CS772: Deep Learning for Natural Language Processing (DL-NLP)

Word2Vec, FFNN, BP

Pushpak Bhattacharyya

Computer Science and Engineering Department

IIT Bombay

Week 5 of 31st Jan, 2022

Example (1/3)

- 4 words: *heavy, light, rain, shower*
  - ○ *Heavy:* U_0 *<0,0,0,1>*
  - ○ *light:* U_1*: <0,0,1,0>*
  - ○ *rain:* U_2*: <0,1,0,0>*
  - ○ *shower:* U_3*: <1,0,0,0>*
- We want to predict as follows:
  - ○ *Heavy* → *rain*
  - ○ *Light* → *shower*

Note

- Any bigram is theoretically possible, but actual probability differs

- E.g., heavy-heavy, heavy-light are possible, but unlikely to occur
- Language imposes constraints on what bigrams are possible
- Domain and corpus impose further restriction

Example (2/3)

- We will call input as U and output as V
  - *Heavy: U_0 <0,0,0,1>, light: U_1: <0,0,1,0>, rain: U_2: <0,1,0,0>, shower: U_3: <1,0,0,0>*

  - *Heavy: V_0 <0,0,0,1>, light: V_1: <0,0,1,0>, rain: V_2: <0,1,0,0>, shower: V_3: <1,0,0,0>*

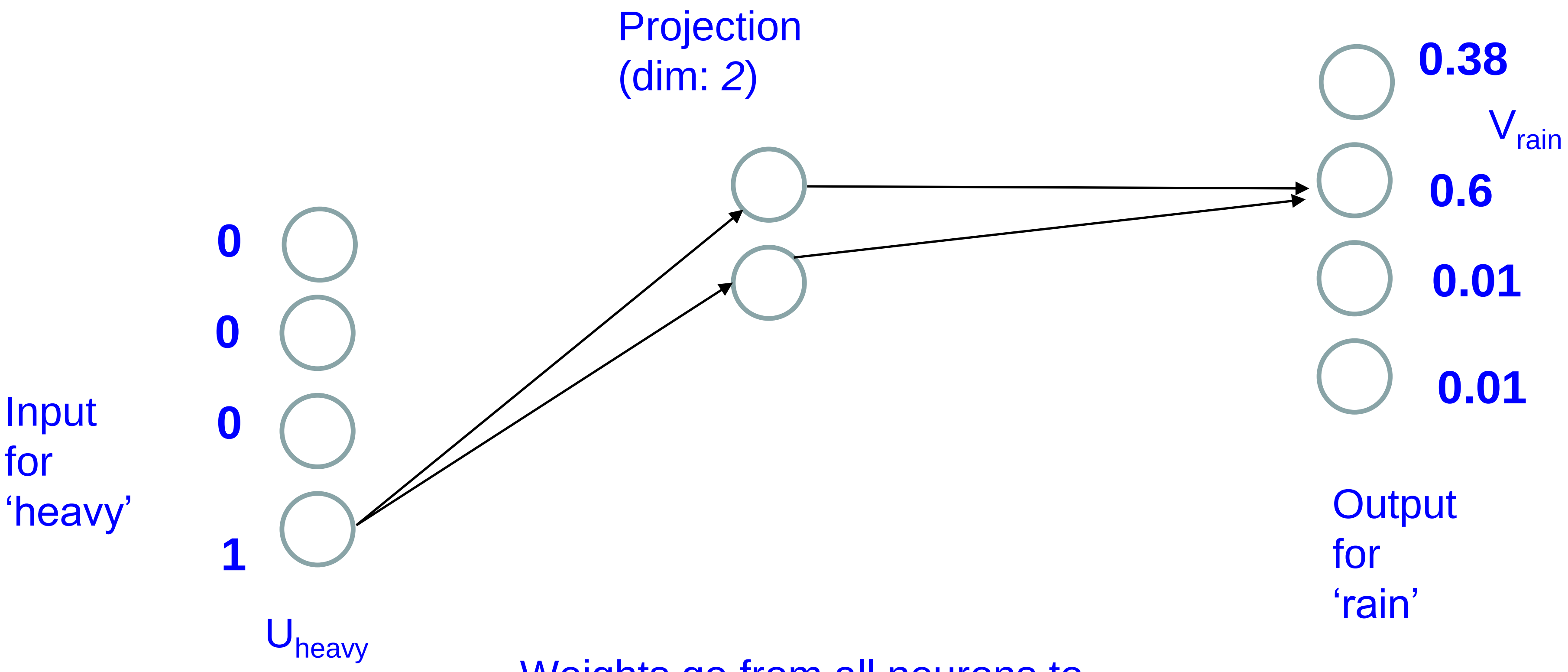
Example (3/3)

- *heavy → rain*
  - *heavy: U_0 <0,0,0,1>*

    →

  - *rain: V_2: <0,1,0,0>*
- *light → shower*
  - *light: U_1: <0,0,1,0>, → shower: V_3: <1,0,0,0>*

Word2vec n/w



Projection
(dim: *2*)

0.38

V_{rain}

0.6

0.01

0.01

0

0

Input
for
‘heavy’

0

1

U_{heavy}

Output
for
‘rain’

Weights go from all neurons to
all neurons in the next layer; shown
For only one input and output

Chain of thinking

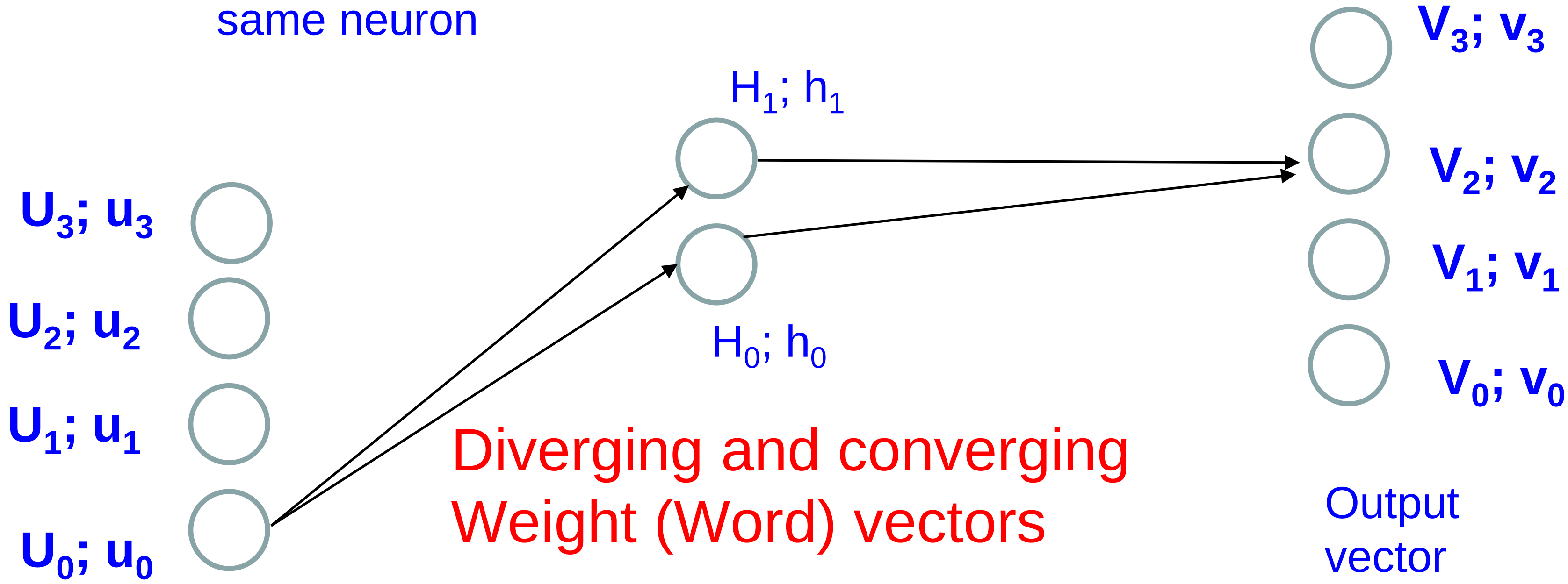
- P(rain|heavy) should be the highest
- So the output from V2 should be the highest because of softmax
- This way of converting an English statement into probability in insightful

Developing word2vec weight change rule

Illustrated with 4 words only

Word2vec n/w

Convention: Capital letter for NAME of neuron; small letter for output from the same neuron



V_3; v_3

V_2; v_2

V_1; v_1

V_0; v_0

H_1; h_1

H_0; h_0

U_3; u_3

U_2; u_2

U_1; u_1

U_0; u_0

Diverging and converging Weight (Word) vectors

Output vector V

Input vector U

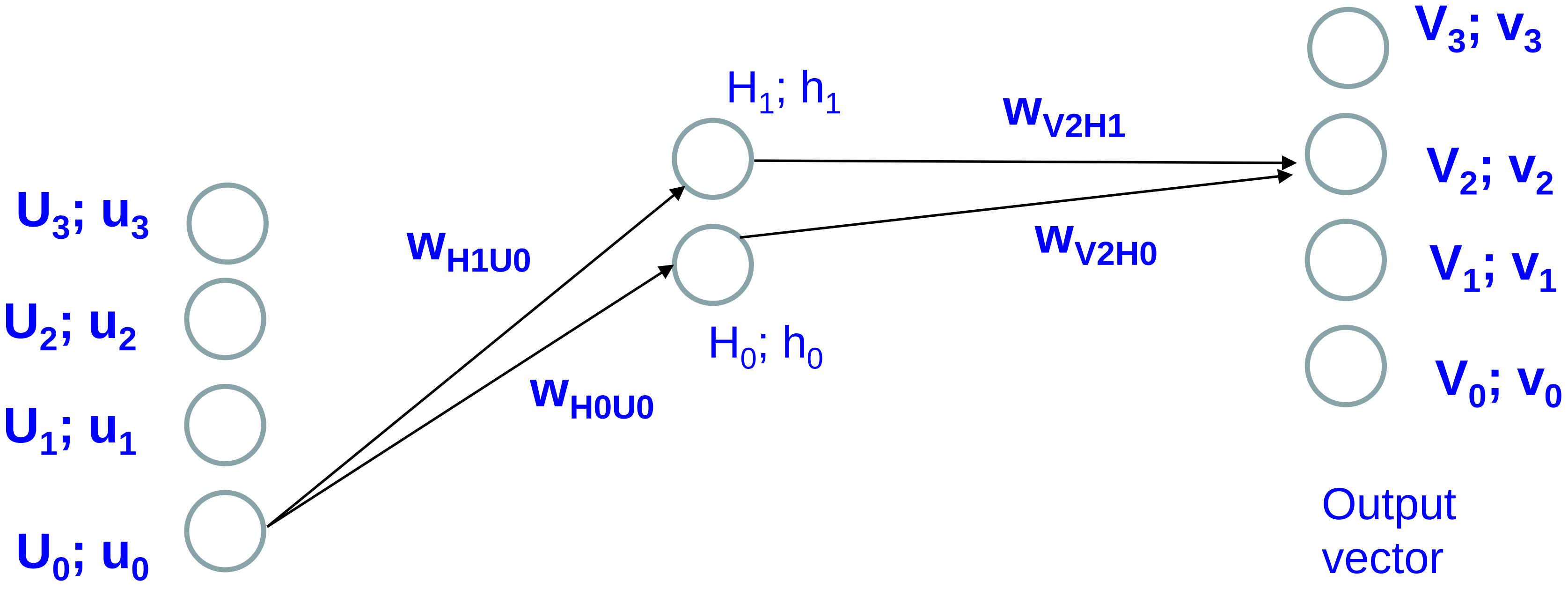
Weights go from all neurons to all neurons in the next layer; shown For only one input and output

Notation Convention

- Weights indicated by small ‘w’
- Index close to ‘w’ is for the destination neuron
- The other index is for the source neuron

Word2vec n/w

Capital letter for NAME of neuron; small letter for output from the same neuron



Output vector V

Input vector U

Weights go from all neurons to all neurons in the next layer; shown For only one input and output

More notation

- Net input to hidden and output layer neurons play an important role in BP

- Net input to hidden layer neurons: net_{H0} and net_{H1}

- Net input to output layer neurons: net_{V0}, net_{V1}, net_{V2}, net_{V3}

Outputs at the outermost layer

- Uses softmax

$$v_0 = \frac{e^{net_{V0}}}{e^{net_{V0}} + e^{net_{V1}} + e^{net_{V2}} + e^{net_{V3}}}$$

$$v_1 = \frac{e^{net_{V1}}}{e^{net_{V0}} + e^{net_{V1}} + e^{net_{V2}} + e^{net_{V3}}}$$

$$v_2 = \frac{e^{net_{V2}}}{e^{net_{V0}} + e^{net_{V1}} + e^{net_{V2}} + e^{net_{V3}}}$$

$$v_3 = \frac{e^{net_{V3}}}{e^{net_{V0}} + e^{net_{V1}} + e^{net_{V2}} + e^{net_{V3}}}$$

Note

- No non-linearity in the hidden layer
- Why?
- Hidden layer should do ONLY dimensionality reduction
- Can be proved: hidden layer with linearity gives the principal components (will discuss of which Matrix)

Why Dimensionality Reduction?

- The vectors of words represent their distributional similarity

- Dimensionality reduction achieves capturing commonality of these distributional similarities across words

Softmax

What is softmax

- Turns a vector of K real values into a vector of K real values that sum to 1
- Input values can be positive, negative, zero, or greater than one
- But softmax transforms them into values between 0 and 1
- so that they can be interpreted as probabilities.

Mathematical form

$$S(\bar{Z})_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{Z_j}},$$

$$LHS\ is\ the\ i^{th}\ component$$

$$of\ the\ soft\max\ output\ vector$$

- *S(.)* is the **softmax** function, returns a vector
- *Z* is the input vector of size *K*
- The RHS gives the *i*i^{th} component of the output vector
- Input to softmax and output of softmax are of the same dimension

Example

$$\bar{Z} \;\; = < 1,\ 2,\ 3 >$$

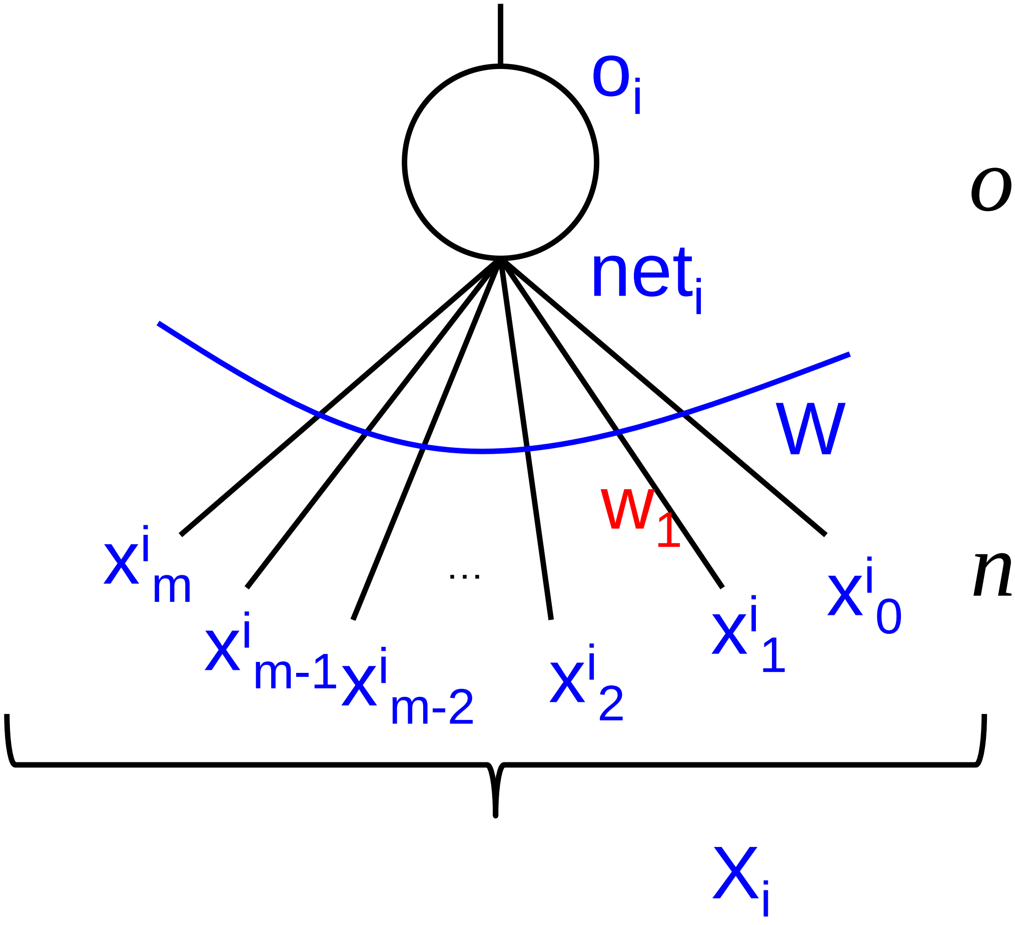
$$Z_1 = 1,\ Z_2 = 2,\ Z_3 = 3$$

$$e^1 = 2.72,\ e^2 = 7.39,\ e^3 = 20.09$$

$$\sigma(\bar{Z}) = < \frac{2.72}{2.72 + 7.39 + 20.09}, \frac{7.39}{2.72 + 7.39 + 20.09}, \frac{20.09}{2.72 + 7.39 + 20.09} >$$

$$= < .09,\ 0.24,\ 0.67 >$$

Sigmoid neuron



$$o_i = \frac{1}{1+e^{-net_i}}$$

$$net_i = W.X_i = \sum_{j=0}^{m} w_j x^i_j$$

Interpreting o_i

- o_i value is between 0 and 1
- Interpreted as probability
- 2-class situation, o_i value is looked upon as probability of class being 1
- That is, *P(Class=1 for ith input)*

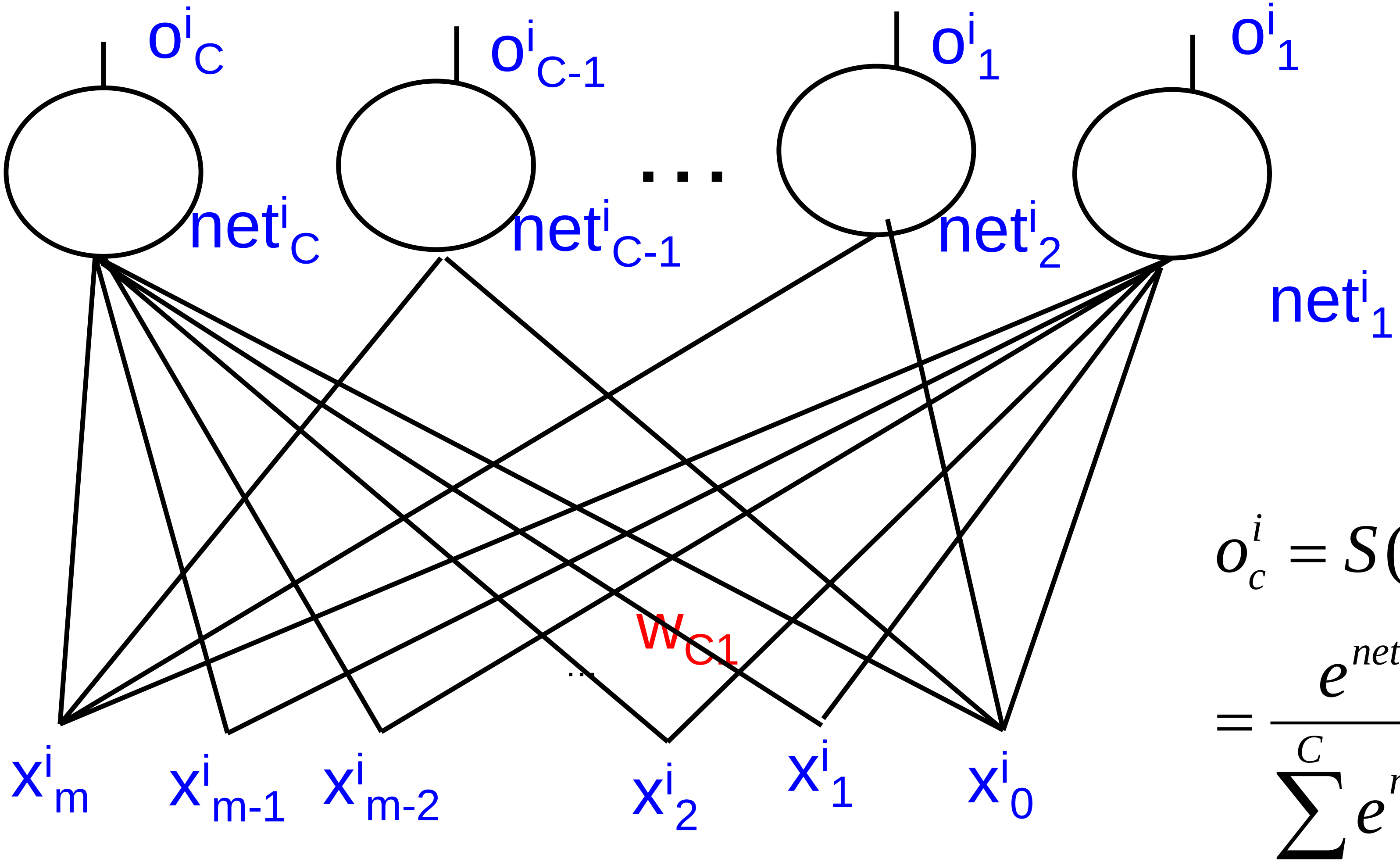
  $= o_i = 1/(1+e^{-neti})$
- Each training data instance is labeled as 1 or 0
- Target value $t_i = 1/0$, for *i*th input

Generalizing 2-class to multiclass: SOFTMAX

$$o_c^i = S(\bar{NET}_i)_c = \frac{e^{net_c^i}}{\sum_{k=1}^{C} e^{net_k^i}},$$

- 2-class → multi-class (C classes)
- Sigmoid → softmax
- *i*th input, *c*th class (small c), *k* varies over classes

Softmax Neuron



$$o_c^i = S(\bar{NET}_i)_c = \frac{e^{net_c^i}}{\sum_{k=1}^{C} e^{net_k^i}},$$

Target Vector, T_i: $<t^i_C\ t^i_{C-1}\ldots t^i_2\ t^i_1>$, i → for i^{th} input.

Only one of these C componets is 1, rest are 0.

Compare and contrast Sigmoid and Softmax

$$sigmoid: o_i = \frac{1}{1+e^{-net_i}},\ for\ i^{th}\ input$$

$$soft\max: o_c^i = \frac{e^{net_c^i}}{\sum_{k=1}^{C} e^{net_k^i}},$$

ith input, *c*th class (small c), *k* varies over classes 1 to *C*

Interpreting o^i_c

- o^i_c value is between 0 and 1
- Interpreted as probability
- Multi-class situation
- o^i_c value is the probability of the class being *'c'* for the i^{th} input

- That is,

$$P(\text{Class of } i^{th} \text{ input}=c)=o^i_c$$

Derivatives

Derivative of sigmoid

$$o_i = \frac{1}{1+e^{-net_i}}, \ for\ i^{th}\ input$$

$$\ln o_i = -\ln(1+e^{-net_i})$$

$$\frac{1}{o_i}\frac{\partial o_i}{\partial net_i} = -\frac{1}{1+e^{-net_i}}.-e^{-net_i} = \frac{e^{-net_i}}{1+e^{-net_i}} = (1-o_i)$$

$$\Rightarrow \frac{\partial o_i}{\partial net_i} = o_i(1-o_i)$$

Derivative of Softmax

$$o_c^i = \frac{e^{net_c^i}}{\sum_{k=1}^{C} e^{net_k^i}},\ i^{th}\ input\ pattern$$

$$\ln o_c^i = e^{net_c^i} - \ln(\sum_{k=1}^{C} e^{net_k^i})$$

Derivative of Softmax: Case-1, class *c* for O and NET same

$$\ln o_c^i = net_c^i - \ln(\sum_{k=1}^{C} e^{net_k^i})$$

$$\frac{1}{o_c^i}\frac{\partial o_c^i}{\partial net_c^i} = 1 - \frac{1}{\sum_{k=1}^{C} e^{net_k^i}}.e^{net_c^i} = 1 - o_c^i$$

$$\Rightarrow \frac{\partial o_c^i}{\partial net_c^i} = o_c^i(1 - o_c^i)$$

Derivative of Softmax: Case-2, class *c’* in $net^i_{c'}$ different from class c of O

$$\ln o_c^i = net_c^i - \ln(\sum_{k=1}^{C} e^{net_k^i})$$

$$\frac{1}{o_c^i}\frac{\partial o_c^i}{\partial net_{c'}^i} = 0 - \frac{1}{\sum_{k=1}^{C} e^{net_k^i}}.e^{net_{c'}^i} = -o_{c'}^i$$

$$\Rightarrow \frac{\partial O_k^c}{\partial net_{c'}^i} = -o_c^i o_{c'}^i$$

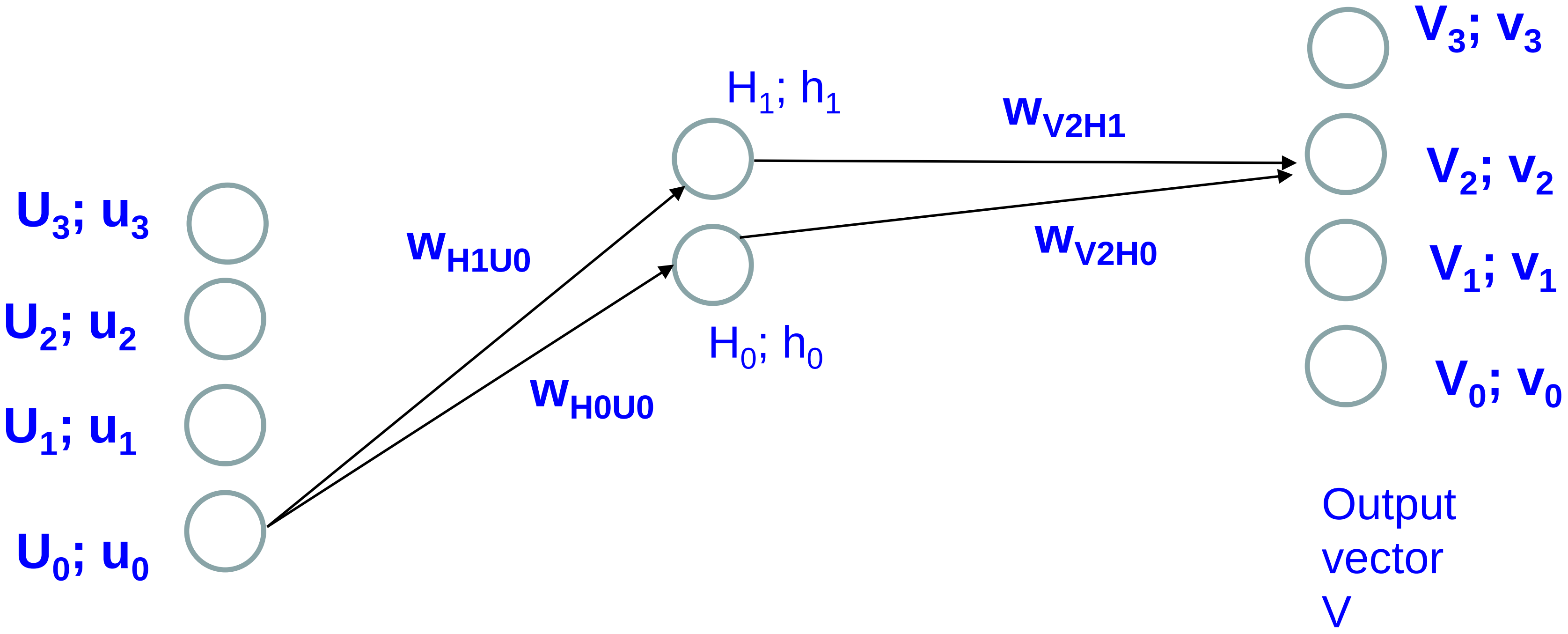
Exercise

- Unify the two cases of derivatives
- Give a SINGLE expression

Back to word2vec

Word2vec n/w

Capital letter for NAME of neuron; small letter for output from the same neuron



Input
vector
U

Weights go from all neurons to all neurons in the next layer; shown For only one input and output

Outputs at the outermost layer

- Uses softmax

$$v_0 = \frac{e^{net_{V_0}}}{e^{net_{V_0}} + e^{net_{V_1}} + e^{net_{V_2}} + e^{net_{V_3}}}$$

$$v_1 = \frac{e^{net_{V_1}}}{e^{net_{V_0}} + e^{net_{V_1}} + e^{net_{V_2}} + e^{net_{V_3}}}$$

$$v_2 = \frac{e^{net_{V_2}}}{e^{net_{V_0}} + e^{net_{V_1}} + e^{net_{V_2}} + e^{net_{V_3}}}$$

$$v_3 = \frac{e^{net_{V_3}}}{e^{net_{V_0}} + e^{net_{V_1}} + e^{net_{V_2}} + e^{net_{V_3}}}$$

Developing “net$_{vi}$” (1/2)

$$net_{V_0} = w_{V_0H_0}h_0 + w_{V_0H_1}h_1$$

$$h_0 = w_{H_0U_0}u_0 + w_{H_0U_1}u_1 + w_{H_0U_2}u_2 + w_{H_0U_3}u_3$$

$$h_1 = w_{H_1U_0}u_0 + w_{H_1U_1}u_1 + w_{H_1U_2}u_2 + w_{H_1U_3}u_3$$

Developing "net$_{vi}$" (2/2)

- For "heavy", only u_0 is 1, u_1=u_2=u_3=0
- So,

$$h_0 = w_{H_0U_0}$$

$$h_1 = w_{H_1U_0}$$

$$net_{v_0} = w_{V_0H_0}w_{H_0U_0} + w_{V_0H_1}w_{H_1U_0}$$

More Notation

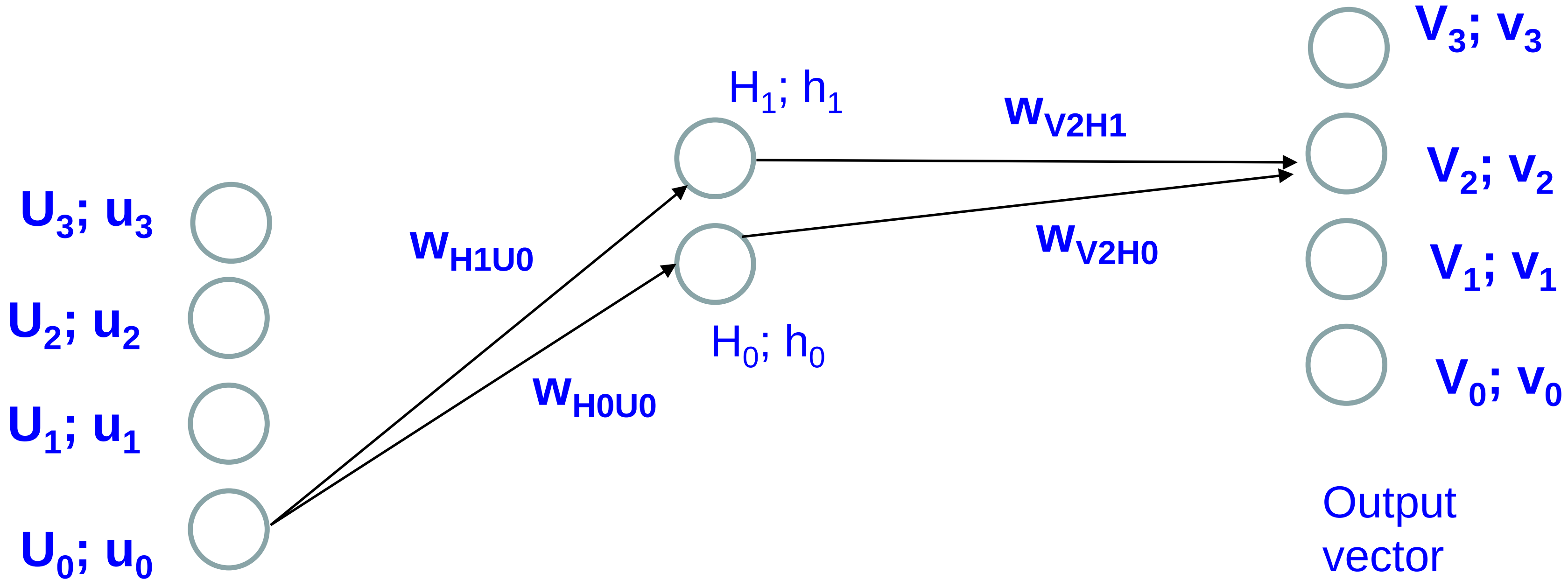
- Weight vector FROM U_0 is called W_{U0} (capital ‘W’)
- Weight vector INTO V_0 is called W_{V0}
- Slight liberty with notation, but has intuitive advantage

For “heavy” (=U_0), the value of net_{v0}

$$net_{V_0} = W_{V_0} . W_{U_0}$$

Word2vec n/w

Capital letter for NAME of neuron; small letter for output from the same neuron



Output vector V

Input vector U

Weights go from all neurons to all neurons in the next layer; shown For only one input and output

For “heavy” (=U_0), values of other net_{vi}s

$$net_{V_1} = W_{V_1}.W_{U_0}$$

$$net_{V_2} = W_{V_2}.W_{U_0}$$

$$net_{V_3} = W_{V_3}.W_{U_0}$$

We want to maximize *P('rain'=V$_2$|'heavy'=U$_0$)*

- This probability is in terms of softmax.

$$P('rain'=V_2\,|'heavy'=U_1)$$

$$=v_2=\frac{e^{net_{V_2}}}{e^{net_{V_0}}+e^{net_{V_1}}+e^{net_{V_2}}+e^{net_{V_3}}}$$

Equivalent to

- minimize *-log[P(‘rain’=V$_2$|’heavy’=U$_0$)]*

$$-\log[P('rain'=V_2\,|'heavy'=U_1)]$$

$$=-net_{V_2}+\log(e^{net_{V0}}+e^{net_{V1}}+e^{net_{V2}}+e^{net_{V3}})$$

$$=-W_{V_2}.W_{U_0}+\log(e^{net_{V0}}+e^{net_{V1}}+e^{net_{V2}}+e^{net_{V3}})$$

Equivalent to

- minimize *-log[P(‘rain’=V$_2$|’heavy’=U$_0$)]*

$$-\log[P('rain'=V_2\,|'heavy'=U_0)]$$

$$=-net_{V_2}+\log(e^{net_{V_0}}+e^{net_{V_1}}+e^{net_{V_2}}+e^{net_{V_3}})$$

$$=-W_{V_2}.W_{U_0}+$$

$$\log(e^{W_{V_0}.W_{U_0}}+e^{W_{V_1}.W_{U_0}}+e^{W_{V_2}.W_{U_0}}+e^{W_{V_3}.W_{U_0}})$$

Error/Loss Function

- minimize -*log*[*P*(*‘rain’*=*V*$_2$|*’heavy’*=*U*$_0$)]

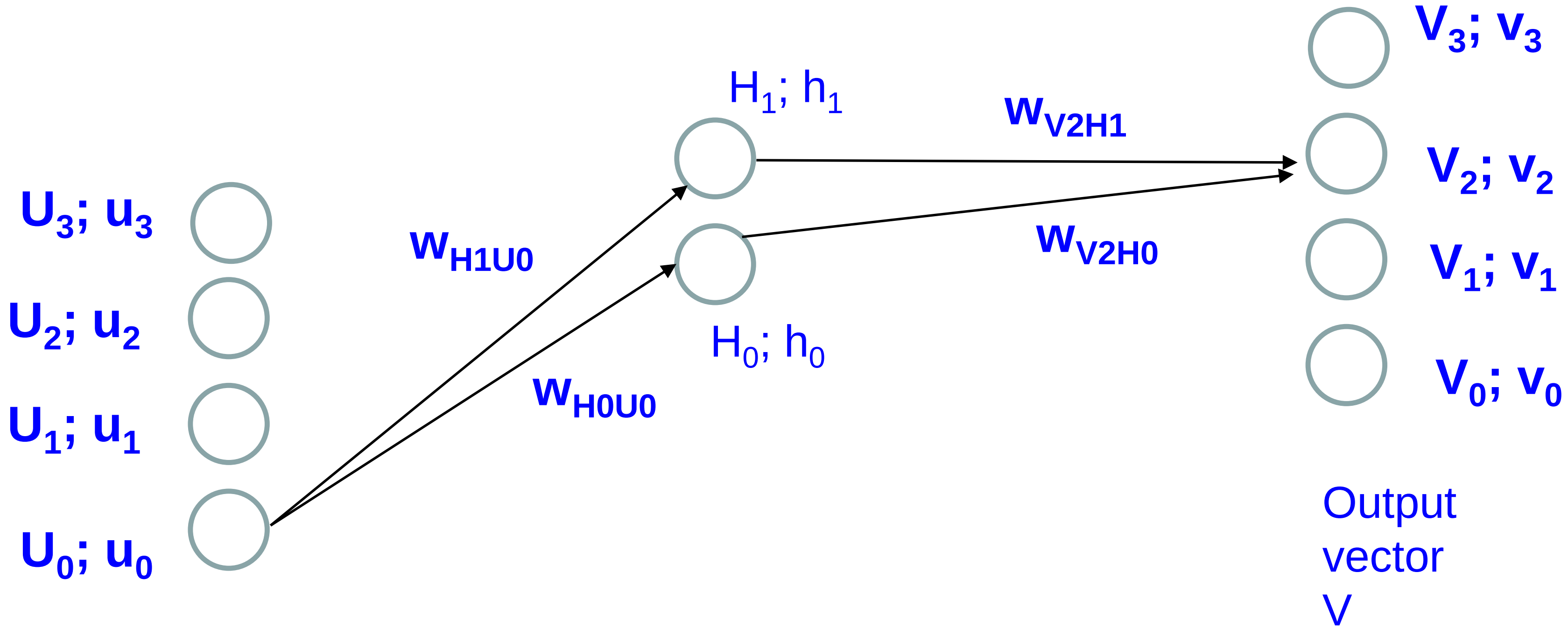
$$E = -W_{V_2}.W_{U_0} +$$

$$\log(e^{W_{V0}.W_{U0}} + e^{W_{V1}.W_{U0}} + e^{W_{V2}.W_{U0}} + e^{W_{V3}.W_{U0}})$$

$$W_{V_2}.W_{U_0} = w_{V_2H_0}w_{H_0U_0} + w_{V_2H_1}w_{H_1U_0}$$

Word2vec n/w

Capital letter for NAME of neuron; small letter for output from the same neuron



Input
vector
U

Weights go from all neurons to all neurons in the next layer; shown For only one input and output

Computing Δw_{V2H0}

$$\Delta w_{V_2H_0} = -\eta \frac{\delta E}{\delta w_{V_2H_0}}$$

$$E = -W_{V_2}.W_{U_0} + \log(e^{W_{V0}.W_{U0}} + e^{W_{V1}.W_{U0}} + e^{W_{V2}.W_{U0}} + e^{W_{V3}.W_{U0}})$$

$$W_{V_2}.W_{U_0} = w_{V_2H_0} w_{H_0U_0} + w_{V_2H_1} w_{H_1U_0}$$

$$\frac{\delta E}{\delta w_{V_2H_0}} = -w_{H_0U_0} + \frac{e^{W_{V2}.W_{U0}}}{e^{W_{V0}.W_{U0}} + e^{W_{V1}.W_{U0}} + e^{W_{V2}.W_{U0}} + e^{W_{V3}.W_{U0}})} .w_{H_0U_0}$$

$$= -w_{H_0U_0} + v_2.w_{H_0U_0}$$

$$\Rightarrow \Delta w_{V_2H_0} = \eta(1-v_2).w_{H_0U_0} = \eta(1-v_2)o_{H_0}$$

o/p of hidden neuron H_0

Interpretation of weight change rule for V_2

- If v2 is close to 1, change in weight too is small
- w_{H0U0} is equal to the input to H_0 (since $u_0=1$) and to its output too, since hidden neurons simply transmit the output.

Change in other weights to output layer, say, *V_1*, due to input *U_0*

$$\Delta w_{V_1H_0} = -\eta \frac{\delta E}{\delta w_{V_1H_0}}$$

$$E = -W_{V_2}.W_{U_0} + \log(e^{W_{V_0}.W_{U_0}} + e^{W_{V_1}.W_{U_0}} + e^{W_{V_2}.W_{U_0}} + e^{W_{V_3}.W_{U_0}})$$

$$W_{V_2}.W_{U_0} = w_{V_2H_0}w_{H_0U_0} + w_{V_2H_1}w_{H_1U_0}$$

$$\frac{\delta E}{\delta w_{V_1H_0}} = -0 + \frac{e^{W_{V_1}.W_{U_0}}}{e^{W_{V_0}.W_{U_0}} + e^{W_{V_1}.W_{U_0}} + e^{W_{V_2}.W_{U_0}} + e^{W_{V_3}.W_{U_0}})} .w_{H_0U_0}$$

$$= v_1.w_{H_0U_0}$$

$$\Rightarrow \Delta w_{V_1H_0} = -\eta v_1 w_{H_0U_0} = -\eta v_1 o_{H_0}$$

Interpretation of weight change rule for V_1

- Assume w_{H0U0} to be positive
- For training $U0 \rightarrow V2$, i.e., ‘heavy’→’rain’, if v_2 is not 1, Δw_{V2H0} is +ve
- For the same input, Δw_{V1H0} is negative
- So the two weight changes are of opposite sign.
- The effect is that while v_2 increases, v_1 decrease for the input U_0, as it should be since we want to increase *P(‘rain’|’heavy’)* and depress all other probabilities

Weight change for input to hidden layer, say, *w_{H0U0}*

$$\Delta w_{H_0U_0} = -\eta \frac{\delta E}{\delta w_{H_0U_0}}$$

$$E = -W_{V_2}.W_{U_0} + \log(e^{W_{V_0}.W_{U_0}} + e^{W_{V_1}.W_{U_0}} + e^{W_{V_2}.W_{U_0}} + e^{W_{V_3}.W_{U_0}})$$

$$W_{V_2}.W_{U_0} = w_{V_2H_0}w_{H_0U_0} + w_{V_2H_1}w_{H_1U_0}$$

Cntd: Weight change for input to hidden layer, say, w_{H0U0}

$$\frac{\delta E}{\delta w_{H_0U_0}}$$

$$= -w_{V_2H_0} + \frac{w_{V_0H_0}e^{W_{V_0}.W_{U0}} + w_{V_1H_0}e^{W_{V_1}.W_{U0}} + w_{V_2H_0}e^{W_{V_2}.W_{U0}} + w_{V_3H_0}e^{W_{V3}.W_{U0}}}{e^{W_{V0}.W_{U0}} + e^{W_{V_1}.W_{U0}} + e^{W_{V_2}.W_{U0}} + e^{W_{V_3}.W_{U0}})}$$

$$= -w_{V_2H_0} + w_{V_0H_0}v_0 + w_{V_1H_0}v_1 + w_{V_2H_0}v_2 + w_{V_3H_0}v_3$$

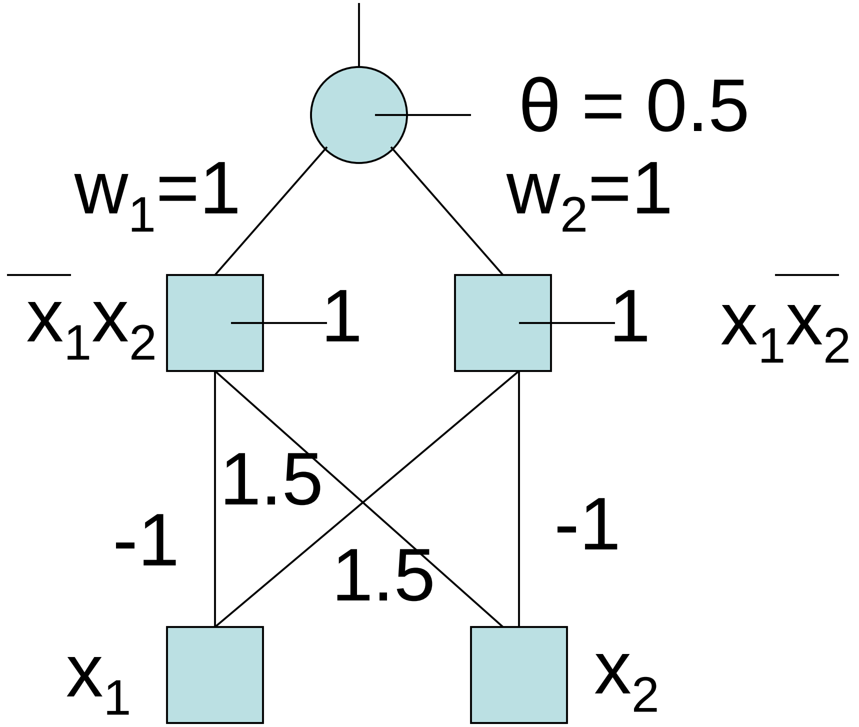
$$\Rightarrow \Delta w_{H_0U_0} = \eta[(1-v_2)w_{V_2H_0} - w_{V_0H_0}v_0 - w_{V_1H_0}v_1 - w_{V_3H_0}v_3]$$

Need for efficiency

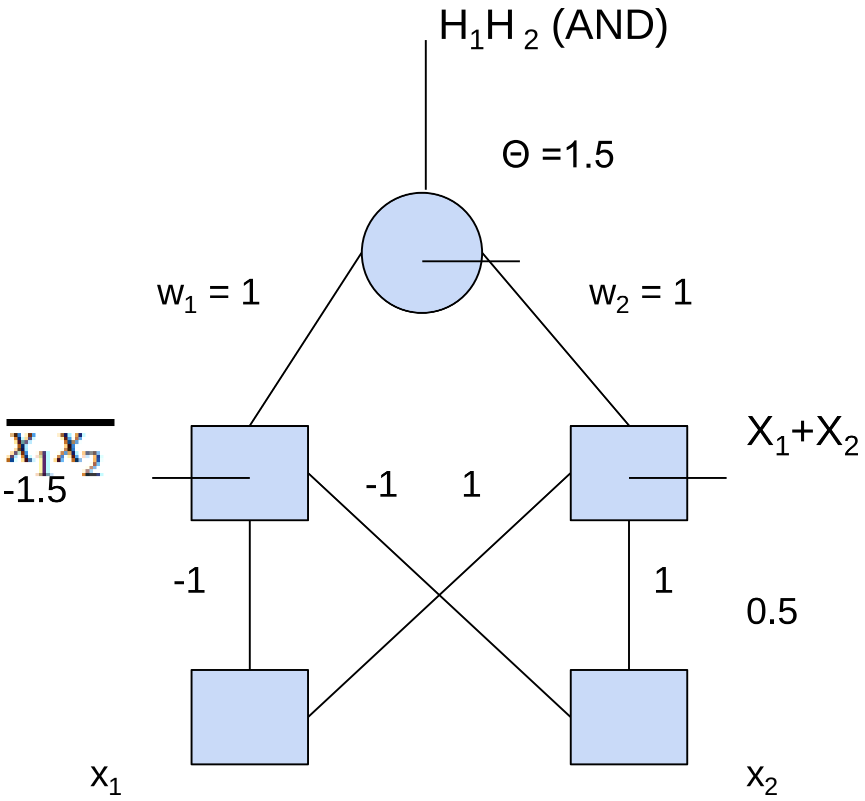
- Hierarchical softmax
- Negative sampling
- We have to update |H|.|V| weights in the hidden to output layer
- |H|=dimension of hidden layer, |V|=vocab size
- For 300 dimension word vector and 100,000 words vocabulary, 30 million weights need to be updated for every input word!!
- Efficiency measures to be discussed

Feedforward Network and Backpropagation

Example - XOR



Alternative network for XOR

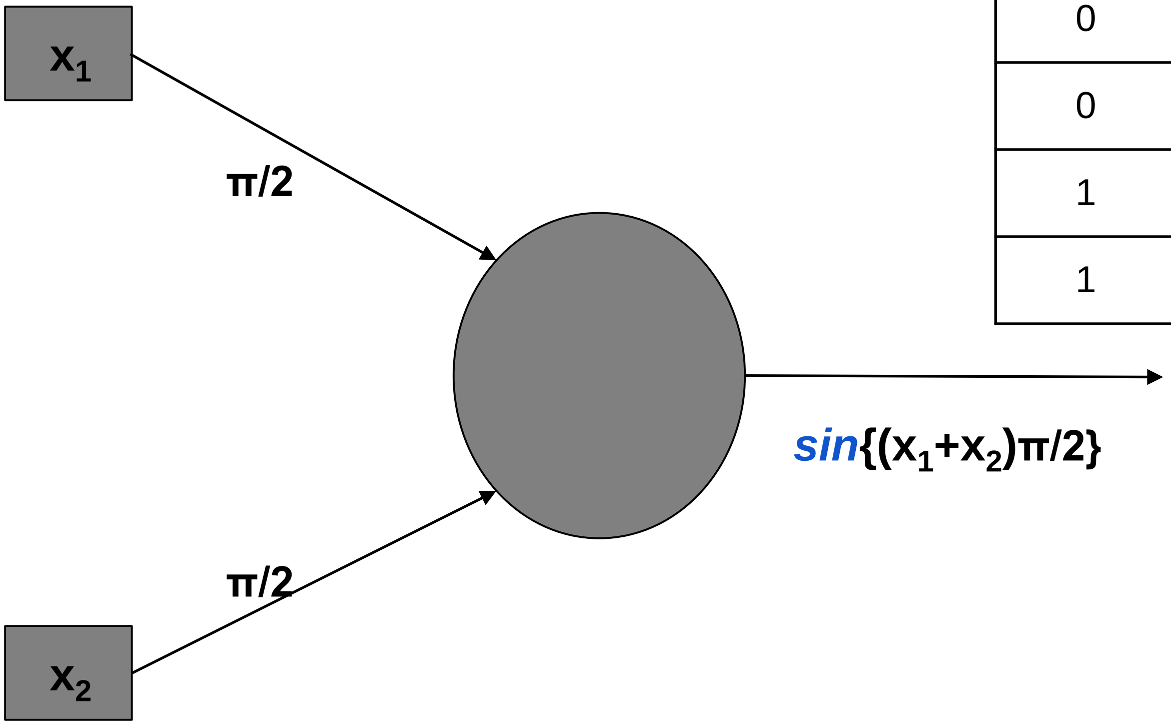


- XOR: not possible using a single perceptron
- Hidden layer gives more computational capability
- Deep neural network: With multiple hidden layers
- Kolmogorov’s theorem of equivalence proves equivalence of multiple layer neural network to a single layer neural network, and each neuron have to correspond to an appropriate functions.

Compositionality

- XOR being computed as OR(X_1'X_2, X_1X_2') or as AND((X_1'+X_2'),(X1+X2)) is an example of a nonlinearly separable function computed as composition of linearly separable functions)
- In general not possible for most practical situations like weather prediction, stock market prediction etc.

XOR neuron with sin()



x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Question

- Since SINE can compute XOR, why do not we use sine neuron for practical applications?

Exercise: Back-propagation

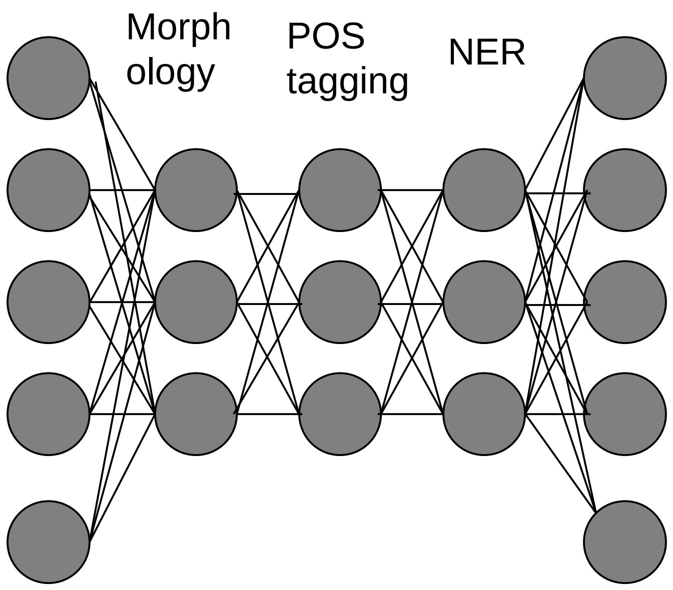
- Implement back-propagation for XOR network

- Observe
  - Check if it converges (error falls below a limit)

  - What is being done at the hidden layer

What a neural network can represent in NLP: Indicative diagram

- Each layer of the neural network possibly represents different NLP stages!!



Batch learning versus Incremental learning

- Batch learning is updating the parameters after ONE PASS over the whole dataset
- Incremental learning updates parameters after seeing each PATTERN (input-ouput pair)
- An epoch is ONE PASS over the entire dataset
  - Take XOR: data set is $V_1=(<0,0>, 0)$, $V_2=(<0,1>, 1)$, $V_3=(<1,0>, 1)$, $V_4=(<1,1>, 0)$
  - If the weight values are changed after each of Vi, then this is incremental learning
  - If the weight values are changed after one pass over all V_is, then it is bathc learning

Can we use PTA for training FFN?

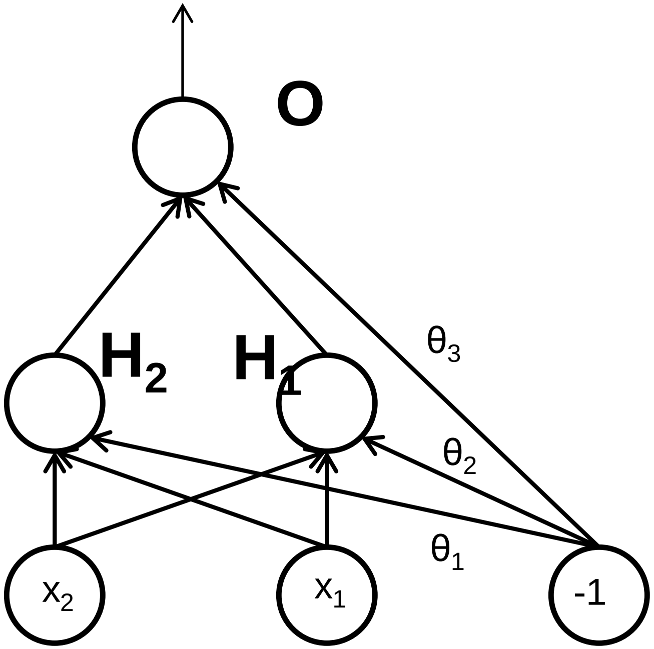
0, 0	0
0, 1	1
1, 0	1
1, 1	0

⇨

-1, 0, 0	0
-1, 0, 1	1
-1, 1, 0	1
-1, 1, 1	0

⇨

1, 0, 0	0
-1, 0, 1	1
-1, 1, 0	1
1, -1, -1	0



No, else the individual neurons are solving XOR, which is impossible.

Also, for the hidden layer neurons we do nothave the i/o behaviour.

Note: This n/w is NOT a pure FFNN; there is jumping of lair.

Gradient Descent Technique

- Let E be the error at the output layer
- *i* goes over *N* neurons in the o/p layer, *j* goes over *P* patterns

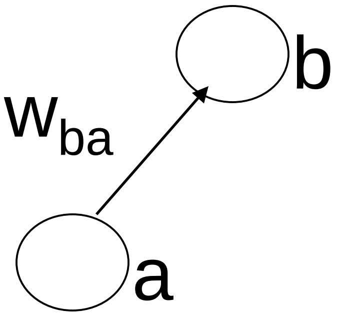
$$E = \frac{1}{2}\sum_{j=1}^{P}\sum_{i=1}^{N}(t_i - o_i)_j^2$$

- t_i = target output; o_i = observed output

- E.g.: XOR:– *P=4 and N=1*

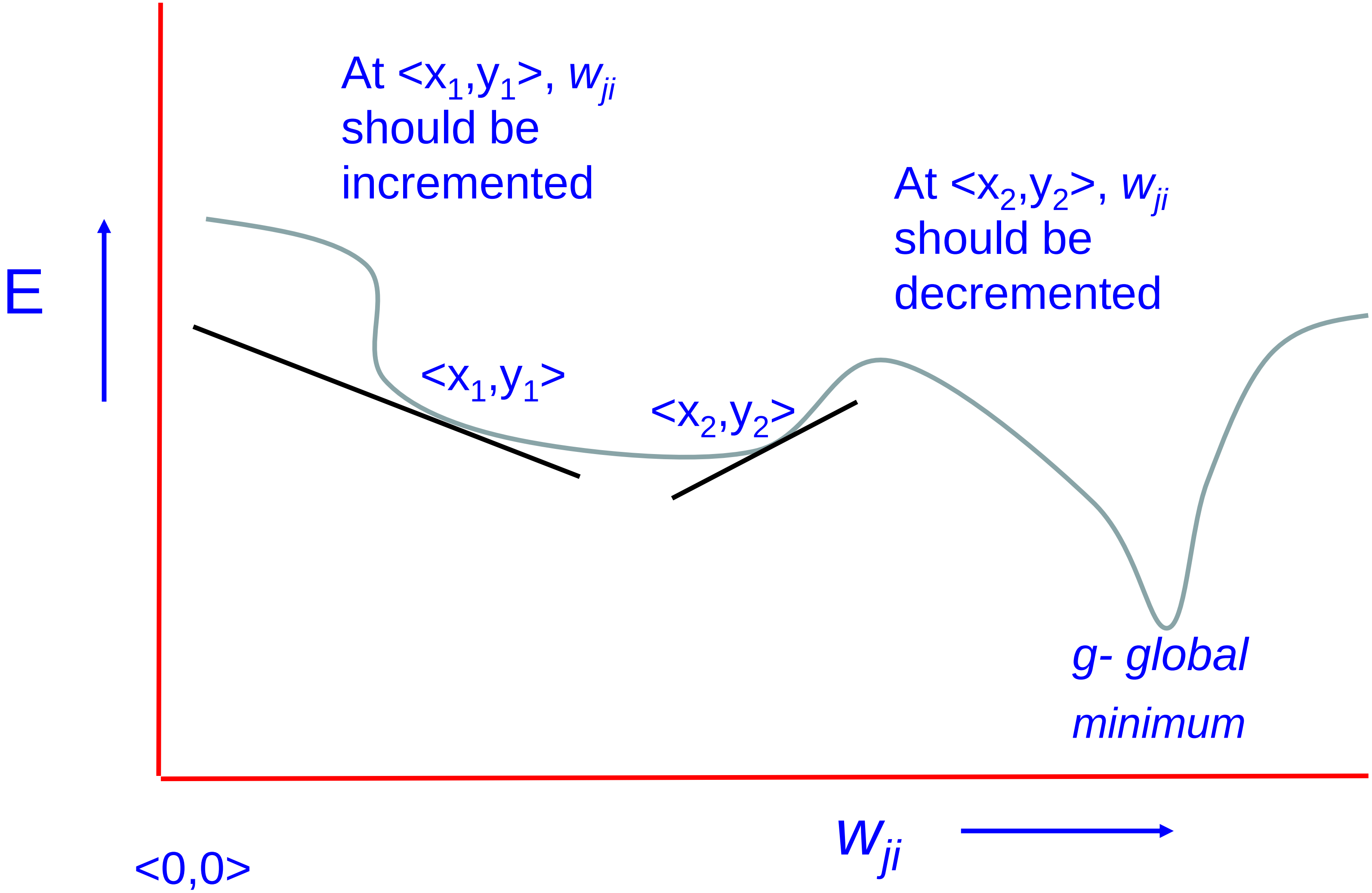
Weights in a FF NN

- w_{ba} is the weight of the connection from the a^{th} neuron to the b^{th} neuron
- E *vs* $\overline{W}$ surface is a complex surface in the space defined by the weights w_{ij}
- $-\frac{\delta E}{\delta w_{ba}}$ gives the direction in which a movement of the operating point in the w_{mn} co-ordinate space will result in maximum decrease in error



$$\Delta w_{ba} \propto -\frac{\delta E}{\delta w_{ba}}$$

Intuition for gradient descent



At $<x_1,y_1>$, w_{ji} should be incremented

At $<x_2,y_2>$, w_{ji} should be decremented

E

$<x_1,y_1>$

$<x_2,y_2>$

g- global minimum

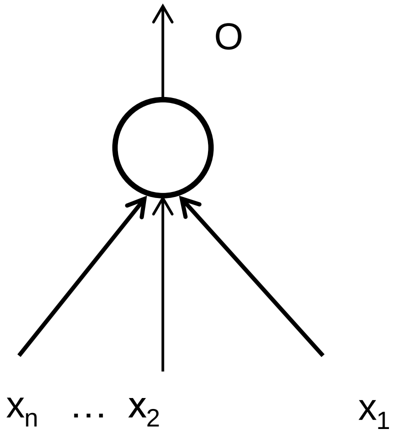
w_{ji}

<0,0>

Pertains to life!!

- Gradient descent in greedy in nature, E ALWAYS decreases
- Can get stuck in local minimum, miss global minimum
- So: “greed does not always pay”, “short term gains may not lead to long term gains”, “local optimizations need not always lead to global optimizations”

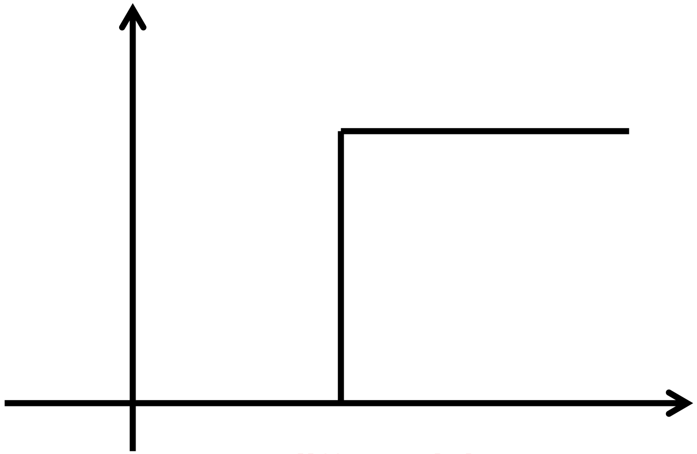
Step function v/s Sigmoid function



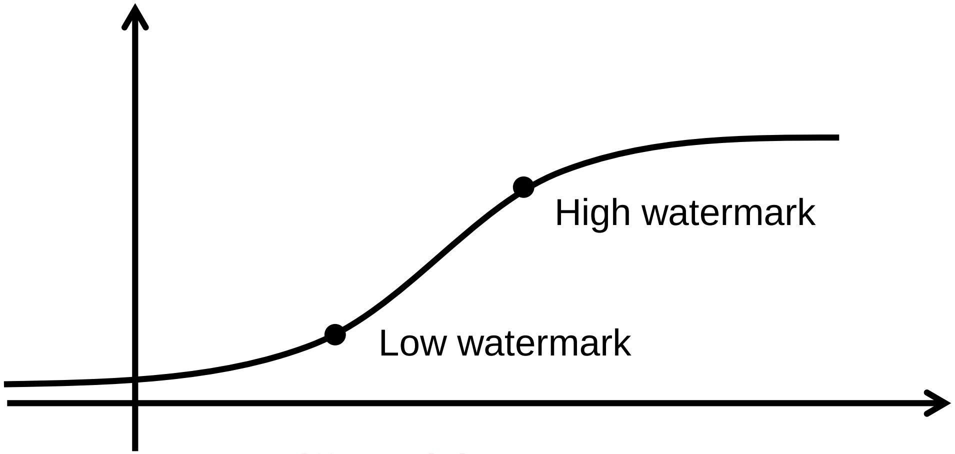
$$
\begin{aligned}
O &= f(\sum w_i x_i) \\
  &= f(net)
\end{aligned}
$$

So partial derivative of O w.r.t. net is

$$\frac{\delta O}{\delta net}$$



Non-differentiable



Differentiable

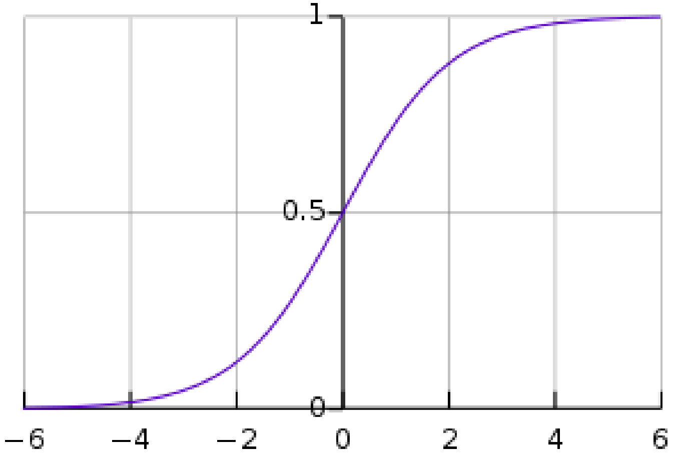
Sigmoid function

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

$$y = \frac{1}{1+e}$$

$$\frac{dy}{dx} = y(1-y)$$

Sigmoid function



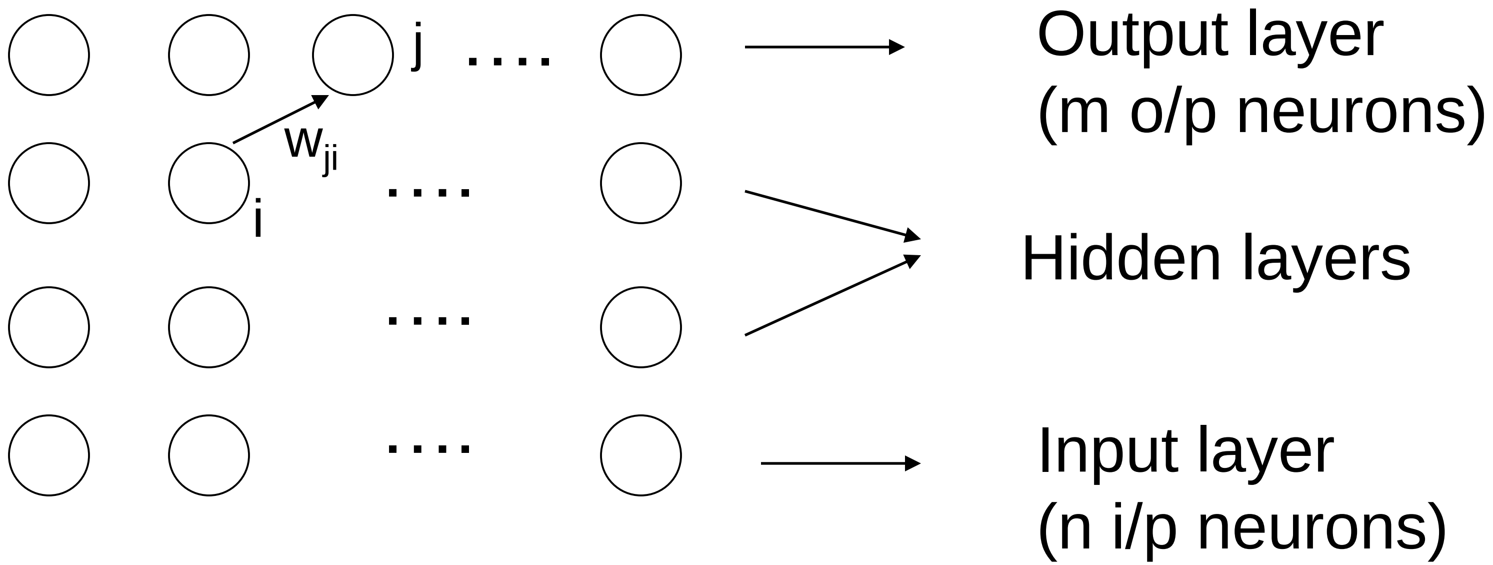
$$f(x) = \frac{1}{1+e^{-x}}$$

$$\begin{aligned}
f(x) &= \frac{1}{1+e^{-x}} \\
\frac{df(x)}{dx} &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{e^{-x}}{(1+e^{-x})^{-2}} \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= f(x).(1 - f(x))
\end{aligned}$$

Interesting point

- Biological (neurophysical) plausibility of sigmoid function
- The saturating behaviour of sigmoid neuron for very large signals (derivative →0) is said to be a “saviour” for the brain
- Intense emotions (joy, sorrow, anger) produce large signals in brain neurons which through positive feedback can lead to brain damage (haemorrhage)
- Saturation avoids this danger

Backpropagation algorithm



- Fully connected feed forward network
- Pure FF network (no jumping of connections over layers)

Gradient Descent Equations

$$\Delta w_{ji} = -\eta \frac{\delta E}{\delta w_{ji}} (\eta = \text{learning rate}, 0 \le \eta \le 1)$$

$$\frac{\delta E}{\delta w_{ji}} = \frac{\delta E}{\delta net_j} \times \frac{\delta net_j}{\delta w_{ji}} (net_j = \text{input at the j}^{th} \text{ neuron})$$

$$\boxed{\frac{\delta E}{\delta net_j} = -\delta j}$$

$$\Delta w_{ji} = \eta \delta j \frac{\delta net_j}{\delta w_{ji}} = \eta \delta j o_i$$

A quantity of great importance

Backpropagation – for outermost layer

$$\delta j = -\frac{\delta E}{\delta net_j} = -\frac{\delta E}{\delta o_j} \times \frac{\delta o_j}{\delta net_j} (net_j = \text{input at the j}^{th} \text{ layer})$$

$$E = \frac{1}{2}\sum_{i=1}^{N}(t_j - o_j)^2$$

$$\text{Hence, } \delta j = -(-(t_j - o_j)o_j(1 - o_j))$$

$$\Delta w_{ji} = \eta(t_j - o_j)o_j(1 - o_j)o_i$$

Observations from Δw_{ji}

$$\Delta w_{ji} = \eta(t_j - o_j)o_j(1-o_j)o_i$$

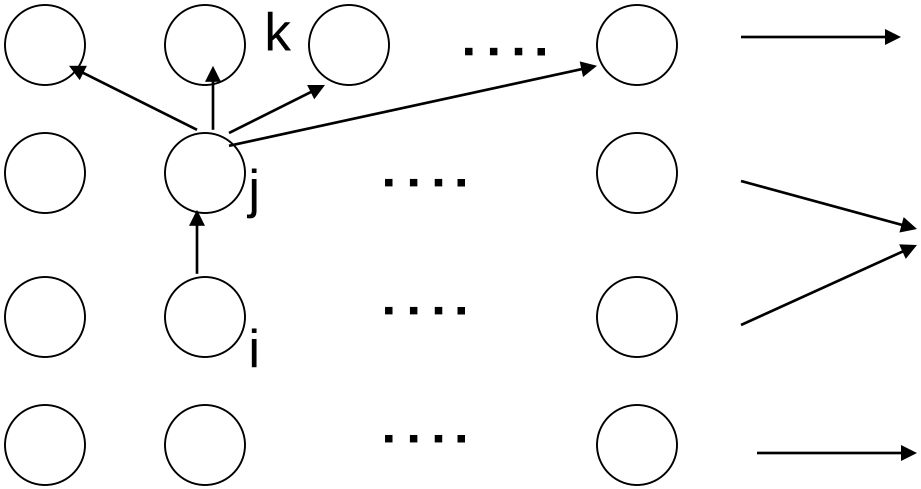
$$\Delta w_{ji} \to 0 \quad \text{if,}$$

$$1.\, o_j \to t_j \quad \text{and/or}$$

$$\left.\begin{array}{l} 2.\, o_j \to 1 \quad \text{and/or} \\ 3.\, o_j \to 0 \quad \text{and/or} \end{array}\right\} \text{Saturation behaviour}$$

$$\left.4.\, o_i \to 0 \quad \right\} \text{Credit/Blame assignment}$$

Backpropagation for hidden layers



Output layer
(m o/p neurons)

Hidden layers

Input layer
(n i/p neurons)

δ_k is propagated backwards to find value of δ_j

Backpropagation – for hidden layers

$$\Delta w_{ji} = \eta \delta j o_i$$

$$\delta j = -\frac{\delta E}{\delta net_j} = -\frac{\delta E}{\delta o_j} \times \frac{\delta o_j}{\delta net_j}$$

$$= -\frac{\delta E}{\delta o_j} \times o_j (1 - o_j)$$

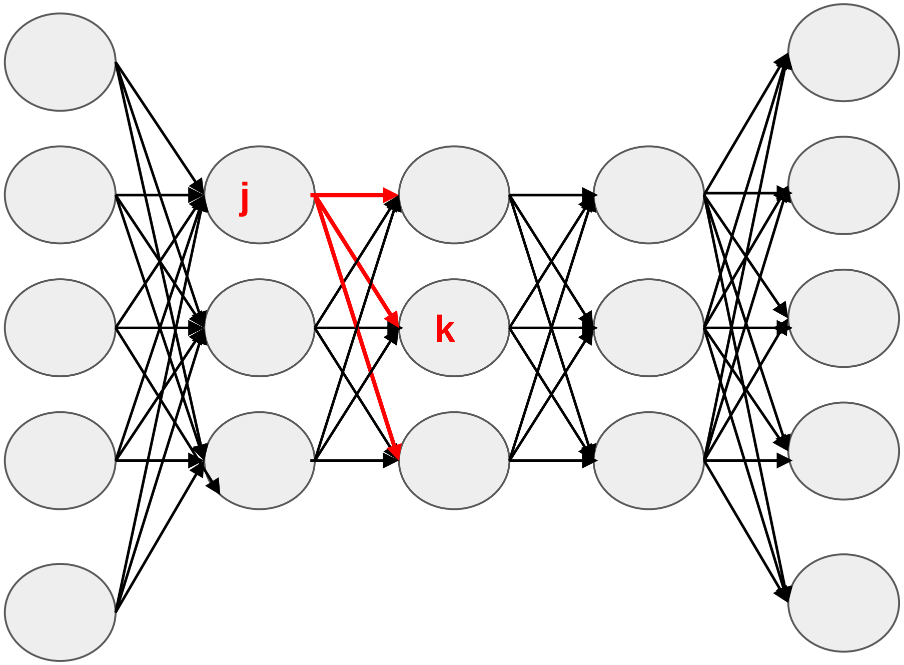
$$= -\sum_{k \in \text{next layer}} \left(\frac{\delta E}{\delta net_k} \times \frac{\delta net_k}{\delta o_j}\right) \times o_j (1 - o_j)$$

This recursion can give rise to vanishing and exploding Gradient problem

$$\text{Hence, } \delta_j = -\sum_{k \in \text{next layer}} (-\delta_k \times w_{kj}) \times o_j (1 - o_j)$$

$$= \sum_{k \in \text{next layer}} (w_{kj} \delta_k) o_j (1 - o_j)$$

Back-propagation- for hidden layers: Impact on net input on a neuron



- O_j affects the net input coming to all the neurons in next layer

General Backpropagation Rule

- General weight updating rule:

$$\Delta w_{ji} = \eta \delta j o_i$$

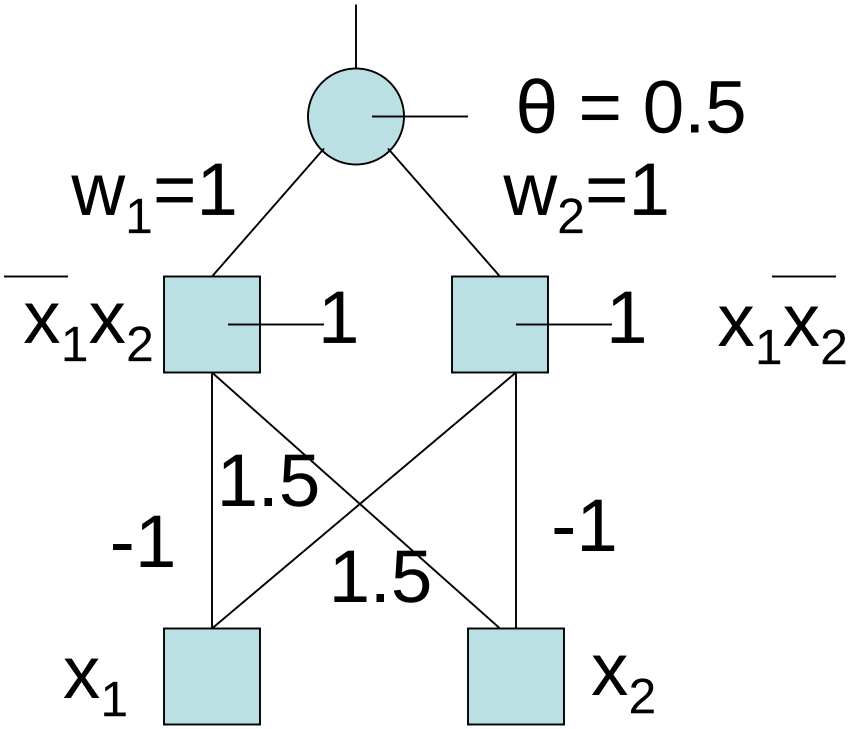
- Where

$$\delta_j = (t_j - o_j) o_j (1 - o_j) \quad \text{for outermost layer}$$

$$= \sum_{k \in \text{next layer}} (w_{kj} \delta_k) o_j (1 - o_j) o_i \quad \text{for hidden layers}$$

How does it work?

Input propagation forward and error propagation backward (e.g. XOR)



Optional Assignment

- Implement your OWN BP on XOR
- Observe what the hidden layer neurons compute