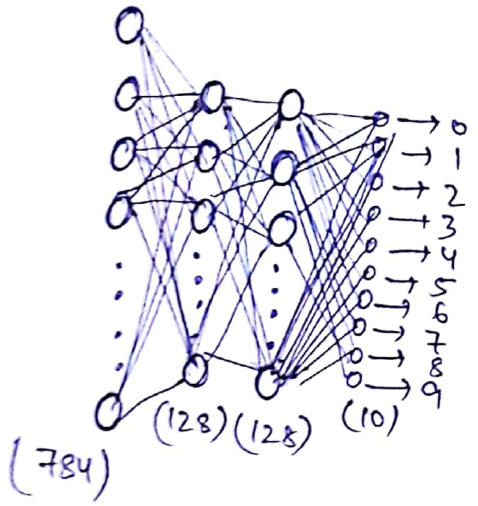
FEED FORWARD NEURAL NETWORK

- Anant Agarwal

Making a feedforward neural network which will classify handwritten digit images from MNIST dataset.
Architecture of our neural net will be something like this,



This is a 4 layered neural net, input will be (784,1), layer 1 will be (128,1), layer 2 will be (128,1) and output layer will be (10,1) for all digits from 0 to 9.

Let's look at the dataset. We'll use Kaggle's MNIST dataset.

labels	pixel 0	pixel 1	pixel 2	......	pixel 783
5	0	0	255		0.
9	.	.	.	. . .	.
3	.	.	. .	. . . .	.
7	.	.	.	. . . . .	.
1	.	.	.	. . . . .	.

defining our 'network' ~~object~~ class.

```
class Network(object):
    def __init__(self, sizes):
        self.num_layers = num(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y,1) for y in sizes[1:]]
        self.weights = [np.random.randn(y,x) for x,y in
                          zip(sizes[:-1], sizes[1:])
```

object = [784, 30, 10]

biases = [b_1 (30,1) , b_2 (10,1)]   weights = [w_1 (30,784) , w_2 (10,30)]

```
def feedforward (self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
    return a
```

In this function, a will be our input feature of dimension, (784, 1). for our current values this loop will go two times for (b_1, w_1) and (b_2, w_2)

$$a_1 = \text{sigmoid}(np.dot(w_1, a_0) + b_1) \rightarrow (30, 1)$$

$$a_2 = \text{sigmoid}(np.dot(w_2, a_1) + b_2) \rightarrow (10, 1)$$

$w_1 \equiv (30, 784)$
$a_0 \equiv (784, 1)$
$b_1 \equiv (30, 1)$

$w_2 \equiv (10, 30)$
$a_1 \equiv (30, 1)$
$b_2 \equiv (10, 1)$

Gradient descent.

```
for _ in range(epochs):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    for i in range(m):
        xx, yy = x[:i], y[:i]
        delta_nabla_b, delta_nabla_w = self.backprop(xx, yy)

        nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    self.weights = [w - (alpha/m) * nw  for w, nw in
                                zip(self.weights, nabla_w)]

    self.biases = [b - (alpha/m) * nb  for b, nb in
                                zip(self.biases, nabla_b)]
```

So in gradient descent, we'll start with matrices of 0 like nabla_b and nabla_w for each epoch.

Inside each epoch,

for each example one by one,

we'll calulalate δ i.e. delta_nabla_b and delta_nabla_w for each example and ~~append~~ add it to our previously 0 initialized vectors nabla_b and nabla_w.

~~Now~~

Now we have matrices nabla_w and nabla_b which have weights and biases ~~changed~~ delta values for all of the examples of that epoch.

Now we'll use this ∇w and ∇b to make changes to our weights.

Backpropagation.

Backpropagation algorithm deals with one example at a time.

And the cost function which we'lll use here is a simple mean-squared error function inspite of the more commonly used cross-entropy error function.

$$\boxed{C = \frac{1}{2}\|y-a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2}$$

Equations of backpropagation

Backprop is about understanding how changing the weights and biases in a network changes the cost function. Ultimately this means computing the partial derivative $\frac{\partial C}{\partial w_{jk}^L}$ and $\frac{\partial C}{\partial b_{jk}^L}$.

But to compute those we'll first introduce a intermediate quantity (δ_j^L) delta, which we can call as the error in the jth ~~of~~ neuron of layer L.

Back propagation will give us a procedure to compute δ_j^L and then will relate (δ_j^L) to $\left(\frac{\partial C}{\partial w_{jk}^L}\right)$ and $\left(\frac{\partial C}{\partial b_j^L}\right)$.

Defing the error in a neuron,

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

$$z \xrightarrow{\sigma} a \xrightarrow{\text{---- operations ----}} C$$

$\delta_j^L = \dfrac{\partial C}{\partial z_j^L}$ ≡ how fast the cost is changing with (z_j^L).

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \left(\frac{\partial C}{\partial a_j^L}\right) \cdot \left(\frac{\partial a_j^L}{\partial z_j^L}\right)$$

≡ (how fast cost is changing with activation) * (how fast activation is changing with z_j^L)

but

$a_j^L = \sigma(z_j^L)$, hence

$$\frac{\partial}{\partial z_j^L}\left(\sigma(z_j^L)\right) = \sigma'(z_j^L)$$

$\therefore$

$$\boxed{\delta_j^L = \frac{\partial C}{\partial z_j^L} = \left(\frac{\partial C}{\partial a_j^L}\right) \cdot \sigma'(z_j^L)} \quad \text{--- (1)}$$

equation for the error in output layer.

The derivative of the sigmoid function will be $g(z) * (1 - g(z))$ where $g(z)$ is the sigmoid function.

Derivative of the cost function we have choosed will be simply $(a^L - y)$.

$\therefore$ the vectorized implementation of equation (1) will be

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\boxed{\delta^L = (a^L - y) \odot \sigma'(z^L)} \quad \text{--- (1) vectorized}$$

An equation for the error δ^L in terms of the error in the next layer, δ^{L+1}.

$$\boxed{\delta^L_j = \frac{\partial C}{\partial z_j^L}} \quad \text{similarly} \quad \boxed{\delta^{L+1}_j = \frac{\partial C}{\partial z_j^{L+1}}}$$

$$\cdots \text{---} \; z^L \; \text{---} \; z^{L+1} \; \cdots \text{---} \; C$$

$\left(\delta^L = \frac{\partial C}{\partial z^L}\right)$, now by using chain rule this could be written as,

$$\delta^L = \left(\frac{\partial C}{\partial z^{L+1}}\right)\left(\frac{\partial z^{L+1}}{\partial z^L}\right)$$

$$\boxed{\delta_L = \left(\frac{\partial z^{L+1}}{\partial z^L}\right) \delta^{L+1}} \quad \text{—(i)}$$

$$\boxed{z^{L+1} = (w^{L+1}) \cdot a^L + b^{L+1}} \quad \text{—(ii)}$$

for combining (i) and (ii) :

$$\frac{\partial z^{L+1}}{\partial z^L} = \frac{\partial}{\partial z^L}\left(w^{(L+1)} \cdot \sigma(z^L) + b^{L+1}\right)$$

$$\boxed{\frac{\partial z^{L+1}}{\partial z^L} = w^{(L+1)} \cdot \sigma'(z^L)}$$

and hence,

$$\boxed{\delta_L = w^{L+1} \cdot \sigma'(z^L) \times \delta^{L+1}} \quad \text{—— } 2$$

$$\boxed{\delta^L = \left((w^{L+1})^T \cdot \delta^{L+1}\right) \odot \sigma'(z^L)} \quad \text{— } 2 \text{ vectorized.}$$

How does cost of the network changes with the bias?

It turns out,

$$\boxed{\frac{\partial C}{\partial b_j^L} = \delta_j^L} \longrightarrow (3)$$

$$\boxed{\frac{\partial C}{\partial b} = \delta} \longrightarrow \text{vectorized } (3)$$

Now, similarly we want to find a way to compute change in cost w.r.t. change in weight of a particular neuron.

Equation (1) gives us a way of finding change in cost with δ.

Equation (2) finds a way to relate weights ~~with~~ of next layer with δ.

$$\boxed{\delta^L = \nabla_a C \odot \sigma'(z^L)} \longrightarrow (1)$$

$$\boxed{\delta^L = \left((w^{L+1})^T \delta^{L+1}\right) \odot \sigma'(z^L)} \longrightarrow (2)$$

Now we'll do some basic algebra to reach a rough conclusion to the required relation,

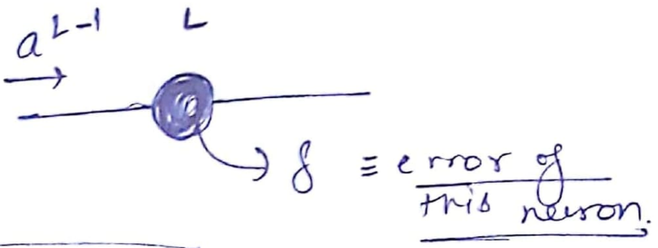
$$\nabla_a C \cancel{\odot \sigma'(z^L)} = \left((w^{(L+1)})^T . \delta^{L+1}\right) \cancel{\odot \sigma'(z^L)}$$

$$\nabla_a C = \boxed{\frac{\partial C}{\partial a^L} = (w^{L+1}) . \delta^{L+}}$$

$$\boxed{\frac{\partial C}{\partial w^{L+1}} = a^L . \delta^{L+1}} \rightarrow \text{now replacing } L \text{ by } L-1$$

$$\boxed{\frac{\partial C}{\partial w^L} = a^{L-1} . \delta^L} \longrightarrow (4)$$

Intuitive meaning of equation 4 -



$$\boxed{\frac{\partial C}{\partial w} = a^{L-1} \cdot \delta^{L}}$$

Code for Backpropagation.

```
def backprop (self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    ## feedforward propagation.
    x = x.reshape(-1,1)    ## adjusting rank 1 python arrays.
    activation = x
    activations = [x]
    zs = [ ]

    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
```

/* Above code does all the feedforward work and stores all the z's and the activations for later use during backward pass step */

```
## backward propagation.

delta = self.cost_derivative(activations[-1], y) * sigmoid_derivative(zs[-1])
    ↳ calculating δ for the first layer
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activation[-2].T)

for l in range(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_derivative(z)
    delta = np.dot(self.weights[-l+1].T, δ) * sp
    nabla_b[-l] = delta
    nabla_w[-l] = np.dot(delta, activations[-l-1].T)
return(nabla_b, nabla_w)
```

/* In the above code, all we did was to implement the equations of backpropagation for all the layers and we made an efficient use of negative indices of python */

Cost Derivative.

For our mean squared cost function, $\frac{1}{2}(a-y)^2$ its derivative suitable here will be, $2(a-y)$ or we can simply use $(a-y)$.

code.

```
## derivative of cost function.

def cost_derivative(self, output_activations, y):
    k = np.zeros((10,1))
    k[y] = 1
    return (output_activation - k)
```