*) sigmoid neuron → building block of Deep Neural Networks.

↓

or the foundational block of DNN.

↓

not limited to sigmoid, can be something like ReLU neuron or any other non-linearity

↓

generalised case : sigmoid neuron.

↓

till now i/ps : real i/ps

a task : biclassification

model : linear (can't operate with non-linear data)

$y = \sum_{i=1}^{n} w_i x_i \geq b$

↓

weights and bias were tuneable (perceptron)

indicator fn. i.e. 1.0 loss ← loss : $\sum_i 1_{\{y_i \neq \hat{y}_i\}}$ or MSE

training : perceptron learning algo. or brute search (MP)

evaluation : accuracy

Note : o/p plotted along with the features would result in a kind of a ramp function (for perceptron if linearly separable)

↓

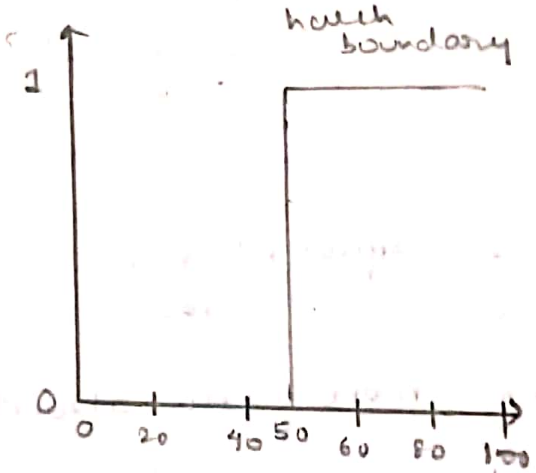
also the decision boundary is very sharp for perceptron. rather practically the decisions are much smoother

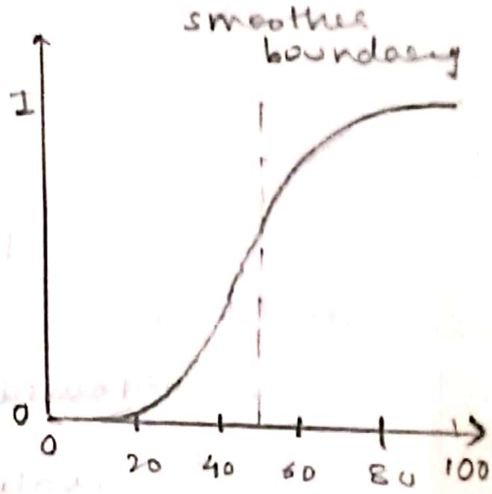
•) a person with salary 50.1 k can afford a car but 49.9 k can't ?

↓

boundary is harsh, and we would rather want a more smooth boundary. where there is some uncertainity associated with a range of i/ps.

salary	buy?
80	1
20	0
65	1
15	0
30	0
49	0
51	1
87	1





↓

the probability on the right of 50 is now ~~too~~ approaching 1 and prob. on the left is not equal to 0.

←

therefore person with a strategy near to 50k also has a prob. of being able to afford a car

↓

which is logical practically.

↓

There probabilities are continous, and hence can also be employed with regression ~~in some way~~, and are not just constrained to ~~to~~ classification.

↓

real valued o/ps.

for sigmoid neuron : i/ps : real

task : classification / regression

↓

real o/ps ✓

model : smooth at boundaries

non - linear

(can deal with non linear fns to a limit, only first step in that direction)

↓

network of neuron

loss : squared error loss

learning : more generic algorithm

evaluation : accuracy & RMSE

(root mean square error)

↓

since this would be used for regression

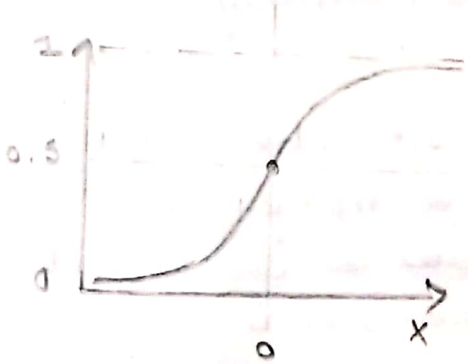
parameters are [w], b

↑

the mapping b/w y and x

? approximated using sigmoid

(*) model : a smoother fn. → sigmoid family of fns.



$$y = \frac{1}{1 + e^{-(wx+b)}}$$

→ single feature

↓

logistic function

↓

always +ve

↓

between 0 & 1 (probability)

substitute values in fn. to visualize.

3. evaluate $A^{\pi}(s,a)$
4. evaluate policy gradien[...] single sample

$\nabla_{\theta} J(\theta) \approx \nabla_{\theta} \log \pi [...] (s) \cdot \hat{A}^{\pi}(s,a)$

*) hue / color can be used to visualize probability smoothness for two feature data, rather than plotting 3D graphs, (with certainity along the ends and uncertainity in the middle)

*) The power of the exponential is the aggregation of the features with their weights.

$$y = \frac{1}{1 + e^{-(w^T x + b)}}$$

↳ scalar quantity.

*) In perceptron model, there is no distinction between the points very close to the decision boundary and the points very far to the decision boundary as the points far from the decision boundary are very sure to be classified correctly, but the ones close would / should still have some uncertainity associated with them.

↓

can be attributed to the harsh boundary and the o/p being either 1 or 0.

↓

The sigmoid neuron deals with this issue as it assigns a ~~higher~~ ~~much~~ ~~probability~~ ~~lower~~ ~~proba~~ higher level of confidence to points far from the decision boundary as compared to the ones close to it.

*) o/p of sigmoid can also be interpreted as probability.

*) does not separate the i/ps, rather giving a more graded o/p.

we probably need to use parallel ...

*) variation according to w and b for the model

w → -ve, then the slope of the curve is -ve, as the denominator ↑, and total value decreases as w ↑

more -ve, steeper curve.

+ve, reverse happens

$$\frac{1}{1+e^{-w}} = \frac{1}{1+\frac{1}{e^{w}}}$$

or $\frac{1}{1+e^{w}}$

$$\frac{1}{1+\frac{1}{e^{w}}} > \frac{1}{1+e^{w}}$$

b → decreasing the value pushes the curve to the right

↓

$$y = \frac{1}{1+e^{-(wx+b)}} = \frac{1}{2}$$ → at the middle

$$\Rightarrow e^{-(wx+b)} = 1$$

$$\Rightarrow wx+b = 0$$

$$\Rightarrow x = \frac{-b}{w}$$ → as b decreases, with constant w, x ↑ (hence shift to the right)

4. evaluate policy π ... (s) . $\hat{A}^{\pi}(s, \ldots)$

$J(\theta) \approx \nabla_\theta \log \pi$

real i/p feature

*) Data and Task

(.) sigmoid neuron can be used for binary classification as well by highlighting which class the feature vector is more closer to . i.e. we are now getting a softer o/p .

↓

to take a decision, some threshold can be applied to these values.

(.) regression : given a feature vector , fitting a real value of y to that based on training data.

↓

for sigmoid , we are regressing the probability.

↓

could be used to o/p the rating for the phone (normalised btw 0 and 1)

*) Loss function : $\sum_{i=1}^{N} (y - \hat{y})^2$ → squared error loss fn

y → true, $\hat{y}$ → prediction

↓

can also be used with binary labels.

↓

for this case the point closer to the labels contribute less to the loss whereas the points away from it contribute more.

we probably ↓ ...

*) learning algorithm

↓

objective: give the data, we want to generalise the mapping between $\hat{x}$ and y, using sigmoid function whose parameters are w and b

↓ need to be tuned to get a appropriate approximation.

i) initialize w, b

ii) iterate over the dataset, and compute loss on predicted value

iii) update w and b, and repeat → the generalised roadmap.

↓

w and b need to updated and approximated in order to minimize the loss over the dataset.

a) method 1: guesswork, (by visualizing the points and the function)

↓

and accordingly taking the decisions to vary w and b.

↓

but, yes not scalable to higher dimensions.

↓

guesswork or random changes would take time to actually approximate

: initialize $w=0$, $b=0$, straight line ($y=0.5$)

↓

change $w=1$, $b=0$, somewhat better (+ve slope)

↓

going from $w=1$ to $w=3$ (slope ~~more~~ steeper)

↓

moving the fn. to the right (decreasing b)

↓

again changing w and b a little works.

↓

in other words, for each iteration, $w = w + \Delta w$

$b = b + \Delta b$

3. evaluate A ... r single ...

4. evaluate policy gradie...

$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi ... |s) \cdot \hat{A}^\pi(s,a)$

*) for guesswork were we guided by the loss function ?

↓

yes in a way the guesswork was guided by the loss function, but it can only get us so far.

(it is easier when there are only 2 or 3 features, but for high dimensional data, this would be next to impossible to determine)

↓

observation: Also after one point of decreasing the loss, the loss began to increase, and we had to move back a little to reach the optimal w and b for minimal loss.

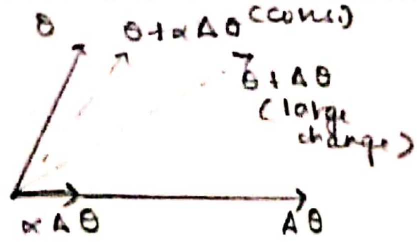
↓

for guesswork, the trajectory of optimizing w and b to minimize loss has been very random and full of fluctuations, and in high end dimensions, where the guesswork will move in random directions and hence reaching the minima will require a large no. of steps.

↓

so this motivates the need for a more principle algorithm that allows us to move such a way that there is gradual decrease in the loss associated with w and b.

b) method 2 : iterative change



$\theta = \begin{bmatrix} w \\ b \end{bmatrix} \quad \Delta\theta = \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix}$

$\theta = \theta + \alpha \Delta\theta$

how to compute $\Delta\theta$?

↓

change or diff. vector

↓

gradient of loss with respect to θ

↓

frameworks like TensorFlow and pyTorch have functions that give these gradients ~~and~~ and these are used to tune θ at each iteration.

*) Taylor series :

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2!} f''(x)\Delta x^2 + \frac{1}{3!} f'''(x)\Delta x^3 + \dots$$

↓

value in close proximity to x, given by RHS

↓

for our problem, we take a small step away from w and we need that the loss after taking the small step decreases (ideally)

↓

Require a relation between the loss before and after the small step.

↓

can be accomplished using Taylor series.

↓

where the value after the small step is equal to the value of f(x) plus some quantity, which comprises of Δx.

↓

if Δx ~~~~ is picked such that the added quantity is negative then it can be concluded that the loss after the step is less than before.

↓

the more -ve, the better, since the loss will decrease by more.

↓

along with Δx and the constants, there are n^{th} order derivatives in the quantity.

↓

loss depends on b as well $\therefore$ $b \rightarrow b + \eta b$

$L(b) > L(b + \eta b)$

↓

function of both, i.e. $\begin{bmatrix} w \\ b \end{bmatrix} + \eta \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix}$

(prediction depends on both w and b)

↓

which affects the calculated loss

$L(w, b) > L(w + \eta \Delta w, b + \eta \Delta b)$

↓

moving to the 'θ' notation

$L(\theta) > L(\theta + \eta \theta)$ $\therefore$ $\theta = [w, b]$

↓ Taylor series for a vector

$$L(\theta + \eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L(\theta) u + \frac{\eta^3}{3!} * \ldots$$

$\Delta\theta = u$

change vector

↓

needs to be found in order to make loss less after the step.

↓

ignoring the higher order terms ($\eta \rightarrow$ very small)

↓

$$L(\theta + \eta u) \simeq L(\theta) + \eta * u^T \nabla_\theta L(\theta)$$

→ first order derivative of a vector

needs to be found in order to make this term -ve

$\nabla_\theta L(\theta) \rightarrow$ gradient of function which depends on θ vector wrt θ

↓

partial derivatives for all specific members.

↓

$$\begin{bmatrix} \partial f/\partial w \\ \partial f/\partial b \end{bmatrix}$$

↓

finally, we get vector of partial derivatives

↓

$L(\theta) \rightarrow$ Real number

$L(\theta + \eta u) \rightarrow$ Real number

$\eta \rightarrow$ scalar / real value corresponding to learning rate

↓

$$\left.\begin{array}{l} u^T : [\Delta w \quad \Delta b] \text{ (vector)} \\ \nabla_\theta L(\theta) : \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix} \end{array}\right\}$$ product will also be a real no.

↓

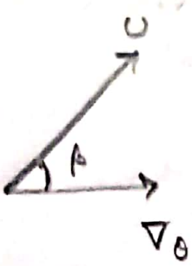
finding u^T such that the dot product becomes −ve.

↓ Now

$$L(\theta + \eta u) - L(\theta) = \eta * u^T \nabla_\theta L(\theta)$$

↓ Here

$$u : \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix}, \quad \nabla_\theta L(\theta) : \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$$



$$\cos\beta = \frac{\vec{u} \cdot \vec{\nabla}_\theta}{|u||\nabla_\theta|}$$

$\vec{u} \cdot \vec{\nabla}_\theta \rightarrow$ this needs to be less than 0. $\rightarrow$ cos to be less than 0

$|u||\nabla_\theta| \rightarrow$ always +ve

The value $u^T \cdot \nabla_\theta L(\theta)$ will be maximum when 180° i.e. when they are in opposite direction to each other, where $\cos\beta = -1$, and $\beta = 180°$

↓

Hence u_T should be chosen in such a way that it is in a direction opposite to that of the gradient.

↓

Gradient Descent Rule.

↓ formally

$\begin{bmatrix} w_{t+1} \\ b_{t+1} \end{bmatrix} \rightarrow \theta_{t+1}$

$\begin{bmatrix} \Delta w_t \\ \Delta b_t \end{bmatrix} \rightarrow \nabla_\theta L(\theta)$

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

$$b_{t+1} = b_t - \eta \Delta b_t$$

where $\Delta w_t = \dfrac{\partial L(w,b)}{\partial w}$ at $w = w_t$ and $b = b_t$

and $\Delta b_t = \dfrac{\partial L(w,b)}{\partial b}$ at $w = w_t$ and $b = b_t$

↓

the gradients can be computed using autograd in frameworks like pytorch or TensorFlow.

↓

The algorithm involves iterating over the dataset, calculating the predictions, corresponding loss, and ∴ loss gradient and then updating the parameters according to the update rule.

↓

Computing partial derivatives,

$$L = \frac{1}{n} \sum_{i=1}^{i=n} \frac{1}{2} (f(x_i) - y_i)^2$$

→ for convenience

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w} \left[\frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} (f(x_i) - y_i)^2 \right]$$

(sum rule)

$$\Delta w = \frac{\partial L}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w} \left(\frac{1}{2} (f(x_i) - y_i)^2\right)$$

↑ (sum of individual derivatives)

•) solving for 1 out of the n terms of the sum, and then generalising.

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

chain rule

$$= \frac{1}{2} * 2(f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)$$

independent of w

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

$$= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \quad \ldots (i)$$

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

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

chain rule

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

$$= f(x) \cdot (1 - f(x)) \cdot x$$

replacing this value in (i), we get

$$\nabla w = (f(x) - y)(f(x))(1 - f(x)) \cdot x$$

computed n no. of times, and summed over

true o/p for the data point

similarly for b $(x = 1)$

$$\nabla b = (f(x) - y)(f(x))(1 - f(x))$$

Scanned by CamScanner

•) generalising the gradient computation over > 2 parameters

↓

x_{ij} : notation to mark the ith data point and jth feature in that data point

↓

$$\therefore z = \sum_{j=1}^{m} w_j \cdot x_{ij} \quad , \quad m: \text{no. of features}, \rightarrow m \text{ parameters.}$$

↓

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

↓

computing dependency of the loss on a particular feature.

↓

$$\Delta w_j = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1-\hat{y}) * x_{ij}$$

*) Evaluation : metric to check the performance of the model.

↓

RMSE : root mean square error

(typically used for regression problems)

↓

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2}.$$

and accuracy can be done in case of classification tasks.