Week 9-10 (7-1-1)—(7-2-3)

PAGE No. | DATE / /

(7-1-1) Introduction to Neural Networks (12:21)

- Objectives
  - Adding layers ↑ modelling power
  - Mathematical expression – Backpropogation algorithm
  - List ways to improve backprop.

Increasing non linearity in models:

1. Linear models (Regressor / Classifier) $L(\sigma(Wx))$
2. Support vector machines → Fixed features $L(\sum_i t_i a_i K(x, x_i))$
3. Neural Networks → Trainable features $L(\sigma(W_2\, \sigma(W_1 x)))$
   [or multiple layers (But not too deep)]
4. Deep Neural Networks $L(\sigma(W_n \ldots (\sigma(W_1 x))))$
   [σ is a non linear function, not necessarily sigmoid]

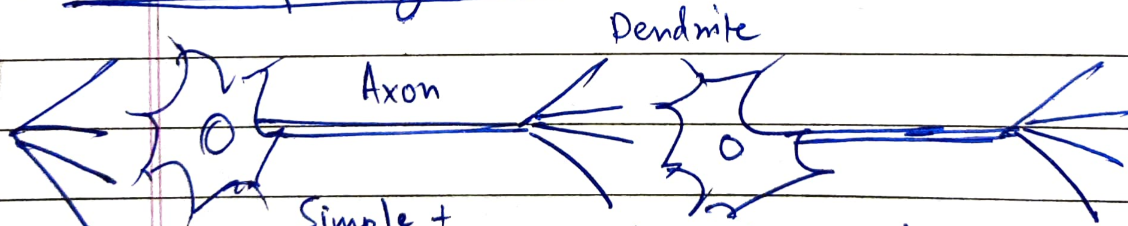
(7-1-2) layered functional representation of NN:

W are matrices for neural networks

Transform feature vectors into other vectors.

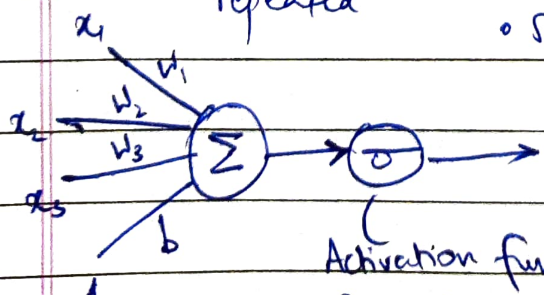
Update based on $\partial L / \partial W_{Lij}$ (chain rule of derivatives)

Structure of Biological Neuron:



Simple + repeated

Knowledge resides in
- Structure of connect^n & Strength of conn^n



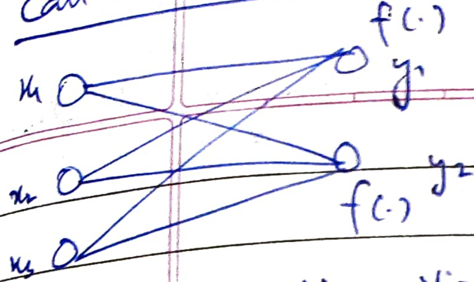
Simplification of Biological Neuron.

Activation function

$W_2(W_1 x) = (W_2 W_1)x = W_3 x$

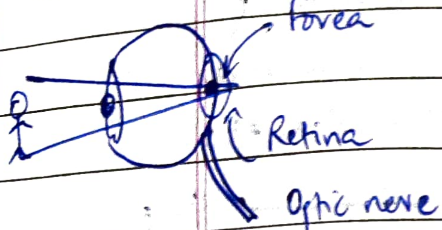
Which is why every layer but the last has some non linear function 'σ'

Can have multiple outputs as well:



$$W \Rightarrow f\left(\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

(7-1-3) Mammalian Visual Cortex:

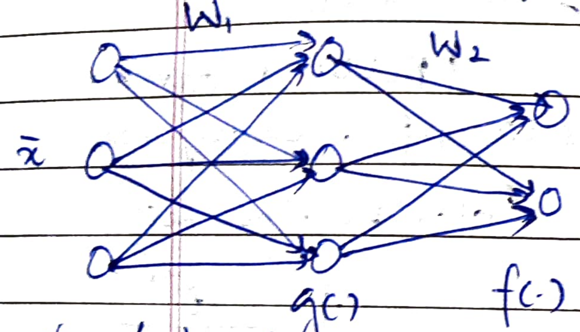


Specialised sensory cells → Optic nerve → V_1 ("shapes") → V_2 (More complex features) → . . .

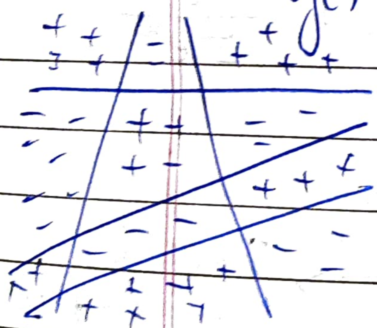
So on.

[ONE DIR^N FLOW]

(7-1-4) Introducing Hidden layer in NN:



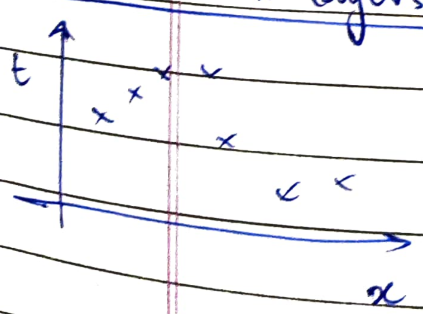
$$\bar{y} = f(W_2(g(W_1 x)))$$



Each linear boundary is one neuron

Hence can do non-linear accurate classification

What hidden layers are doing:-



Basic $f(x) = c$

Lin. $f(x) = W_1 x + W_0$

Non lin. : $f(x) = W_1 \phi_1 x + W_2 \phi_2 x + b$

SVM $f(x) = \sum W_i K(x, x_i)$

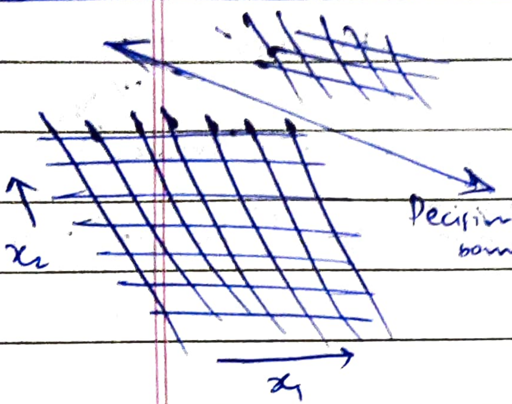
NN func = Summation of activation function with weights & biases.

Repeated combination of canonical function with W & b & layers.

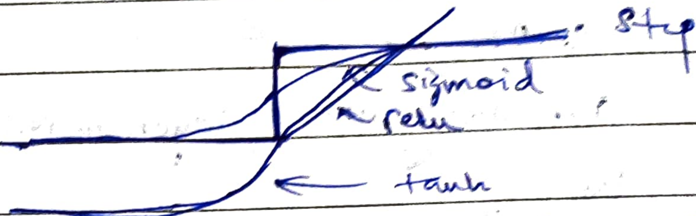
Universal Approximation Theorem:

- $f \in C(W_2\, \sigma(W_1 x))$ can approximate with error ϵ in a compact interval any smooth $f(x)$
  - Provided size of W is arbitrary
  - σ is also smooth, but NOT a polynomial

(7-1-5) Activation function:



We need a smoother version of a step function with non zero gradients $\Rightarrow$ sigmoid etc.



Step: Classification only. Not used anymore	σ & tanh: trainable appr. of step	ReLU: Preferred as fast convergence

Softmax: Generalized σ for multiclass classification net

$\frac{\mathrm{sgn}(x)+1}{2}$	$\frac{1}{1+e^{-x}}$	$\tanh(x)$	$\max(0, x)$	$\frac{e^{x_i}}{\Sigma e^{x_i}}$

Problems with too many layers:
- Too many parameters
- Gradient dilution

Hyperparameter: No. of hidden layers. → Input & output dim fixed
Dimension of layers.

The training process is gradient descent

Gradient of vectors: ∇f $\quad \nabla = \partial L / \partial W_{ij}$ matrix

(7-2-1) chain rule & backpropagation:

$f(x) = g(h(x))$

$f'(x) = g'(y)\, h'(x) \qquad y = h(x).$ so on.

eg $\frac{d(\sin(x^2))}{dx} = \cos(x^2)\, 2x$

Algorithm:

- Make a forward pass & store partial derivatives
- During backward pass, multiply ∂ [Library handles this]

(7-2-2) Backpropogation for vector variables :-

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, x_3) \\ f_2(x_1, x_2, x_3) \end{bmatrix} \quad \xrightarrow{x_j}$$

$$J(f) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{bmatrix} = f_i \downarrow \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \end{bmatrix}$$

$$J(f) = \begin{bmatrix} \partial f_i/\partial x_j \end{bmatrix}$$

i - row index
j - col index

$[f_i \times x_j]$
$|I| \times |J|$

Questions:

- Scale all weights & biases by factor? $\Longrightarrow$ Sign no change
  NOT UNIQUE !! $\leftarrow$ Need to regularize
- Gradients in deep neural networks?

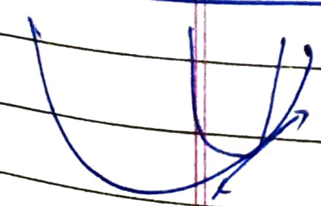
$\prod W_i$ - if $|W|<1$ then we have vanishing ∇
- if $|W|>1$ then we have exploding ∇ (depends on initializ^n)

Vanilla gradient descent :

- Iteration loop n
  - Sample loop i
    - $\nabla_\theta L \leftarrow \nabla L_i + \nabla L_i$
  - $\theta \leftarrow \theta - \eta \nabla_\theta L$

Slow model: We have to go through all training samples before we can update even once

(7-2-3) Role of step size & learning rate :-



Same gradient $\Longrightarrow$ Same step for same η

- Same value + Same gradient BUT:
- Different Hessian • Different step sizes needed

$\Longrightarrow$ We need learning rate scheduling or some function to model η

PAGE No. | DATE / /

Issues with gradient descent:

- Need to find good step size η
- ~~lots of computation before each update~~
- Can get stuck in local minima

Solutions: Stochastic GD or Batch GD

1.) Stochastic Gradient Descent:

- Pick any training point randomly

Iteration loop

- Select point
- l_i (compute)
- $\nabla_\theta l_i$
- $\theta \leftarrow \theta - \eta \nabla_\theta l_i$

Need more updates but updates much quicker
$\Rightarrow$ Noisy updates are possible
But unlikely to settle in local minima

2.) Batch gradient descent: (GPU utili zn)

[Hybrid of Vanilla GD & Stochastic GD] Most popular.

- Batch formation loop
  - shuffle training pt
  - Divide into batches
- Epoch Loop
  - Batch loop
    - L_{Batch}
    - $\nabla_\theta L_{Batch_j}$
  - $\theta \leftarrow \theta - \eta \nabla L_{\theta\,batch_j}$

Keep increasing batch till GPU error.
Faster than stocha.

Double derivative speed up of Hessian & Jacobian:

Let $f(x) = ax^2 + bx + c$

Assuming $a<0$ [Inverse paraboloid]

Minima at $-b/2a$

PAGE No.
DATE / /

For any 'x' the perfect step :-

$$\left(-\frac{b}{2a} - x\right) = -\left(\frac{2ax+b}{2a}\right) = -\frac{f'(x)}{f''(x)}$$

step

$$\implies \eta = \frac{1}{f''(x)}$$

Same logic for:

$$x \leftarrow x - H(f(x))^{-1} \nabla(f(x))$$

[Not used but approximations of these speed up]

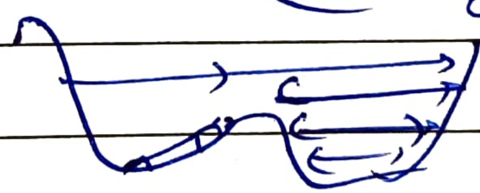
→ Note that it is NOT important to find global minima.
(Since Global Min of training data ≠ That of testing data)

Adding momentum in-stead of 2nd derivative :-

$$\theta^{n} \leftarrow \theta^{n-1} - \eta \nabla_{\theta} L + \alpha\left(\theta^{n-1} - \theta^{n-2}\right) \qquad \alpha =$$

0 & 1 (0.8 & 0.9)

speed up



← slow down

Like momentum intuition

[L2 - Weight decay] in NN