Adagrad optimizers:

- GD uses the same step size (learning rate) for each parameter or I/p feature
- This leads to have different amounts of curvature in different dimensions.
- So, algorithm needs a different step sized to each parameter.

Need of Adagrad

→ Adagrad is an extension of the GD optimization.
→ It allows different learning rate (η) for each weight
→ In real-world dataset, some I/p features are sparse (most of the features are zero) and some I/p features are dense (most features are non-zero).
→ In this scenario, same learning rate(η) for all the weights is not good for optimization.
→ So, Adagrad algorithm adapts the learning rate to the parameters.
* It is using low learning rates for parameters associated with frequently occuring features
* High learning rates for parameters associated with rare features.
→ This is well suitable for the sparse data.

e.g.

During training some parameters might reached near to converging stage where only fine adjustment is required, but some parameters need to be adjusted significantly due to small no. of matching samples ie, far away from convergence.

→ The weight update rule for adagrad is

$$\omega_t = \omega_{t-1} - \eta_t \frac{\partial E}{\partial \omega_{t-1}}$$

$$\eta_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

where α_t - different learning rates for each weight at each iteration.

ϵ - small the constant to avoid division by 0 error

where $\alpha_t \to 0$, $\epsilon = 10^{-8}$

* If α_t becomes 0 then η_t will become 0 which lead to small convergence ie, no change in ω.

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial E}{\partial \omega_{t-1}} \right)^2$$

$\frac{\partial E}{\partial \omega_{t-1}}$ is derivative of loss w.r.t. weight and $\left(\frac{\partial E}{\partial \omega_{t-1}} \right)^2$ will always be +ve since it is a square term.

so, α_t is always +ve due to squaring the gradients, hence $\alpha_t >= \alpha_{t-1}$

→ From this, α_t and η_t is inversely proportional to one another.

$$\eta_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

→ This implies that when α_t will increase, η_t will decrease.

*so, when the no. of iterations will increase, the learning rate will reduce adaptively. Hence, no need to select the learning rate manually.

Advantages

→ No manual tuning of the learning rate is required.

→ Faster convergence and more reliable.

Disadvantages

~~When α_t increases, η_t will decrease~~ when α_t become large, increases the no. of iterations. Due to this, η_t will decrease at the larger rate. This will make the old weight almost equal to the new weight which may lead to slow convergence.

② RMSProp optimizer:

→ It is an adaptive learning rate optimization algorithm

→ It is a variant of SGD and Adagrad algorithms.

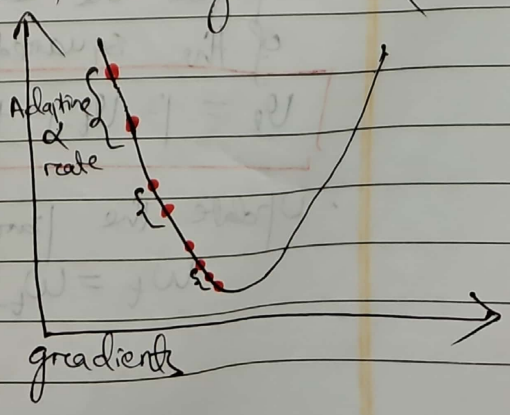
→ It improves the convergence speed and stability of the model training process.

→ It uses moving average of the squared gradients i.e; $(\nabla_w E(W_t))^2$ to scale (exponentially decay) the learning rate for each parameter.

→ It makes the learning rate is less than a certain threshold.

→ RMSprop is designed to dramatically reduce the computations in training neural networks.

→ This helps reduce the learning rate by forgetting each gradients, and focus on the most recently observed partial gradients.



during the progress of the search, overcoming the limitation of Adagrad.

→ It leads to smoothly adjust the learning rate for each parameter in the network providing a better performance than batch GD

→ RMSProp algorithm utilizes exponentially weighted moving averages of squared gradients to update the parameters ie; $(\nabla_w E(w_t))^2$

RMSProp Algorithm

→ Initialize parameters:

- Learning rate: α, decay rate for averaging β.
- Small constant for numerical stability ϵ, initial parameter values w.

→ Initialize accumulated gradients (Exponentially weighted average)

- Accumulated squared gradient for each parameter: $v_t = 0$

→ Repeat until convergence or maximum iterations:

- Compute the gradient of the objective function w.r.t. the parameters:
  $g_t = \nabla_w E(w_t)$
- Update the exponentially weighted average of the squared gradients:

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

- Update the parameters:

$$w_t = w_{t-1} - \alpha \cdot \frac{g_t}{\sqrt{v_t + \epsilon}}$$

Where g_t → gradient of the loss function w.r.t. the parameters at time t.

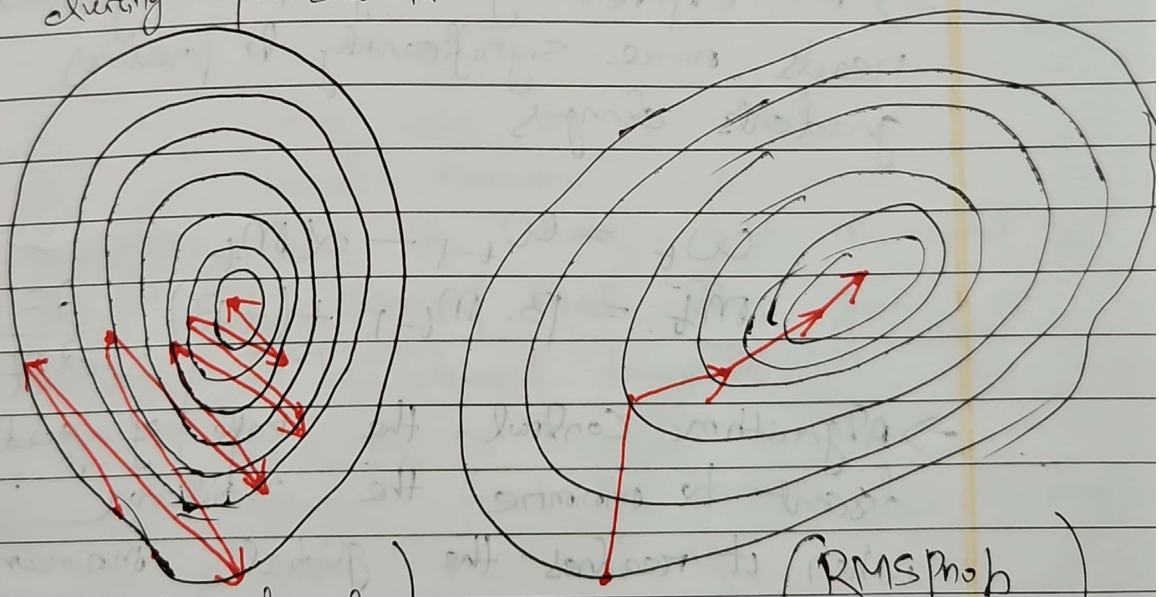
β → decay factor

v_t → exponentially weighted average of the squared gradients.

ϵ → small constant to prevent division by 0

→ This process is repeated for each parameter, and it helps adjust the learning rate for each parameter based on the historical gradients (t-1).

→ The exponential moving average allows the algorithm to give more importance to recent gradients, and dampen the effect of older gradients, providing stability during optimization.



(Gradient descent)

(RMSprop)

③ Adam (Adaptive Momentum estimation) optimizer

→ Adam is an extension to SGD algorithm
→ It is a 1st order derivative optimization algorithm.
→ It uses the SGD extensions of Adagrad and RMSprop, to deal with ML and DL problems involving large datasets and high dimensional parameter spaces.

Momentum :

→ Momentum is used to accelerate the GD algorithm by considering the exponentially weighted average of the gradients.
→ These averages makes the convergence towards the minima faster
→ An exponentially weighted moving average reacts more significantly to previous gradients changes.

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

$$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot \frac{\partial E}{\partial w_t}$$

→ Algorithm control the rate of gradient descent to minimize the oscillation when it reaches the global minimum while taking big enough steps (step size)

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w_t}\right)^2$$

→ It makes the algorithm to pass the local minima hurdles in a search space.
→ Hence, combine the features of the above methods to reach the global minimum e.

Mathematical Aspect of Adam optimizer

→ First momentum update

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\left(\frac{\partial E}{\partial w_t}\right)$$

→ second momentum update

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w_t}\right)^2$$

where,

β_1 & β_2 : exponential decay rates average of gradients ($\beta_1 = 0.9$, $\beta_2 = 0.999$)

These are estimating the moments.

→ Exponential Decay - Learning rate schedule where decay (decrease) the α with more iterations using the exponential function :

$$\alpha = \alpha \cdot e^{-kt}$$

k : decay rate

t : iteration number

since m_t and v_t have both initialized as 0 ; it is observed that they gain a tendency to be biased towards 0 as both β_1 & $\beta_2 \approx 1$

→ This optimizer fixes this problem by computing bias-corrected m_t and v_t . This controls the weights while reaching the global minimum to prevent high oscillations when near it.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

→ Initially, learning is adapting to the GD after every iteration so that it remains controlled and unbiased throughout the training.

→ Now, instead of our normal weight parameters m_t and v_t, we take the bias-corrected weight parameters $\hat{m}_t$ and $\hat{v}_t$.

→ Use these into general equation as,

$$w_t = w_{t-1} - \hat{m}_t \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}}$$

ϵ → small the constant to avoid division by 0 error when $v_t \to 0$, $\epsilon = 10^{-8}$.

Algorithm :- Adam

β_1 and β_2 = decay rates of average of momentum and RMSprop methods ($\beta_1 = 0.9$, $\beta_2 = 0.999$)

w_0 : Initial parameter vector

$f(w)$: stochastic objective function.

- Initialize $m_0 = 0$ // First moment vector.
- $v_0 = 0$ // 2nd " "
- $t = 0$ // timestep

Repeat {

$t = t + 1$

$g_t = \nabla(w) \cdot f_t(w_{t-1})$

$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ // update biased 1st moment estimate

$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ // update biased 2nd raw moment estimate

$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$ // bias-corrected 1st moment estimate

$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$ // bias-corrected 2nd moment estimate

$w_t = w_{t-1} - \hat{m}_t \left(\frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\right)$ // update the parameter

→ until w_t reach convergence

①	ML	DL
Data	Requires less amount of data.	Requires large amount of data.
H/w requirements	Work on low-end machines	Heavily depend on high-end machines.
Training time	Takes less time for train	Takes more time for training.
Data Interpretation	Fails to interpret the result.	Easy to interpret the result.
Feature engineering	Identify by expert. Hand-coded by expert as per domain and data type.	Learns automatically high-level features from data.

③ Ideal value for η → 0.001

④ Applications of DL:
- → ~~Image recognition~~
- → Natural language processing
- → Medical diagnosis
- → self driving cars
- → Information retrieval

③
- → Can handle structured & unstructured
- → High degree of accuracy
- → Automatic feature extraction
- → DL models can generalize well to new unseen data.