Vanishing Gradient Problem.

→ The vanishing gradient problem is a common issue that can arise in deep networks during training.

→ It occurs when the gradients, which are used to update the parameters of the network during the back-propagation process, become very small as they are propagated backward through the network.

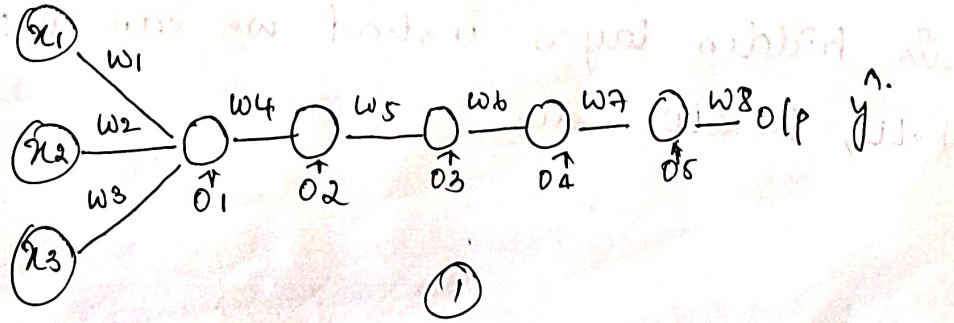
→ This problem is particularly common in deep neural networks that have many layers, as the gradients have to pass through all of these layers to update the parameters of the network.

→ As the gradients are multiplied by the weights at each layer during the backpropagation process, they can become exponentially small as they propagated backwards through the network.

→ This means that the weights at the earlier layers of the network are updated very slowly, and may not change significantly during training.

→ This vanishing gradient is a common problem, if we are using activation functions such as sigmoid and tanh activation functions.

→ Ex: consider a deep neural network as shown



During backpropagation, weight updation formula fi

$$w_{1new} = w_{1old} - \eta \frac{\partial L}{\partial w_{1new}}$$

$$\frac{\partial L}{\partial w_{1new}} = \frac{\partial L}{\partial o_{51}} * \frac{\partial o_{51}}{\partial o_{41}} * \frac{\partial o_{41}}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial w_1} \rightarrow (1)$$

↑ using chain Rule

Suppose we are using sigmoid activation function at each layer, we have formula as,

$$f(x) = \frac{1}{1+e^{-x}} \rightarrow 0 \text{ or } 1 \quad \text{if value } \geq 0.5 = 1, \; < 0.5 \; 0.$$

→ we need to find the derivative of sigmoid function at each output layer.

→ The derivative of sigmoid function $f(x)$ value will be always between 0 to 0.25

→ As we calculate the derivative of sigmoid at each layer using chain rule defined at (1), the gradient value decreases and at some point it will be very small value

→ Because of this weight updation formula fails to update new weights and hence training process looks paused.

→ This problem is called as "vanishing gradient" problem.

Solution

→ we should not use sigmoid activation function in hidden layers instead we can opt for ReLU, PReLU etc.

Fine Tuning Neural network: Number of hidden layers, number of neurons per hidden layer, learning rate, batch size and other hyperaters.

→ Hyperparameter tuning is the process of selecting the optimal set of hyperparameters for a machine learning model that provides the best performance on a given dataset.

→ Hyperparameters are the parameters that cannot be learned from the training data and need to be specified by the user before training the model, such as the learning rate, the number of layers, the number of neurons per layer, the activation function etc.

→ There are different methods for hyperparameter tuning, such as grid search, random search, Bayesian optimization, genetic algorithm etc.

→ Grid search involves evaluating the model performance for all possible combinations of hyperparameters specified in a predefined search space.

→ Random search selects random combinations of hyperpara-meters from the search space and evaluates the model performance.

→ Bayesian optimization and genetic algorithms use advanced optimization techniques to intelligently search the hyperparameter space and find the optimal set of hyperparameters.

a) Hyperparamiter tuning to decide number of hidden layers.

→ Deciding the number of hidden layers is a critical hyperparameter for building a neural network, and it can significantly affect the model's performance.

→ A popular approach is to use grid search or random search to evaluate different numbers of hidden layers and select the one that performs the best.

→ Another approach is to determine the optimal number of hidden layers is to use a progressive network architecture. In this approach, the model is trained with a small number of hidden layers and gradually increases the number of hidden layers while keeping the weights of the previous layers fixed.

5) choosing number of hidden neurons.

→ Optimal number of hidden neurons may depend on the specific problem, dataset and other hyperparameter used in the network.

→ Hence it is important to experiment with different values and approaches to find the best choice for a particular problem.

→ There are several approaches for choosing the number of hidden neurons such as

a) Rule of Thumb: According to this, the number of hidden neurons are choosen between input layer and output layer. But it is not optimal for all the problems.

other methods can be gridsearchcv, Randomsearchcv etc.

c) Learning Rate: It controls, how much the weights are adjusted during training.

→ A learning rate that is too high can cause the model to overshoot the optimal weights, while a learning rate that is too low can cause the model to converge too slowly.

→ some of the approaches can be: manual tuning, gridsearch, cyclic learning, Adaptive learning etc.

d) Batch size: It determines the number of samples that are processed by the model in each training iteration. Batch-size can be tuned used methods such as manual tuning, Gridsearch, gradual increasing of batch size etc.

* Exploding Gradient Problem.

→ It is a common issue that occur during the training of deep neural networks.

→ It arises when the gradients used to update the model's weights become very large, causing the model to become unstable and fail to converge to a good solution.

→ During training, the model calculates the gradient's of the loss function with respect to the models weight. using backpropagation algorithm.

→ The gradient's are then used to update the weights in the direction that reduces the loss function.

→ If the gradients become too large, the weight updates can become too extreme cause the model's optimal

solution to oscillate between different values. This can cause the loss function to increase or become stuck at a high value, preventing the model from improving.

→ The exploding gradient problem is often caused by deep neural networks with many layers and with activation functions that have large derivatives (Eg: sigmoid) and with weight-initialization methods that create very large initial weights.

→ There are several approaches to handle exploding gradient problem.

1. Gradient clipping: One approach is to clip the gradient to a maximum value, preventing them to become too large. This can be done by setting a maximum threshold on the size of the gradients during training.

2. Weight Initialization: Proper initialization of the weights is crucial in preventing the exploding gradient problem. One approach is to use weight initialization methods such as Xavier or He initialization, which set the initial weights to values that ensure a more stable and effective training process.

3. Using different activation functions: Activation functions like sigmoid or hyperbolic tangent can lead to the exploding gradient problem due to their saturation nature. ReLU and its variants, which have a smaller output range are less prone to this issue and can help stabilize the gradient.

4. Batch Normalization: Batch normalizing is a technique that involves normalizing the inputs to each layer of the neural network. This ensures more stable training process.

5. Learning Rate scheduling: Adjusting the learning rate during the training process can help to stabilize the gradients and prevent them becoming too large.

* Weight Initialisation Techniques.

→ weight initialization is an essential step in the training of neural networks. some commonly used weight initialization techniques are as follows.

1. Random initialization: Here, weights are initialized to random values drawn from a distribution such as normal or uniform distribution

2. Xavier initialization: It is also known as glorot initialization, sets the initial weights based on the number of inputs and outputs to a layer. The motivation behind Xavier initialization is to ensure that the variance of the activations remains constant as the data propagates through the network.

→ Xavier initialization technique is particularly suitable for activation functions such as tanh, sigmoid etc.

→ In xavier normal initialization, the weights are initialized using a normal distribution with a mean of 0 and standard deviation of $\sqrt{\frac{2}{n\_ip + n\_out}}$

→ In xavier uniform distribution, the weights are initialized using a uniform distribution with a range [-sqrt(6/(n_in + n_out)), sqrt[6/(n_in + n_out)]]

③ He initialization: It is similar to Xavier initialization but it is used for activation functions that have a Rectified linear unit (ReLU) or its variants.

→ He normal initialization, the weights are initialized using a normal distribution with a mean of 0 and a standard deviation of sqrt(2/n), where n is the number of input neurons.

→ In He Uniform initialization, the weights are initialized using a uniform distribution with a range of [-sqrt(6/n), sqrt(6/n)], where n is the number of input neurons.

④ LeCun Initialization: It is similar to xavier initialization but is used for networks with convolutional layers. In this weights are assigned based on the size of the filter, the number of input channels, and the number of output channels.

⑤ orthogonal Initialization: In Orthogonal initialization, the weights are initialized to an orthogonal matrix. This ensures that the weights do not affect the magnitude of the input, hence can help in Exploding or vanishing gradient problem.

Batch Normalization.

→ It is a technique used in deep learning to normalize the input to each layer of a neural network by adjusting and scaling activation.

→ It has the effect of stabilizing the learning process and reduces the number of training epochs required to train deep networks.

→ Batch normalisation is also a regularization technique, but it doesnot work like l1, l2, drop out regularization.

→ Adding batch normalisation we can reduce the internal covariate shift and instability in distributions of layers activations.

→ In batch normalization, the normalization is applied to each layer for every mini-batch.

→ The idea behind batch normalization is to normalize the input to each layer by subtracting the mean and deviding by the standard deviation of the batch of data being processed.

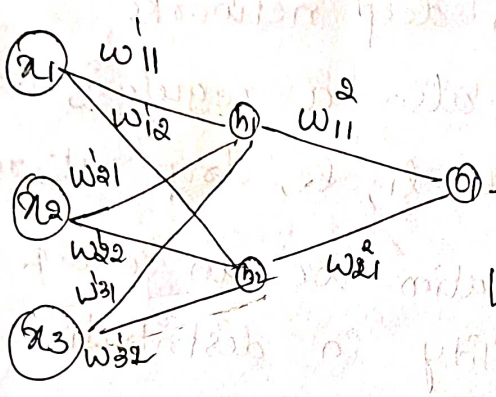
→ Batch normalization can be applied to any layer of a neural network, including fully connected layers, convolutional layers, and recurrent layers, It can be inserted between the linear and non-linear activation functions of a layer

$$y = \text{gamma} * (x - \text{mean}) / \text{sqrt}(\text{variance} + \text{epsilon}) + \beta$$

where x is the input to the layer, mean and variance are the mean and variance of the input batch, epsilon is a small constant used to avoid division by zero, gamma and beta are learnable parameters

Optimizers.

→ In deep learning, optimizers are algorithms that are used to update the weights of a neural network during training. The goal of an optimizers is to minimize the loss function by adjusting the weights in the network.



$Loss = (y - \hat{y})$

* optimizer helps to reduce loss by updating weights during backpropagation using weight updation formula

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}}$$

a)

→ Gradient descent is one of the basic optimizer which uses entire datapoints to calculate the gradient and hence it updates the weights in every epoch.

→ The major problem of gradient descent is that, it require more RAM to store and process data.

→ Convergence in case of deep neural network will take time and hence it is computationally more expensive.

b) Stochastic Gradient descent (SGD)

→ It is the variation of gradient descent to update the weights during backpropagation.

→ Here, instead of taking all the records of a dataset for gradient calculation and updation, one record at a time is considered for one epoch.

But again, the computational time for performing iterations in each epoch will be more.

→ Hence, convergence will be slow, but it requires less RAM., Loss = $(y-\hat{y})$

c) mini-batch stochastic gradient descent (mini-batch SGD)

→ Here, for every epoch some batch-size of data is selected for performing weight-updation.

→ It requires less time compared to SGD and convergence will be faster.

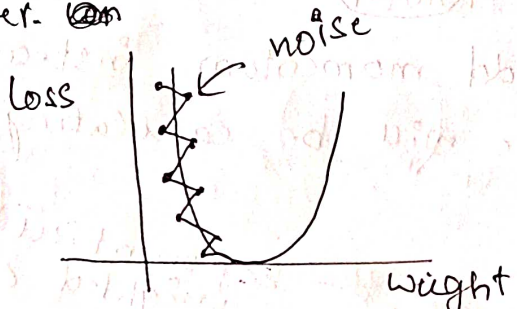


Fig 1: mini-batch gradient descent

if K is the batch size,

$$\text{Loss} = \sum_{i=1}^{K}(y-\hat{y})$$

d) SGD with momentum:

→ It is a variation of basic SGD algorithm that incorporates the notion of momentum to accelerate the convergence to the minimum of the loss function.

→ In traditional SGD, the weight updates at each iteration are based solely on the gradient of the loss function with respect to the parameters. In contrast, SGD with momentum takes into account the previous updates and the current gradient to determine the new update direction. This helps to smooth out the oscillations that can occur in the gradient descent trajectory and results in a faster convergence.

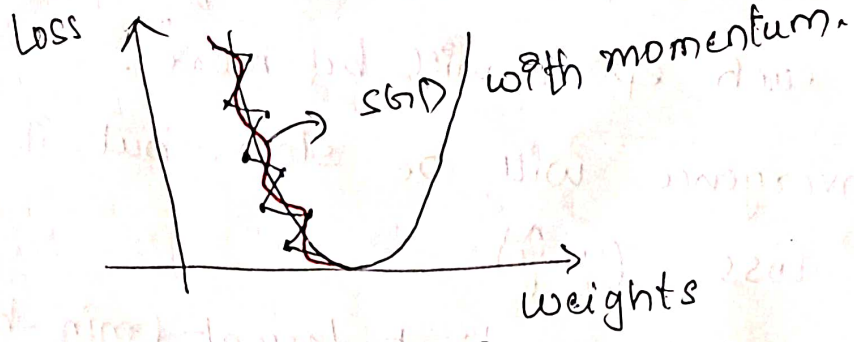


fig: SGD with momentum

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}}$$

$$= w_t = w_{t-1} - \eta \frac{\partial L}{\partial w_{t-1}} \leftarrow \text{gradient descent}$$

If we want to add momentum, instead of calculating gradient descent we will be calculating exponential weighted average.

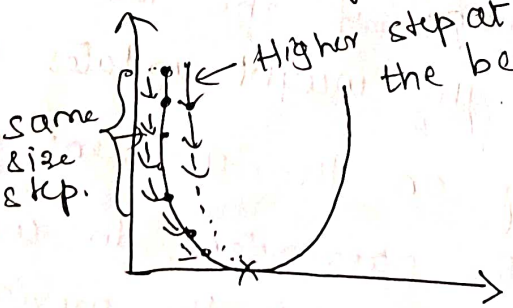
i.e $w_t = w_{t-1} - \eta V dw_t$ ← exponential weighted average

$$V dw_t = \beta \times V dw_{t-1} + (1-\beta) \times \frac{\partial L}{\partial w_t} \quad \text{where } \boxed{\beta = 0.95}$$

Because of this, noise will be reduced and curve to reach global minima will be smoothened.

e) Adagrad- (Adaptive Gradient Descent)

→ Here the learning rate(η) is not fixed.



→ learning rate is changed dynamically during training process.

→ Hence, reaching global minima is faster compared to normal gradient descent curve.

→ weight updation formula,

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w_{t-1}} \rightarrow \text{gradient descent}$$

$$w_t = w_{t-1} - \eta' \frac{\partial L}{\partial w_{t-1}} \rightarrow \text{Adagrad with adaptive learning rate.}$$

where, $\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$ → Initial learning rate

* where ϵ is a small positive number to avoid divide by zero error.

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

why we need η' because the learning rate should be decreased slowly.

→ α_t during iterations becomes very large during each iteration.

→ Hence, at each iteration η value gets reduced.

Disadvantage

1. If we have a very deep neural network, at some point during iteration, α_t will be a very large number and consequently learning rate (η') becomes very very small,

→ such as $\boxed{w_t \approx w_{t-1}}$, causing negligible change in weights.

→ Training of neural network stops and convergence will not happen.

f) Adadelta - RMSPROP optimizer.

→ This is mainly used to overcome the drawback of adagrad optimizer

→ This is also having adaptive learning rate

→ weight updation formula,

$$w_t = w_{t-1} - \eta' \frac{\partial L}{\partial w_{t-1}}$$

where $\eta' = \frac{\eta}{\sqrt{sdw + \epsilon}}$

where $sdw_t = \beta sdw_{t-1} + (1-\beta)\left(\frac{\partial L}{\partial w_t}\right)^2$

(weighted average; β: 0.95; large growth of value will be minimized.)

→ Hence learning rate will not be reduced to very very small value.

g) Adam - optimizer [Adaptive Moment Estimation)

→ It is the most efficient optimizer.

→ It uses the concept of momentum for weight calculation as well as adaptive learning used in RMS-PROP optimizer.

→ The weight updation formula

$$w_t = w_{t-1} - \frac{\eta \times Vdw}{\sqrt{Sdw + \epsilon}}$$

where $Vdw_t = \beta_1 Vdw_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$

$Sdw = \beta_2 \times sdw_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$