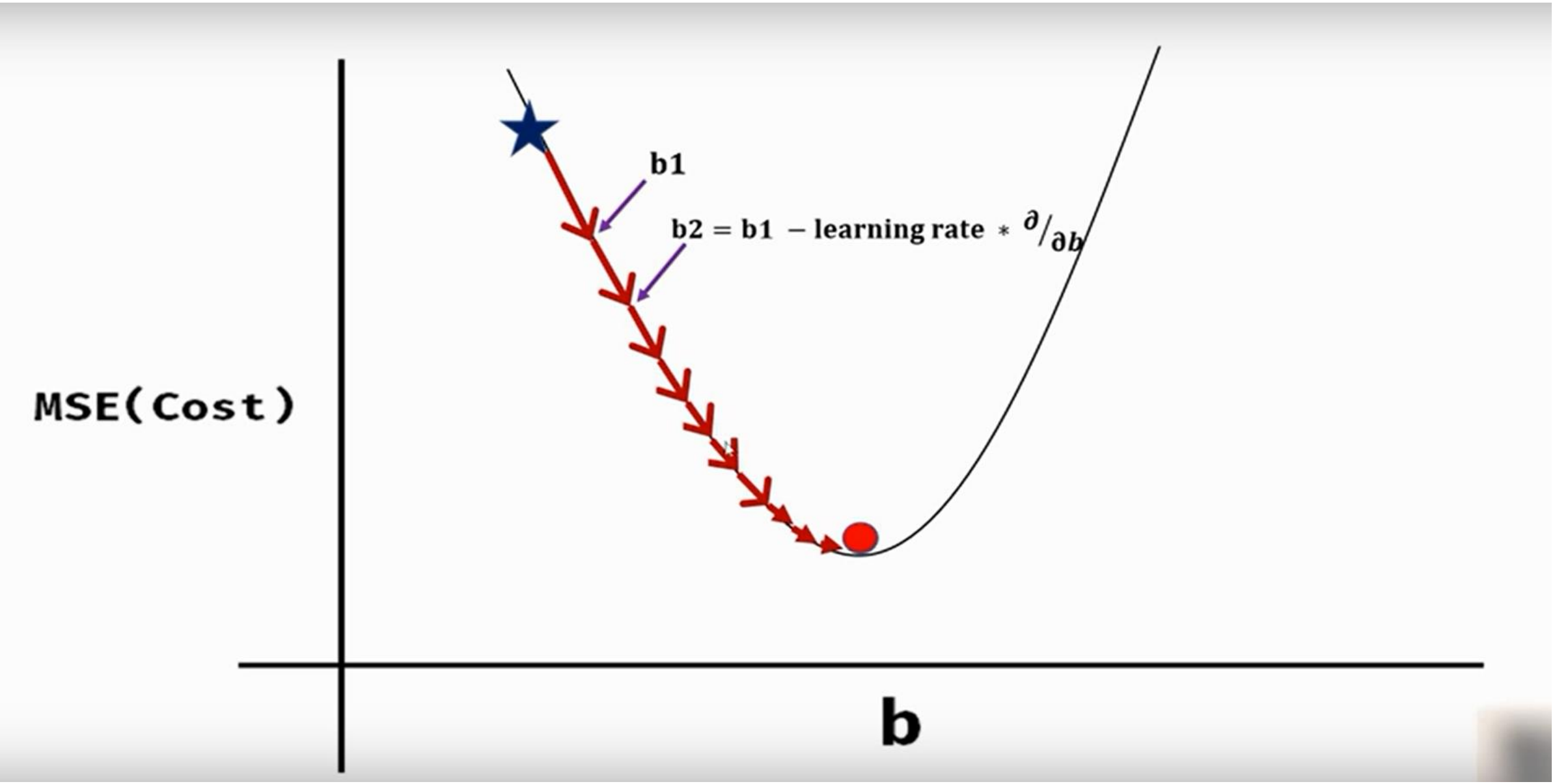
$$ms\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2$$

$$\partial/_{\partial m} = \frac{2}{\mathrm{n}}\sum_{i=1}^{n} -x_i\left(y_i - (mx_i + b)\right)$$

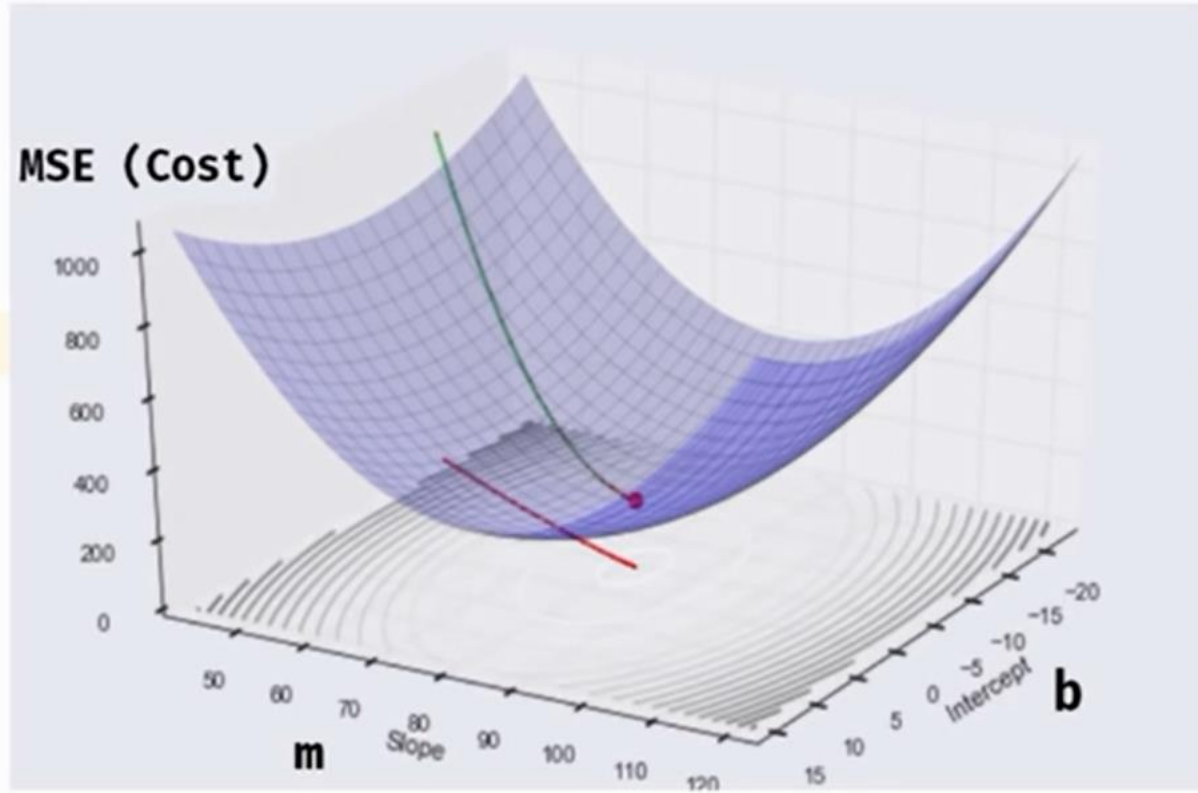
$$\partial/_{\partial b} = \frac{2}{\mathrm{n}}\sum_{i=1}^{n} -\left(y_i - (mx_i + b)\right)$$

$$m = m - \text{learning rate} * \partial/\partial m$$

$$b = b - \text{learning rate} * \partial/\partial b$$


b1
b2 = b1 − learning rate ∗ ∂/∂b
MSE(Cost)
b

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 1000
    


x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])

gradient_descent(x,y)
```



```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 1000
    n = len(x)

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum((y-y_predicted))


x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])
```

$$\partial/_{\partial m} = \frac{2}{n}\sum_{i=1}^{n} -x_i\left(y_i - (mx_i + b)\right)$$

$$\partial/_{\partial b} = \frac{2}{n}\sum_{i=1}^{n} -\left(y_i - (mx_i + b)\right)$$

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 1000
    n = len(x)
    learning_rate = 0.001

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
        print ("m {}, b {}, cost {} iteration {}".format(m_curr,b_curr,ci))

x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])

gradient_descent(x,y)
```

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - y_{predicted}\right)^2$$

Cost Function

```
cost
callable(object)                    builtins
chr(i)                              builtins
classmethod                         builtins
compile(args, kwargs)               builtins
complex                             builtins
copyright(args, kwargs)             builtins
credits(args, kwargs)               builtins
b_curr
m_curr
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 1000
    n = len(x)
    learning_rate = 0.001

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
```

$$m = m - \text{learning rate} * {}^{\partial}/_{\partial m}$$

$$b = b - \text{learning rate} * {}^{\partial}/_{\partial b}$$

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 10000
    n = len(x)
    learning_rate = 0.001

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
        print ("m {}, b {}, cost {} iteration {}".format(m_curr,b_curr,cost, i))

x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])
```

Run gradient_descent

```
m 2.0214562692718476, b 2.922536059891256, cost 0.0010929644012168923 iteration 9990
m 2.0214490149885194, b 2.9225622501558424, cost 0.0010922254719085017 iteration 9991
m 2.0214417631578367, b 2.9225884315655994, cost 0.0010914870421740582 iteration 9992
m 2.0214345137789707, b 2.9226146041235213, cost 0.0010907491116757683 iteration 9993
m 2.021427266851092, b 2.9226407678326005, cost 0.0010900116800761411 iteration 9994
m 2.0214200223733725, b 2.9226669226958286, cost 0.0010892747470378908 iteration 9995
m 2.0214127803449835, b 2.922693068716197, cost 0.0010885383122239487 iteration 9996
m 2.021405540765097, b 2.9227192058966946, cost 0.0010878023752974074 iteration 9997
m 2.0213983036328846, b 2.9227453342403105, cost 0.0010870669359217566 iteration 9998
m 2.021391068947519, b 2.9227714537500327, cost 0.0010863319937606147 iteration 9999

Process finished with exit code 0
```

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 10
    n = len(x)
    learning_rate = 0.001

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
        print ("m {}, b {}, cost {} iteration {}".format(m_curr,b_curr,cost, i))

x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])
```

gradient_descent

```
C:\ProgramData\Anaconda3\python.exe C:/Code/py/ML/3_gradient_descent/gradient_descent.py
m 0.062, b 0.018000000000000002, cost 89.0 iteration 0
m 0.122528, b 0.035592000000000006, cost 84.881304 iteration 1
m 0.181618832, b 0.052785648000000004, cost 80.955185108544 iteration 2
m 0.239306503808, b 0.069590363712, cost 77.21263768455901 iteration 3
m 0.29562421854195203, b 0.086015343961728, cost 73.64507722605434 iteration 4
m 0.35060439367025875, b 0.10206956796255283, cost 70.2443206760065 iteration 5
m 0.40427867960173774, b 0.11776180246460617, cost 67.00256764921804 iteration 6
m 0.4566779778357119, b 0.13310060678206653, cost 63.912382537082294 iteration 7
m 0.5078324586826338, b 0.14809433770148814, cost 60.966677449199324 iteration 8
m 0.5577715785654069, b 0.16275115427398937, cost 58.15869595270883 iteration 9

Process finished with exit code 0
```

```python
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 10000
    n = len(x)
    learning_rate = 0.08

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
```

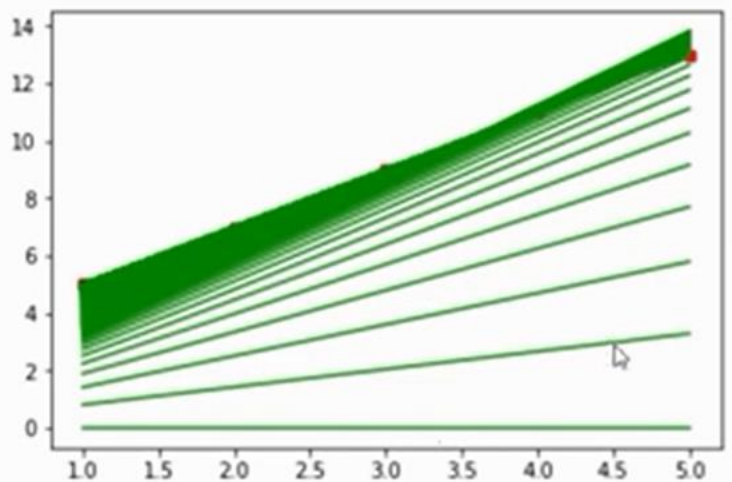
gradient_descent

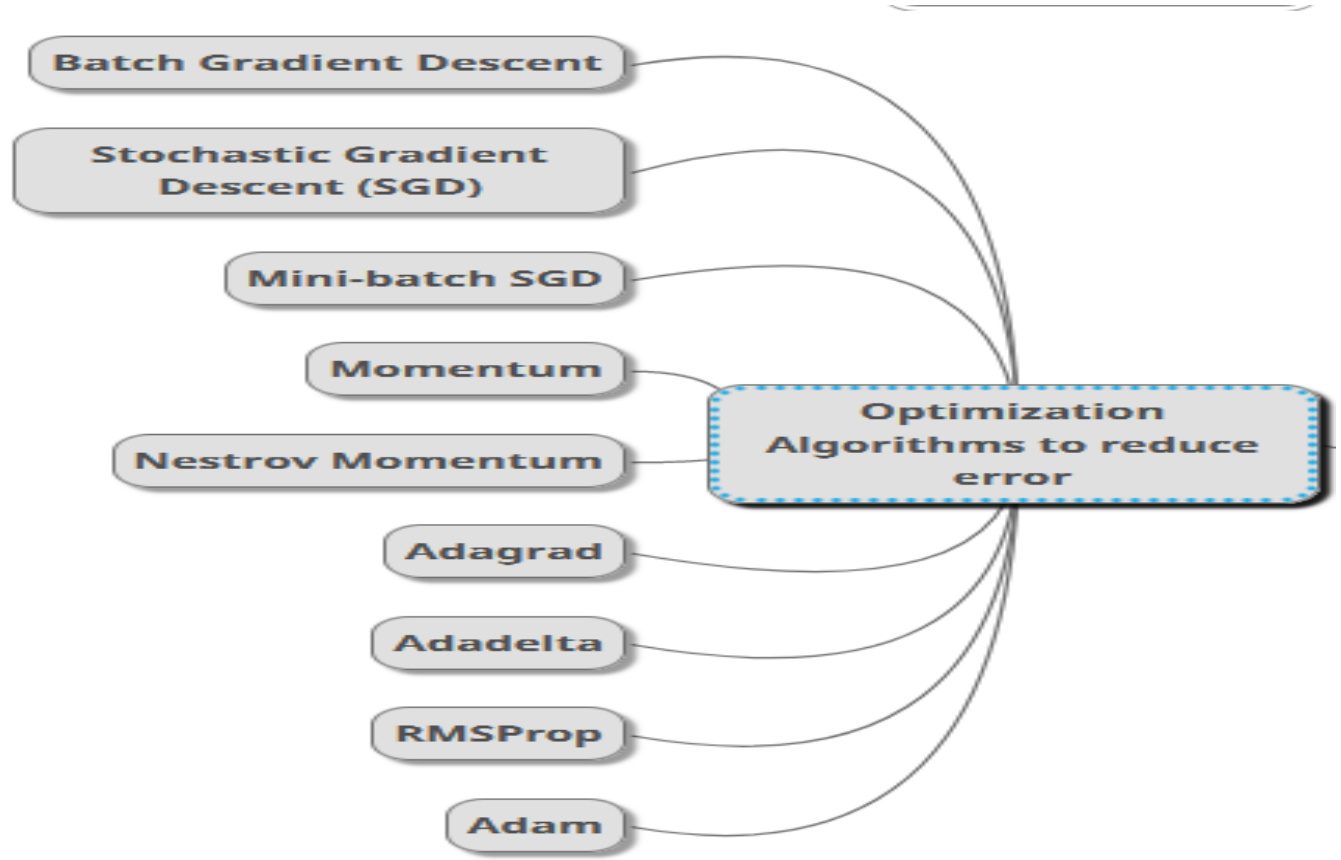
```
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9982
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9983
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9984
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9985
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9986
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9987
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9988
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9989
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9990
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9991
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9992
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9993
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9994
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9995
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9996
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9997
m 2.000000000000001, b 2.9999999999999947, cost 1.0255191767873153e-29 iteration 9998
m 2.000000000000002, b 2.999999999999995, cost 1.0255191767873153e-29 iteration 9999

Process finished with exit code 0
```

```
        m_curr = m_curr - rate * md
        b_curr = b_curr - rate * yd
```

In [28]:
```
x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])
```

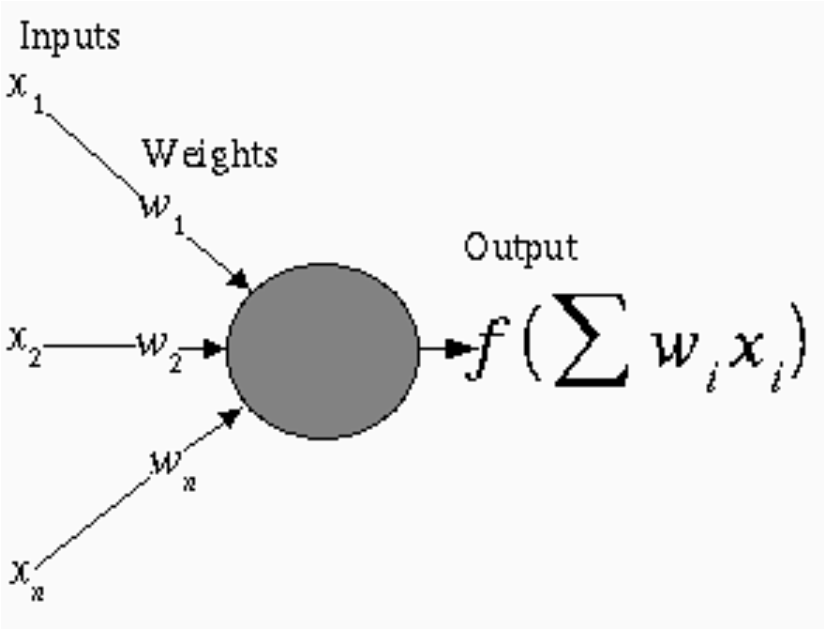
In [35]:
```
gradient_descent(x,y)
```




Batch Gradient Descent
Stochastic Gradient Descent (SGD)
Mini-batch SGD
Momentum
Nestrov Momentum
Optimization Algorithms to reduce error
Adagrad
Adadelta
RMSProp
Adam

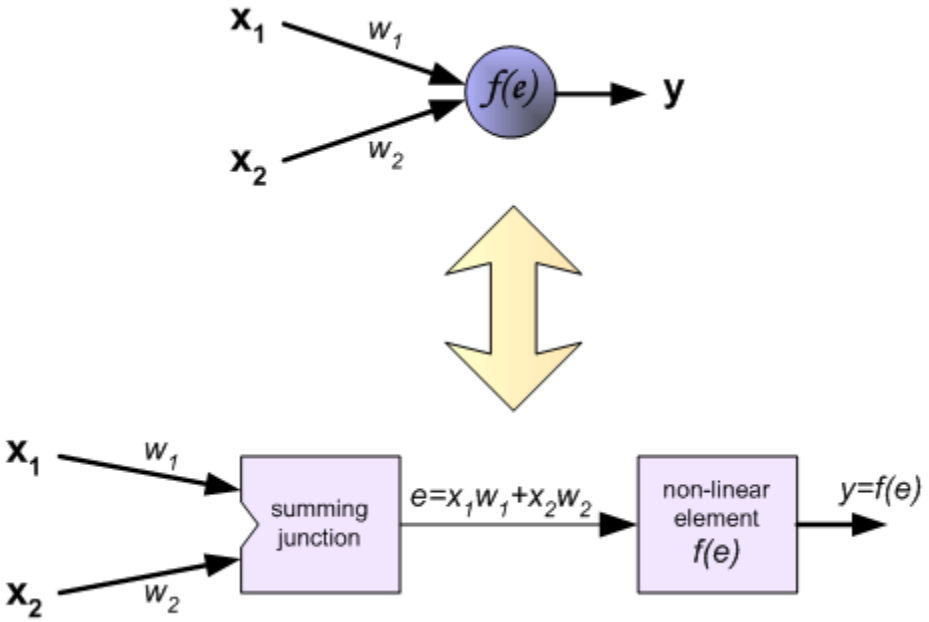
Basic Neuron Model In A Feedforward Network

- **Inputs x_i** arrive through pre-synaptic connections
- Synaptic efficacy is modeled using real **weights w_i**
- The response of the neuron is a **nonlinear function *f*** of its weighted inputs



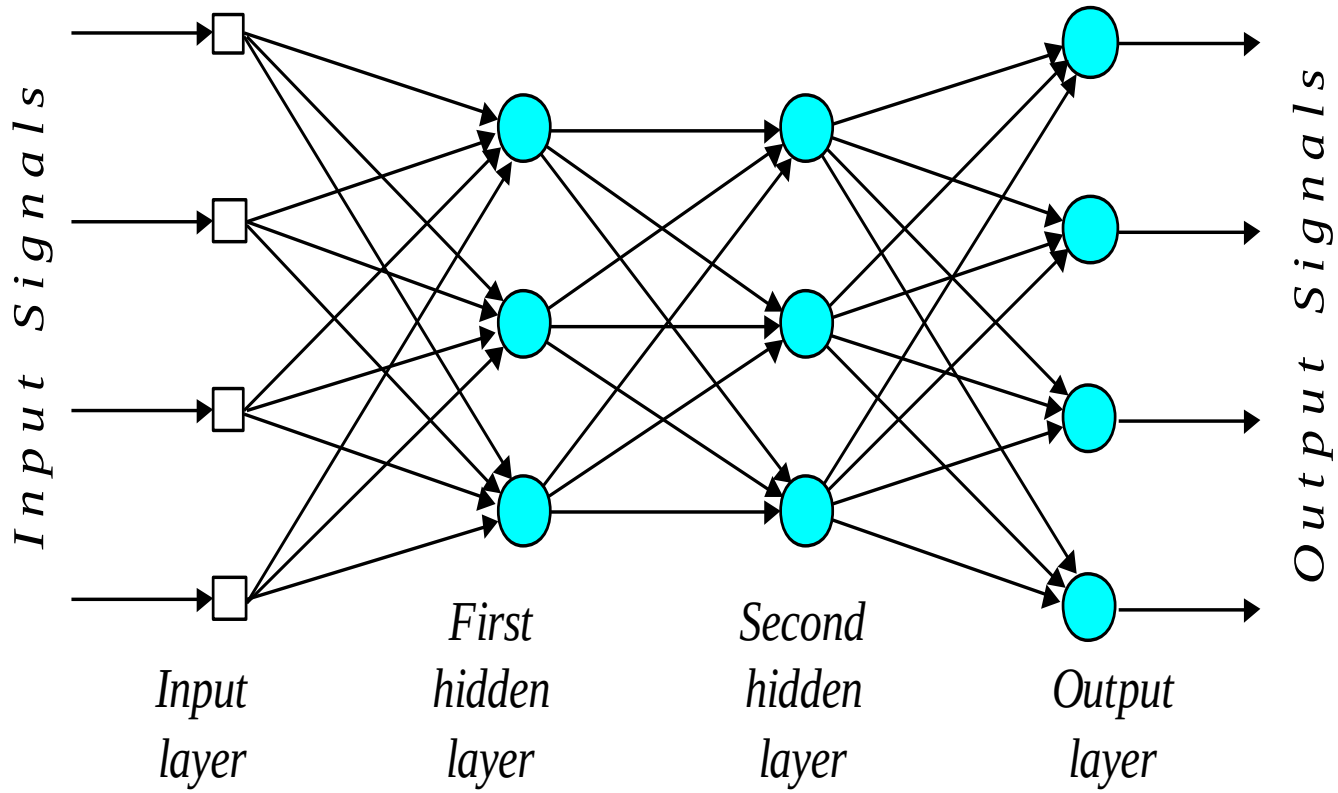
Learning Algorithm: Backpropagation

Each neuron is composed of two units. First unit adds products of weights coefficients and input signals. The second unit realise nonlinear function, called neuron transfer (activation) function. Signal *e* is adder output signal, and *y = f(e)* is output signal of nonlinear element. Signal *y* is also output signal of neuron.

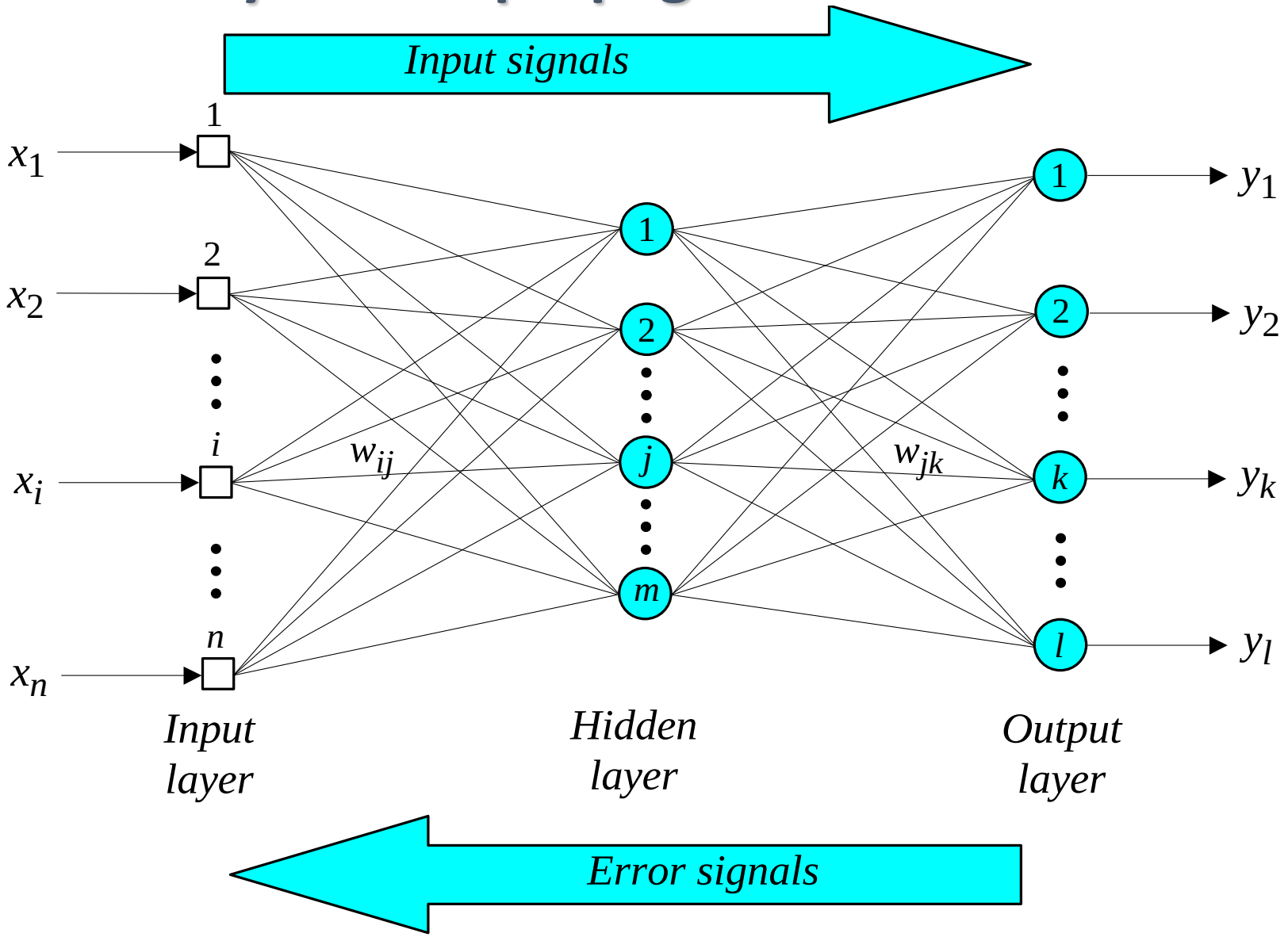


- **Epoch** – The number of times the algorithm runs on the whole training dataset.
- **Sample** – A single row of a dataset.
- **Batch** – It denotes the number of samples to be taken to for updating the model parameters.
- **Learning rate** – It is a parameter that provides the model a scale of how much model weights should be updated.
- **Cost Function/Loss Function** – A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.
- **Weights/ Bias** – The learnable parameters in a model that controls the signal between two neurons.
- Now let’s explore each optimizer.

Artificial Neural Networks

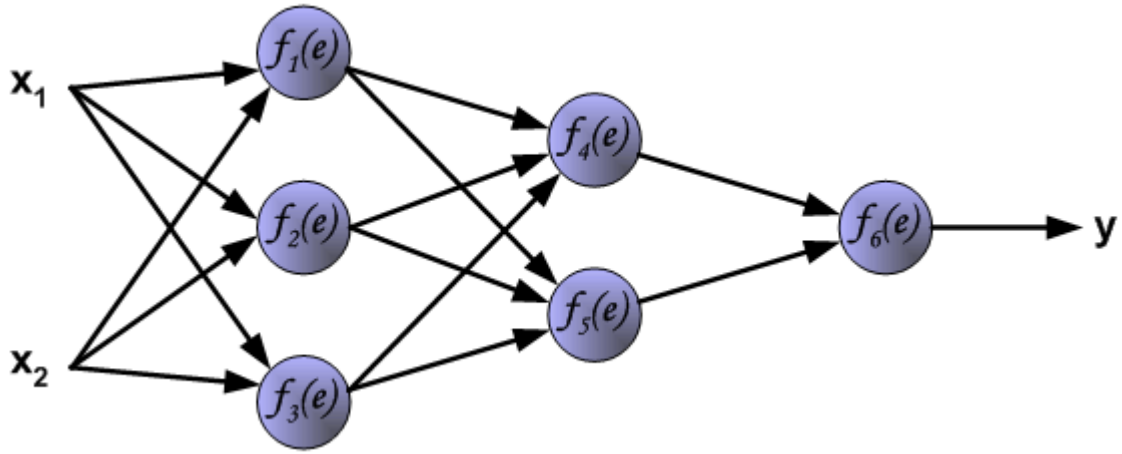


Three-layer back-propagation neural network



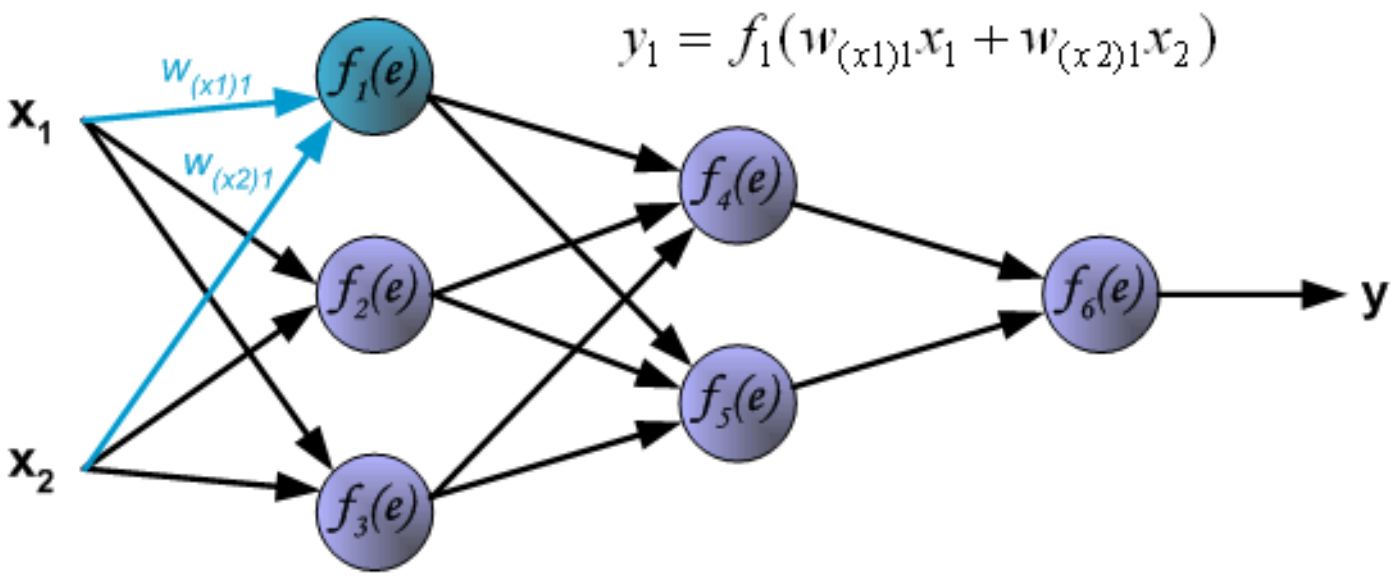
Learning Algorithm: Backpropagation

The following slides describes **teaching process** of multi-layer neural network employing **backpropagation** algorithm. To illustrate this process the three layer neural network with two inputs and one output,which is shown in the picture below, is used:

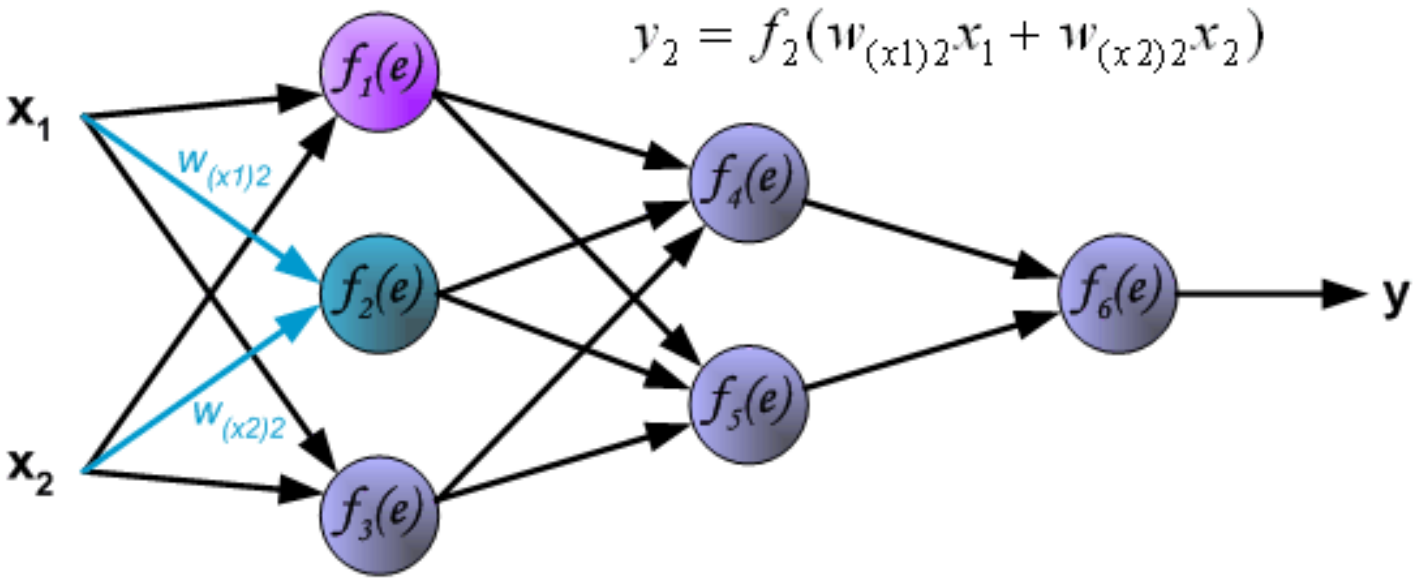


Learning Algorithm: Backpropagation

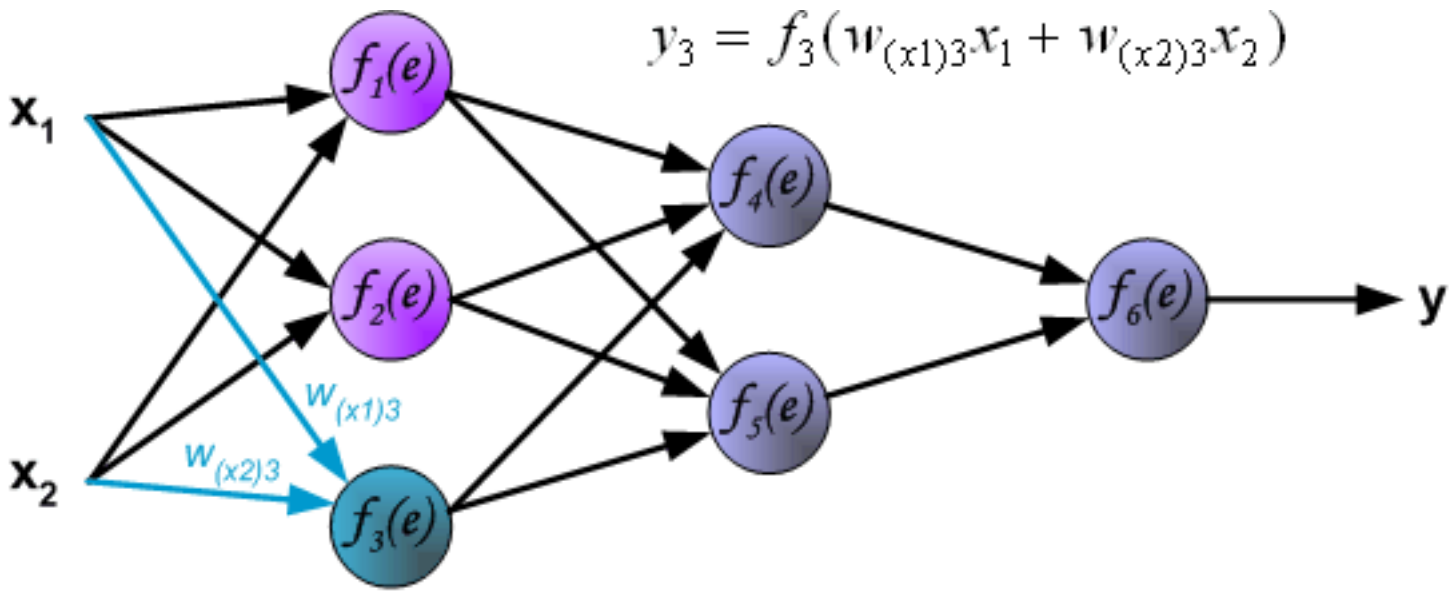
Pictures below illustrate how signal is propagating through the network, Symbols $w_{(xm)n}$ represent weights of connections between network input x_m and neuron n in input layer. Symbols y_n represents output signal of neuron n.



Learning Algorithm: Backpropagation

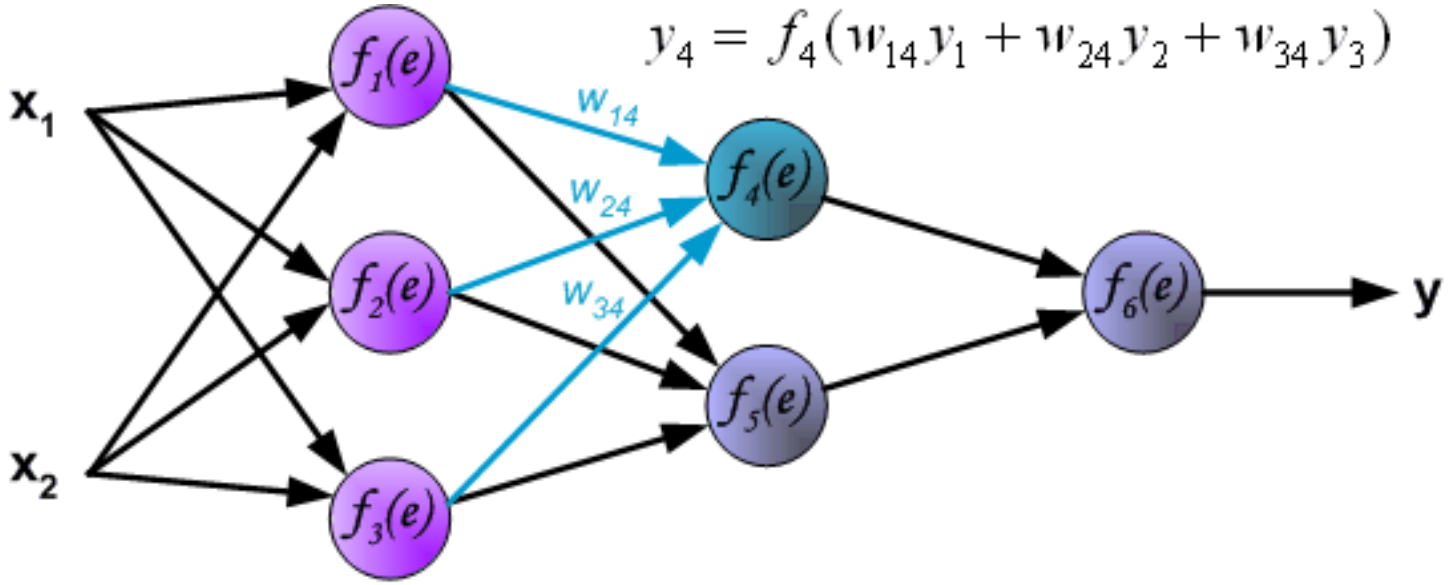


Learning Algorithm: Backpropagation

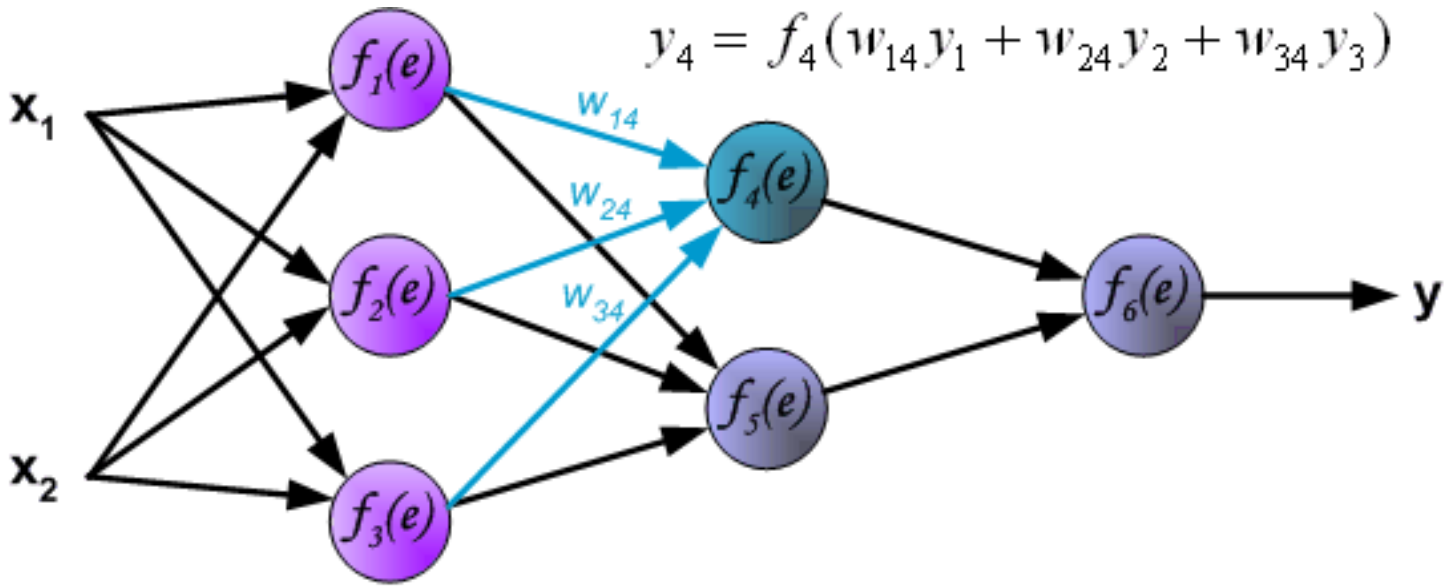


Learning Algorithm: Backpropagation

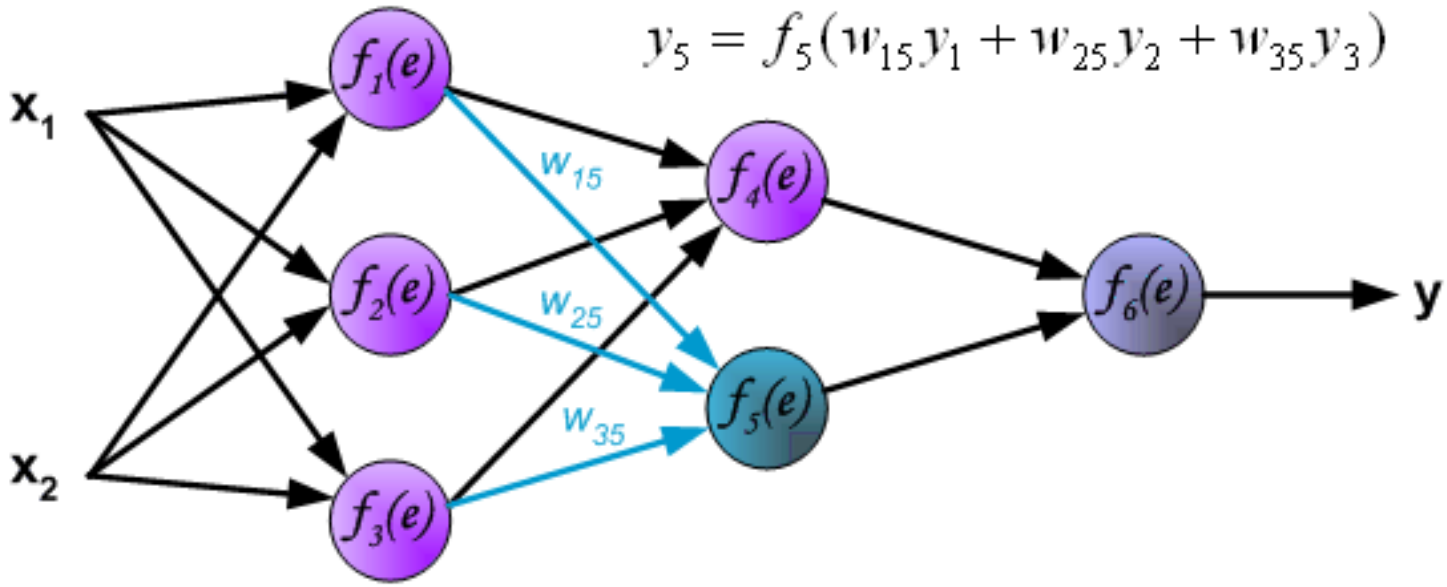
Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron *m* and input of neuron *n* in the next layer.



Learning Algorithm: Backpropagation

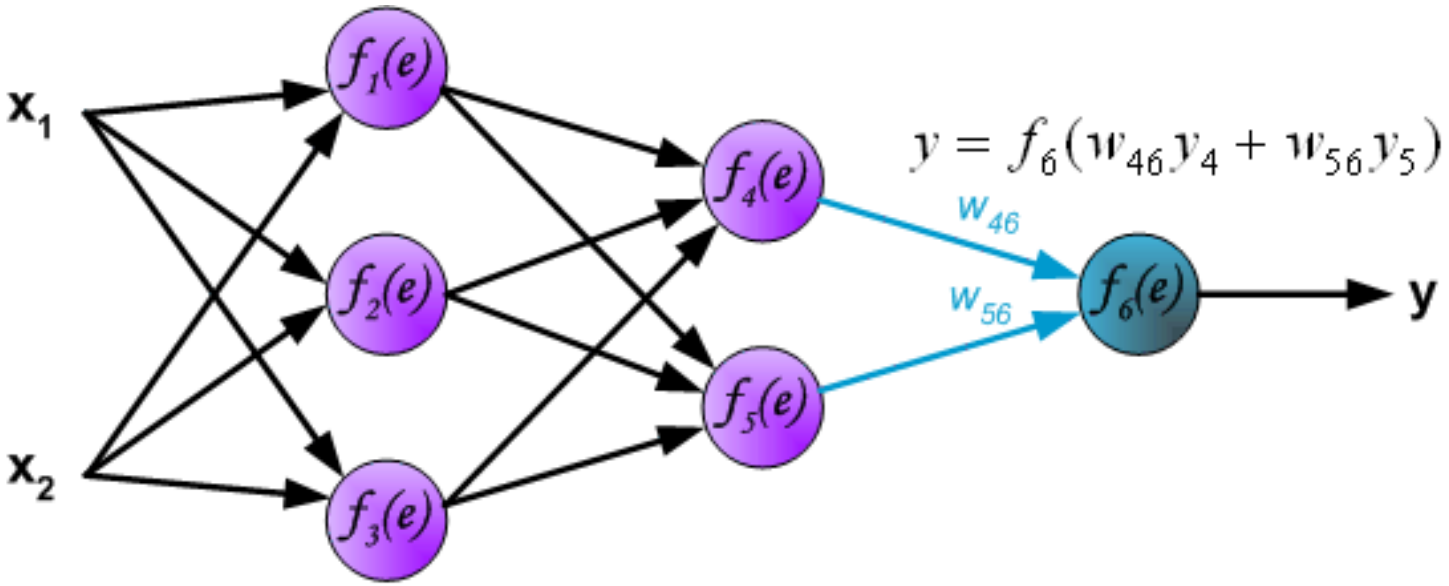


Learning Algorithm: Backpropagation



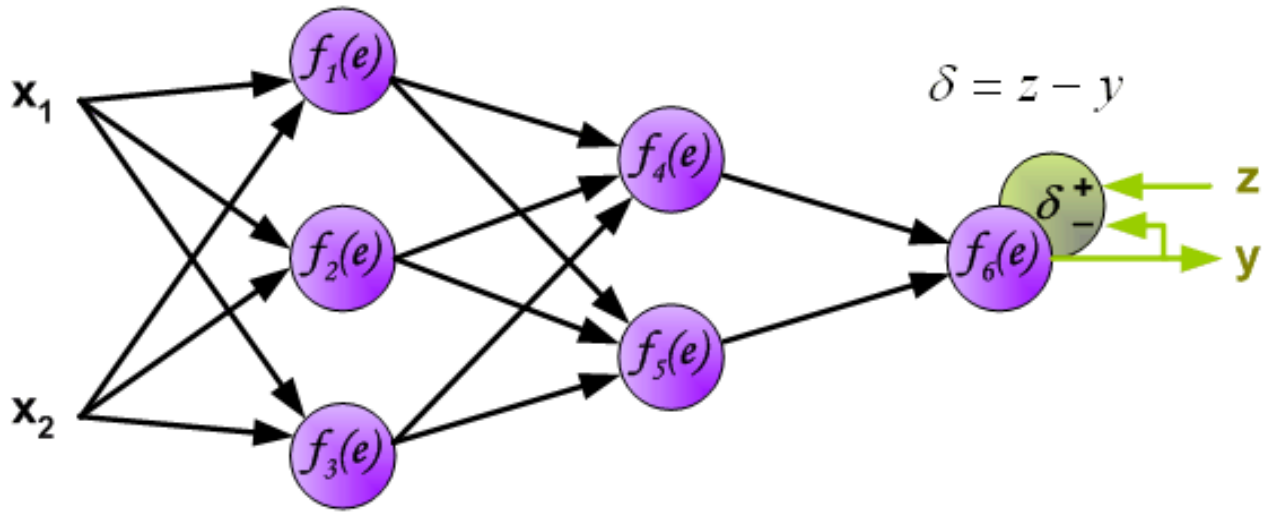
Learning Algorithm: Backpropagation

Propagation of signals through the output layer.



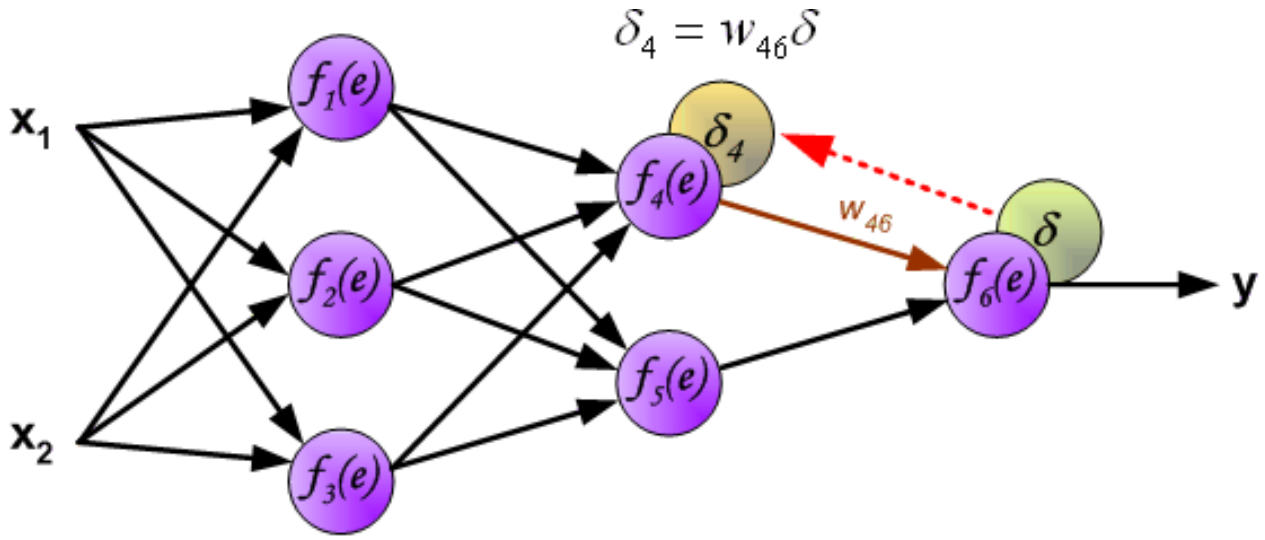
Learning Algorithm: Backpropagation

In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron



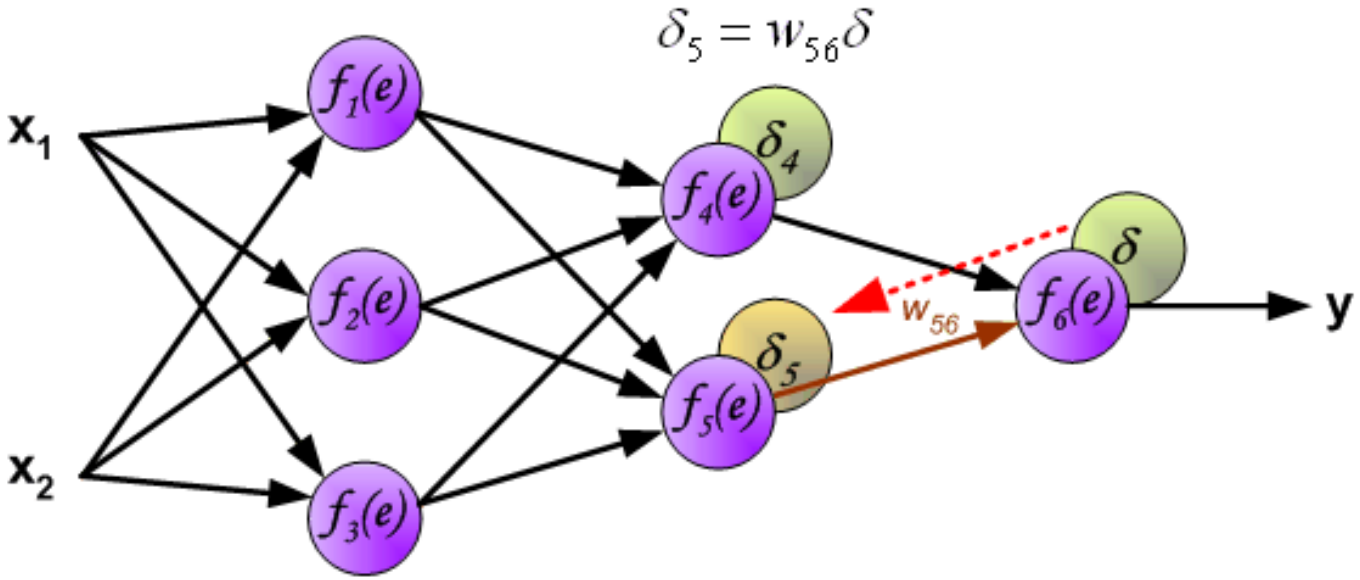
Learning Algorithm: Backpropagation

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.



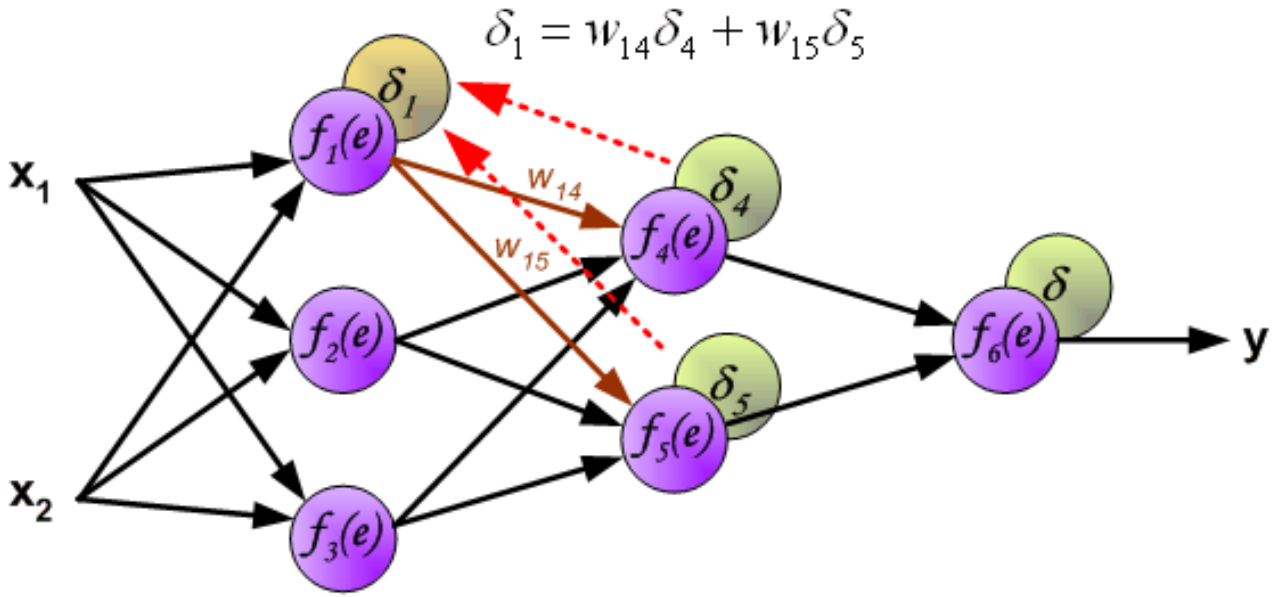
Learning Algorithm: Backpropagation

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.



Learning Algorithm: Backpropagation

The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:

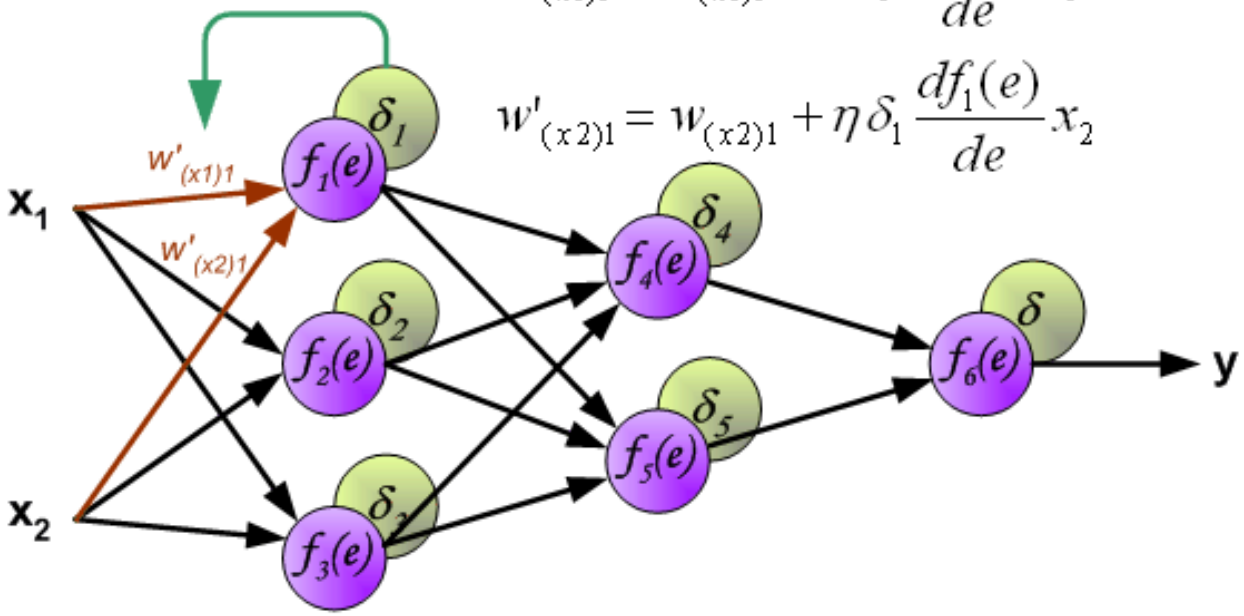


Learning Algorithm: Backpropagation

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$

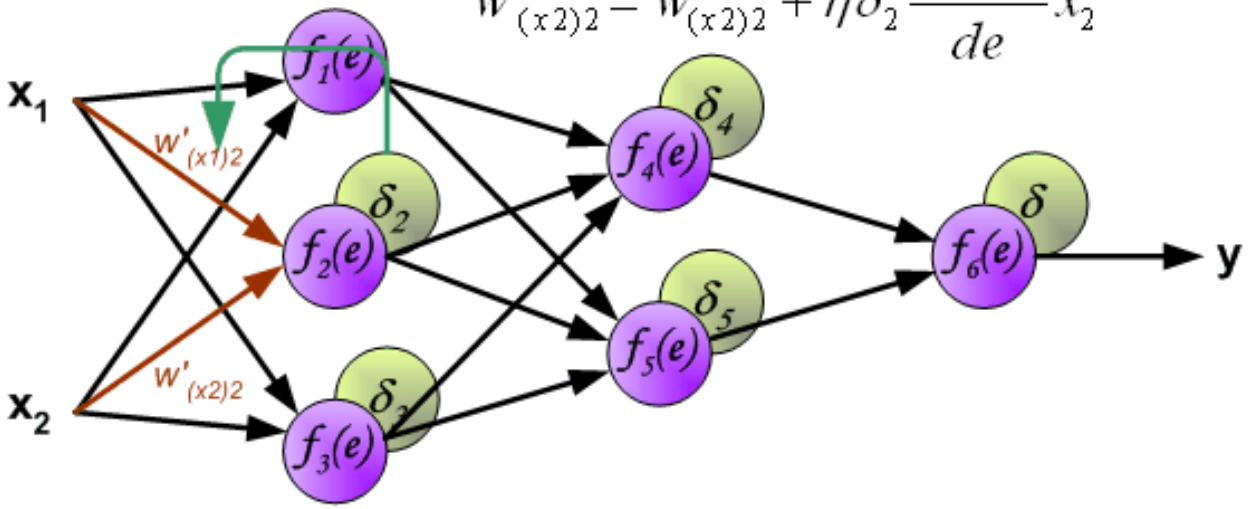


Learning Algorithm: Backpropagation

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$

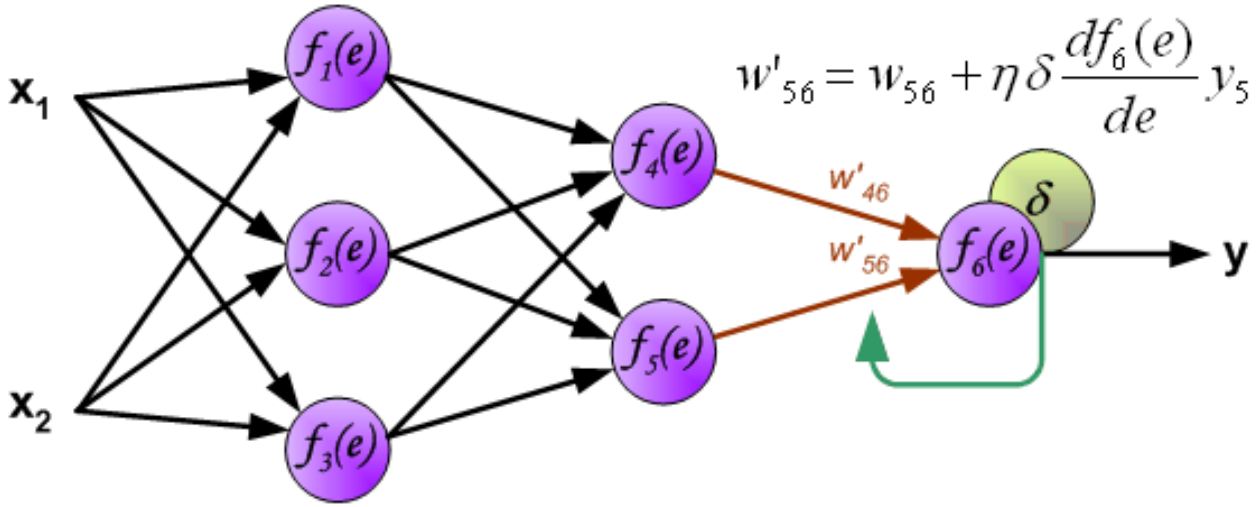


Learning Algorithm: Backpropagation

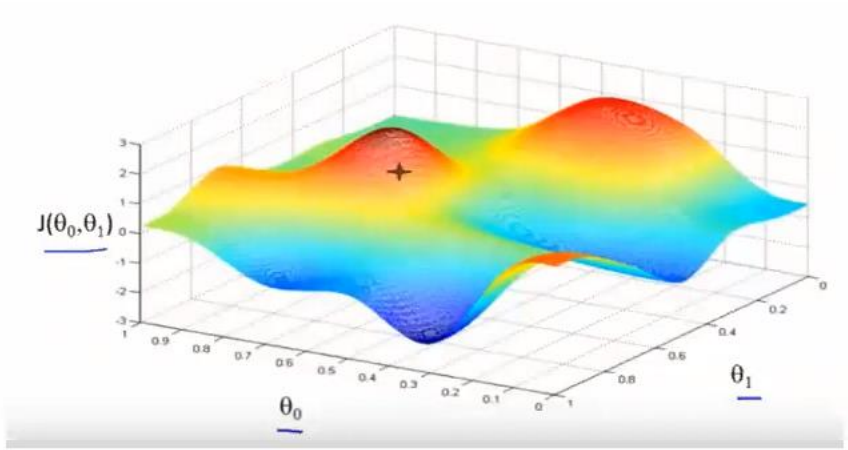
When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

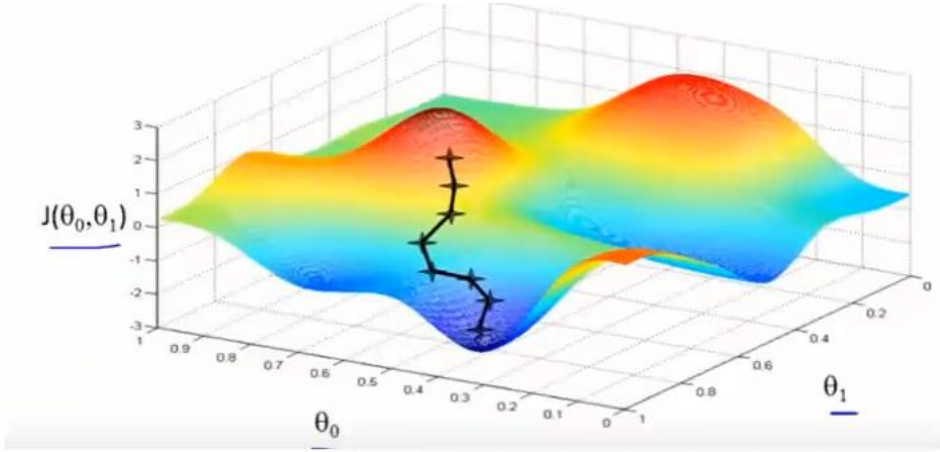
$$w'_{46} = w_{46} + \eta \delta \frac{df_6(e)}{de} y_4$$

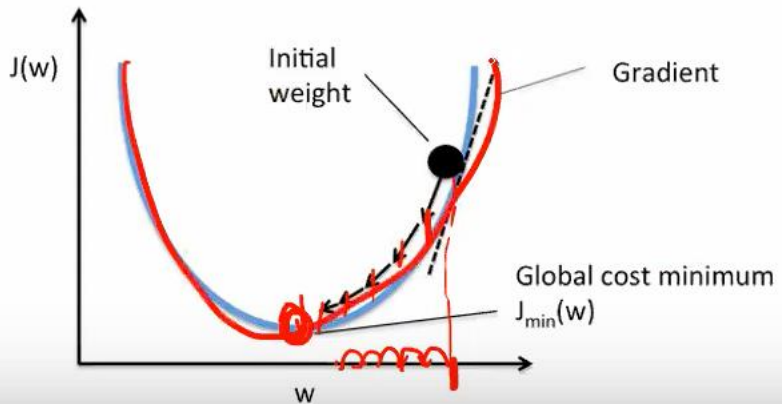
$$w'_{56} = w_{56} + \eta \delta \frac{df_6(e)}{de} y_5$$

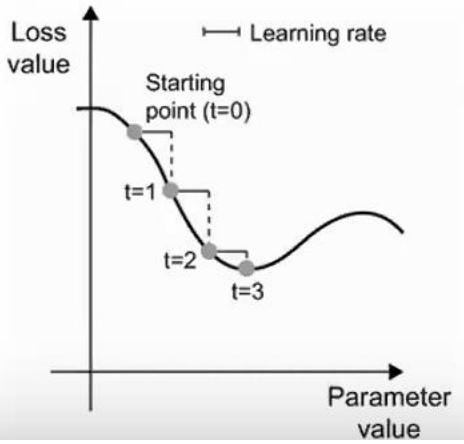


Challenges to moving down the hill

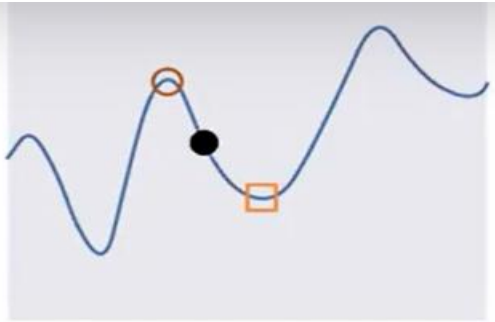




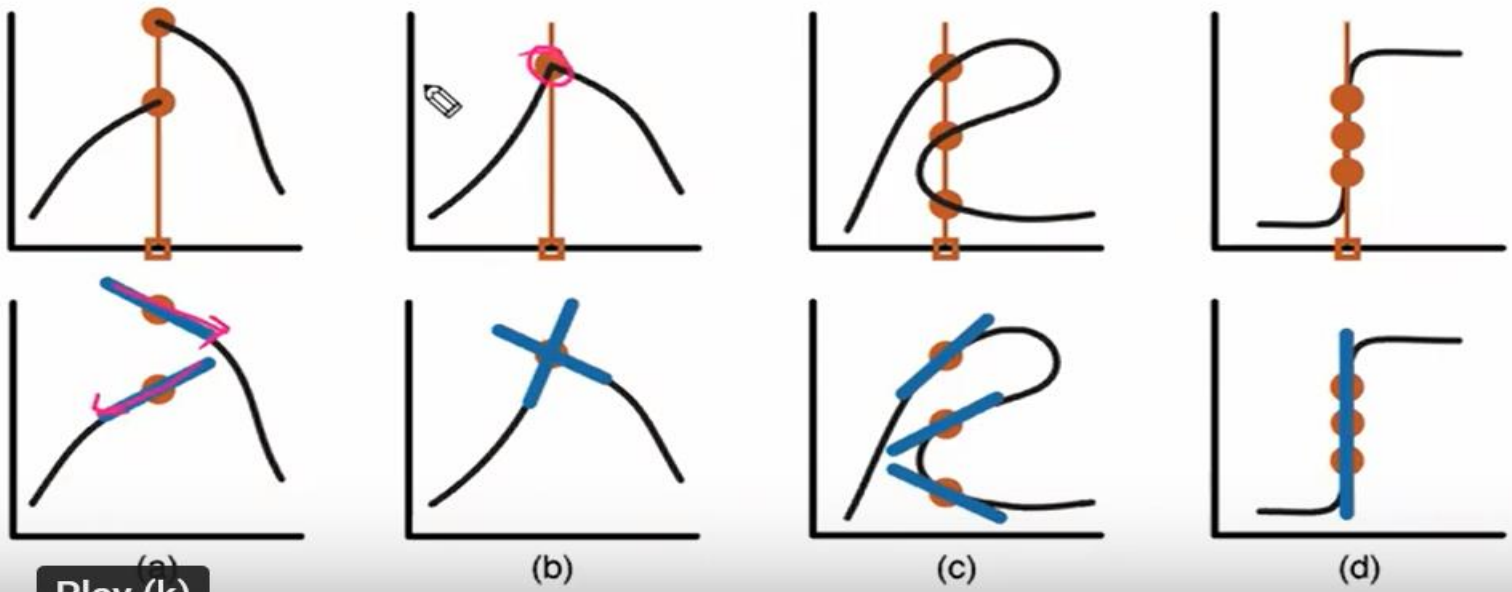




Consider the neighbourhood around the point selected

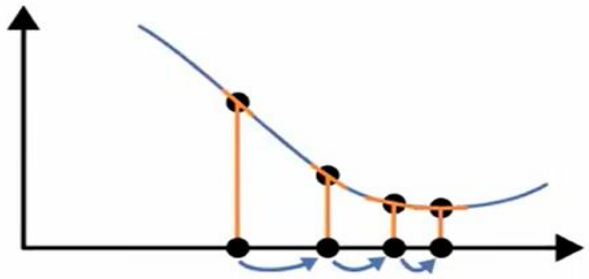


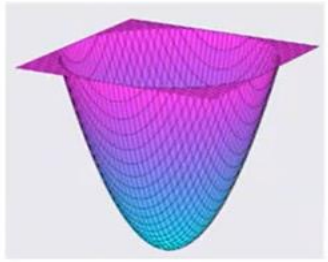
At each point on the curve, we can draw a line whose slope is given by the shape of the curve at that point. This is the tangent line.

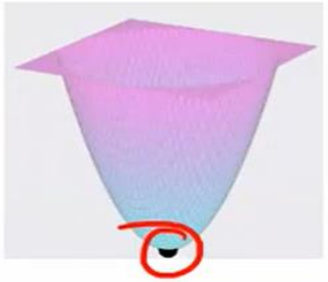


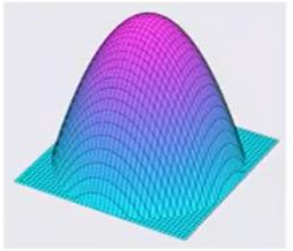
Play (k)

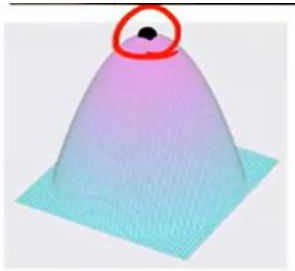
You should not stuck to saddle point or plateau

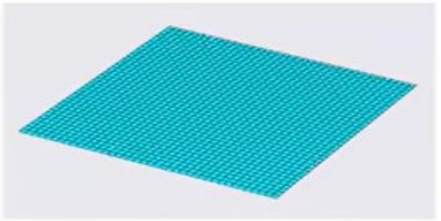


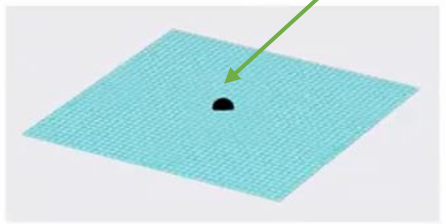


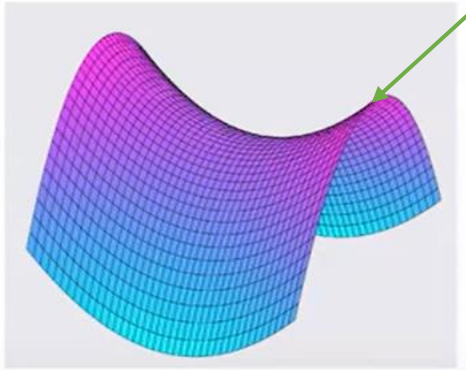


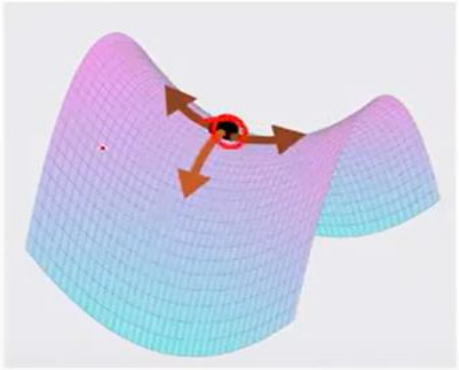






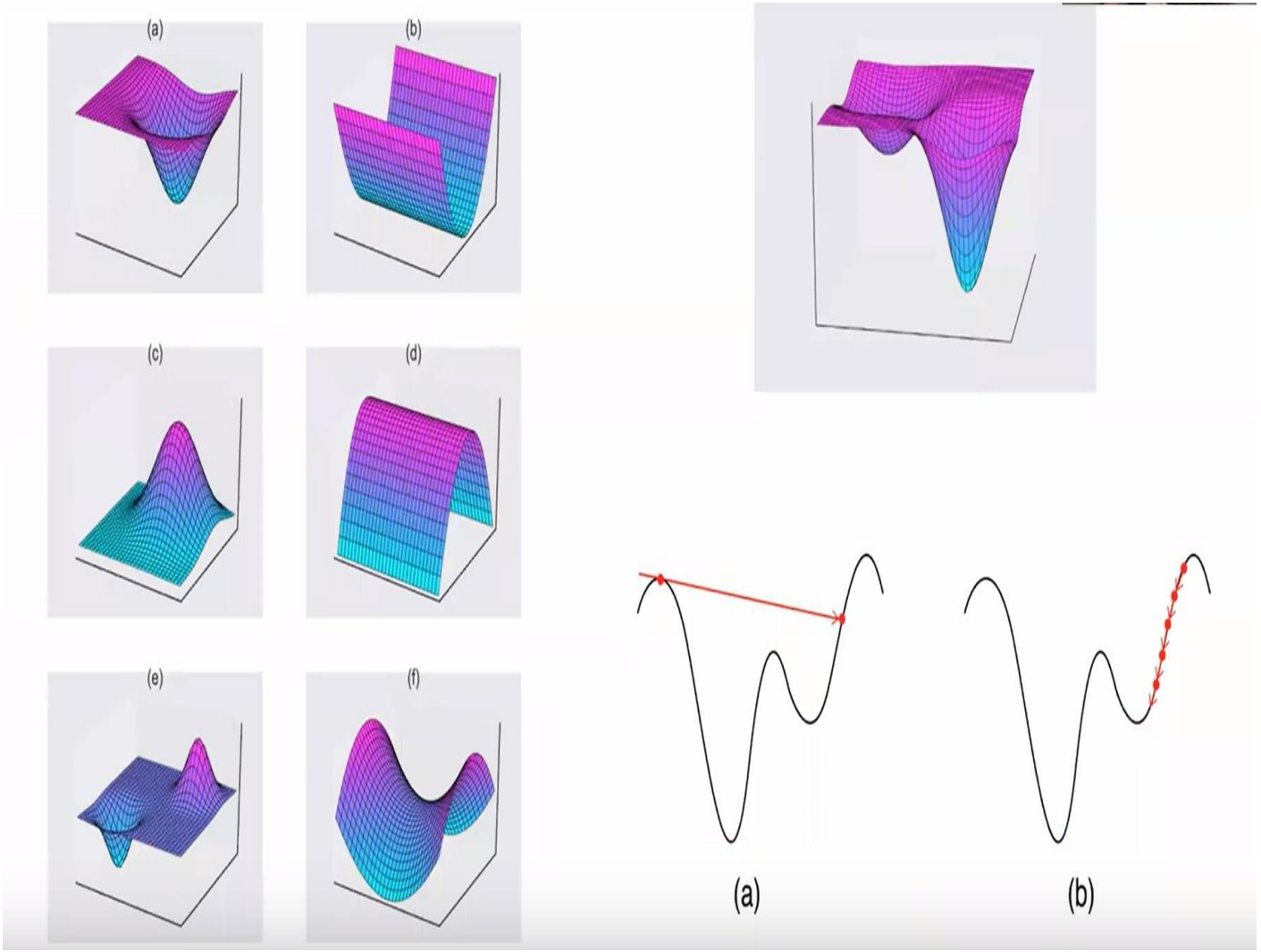






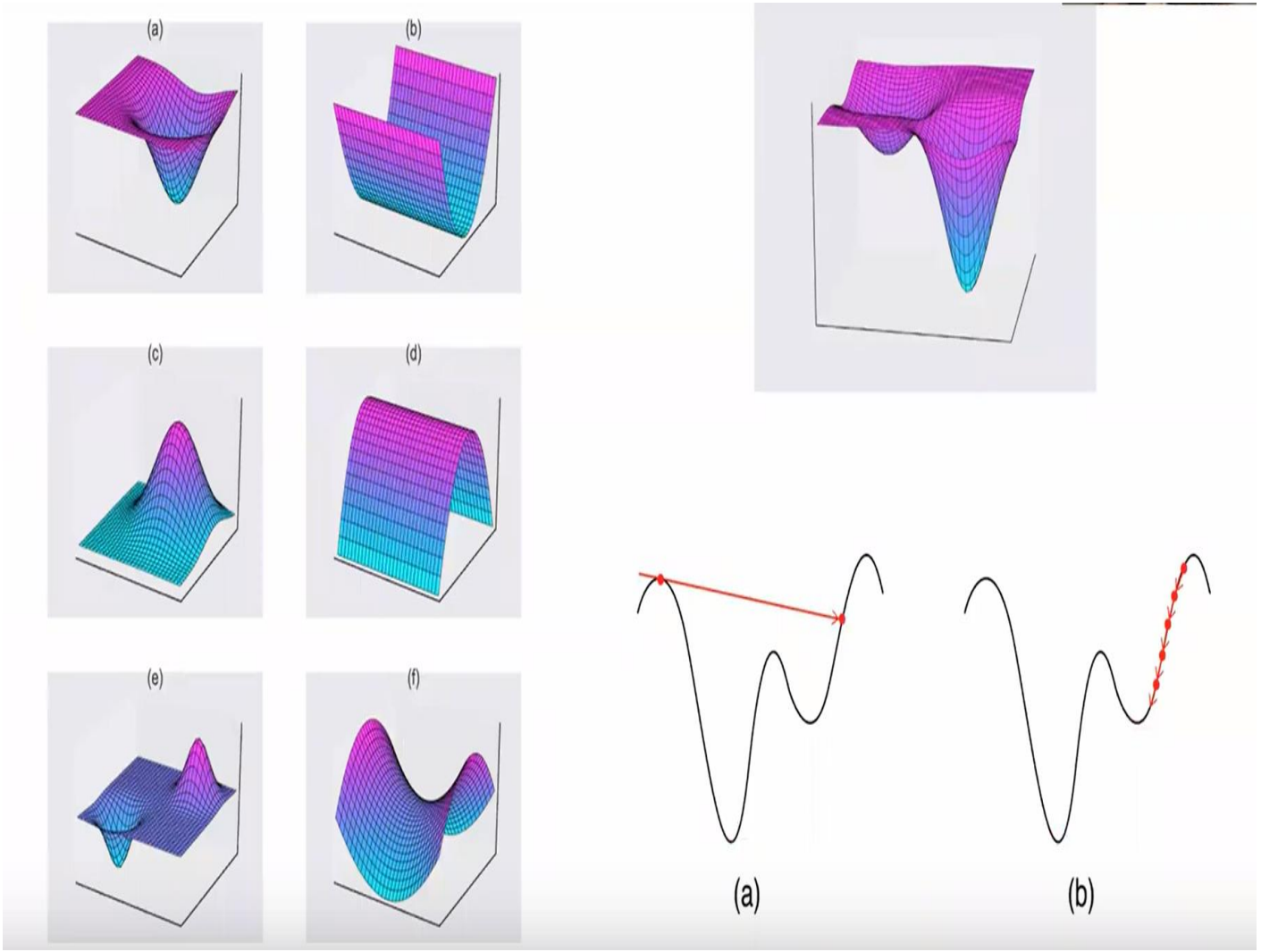
Exploding gradient

- Let us assume you are giving training to the model,wights are selected greater than one
- W1=1.1 and w2=1.3
- These wights are going multiply and added resultant weight will be 1.4 in second layer.
- It will be 1.8 in next layer .
- Weight will increase and will never reach to global minima.
- This is called exploding gradient



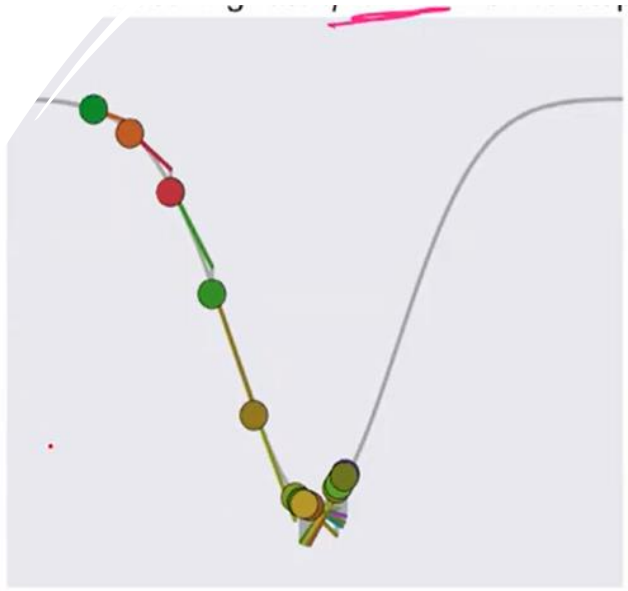
Vanishing gradient

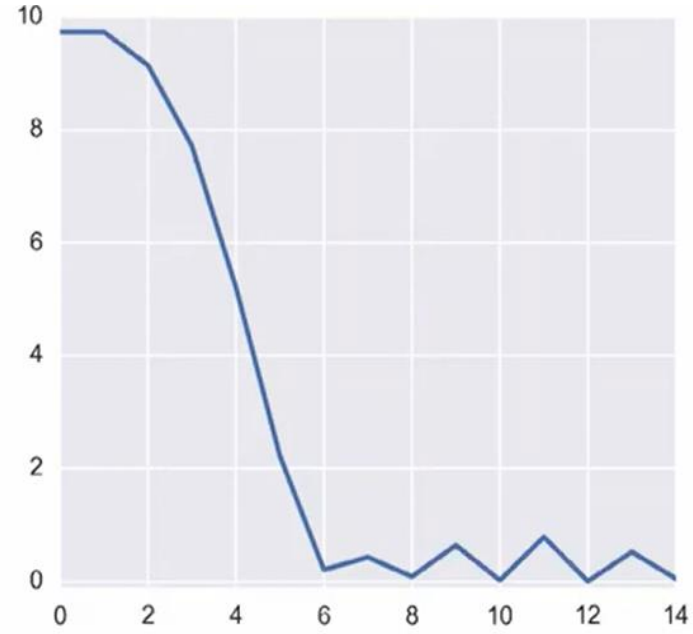
- Let us assume you are giving training to the model,wights are selected less than one
- W1=0.7and w2=0.8
- These wights are going multiply and added resultant weight will be 0.72 in second layer which less than earlier
- It will reduce further in next layer .
- Weight will decrease and will never reach to global minima.
- This is called vanishing gradient problem.



Challenges of keeping learning rate constant

- At flat surface weight will reduce moderately
- But at slope it will change drastically
- It will never converge to minima
- It will be having bouncing around the problem

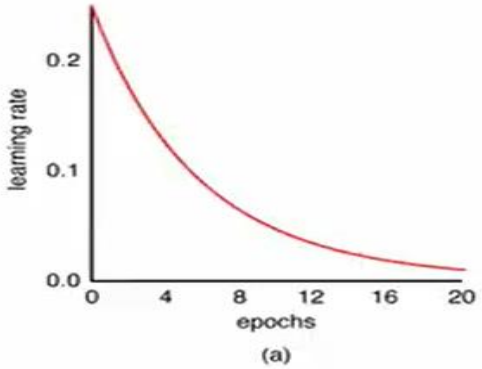


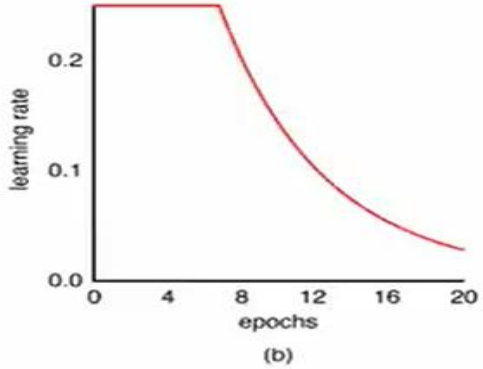


Bouncing around the bottom of the bowl when we use constant learning rate

Challenges in change in learning rate(decay parameters)

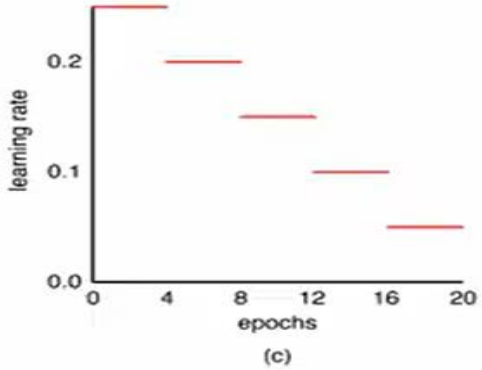
exponential approach

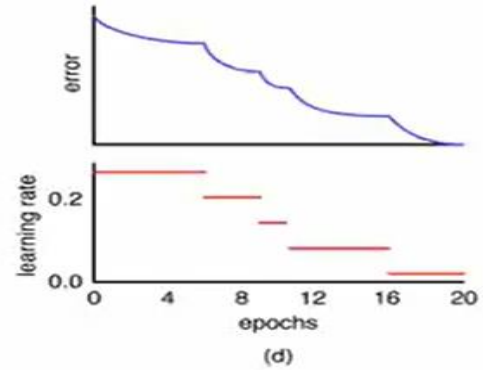




variant of exponential approach

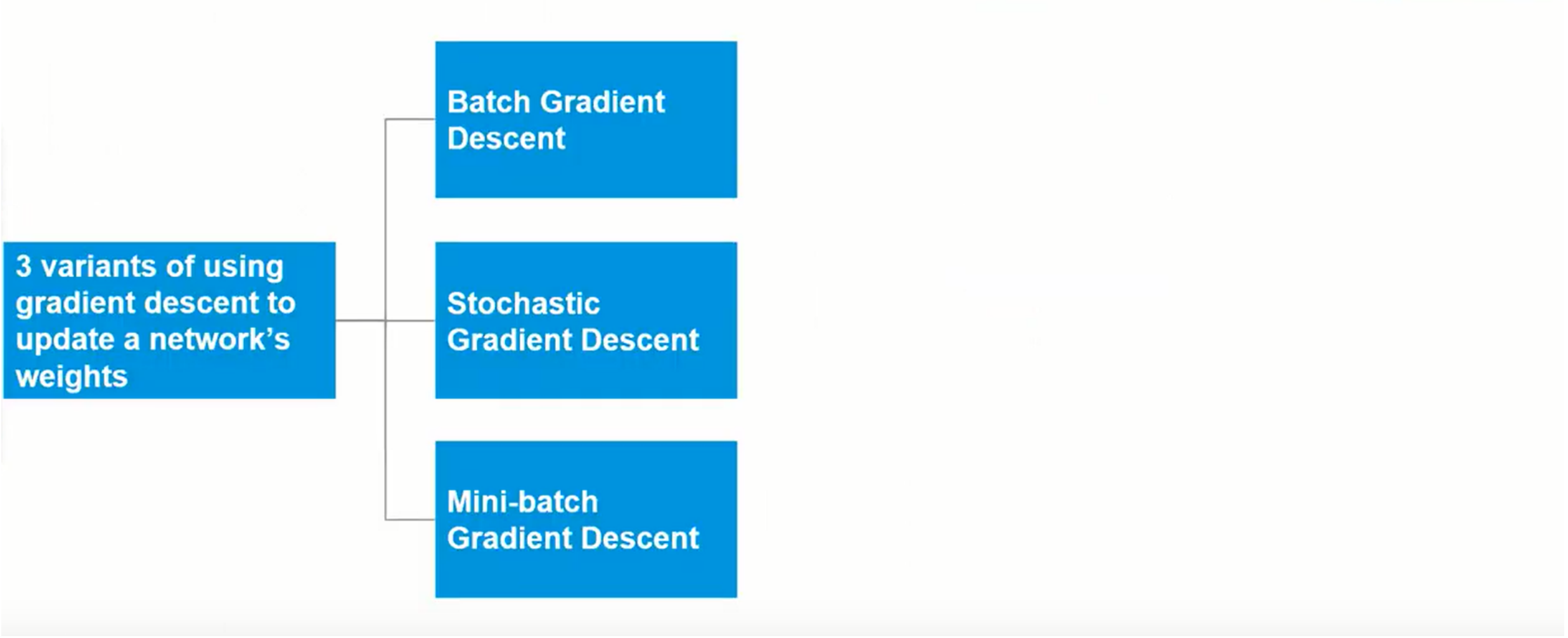
step approach



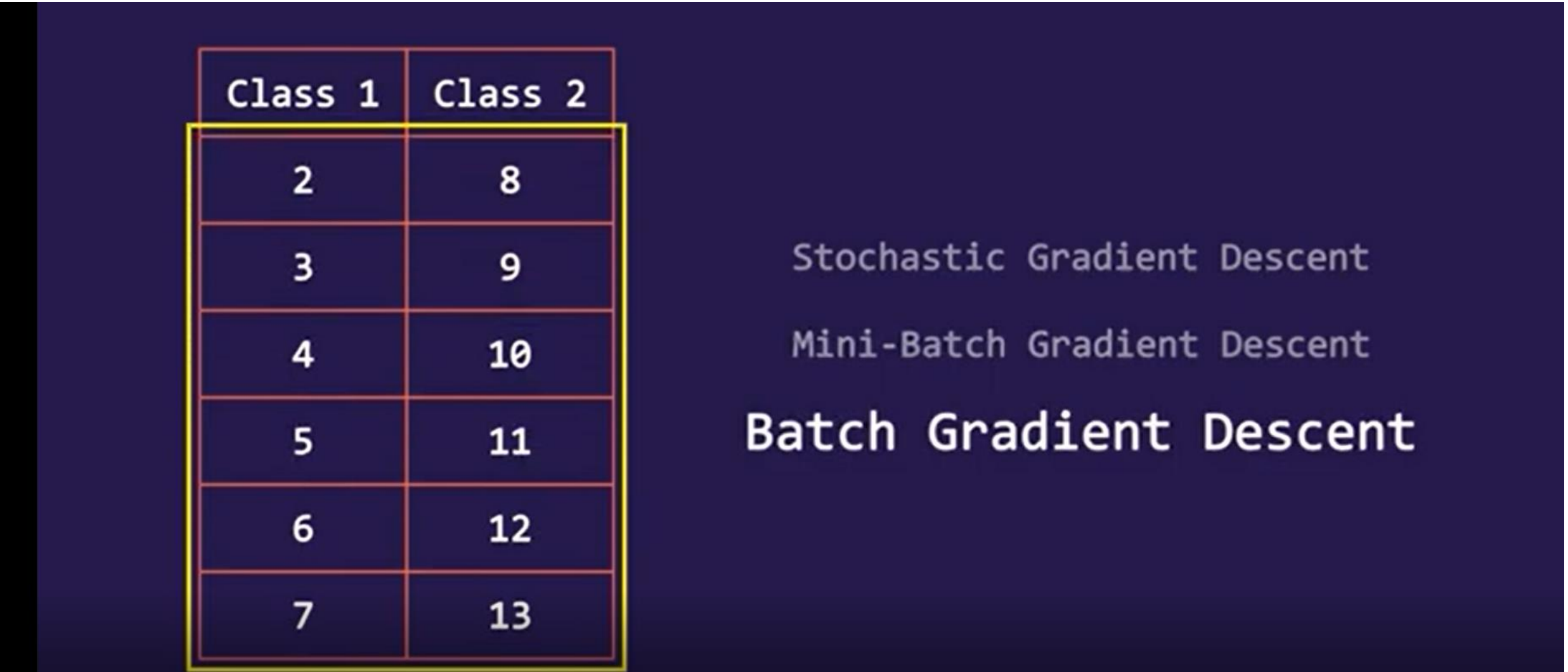


Bold drive approach

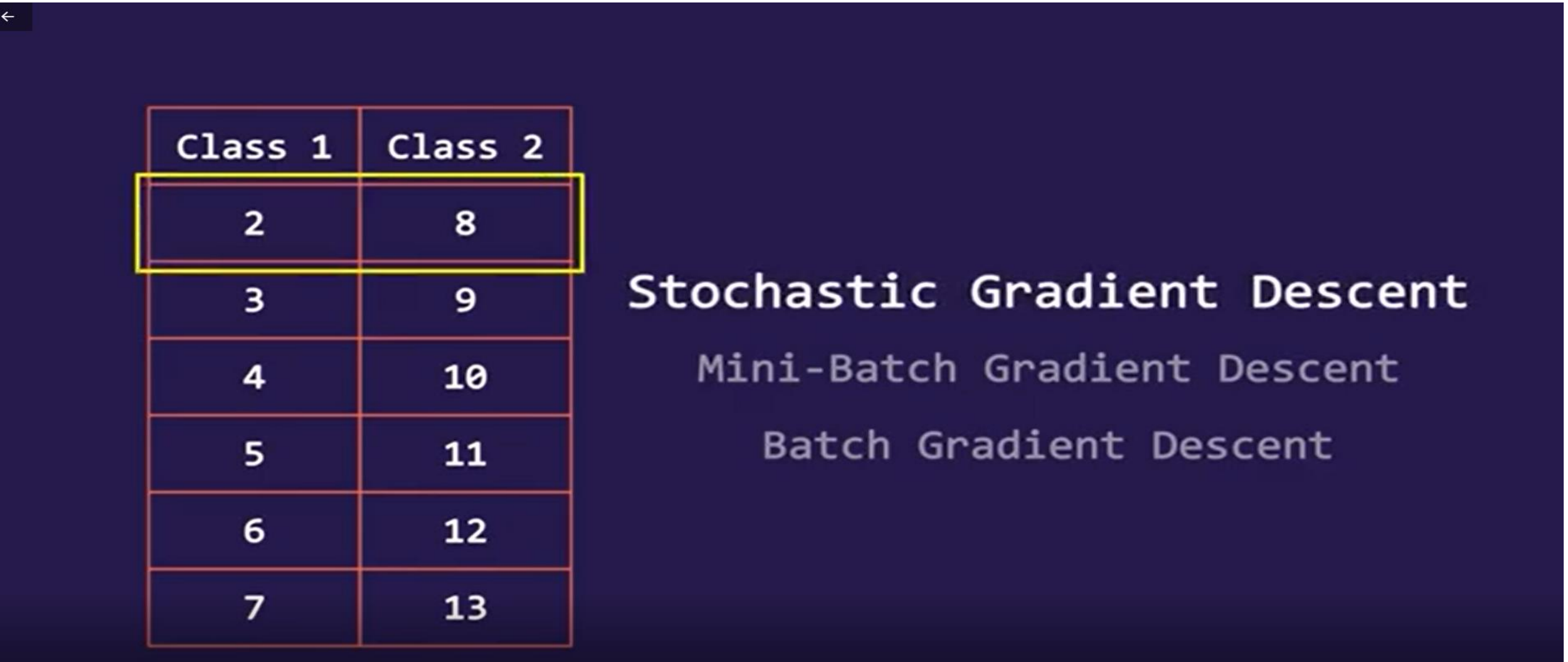
To address these challenges



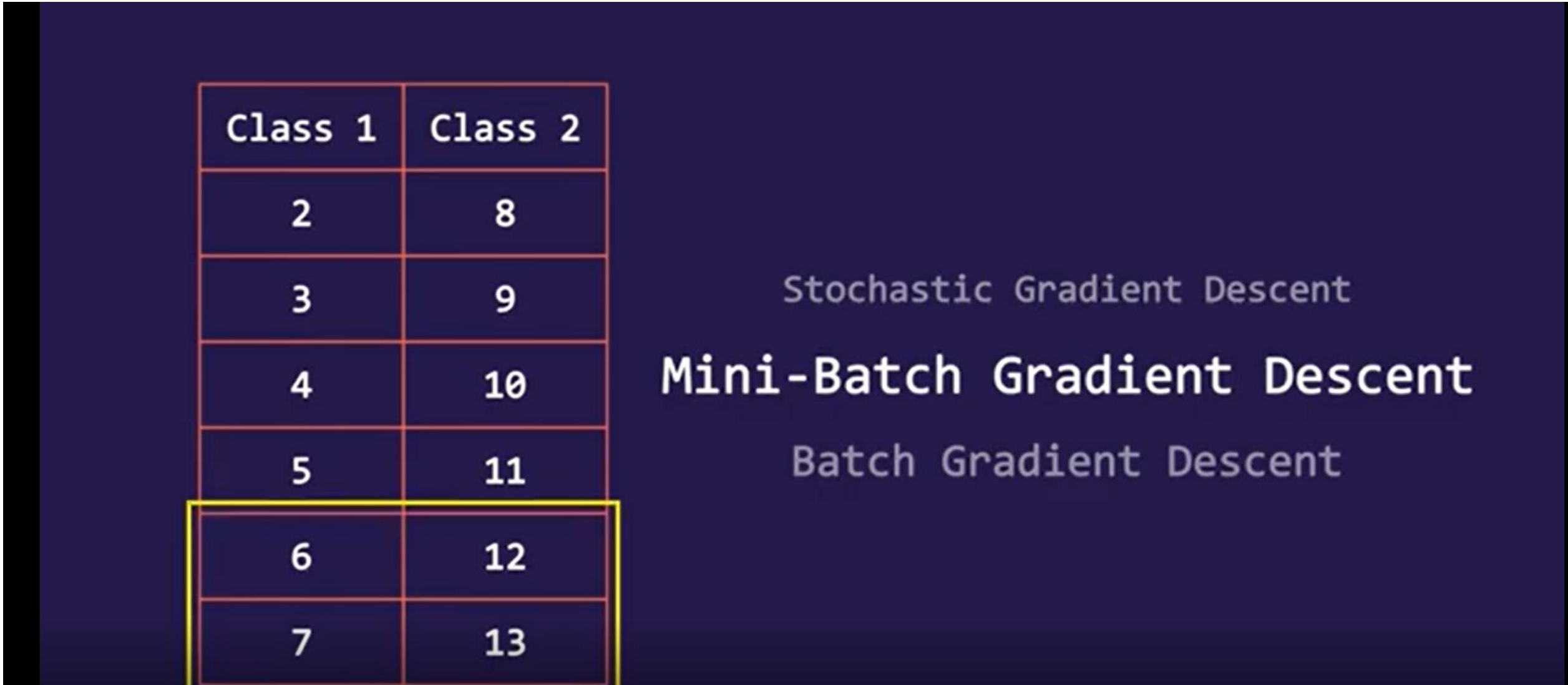
In batch Gradient Descent entire data used to adjust weight and bias



Weights and bias are adjusted considering single row at a time



Set of rows are used to update wights and bias.This is most widely used technique



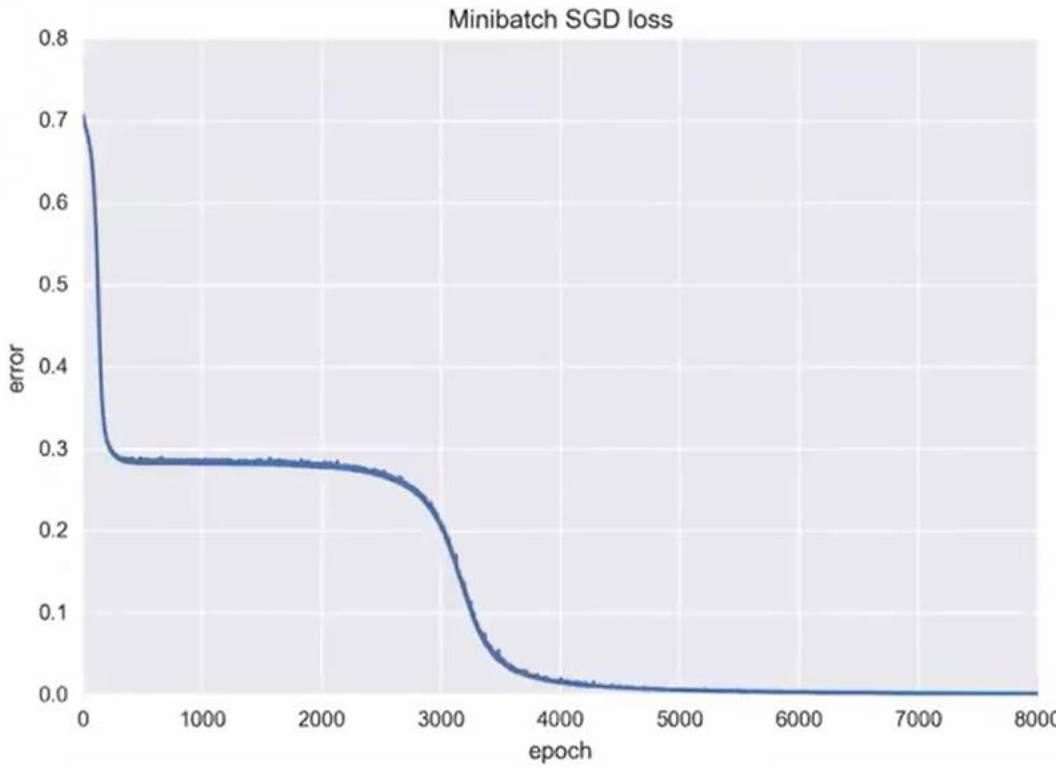
Mini batch gradient

- We can find a nice middle ground between the extremes of batch gradient descent, which updates once per epoch, and stochastic gradient descent, which updates after every sample. This compromise is called mini-batch gradient descent

- We update the weights after some fixed number of samples have been evaluated. This number is almost always considerably smaller than the batch size

- The mini-batch size is frequently a power of 2 between about 32 and 256, and often chosen to fully use the parallel capabilities of our GPU, if we have one. But that's just for speed purposes. We can use any size of mini-batch that we like

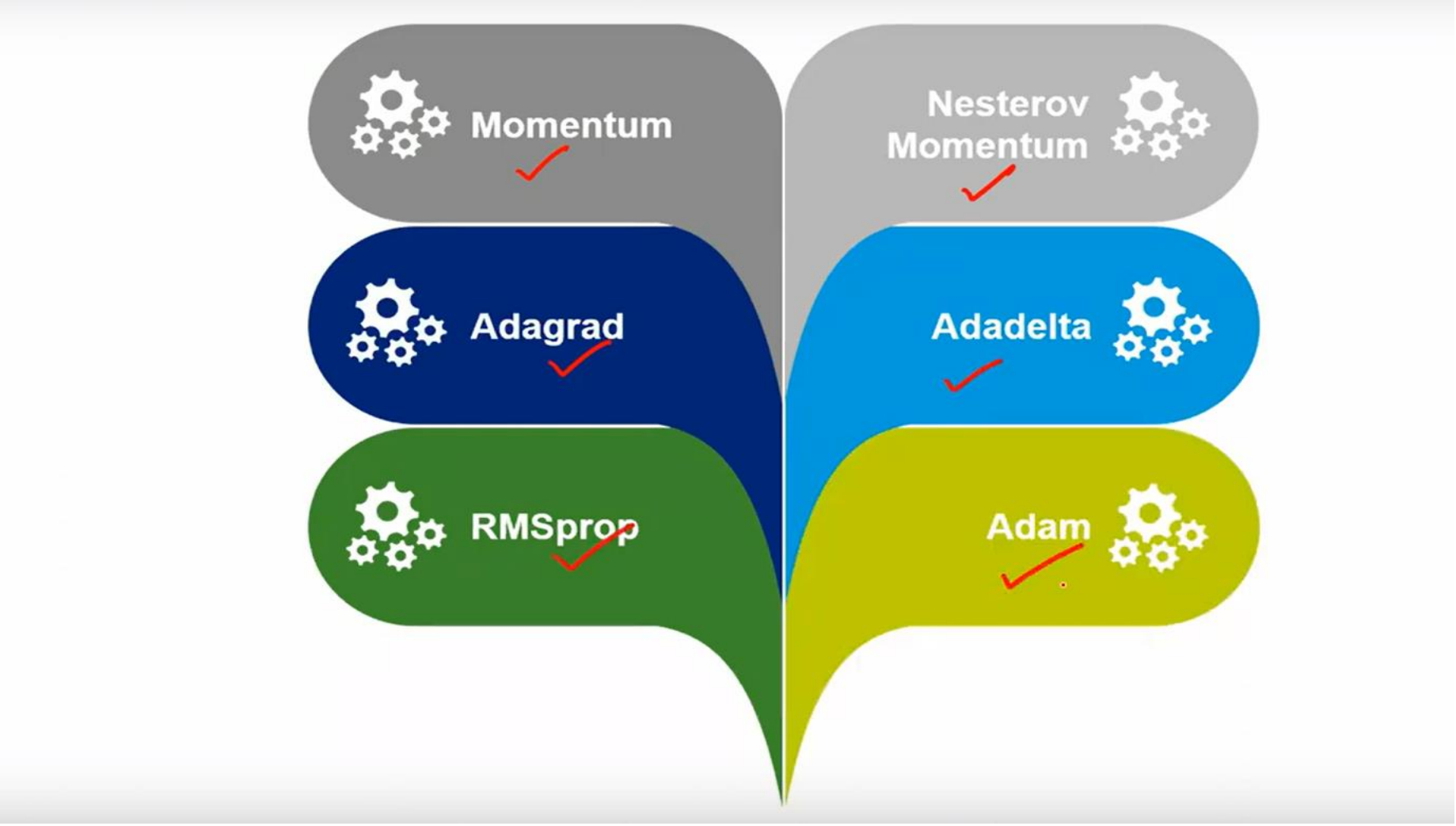
- How many updates did mini-batch SGD perform?



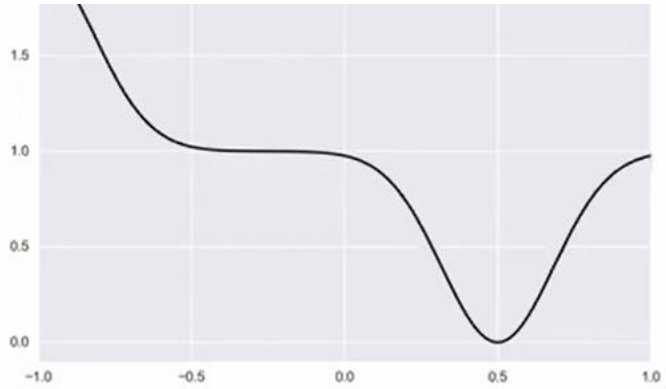
Mini batch SGD has got following challenges

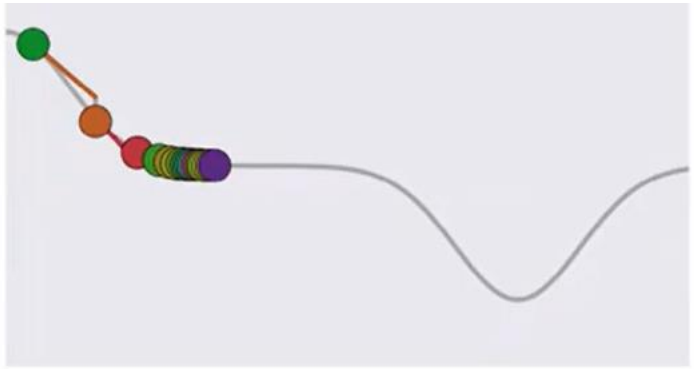
- We have to specify what value of the learning rate η we want to use, and that's notoriously hard to pick ahead of time
- A value that's too low can result in long learning times and getting stuck in shallow local minima
- A value that's too high can cause us to overshoot deep local minima, and then get stuck at bouncing around inside a minimum when we do find it
- If we try to avoid the problem by using a decay schedule to change η over time, we still have to pick that schedule and its parameters
- We have to pick the size of the mini-batch
- We have been updating all the weights with a "one update rate fits all" approach

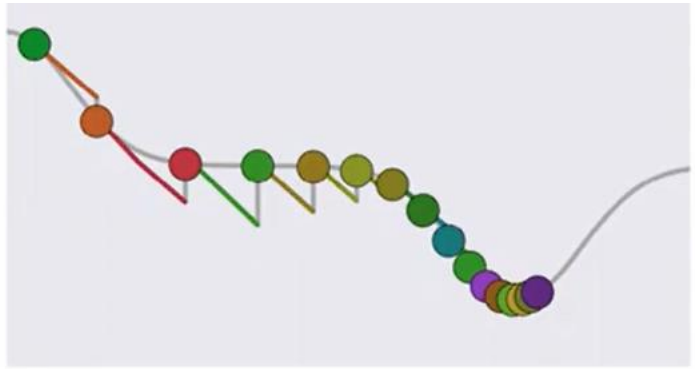
To overcome the problems of minibatch various algorithms are used.

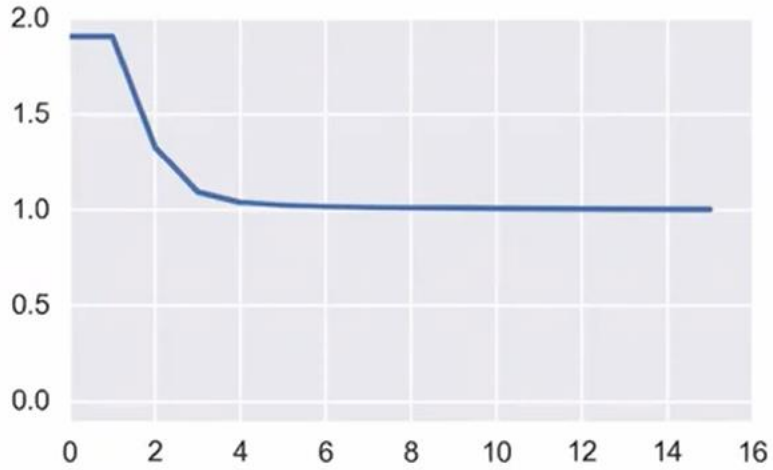


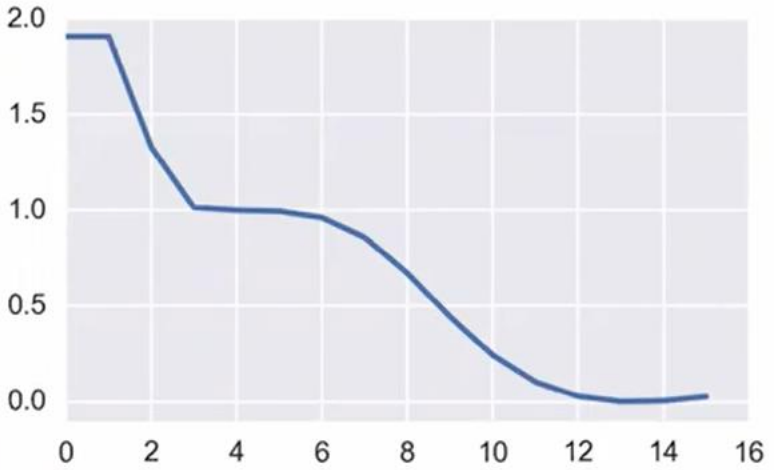
momentum


1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0



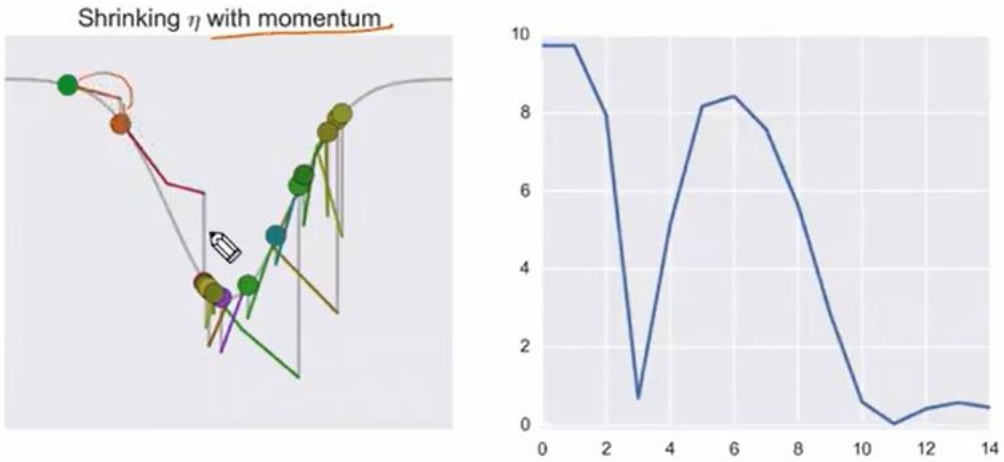



2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10
12
14
16


2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10
12
14
16

Nesterov momentum

While Momentum let us reach into the past for information to help us train, Nesterov Momentum let's us reach into the future



Adagrad

- **Adagrad stands for an adaptive gradient**. Adagrad is an effective algorithm for gradient-based optimization.
- It adapts the learning rate to the parameters, using low learning rates for parameters associated with frequently occurring features, and using high learning rates for parameters associated with rare features.
- Therefore, it is well suited when dealing with sparse data..

Adagrad

- The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms.
- This is because it uses different learning rates for each iteration.
- The change in learning rate depends upon the difference in the parameters during training.
- The more the parameters get changed, the more minor the learning rate changes.
- This modification is highly beneficial because real-world datasets contain sparse as well as dense features.
- So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

- Adagrad removes the need to manually tune the learning rate.
- There are mainly three problems that arise with the Adagrad algorithm.
- The learning rate is monotonically decreasing.
- The learning rate in the late training period is very small.
- it needs to be set by doing the initial global learning rate.

Ada delta

- Adadelta is an extension of Adagrad and also tries to reduce Adagrad's rate of learning, excessively.

- It does this by limiting the gradient window that has been exceeded to a certain size ***w***. Running average at time ***t*** then depends on the previous average and the current gradient.
- In Adadelta, we don't have to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

- **Adam Optimizer in Deep Learning**
- Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.
- The name “Adam” is derived from “adaptive moment estimation,” highlighting its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.

Adagrad,Adadelta and Adam