Introduction to deep learning

→ Each neuron in first layer of network represents a particular parameter
→ Every neuron of a inter-mediate layer is connected to each neuron of prev layer

Supervised learning

→ For image recognition usually convoluted neural networks are used
→ For sequenced data like audio which is sent over time as a 1D time series we use — RNN — Recurring neural network
→ For translation of texts — also have time component i.e are sequenced as characters so we use advanced RNN
→ Self driving — custom / hybrid neural networks

Structured data —

→ Databases of data
→ Each column will give u a parameter
Eg for house → size, bedroom etc upto price

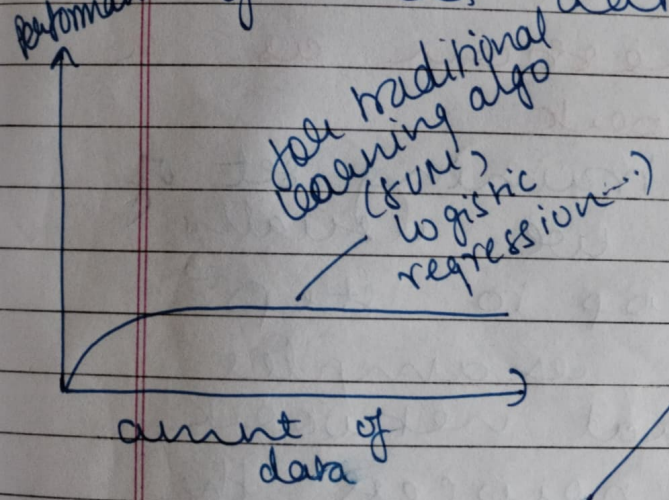
represented as an array in the computer

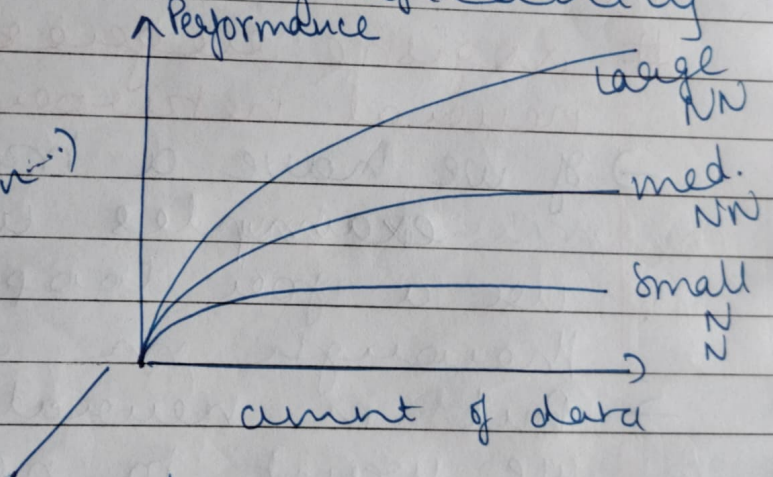
PAGE No.
DATE / /

→ the table will include initial inputs, desired output and all the middle parameters
→ All data has very well defined meaning

Unstructured data

→ Eg - raw audio, images, text
→ unstructured not well defined (Eg if u give image then data is pixel values)
→ More difficult to interpret unstructured data
→ Neural networks have improved interpretation of this data significantly





We need a bigger neural network and more data for better machine learning
→ Most reliable method to get better performance is to train a bigger network or throw more data at it

→ This only works upto a point as we run out of data or network is so big that it takes too long to train
→ All data used must be labelled
→ m :- no of training examples
→ If u dont have large training set how you train it determines performance
→ Algorithmic changes also affect deep learning progress (Eg: switching from sigmoid to RELU)

→ Idea → code → experiment

Logistic regression as a neural network

→ If we have a training set of m examples we usually use a for loop to step through m examples
→ But in neural network we want to process the training set by iterating without using an explicit for loop.
→ When we organize computation of neural network we have a forward propogation step followed

by a backward propagation step

Logistic regression

- form of binary classification
- for example - for a recognizing a cat image
  1 → cat or 0 → no cat
- In a computer to store images 3 separate matrices are stored corresponding to the red green and blue channels
- If input is 64 × 64 pixels then we have 3 64 × 64 matrices corresponding to red, green, blue pixel intensity values
- To turn pixel intensity values into a feature vector we unroll all the pixel values into an input feature vector x
- To unroll we define a feature vector x corresponding,

$$x = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ G_1 \\ G_2 \\ \vdots \\ B_1 \\ B_2 \end{bmatrix}$$

no of values = $64 \times 64 \times 3 = 12288$

$n = n_x = 12288$

→ In binary classification our goal is to learn a classifier that can input image represented by feature vector x and predict y either as 1 or 0

→ A training set with m examples is represented as $(x^{(1)}, y^{(1)})\ (x^{(2)}, y^{(2)}) \cdots (x^{(m)}, y^{(m)})$

→ $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ ← n_x

more compact notation for training set

$\leftarrow m \rightarrow$

$X \in \mathbb{R}^{(n_x) \times (m)}$

→ X.shape (n_x, m)

Command for shape of matrix in Python

output $Y = [y^{(1)}, y^{(2)} \cdots, y^{(m)}]$

$Y \in \mathbb{R}^{1 \times m}$

Y shape $= (1 \times m)$

→ Hence given X we need an algorithm that would give output $\hat{y} = P(y = 1 \text{ given } x)$ probability $(0 \le \hat{y} \le 1)$

$X \in \mathbb{R}^{n_x}$

x dimensional vector

PAGE No.
DATE / /

Parameters of logistic regression $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = w^T x + b$, — doesn't work

Linear func of input x

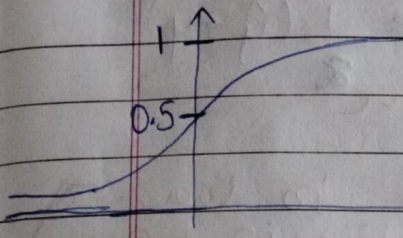
What I would use if doing linear regression

not a very good algorithm bcoz we want chance that $\hat{y}$ is equal to 1

$\hat{y}$ should be btw 0 and 1

$w^T x + b$ — can be >1 or -ve which doesn't make sense

So in logistic regression our output is $\hat{y} = \sigma(w^T x + b)$



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

If z large

$$\sigma(z) \approx \frac{1}{1+0} = 1$$

If z is very small or very large -ve no

$$\sigma(z) \approx \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{Bigno}} \approx 0$$

→ When we program neural networks we usually keep W & B seperate where B corresponds to an inter-spectrum

→ In some conventions we define extra feature $x_0 = 1$

$x \in \mathbb{R}^{n_x+1}$ $\hat{y} = \sigma(\theta^T x)$

PAGE NO.
DATE / /

Just an alternate way of representation

$\theta = \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ — θ_0 ← b, $\theta_1 \ldots \theta_{n_x}$ ← w

Easier to keep b & w as seperate.

Logistic regression cost function

→ $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$ where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

$z^{(i)} = w^T x^{(i)} + b$

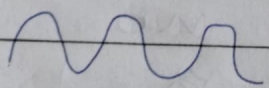
Given $\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$

we want $\hat{y}^{(i)} \approx y^{(i)}$ ← ideal

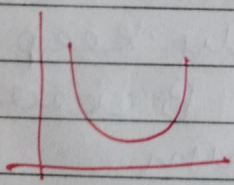
→ loss (error) function $= \frac{1}{2}(\hat{y} - y)^2$ ← in logistic regression this is not done

takes adient scent t ork ell

bcoz when we come to learn parameters the optimization problem becomes non convex.



← we will end up with multiple local minima so gradient dissent may not find a global optimum



← what we shld use

$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$

we need $-\log \hat{y}$ small as possible

$\therefore \log \hat{y}$ large as possible

$\therefore \hat{y}$ shld be large

If $y = 0$ $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$

shld be as small as possible

$\therefore 1 - \hat{y}$ shld be large

$\therefore$ we want $\hat{y}$ as small as possible.

Cost function = $J(w, b)$

measures how well we do over entire training set

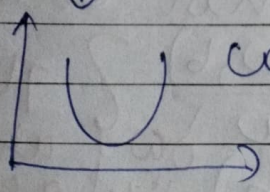
$$= \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

→ So in training we are going to try to find parameters w & b that minimize overall cost func J

Gradient descent

→ we need w, b that minimize $J(w, b)$



→ To find a good value of w, b we first initialize them to some initial values
→ Gradient descent takes a step in steepest downward descent again and again until we converge or get close to global maximum
→ We keep updating w

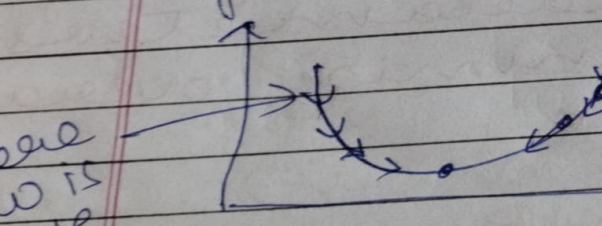
$$w = w - \alpha \frac{dJ(w)}{dw}$$

α / learning rate that controls how big step we take in each iteration to make to para

$$w = w - \alpha dw$$

dw represents $\frac{dJ(w)}{dw}$ in code

⇒ if dw is −ve we take ↓ w
if dw is +ve we ↑ w



$$\left(dw = \frac{dJ(w)}{dw} \right)$$

Since $J(w, b)$ (w not only parameter)

$$w = w - \alpha \frac{\partial J(w,b)}{\partial w} \quad \text{dw in code}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b} \quad \text{db in code}$$

These are actual updates

Derivatives

If $\frac{df(a)}{da} = x$

* If a is increased by Δa then $f(a)$ res by $x \Delta a$

Computation graph

→ Computations of a neural network are organized in terms of a forward pass or forward propogation step followed by a in which we compute o/p of neural network followed by a backward pass or back propogation step which we use to compute gradients or derivatives

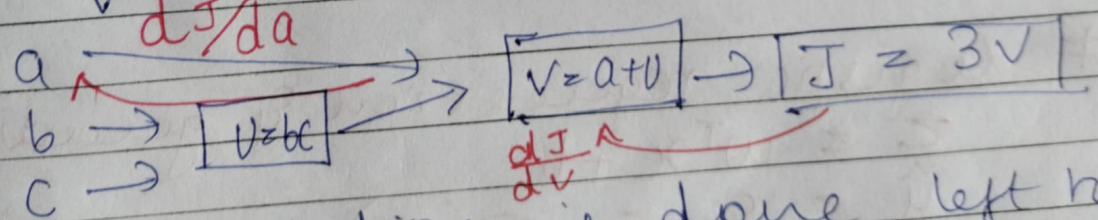
→ computation graph explains why its ordered this way

→ Suppose we hv a func

$J(a, b, c) = 3(a + bc)$

$bc = U$

$V = a + U \qquad J = 3V$



→ computation is done left to right for function J

→ for derivatives computation is from right to left

Computing Derivatives

→ $\frac{dJ}{dv} = \frac{\Delta J}{\Delta v} = 3$

→ $\frac{dJ}{da} = \frac{\Delta J}{\Delta a} = 3 = \frac{dJ}{dv} \times \frac{dv}{da} = (3 \times 1)$ (chain rule)

→ $\frac{dv}{da} = \frac{\Delta v}{\Delta a} = 1$

→ When writing codes to implement backpropagation we will find some final output variable that we really care about (Eg:- J) → J is really the last node of the computational graph.

→ We will be calculating $\frac{d \text{ final O/P var}}{d \text{ var}}$

In code we represent by (dvar) (var = an intermediate variable)

→ $\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{db} = 3 \times c = 3c$

Logistic regression & gradient descent

$z = w^T x + b$

$\hat{y} = a = \sigma(z)$

$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$

$x_1 \rightarrow$
$w_1 \rightarrow$
$x_2 \rightarrow$ $z = w_1 x_1 + w_2 x_2 + b$ $\rightarrow$
$w_2 \rightarrow$ "dz" $= \frac{dL}{dz} = da \frac{(a,y)}{dz}$ $\leftarrow$
$b \rightarrow$ ↓ backcode $dz = a - y$

$\hat{y} = a = \sigma(z) \rightarrow \mathcal{L}(a,y)$ $= \frac{dL}{da} \times \frac{da}{dz}$

"da" $= \frac{d\mathcal{L}(a,y)}{da}$ $\leftarrow$
↑ backcode $= \frac{-y}{a} + \frac{1-y}{1-a}$

$dw_1 = x_1 dz \quad dw_2 = x_2 dz$

$db = dz$

$w_1 = w_1 - \alpha dw_1$
$w_2 = w_2 - \alpha dw_2$
$b = b - \alpha db$

Gradient descent for m examples

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$$\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{w_1} \mathcal{L}(a^i, y^i)$$

$\frac{\partial}{w_1} \mathcal{L}(a^i, y^i) = dw_1^{(i)}$

Lets initialize $J = 0, dw_1 = 0, dw_2 = 0, db = 0$

→ For i = 1 to m (For loop)

$z^{(i)} = w^T x^{(i)} + b$
$a^{(i)} = \sigma(z^{(i)})$

$J += -\left[y^{(i)} \log(a^i) + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$

PAGE No.
DATE / /

$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$
$$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$
$$db \mathrel{+}= dz^{(i)}$$

$n = 2$ (only 2 features ie 2 neurons in 1st layer)

we have to divide by m since avg of m examples

$$J /= m, \quad dw_1 /= m; \quad dw_2 /= m$$
$$db /= m$$

$$w_1 = w_1 - \alpha\, dw_1$$
$$w_2 = w_2 - \alpha\, dw_2$$
$$b = b - \alpha\, db$$

→ Vectorization will help us do this without for loops