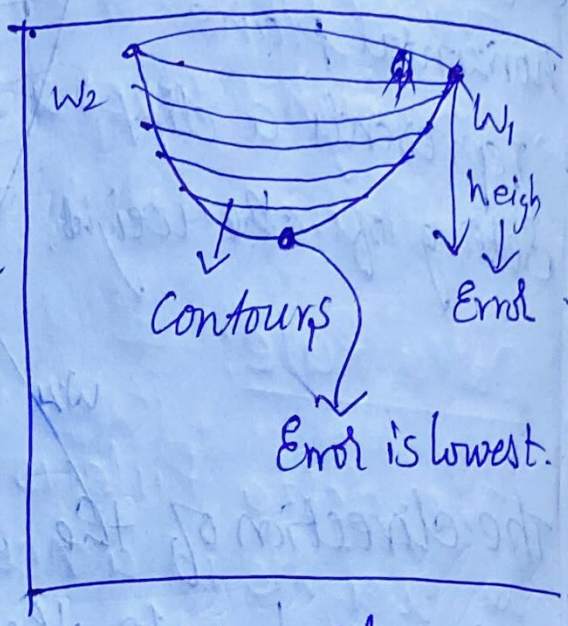
Unit 1 Chapter 2

Gradient Descent:



G.D. is known as one of the most commonly used optimization algorithm to train ML models by means of minimizing errors between actual & Expected results.

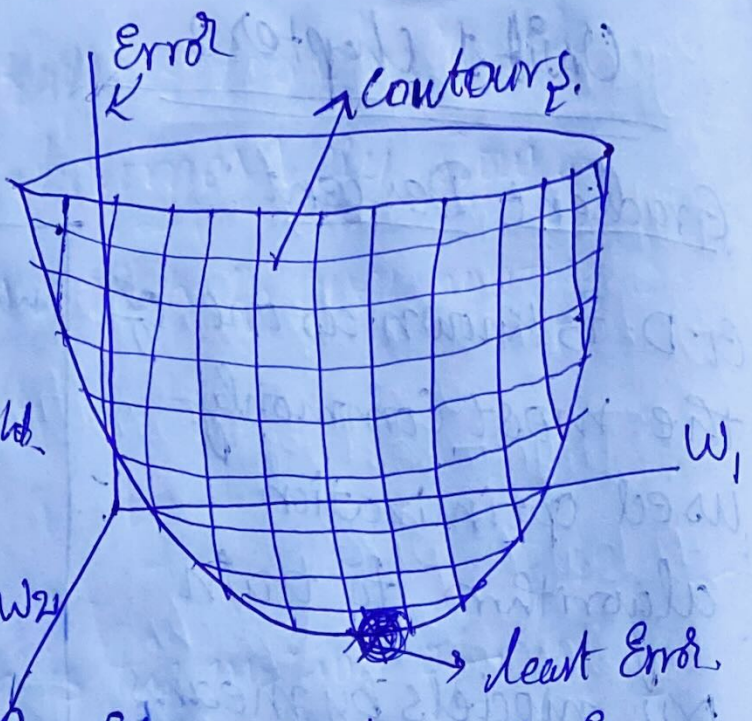
→ It helps to find the local minimum of a function

→ The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.

→ The 3D space where the horizontal dimensions corresponds to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the Error function E.

→ As shown in figure the circles are Elliptical "Contours". where the minimum Error is at the center of the ellipse.

* Each point on the horizontal plane represents a diff setting of the weight.



The direction of the steepest descent is always Ire to these contours.

→ We start by initializing the weights (w_1 & w_2) randomly.

→ Evaluate the gradient at this starting position to find the direction of steepest descent.

→ Take a step in the direction of this gradient

→ After the step, you will be at the new position with a potentially lower Error.

→ Reevaluate the gradient at this new position and again move in the direction of steepest descent.

→ This Iterative Evaluating the gradient and taking steps in its directions, we

→ Progressively move closer to the point of minimum Errors.

→ This Iteration process is the main in gradient descent.

The Descent Rule & Learning Rule :-

→ In neural networks, apart from weights, there are other parameters known as hyperparameters that help guide the training process. one such "hyperparameter is "Learning Rate."

→ The "Learning Rate" controls how much we adjust the weights of our Networks wrt to the gradient.

→ When using the gradient descent the learning rate determines how big a steap we take at each iteration.

→ The Steepness of the Error Surface can indicate how close we are to minimum Error.

- → when Surface is flat → we are closer to minimum. Should take smaller steps.
- → Surface is steep → take larger steps.

→ If the learning Rate is too small, the training process becomes very slow.

→ if too high, there's a risk of overshooting the minimum.

Delta Rule:

The delta Rule is the method of updating the weights in a neural network.

It involves calculating the gradient of the Error function wrt each weight.

By apping the gradient and the learning rate, the delta rule adjusts each weight to reduce the error.

we know that:

$$\Delta W_k = -\epsilon \frac{\partial E}{\partial W_k} \quad \text{—①}$$

We also know that

$$E = \frac{1}{2} \sum (t_k - O_d)^2 \rightarrow ②$$

(t_k: target, O_d: predicted)

② in ①

$$\Delta w_k = -\epsilon \frac{\partial}{\partial w_k}\left[\frac{1}{2}\sum (t_i - o_d)^2\right]$$

$$= -\epsilon \frac{1}{2} \cancel{2}(t_i - o_d) \frac{\partial}{\partial w}(t_i - o_d)$$

$$= -\epsilon (t_i - o_d) \frac{\partial}{\partial w}\left(\frac{t_i}{0} - \frac{(w_i \cdot x)}{1}\right) \quad \boxed{\because o_d = w_i \cdot x}$$

$$= -\epsilon (t_i - o_d)(-x)$$

$$\boxed{\Delta w_k = \sum_i \epsilon x (t_i - o_d)}$$

Applying this method of changing the weights at Every Iteration, we are finallyable to utilize gradient descent.

Gradient Descent with Sigmoidal Neurons

Lets recall the mechanism by which logistic neurons compute their output value from their inputs:

$$z = \sum_k w_k \cdot x_k$$

$$y = \frac{1}{1 + e^{-z}}$$

The neuron computes the weighted sum of its inputs, the bias "z". It then feeds the bias into the input function to compute y, its final output.

To do so, we start by taking the derivatives of the bias wrt input & the weight:

$$\nabla w_k = \frac{\partial E}{\partial w_k}$$

$$\therefore E = \frac{1}{2}(f(x) - y)^2$$

$$\because f(x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$\frac{\partial E}{\partial w_k} = \frac{1}{2}\left[\frac{1}{1+e^{-(wx+b)}} - y\right]^2$$

$$= \frac{1}{2}\left[2 \times (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)\right]$$

$$\frac{\partial E}{\partial w_k} = (f(x) - y) \times \frac{\partial}{\partial w}(f(x) - y) \quad \text{---(1)}$$

$$\therefore \frac{\partial}{\partial w}(f(x) - y)$$

$$= \frac{\partial}{\partial w}\left[\frac{1}{1+e^{-(wx+b)}} - y\right]$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} \frac{\partial}{\partial w}\left(e^{-(wx+b)}\right)$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} * e^{-(wx+b)} \frac{\partial}{\partial w}\left(-(\underset{1}{w}x + \underset{0}{b})\right)$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} \times e^{-(wx+b)} (-x)$$

$$= (+x) \frac{-1}{(1+e^{-(wx+b)})^2} \times e^{-(wx+b)}$$

$$= \frac{-1}{1+e^{-z}} \times \frac{e^{-z}}{1+e^{-z}} (-x) \qquad (z = wx+b)$$

$$\frac{\partial f}{\partial w_k} = f(x) \times (1 - f(x)) (-x) \rightarrow (2)$$

keep (2) in (1) $\quad f(x) = y$

$$= [f(x) - y]\,[f(x) \times (1 - f(x)) (-x)]$$

$$= -x\, f(x)(f(x) - y)(1 - f(x)) \frac{\partial y}{\partial w_k} \qquad \Delta E_k = -\epsilon \frac{\partial E}{\partial w_k}$$

$$\Delta w_k = -x\, f(x)(f(x) - y)(1 - f(x))$$

$$w_{t+1} = w_t - \eta \nabla_{w_t}$$

$$= w_t - \eta \sum_i (f(x) - y)(1 - y)(t - y)$$

$$w_{t+1} = w_t - \eta (f(x) - y) f(x)(1 - f(x)) x$$

As you notice, the new modification rule is just like the delta rule, except with extra multiplicative terms included to account for the logistic component of the Sigmoidal neuron.

Backpropagation algorithm: & reducing Error.

algorithm:

Step 1: Calculate the Error at the output

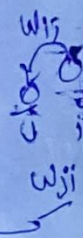
* Start with the output layer of the neural network
* for each neuron i in the output layer, calculate the Error (E) as the Sum of the Squares of the difference between the Predicted output (y_i) and the true target (t_i)

$$E = \frac{1}{2}\sum_j (t_j - y_j)^2$$

* compute the derivate of the Error wrt each output neuron's activity:

$$\frac{\partial E}{\partial y_j} = \frac{1}{2} \times 2 (t_j - y_j) \frac{\partial}{\partial y_j} (t_j - y_j)$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



Step 2: Back propagate the Error.

* Presume that the Error derivatives for layer j are known.
* To calculate the Error derivates for the Preceding layer i, accumulate info on how each neuron in layer "i" influences the neuron in layer "j".
* This can be done as follows using the Partial derivate of the bias wrt incoming Output data from the layer beneath is merely the weight of the connection w_{ij}.

$$\frac{\partial E}{\partial y_i} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial y_i}$$

$$= w_{ji} \frac{\partial E}{\partial z_j} \rightarrow (1) \quad \left(\frac{\partial z_j}{\partial y_i} = w_{ji} \right)$$

$y_i(1-y_i)$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_i} y_i(1-y_i) \rightarrow (2)$$

(1) & (2)

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ji} y_j (1-y_j) \frac{\partial E}{\partial z_j}$$

Step3: update the weights.

* After calculating the partial derivatives for all units, update the weights to minimize the Error, using the learning rate "η"

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \quad \text{—(3)}$$

We know

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial z_j}{\partial w_{ji}} \frac{\partial \epsilon}{\partial z_j}$$

From (2)

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times \frac{\partial z_j}{\partial w_{ji}}$$

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times y_i \qquad \left[\because \frac{\partial z_j}{\partial w_{ji}} = y_i\right]$$

~~Now keep the~~

Now for all the training Examples, sum up the partial derivatives to adjust the weights:

$$\Delta w_{ji} = -\eta \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times y_i.$$

$$\boxed{\Delta w_{ji} = -\eta \sum_{x \in \text{dataset}} y_i\, y_j(1-y_j) \frac{\partial \epsilon}{\partial y_j}}$$

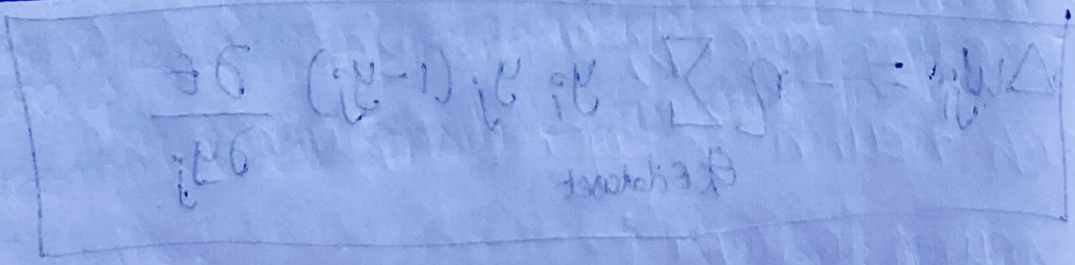
Step 4: Iterate the process:

Repeat ~~steps~~ 1 to 3 for each layer in the network, moving from the output layer towards the input layer.

Iterate the process across all layers for multiple epochs or until the network converges to a satisfactory level of Error.

Stochastic and Minibatch Gradient

- after Evaluating Each individual training Example.
  - It is a compromise b/w batch gradient descent & Stochastic gradient descent.
  - It Computes the gradient against a Subset of the data known as mini batch.



Different techniques to prevent overfitting.

1) ~~Regularization~~

Overfitting :-

the model becomes very good at predicting or fitting the data it has been but fails to generalize to data that hasn't been trained on.

1) **Regularization :-**

* Regularization is a technique to prevent overfitting by adding a penalty for larger weights in the objective function.

* The modified objective function becomes Error + $\lambda f(\theta)$, where $f(\theta)\uparrow$ with larger weights & λ is the regularization strength, a hyperparameter.

* Choosing λ is crucial : $\lambda = 0$ means no Regularization while a large λ might prioritize small weights over fitting the training data.

L2 Regularization (Weight Decay):

The most common regularization in ML, L2 Reg. adds $\frac{1}{2}\lambda W^2$ to the Error function for every weight "W".

It penalizes large weights, Preferring more evenly distributed weight values and Prevents Overfitting by encouraging the whole network to use all inputs rather than a few inputs a lot.

During gradient descent, L2 causes weights to decay linearly towards zero.

3) L1 Regularization:

* Adds the term |W| for each weights in the network.
* Leads to Sparsity in the weight vector, making the model less sensitive to noise in the inputs.
* useful for feature Selection as it indicates which features ~~descent~~ are most important for Predictions.

4) Max Norm Constraints:-

* Enforce an absolute upper bound on the magnitude of the weights vectors.
* If the weight exceeds the bound during the gradient descent update, it is projected back onto a predefined bound.
* Prevents weights from growing too large, ensuring stability in the learning process.

5) Droupout:-

* A technique where each neuron is kept active with a certain probability P during training, & set to zero otherwise.
* makes the network less reliant on any single neuron, effectively combining many different neural network architectures.
* Dropout can prevent Overfitting by ensuring the network remains accurate. Even when certain neurons are dropped.

6) Inverted Dropout:

* A variation of dropout where the neuron's output is scaled at training time, rather than test time.
* Ensure that a test time, the neuron's output do not need to be scaled, maintaining the expected output level.