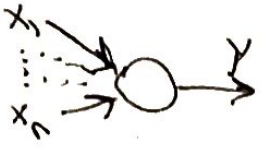
NNet & Deep Learning

(model)

Perceptron = node with several binary inputs $x_1, x_2, \ldots, x_n$ which produces a single binary output y

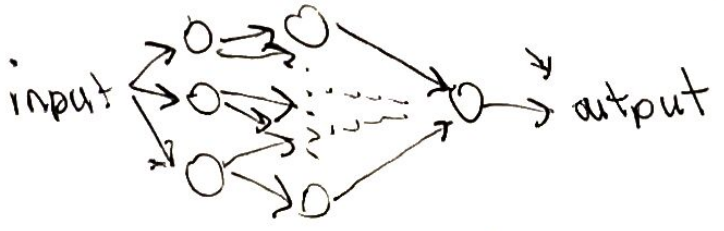


where each input, x_j, a uni-directional weighted edge

$$y = \begin{cases} 0, & \sum_j w_j x_j \leq \tau \\ 1, & \sum_j w_j x_j > \tau \end{cases}$$

where τ is the threshold value

∴ a trivial NNet can be modeled as such:



note: even though perceptron has one output, it can flow through many output edges such that

$$y = \{y_i \mid y_i == y\}$$

(dot product)

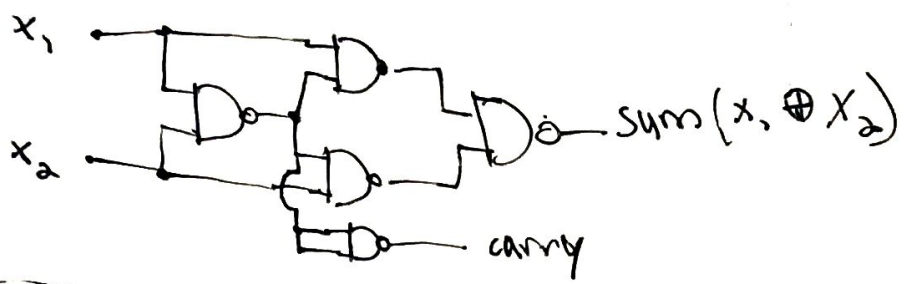
Let $\sum_j w_j x_j = w \cdot x$ where w and x are vectors and

to represent the threshold as a bias where $b = -\text{threshold}$

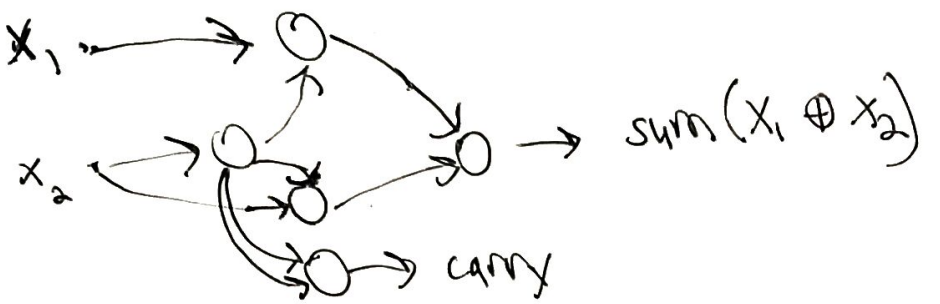
now $y = \begin{cases} 0, & w \cdot x + b \leq 0 \\ 1, & w \cdot x + b > 0 \end{cases}$ (output)

hence forth, bias will represent -threshold

NAND gate with perceptrons



← sum using logical NAND:

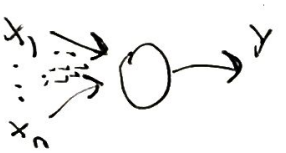


each weighted edge value is -2 and a bias of 3

Just as in hardware architecture NAND's are universal, so are perceptrons for computation

---

Sigmoid Neurons - same as perceptron except the output

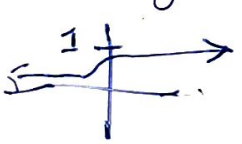


where $y = \sigma(z) \equiv \frac{1}{1+e^{-z}}$

$z = (w \cdot x + b)$

$\Rightarrow \sigma(w \cdot x + b)$

σ is sigmoid function



or

$$\frac{1}{1+e^{-(\sum_j w_j x_j - b)}}$$

i.e. output is always in range of $[0, 1]$

$$\therefore \Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

$\frac{\partial d}{\partial b}$ = partial derivative of d with respect to b

"feedforward" - network where output from one layer is used as input for the next

"recurrent" network - loops are allowed

Gradient Descent

Let $x = \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}$ where x represents the vectorized 28×28 image and a_i is the grayscale value of the pixel.

Let $y = y(x)$ where $y = \begin{bmatrix} a_1 \\ \vdots \\ a_{10} \end{bmatrix}$ representing the 10-dim. output vector

cost function:

$$C(w,b) = \frac{1}{2n} \sum_x \| y(x) - a \|^2$$

where w = weights

b = biases

$\|v\|$ = length of vector

n = # of training inputs

a = output vector containing the correct digit value

$C(w,b)$ aka MSE or mean squared error

→ The closer $C(w,b)$ is to zero the, the more correct the network is.

∴ goal of any nnet is to minimize the cost function ← done via gradient descent

i.e. "learning" means to find the weights and biases that minimize the quadratic cost function

- can't calculate by hand b/c finding derivative of a function with thousands of variables isn't worth it.
- So the nnet will do this by find the slope (or gradient) of the function and determine the min through many iterations

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \quad \text{where } \Delta v = (\Delta v_1, \Delta v_2)^T$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

$$\Delta C \approx \nabla C \cdot \Delta v \quad , \quad \Delta v = -\eta \nabla C$$

$$\Delta C \approx \nabla C \cdot -\eta \nabla C = -\eta \| \nabla C \|^2$$

$\therefore$ the gradient is $v \to v' = v - \eta \nabla C$

i.e. compute gradient repeatedly going in the negative direction

note: η needs to be small enough s.t. $\Delta C \leq 0$, otherwise the network will "go uphill" and go away from local min.

gradient descent - find weights (w_k) & biases (b_l) which minimize the cost of $C(w,b) = \frac{1}{2n} \sum_x \| y(x) - a \|^2$

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

} update rule for w, b

Stochastic Gradient Descent - estimate gradient of ∇C by instead computing ∇C_x for a small sample of inputs that are randomly chosen. Avg'ing over this sample will give a good estimate of the true gradient ∇C

for a random sample of inputs $X_1, X_2, \ldots, X_m$ given m is a large enough sample size

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_{x} \nabla C_x}{n} = \nabla C$$

$$\Rightarrow \nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$

in stoch. gradient descent

proof:

claim: given a large enough sample size m, stochastic gradient descent estimates any ∇C_x such that

$$\nabla C_x \approx \nabla C$$

∴ Stoch. grad. descent in a neural network will look like: Let w_k, b_l be the weights & biases

$$w_k \to w_k' - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

epoch – Let I be a randomly sorted list of inputs. Perform SGD (stoch. grad. desc.) on all the subsets S such that $I_S = I - S$ and $I_S, S \in I$. An epoch has occured when $|I_S| < |S|$, Therefore, I needs to be reshuffled.

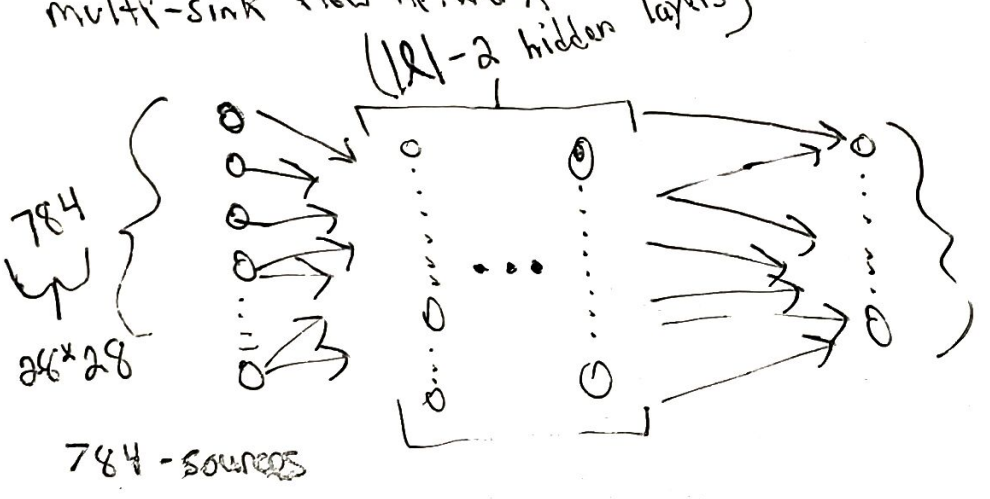
Conv. NNet implementation

Let $\mathcal{L}$ be the "layers" of NNet s.t. $\{l_i \mid l_i \in \mathcal{L}\}$.

l_i is a partition in a $|\mathcal{L}|$-partitioned, edge-weighted flow network of sigmoid neuron nodes with no loops or self-loops. Let $\mathcal{W}$ be the matrix of all the weights of the edges s.t. $\mathcal{W} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{bmatrix}$, $w_{ij} = \text{edge}(l_i, l_j)$. Let $\mathcal{B}$ be the vector of all the biases of each sigmoid s.t. $\mathcal{B} = (b_1, b_2, \ldots, b_n)^T$ where $b_i = -t$, t is the threshold of the sigmoid neuron.

abstract:

multi-src
multi-sink flow network



---

initial values: seed each $w_{ij} \in \mathcal{W}$ and $b_i \in \mathcal{B}$ with a random scalar from the Gaussian distribution (normal dist)

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

used $\mu = 0$
$\sigma = 1$

$$\therefore \quad \mathcal{W} = \{ [w_{ij}] \in \mathcal{W} \mid w_{ij} = \mathcal{N}(w_{ij} \mid 0, 1) \}$$
$$\mathcal{B} = \{ [b_i] \in \mathcal{B} \mid b_i = \mathcal{N}(b_i \mid 0, 1) \}$$

---

feed forward - Given an input $\mathcal{I}_i$, $\mathcal{FF}(\lambda)$ "feeds" the input down the flow network

for every λ in $\mathcal{I}_i$ s.t. $\{\lambda \mid \lambda \in \mathcal{I}\}$
$$\mathcal{FF}(\lambda) = \sigma(\mathcal{W}(\lambda) \cdot \lambda + \mathcal{B}(\lambda))$$
end

σ is sigmoid func., $\mathcal{W}(\mathcal{I}_i)$ is matrix of weights s.t. $\mathcal{W}(\mathcal{I}_i) = \{ [w_{ij}] \in \mathcal{W} \mid w_{ij} = \text{edge}(\mathcal{I}_{ij}, \overrightarrow{\lambda_{i+1,j}}) \}$
↑ weight of ( )
λ is layer of nodes, $\mathcal{B}$ is vector of biases of λ

Stochastic Gradient Descent - Given, TD, a list of 2-tuples (x, y) corresponding to the input, x, and desired output, y. bs is the size of each subset of TD. η is learning rate. epoch is number of iterations ("time") to train for.

for i where $i \in \{0, 1, 2, \ldots, |epoch|\}$

- randomly order of the sets in TD

Let $\mathcal{L} = [A, A_2, \ldots, A_n]$ where $n = |TD|$ and

$$a_i = \{a_i \in \mathcal{L} \mid a_i = (x, y)$$

$$A = [(x,y)_k, (x,y)_{k+1}, \ldots, (x,y)_{k+s}]$$

where $k = \{0, |mb|, 2|mb|, \ldots, \frac{n}{|mb|}|mb|\}$

$\mathcal{L}$ is the list of subsets of training data for each epoch

for each, τ, subset in $\mathcal{L}$

| update_batch(τ, η) where η is learning rate

end

- if you want to evaluate the nnet with, τ, test data

| evaluate(τ) for each τ

end

end

Update Batch - Given a subset $\tau \in \mathcal{D}$, apply gradient descent updating W and β. η is learning rate

for each (x, y) - 2-tuple $\in \tau$

$\Delta(\nabla W) = \text{backpropagation}(x)$

$\Delta(\nabla \beta) = \text{backpropagation}(y)$

end

for each $w_i \in W$ and $\nabla w_i \in \nabla W$

$w_i = w_i - \left(\frac{\eta}{|\tau|}\right) \cdot \nabla w_i$

end

for each $b_i \in \beta$ and $\nabla b_i \in \nabla \beta$

$b_i = b_i - \left(\frac{\eta}{|\tau|}\right) \cdot \nabla w_i$

end

Back propagation

Let $w^l_{jk} \in W$ s.t. w^l_{jk} = weight of edge(j,k) in layer l

Let $b^l_j \in \beta$ s.t. b^l_j = $-$threshold of j^{th} neuron in layer l

Let a^l_j be the activation value of the j^{th} neuron in layer l s.t.

$$a^l_j = \sigma\left(\sum_k w^l_{jk} \cdot a^{(l-1)}_k + b^l_j\right)$$

or in vectorized form ← remember $(wa + \beta)$

$$a^l = \sigma(w^l a^l + \beta^l)$$

$$\sigma\left(\begin{bmatrix} \cdots \\ \cdots \end{bmatrix}\begin{bmatrix} \vdots \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}\right)$$

Let $z^l \equiv w^l a^{l-1} + \beta^l$ be "weight input to the l layer"

goal of back prop is to compute partial derivatives,

$\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$

$$C = \frac{1}{2n}\sum_x \left\| y(x) - a^l(x) \right\|^2$$

n = # of x

remember $C \approx \frac{1}{n}\sum_x C_x$

∴ it can be assumed that computing $\frac{\partial C_x}{\partial w}$, $\frac{\partial C_x}{\partial b}$ for a single subset of training input, $\frac{\partial C}{\partial w}$, $\frac{\partial C}{\partial b}$ can be found by

$$C \approx \frac{1}{n} \sum_x C_x$$

Hadamard product - (s ⊙ t)

$$s \odot t = \begin{bmatrix} i \\ j \end{bmatrix} \odot \begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} i \cdot m \\ j \cdot n \end{bmatrix}$$

Let $\delta \equiv \frac{\partial C}{\partial z_j^l}$ δ_j^l is the error in j^{th} neuron in l layer. So instead of computing

* $\rightarrow \sigma(z_j^l)$, the neuron outputs
* $\rightarrow \sigma(z_j^l + \Delta z_j^l)$ effecting the cost by $\left(\frac{\partial C_j}{\partial z_j^l} \Delta z_j^l\right)$

- So back prop is a way of calculating δ^l for each layer in relation to $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$

remember $\Rightarrow \delta \equiv \frac{\partial C}{\partial z_j^l}$

$$z^l \equiv w^l a^{l-1} + b^l$$

$$\therefore a_j^l = \sigma(z^l)$$

$$\Rightarrow \quad \delta_j^L = \frac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$$

$\frac{\partial C}{\partial a_j^L}$: how fast cost is changing as a func. of j^{th} activation. i.e. how much a depends on the value of C to activate

$\sigma'(z_j^L)$: how fast activation func is changing at z_j^L

* σ' is derivative of sigmoid function

in linear algebra, the above, element-wise, operation can be simplified to

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

$$\nabla_a C = \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix}, \quad \left\{ a_i \in \nabla_a C \;\middle|\; a_i = \frac{\partial C}{\partial a_j^L} \right\}$$

$$\Rightarrow \quad \delta^L = (a^L - y) \odot \sigma'(z^L)$$

Let $k = L+1$

$$\delta^L = \left((w^k)^T \cdot \delta^k\right) \odot \sigma'(z^L)$$

given we know w^{L+1} and $\delta^{(+1)}$, we can calculate δ^L. i.e. "moving the error backs through the network" or "backs" propagation.

So δ_j^l can be used to calculate δ^l then δ^l will be used for δ^{l-1} and so on working backward.

the conclusion can also be drawn that:

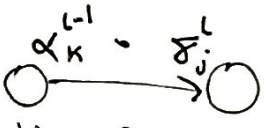
$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial a_j^l}\sigma'(z_j^l)$$

$$\Rightarrow \delta_j^l = \frac{\partial C}{\partial b_j^l}$$

rate of change of cost with respect to any bias

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \cdot \delta_j^l$$

rate of change of cost with respect to any weight

abstractly: 

a is activation of a neuron to the weight and δ is the error of the neuron's output from the weight

- when activation value is closer to zero, the less the gradient will change which means the neuron will "learn" more slowly
- remember also the σ' is derivative of sigmoid  so $\lim_{n \to 1} \sigma'(n) = 0$. Therefore, it also "learns" slowly when activation is high ($\sigma(z_j)$ = near one).

So when activation is near 0 or 1, neuron will "stop" learning

a further conclusion can be drawn. Any weight that is an input to a "saturated neuron" (a neuron whose activation is near 0 or 1), will also learn slowly.

Corollary — proof: definition of δ^L

$\therefore$ Any weight w_j^L will "learn slowly" if either input neuron has a near 0 activation or the output neuron is saturated.

So the 4 functions that back propagation hinge on are:

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

2) $\delta^l = ((w^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l)$

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

4) $\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1} \cdot \delta_j^L$

in essence this is just applying the chain rule in order to approximate closely the gradient of the cost function

Back propagation - compute the gradient of the cost function

- set the activation, a, for the input layer
- "feed forward" - for each layer, l,

$$z^l = w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

- "output error"

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

- back prop the error, for each layer $l = L-1, L-2, \ldots$

$$\delta^l = \left((w^{l+1})^T \cdot \delta^{l+1}\right) \odot \sigma'(z^l)$$

- gradient descent, update weights and biases s.t.

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

, for each layer