Artificial Neural Networks

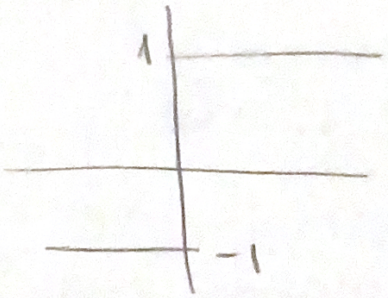
Perceptrons: Takes inputs, calculates a linear combination, prompts 1 if result > threshold & -1 if otherwise

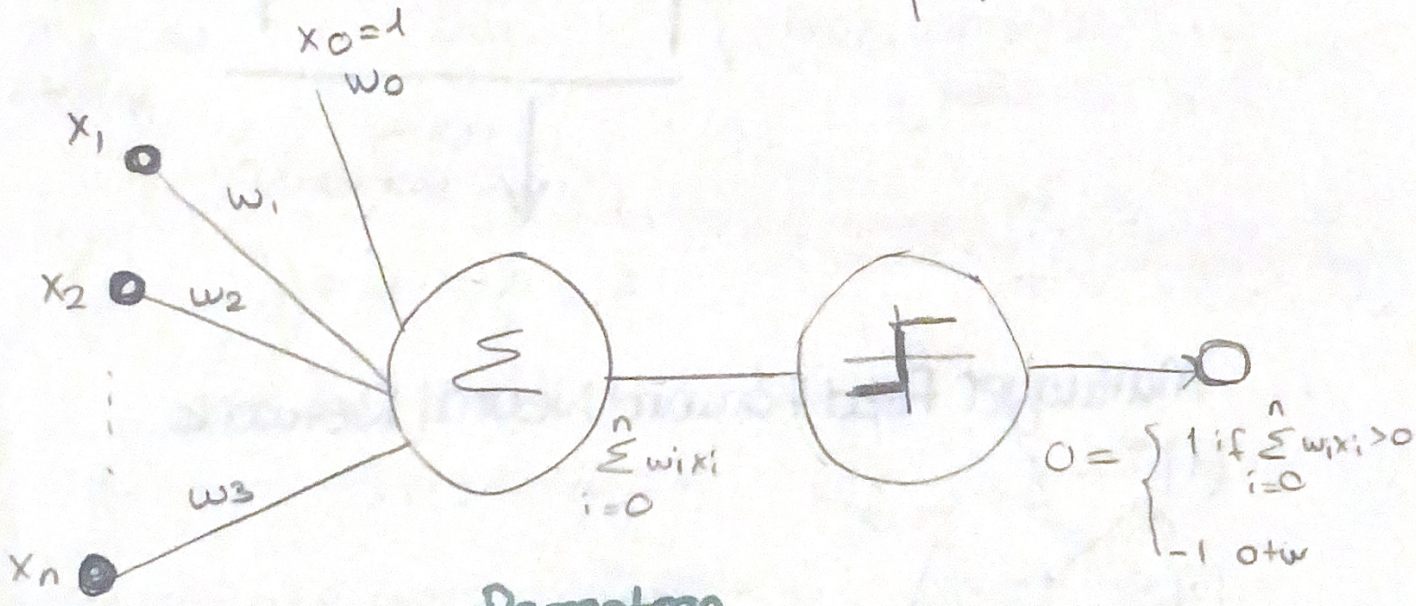
$$o(\underbrace{x_1, \dots, x_n}_{\text{inputs}}) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

output

↳ signum function (sgn)

$-w_0 \rightarrow$ threshold





Perceptron

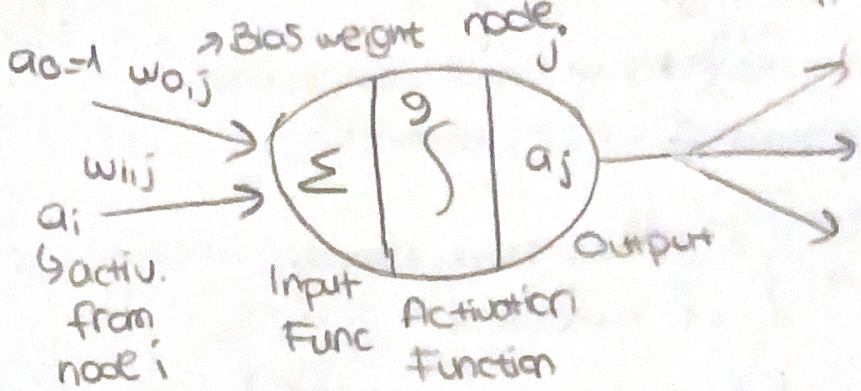
Weights for AND → $w_0 = -0.8$ $w_1 = w_2 = 0.5$

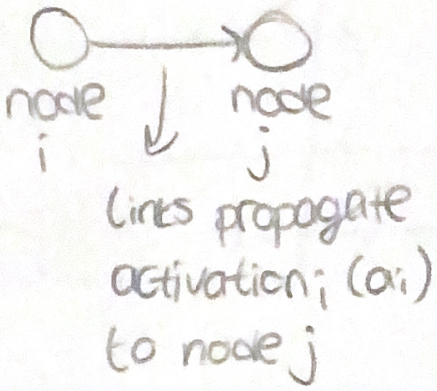
x_1	x_2	res	
0	1	0.5	→ < 0.8 = 0
1	1	1	→ 1
0	0	0	→ < 0.8 = 0

OR → $w_0 = 0.3$ $w_1 = w_2 = 0.5$

x_1	x_2	res	
0	1	0.5	→ ≥ 0.3 = 1
1	1	1	

Neural Networks





w_{ij} : weight of link i,j

a_0 : dummy (bias)

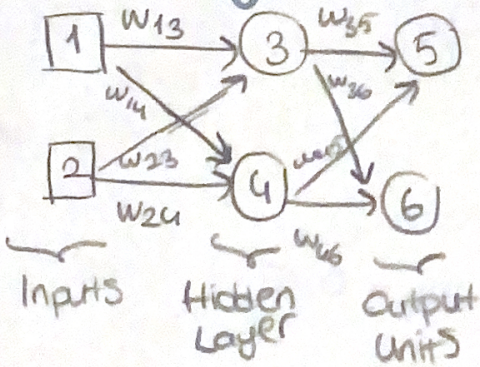
$w_{0,j}$: weight of bias

$$in_j = \sum_{i=0}^{n} w_{i,j}\, a_i$$ → weighted sum of unit j's inputs

↓ apply activation

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j}\, a_i\right)$$

Multilayer Feed Forward Neural Network



$$a_5 = g(w_{0,5} + w_{3,5}\, a_3 + w_{4,5}\, a_4)$$

$$= g(w_{0,5} + w_{3,5}\, (g(w_{1,3}\, a_1 + w_{2,3}\, a_2 + w_{0,3})))$$

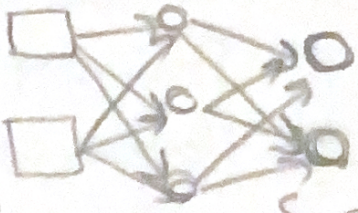
$$w_{4,5}\, (g(w_{1,4}\, a_1 + w_{2,4}\, a_2 + w_{0,4}))$$

↓ replace with inputs x_1, x_2

Backpropagation

$$\frac{\partial}{\partial w} Loss(w) = \frac{\partial}{\partial w} |y - h_w(x)|^2 = \frac{\partial}{\partial w} \sum (y_k - a_k)^2 =$$

$$\sum_k \frac{\partial}{\partial w} (y_k - a_k)^2 \rightarrow$$



- Loss calculation of perceptron
- target
- activation in output node k
- output nodes (k)

- In multilayer networks we take derivative of overall error gradient.

  With multiple output units (k):

  Err_k : k^{th} component of error vector $y - h_w$

  $\Delta_k = Err_k \times g'(in_k)$ } modified error

  $w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k$ } weight update rule

- To update connections between input units & hidden units we need to define a quantity analogous to the error term for output nodes. (where we need error backprop)
- Hidden node j is responsible for some fraction of Δ_k
- **Δ_k values are divided according to strength of connection between hidden node & output node.**

  They are propagated back to provide Δ_j for hidden layer.

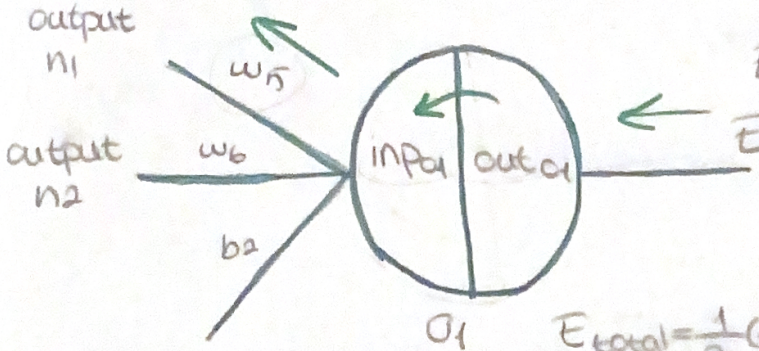
→ Propagation rules for Δ values : $\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$

- Backprop in a nutshell

1. Compute Δ values for output units, use observed error
2. For each layer until the earliest hidden layer:
   Propagate Δ values back to previous layer
   Update weights between two layers

Backprop + Chain Rule Visual

$$\frac{\partial inp_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial inp_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

1

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * (-1) + 0$$

$$= -(target_{o1} - out_{o1}) = 0.741$$

2

$$out_{o1} = \frac{1}{1 + e^{-inp_{o1}}} \rightarrow \frac{\partial out_{o1}}{\partial inp_{o1}} = out_{o1}(1 - out_{o1}) = 0.186$$

3 $inp_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$

$$\frac{\partial inp_{o1}}{\partial w_5} = out_{h1} = 0.593$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial inp_{o1}} * \frac{\partial inp_{o1}}{\partial w_5} = 0.741 \times 0.18 \times 0.593$$

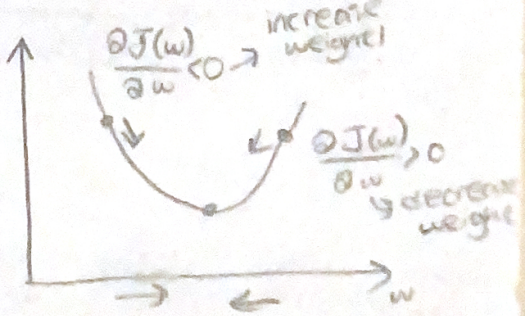
$$= 0.082$$

Delta Rule

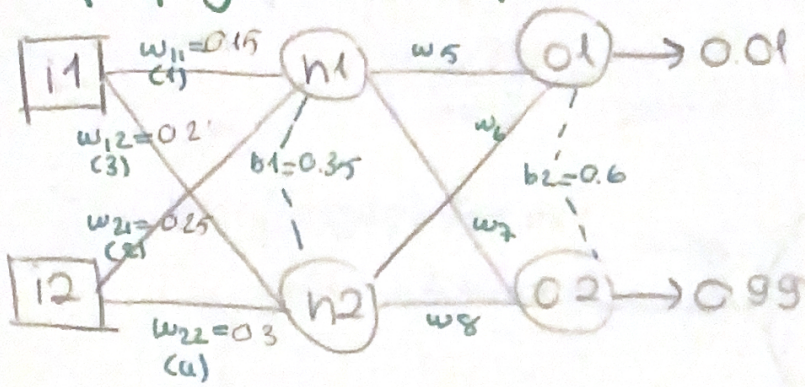
$$\frac{\partial E_{total}}{\partial w_5} = -\underbrace{(target_{o1} - out_{o1})}_{\frac{\partial E_{total}}{\partial out_{o1}}} * \underbrace{out_{o1}(1 - out_{o1})}_{\frac{\partial out_{o1}}{\partial inp_{o1}}} * \underbrace{out_{h1}}_{\frac{\partial inp_{o1}}{\partial w_5}}$$

Then update $w := w - \alpha \frac{\partial J(w,b)}{\partial w}$

$$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$$



Backpropagation Example



$i_1 = 0.05$

$i_2 = 0.10$

$out_{o1} = g(input_{o1})$

Forward Pass

→ Input of h1: $w_1 \cdot i_1 + w_2 \cdot i_2 + b_1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 = 0.3775$

← Output of h1: $\frac{1}{1+e^{-0.3775}} = 0.593$

← Output of h2: 0.596

→ Input of o1 : $w6 \times 0.596 + w5 \times 0.593 + 0.6 = 1.105$

← Output of o1 = $\frac{1}{1+e^{-1.105}} = \boxed{0.751}$

← Output of o2 = $\boxed{0.772}$

Error Calculation

$$E_{total} = \sum \frac{1}{2} (target - output)^2$$

$$= \frac{1}{2}\left((0.01 - 0.751)^2 + (0.1 - 0.772)^2\right) = 0.298$$

↙ Total Error

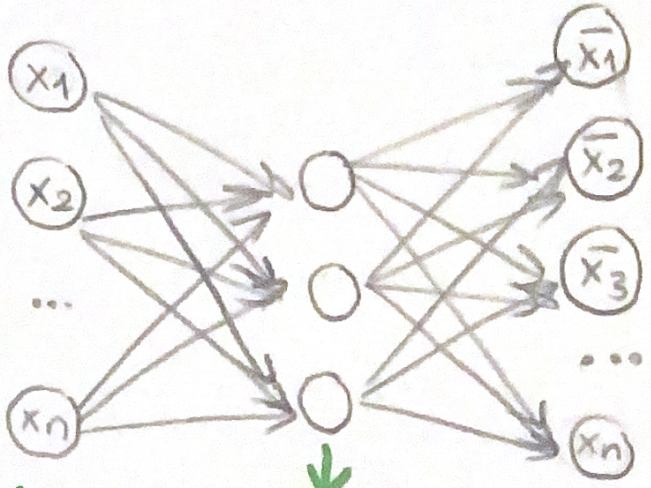
Backward Pass

We want to know how much change in w5 affects total error $\left(\frac{\partial E_{total}}{\partial w_5}\right)$

Chain Rule

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial input_{o1}} * \frac{\partial input_{o1}}{\partial w_5}$$

Autoencoders



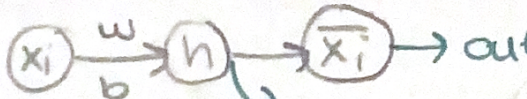
input layer → h (latent variables) → output layer

encoder | decoder

Idea: Autoencoders are trained to reconstruct input.

- When hidden layer is smaller than input & output layers, the model only learns the most salient features
  This is called undercomplete autoencoder.
- Autoencoders learn unsupervised.
- $h_{w,b}(x) \approx x$
  ↓
  identity function (we're trying to learn this)

LOSS → $(\bar{x}_i - x_i)$



output $= \bar{x} = s(w' y + b')$

output $= y = s(wx + b)$

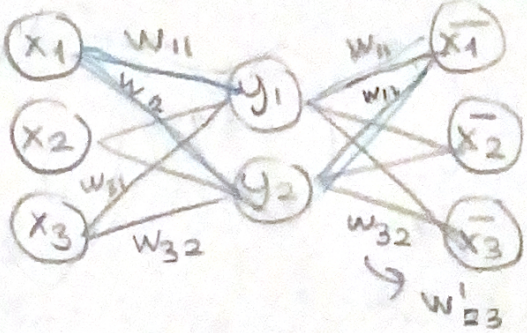
$L(x - l)_{n \times 1}$

weights and biases of encoder

these are weights and biases of decoder

$W' = W^T$

one is transpose of another!



$$W = \begin{bmatrix} x_1 & x_2 & x_3 \\ w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}$$

$W' = W^T \rightarrow$

$$W' = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

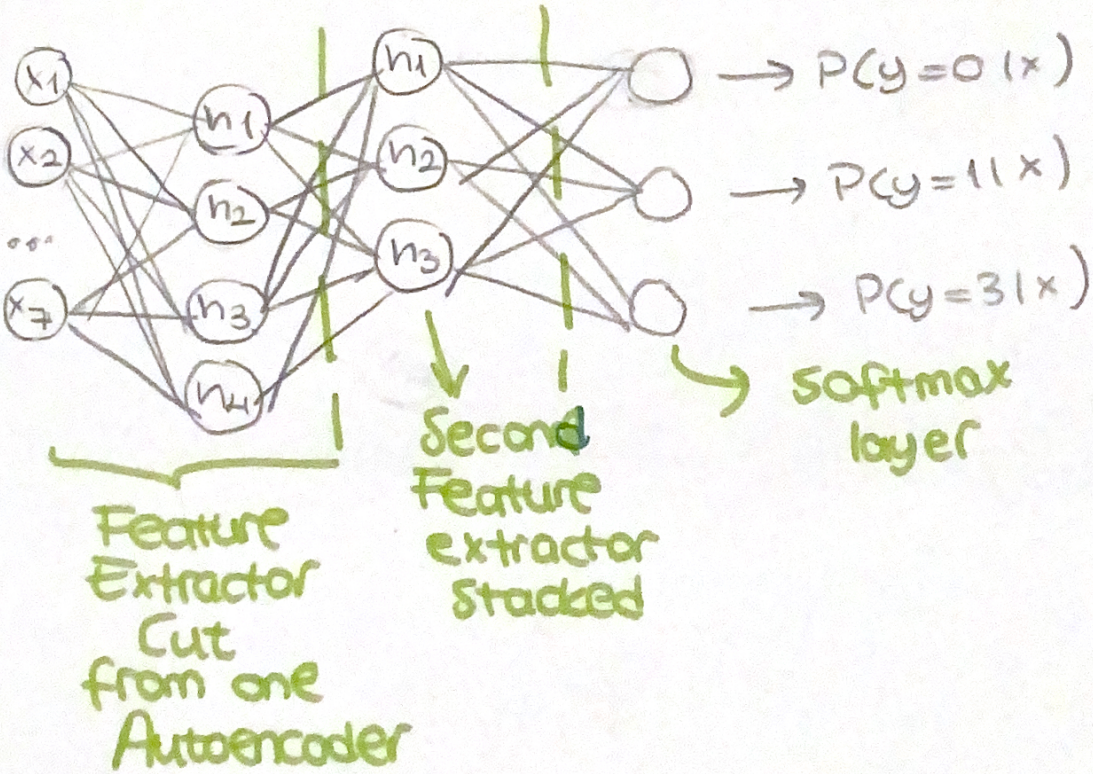
$L(x, \bar{x}) = \|x - \bar{x}\|^2$

Autoencoders (contd.)

Stacked Autoencoder

1. Train autoencoders to be feature extractor,
2. Add couple of more layers (optional)
3. Add a classifier layer on top → Learning becomes supervised
4. Train with domain specific data

→ This is used to avoid vanishing gradients.



Denoising Autoencoder

Changing the reconstruction error term.

decoder output

$g(f(x))$ ← hidden layer output

Traditional reconstruction loss → $L(x, \hat{x})$

Denoiser autoencoder loss → $L(x, g(f(\tilde{x})))$

$\tilde{x}$ ↓ corrupted input

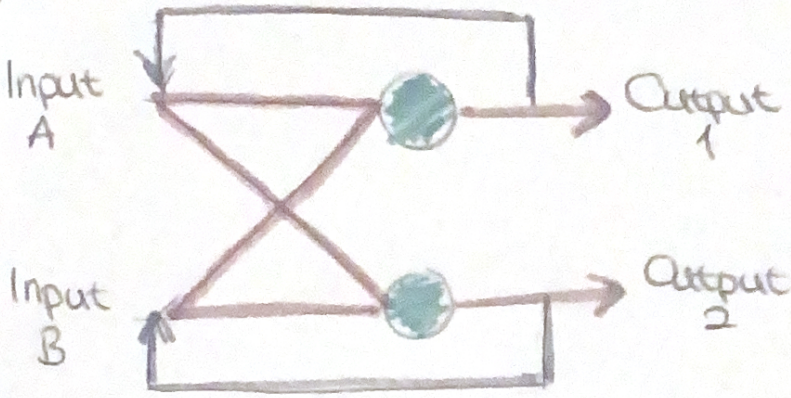
It receives a corrupted datapoint as input and tries to predict uncorrupted original data point.

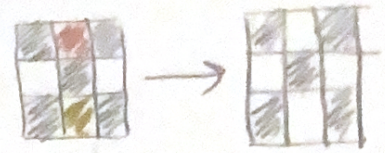
$C(\tilde{x}|x)$ → conditional distr over corrupted samples ($\tilde{x}$)

$P_{reconstruct}(x|\tilde{x})$ → reconstruction distribution estimated from training pairs.

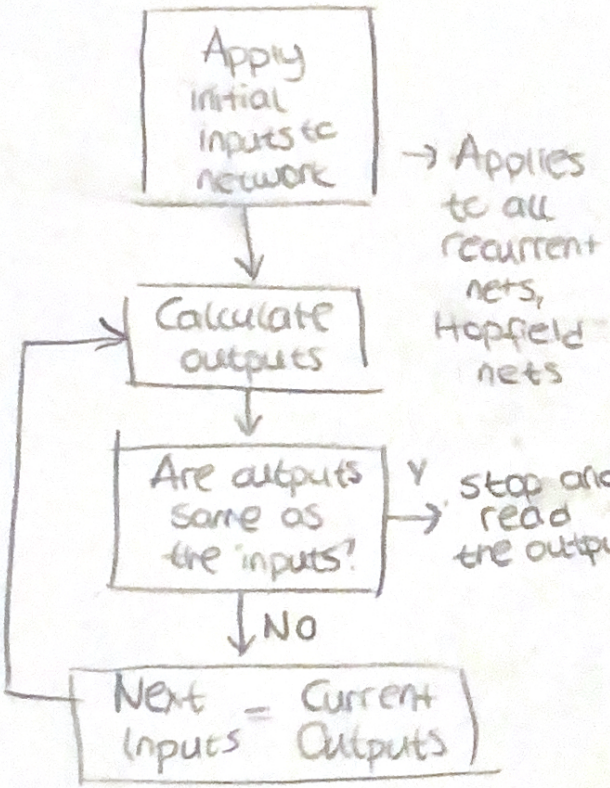
$P_{reconstr}(x|\tilde{x}) = \underbrace{P_{decoder}(x|h)}_{g(h)}$ → output of encoder $f(\tilde{x})$

Recurrent Neural Network





Network keeps a memory!



Hopfield Networks

- Weight matrix is symmetrical

$$w_{ij} = w_{ji}$$

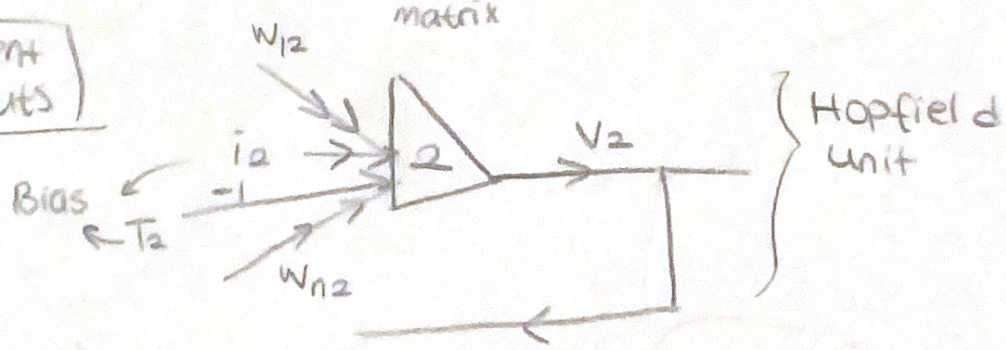
- Output $v_i = -1$ if $net_i < 0$
  $v_i = 1$ if $net_i \geq 0$

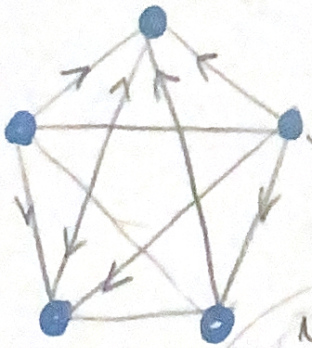
(net_i: total input coming to neuron)

Activation function used is the sign function

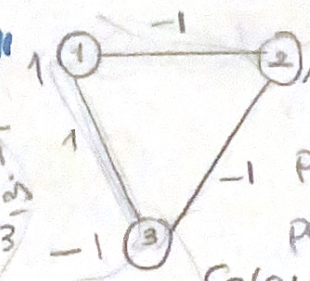
$$net_i = w_i \cdot v + i_i - T_i$$

(w_i: weight matrix; v: output matrix; T_i: Bias inputs)





ex:



$w_{21} = w_{12} = -1$
$w_{31} = w_{13} = 1$
$w_{23} = w_{32} = -1$

Provide a pattern → $V_0 = [1, 1, -1]$

$$w_i = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & -1 & 1 \\ -1 & 0 & -1 \\ 1 & -1 & 0 \end{bmatrix} \end{array} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

Calculate net_1

$net_1 = 1 * (-1) + (-1) * 1 = -2$

$net_2 = 1 * (-1) + (-1)(-1) = 0$

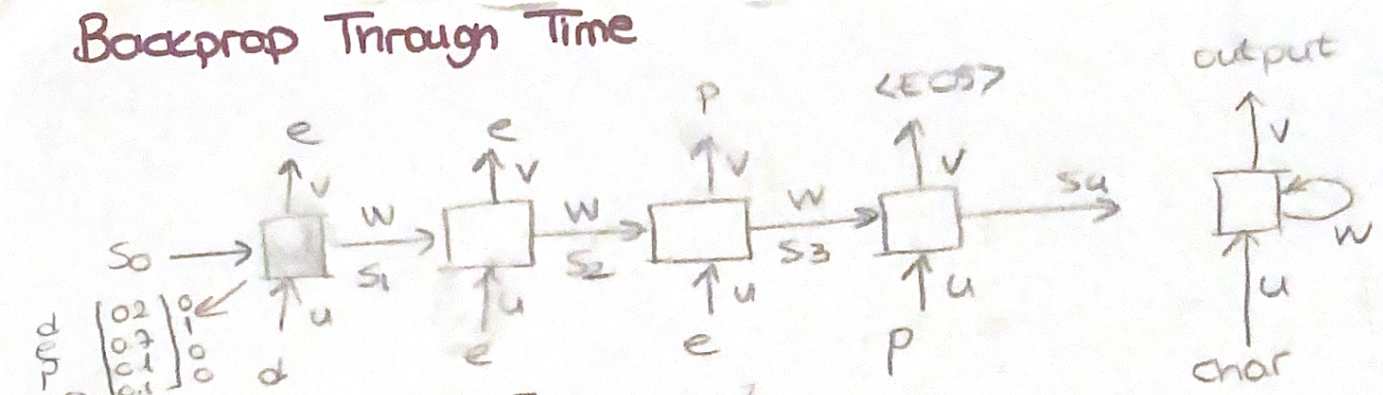
$net_3 = 1 * 1 + 1 * (-1) = 0$

$V_1 = [-1, 1, 1]$

HW

$w_{12} \cdot V_{12} + w_{13} \cdot V_{13}$

Backprop Through Time



Total Loss $\rightarrow \sum_{t=1}^{T} \mathcal{L}_t(\theta)$ } Sum of loss over all steps

$\mathcal{L}_t(\theta) = -\log(y_{tc})$

↳ prob produced for true class at time step t (0.7 in above example, log will decrease if it gets closer to 1)

For backprop we compute gradients for W, U, V

For V: $\frac{\partial \mathcal{L}(\theta)}{\partial V} = \frac{\partial \mathcal{L}_1(\theta)}{\partial V} + \frac{\partial \mathcal{L}_2(\theta)}{\partial V} + \cdots$ } standard backprop

↙ this is not being affected by states

↙ V is a matrix

s_4 depends on s_3 & w
s_3 depends on s_2 & w
...

For W: $\frac{\partial \mathcal{L}(\theta)}{\partial W}$

$\frac{\partial \mathcal{L}_4(\theta)}{\partial W} = \frac{\partial \mathcal{L}_4(\theta)}{\partial s_4} \cdot \frac{\partial s_4}{\partial W}$ → this is calculated with regular BP

we can't calculate this
$s_4 = G(W s_3 + b)$

Total derivative has 2 parts.

Explicit → $\frac{\partial^+ s_4}{\partial W}$ } treats all inputs as constant

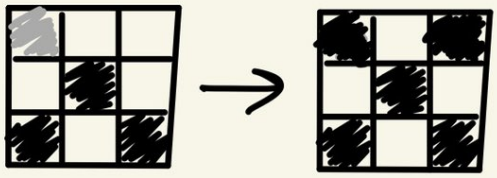
Implicit → Sum all indirect paths from s_4 to W.

$\frac{\partial s_4}{\partial W} = \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3} \cdot \frac{\partial s_3}{\partial W} = \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\left[\underbrace{\frac{\partial^+ s_3}{\partial W}}_{\text{exp}} + \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W}\right]$

...

$\frac{\partial^+ s_2}{\partial s_1} \cdot \frac{\partial s_1}{\partial W}$

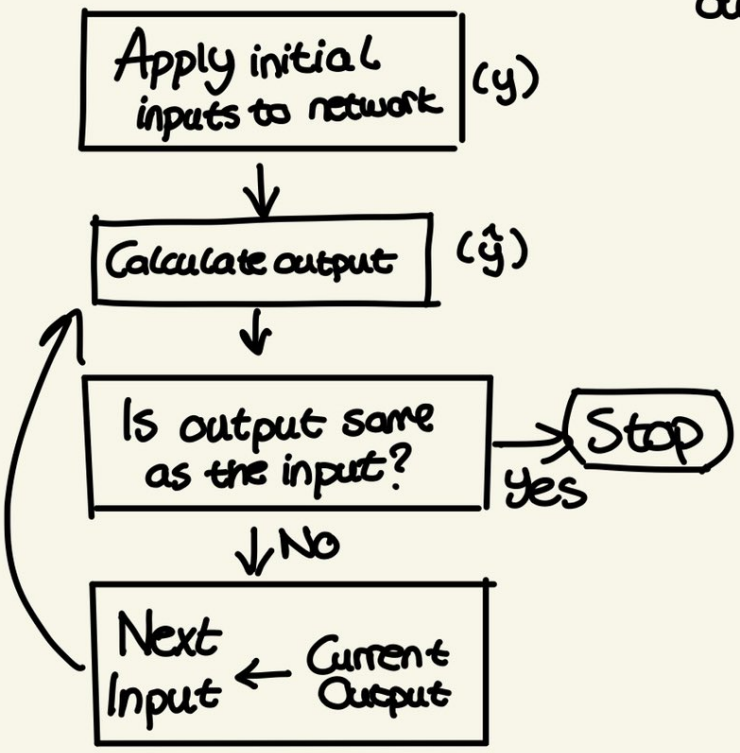
Hopfield Networks



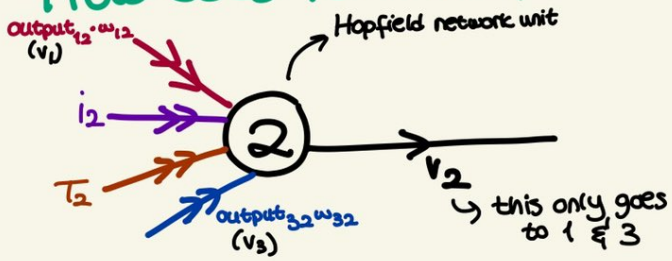
↳ input pattern

↳ converges to this output

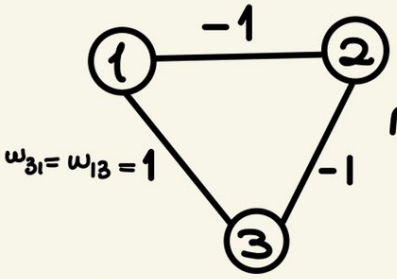
Network keeps a memory!



How does it work?



Example Hopfield Network



$V_0 = [1, 1, -1]$

$net_1 = w_{12} \cdot V_2 + w_{13} \cdot V_3$

Calculate net_i:

$net_1 = 1 \cdot (-1) + (-1) \cdot 1 = -2 \rightarrow -1$

(V_1 w_{12} V_3 w_{13})

$net_2 = 1 \cdot (-1) + (-1) \cdot (-1) = 0 \rightarrow 1$

$net_3 = 1 \cdot 1 + 1 \cdot (-1) = 0 \rightarrow 1$

$V_1 = [-1, 1, 1]$

Properties:

- Weight matrix is symmetrical. ($w_{ij} = w_{ji}$)
- Activation function is sign function.

Output $v_i = -1$ if $net_i < 0$

$v_i = 1$ if $net_i \geq 0$

↳ total input coming to neuron

ConvNets

→ Used for problems that have grid-like topology.

* 1D → Time series
* 2D → Image recognition

Convolution

$$s(t) = (x * w)(t) \quad \left\{ \text{feature map} \right.$$

input, kernel

$$\int x(a)\, w(t-a)\, da$$

$$2D \rightarrow S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)\, K(m,n)$$

K → Kernel, I → Input

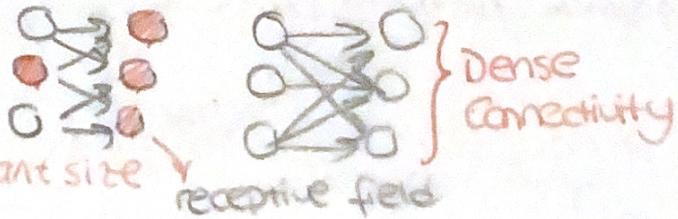
Why?

→ Sparse Interactions: Features can be detected with less params

→ Parameter Sharing

→ Equivariant Representations

→ We can work with inputs of variant size



- Sparsity: We store less info thanks to kernels.

m input, n output } m×n params, O(m×n)

k → number of connections each input can have

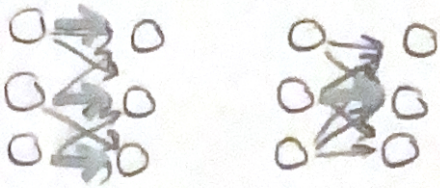
O(k×n) → sparsely connected approach

- Parameter Sharing: Using same parameter for more than one function in a model.

ex// Dense Nets → Each element of weight matrix is multiplied for once with input

CNNs → Has tied weights, value of the weight applied to one input is tied to another value elsewhere. Each element in kernel is used at every position of the input. (Except for boundary pixels)

Parameter Sharing



CNN DNN

→ Connections that use a particular parameter.
(Reduces storage requirements)

Equivariance

An equivariant function means if the input changes, output changes in the same way.

$$f \longleftrightarrow g \text{ (equivariant)} \quad \text{if} \quad f(g(x)) = g(f(x))$$

- For convolution → If we move object in the input, it's representation will move the same amount in the input. Thus we can create 2D map of where certain features appear.

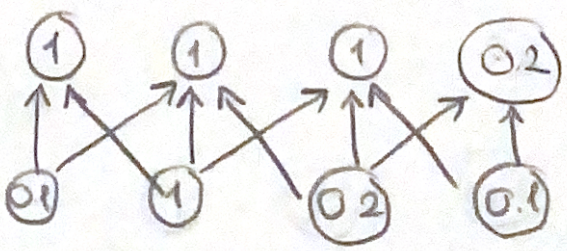
Pooling

→ Pooling replaces outputs of the net at a certain location with a summary statistic of the nearby outputs.

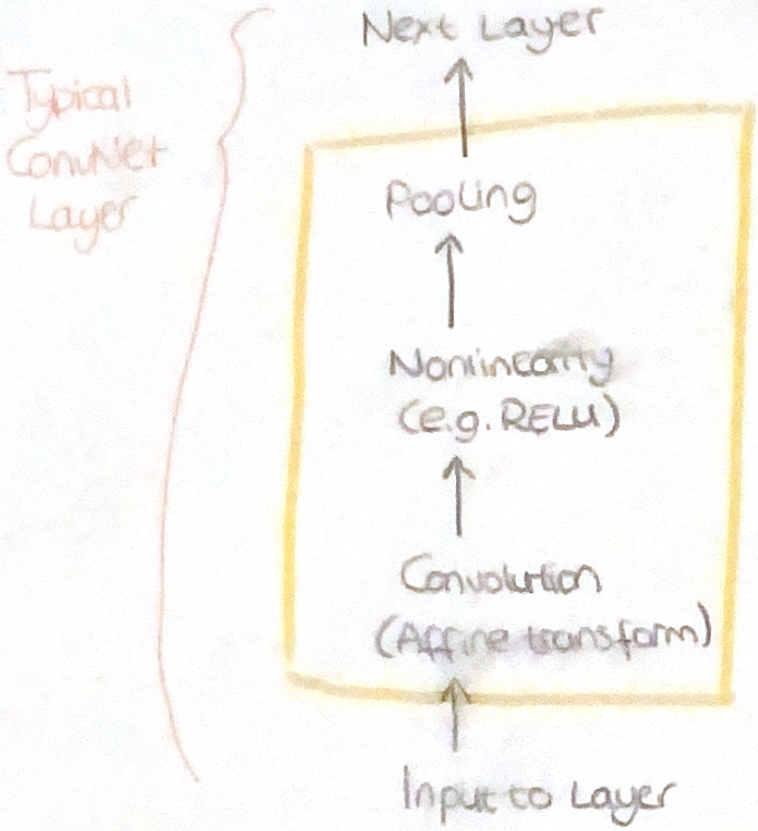
→ Max pooling reports the maximum outputs within a rectangular nbhd. (Other pooling func: L^2 norm, average, weighted average)

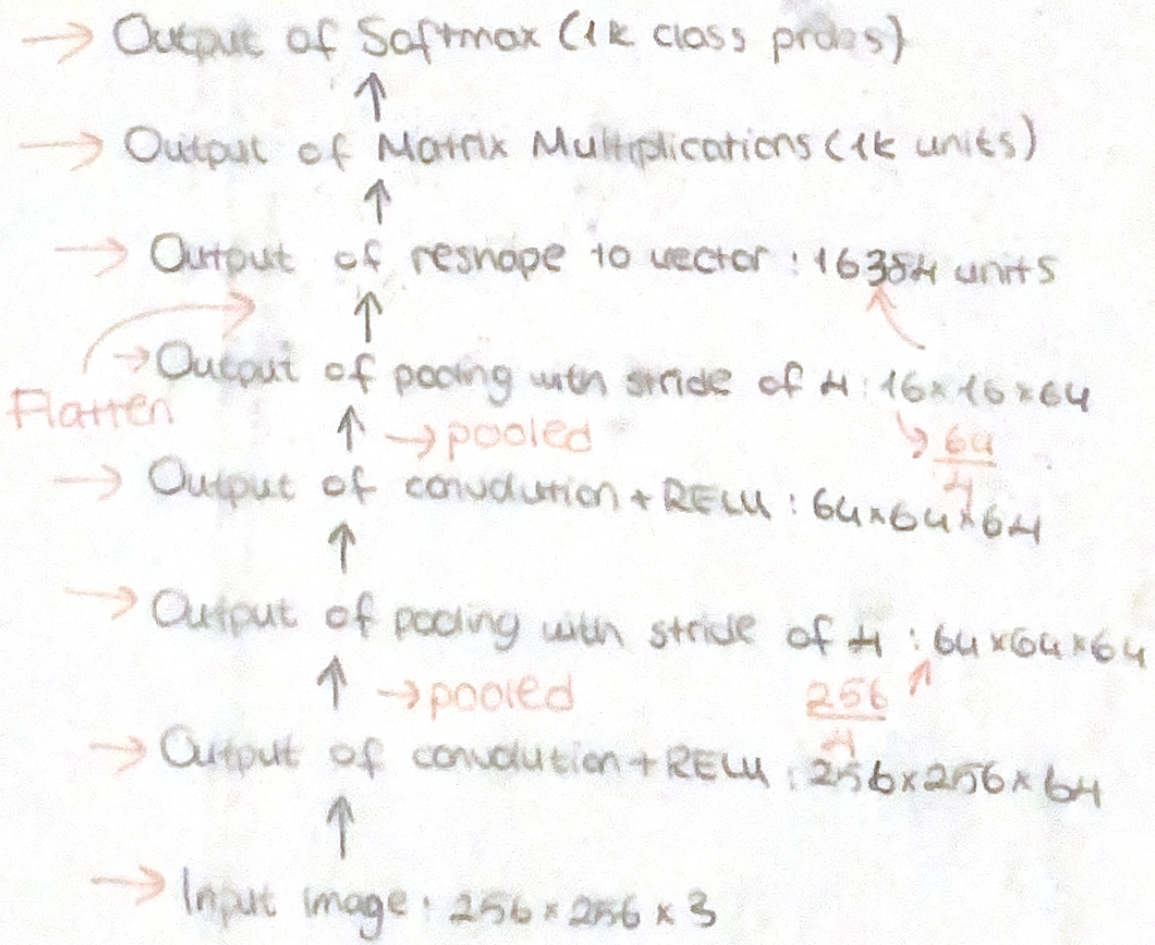
→ The representation becomes invariant to small translations. (Useful when some feature is present than exactly where it is.)
e.g. we don't need to know exact location of eyes, we need to know that a face has eyes on right & left.

→ Pooling is essential to handle varying input sizes. (Varying size of an offset between pooling regions such that classification layer always receives the same number of summary stats regardless of input size.



Max Pooling
(Invariance)


Next Layer
Typical ConvNet Layer
Pooling
Nonlinearity (e.g. RELU)
Convolutional Layer
Convolution (Affine transform)
Input to Layer


→ Output of Softmax (1k class probs)
→ Output of Matrix Multiplications (1k units)
→ Output of reshape to vector : 16384 units
Flatten
→ Output of pooling with stride of 4 : 16×16×64
→ pooled
64
→ Output of convolution + RELU : 64×64×64
→ Output of pooling with stride of 4 : 64×64×64
→ pooled
256
→ Output of convolution + RELU : 256×256×64
→ Input image : 256×256×3