KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE

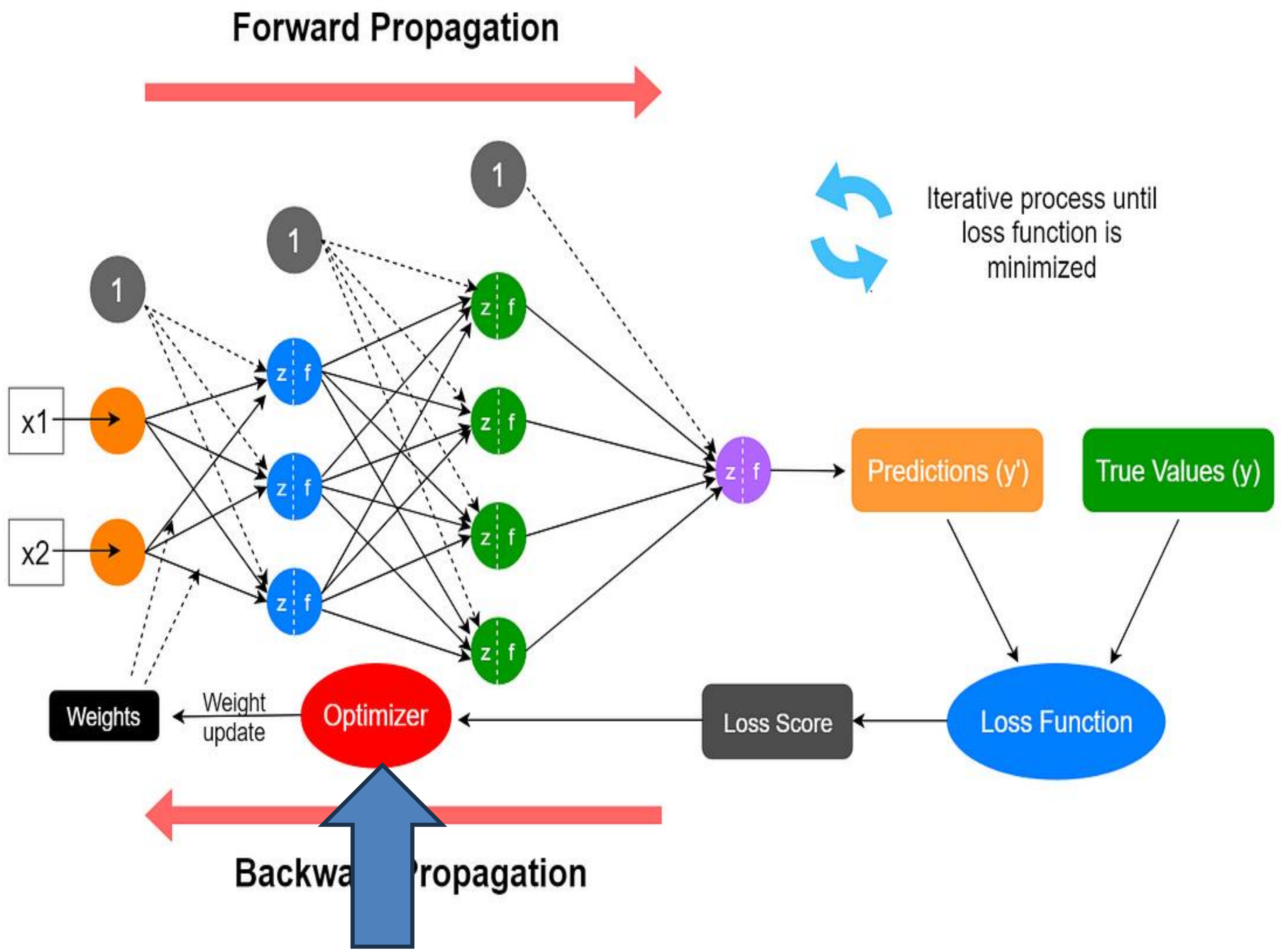
Narayanaguda, Hyderabad.

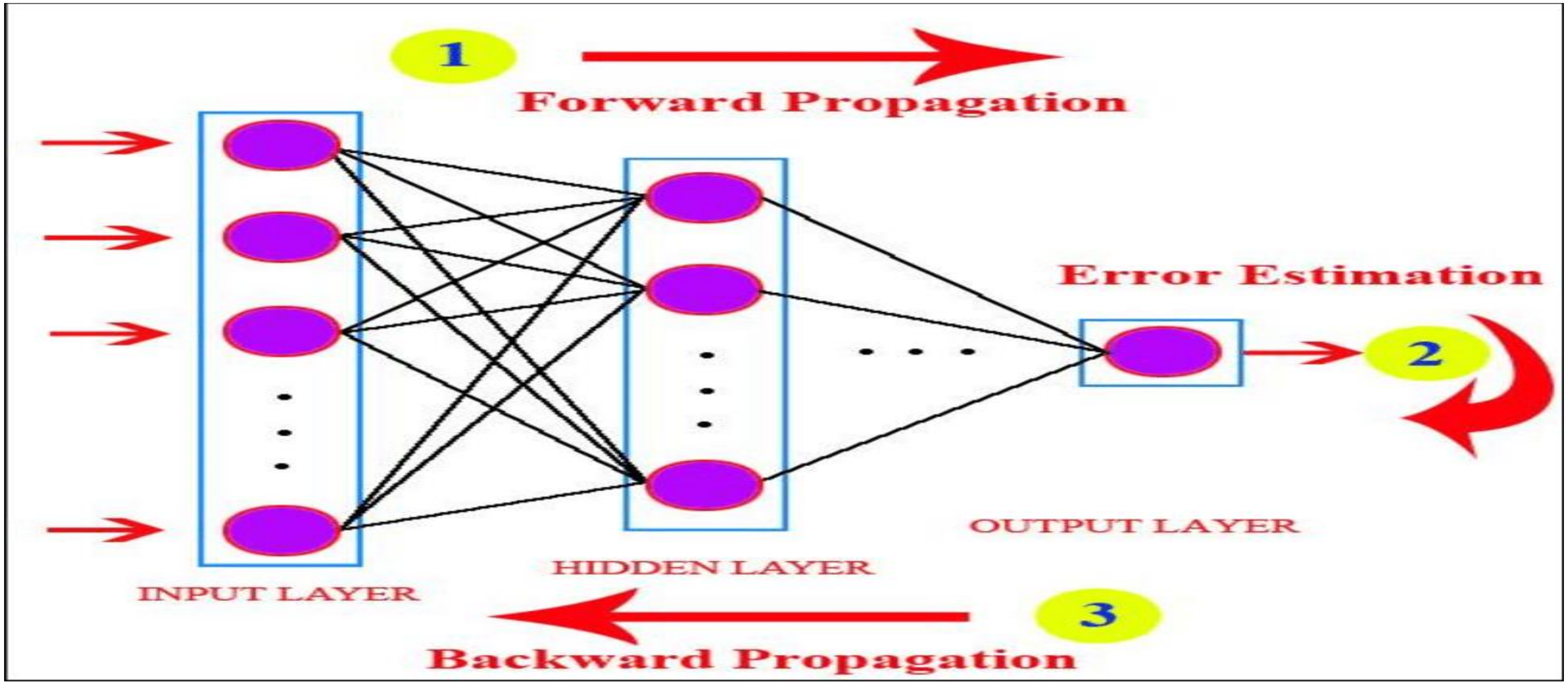
Deep Learning Optimizers

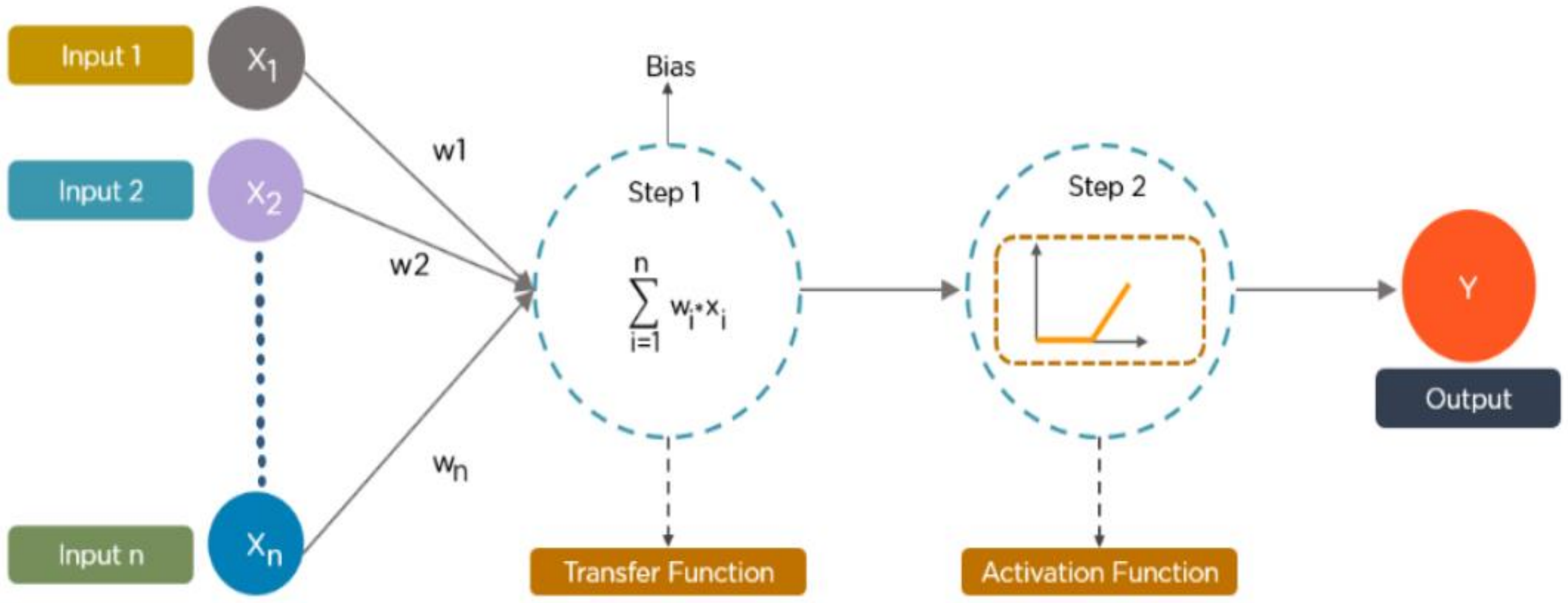
28-09-2024

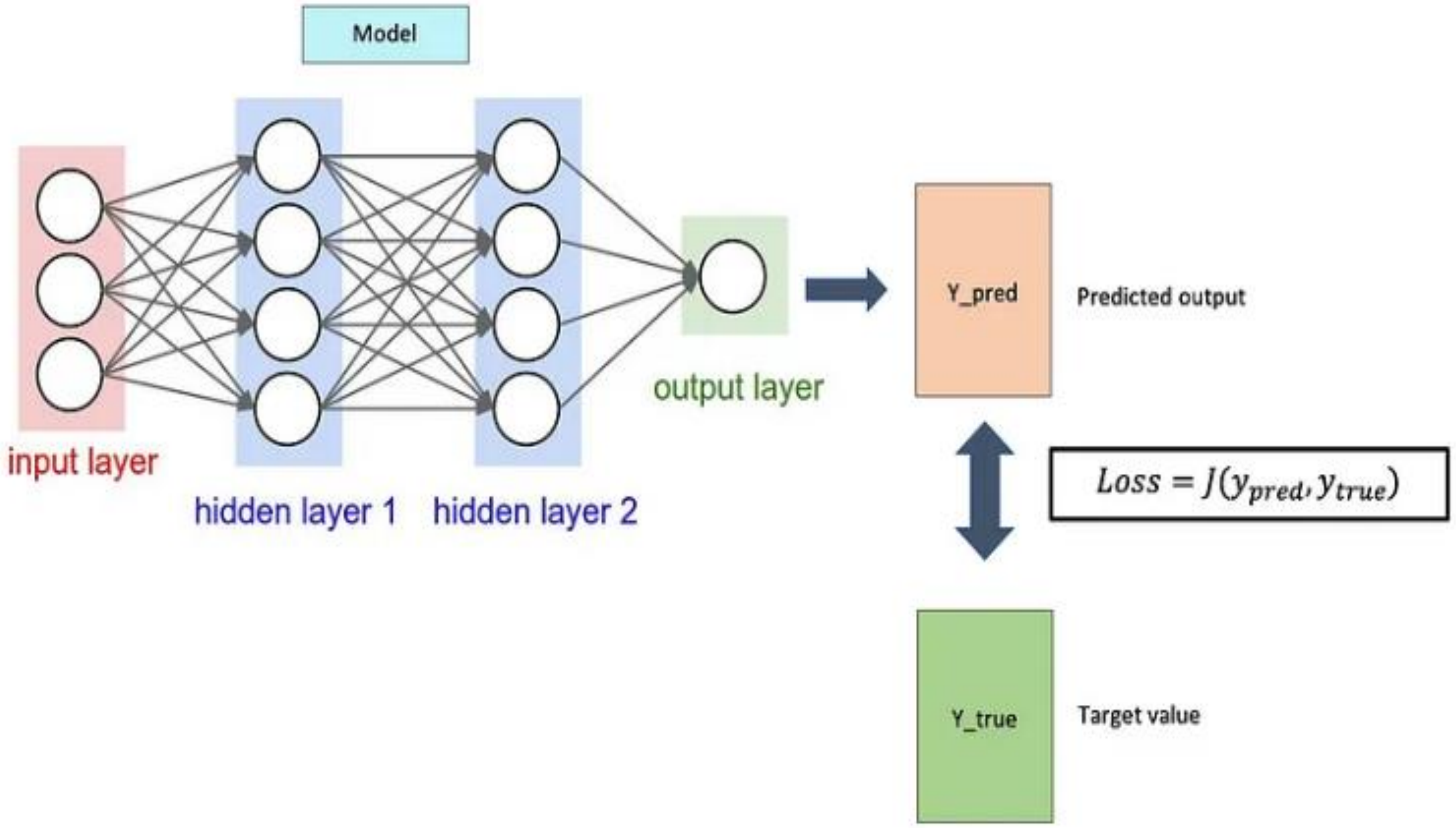
BY

ASHA


Forward Propagation
1
1
1
Iterative process until loss function is minimized
x1
x2
z f
z f
z f
z f
z f
z f
z f
z f
Predictions (y')
True Values (y)
Weights
Weight update
Optimizer
Loss Score
Loss Function
Backwa ropagation

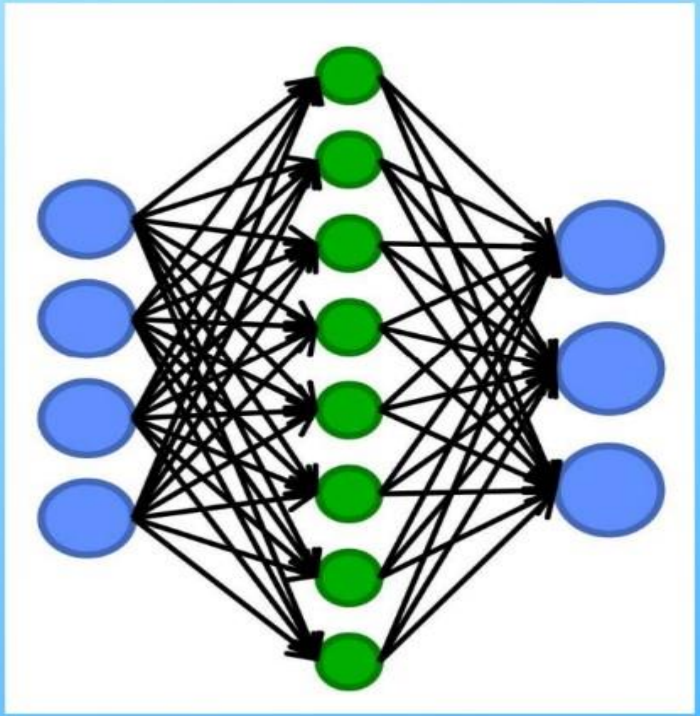

1
Forward Propagation
Error Estimation
2
INPUT LAYER
HIDDEN LAYER
OUTPUT LAYER
3
Backward Propagation


Input 1
Input 2
Input n
x_1
x_2
x_n
w1
w2
w_n
Bias
Step 1
$\sum_{i=1}^{n} w_i * x_i$
Step 2
Y
Output
Transfer Function
Activation Function


Model
input layer
hidden layer 1
hidden layer 2
output layer
Y_pred
Predicted output
$Loss = J(y_{pred}, y_{true})$
Y_true
Target value

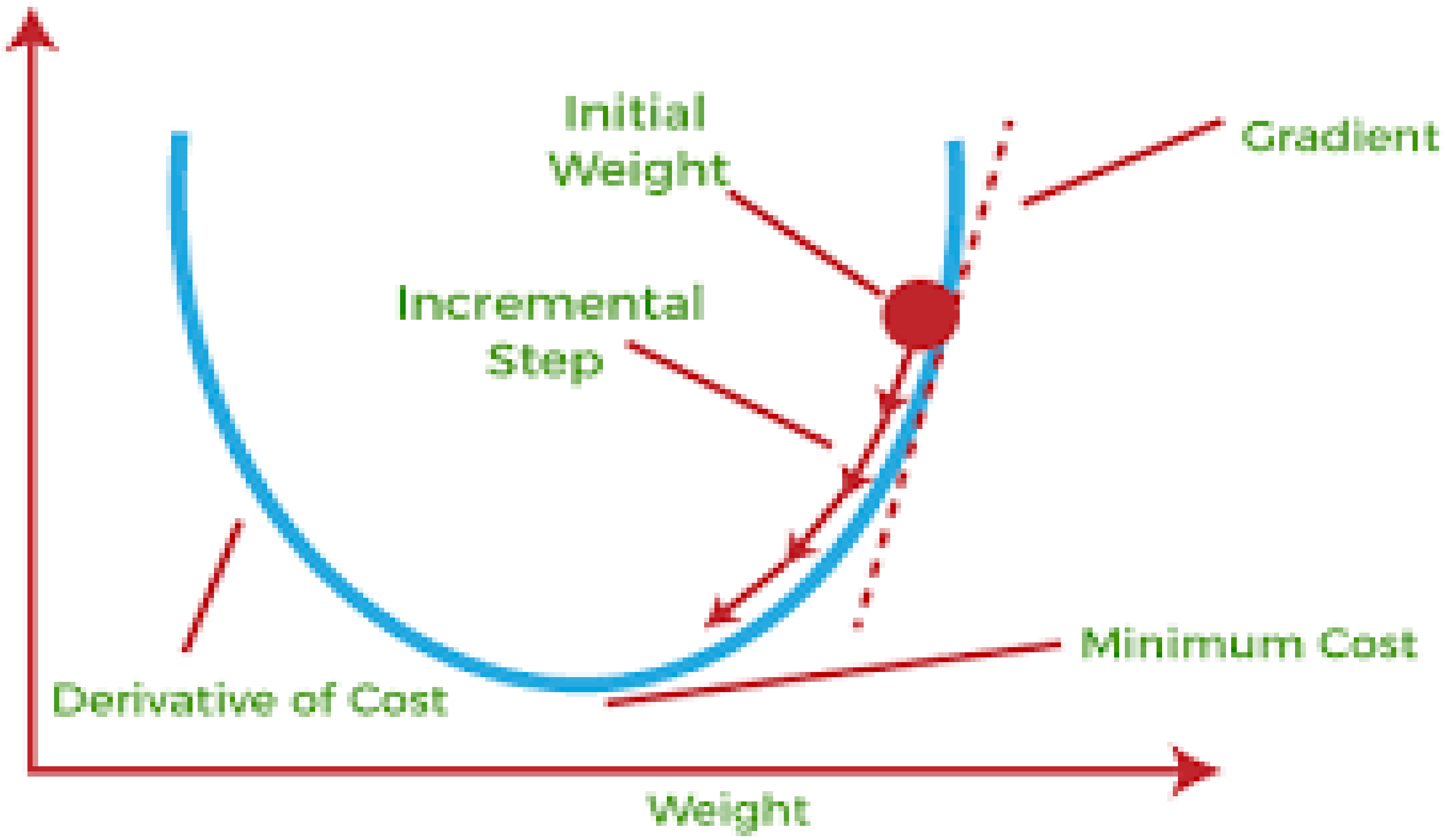
Backpropagation

Learning algorithms use backpropagation to:

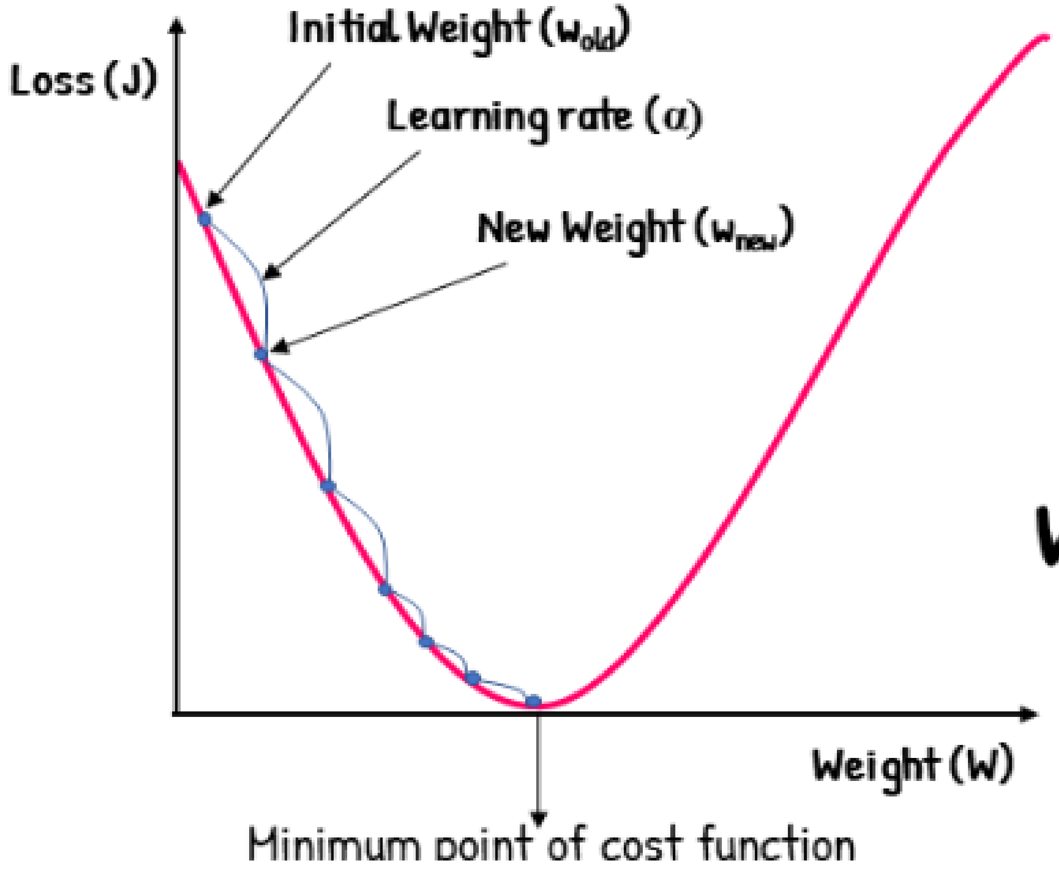


- ✓ Compute a gradient descent with respect to weights.
- ✓ Comparing outputs to desired system outputs.
- ✓ Adjust connection weights to narrow the difference between the two.

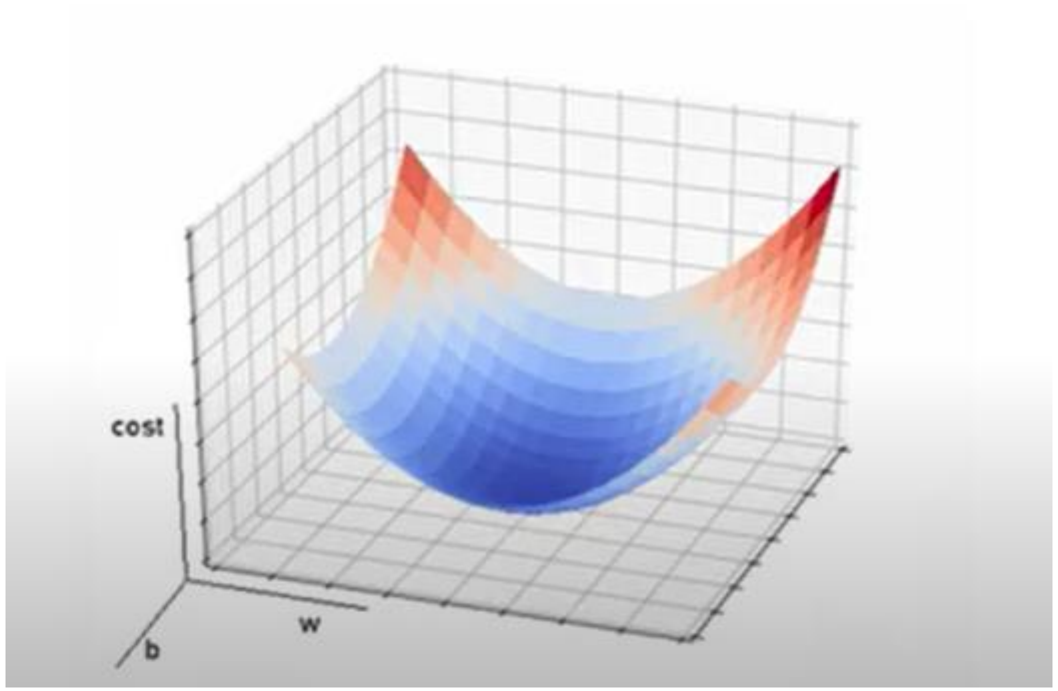
- **Gradient Descent** is the most common optimization algorithm in *machine learning* and *deep learning*.
- It is a first-order optimization algorithm. This means it only takes into account the first derivative when performing the updates on the parameters.
- On each iteration, we update the parameters in the opposite direction of the gradient of the cost function w.r.t the parameters where the gradient gives the direction of the steepest ascent.
- The size of the step we take on each iteration to reach the local minimum is determined by the learning rate α.
- Therefore, we follow the direction of the slope downhill until we reach a local minimum.

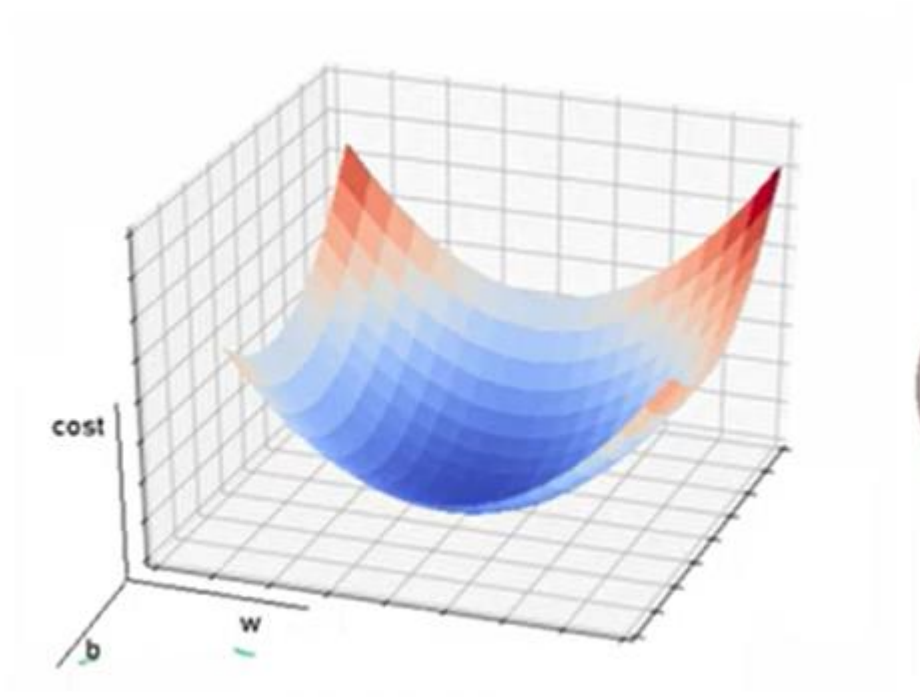

Initial Weight
Gradient
Incremental Step
Minimum Cost
Derivative of Cost
Weight

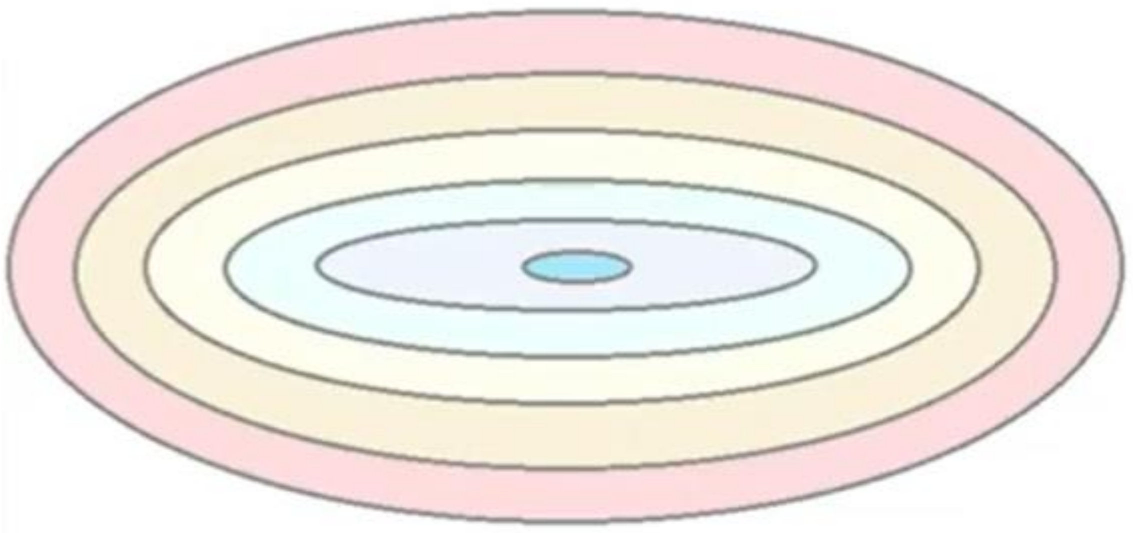
Gradient Descent



$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$


cost
w
b


cost
w
b

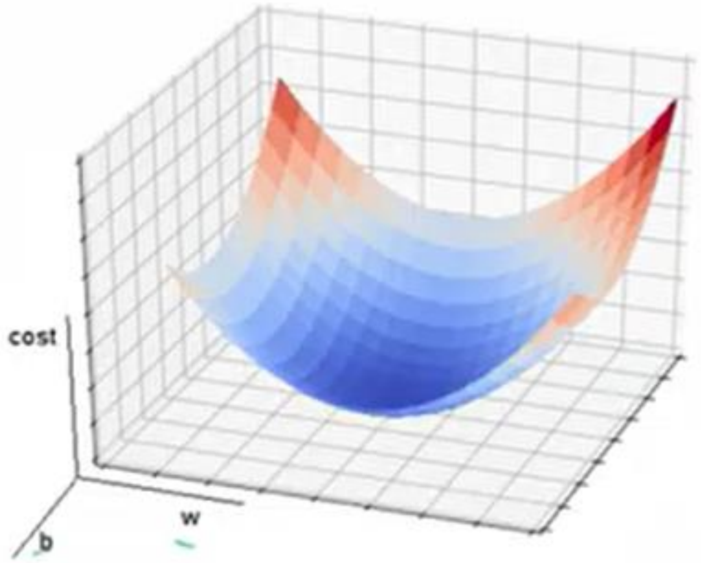


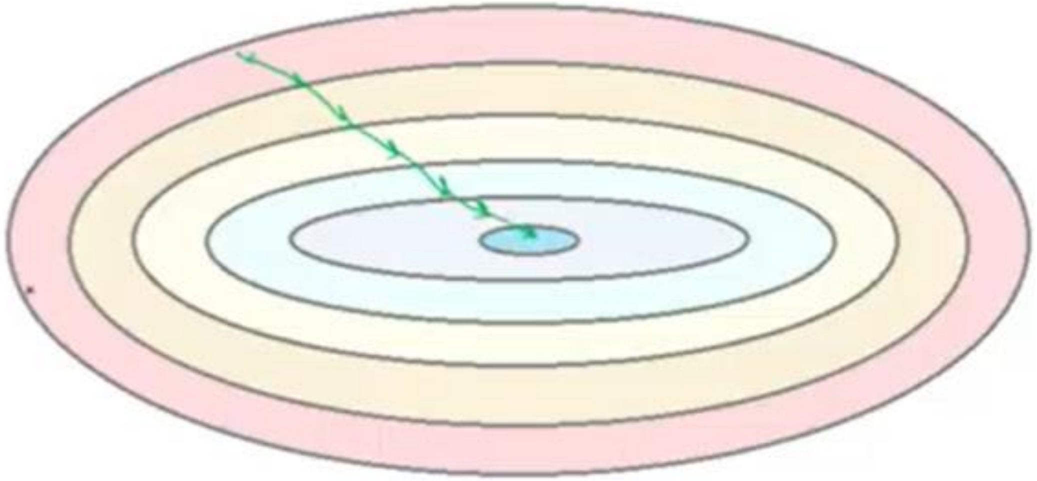
Types of GRADIENT DESCENT

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Batch Gradient Descent

- Batch Gradient Descent is a type of Gradient Descent which calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This can be computationally expensive and hence can be slow on very large datasets.


cost
w
b



The main advantage:

- It has straight trajectory towards the minimum and it is guaranteed to converge in theory to the global minimum.

The main disadvantages:

- Even though we can use vectorized implementation, it may still be slow to go over all examples especially when we have large datasets.
- Each step of learning happens after going over all examples where some examples may be redundant and don’t contribute much to the update.

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) is a type of Gradient Descent where the step size is typically much larger, leading to a lot more randomness in the descent down the hill. This randomness can help the algorithm jump out of local minima, finding the global minimum. SGD performs a parameter update for each training example, which is less computationally expensive than Batch Gradient Descent.

Mini-batch Gradient Descent

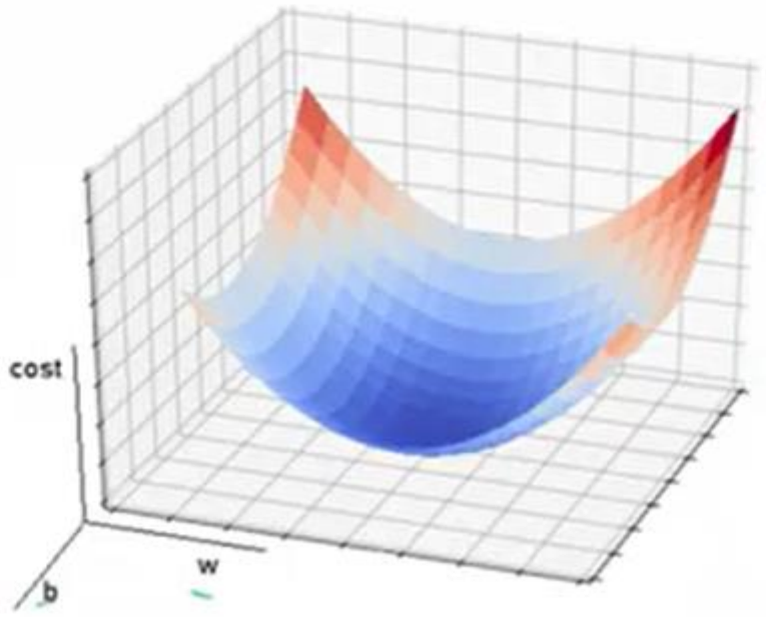
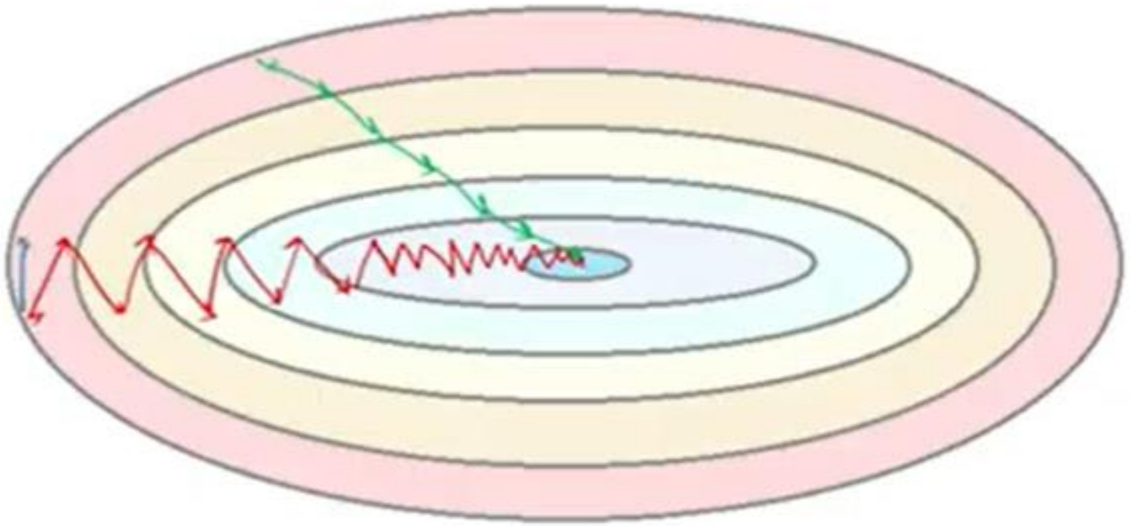
- Mini Batch Gradient Descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. It combines the advantages of Batch Gradient Descent and Stochastic Gradient Descent by performing an update for every batch of n training examples.

The main advantages:

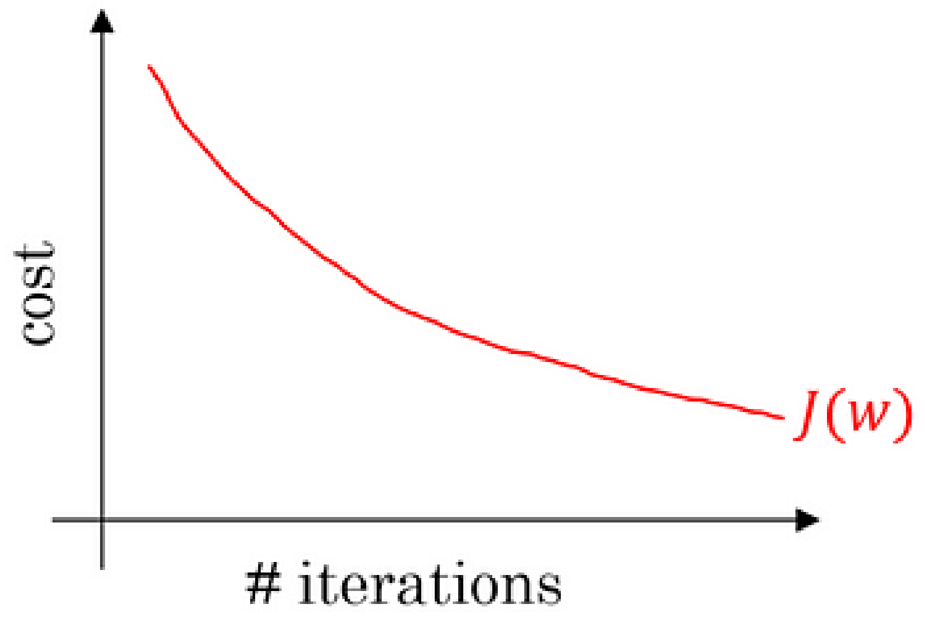
- Faster than Batch version because it goes through a lot less examples than Batch (all examples).

The main disadvantages:

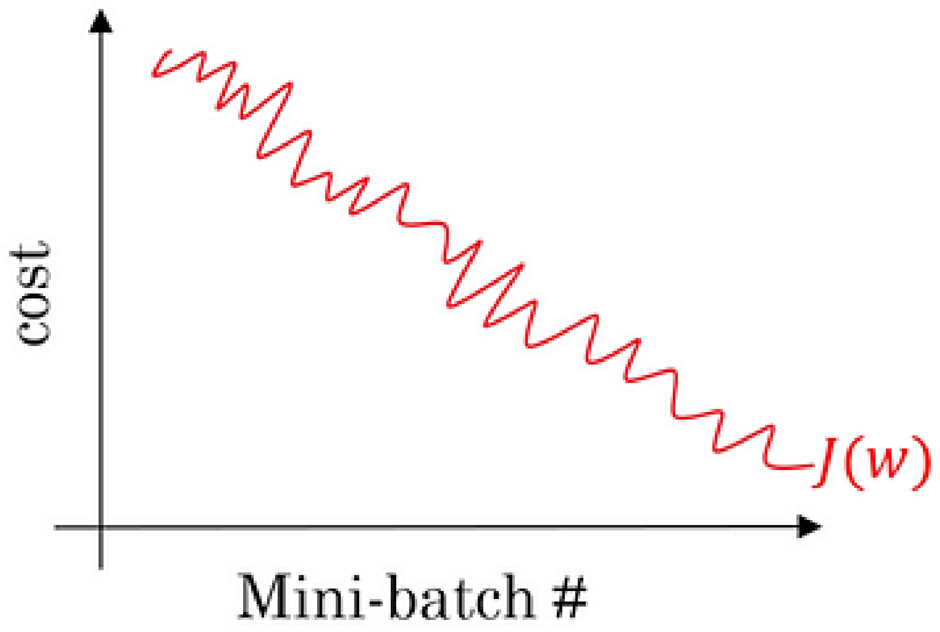
- It won’t converge. On each iteration, the learning step may go back and forth due to the noise. Therefore, it wanders around the minimum region but never converges.
- Due to the noise, the learning steps have more oscillations (see figure 4) and requires adding learning-decay to decrease the learning rate as we become closer to the minimum.


cost
w
b


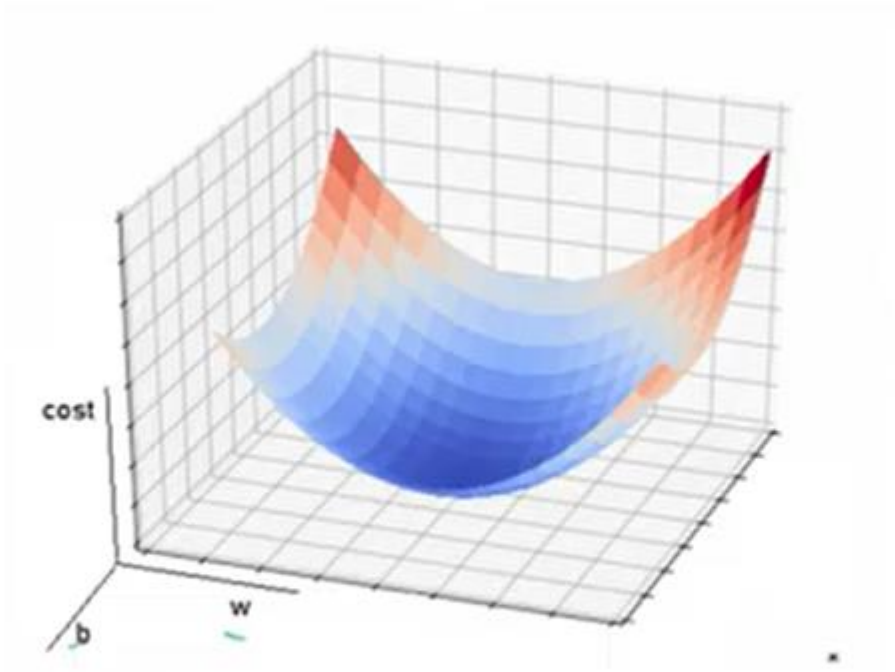
Batch gradient descent

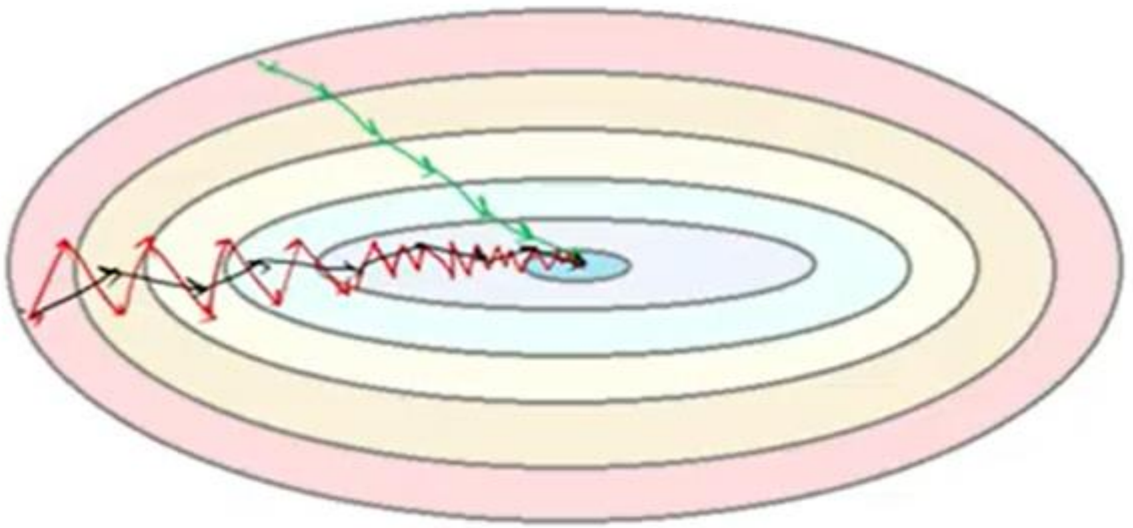


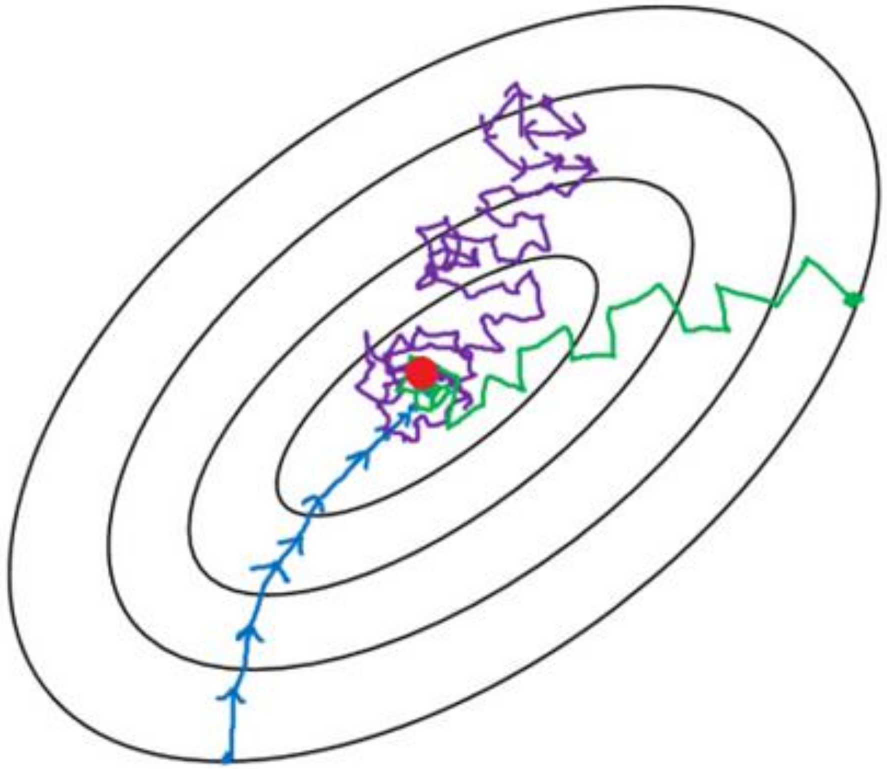
Mini-batch gradient descent



Optimization : is a technique which speed up the training/learning of a model in Deep Learning, while implementing mini-batch gradient descent or stochastic gradient descent







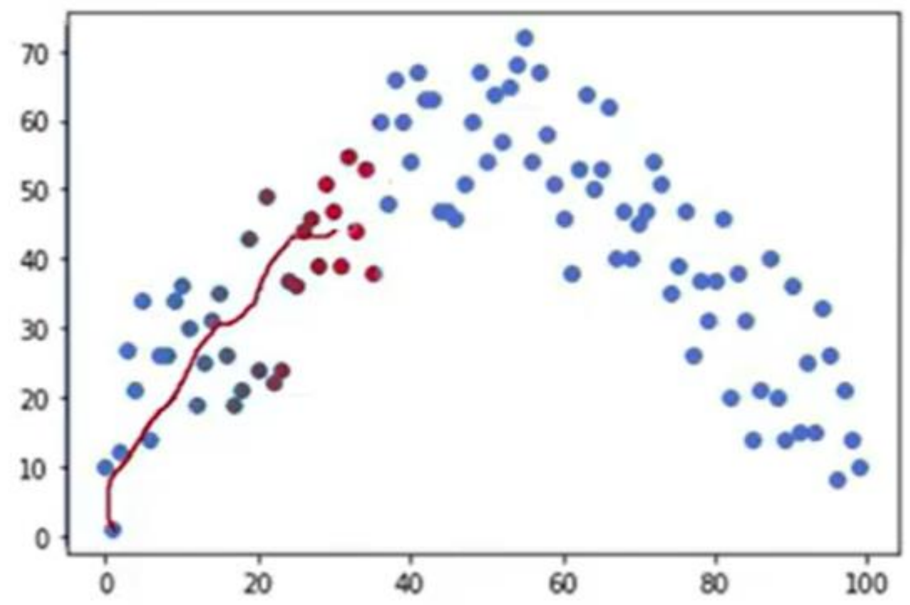
— Batch gradient descent

— Mini-batch gradient Descent

— Stochastic gradient descent

- The choice of Gradient Descent type influences the speed and quality of the optimization of a Neural Network.
- Batch Gradient Descent, while computationally expensive, provides a stable and steady descent towards the minimum.
- Stochastic Gradient Descent is faster and has the ability to jump out of local minima, but it also has a higher variance in the optimization path.
- Mini Batch Gradient Descent offers a balance between the two, providing a blend of stability and speed.

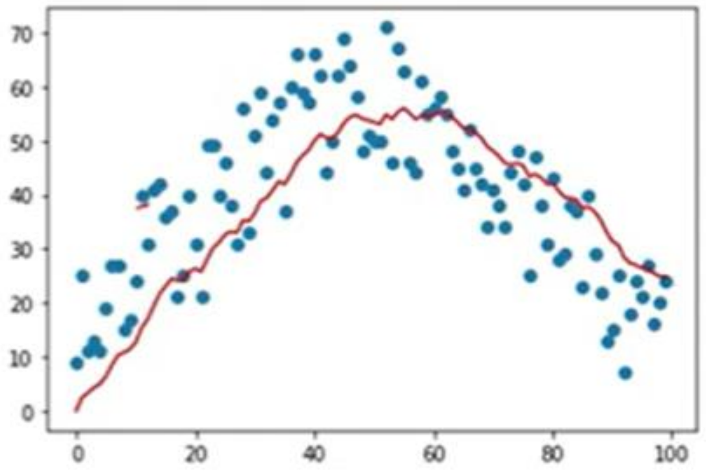
What is Exponentially Weighted Moving Average?



$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$

$$V_t = \beta * V_{t-1} + (1-\beta) * \theta_t$$

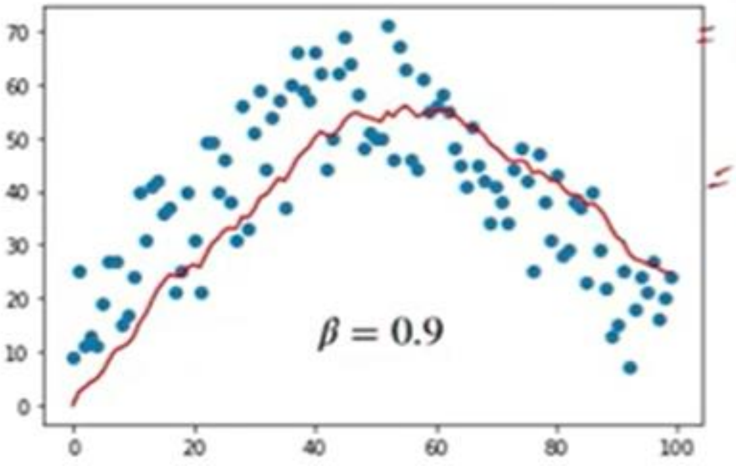
$V_0 = \theta$
$V_1 = \beta * V_0 + (1-\beta) * \theta_1$
$V_2 = \beta * V_1 + (1-\beta) * \theta_2$
$V_3 = \beta * V_2 + (1-\beta) * \theta_3$
$V_4 = \beta * V_3 + (1-\beta) * \theta_4$
$V_5 = \beta * V_4 + (1-\beta) * \theta_5$
$V_6 = \beta * V_5 + (1-\beta) * \theta_6$
$V_7 = \beta * V_6 + (1-\beta) * \theta_7$
$V_8 = \beta * V_7 + (1-\beta) * \theta_8$
$V_9 = \beta * V_8 + (1-\beta) * \theta_9$

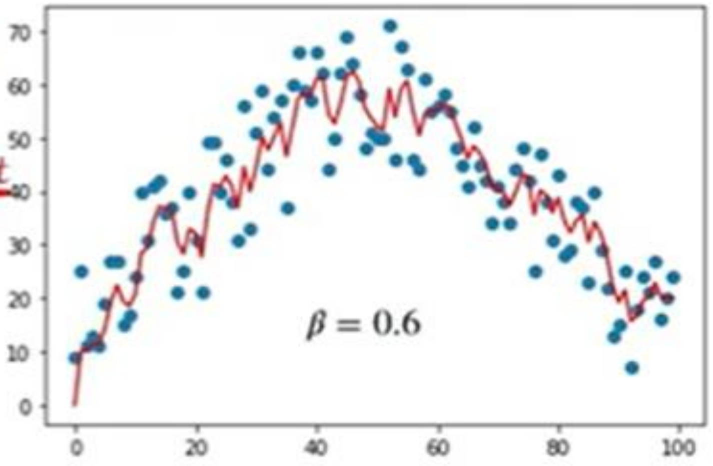


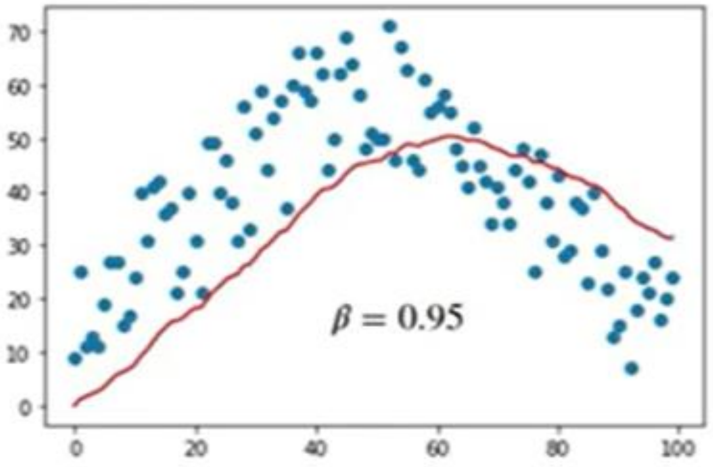
$$V_t = \beta * (V_{t-1}) + (1 - \beta) * \theta_t$$

$= 0.9(V_{t-1}) + 0.1\theta_t$

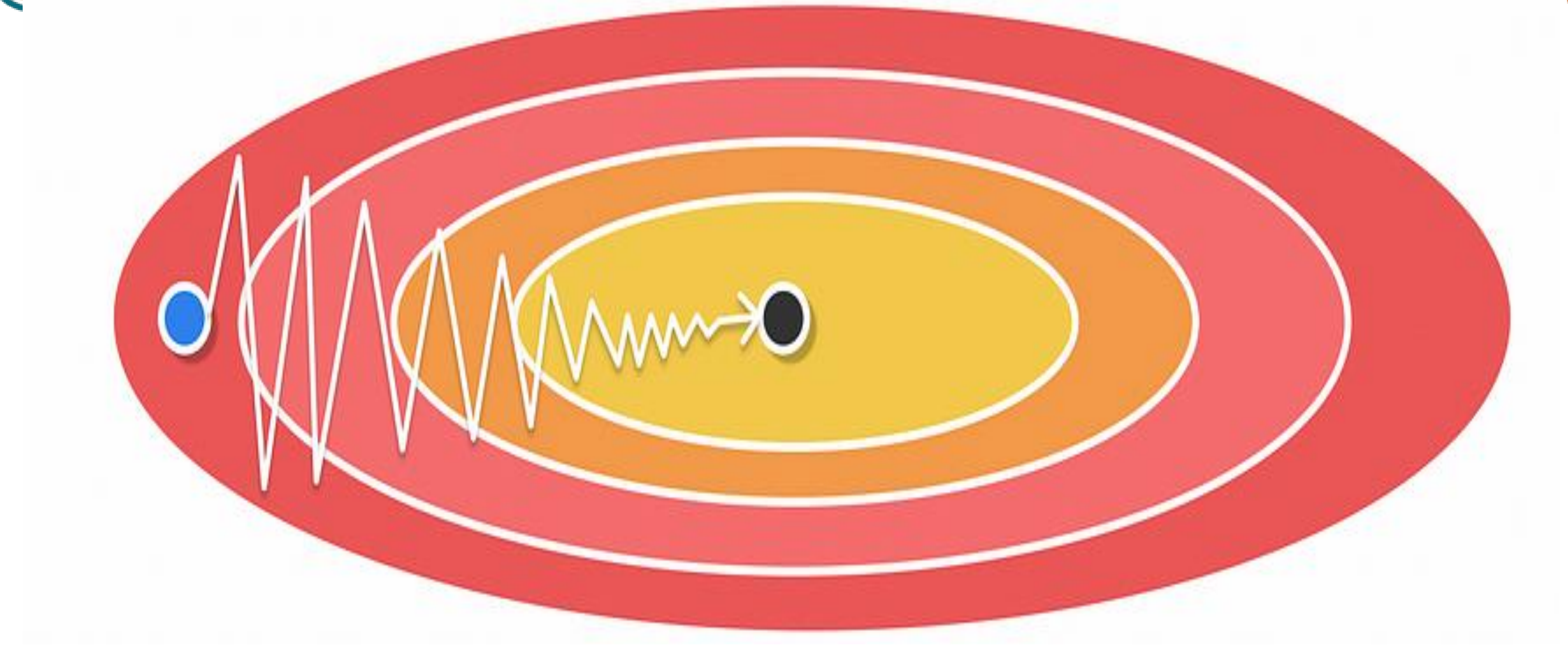
$= 0.6 V_{t-1} + (0.4)\theta_t$







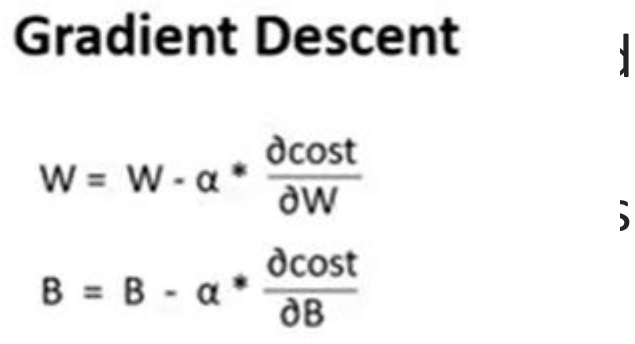
$\boxed{\beta = 0.9}$



The starting point is depicted in blue and the local minimum is shown in black.

Momentum

- Based on the example above, it would be desirable to make a loss function performing larger steps in t



Gradient Descent

$$W = W - \alpha * \frac{\partial cost}{\partial W}$$

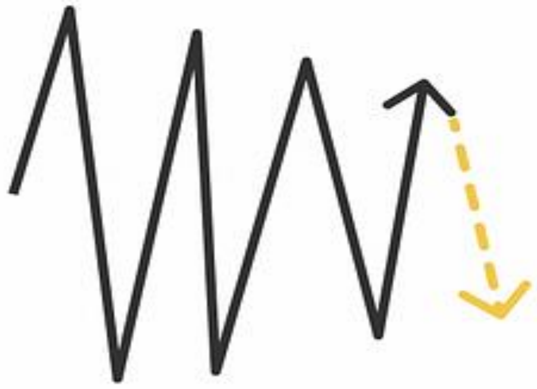
$$B = B - \alpha * \frac{\partial cost}{\partial B}$$

smaller steps in the vertical. This way, the convergence effect is exactly achieved by Momentum.

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t$$

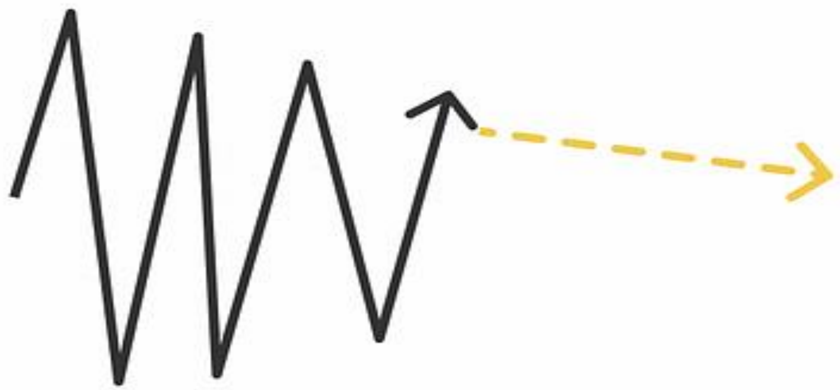
$$w_t = w_{t-1} - \alpha v_t$$

Gradient descent



the gradient computed on the current iteration does not prevent gradient descent from oscillating in the vertical direction

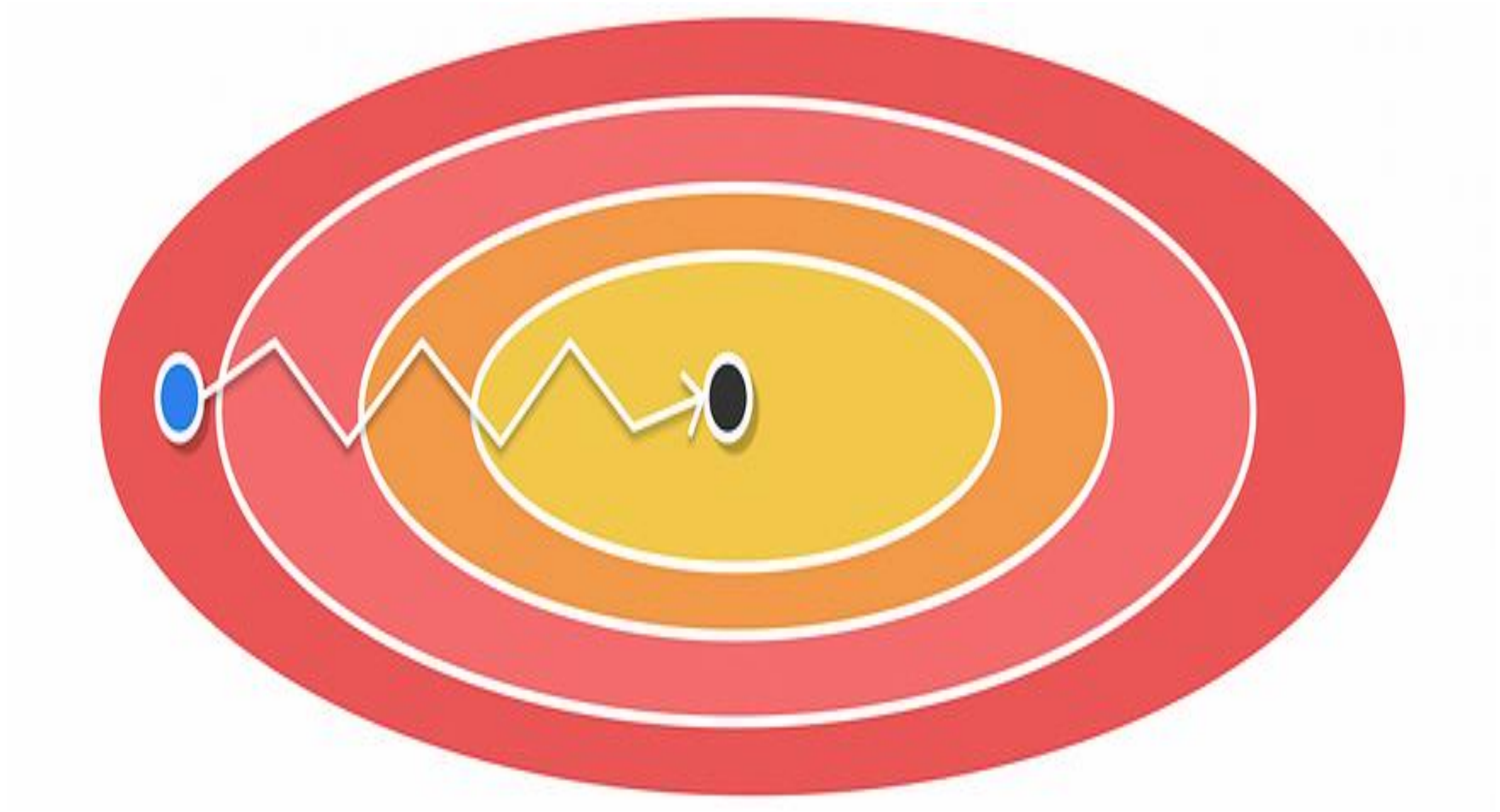
Momentum



the average vector of the horizontal component is aligned towards the minimum

the average vector of the vertical component is close to 0

- Optimization with Momentum



- In practice, Momentum usually converges much faster than gradient descent. With Momentum, there are also fewer risks in using larger learning rates, thus accelerating the training process.
- In Momentum, it is recommended to choose β close to 0.9.

AdaGrad (Adaptive Gradient Algorithm)

- AdaGrad is another optimizer with the motivation to adapt the learning rate to computed gradient values. There might occur situations when during training, one component of the weight vector has very large gradient values while another one has extremely small. **This happens especially in cases when an infrequent model parameter appears to have a low influence on predictions**.

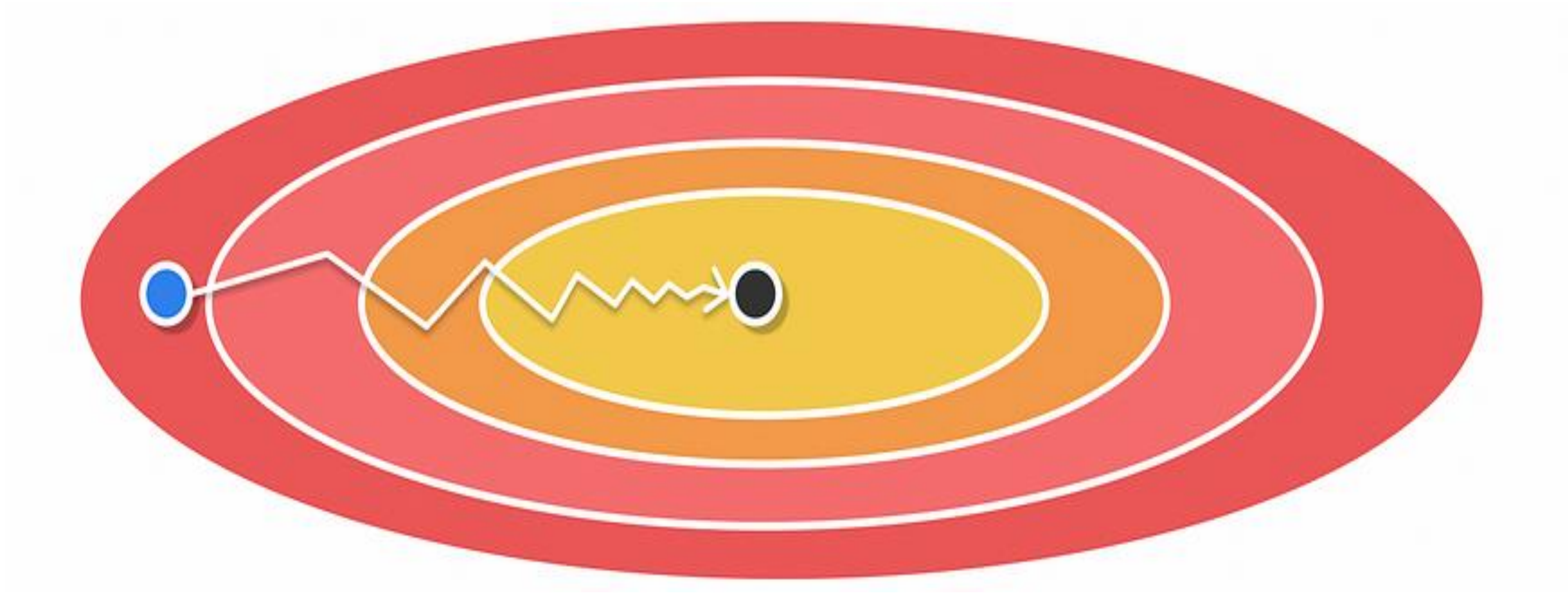
- AdaGrad deals with the aforementioned problem **by independently adapting the learning rate for each weight component**. If gradients corresponding to a certain weight vector component are large, then the respective learning rate will be small. Inversely, for smaller gradients, the learning rate will be bigger. This way, Adagrad deals with vanishing and exploding gradient problems.

$$v_t = v_{t-1} + dw_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t + \varepsilon}} dw_t$$

- The greatest advantage of AdaGrad is that there is no longer a need to manually adjust the learning rate as it adapts itself during training.
- Nevertheless, there is a negative side of AdaGrad: **the learning rate constantly decays with the increase of iterations** (the learning rate is always divided by a positive cumulative number). Therefore, the algorithm tends to converge slowly during the last iterations where it becomes very low.
-

Optimization with AdaGrad



RMSProp (Root Mean Square Propagation)

- RMSProp was elaborated as an improvement over AdaGrad which tackles the issue of learning rate decay. Similarly to AdaGrad, RMSProp uses a pair of equations for which the weight update is absolutely the same.

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t + \varepsilon}} dw_t$$

Adam (Adaptive Moment Estimation)

- For the moment, Adam is the most famous optimization algorithm in deep learning. At a high level, Adam combines Momentum and RMSProp algorithms. To achieve it, it simply keeps track of the exponentially moving averages for computed gradients and squared gradients respectively.

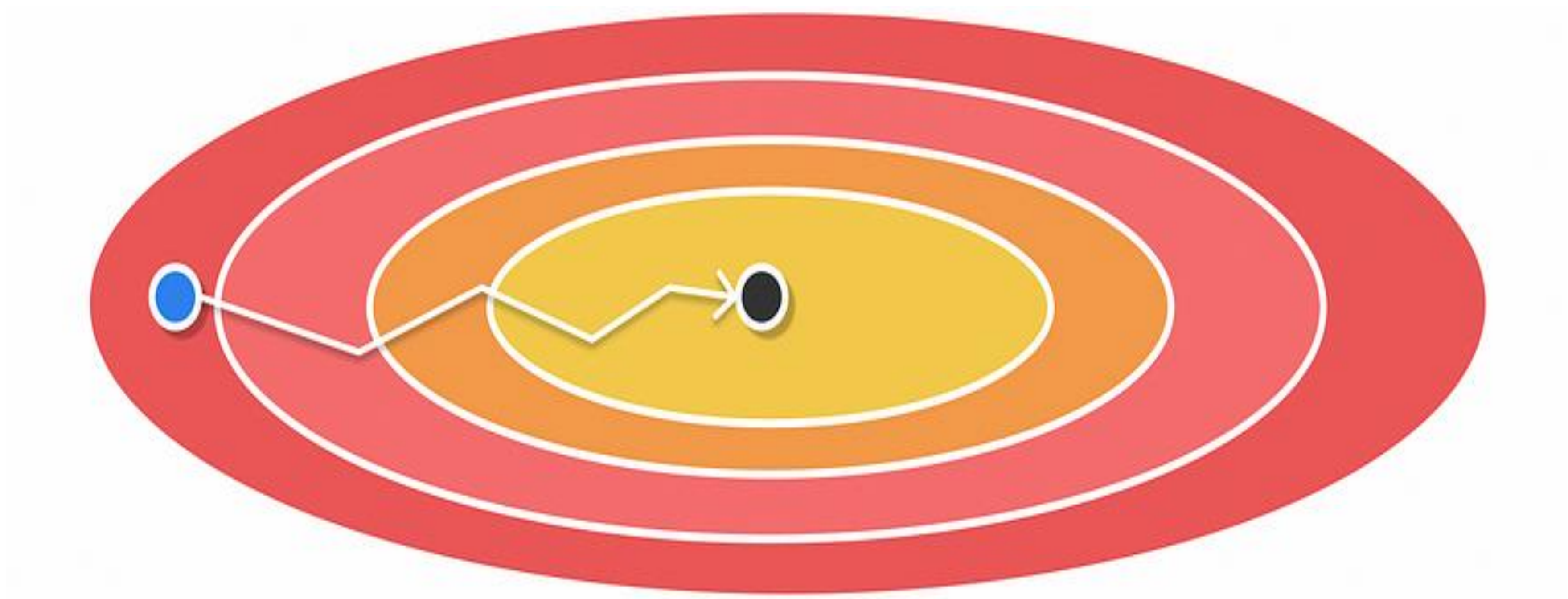
$$v_t = \beta_1 v_{t-1} + (1 - \beta_1)dw_t \xrightarrow[\text{bias}]{} \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2)dw_t^2 \xrightarrow{\text{correction}} \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \frac{a\hat{v}_t}{\sqrt{\hat{s}_t} + \varepsilon}dw_t$$

- Furthermore, it is possible to use bias correction for moving averages for a more precise approximation of gradient trend during the first several iterations. The experiments show that Adam adapts well to almost any type of neural network architecture taking the advantages of both Momentum and RMSProp.

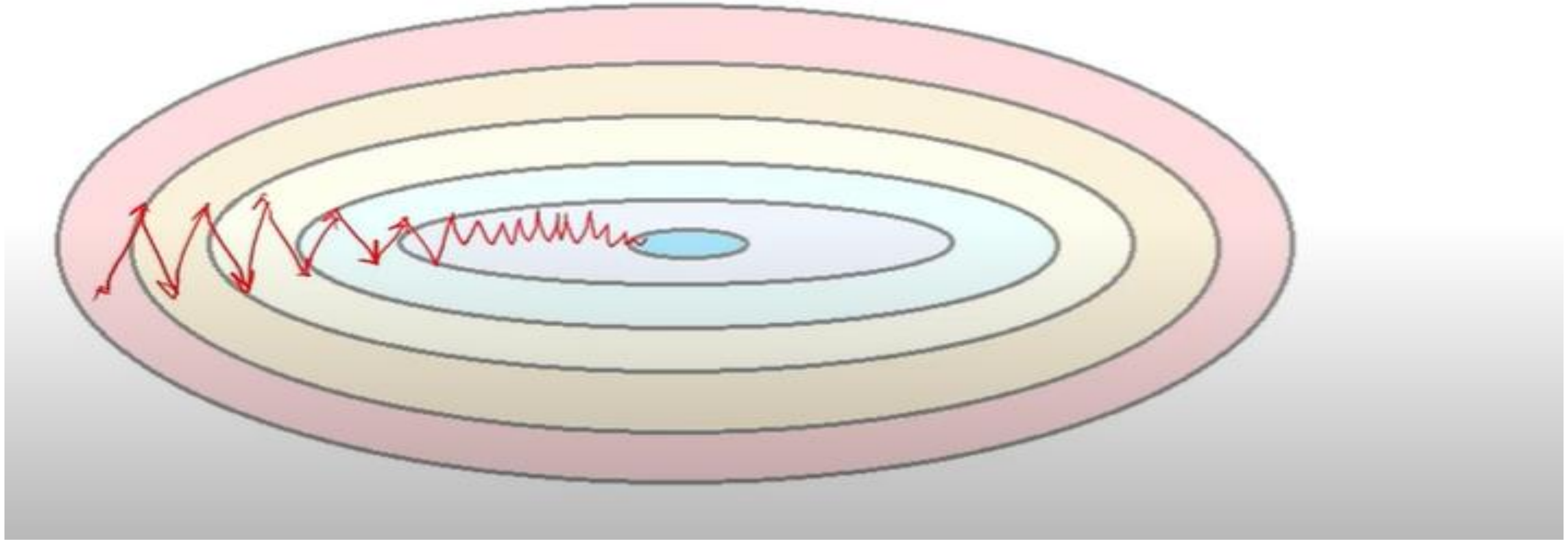
Optimization with Adam



Adam

RMSprop

Momentum



Momentum

$$V_{dW} = \beta \cdot V_{dW_{prev}} + (1-\beta)\, dW$$

$$V_{dB} = \beta \cdot V_{dB_{prev}} + (1-B) \cdot dB$$

$$W = W - \alpha \cdot V_{dW}$$

$$\beta = B - \alpha \cdot V_{dB}$$

RMSprop

$$S_{dW} = \beta \cdot S_{dW_{prev}} + (1-\beta)(dW)^2$$

$$S_{dB} = \beta \cdot S_{dB_{prev}} + (1-\beta)(dB)^2$$

$$W = W - \alpha \cdot (dW / \sqrt{S_{dW} + \varepsilon})$$

$$B = B - \alpha \cdot (dB / \sqrt{S_{dB} + \varepsilon})$$

Momentum

$$V_{dW} = \beta_1 V_{dW_{prev}} + (1-\beta_1)\, dW$$

$$V_{dB} = \beta_1 \cdot V_{dB_{prev}} + (1-\beta_1) \cdot dB$$

$$W = W - \alpha \cdot V_{dW}$$

$$B = B - \alpha \cdot V_{dB}$$

RMSprop

$$S_{dW} = \beta_2 \cdot S_{dW_{prev}} + (1-\beta_2)(dW)^2$$

$$S_{dB} = \beta_2 \cdot S_{dB\,prev} + (1-\beta_2)(dB)^2 \quad B)^2$$

$$W = W - \alpha \cdot \left(dW / \sqrt{S_{dW} + \varepsilon} \right)$$

$$B = B - \alpha \cdot \left(dB / \sqrt{S_{dB} + \varepsilon} \right) \quad)$$

Adam (Adam moment estimation)

$$\begin{cases} W = W - \alpha \cdot \dfrac{V_{dW}}{\sqrt{S_{dW} + \varepsilon}} \\[2em] B = B - \alpha \dfrac{V_{dB}}{\sqrt{S_{dB} + \varepsilon}} \end{cases}$$

$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\varepsilon = 10^{-8}$

```
def initialize_adam(n_x, n_h, n_y):
        # Your code here
        return vW1, vb1, vW2, vb2, sW1, sb1, sW2, sb2
```

Expected Output:
First Moment Vectors (Velocity Terms):

1. vW1: A NumPy array of shape (n_h, n_x) initialized to zeros, representing the velocity term for the weight matrix connecting the input layer to the hidden layer.
2. vb1: A NumPy array of shape (n_h, 1) initialized to zeros, representing the velocity term for the bias vector of the hidden layer.
3. vW2: A NumPy array of shape (n_y, n_h) initialized to zeros, representing the velocity term for the weight matrix connecting the hidden layer to the output layer.
4. vb2: A NumPy array of shape (n_y, 1) initialized to zeros, representing the velocity term for the bias vector of the output layer.

Second Moment Vectors (Squared Gradient Terms):

1. sW1: A NumPy array of shape (n_h, n_x) initialized to zeros, representing the squared gradient term for the weight matrix connecting the input layer to the hidden layer.
2. sb1: A NumPy array of shape (n_h, 1) initialized to zeros, representing the squared gradient term for the bias vector of the hidden layer.
3. sW2: A NumPy array of shape (n_y, n_h) initialized to zeros, representing the squared gradient term for the weight matrix connecting the hidden layer to the output layer.
4. sb2: A NumPy array of shape (n_y, 1) initialized to zeros, representing the squared gradient term for the bias vector of the output layer.

```
def update_parameters_with_adam(W1, b1, W2, b2, dW1,
db1, dW2, db2, vW1, vb1, vW2, vb2, sW1, sb1, sW2, sb2,
t, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-
8):
    # Your code here
    return W1, b1, W2, b2, vW1, vb1, vW2, vb2, sW1, sb1,
    sW2, sb2
```

Parameters:

W1: A NumPy array of shape (n_h, n_x) representing the weights of the layer connecting the input to the hidden layer.

b1: A NumPy array of shape (n_h, 1) representing the biases of the hidden layer.

W2: A NumPy array of shape (n_y, n_h) representing the weights of the layer connecting the hidden layer to the output layer.

b2: A NumPy array of shape (n_y, 1) representing the biases of the output layer.

dW1: A NumPy array of shape (n_h, n_x) representing the gradient of the weights of the layer connecting the input to the hidden layer.

db1: A NumPy array of shape (n_h, 1) representing the gradient of the biases of the hidden layer.

dW2: A NumPy array of shape (n_y, n_h) representing the gradient of the weights of the layer connecting the hidden layer to the output layer.

db2: A NumPy array of shape (n_y, 1) representing the gradient of the biases of the output layer.

vW1: A NumPy array of shape (n_h, n_x) representing the moving average of the gradients for W1.

vb1: A NumPy array of shape (n_h, 1) representing the moving average of the gradients for b1.

vW2: A NumPy array of shape (n_y, n_h) representing the moving average of the gradients for W2.

vb2: A NumPy array of shape (n_y, 1) representing the moving average of the gradients for b2.

sW1: A NumPy array of shape (n_h, n_x) representing the moving average of the squared gradients for W1.

sb1: A NumPy array of shape (n_h, 1) representing the moving average of the squared gradients for b1.

sW2: A NumPy array of shape (n_y, n_h) representing the moving average of the squared gradients for W2.

sb2: A NumPy array of shape (n_y, 1) representing the moving average of the squared gradients for t: An integer representing the current time step (iteration) of the optimization process.

learning_rate: A float representing the learning rate for the optimization (default is 0.001).

beta1: A float representing the exponential decay rate for the first moment estimates (default is 0.9).

beta2: A float representing the exponential decay rate for the second moment estimates (default is 0.999).

epsilon: A small constant to prevent division by zero (default is 1e-8).

Conclusion

- We have looked at different optimization algorithms in neural networks. Considered as a combination of Momentum and RMSProp, Adam is the most superior of them which robustly adapts to large datasets and deep networks. Moreover, it has a straightforward implementation and little memory requirements making it a preferable choice in the majority of situations.