**A project report on applying calculus and implementation of gradient descent and back propagation.**

**Backpropagation**

Backpropagation training algorithm is gradient descent using an efficient technique for computing the gradients automatically in just two passes through the network (forward and backward). The backpropagation algorithm is able to compute gradient of the network’s error with regard to every single model parameter.

For each training instance, the backpropagation algorithm first makes a prediction(forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection(reverse pass), and finally tweaks the connection weights to reduce the error.

Foundations of Backpropagation:

1. Forward pass: Input is passed through network layer by layer to obtain output
2. Backward pass: The gradient of the loss with respect to the weights of the network is calculated using chain rule of derivates.
3. Loss calculation: Difference between predicted output and actual output. Computed using a loss function

Weight update: Weights are update using gradient descent to reduce the loss. *z*[1]=*W*[1]⋅*x*+*b*[1]

Let’s apply both forward and backward operations from scratch on a moderately complex function. The function involves a forward pass through the network and a backward pass to compute gradients and update the weights using the chain rule of calculus.

Lets say we have 2 neuron for input layer and hidden layer and 1 neuron for output layer

The activation function use the sigmoid function which is given by:

Assume

**Forward pass :**

Hidden layer Calculation is performed using

We get,

. + = , the result is obtained after performing matrix multiplication

**Activation function:**

= =

**Output layer Calculation:**

*W*[2]. *a*[1]. +*b*[2] = + 0.3 = 28.8

Sigmoid Activation

sigmoid( ≈ 1 since close to small number.

**Loss calculation**

L = =

**Backward pass:**

It involves computing gradient of the loss with respect to the weights and biases.

**Gradient of loss with respect to output :**

= as

**Gradient of loss with respect to output :**

= .

Code implementation:

Import numpy as np

x = np.array([1, 2])

y = 0

W1 = np.array([[2, 2], [3, 4]])

b1 = np.array([0.1, 0.2])

W2 = np.array([1, 2])

b2 = 0.3

alpha = 0.1

def sigmoid(z):

return 1 / (1 + np.exp(-z))

def relu(z):

return np.maximum(0, z)

z1 = np.dot(W1, x) + b1

a1 = relu(z1)

z2 = np.dot(W2, a1) + b2

a2 = sigmoid(z2)

loss = 0.5 \* (a2 - y) \*\* 2

dL\_da2 = a2 - y

dL\_dz2 = dL\_da2 \* a2 \* (1 - a2)

dL\_dW2 = dL\_dz2 \* a1

dL\_db2 = dL\_dz2

dL\_da1 = dL\_dz2 \* W2

dL\_dz1 = dL\_da1 \* (z1 > 0)

dL\_dW1 = np.outer(dL\_dz1, x)

dL\_db1 = dL\_dz1

W2 -= alpha \* dL\_dW2

b2 -= alpha \* dL\_db2

W1 -= alpha \* dL\_dW1

b1 -= alpha \* dL\_db1

**What is gradient descent**

Gradient descent is an optimization algorithm used for minimizing any function. It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function. Mathematically, gradient descent is described as:

Repeat until convergence

Here, is a cost function :

If is too small : Gradient descent may be slow

If is too large : Gradient descent may overshoot, never reach minimum

If point is already at local minima gradient descent remain unchanged

Implementation of gradient descent in code:

def gradient\_function(x,y,w,b):    m = x.shape[0]    dj\_dw = 0    dj\_db = 0        for i in range(m):        f\_wb = w\*x[i] + b        dj\_dw\_i = (f\_wb - y[i]) \* x[i]        dj\_db\_i = f\_wb - y[i]        dj\_dw = dj\_dw + dj\_dw\_i        dj\_db = dj\_db + dj\_db\_i    dj\_dw = dj\_dw/m    dj\_db = dj\_db/m    return dj\_dw, dj\_dw

def gradient\_descent(x,y,w\_in,b\_in,alpha,iters):    b = b\_in    w = w\_in    for i in range(iters):        dj\_dw , dj\_db = gradient\_function(x,y,w,b)        w = w - alpha \*dj\_dw        b = b- alpha \* dj\_db    return w , b

To find the minima pf a moderately complex function for 4th- order polynomial using gradient descent:

* Initialize x randomly.
* Compute f′(x).
* Update x using the gradient descent rule.
* Repeat until convergence or a set number of iterations.