DEEP LEARNING (CSE436)

BY: Nidhi S. Periwal,
Teaching Assistant,
COED, SVNIT, Surat

Neural Networks Basics & Deep Neural Networks

(Partial)

Parameters vs Hyperparameters

- Parameters are learned by the model during the training time,
  - Parameters of a deep neural network are W and b, which the model updates during the backpropagation step.
- Hyperparameters can be changed before training the model.
  - Hyperparameters for a deep NN, including:
    - Learning rate – α
    - Number of iterations
    - Number of hidden layers
    - Units in each hidden layer
    - Choice of activation function

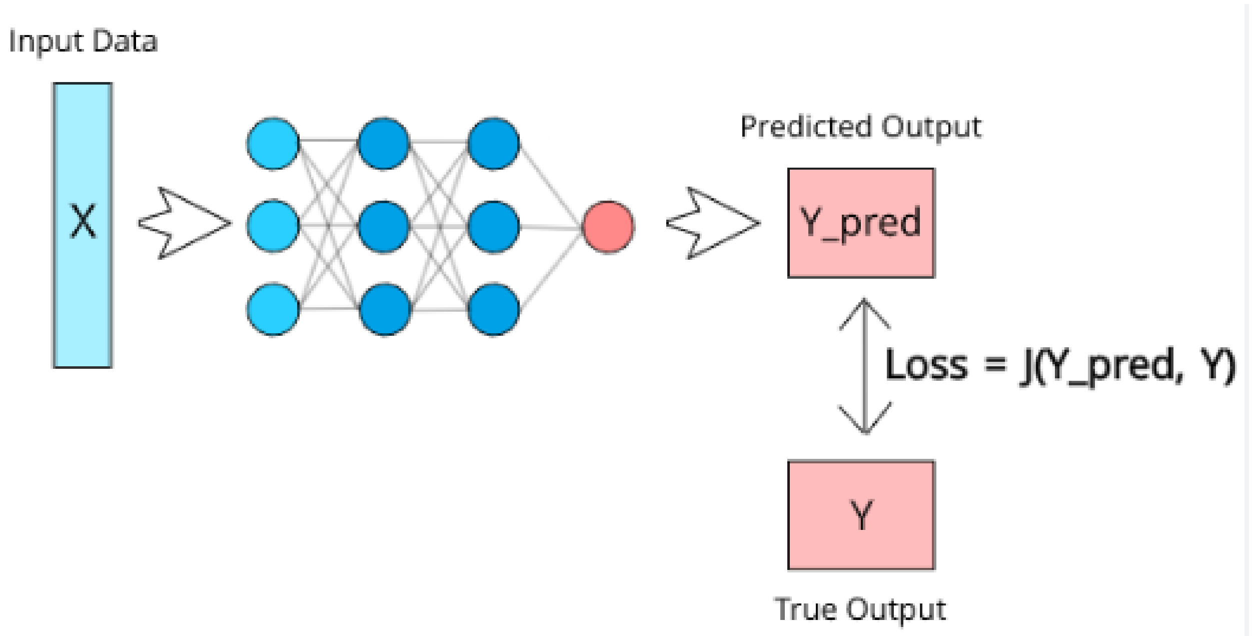
Learning in NN

- Start with arbitrary value of Weights and bias.
- Weights and bias changes with **multiple iterations until there is no wrong prediction.**
- In supervised learning, the objective is to **reduce the number of erroneous prediction**.
- On increase in number of layers or interconnection weights, the problem becomes complex
- These parameters are learnt using back propagation.

Gradient Descent

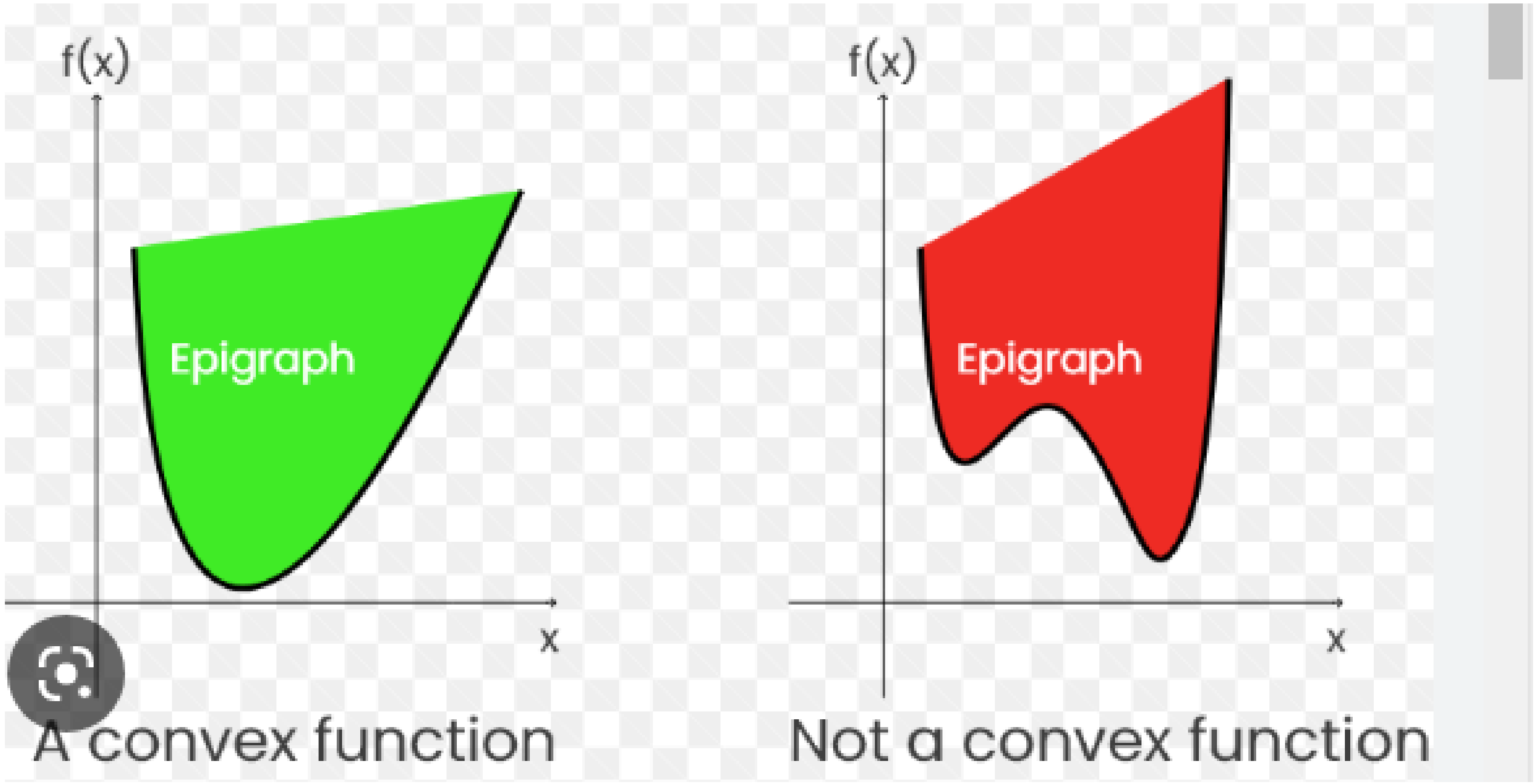
- Gradient Descent – key concept in back propagation
- *Gradient Descent is defined as* ***one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models.*** *It helps in finding the local minimum of a function*
- *Gradient calculates* ***the partial derivative of loss function by each interconnection weight and bias.***
- ***i.e. identify the "gradient" or extent to change of weight or bias required to minimize loss function****.*

- **Loss Functions** are used to calculate the error between the known **predicted output and the actual output** generated by a model, Also often called Cost Functions.
- Measures extent of prediction error of supervised model

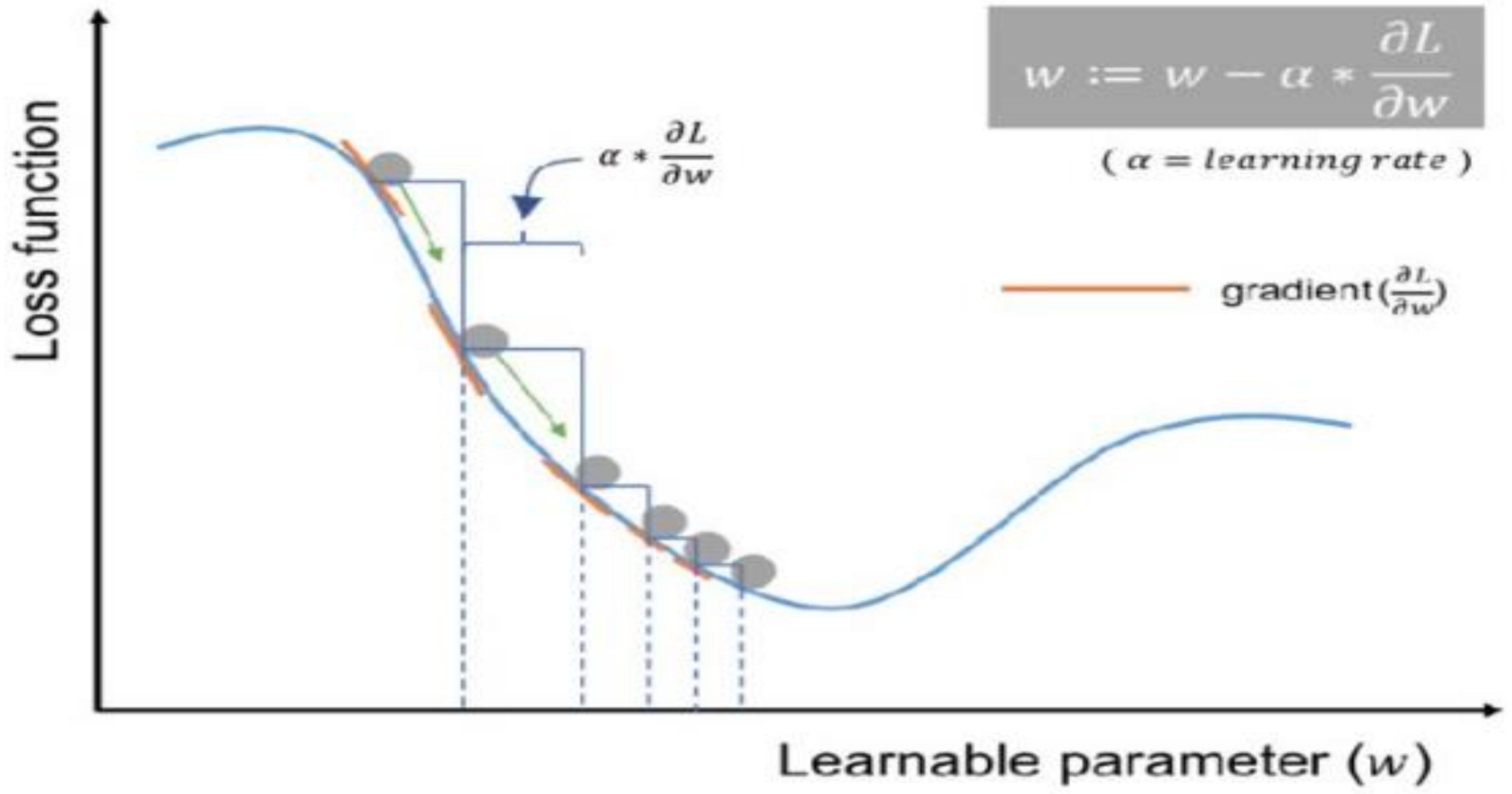


Gradient Descent

- Convex problems have only one minimum; that is, only one place where the slope is exactly 0. **That minimum is where the loss function converges**.
- **Gradient Descent** is an **iterative** optimization method for finding the minimum of a function. **On each iteration the parameters in a model are amended in the direction of the negative gradient of the output until the optimum parameters for the model are identified**

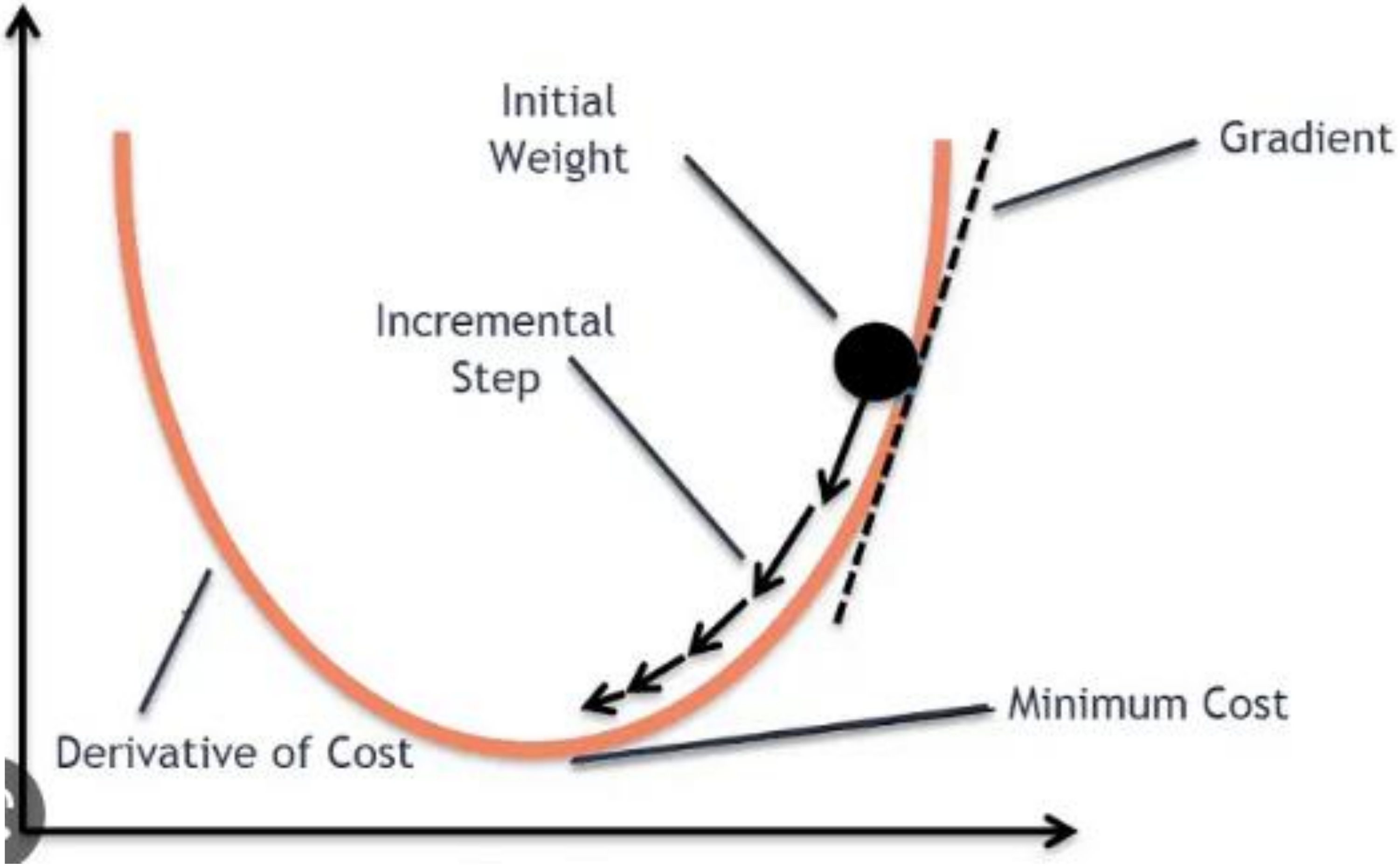

f(x)
Epigraph
x
A convex function
f(x)
Epigraph
x
Not a convex function

Gradient Descent

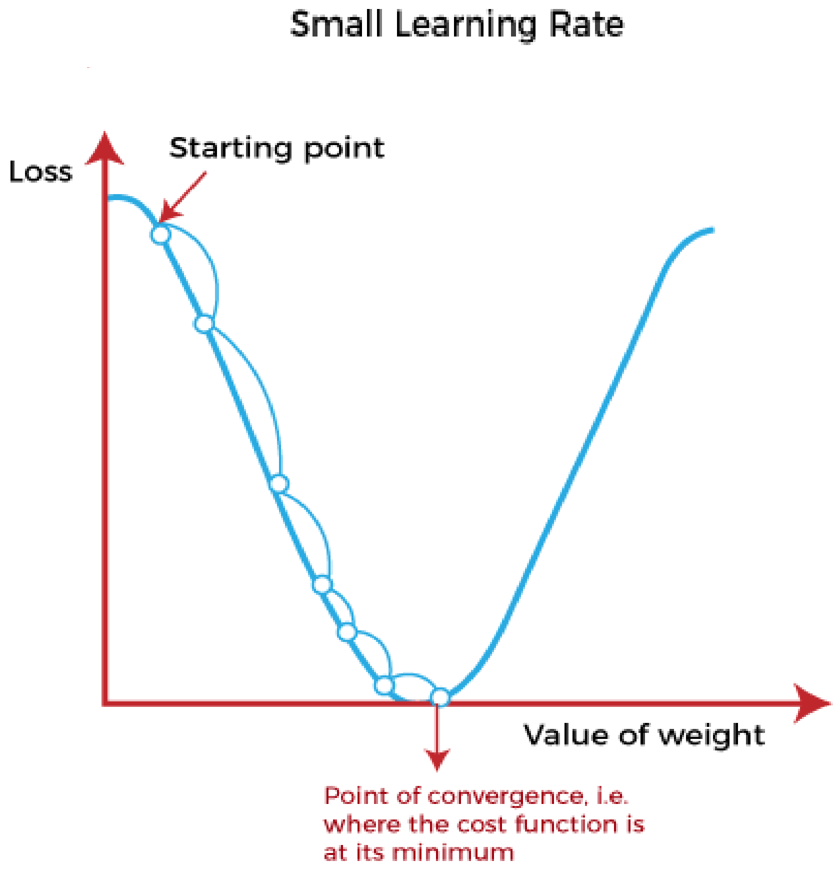


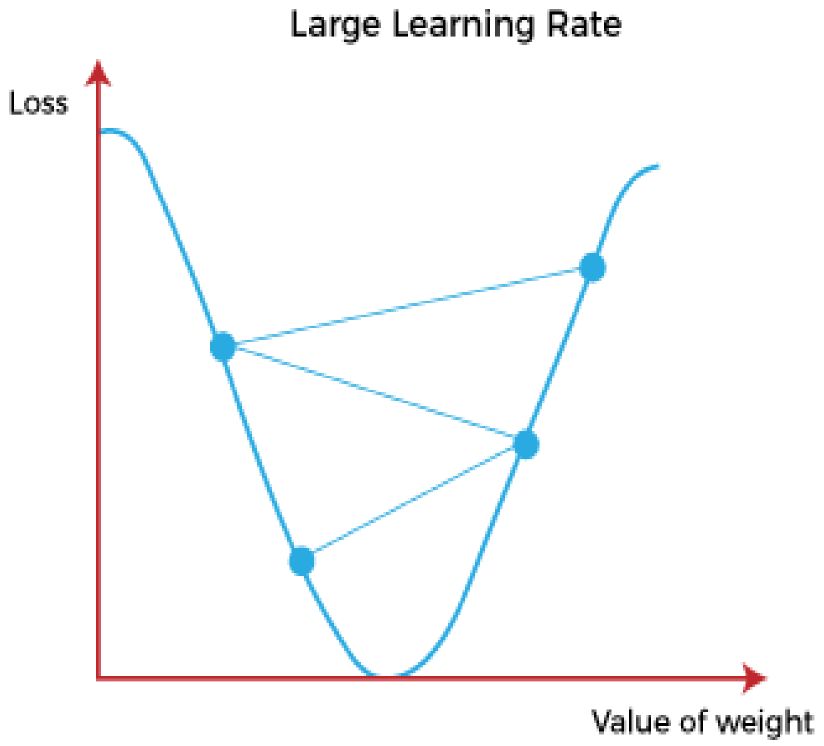
$w := w - \alpha * \frac{\partial L}{\partial w}$
$(\alpha = learning\ rate)$
$\alpha * \frac{\partial L}{\partial w}$
gradient$(\frac{\partial L}{\partial w})$
Loss function
Learnable parameter (w)

Gradient Descent

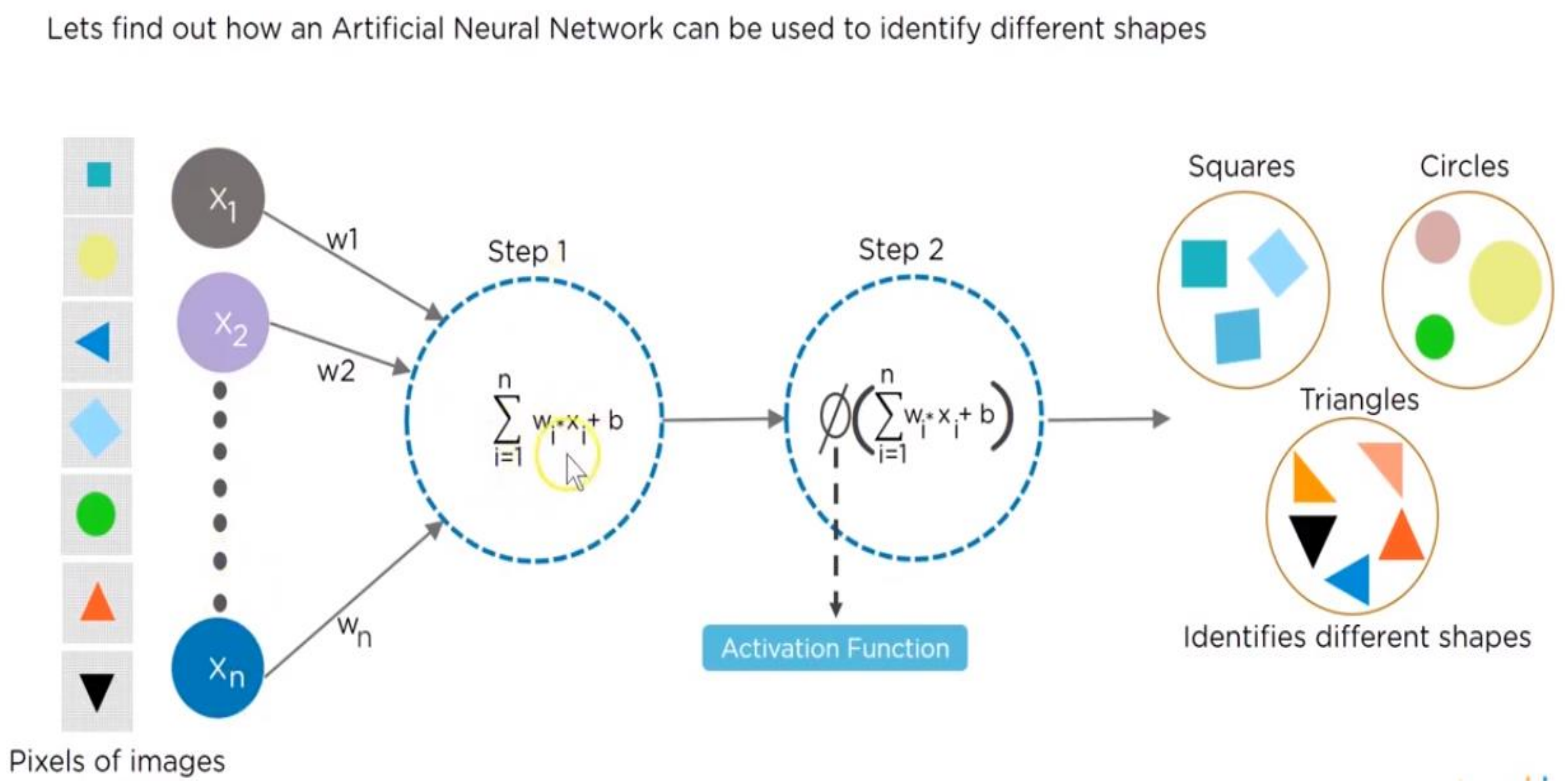


- **Learning Rate:**
- It is defined as the **step size** taken to reach the minimum or lowest point.
- This is typically **a small value that is evaluated and updated based on the behavior of the cost function**.
- If the learning rate is **high, it results in larger steps but also leads to risks of overshooting the minimum**.
- A **low learning rate shows the small step sizes, which compromises overall efficiency** but gives the advantage of more precision.

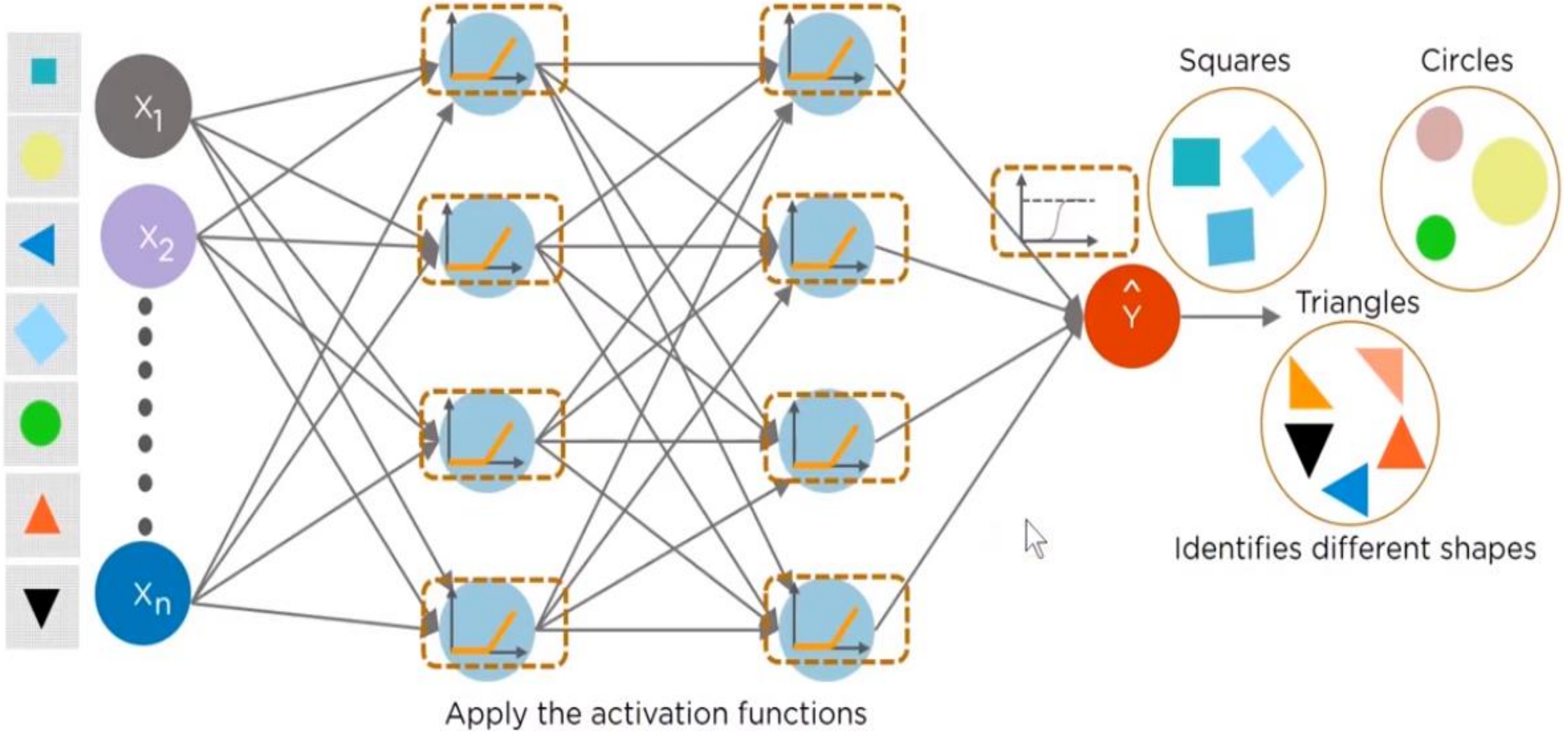

Small Learning Rate
Loss
Starting point
Value of weight
Point of convergence, i.e. where the cost function is at its minimum

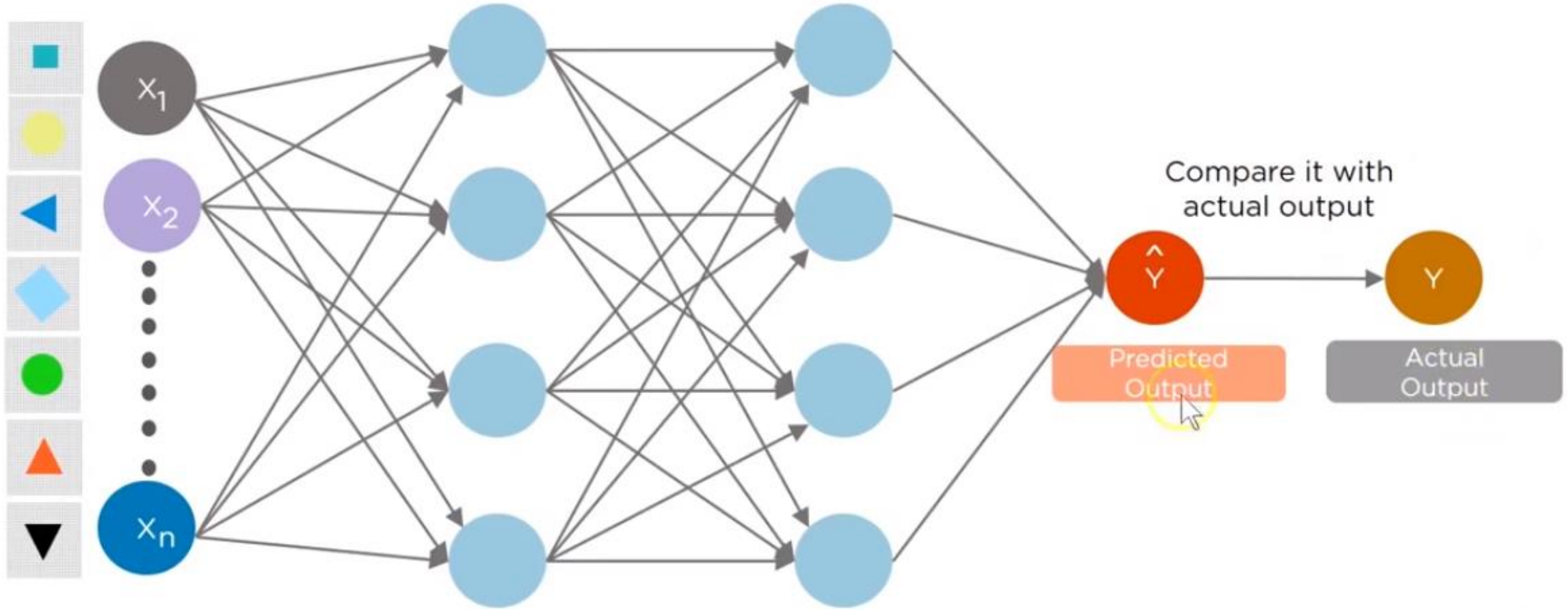

Large Learning Rate
Loss
Value of weight

Working of Simple NN

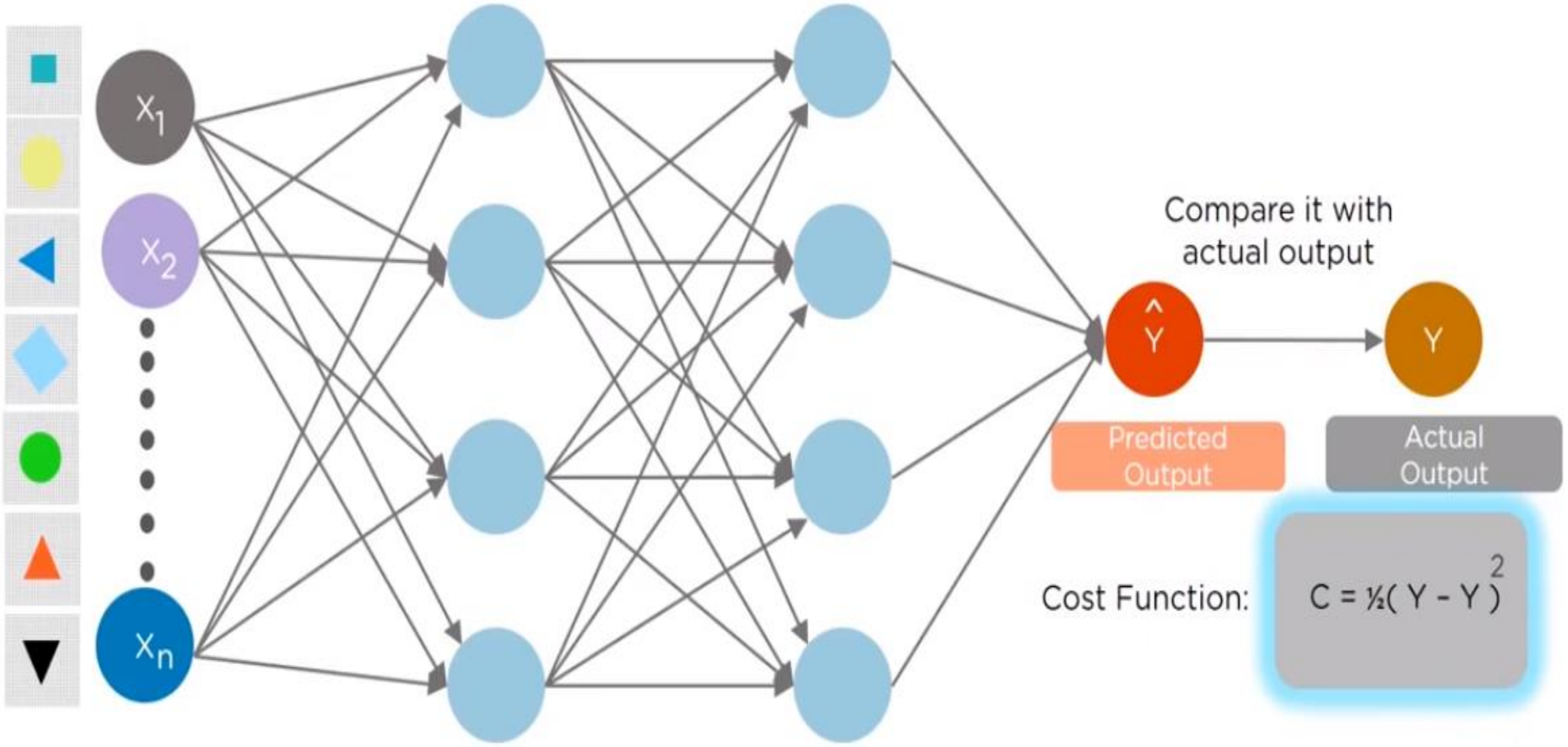


Lets find out how an Artificial Neural Network can be used to identify different shapes
x_1
x_2
x_n
w1
w2
w_n
Step 1
$\sum_{i=1}^{n} w_i * x_i + b$
Step 2
$\phi\left(\sum_{i=1}^{n} w_i * x_i + b\right)$
Activation Function
Squares
Circles
Triangles
Identifies different shapes
Pixels of images

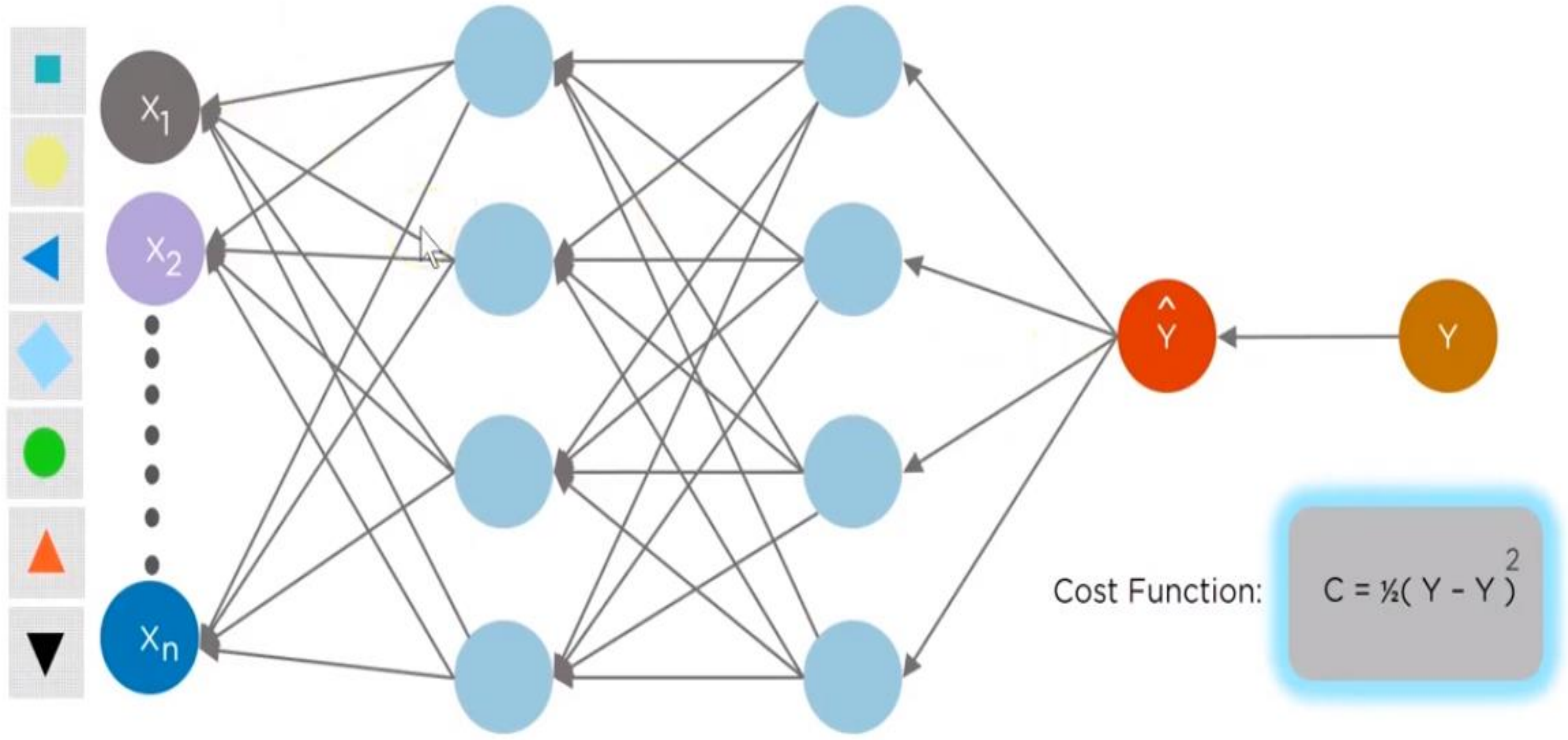

X_1
X_2
X_n
$\hat{Y}$
Squares
Circles
Triangles
Identifies different shapes
Apply the activation functions

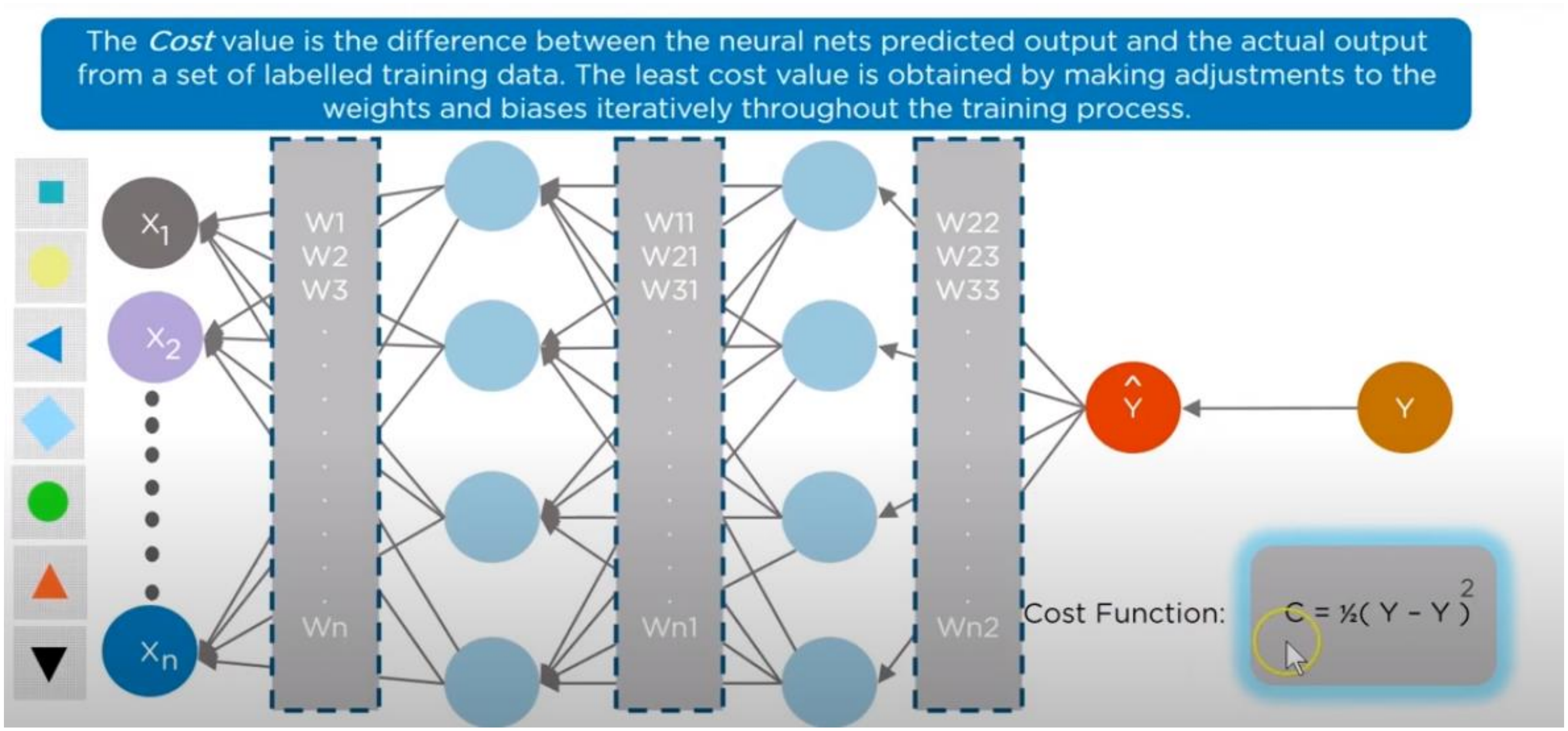

x_1
x_2
x_n
Compare it with actual output
$\hat{Y}$
Y
Predicted Output
Actual Output

Applying the cost function to minimize the difference between predicted and actual output using gradient descent algorithm

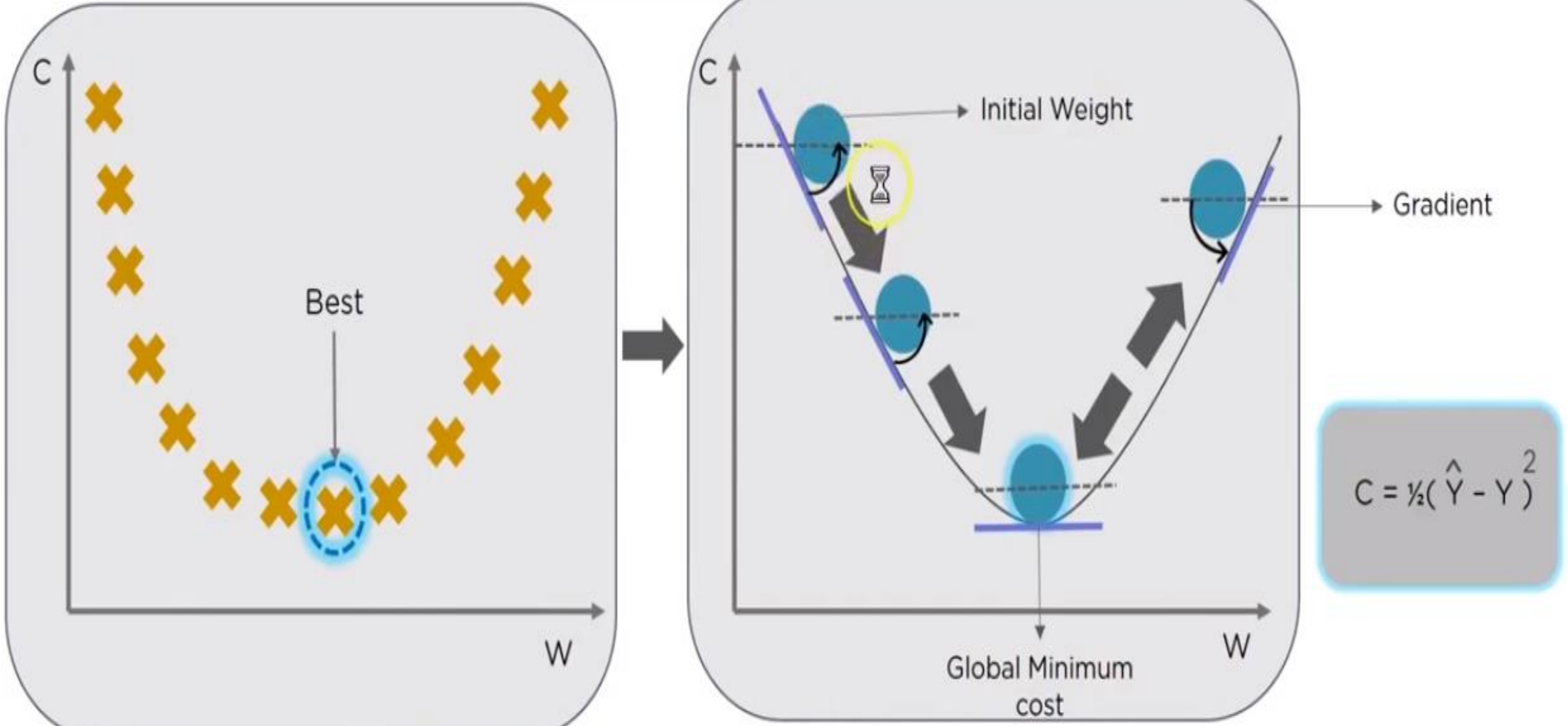


Neural Networks use *Backpropagation* method along with improve the performance of the Neural Net. A *cost function* is used to reduce the error rate between predicted and actual output.




The *Cost* value is the difference between the neural nets predicted output and the actual output from a set of labelled training data. The least cost value is obtained by making adjustments to the weights and biases iteratively throughout the training process.

Gradient Descent is an optimization algorithm for finding the minimum of a function



$$C = \frac{1}{2}(\hat{Y} - Y)^2$$

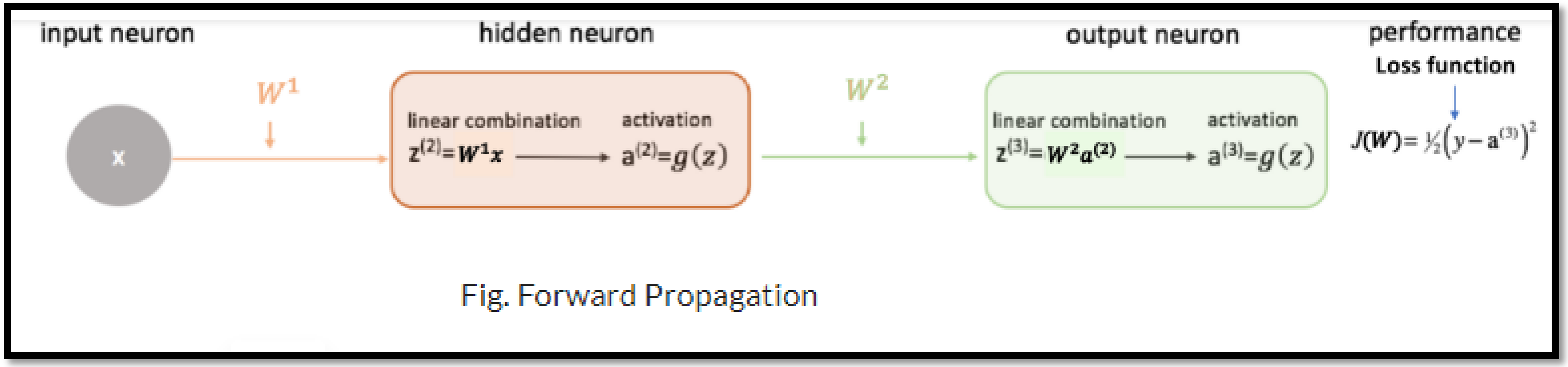
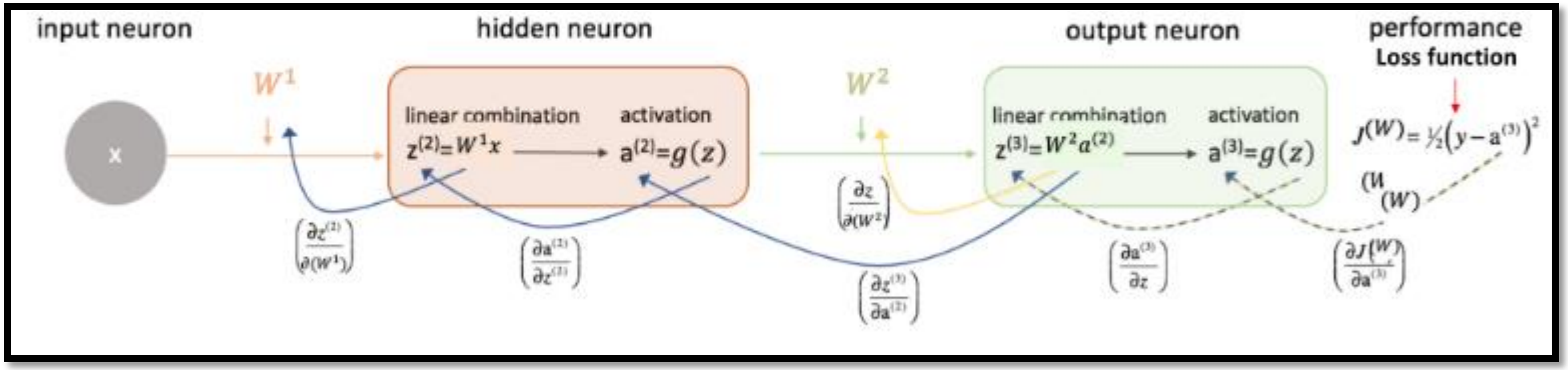
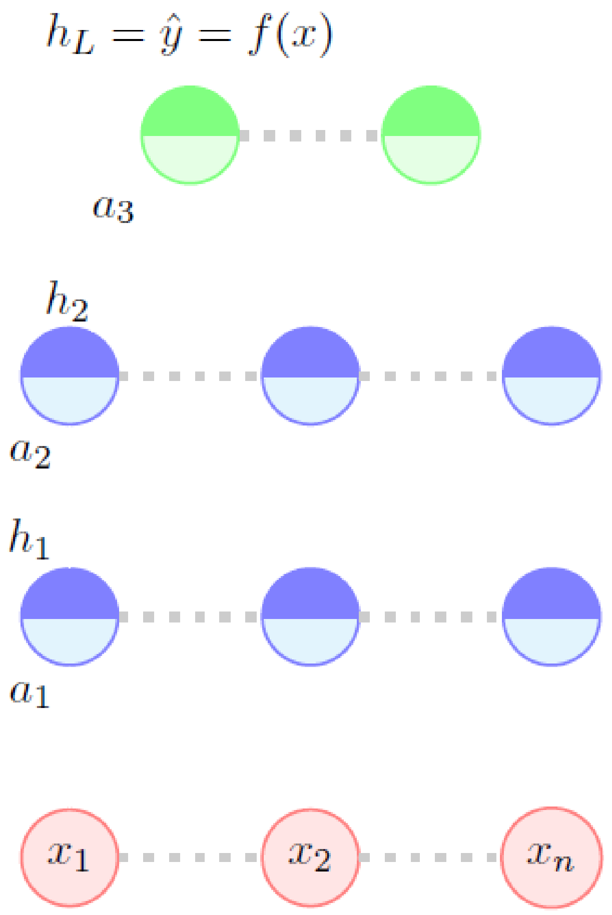


Fig. Forward Propagation

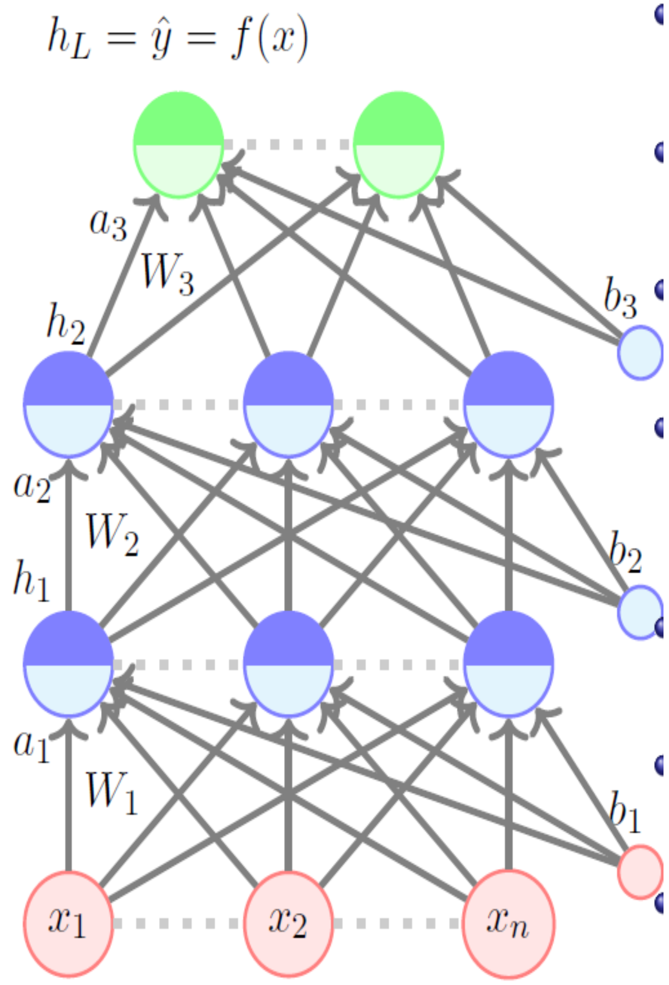

input neuron
hidden neuron
output neuron
performance
Loss function
W^1
W^2
x
linear combination
$z^{(2)}=W^1x$
activation
$a^{(2)}=g(z)$
linear combination
$z^{(3)}=W^2a^{(2)}$
activation
$a^{(3)}=g(z)$
$J(W)=\frac{1}{2}(y-a^{(3)})^2$
$J(W)$
$\left(\frac{\partial z^{(2)}}{\partial(W^1)}\right)$
$\left(\frac{\partial a^{(2)}}{\partial z^{(2)}}\right)$
$\left(\frac{\partial z}{\partial(W^2)}\right)$
$\left(\frac{\partial z^{(3)}}{\partial a^{(2)}}\right)$
$\left(\frac{\partial a^{(3)}}{\partial z}\right)$
$\left(\frac{\partial J(W)}{\partial a^{(3)}}\right)$

Deep L-Layer Neural Network



- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation (a_i and h_i are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the (L)-th layer

Deep L-Layer Neural Network

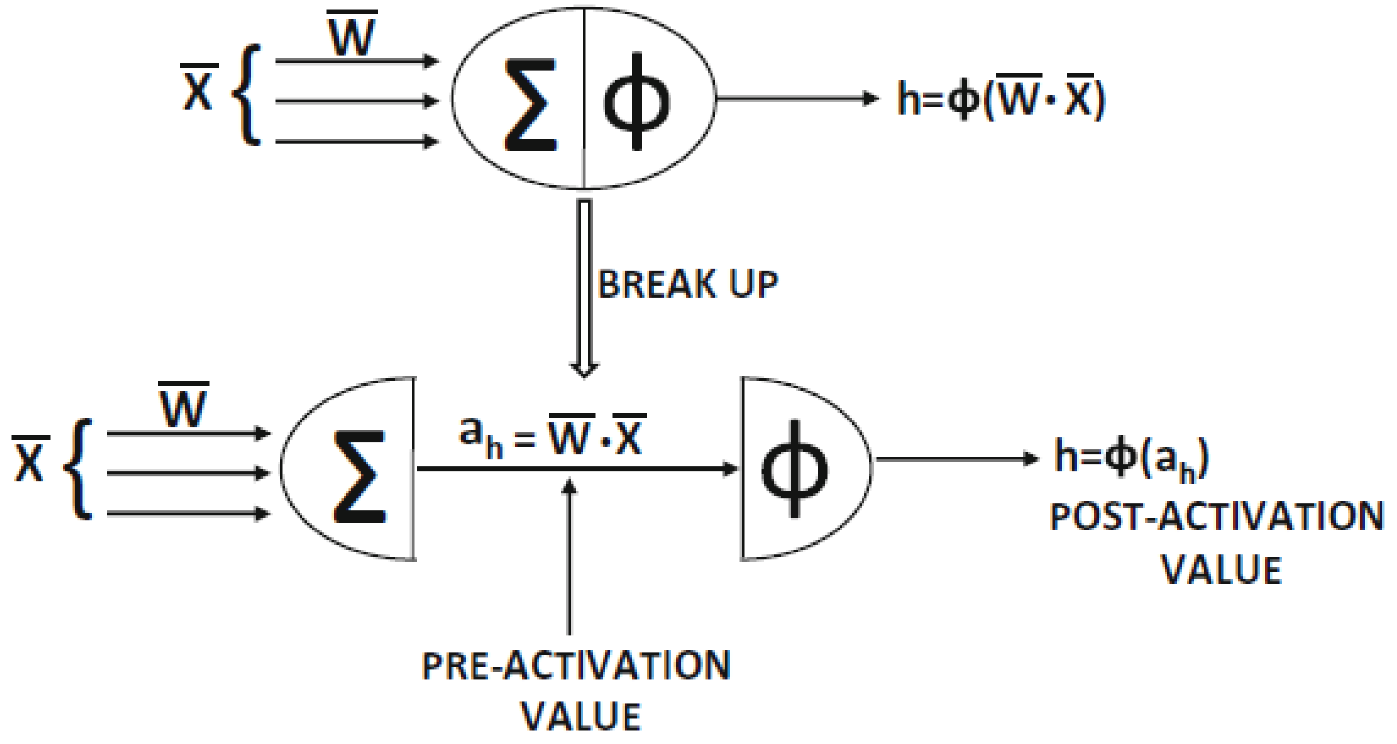


- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and i $(0 < i < L)$
- $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer

Deep L-Layer Neural Network

- Summation : $Z_1^{[1]} = W_1^{[1]} X + b_1^{[1]}$
- Activation : $A_1^{[1]} = f_1^{[1]}(Z_1^{[1]})$
- Superscript→ Layer in the network
- Subscript→ Sequence number of the specific neuron in the network

Neural Network



Pre-activation and post-activation values within a neuron

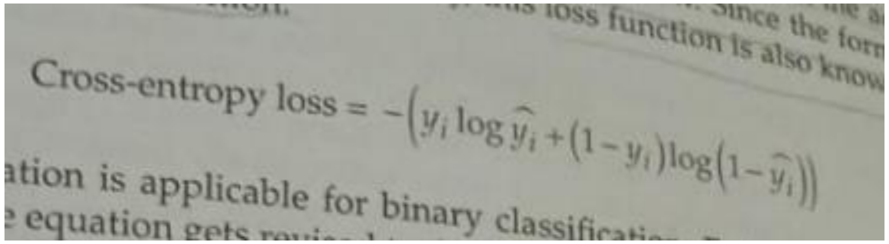
Cost/Loss Functions

- A loss function helps to measure the extent to which the supervised model is **going wrong in predicting the value of target variable of the test instances.**
- It tells how **bad** the supervised model is performing.
- The goal is to **minimize** the loss function.
- If prediction deviate too much from the ground truth, the loss function will churn out a very high number.
  - Classification Loss
    - Cross Entropy Loss
  - Regression Loss
    - Mean Squared Error
    - Mean Absolute Error

Classification Loss

- Cross Entropy loss is most commonly adopted loss function for classification problems.
- It **increases as the predicted probability** diverges from the actual label.
- As it includes **–ve log value** of predicted probability, this loss function is known as

Negative log likelihood or log loss function.



loss function. Since the form

loss function is also know

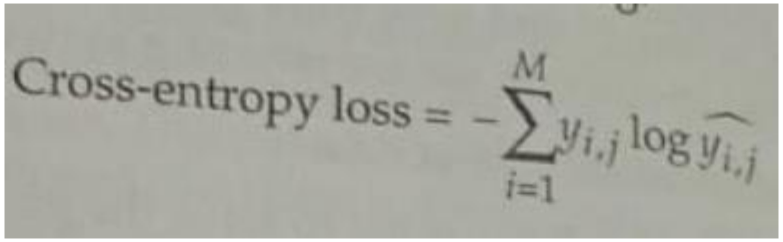
Cross-entropy loss $= -\left(y_i \log \widehat{y_i} + (1-y_i)\log\left(1-\widehat{y_i}\right)\right)$

ation is applicable for binary classificati

equation gets

Classification Loss

- For multi-class classification with M classes can be calculated as follows:



$$\text{Cross-entropy loss} = -\sum_{i=1}^{M} y_{i,j} \log \widehat{y_{i,j}}$$

Cost/Loss Functions

- **Regression Cost Function**
  - Regression models deal with predicting a continuous value for example salary of an employee, price of a car, loan prediction, etc.
  - A cost function used in the regression problem is called "Regression Cost Function".
  - They are calculated on the distance-based error as follows:
  - **Error = y-y'**
    - y – Actual Input
    - y' – Predicted output

- **MSE/L2/ Quadratic Loss**
- Means Square error is one of the most commonly used Cost function methods.
- Because of the square of the difference, it avoids any possibility of negative error.
- The formula for calculating MSE is given below:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \widehat{y_i}\right)^2$$

- Mean squared error is also known as **L2 Loss**.
- In MSE, **each error is squared, and it helps in reducing a small deviation in prediction as compared to MAE.**
- But if the dataset has outliers that generate more prediction errors, then squaring of this error will further increase the error multiple times. **Hence, we can say MSE is less robust to outliers**

- **Mean Absolute Error (MAE)**
- Mean Absolute error also **overcome the issue of the Mean Square error cost function by taking the absolute difference between the actual value and predicted value**.
- The formula for calculating Mean Absolute Error is given below:

$$\mathrm{MAE} = \frac{1}{n}\left|y_i - \widehat{y_i}\right|$$

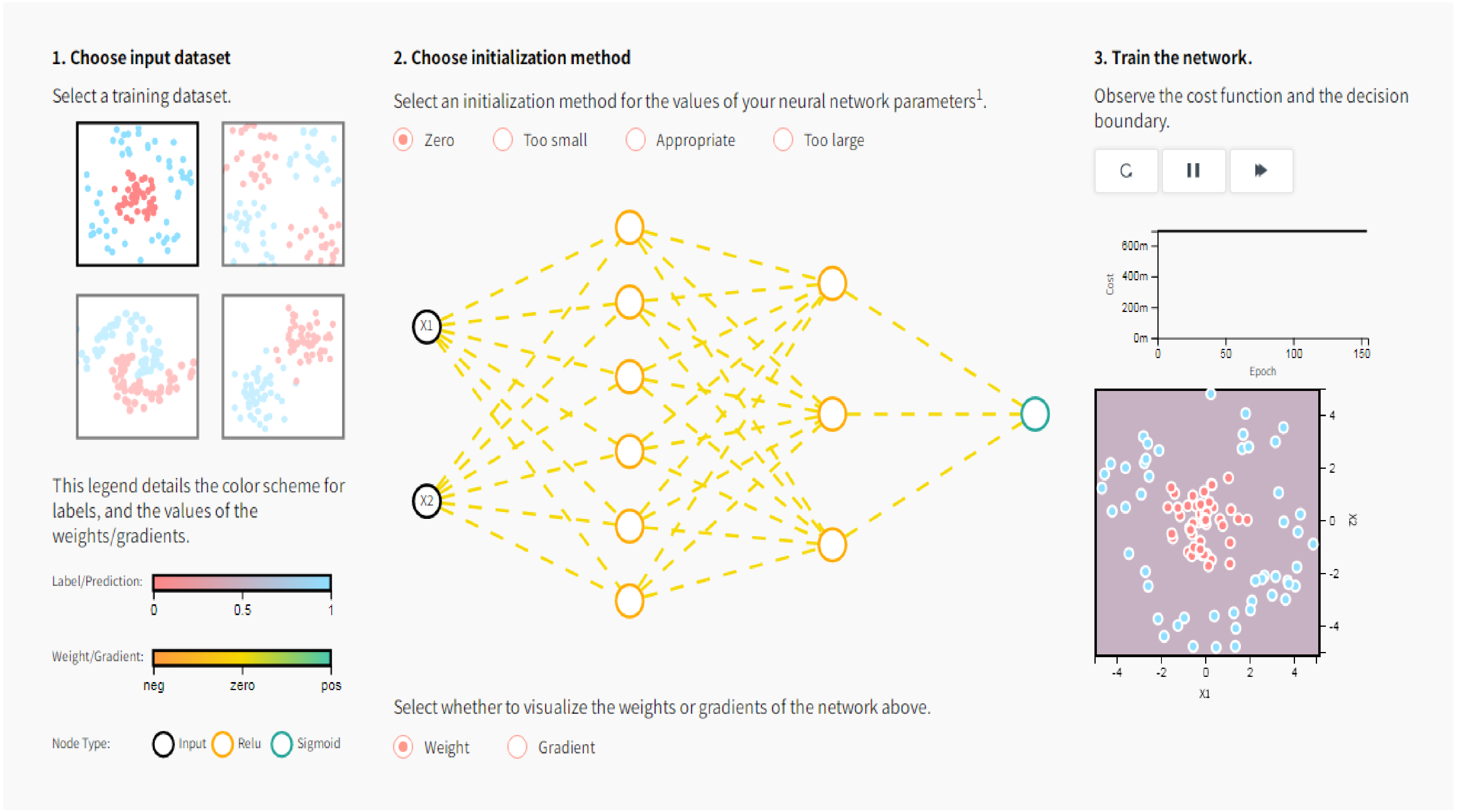
- This means the Absolute error cost function is also known as **L1 Loss**. It is not affected by noise or outliers, hence giving better results if the dataset has noise or outlier.

Cost Functions

- The design of a deep neural network is the choice of the cost function.
- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- The cross-entropy between the training data and the model’s predictions as the cost function.
- The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term.

Weight Initialization

https://www.deeplearning.ai/ai-notes/initialization/index.html



1. Choose input dataset

Select a training dataset.

This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 — 0.5 — 1

Weight/Gradient: neg — zero — pos

Node Type: Input, Relu, Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

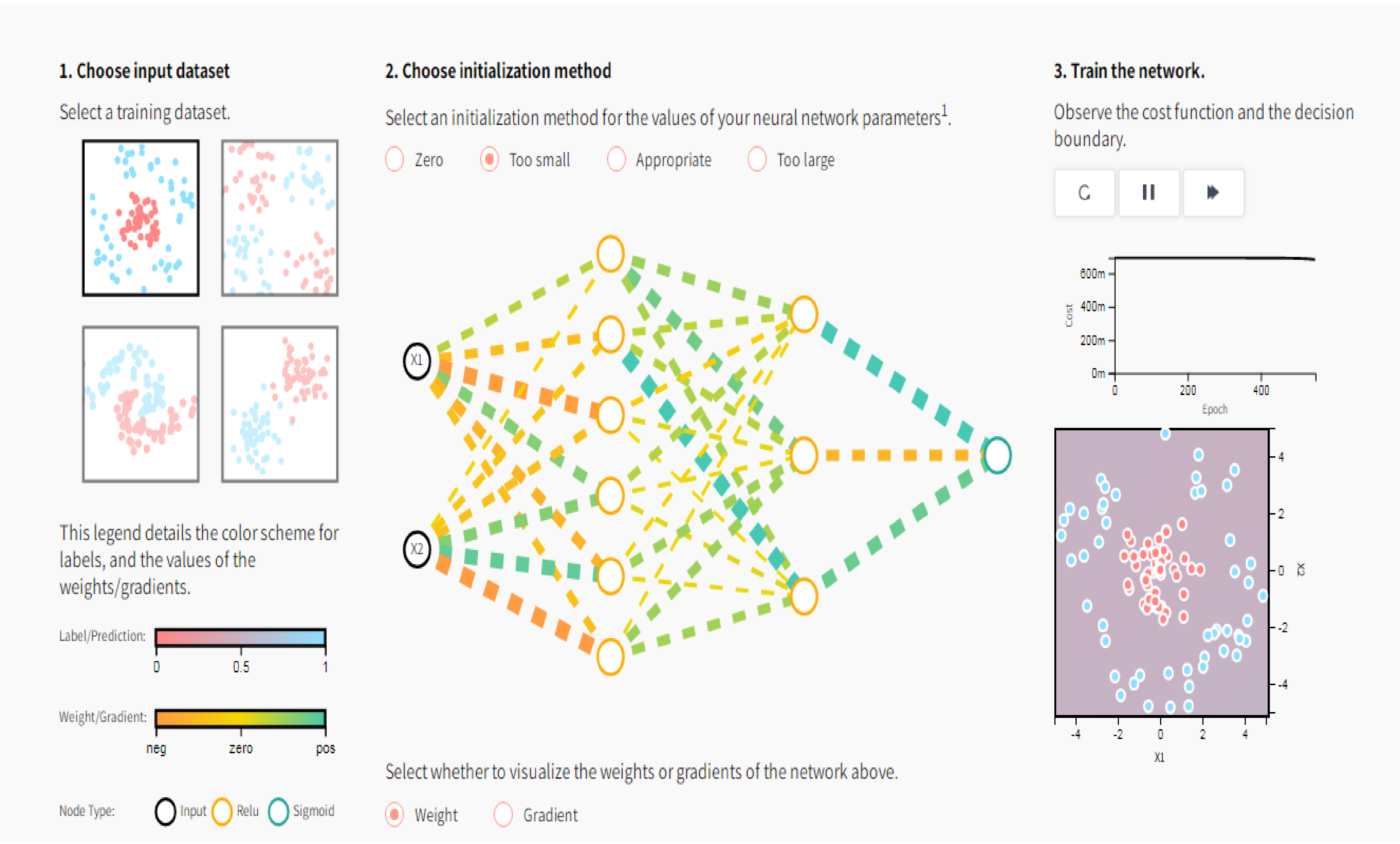
[✓ Zero] Too small Appropriate Too large

Select whether to visualize the weights or gradients of the network above.

[✓ Weight] Gradient

3. Train the network.

Observe the cost function and the decision boundary.



1. Choose input dataset

Select a training dataset.

This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

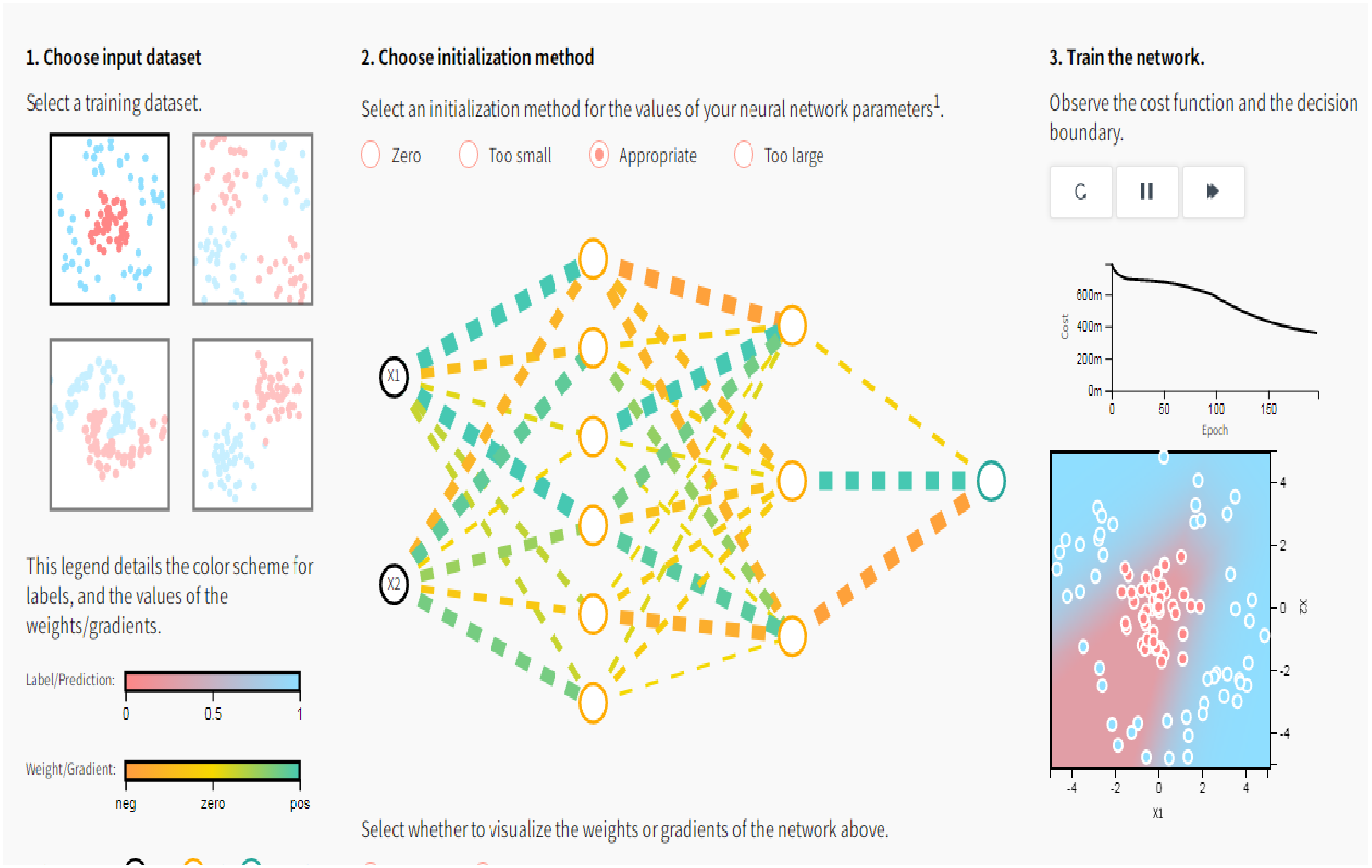
Zero [✓ Too small] Appropriate Too large

Select whether to visualize the weights or gradients of the network above.

[✓ Weight] Gradient

3. Train the network.

Observe the cost function and the decision boundary.



1. Choose input dataset

Select a training dataset.

This legend details the color scheme for labels, and the values of the weights/gradients.

2. Choose initialization method

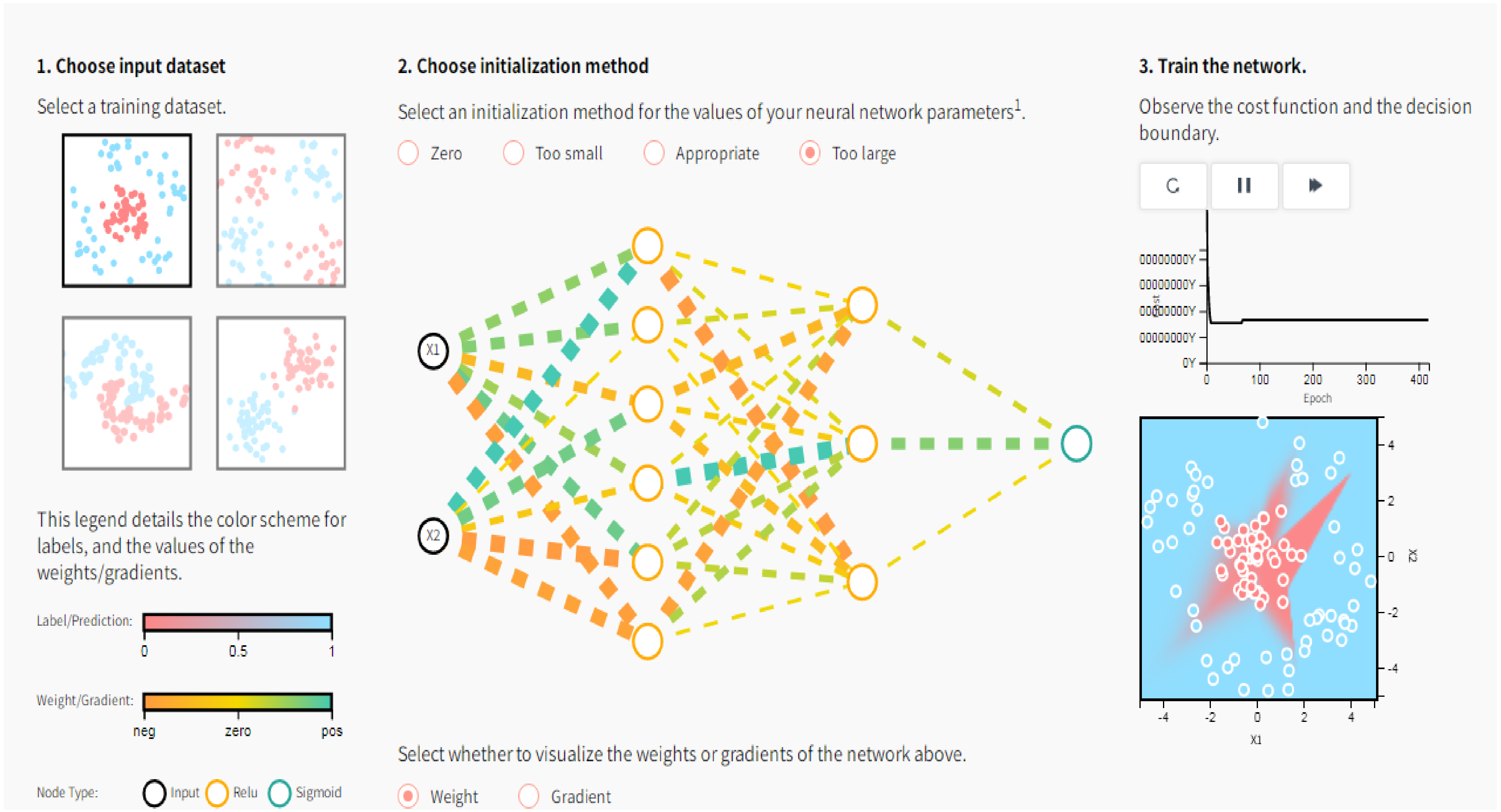
Select an initialization method for the values of your neural network parameters[1].

Zero Too small [✓ Appropriate] Too large

Select whether to visualize the weights or gradients of the network above.

3. Train the network.

Observe the cost function and the decision boundary.



1. Choose input dataset

Select a training dataset.

This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

Zero Too small Appropriate [✓ Too large]

Select whether to visualize the weights or gradients of the network above.

[✓ Weight] Gradient

3. Train the network.

Observe the cost function and the decision boundary.

Weight Initialization

Zero

- *Initializing all the weights with **zeros leads the neurons to learn the same features during training***
- Since the value is same for all neurons **, all the neurons would be symmetric(between neuron and all its subsequent connections) and will receive same updates** .
- We want **each** neuron to **learn a certain feature** and this initialization technique wont let that happen

Weight Initialization

***Zero* or Random Constant**

- Consider a *neural network* with two hidden units, and assume we initialize all the biases **to 0 and the weights with some constant α.**
- **W^k_i=c for all i,** k , that is weights of all layers and in between any two **nodes are zero is same and has the value c**
- If we forward propagate an input (x_1,x_2) in this network, the output of both hidden units will be *relu*(αx1+αx2).
- Thus, both hidden units will have identical influence on the cost, which will **lead to identical gradients.**
- Thus, both neurons will evolve symmetrically **throughout training, effectively preventing different neurons from learning different things**

Weight Initialization

A too-large initialization leads to exploding gradients

- If the gradients get **LARGER** as our backpropagation progresses, we would end up with exploding gradients **having big weight updates, leading to the divergence of the gradient descent algorithm**
- **Initial weights** assigned to the neural nets creating **large losses**.
- The gradients of the cost with the respect to the parameters are too big.
- **This leads the cost to oscillate around its minimum value.**
- Model weights can become NaN very quickly

Weight Initialization

A too-small initialization leads to vanishing gradients

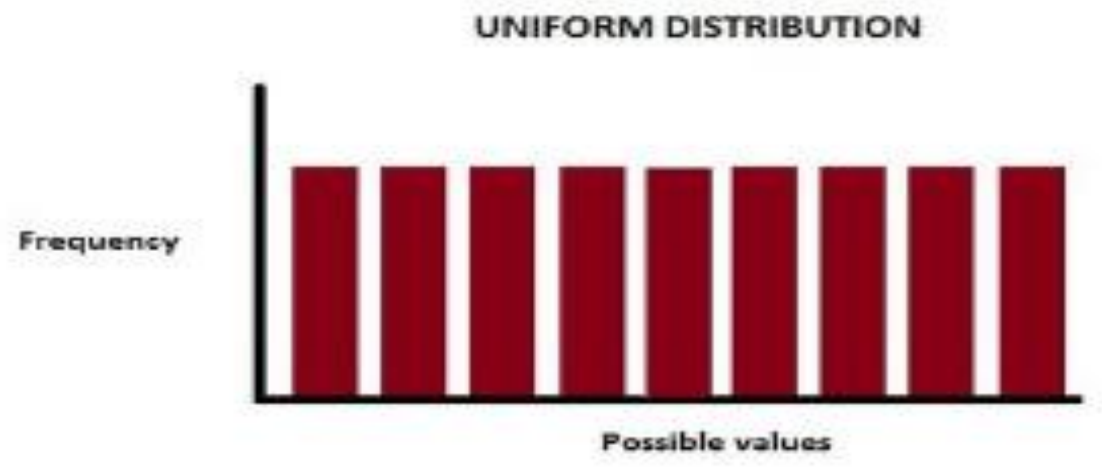
- The **gradients frequently become SMALLER until they are close to zero**, the new model weights (of the initial layers) will be virtually **identical to the old weights** without any updates.
- The gradient descent algorithm never converges to the optimal solution.
- It causes **unstable** behaviors of neural nets.
- The model learns **slowly and often times, training stops after a few iterations**

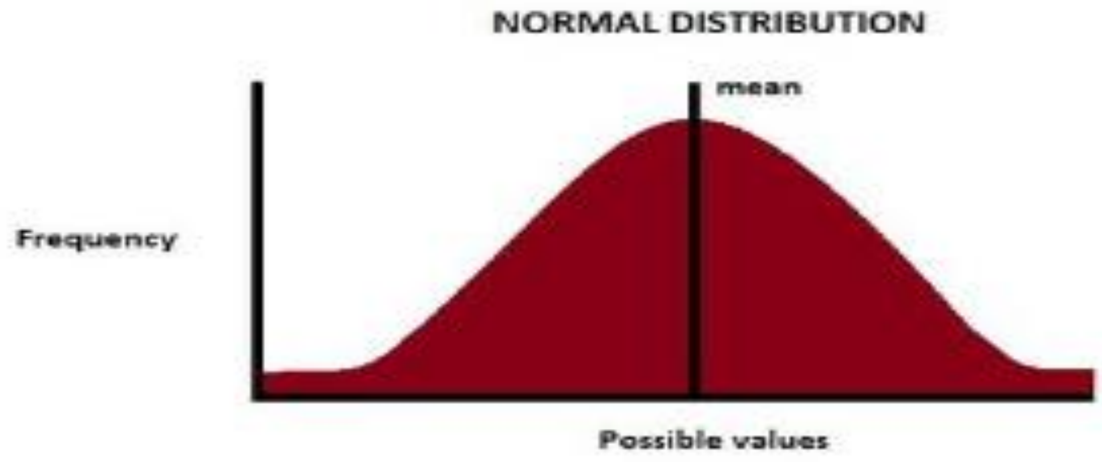
Weight Initialization

- **Uniform Initialization**
- In uniform initialization of weights , weights belong to a uniform distribution in range a,b with values of a and b as below.
- Works well for Sigmod Distribution

$$W \approx U(a,b) \qquad a= \frac{-1}{\sqrt{f\_in}} \ , \ b= \frac{1}{\sqrt{f\_in}}$$

- f_in -> number of inputs to that neuron


UNIFORM DISTRIBUTION
Frequency
Possible values


NORMAL DISTRIBUTION
mean
Frequency
Possible values

Weight Initialization

Xavier initialization or Glorot initialization

- Keeps the variance the same across every layer.
- Assume that our layer’s activations are normally distributed around zero.
- Glorot and Xavier had a belief that if they **maintain variance of activations in all the layers going forward and backward convergence will be fast.**
- Works well for tanh/ sigmoid

Weight Initialization

Xavier initialization or Glorot initialization

- In Xavier **Normal** Distribution, weights belong to normal distribution where mean is zero and standard deviation is as below

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in + f\_out}}$$

Weight Initialization

- **Xavier initialization or Glorot initialization**
- In Xavier **Uniform** Distribution , weights belong to uniform distribution in range as below:

$$W \approx U(a,b)$$

$$a = -\sqrt{\frac{6}{f\_in + f\_out}}, \quad b = \sqrt{\frac{6}{f\_in + f\_out}}$$

Weight Initialization

He/Kaiming

- **Kaiming Initialization**, or **He Initialization**, is an initialization method for neural networks that takes into account the non-linearity of activation functions, such as ReLU activations.
- Weights belong to normal distribution

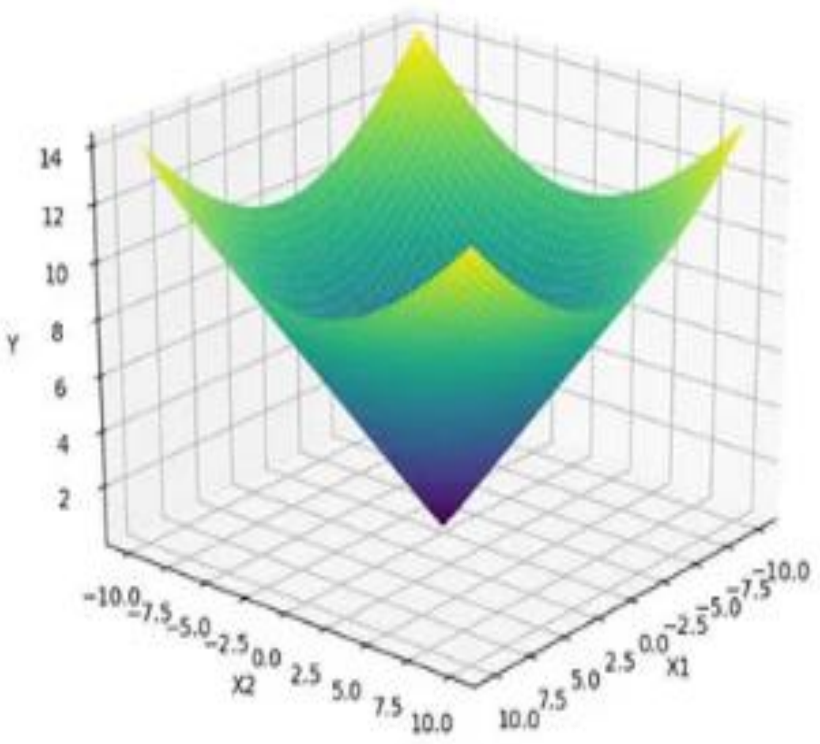
Benefits of Weight Initialization

- They serve as good starting points for weight initialization and they reduce the chances of exploding or vanishing gradients.
- Do not vanish or explode too quickly, as the weights are neither too much bigger than 1 nor too much less than 1.
- They help to avoid slow convergence and ensure that we do not keep oscillating off the minima.

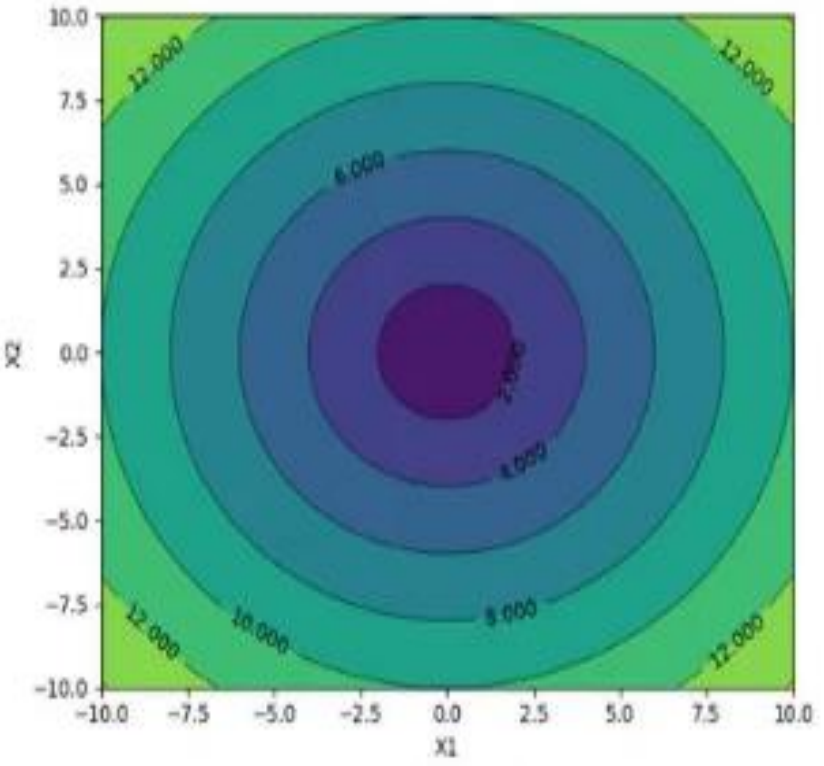
Sum it up…..

- Zero initialization causes the neuron to memorize the same functions almost in each iteration.
- Random initialization is a better choice to break the symmetry. However, initializing weight with much high or low value can result in slower optimization.
- Using an extra scaling factor in **Xavier initialization, He-et-al Initialization**, etc can solve the above issue to some extent. That’s why these are the more recommended weight initialization methods among all.

Representation of Gradient in form of 2D Contour



3D Plot

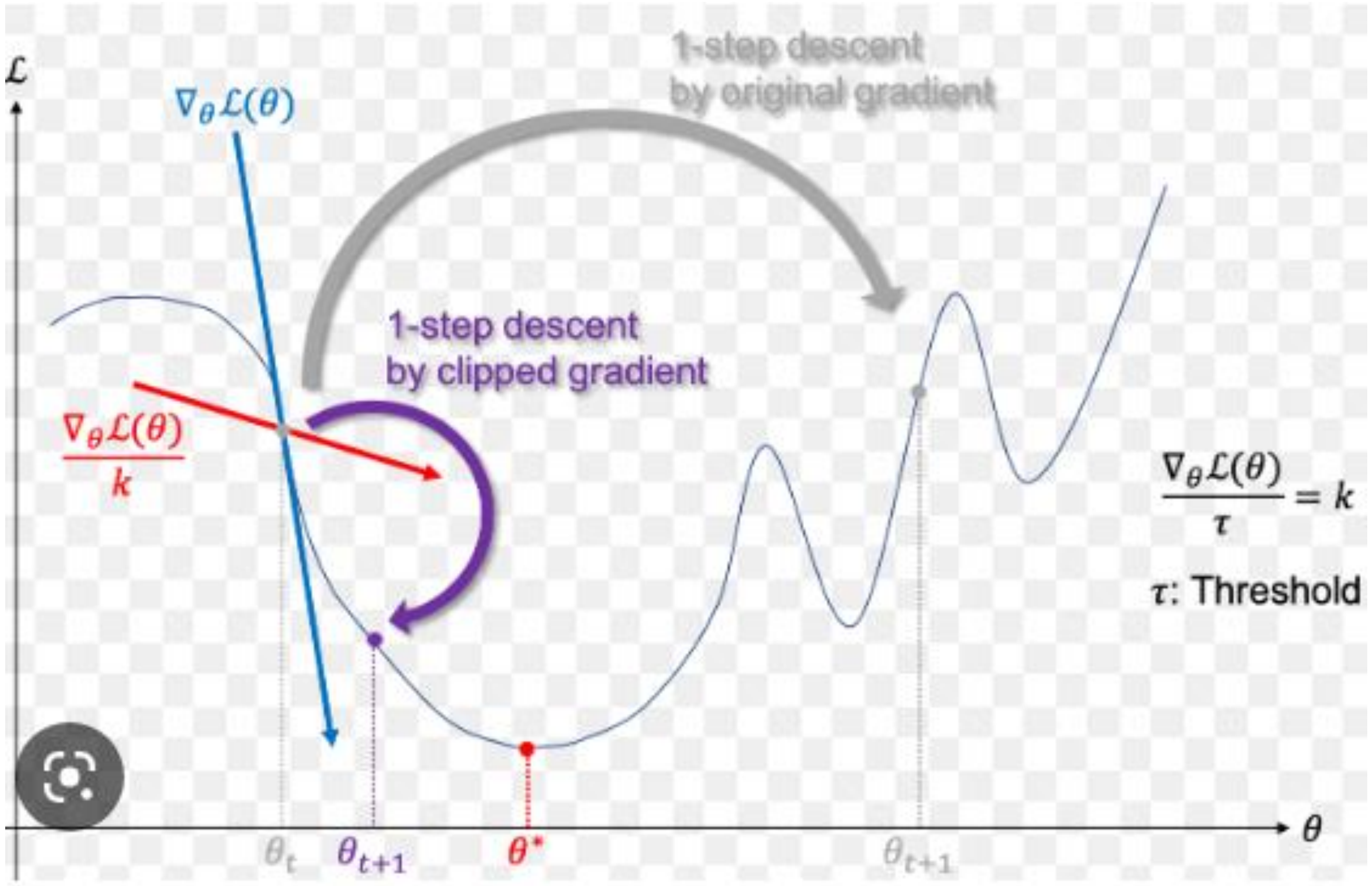


Contour Plot

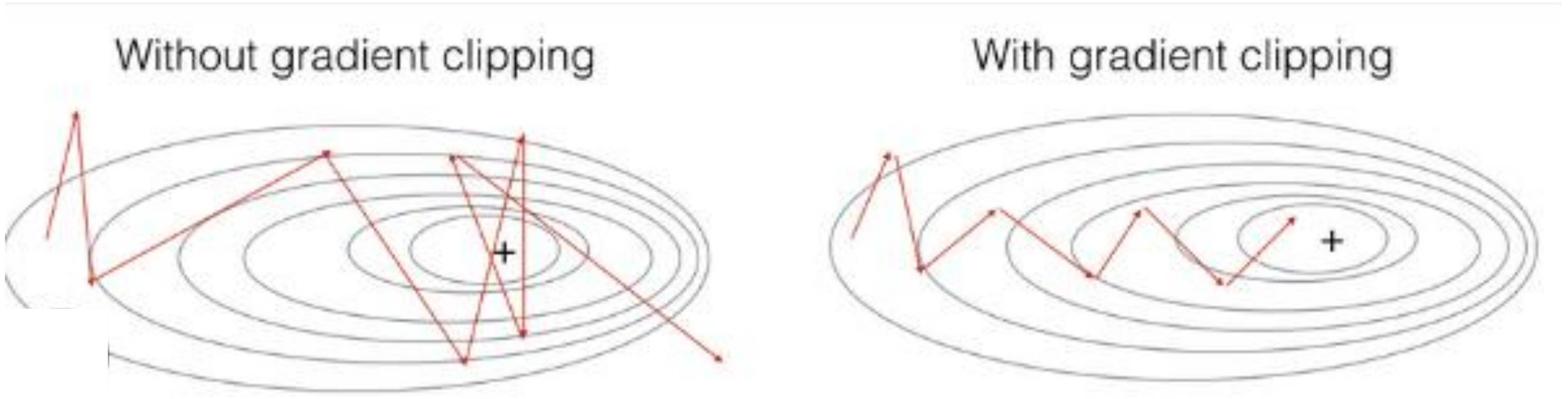
Gradient Clipping

- Gradient Clipping is **a method where the error derivative is changed or clipped to a threshold during backward propagation through the network, and using the clipped gradients to update the weights**.
- **Gradient clipping** is a technique to prevent exploding gradients in very deep networks, usually in recurrent neural networks.
- *When the traditional gradient descent algorithm proposes to make a **very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.***
- It is a method that only addresses the **numerical stability of training deep neural network** models and does **not** offer any general improvement in **performance**.

Gradient Clipping



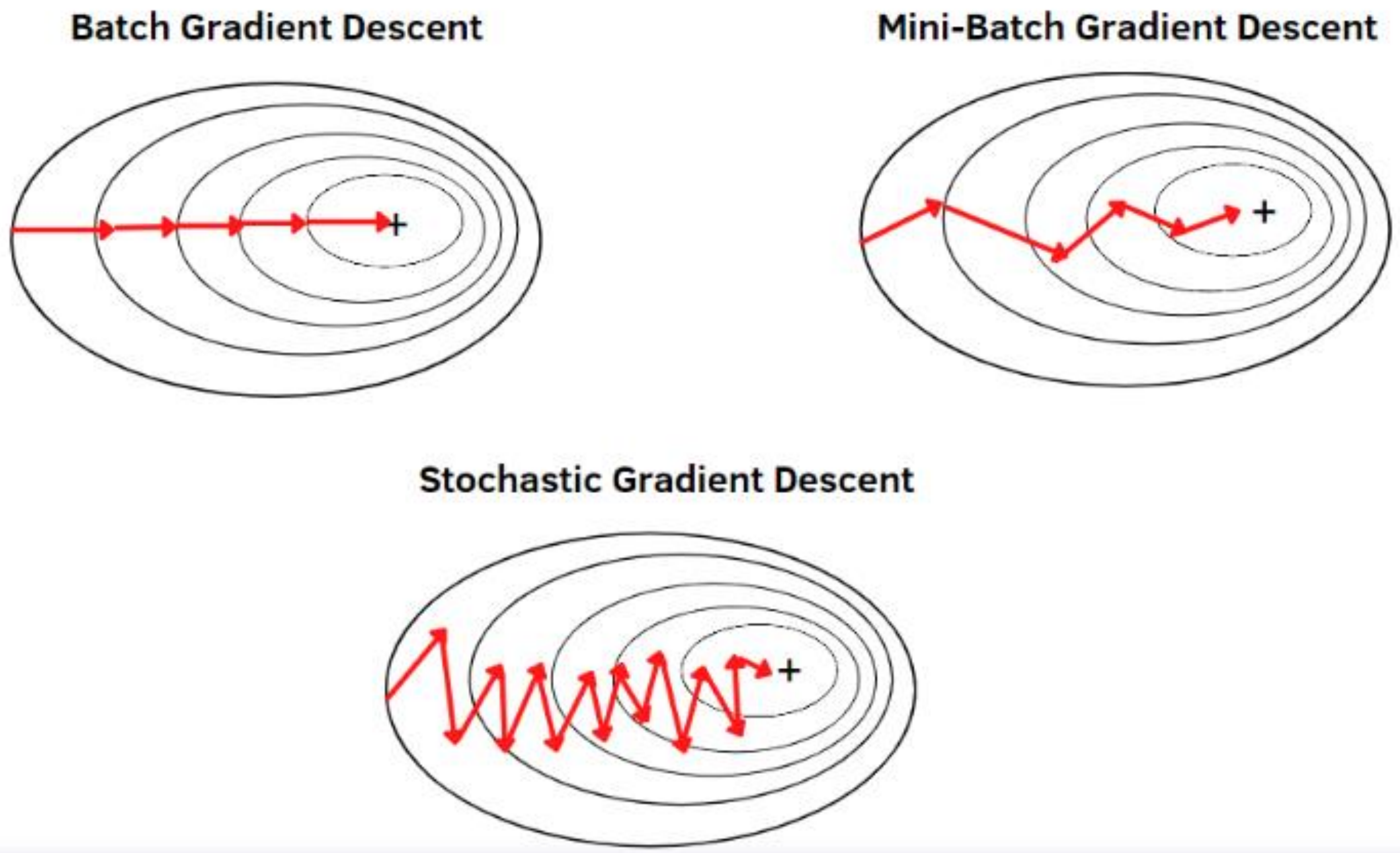
Gradient Clipping

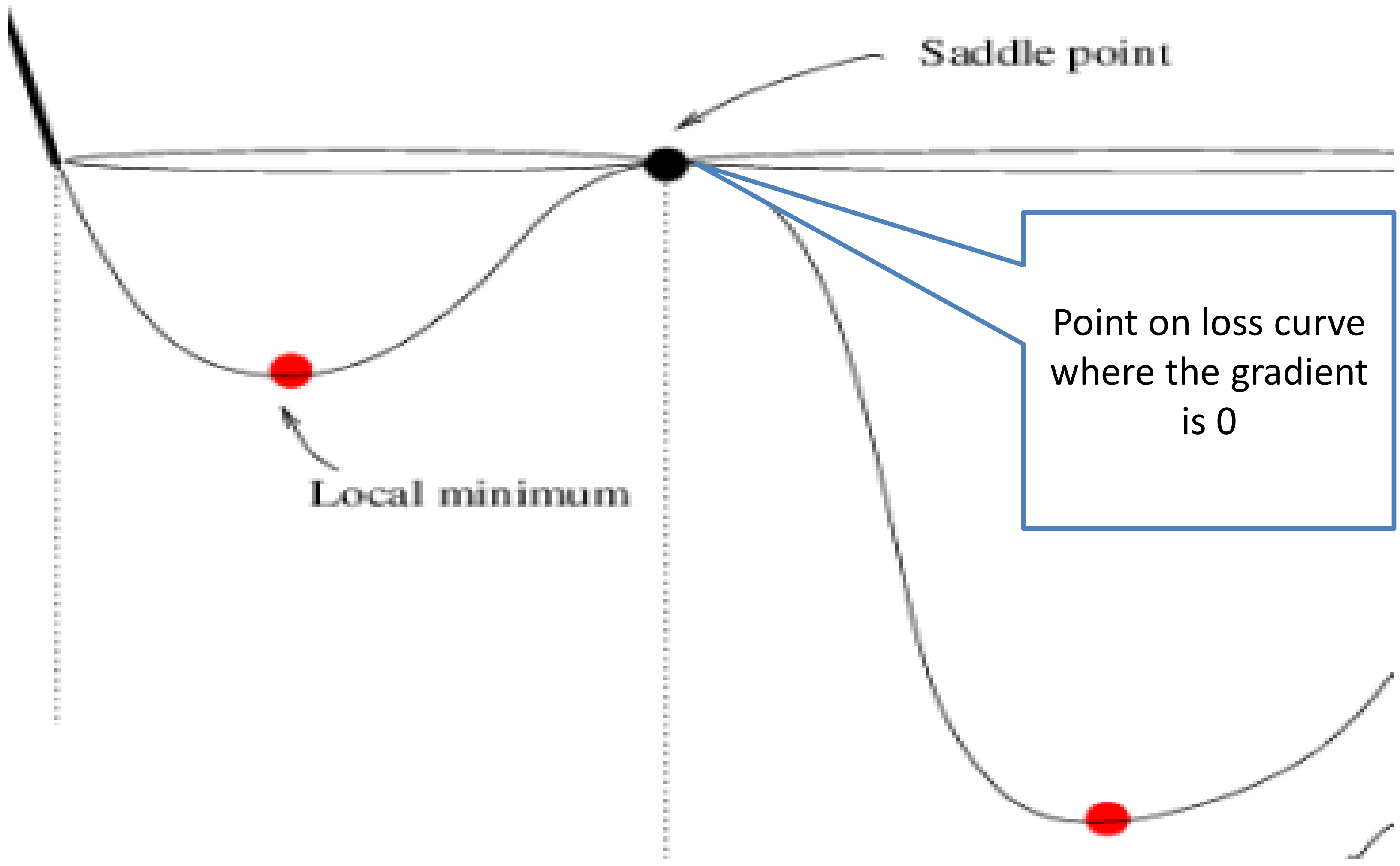


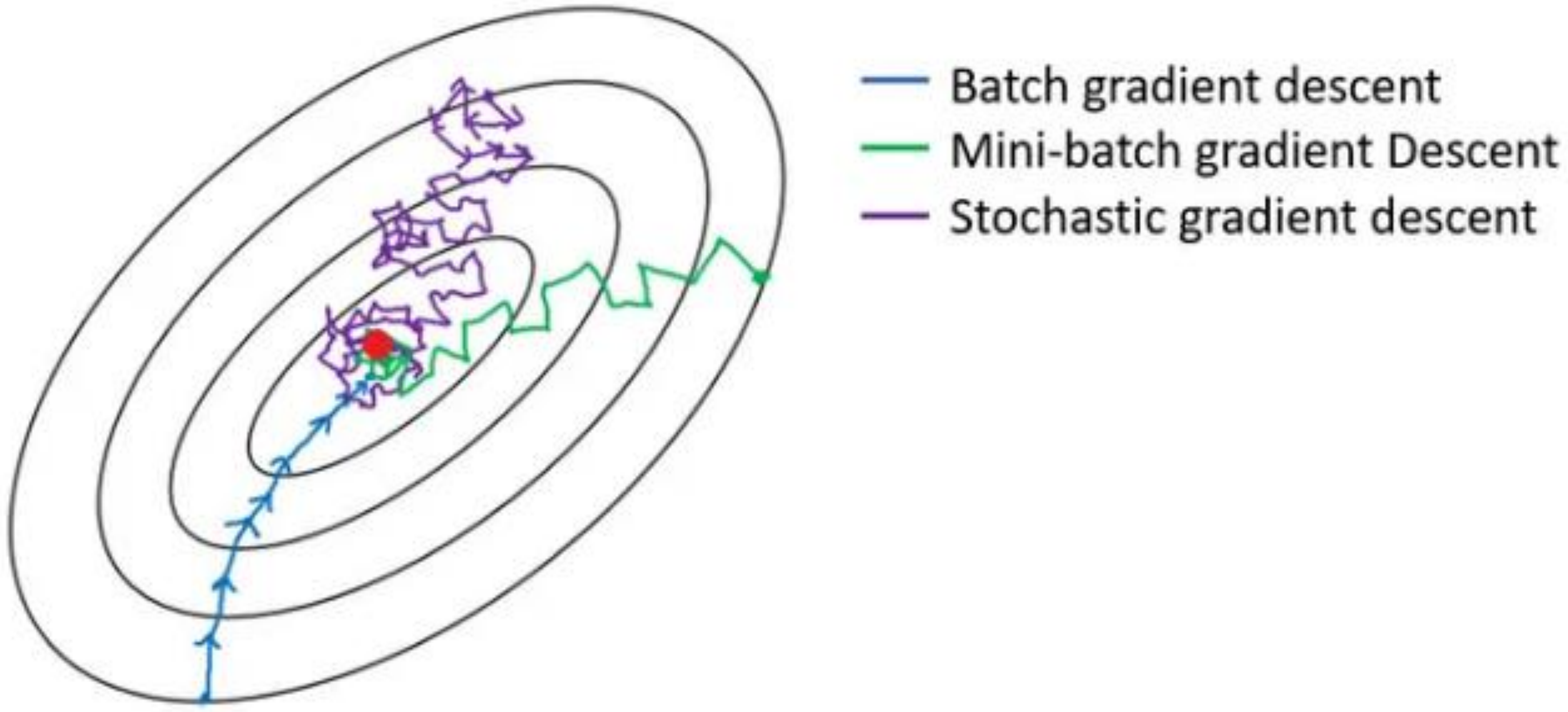
Challenging Optimization

- Training deep learning neural networks is very challenging.
- The best general algorithm known for solving this problem is stochastic gradient descent, where model weights are updated each iteration using the backpropagation of error algorithm.

Types of Gradient Descent




Saddle point
Local minimum
Point on loss curve where the gradient is 0

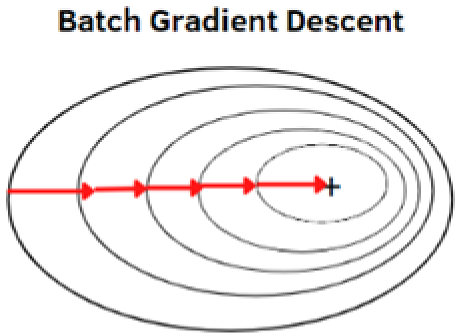

Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent

GRADIENT DESCENT

- Gradient descent is an **iterative algorithm whose purpose is to make changes to a set of parameters in hopes of reaching an optimal set of parameters that leads to the lowest loss function value** possible.
- The gradient descent algorithm is an **optimization algorithm mostly used in machine learning and deep learning**.
- Gradient descent adjusts parameters to minimize particular functions to local minima.
- The algorithm objective is to identify model parameters like weight and bias that reduce model error on training data.
- **A gradient measures how much the output of a function changes if you change the inputs a little bit.**
- In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the **gradient simply measures the change in all weights about the change in error.**

BATCH GRADIENT DESCENT

- Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset.
- Batch gradient descent calculates error Still, the model is not changed until every training sample has been assessed. The entire procedure is referred to as a cycle and a training epoch.
- **Batch gradient descent calculates the error based on each training record but updates the parameters after evaluating all training records.**
- Benefits of batch are its **computational efficiency, which produces a stable error gradient and a stable convergence.**



BATCH GRADIENT DESCENT

Advantages

- Fewer model updates mean that this variant of the steepest descent method is more **computationally efficient** than the stochastic gradient descent method.
- Reducing the update frequency provides a **more stable error gradient and a more stable convergence for some problems.**
- Separating forecast error calculations and model updates provides **a parallel processing-based algorithm implementation**.

Disadvantages

- A more stable error gradient can cause the model to **prematurely converge to a suboptimal set of parameters.**
- End-of-training epoch updates require the additional complexity of accumulating prediction errors across all training examples- **Local Minima**
- **Saddle Point Problem.**
- The batch gradient descent method typically requires the entire training dataset in **memory** and is implemented for use in the algorithm.
- **Large datasets** can result in very slow model updates or training speeds.
- Slow and require more computational power.

STOCHASTIC GRADIENT DESCENT (SGD)

- Stochastic gradient descent (SGD) **changes the parameters for each training sample one at a time for each training example in the dataset**. Depending on the issue, this can make SGD faster than batch gradient descent.
- One benefit is that the regular updates give us a **fairly accurate idea of the rate of improvement**.
- However, the batch approach is less computationally expensive than the frequent updates. The frequency of such updates can also produce **noisy gradients**, which could cause the error rate to fluctuate rather than gradually go down.

STOCHASTIC GRADIENT DESCENT (SGD)

Advantages

- **Overcome Saddle Point issue to some extent as compared to Gradient Descent.**
- Model’s performance and improvement rates with frequent updates.
- This variant of the steepest descent method is probably the easiest to understand and implement.
- Increasing the frequency of model updates will allow you to learn more about some issues faster.
- **Faster and require less computational power.**
- **Suitable for the larger dataset.**

STOCHASTIC GRADIENT DESCENT (SGD)

Disadvantages

- Frequent model updates are more **computationally intensive than other steepest descent configurations**, and it takes **considerable time to train the model with large datasets.**
- Frequent updates can result in noisy gradient signals. This can result in model parameters and cause errors to fly around (more variance across the training epoch).
- A noisy learning process along the error gradient can also make it difficult for the algorithm to commit to the model's minimum error.
- Suffers from problem of **oscillating trajectory of descent.**

MINI-BATCH GRADIENT DESCENT

- Mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique.
- It divides the training dataset into manageable groups and updates each separately. **This strikes a balance between batch gradient descent's effectiveness and stochastic gradient descent's durability.**
- Mini-batch sizes typically range from 50 to 256, although, like with other machine learning techniques, there is no set standard because **it depends on the application**.
- The most popular kind in deep learning, this method is used when training a neural network.

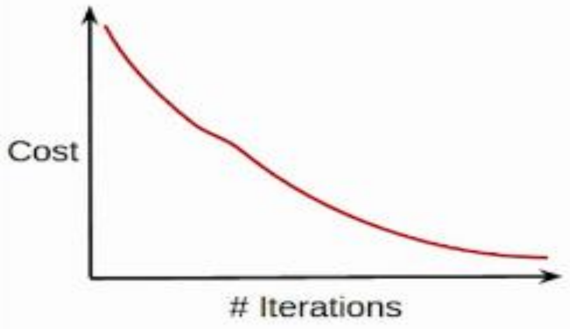
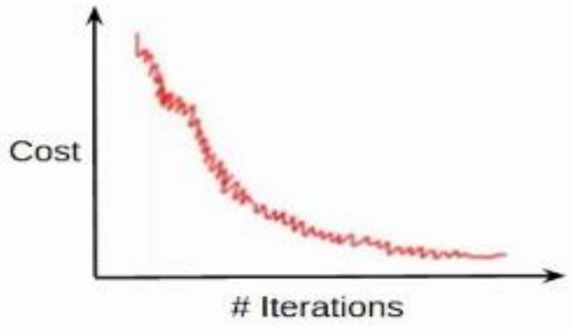
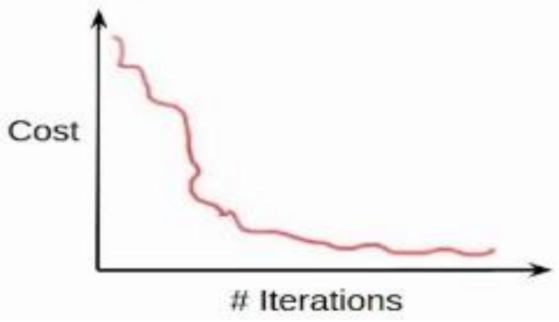
MINI-BATCH GRADIENT DESCENT

Advantages

- The model is updated more frequently than the gradient descent method, allowing for **more robust convergence and avoiding local minima.**
- Batch updates provide a **more computationally efficient process than stochastic gradient descent**.
- Batch processing allows for both the efficiency of not having all the training data in memory and implementing the algorithm.

Disadvantages

- **Mini-batch requires additional hyperparameters "mini-batch size" to be set for the learning algorithm.**
- Error information should be accumulated over a mini-batch of training samples, such as batch gradient descent.
- **It will generate complex functions.**

Batch Gradient Descent	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
• Entire dataset for updation	• Single observation for updation	• Subset of data for updation
• Cost function reduces smoothly	• Lot of variations in cost function	• Smoother cost function as compared to SGD
		
• Computation cost is very high	• Computation time is more	• Computation time is lesser than SGD • Computation cost is lesser than Batch Gradient Descent

Sum it up…

- **The mini-batch steepest descent method is the recommended method because it combines the concept of batch steepest descent with SGD**. Simply divide your training dataset into manageable groups and update each individually. This balances the effectiveness of batch gradient descent with the durability of stochastic gradient descent.
- When using batch gradient descent, adjustments are made after calculating the error for a certain batch**. One advantage of the batch gradient descent method is its computational efficiency, which produces a stable error gradient and a stable convergence**.
- Stochastic Gradient Descent (**SGD) sequentially modifies the parameters of each training sample in each training sample of the dataset. This allows SGD to be faster than batch gradient descent. One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement**.

Real World…

- Gradient Descent may never reach the exact minima. It reaches to points close to minima, hence loss value does not change much.
- Hence, networks are generally designed to run for a specific number of iterations (epochs) but stopped **when the loss values stop to improve.**

Other Optimization Algorithms

- Gradient Descent with Momentum
- Adagrad
- Adadelta
- RMSprop
- Adam

Gradient Descent with Momentum

- Momentum is an extension to the gradient descent optimization algorithm, often referred to as **gradient descent with momentum**.
- **It is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g.** result in a better final result.
- A problem with the gradient descent algorithm is that the progression of the search can **bounce around the search space based on the gradient.**
- For example, the search may progress downhill towards the minima, but during this progression, it may move in another direction, even uphill, depending on the gradient of specific points (sets of parameters) encountered during the search.
- This can **slow down** the progress of the search, especially for those optimization problems where the broader trend or shape of the search space is more useful than specific gradients along the way.

Gradient Descent with Momentum

- **Agenda: To add history to the parameter update equation based on the gradient encountered in the previous updates.**
- This change is based on the metaphor of momentum from physics where **acceleration in a direction can be accumulated from past updates.**
- Momentum involves adding an additional hyperparameter that controls the amount of history (momentum) to include in the update equation, i.e. the step to a new point in the search space.
- The value for the hyperparameter is defined in the range 0.0 to 1.0 and often has a value close to 1.0, such as 0.8, 0.9, or 0.99. A momentum of 0.0 is the same as gradient descent without momentum.

Gradient Descent with Momentum

- The target function *f()* returns a score for a given set of inputs, and the derivative function *f'()* gives the derivative of the target function for a given set of inputs. A starting point (*x*)
- We move against the gradient

  x = x – step_size * f'(x)
- Gradient descent update equation down into two parts**: the calculation of the change to the position and the update of the old position to the new position.**
- The change in the parameters is calculated as the gradient for the point scaled by the step size.

  change_x = step_size * f'(x)
- The new position is calculated by simply subtracting the change from the current point

  x = x – change_x

Gradient Descent with Momentum

- The update at the current iteration (t) will add the change used at the previous iteration (t-1) weighted by the momentum hyperparameter, as follows:

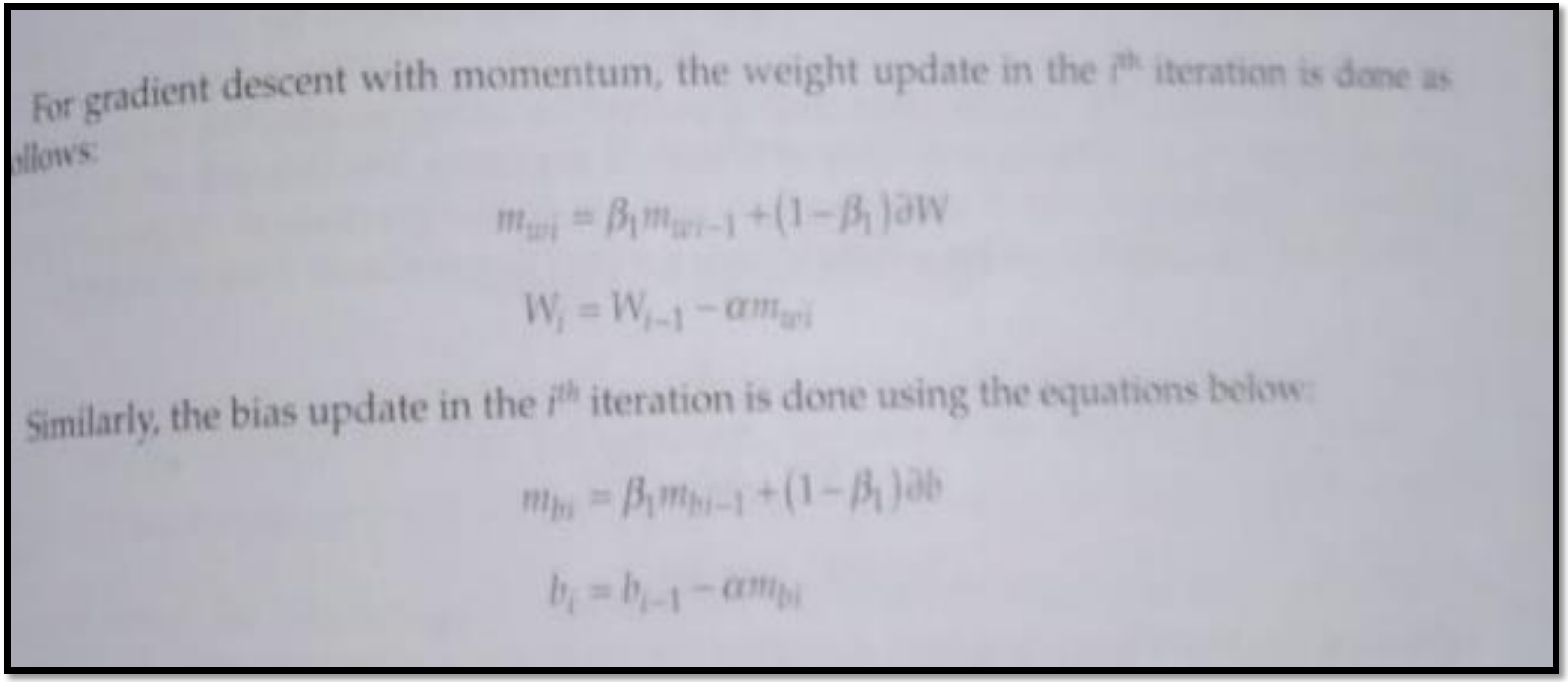
  change_x(t) = step_size * f'(x(t-1)) + momentum * change_x(t-1)
- The update to the position is then performed as before.

  x(t) = x(t-1) – change_x(t)
- The change in the position accumulates magnitude and direction of changes over the iterations of the search, proportional to the size of the **momentum hyperparameter**.
- Eg: A large momentum (e.g. 0.9) will mean that the update is strongly influenced by the previous update, whereas a modest momentum (0.2) will mean very little influence.

Gradient Descent with Momentum

- Momentum has the effect of dampening down the change in the gradient and, in turn, the step size with each new point in the search space.
- Momentum is most useful in optimization problems where the objective function has a **large amount of curvature (e.g. changes a lot), meaning that the gradient may change a lot over relatively small regions of the search space**.
- Momentum is helpful when the search space **is flat or nearly flat, e.g. zero gradient.**

Gradient Descent with Momentum



For gradient descent with momentum, the weight update in the i^{th} iteration is done as follows:

$$m_{wi} = \beta_1 m_{wi-1} + (1-\beta_1)\partial W$$

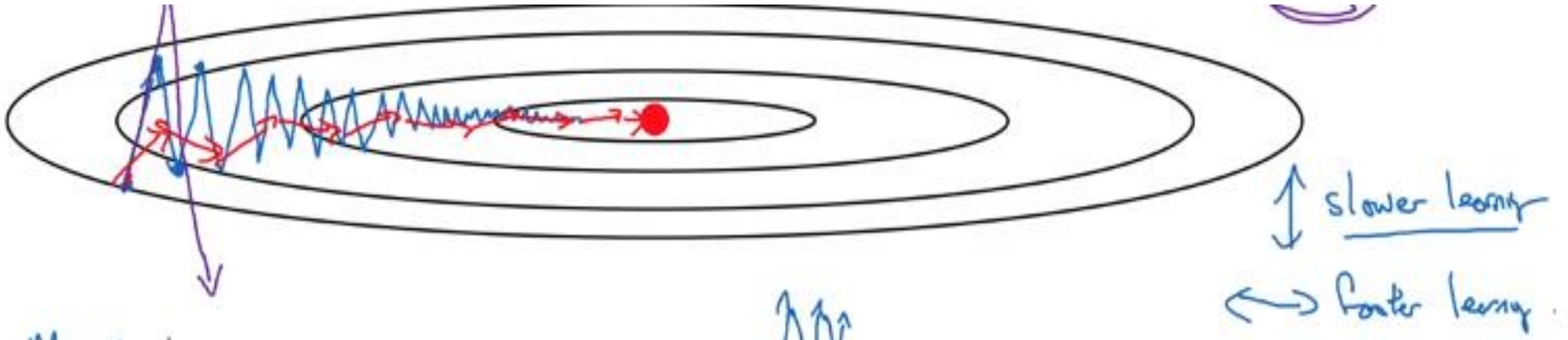
$$W_i = W_{i-1} - \alpha m_{wi}$$

Similarly, the bias update in the i^{th} iteration is done using the equations below:

$$m_{bi} = \beta_1 m_{bi-1} + (1-\beta_1)\partial b$$

$$b_i = b_{i-1} - \alpha m_{bi}$$

- β_1 – Momentum hyperparameter
- **Generally- 0.9**



Momentum:

On iteration t:

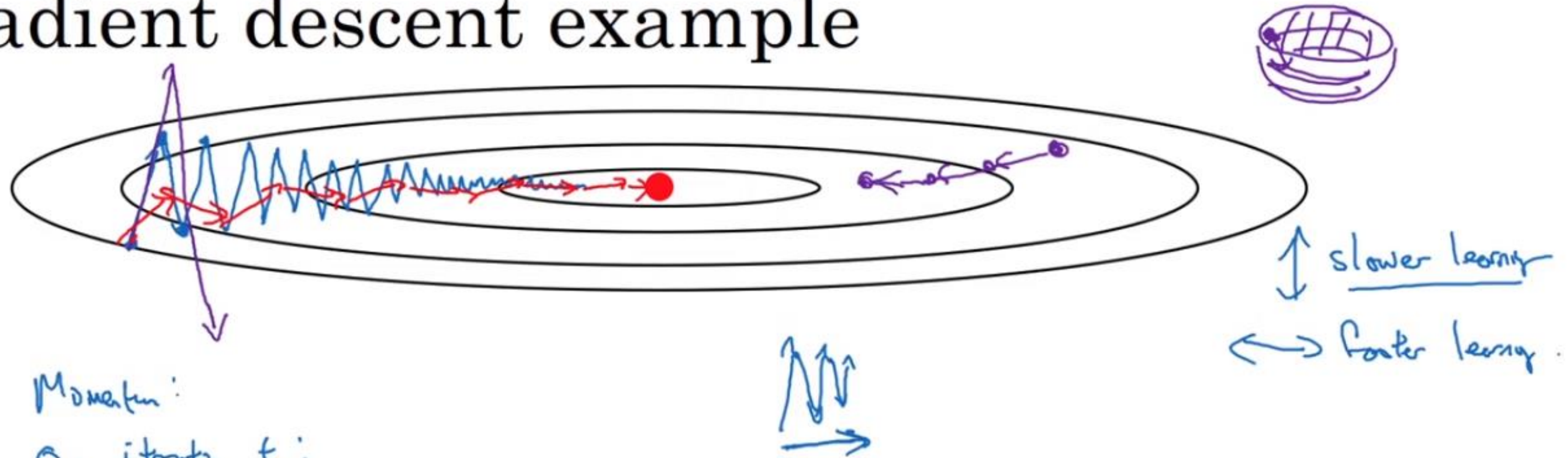
Compute dW, db on current mini-batch.

$V_{dW} = \beta V_{dW} + (1-\beta)\underline{dW}$ "$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

$V_{db} = \beta V_{db} + (1-\beta) db$

$W := W - \alpha V_{dW}$, $b := b - \alpha V_{db}$

Gradient descent example



Momentum:

On iteration t:

Compute dW, db on current mini-batch.

$V_{dW} = \beta V_{dW} + (1-\beta)dW$

$V_{db} = \beta V_{db} + (1-\beta)db$

friction → β ← velocity ← acceleration

"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

$W := W - \alpha V_{dW}$, $b := b - \alpha V_{db}$

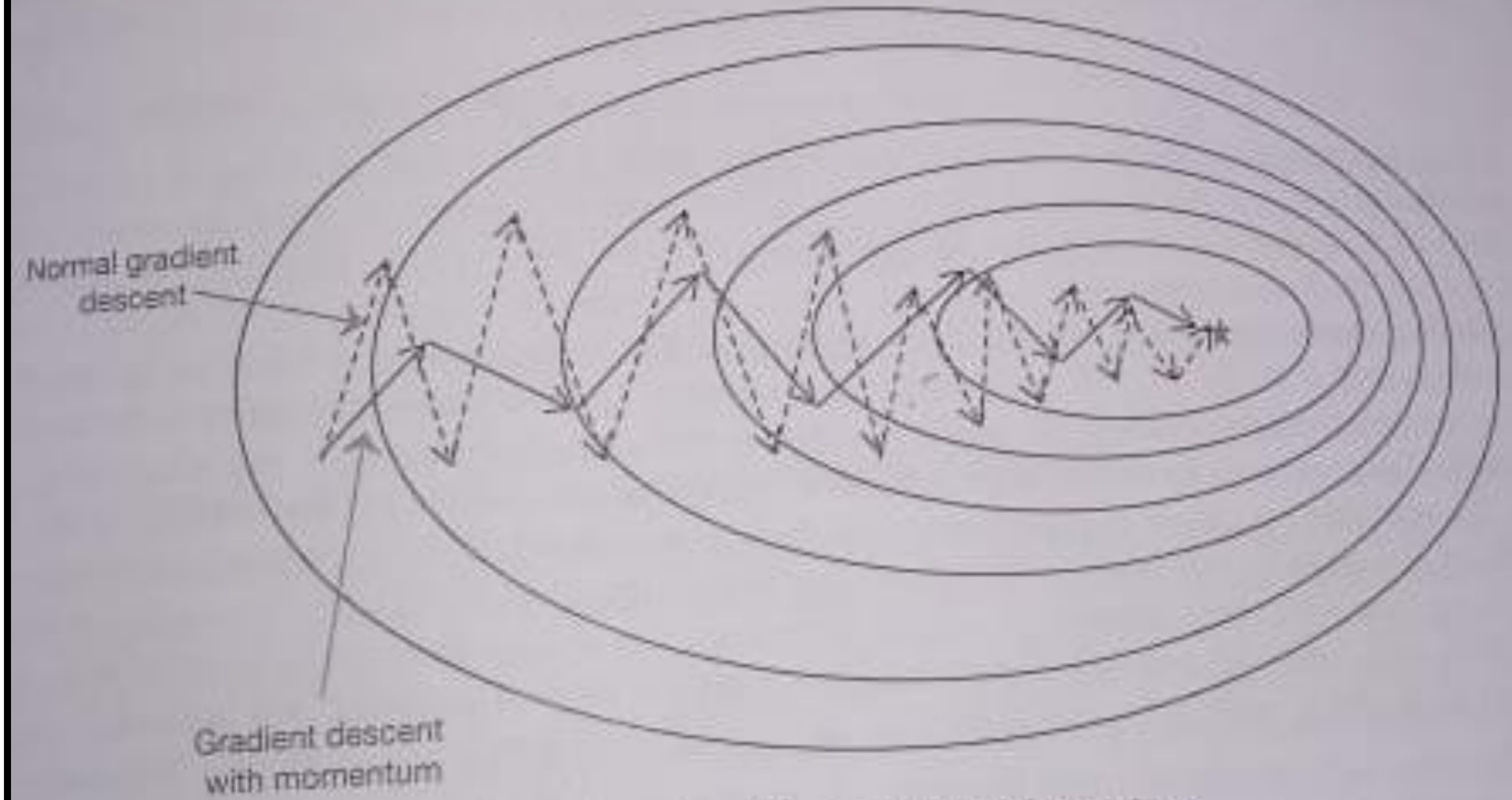
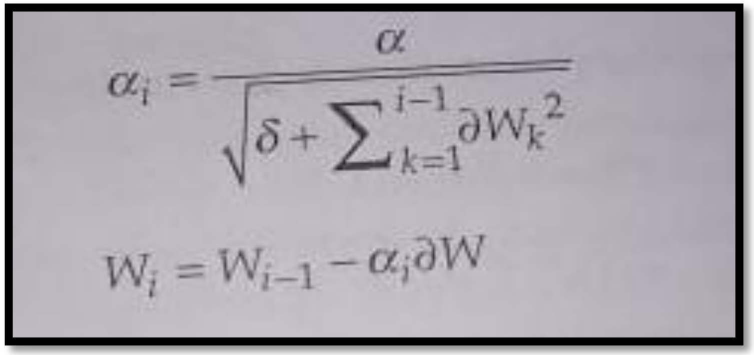


FIGURE 5.19 Gradient Descent with Momentum

Adagrad Optimizer

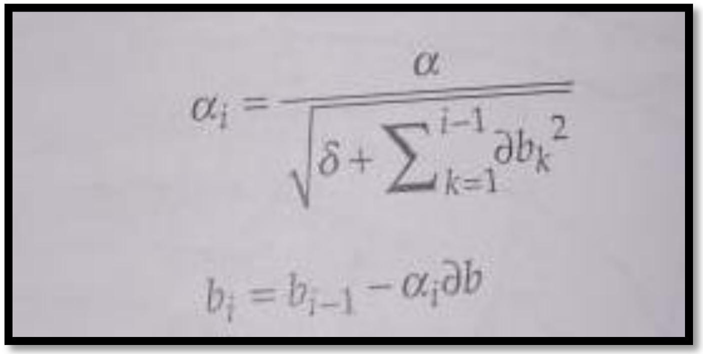
- **Adagrad** stands for **Adaptive Gradient Optimizer**.
- **Issue in GD with Momentum**: Learning Rate is same. For higher Learning rates, there is possibility of overshooting minima. If LR is lower, then takes long time to train.
- Hence, during initial iterations of GD, LR needs to be higher. As GD iterations progress and come closer to minima, the LR needs to be reduced.
- **Adagrad helps in automating the LR tuning process by using an adaptive LR tuning process using an adaptive LR method based on an accumulative value of historical gradients.**

Adagrad Optimizer



$$\alpha_i = \frac{\alpha}{\sqrt{\delta + \sum_{k=1}^{i-1} \partial W_k^2}}$$

$$W_i = W_{i-1} - \alpha_i \partial W$$

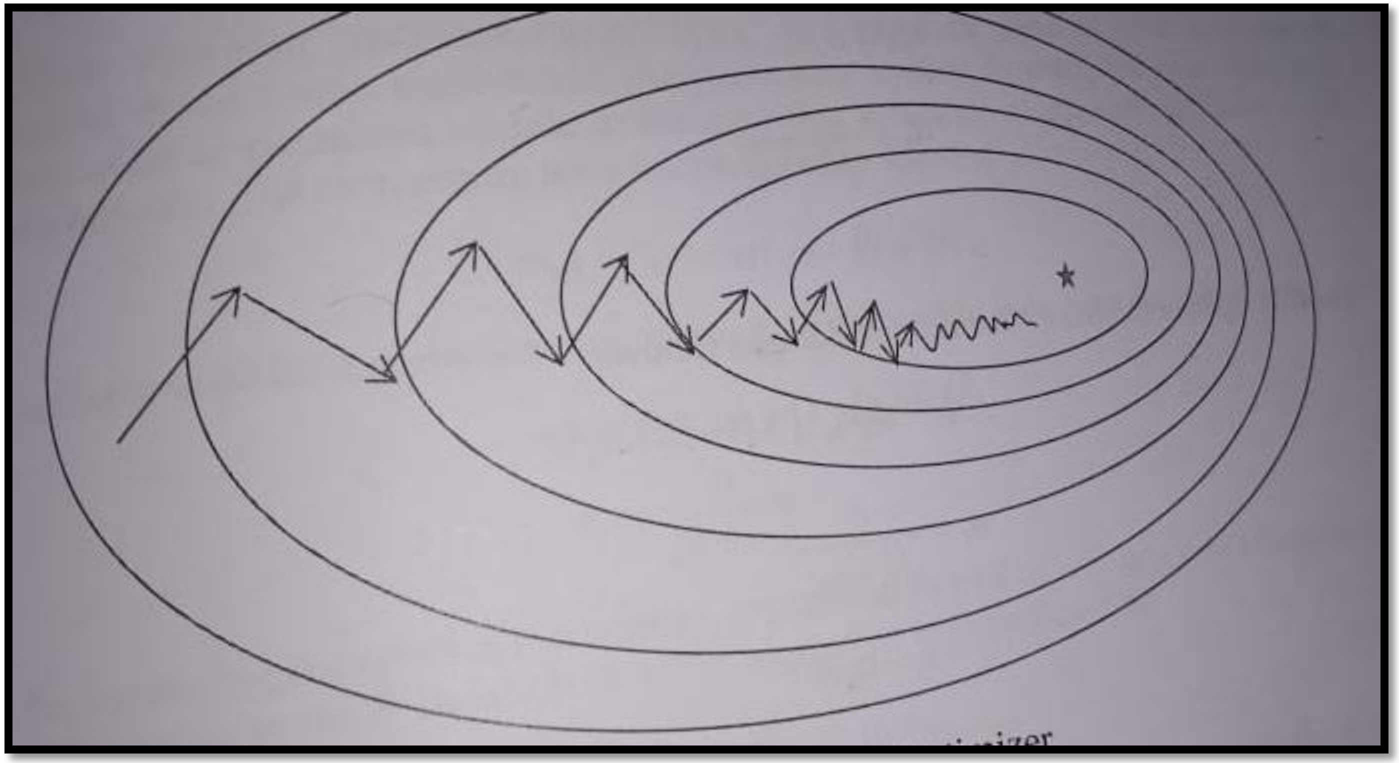


$$\alpha_i = \frac{\alpha}{\sqrt{\delta + \sum_{k=1}^{i-1} \partial b_k^2}}$$

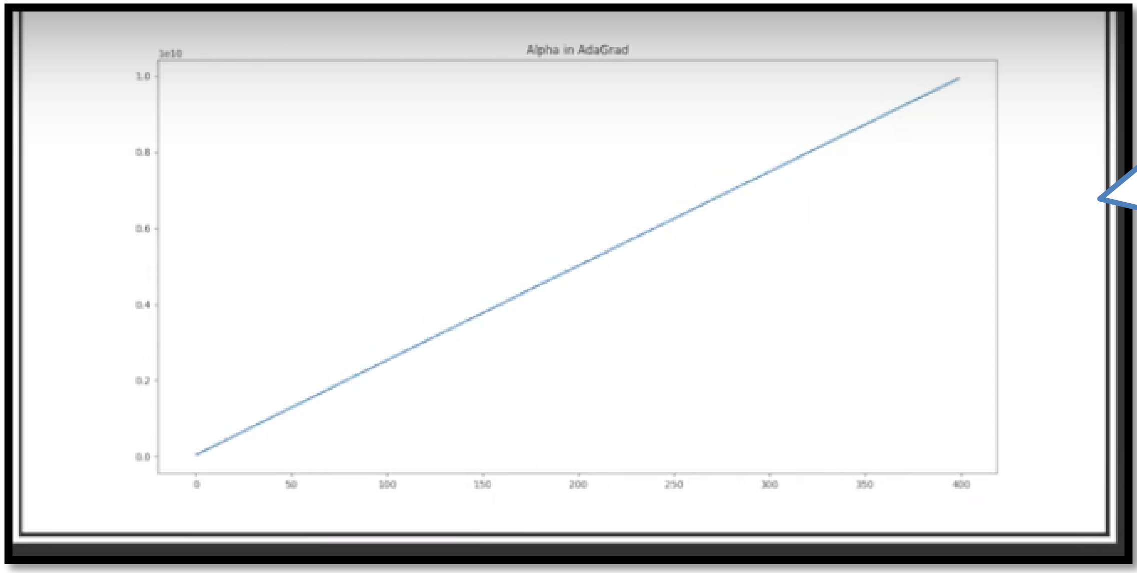
$$b_i = b_{i-1} - \alpha_i \partial b$$

- Value of delta is very small 10^{-5} to 10^{-7} to avoid divide by 0
- It is suitable for **sparse data** .

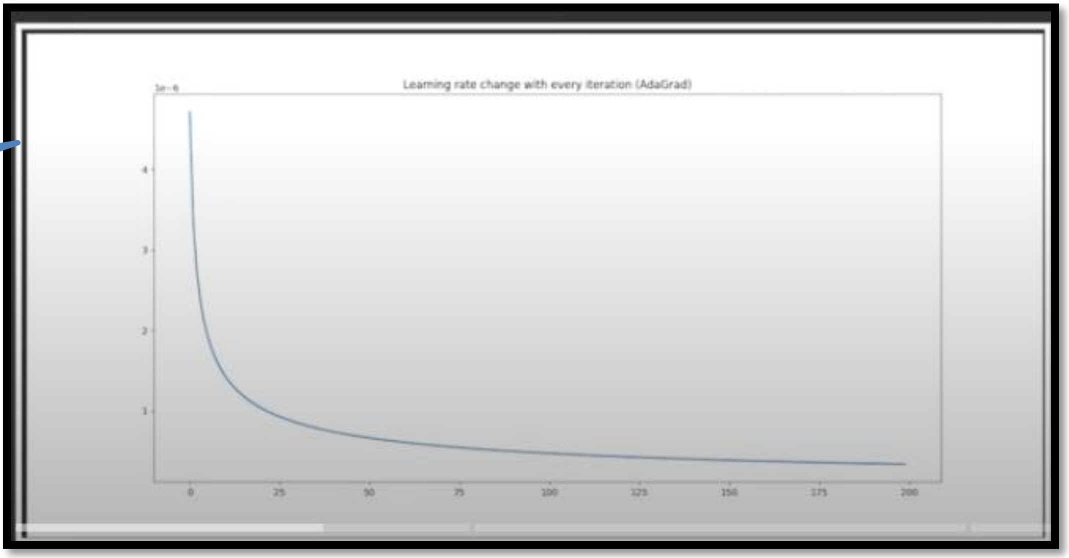
Adagrad Optimizer



- Limitation: **Learning rate decay over iterations due to sum of squared gradients in denominator.** Hence, after certain iterations algorithm remains stuck forever.



Alpha in Adagrad



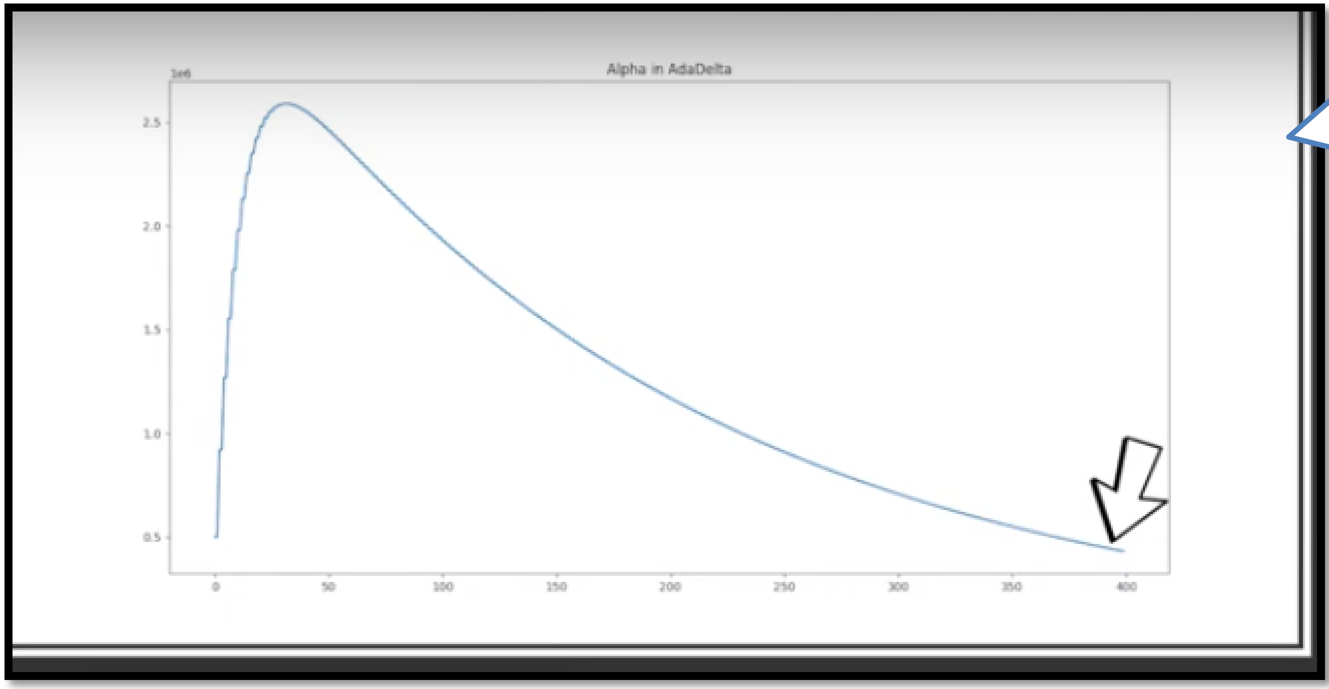
Learning Rate decay over iterations in Adagrad

AdaDelta Algo

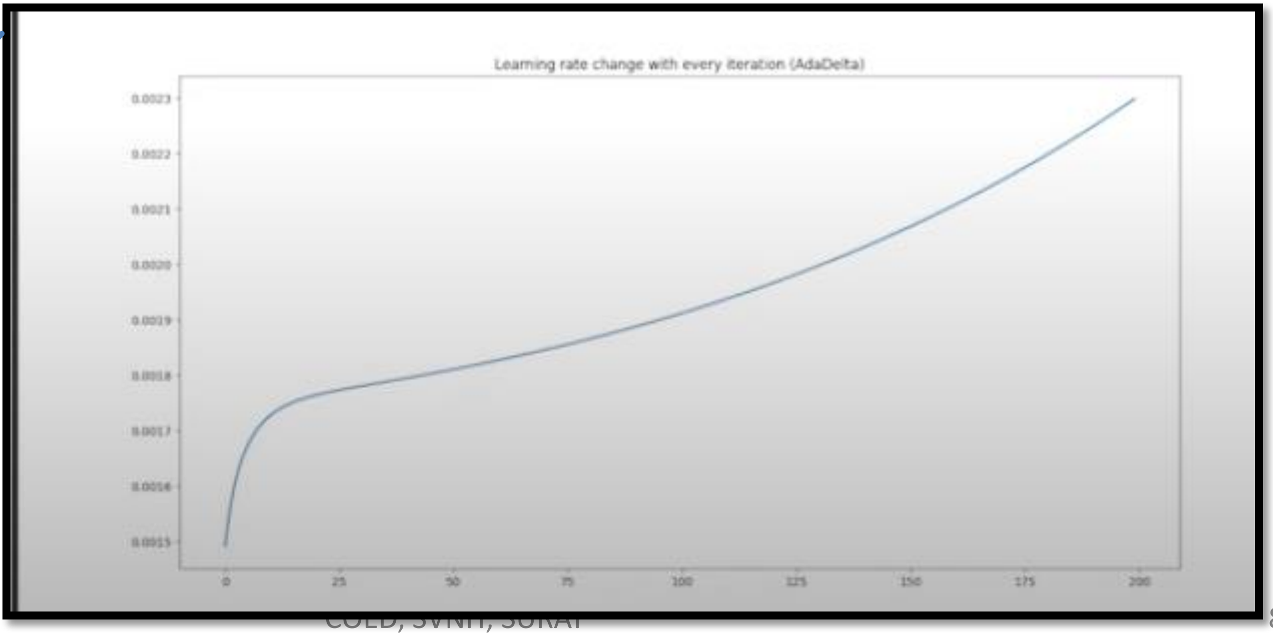
- Adadelta is similar to Adagrad Algorithm.
- It makes a small modification in calculation of weights across different iterations.
- **Instead of squared gradients of all past iterations, it takes the gradients for a specific windows or number of iterations.**
- Eg: If window size used is 3, it will take squared gradients only for 3 preceding iterations.
- **This will help in addressing the issue faced by Adagrad, i.e drastic decay of learning rate over the iterations.**

AdaDelta Algo

- Adadelta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address following drawback:
  - **The continual decay of learning rates throughout training.**
- Adadelta is a more **robust** extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.
- **This way, Adadelta continues learning even when many updates have been done.**



Alpha in Adadelta



Learning Rate in Adadelta

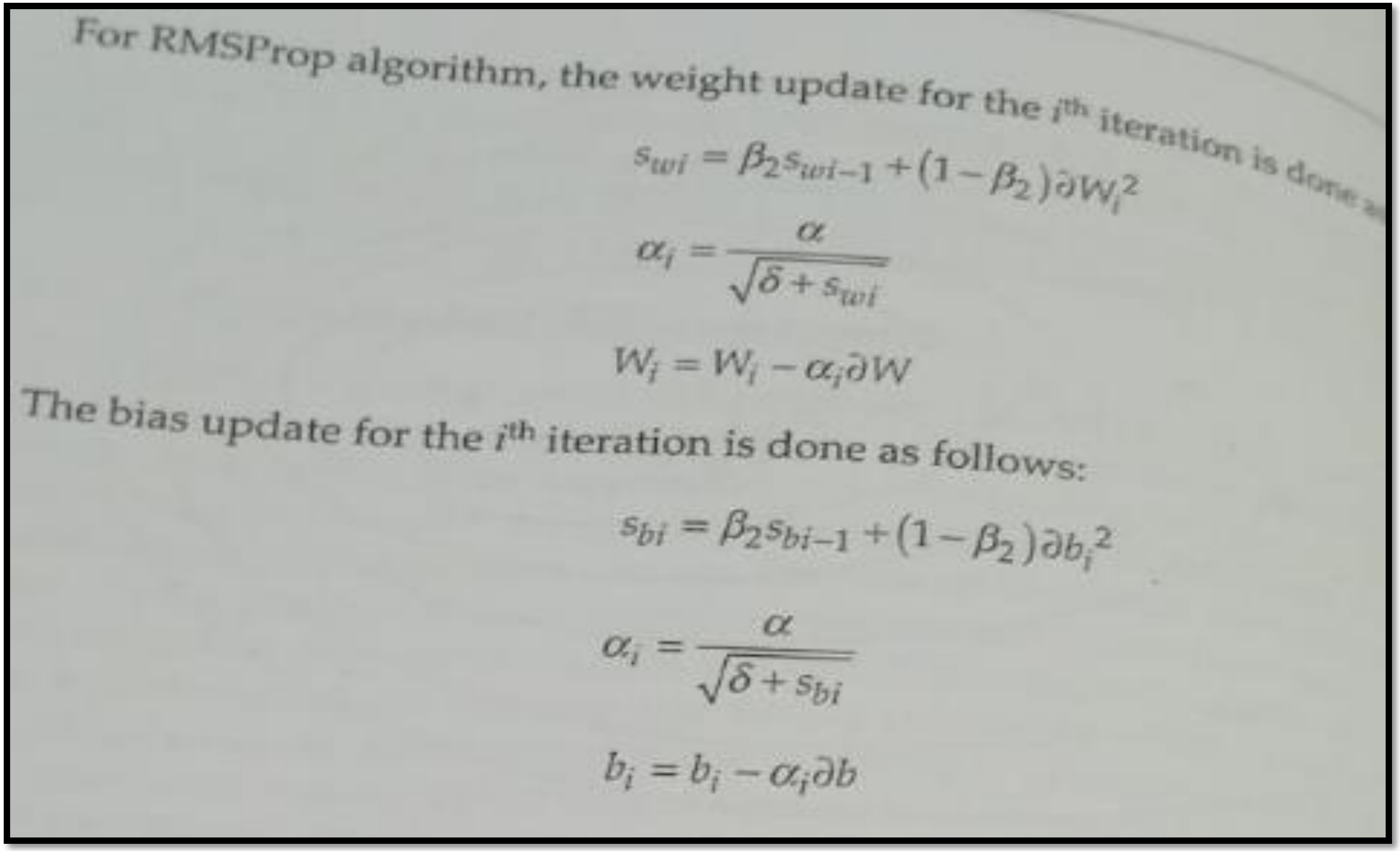
RMSProp Algorithm

- The **RMSProp** algorithm modifies AdaGrad to perform better in the **non-convex** setting by changing the gradient accumulation into an exponentially weighted moving average.
- **Adagrad Disadvantages**
  - **Designed to converge rapidly when applied to a convex function.**
  - When applied to a non-convex function to train a neural network, the learning trajectory may pass through many different structures and eventually arrive at a region that is a **locally** convex bowl.
  - AdaGrad **shrinks the learning rate according to the entire history of the squared gradient** and may have made the learning rate too small before arriving at such a convex structure.

RMSProp Algorithm

- This algorithm **doesnot take a flat cumulative value of past gradients.**
- It uses a decay factor β_2 which helps to give more weightage to recent gradients and less weights to older gradients. β_2 value is often 0.999.

RMSProp Algorithm



For RMSProp algorithm, the weight update for the i^{th} iteration is done a

$$s_{wi} = \beta_2 s_{wi-1} + (1-\beta_2)\partial W_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + s_{wi}}}$$

$$W_i = W_i - \alpha_i \partial W$$

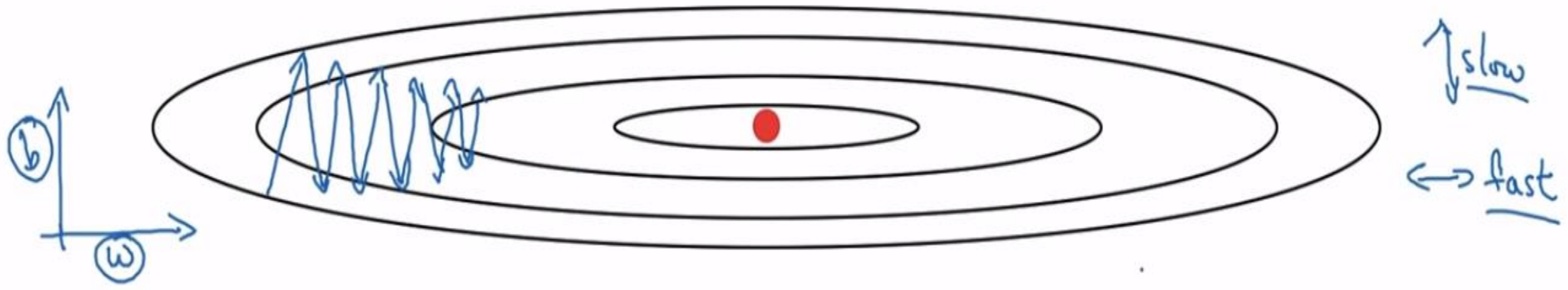
The bias update for the i^{th} iteration is done as follows:

$$s_{bi} = \beta_2 s_{bi-1} + (1-\beta_2)\partial b_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + s_{bi}}}$$

$$b_i = b_i - \alpha_i \partial b$$

RMSprop



On iteration t:

Compute dW, db on current mini-batch

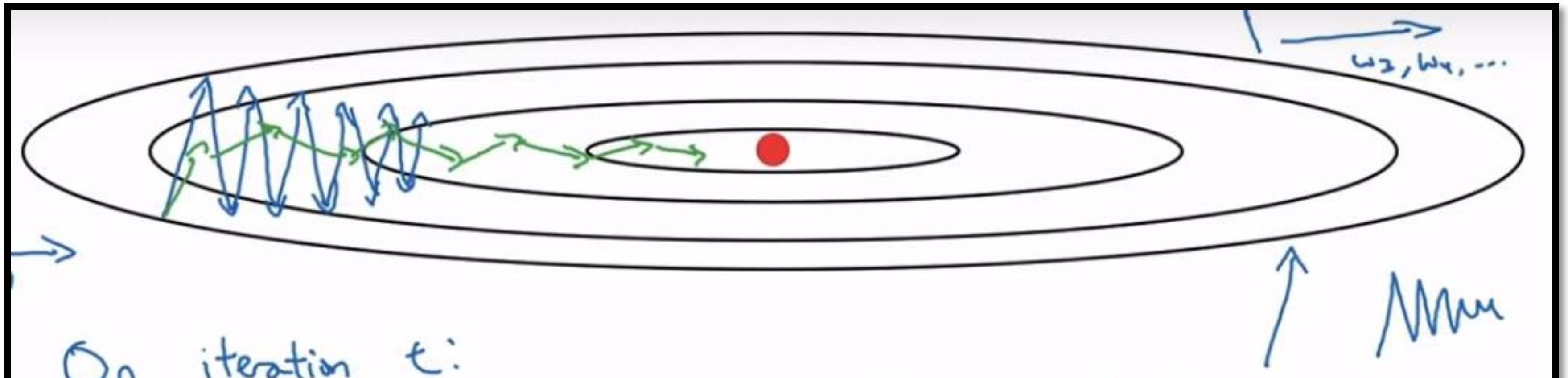
$$S_{dW} = \beta S_{dW} + (1-\beta)\underline{dW^2} \longleftarrow \text{small}$$

element-wise

$$S_{db} = \beta S_{db} + (1-\beta) db^2 \longleftarrow \text{large}$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}} \leftarrow \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}}} \leftarrow$$

RMSProp Algorithm



On iteration t:

Compute dW, db on current mini-batch

element-wise

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, \underline{dW^2} \leftarrow$ small

$\rightarrow \underline{S_{db}} = \beta_2 S_{db} + (1-\beta_2)\, \underline{db^2} \leftarrow$ large

$W := W - \underline{\alpha} \frac{dW}{\sqrt{S_{dW}+\varepsilon}} \quad b := b - \underline{\alpha} \frac{db}{\sqrt{S_{db}+\varepsilon}}$

$\varepsilon = 10^{-8}$

RMSProp Algorithm

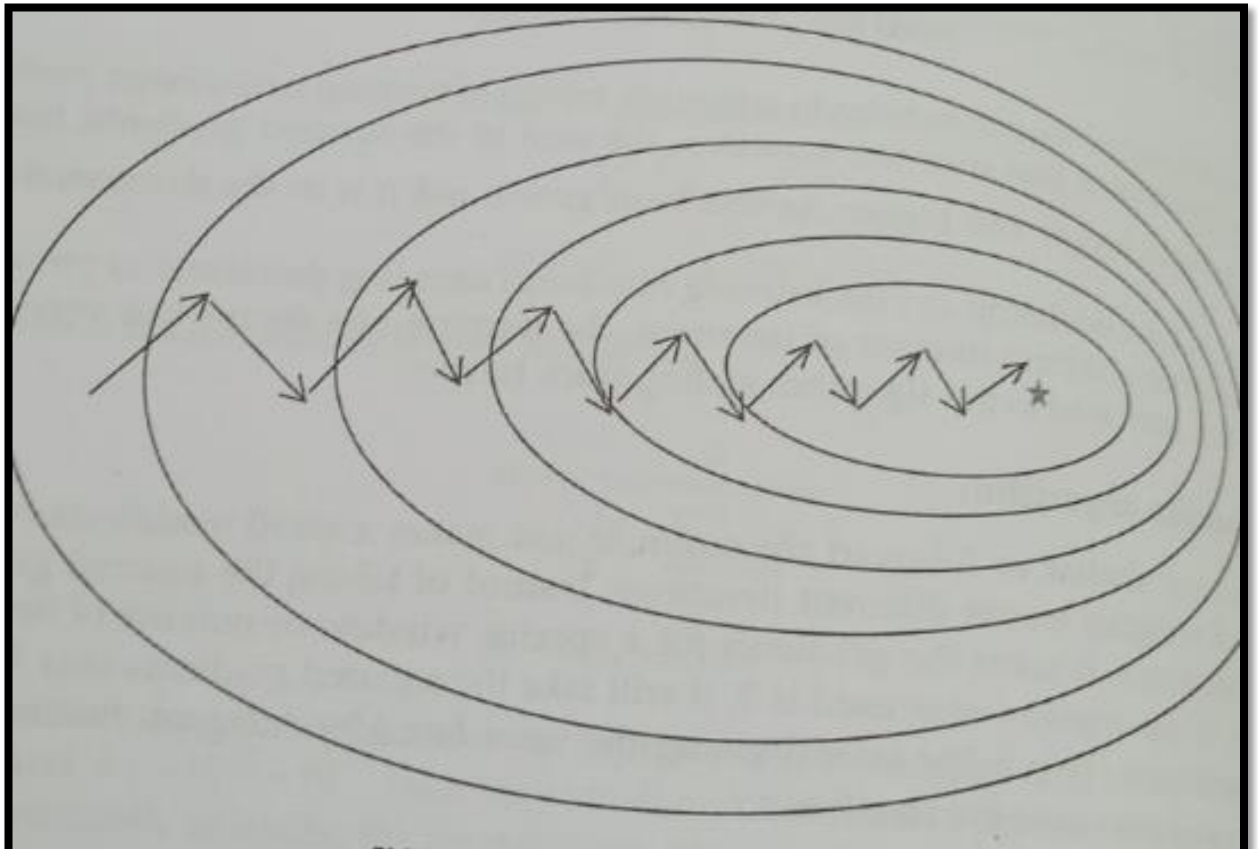


FIGURE 5.21 RMSProp Optimizer

RMSProp Algorithm

- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.
- **RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.**

Adam

- Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent.
- **The method is really efficient when working with large problem involving a lot of data or parameters.**
- **It requires less memory and is efficient.**
- Intuitively, it is a combination of the ‘gradient descent with momentum’ algorithm and the ‘RMSP’ algorithm.
- It also exponentially smooths the first-order gradient in order to incorporate momentum into the update.
- It also directly addresses the bias inherent in exponential smoothing when the running estimate of a smoothed value is unrealistically initialized to 0.

Adam optimization algorithm

$V_{dw} = 0,\ S_{dw} = 0.\ \ V_{db} = 0,\ S_{db} = 0$

On iteration t:

Compute dW, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dW,\ \ V_{db} = \beta_1 V_{db} + (1-\beta_1) db \leftarrow$ "moment" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2,\ \ S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \leftarrow$ "RMSprop" β_2

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t),\ \ V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t),\ \ S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Andrew N

Hyperparameters choice:

$\rightarrow \alpha$: needs to be tune

$\rightarrow \beta_1$: 0.9 $\longrightarrow (\underline{dw})$

$\rightarrow \beta_2$: 0.999 $\longrightarrow (\underline{dw^2})$

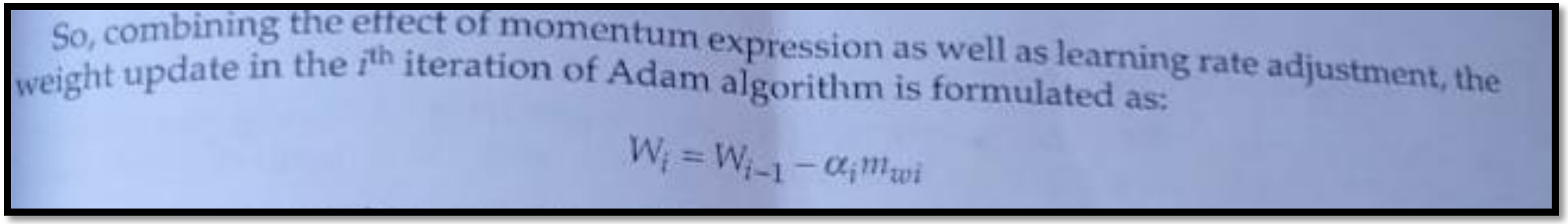
$\rightarrow \varepsilon$: 10^{-8}

Adam: Adaptive moment estimation

Adam Coates

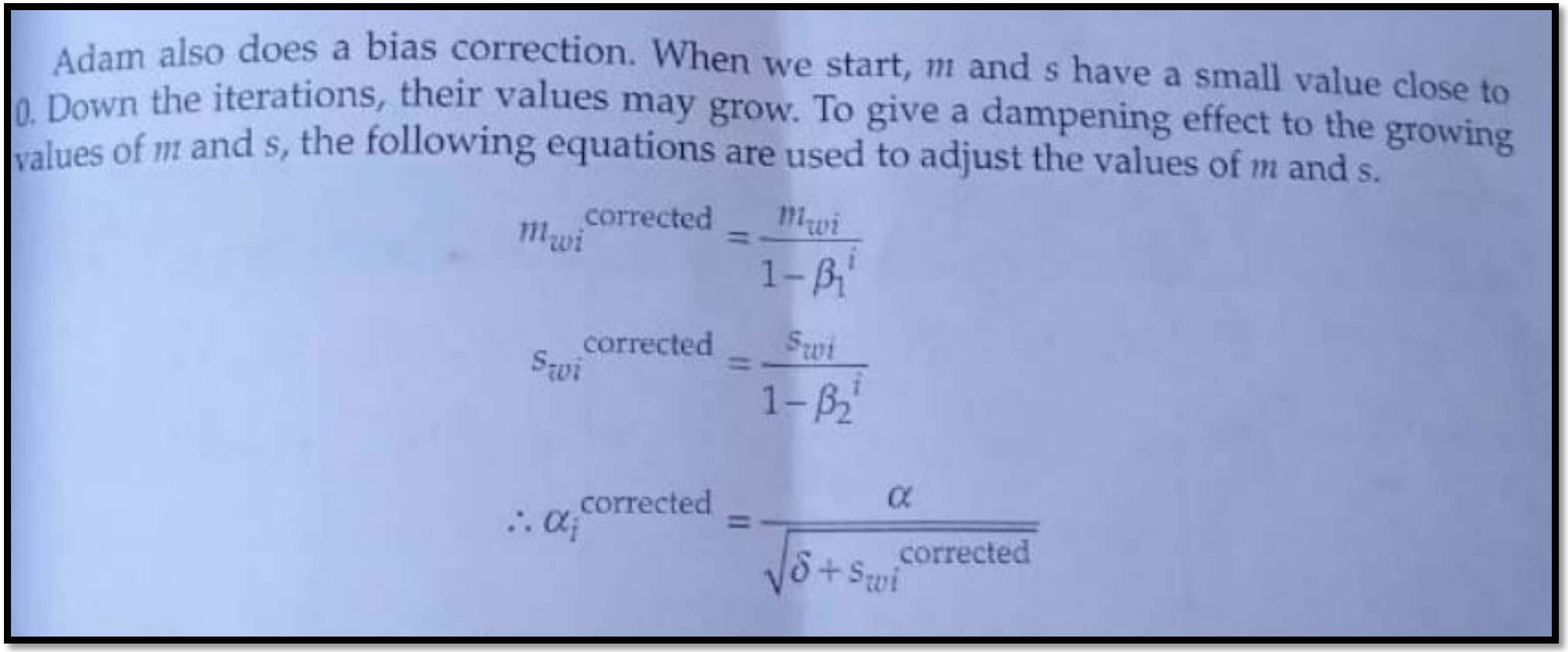
Andrew Ng

Adam



So, combining the effect of momentum expression as well as learning rate adjustment, the weight update in the i^{th} iteration of Adam algorithm is formulated as:

$$W_i = W_{i-1} - \alpha_i m_{wi}$$



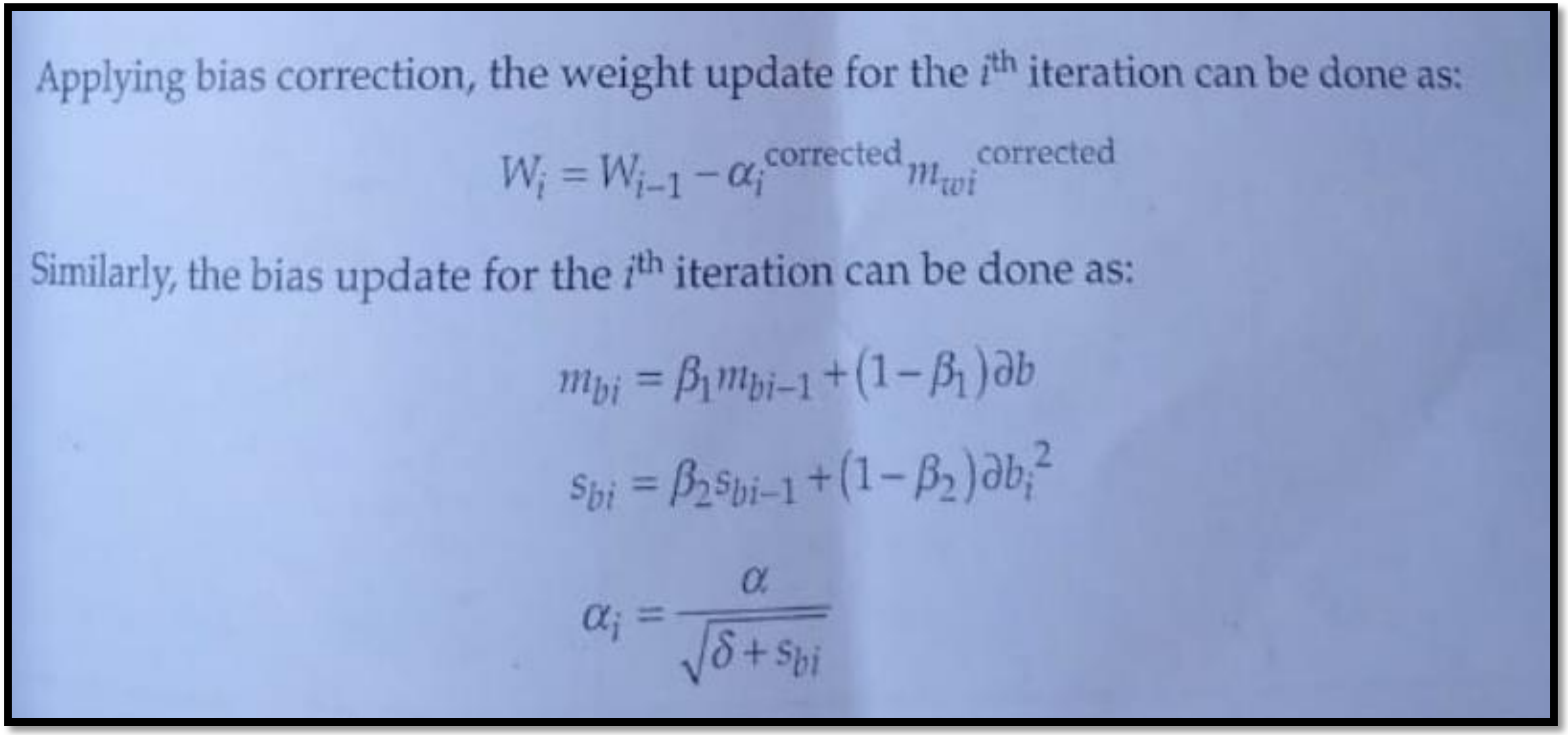
Adam also does a bias correction. When we start, m and s have a small value close to 0. Down the iterations, their values may grow. To give a dampening effect to the growing values of m and s, the following equations are used to adjust the values of m and s.

$$m_{wi}^{\text{corrected}} = \frac{m_{wi}}{1-\beta_1^{\,i}}$$

$$s_{wi}^{\text{corrected}} = \frac{s_{wi}}{1-\beta_2^{\,i}}$$

$$\therefore \alpha_i^{\text{corrected}} = \frac{\alpha}{\sqrt{\delta + s_{wi}^{\text{corrected}}}}$$

Adam



Applying bias correction, the weight update for the i^{th} iteration can be done as:

$$W_i = W_{i-1} - \alpha_i^{\text{corrected}} m_{wi}^{\text{corrected}}$$

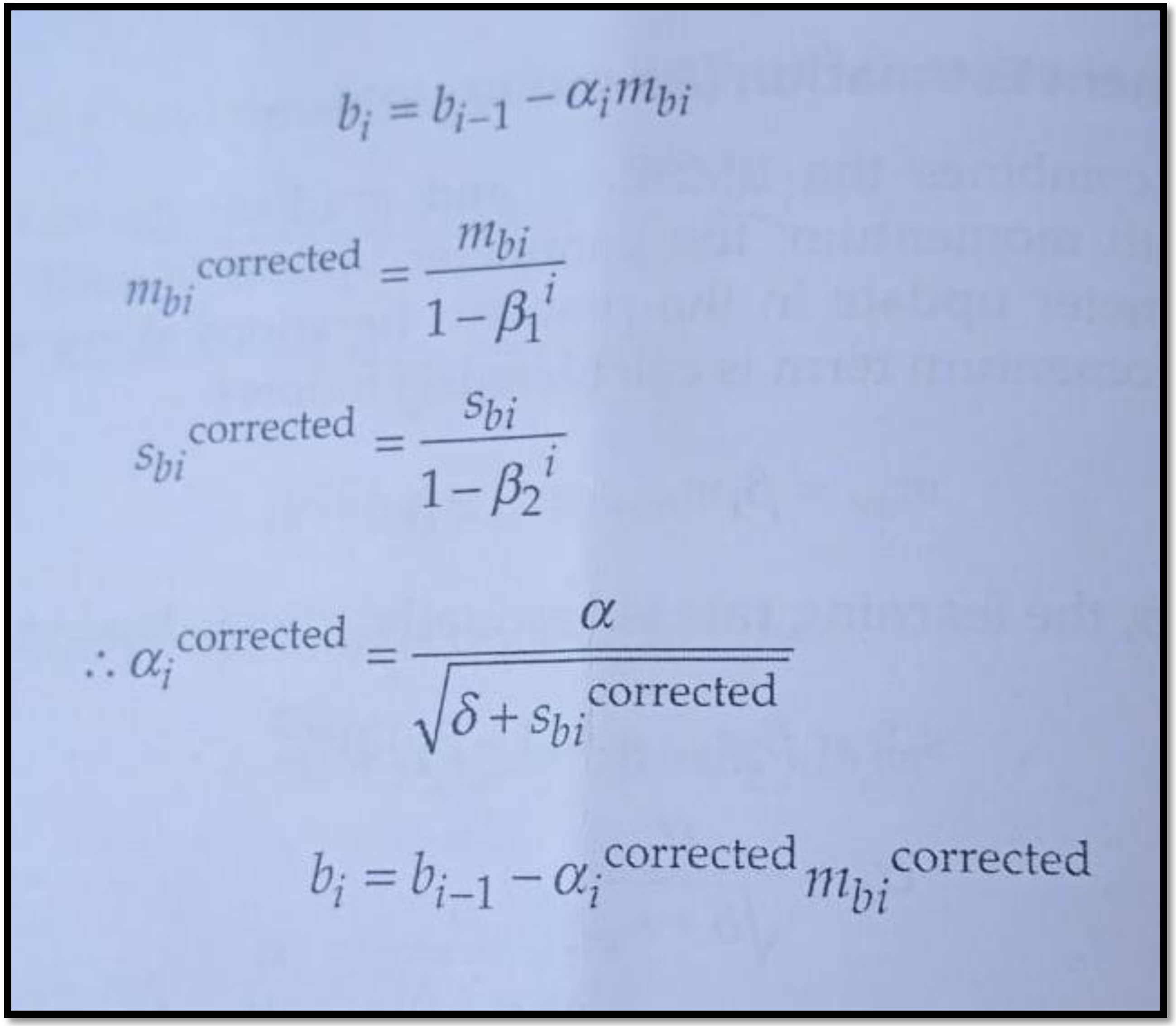
Similarly, the bias update for the i^{th} iteration can be done as:

$$m_{bi} = \beta_1 m_{bi-1} + (1-\beta_1)\partial b$$

$$s_{bi} = \beta_2 s_{bi-1} + (1-\beta_2)\partial b_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + s_{bi}}}$$

Adam



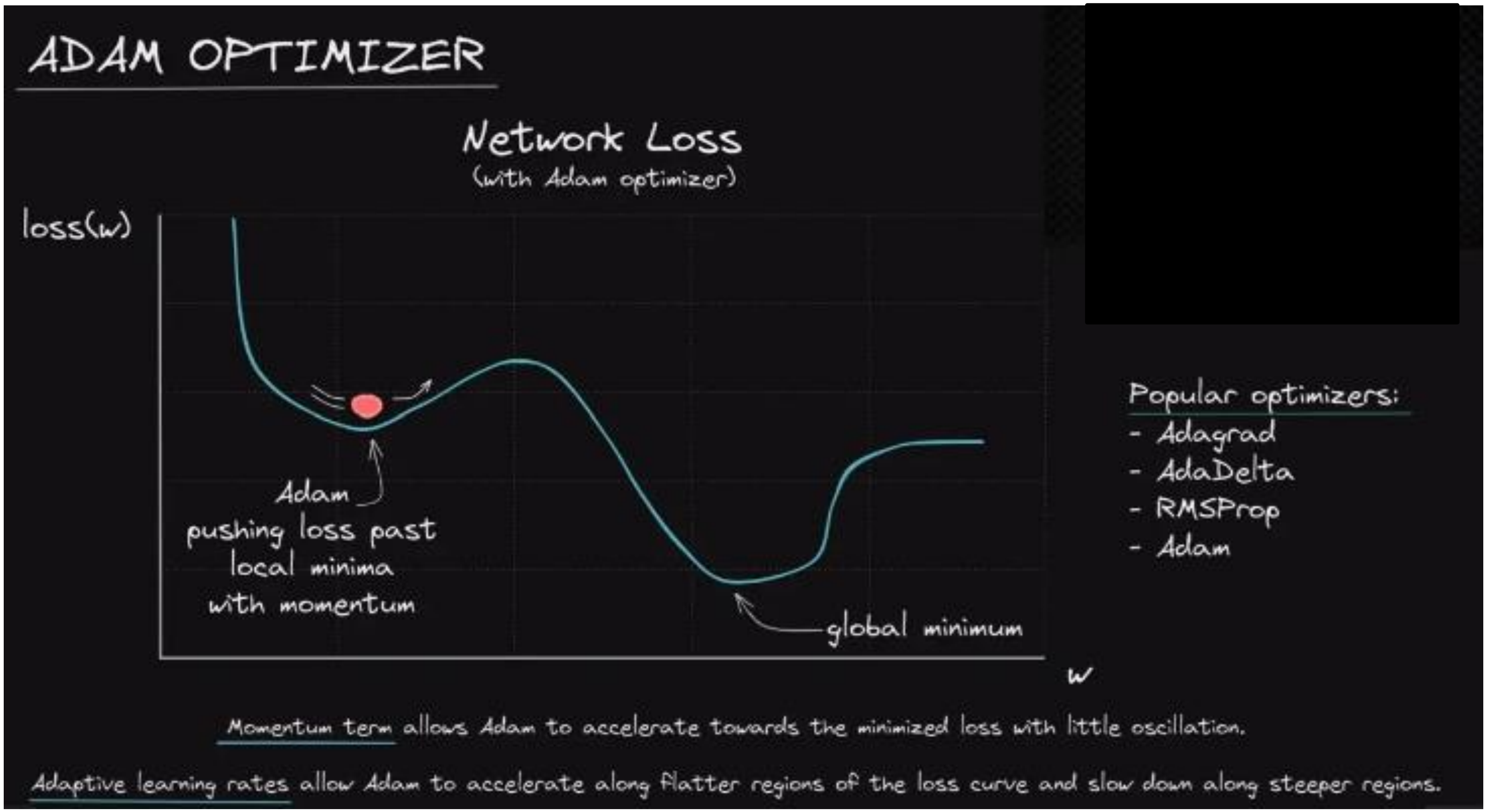
$$b_i = b_{i-1} - \alpha_i m_{bi}$$

$$m_{bi}^{\text{corrected}} = \frac{m_{bi}}{1-\beta_1{}^i}$$

$$s_{bi}^{\text{corrected}} = \frac{s_{bi}}{1-\beta_2{}^i}$$

$$\therefore \alpha_i^{\text{corrected}} = \frac{\alpha}{\sqrt{\delta + s_{bi}^{\text{corrected}}}}$$

$$b_i = b_{i-1} - \alpha_i^{\text{corrected}} m_{bi}^{\text{corrected}}$$



ADAM OPTIMIZER

[redacted]

Popular optimizers:
- Adagrad
- AdaDelta
- RMSProp
- Adam

Momentum term allows Adam to accelerate towards the minimized loss with little oscillation.

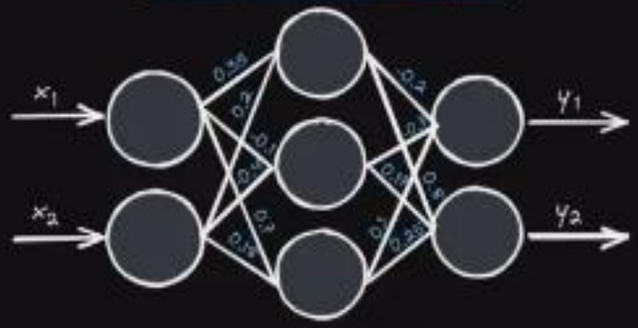
Adaptive learning rates allow Adam to accelerate along flatter regions of the loss curve and slow down along steeper regions.

Optimizer choices….

- For straight forward problems, normal gradient descent (mini-batch incase if the volume of data is huge) can be used.
- If it doesn't yields satisfactory result, RMSProp can be used or to speed up the learning process, gradient descent with momentum can be used.
- For large and sparse data like text data, Adagrad or Adadelta can be used.
- Adam is most popularly adopted optimizer by DL community.

OPTIMIZATION ALGORITHMS

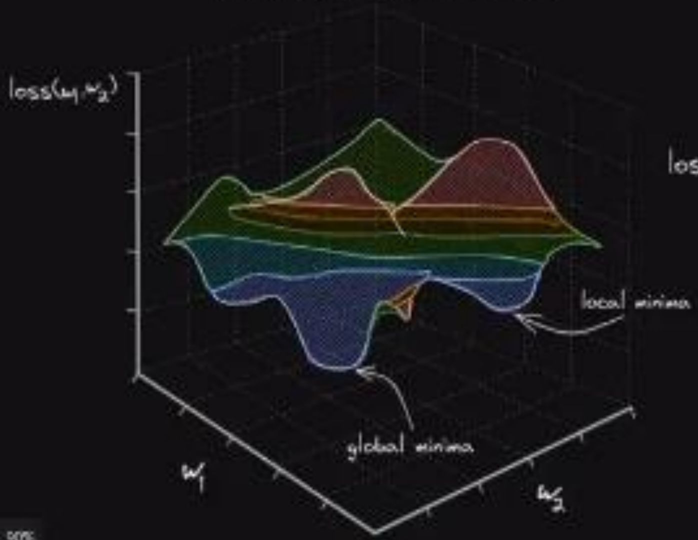
Randomly Initialized Weights



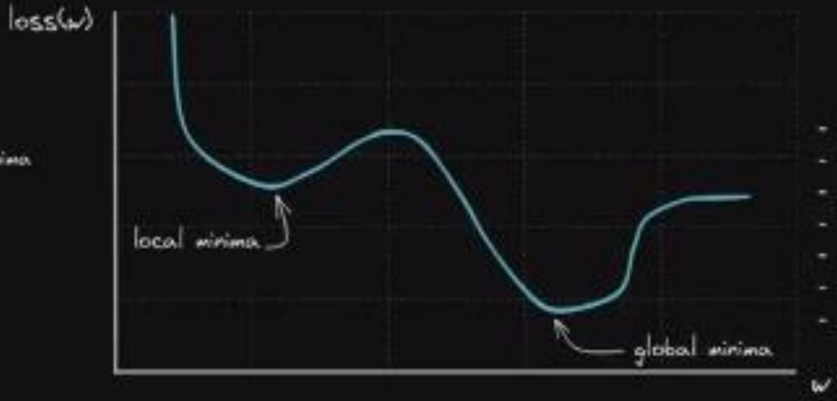
Training Ste

1. Forward pass input data through the network.
2. Obtain output from the network.
3. Measure loss between network output and true labels of data.
4. Complete backwards pass to obtain the gradients (backpropagation).
5. Update network weights with the gradients to step towards minimized loss.

Network Loss
(as a function of weights w_1 and w_2)



Network Loss
(as a function of weight w)



Optimizers

- Batch gradient descent
- Stochastic gradient descent (SGD)
- Mini-batch gradient descent
- Adagrad
- AdaDelta
- RMSProp
- Adam

Thank You!