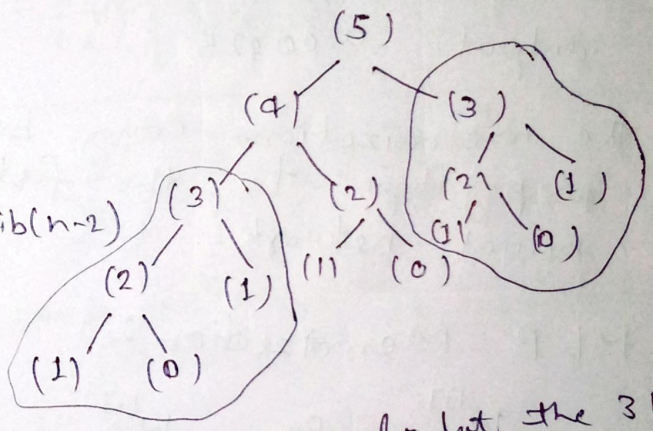
Date - 26-Aug 2024

Memoization:

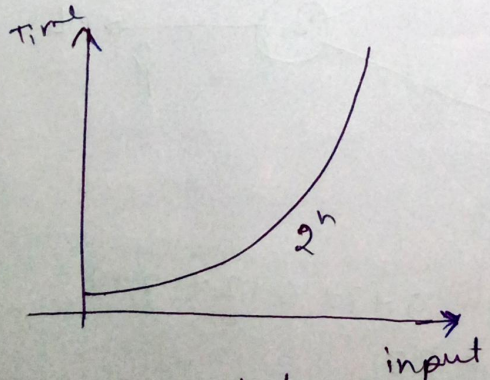
- In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function call and returning the cached result when the same inputs occur again.
- here the disadvantage is that it require more storage.
- example. without memoization

```
import time.
def fib(n)
    if n==1 or n==0:
        return 1
    else:
        return fib(n-1)+fib(n-2)
start = time.time()
fib(38)
print(time.time()-start)
```

output: 19.80



Here we calculate the 3! two time that means it is waste of time and storage.



exponential graph.

Time complexity = $O(2^n)$

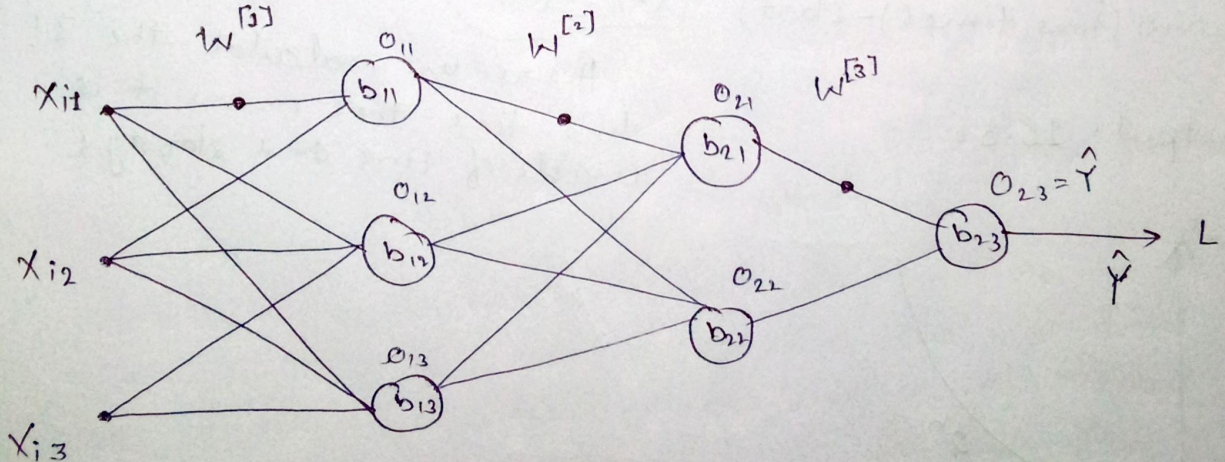
- To resolve this problem we can use the memoization.

```
import time
def fib(n,d):
    if n in d:
        return d[n]
    else:
        d[n] = fib(n-1,d) + fib(n-2,d)
        return d[n]
start = time.time()
d = {0:1, 1:1}
fib(38, d)
print(time.time() - start)
```

output: 0.00025

- The memoization can be used in the back-propagation to the faster execution in the neural network.

MLP Memoization:



Total trainable parameter = 9 + 3 + 6 + 2 + 2 + 1

= 23

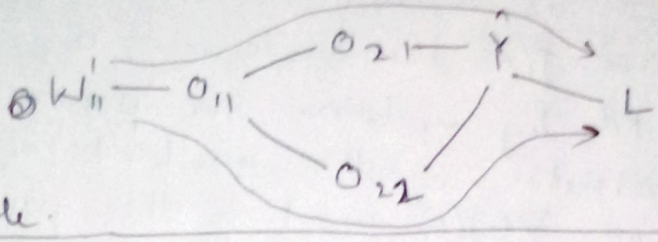
$L \to \hat{Y} \to W_{11}^3$

$$\frac{\partial L}{\partial W_{11}^3} = \frac{\partial L}{\partial \hat{Y}} \times \frac{\partial \hat{Y}}{\partial W_{11}^3} = \quad \text{——— (i)}$$

$L \to \hat{Y} \to O_{21} \to W_{11}^2$

$$\frac{\partial L}{\partial W_{11}^2} = \frac{\partial L}{\partial \hat{Y}} \times \frac{\partial \hat{Y}}{\partial O_{21}} \times \frac{\partial O_{21}}{\partial W_{11}^2} = \quad \text{——— (ii)}$$

$L \to \hat{Y} \to O_{22} \to O_{11} \to W_{11}^1$



example:

$$x \to f(x), g(x) \to h(f(x), g(x)) \qquad \frac{\partial h}{\partial x} = \left[\frac{\partial h}{\partial f(x)} \times \frac{\partial f(x)}{\partial x}\right] + \left[\frac{\partial h}{\partial g(x)} \times \frac{\partial g(x)}{\partial x}\right]$$

$$\frac{\partial L}{\partial W_{11}^1} = \frac{\partial L}{\partial \hat{Y}}\left[\frac{\partial \hat{Y}}{\partial O_{21}} \times \frac{\partial O_{21}}{\partial O_{11}} \times \frac{\partial O_{11}}{\partial W_{11}^1}\right] + \frac{\partial \hat{Y}}{\partial O_{22}} \times \frac{\partial O_{22}}{\partial O_{11}} \times \frac{\partial O_{11}}{\partial W_{11}^1} \quad \text{——— (iii)}$$

* Backpropagation → chain differenciation + memoization [rule]

Date - 29 Sep 2024

GD in Neural Network

- Gradient descent is one of the most popular algorithm to perform optimization and by far the most common way to optimize neural networks.

- Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t to the parameters.

- The learning rate η determines the sizes of the steps we take to reach a (local) minimum. in other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

$J(\theta)$ - loss function

Types of Gradient Descent

- Batch.
- Stochastic
- mini batch.

variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update

(i) Batch GD

pseudo code:

```
epoch = 10
for i in range (10):
    select the entire dataset value.
    y_hat = np.dot (X.W) + b
    calculate loss using y_hat, y
    now update w, b using  w_n = w_o - η ∂L/∂w
    print(loss)
```

here update the w and b 10 times (number of epoch)

code:

```
for i in range (nb_epochs):
    params_grad = evaluate_gradient (loss_function, data, params)
    params = params - learning_rate * params_grad
```

(ii) Stochastic GD:

pseudo code:

```
epoch = 10
for i in range(epochs):
    for j in range ( X.Shape[0]):
        select 1 raw (random)
        predict (using forward propagation)
        calculate loss (using loss function → mse)
        update weights and bias using GD
                W_n = W_o - η ∂L/∂W
    calculate avg loss for epoch
```

Code:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

* Here frequency of weigh update is higher

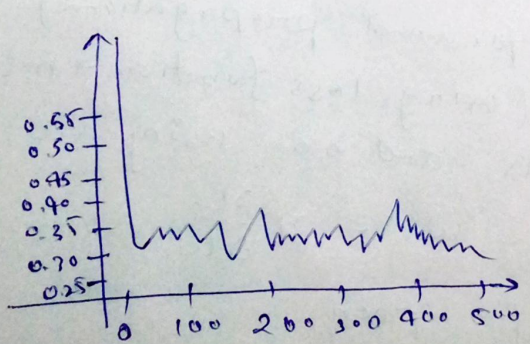
Ques which is faster Batch GD or stochastic GD. (given same number of epchs)

Answer – Batch GD.

* if in the code batch_size = 1 then it is stochastic GD if batch_size = number of raw in data. then it is batch GD

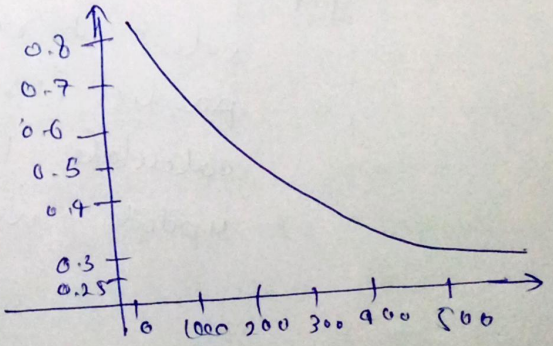
Ques which is the faster Batch GD or stochastic GD to converge (given same number of epochs)

Answer – stochastic (Because of more number of updates)



stochastic GD

Here the loss reduce in very unstable way



Batch GD.

Here the loss reduce in stable/smooth way

Stochastic GD → select random point → updates (w, b)

Batch GD → select all point → updates (w, b)

* SGD help the algo to move out of local minima. but in the Batch GD we got a approximate solution not a exact solution.

vectorization:

np.dot(x, w) + b

- for very large dataset it not works
- faster for small dataset, replacement of for loop

(iii) Mini Batch GD:

code:

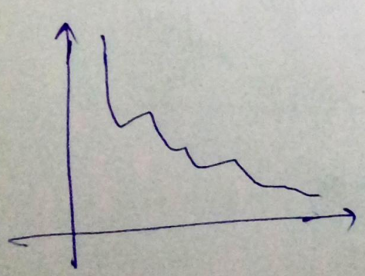
```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

* it resolve the all the drawbacks of batch GD and stocastic GD.

320 rows → 10 batch → 10 updates
(32 row each)

pseudo code:

```
for i in epochs:
    for j in number of batches:
        select 1 batch.
        calculate Y_pred (vectorization)
        calculate loss
        update (w, b)
```



BGD > mB GD > SG
convergence → bgd < mbgd < sgd

Ques Why batch-size is provided in keras in multiplicative of "2" ?

2, 4, 8, 32, 64, 128, 256

for use RAM effective. (optimization technique)

ans What if batch_size doesn't divide perfectly # raws properly.

e.g. # of raws n = 400

batch size = 150

$$\text{Batch} = \frac{400}{150} = 2.66$$

then

150, 150, left 100

↑ 1st Batch

↑ 2nd Batch

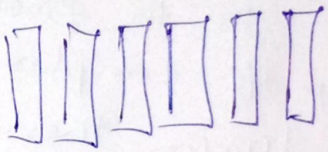
↑ 3rd Batch.

Vanishing Gradient Problem

- in ML, the vanishing gradient problem is encountered when training ANN with GD and backpropagation. In such methods, during each iteration of training each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight.
- The problem is that in some cases, the gradient will be vanishingly small effectively preventing the weight from changing its value. In the worst case, this may completely stop neural network from further training

(i) $0.1 \times 0.1 \times 0.1 \times 0.1 \rightarrow 0.0001$

(ii) Deep NN →



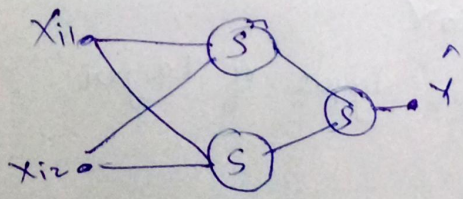
(iii) sigmoid / tanh

example:

sigmoid & derivation [0, 1]

$$\frac{\partial L}{\partial w_{11}^{1}} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w_{11}^{1}}$$

$0<x<1$ $\quad 0<x<1$ $\quad 0<x<1$ $\quad 0<x<1$



S → sigmoid function.

$$w_n = w_o - \eta \frac{\partial L}{\partial w}$$

~~let~~ let $w_0 = 1$

$w_n = 1 - 0.01 \times 0.001$

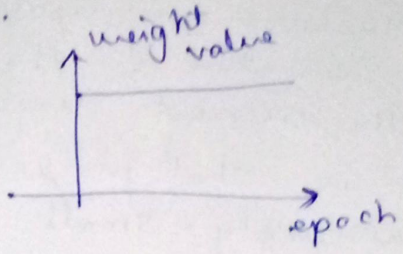
$w_n = 0.99999$

(vanishingly small) → very minute change occur

* due to this backpropagation not able to converge and model not able to training.

How to recognize?

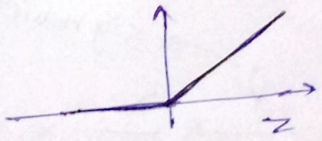
1. Loss focus → epoch → no change → VGP
2. weights graph.



How to handle Vanishing Gradient Problem:

(i) Reduce model complexity
not every time use. because in NN our aim is to find complex pattern with the help of complex architecture, if we reduce the layer then not able to find complex patterns.

(ii) use different activation function. e.g. ReLU:
max(0, z)



(-1000, 100)
↓
(0, 1000)

Here is one problem dying ReLU → derivative 0
to Resolve this issue use Leaky ReLU

(iii) proper weight initialization:
use example- glorot, xavior

(iv) Batch normalization: it is type of layer.

(v) using a Residual Network:
it is type of block used in ANN
↘ CNN → ResNET