<u>**Fundamentals of Deep Learning**</u> :

Gradient Descent Optimizers

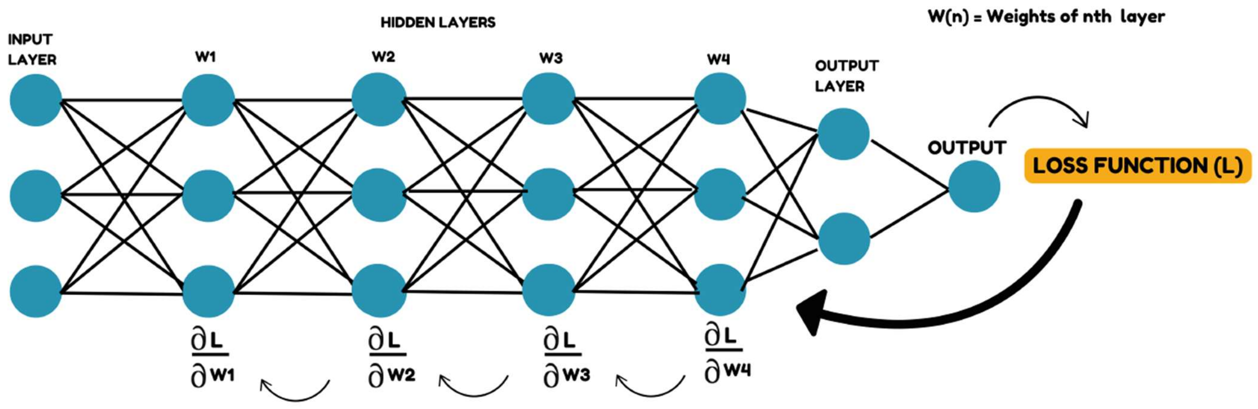
Gradient descent is the method of minimizing the loss function by iteratively updating the parameters of a neural network based on the gradients of the loss function with respect to those parameters.

⟶ The **loss function** of a neural networks calculates its error, given its output value (F(x)) and true value (y).

⟶ The **cost function** applies the loss function to calculate the error of the entire neural network for the entire dataset.

⟶ To train the neural network, we must update its weights and parameters and in order to update the weights, we take the derivative of the cost function with respect to parameters.

Starting at the penultimate layer, we take derivatives and update the weights, and then work our way backwards to the first hidden layer.


HIDDEN LAYERS
W(n) = Weights of nth layer
INPUT LAYER
W1
W2
W3
W4
OUTPUT LAYER
OUTPUT
LOSS FUNCTION (L)
$\frac{\partial L}{\partial W1}$
$\frac{\partial L}{\partial W2}$
$\frac{\partial L}{\partial W3}$
$\frac{\partial L}{\partial W4}$

The process and outcome of gradient descent relies on :

a) Learning Rate (η)

b) Updating of weights/ parameters (W)

c) The loss function J(w,b) (at specific iterations)

There are various methods that can optimize gradient descent by adjusting these two factors in different ways :

Method	Working
1) AdaGrad (Adaptive Gradient algorithm)	The factor G (Gi(0)=0), is updated as : $$G_i(t) = G_i(t-1) + (\frac{\partial L}{\partial W_i}(t))^2$$
⟶ Adaptive optimization algorithm ⟶ Scales the update for each weight separately ⟶ Keeps running sum of previous updates ⟶ Divides new updates by a factor of previous sum ⟶ The function L and ∇J are the loss function and its derivatives respectively. ⟶ The factor G represents the sum of the squares of the gradients of the loss function w.r.t the parameters over all iterations up to the current iteration. ⟶ The learning rate (η), divided by G factor at each step, also decreases, helping avoid overshooting. ⟶ Adjusts the weights as well as the learning rate togther, making convergence faster and reliable. + Reduces the need for tuning the learning rate	This is used to update the weights as well as the learning rate : $$W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t + \epsilon}} . \bigtriangledown J(W_t)$$ The learning rate is decreasing at each step here as : $$\frac{\eta}{\sqrt{G_t + \epsilon}}$$

2) RMSProp (Root Mean Square Propagation)

⟶ Adaptive optimization algorithm

⟶ When updating the parameters, the more recent gradient updates have more impact than older gradient updates, which are decayed.

⟶ Decay factor (β) is used to decay older updates, and update the parameters and learning rate based more on recent updates.

⟶ The function L and ∇J are the loss function and its derivatives respectively.

⟶ g(t,i denotes the squared gradient of the loss function with respect to the i-th parameter at a particular iteration t.

⟶ The factor G represents the average of the squared gradients for parameter W(i) at iteration t, including the decay factor.

⟶ The parameters, as well as the learning rate is updated with the G factor.

+ Stabilizes the learning process by adapting the learning rate based on recent gradients

+ Suitable for problems with sparse data or high-dimensional spaces

+ Reduces the need for tuning learning rate

The factor G is calculated by including the decay rate :

$$G_i^t = \beta . G_{t-1}^i + (1-\beta).(g_i^t)^2$$

where the older gradients are being decayed by multiplying with the decay factor :

$$\beta . G_{t-1}^i$$

Using the G factor that emphasizes the effect of decaying older gradients, we update our parameters and the learning rate as well :

$$W_{t+1}^i = W_i^t - \frac{\eta}{\sqrt{G_t^i + \epsilon}} . g_i^t$$

here, the learning rate is also being adjusted, also factoring recent learning rate updates more while decaying older updates as :

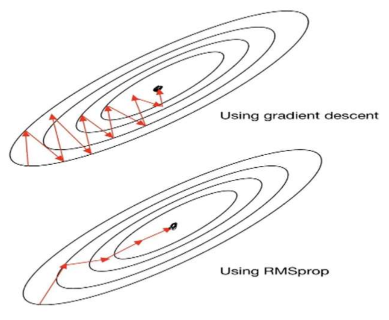
$$\frac{\eta}{\sqrt{G_t^i + \epsilon}}$$

AdaGrad

$$g_0 = 0$$
$$g_{t+1} \leftarrow g_t + \nabla_\theta \mathcal{L}(\theta)^2$$
$$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$$

RMS Prop

$$g_0 = 0, \alpha \simeq 0.9$$
$$g_{t+1} \leftarrow \alpha \cdot g_t + (1-\alpha) \nabla_\theta \mathcal{L}(\theta)^2$$
$$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$$



3) Momentum

In standard gradient descent, we update the weights by :

$$W_{new} = W_{old} - \alpha . \nabla J$$

where α is the learning rate and del J is the derivative of the loss function.

⟶ Simple acceleration of gradient descent

⟶ Smoothen the gradient descent process by taking a running average of each of the steps

⟶ The momentum (η) hyperparameter value is used with a factor Vt that smoothens out the variation for each of the individual steps

⟶ For V at step t, i.e. V(t), it considers the value of V(t-1) and the current gradient at step t.

⟶ It is influenced by the values :

1) Momentum parameter η
2) Learning rate (alpha)

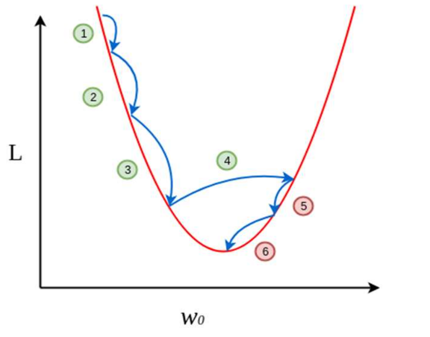
⟶ The Nesterov Momentum improves on the generic concept of momentum by addressing the problems of overshooting, by looking ahead.

The V(t) factor for every iteration is calculated considering the momentum parameter as :

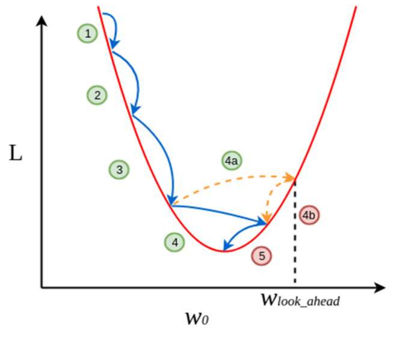
$$V_t = \eta . V_{t-1} - \alpha . \nabla J$$

This value is used to update the weights, in momentum accelerated gradient descent :

$$W_{new} = W_{current} - V_t$$



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

4) Adam (Adaptive Moment Estimation)

⟶ Adaptive optimization algorithm

⟶ As said by Dr. Combines aspects of both momentum optimization and RMSProp.

⟶ The adaptive learning rates allow Adam to adapt to different types of data and architectures, making it suitable for a wide range of deep learning tasks.

⟶ Adam has several hyperparameters to tune, including the learning rate η\etaη, the exponential decay rates β1 and β2, and the small constant ϵ.

⟶ Techniques like weight decay can be used alongside Adam to prevent overfitting.

In the equation for momentum :

$$V_t = \eta . V_{t-1} - \alpha . \bigtriangledown J$$

set η as **β1** and a as (1-**β1)** , we can get a momentum m(t) value :

$$m_t = \beta_1 . m_{t-1} + (1 - \beta_1) . \bigtriangledown J$$

which can further be adjusted to correct bias when values of t are small :

$$\hat{m}_t = \frac{\hat{m}_t}{1 - \beta_t^1}$$

In the equation for RMSProp, which prioritises new gradients more :

$$G_i^t = \beta . G_{t-1}^i + (1 - \beta) . (g_i^t)^2$$

we can interpret as :

$$V_t = \beta_2 . V_{t-1} + (1 - \beta_2) \bigtriangledown J$$

which can adjusted to correct biases for small values of t :

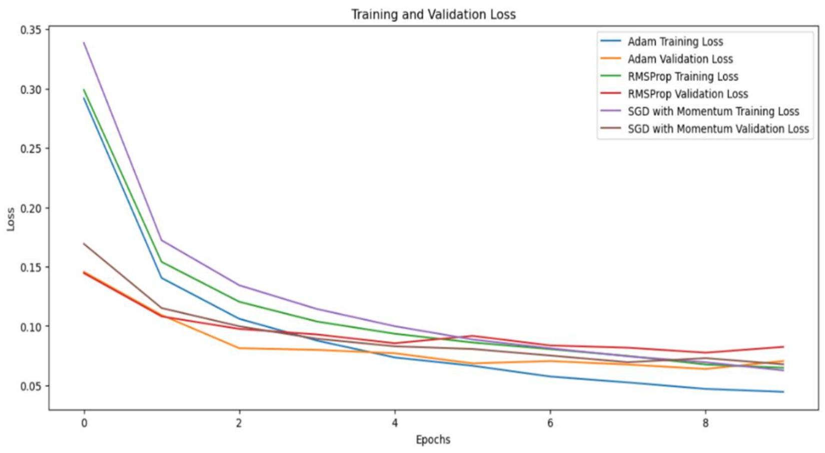
$$\hat{V}_t = \frac{V_t}{1 - \beta_2^t}$$

On combining both, we get :

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{V}_t + \varepsilon}} . \hat{m}_t$$

which describes the weights as well the learning rate being updated.

On training a neural network for classifying MNIST images for ten epochs using different optimizers :



Thank You !

If you found this article helpful, please do leave a like and comment any feedback you feel I could use.

While I'm working on the next project, I'll be over at :

www.linkedin.com/in/shivang-kainthola-2835151b9/

shivang.kainthola64@gmail.com ☕

https://shivangkainthola28.medium.com/

https://github.com/HeadHunter28