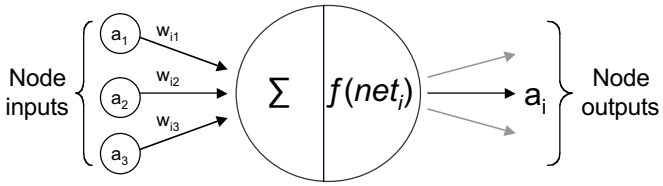
Connectionist and Statistical Language Processing

Lecture 2: Learning and Training

Matthew W Crocker

Computerlinguistik
Universität des Saarlandes

Basic Structure of Nodes

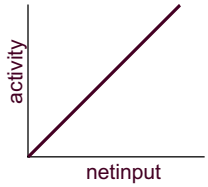


- A node can be characterised as follows:
  - Input connections representing the flow of activation from other nodes or some external source
  - Each input connection has its own *weight*, which determines how much influence that input has on the node
  - A node *i* has an output activation $a_i = f(net_i)$ which is a function of the weighted sum of its input activations, *net*.

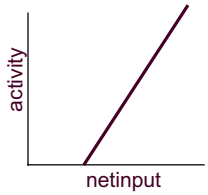
- The net input is determined as follows: $net_i = \sum_j w_{ij} a_j$

Activation functions

- The activation function determines the activation a_i for node *i* from the net input (net_i) to the node: f(net_i)
- Linear activation function
  - (McCulloch-Pitts neurode, perceptron)
  - Identity: the $a_i = net_i$

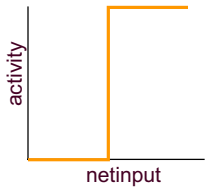


- Threshold activation function:
  - IF $net_i > T$ THEN $a_i := net_i - T$
  - ELSE $a_i := 0$



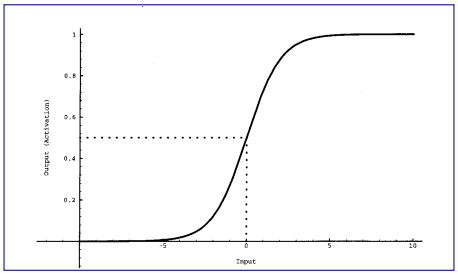
More Activation Functions

- Binary threshold activation function:
  - IF $net_i > T$ THEN $a_i := 1$
  - ELSE $a_i := 0$



- Non-linear activation function
  - It is often more useful to use the "sigmoidal" logistic function:

$$a_i = f(net_i) = \frac{1}{1 + e^{net_i}}$$



Calculating the activation: net_i is 1.25

- Linear activation:

$$f(net_i) = net_i$$
$$f(1.25) = 1.25$$

- Linear threshold: T=0.5

$$\text{IF } net_i > T \text{ then } f(net_i) = net_i - T$$
$$\text{ELSE } f(net_i) = 0$$
$$f(1.25) = 1.25 - 0.5 = 0.75$$

- Binary threshold: T=0.5

$$\text{IF } net_i > T \text{ then } f(net_i) = 1$$
$$\text{ELSE } f(net_i) = 0$$
$$f(1.25) = 1$$

- Non-linear activation:
  - Sigmoid or „logistic“ function

$$f(net_i) = \frac{1}{1 + e^{net_i}}$$
$$f(1.25) = 0.777$$

About activation functions

- The activation function defines the relationship between the net input to a node, and its activation level (which is also its output).
- Neurons in the brain have thresholds, only fire with sufficient net input.
- Non-linearity (i.e.where low input lead to zero activation) can be useful to reduce the effects of spurious inputs, noise.
- Most common in connectionist modelling: sigmoid/logistic
  - Activation ranges between 0 and 1
  - Rate of activatation increase is highest for net inputs around 0
  - Models neurons by implementing thresholding, a maximum activity, and smooth transition between states.
- The sigmoid function also has nice mathematical properties

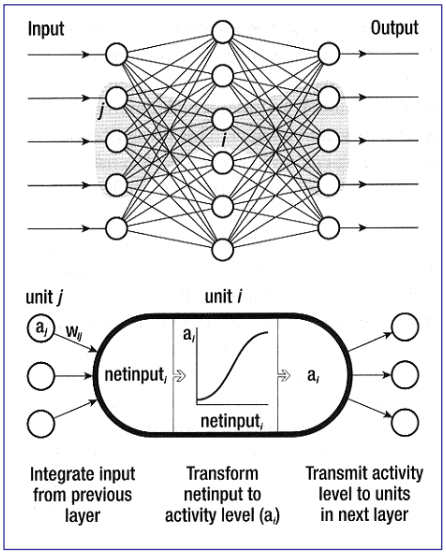
Summary of network architecture

- The activation of a unit *i* is represented by the symbol a_i.
- The extent to which unit *j* influences unit i is determined by the weight w_{ij}
- The input from unit *j* to unit *i* is the product: $a_j * w_{ij}$
- For a node *i* in the network:

$$netinput_i = \sum_j w_{ij} a_j$$

- The output activation of node *i* is determined by the activation function, e.g. the logistic:

$$a_i = f(netinput_i) = \frac{1}{1+e^{net_i}}$$



Learning in connectionist networks

- **Supervised learning** in connectionist networks involves successively adjusting connection weights to reduce the discrepancy between the *actual output activation* and the *correct output activation*
  - An input is presented to the network
  - Activations are propagated through the network to its output
  - Outputs are compared to "correct" outputs: difference is called *error*
  - Weights are adjusted
- The Delta Rule:

$$\Delta w_{ij} = [a_i(\text{desired}) - a_i(\text{obtained})] a_j \varepsilon$$

  - [a_i(desired)-a_i(obtained] is the difference between the desired output activation and the actual activation produced by the network
    - What is the "error"?
  - a_j is the activity of the contributing unit *j*
    - How much activation is this unit responsible for?
  - ε is the learning rate parameter.
    - How rapidly do we want to make changes?

Training the Network

- Consider the AND function
  - Present stimulus: 0 0
  - Compute output activation
  - Compared with desired output (0)
  - Use Delta rule to change weights
  - Present next stimulus: 0 1
  - ...

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

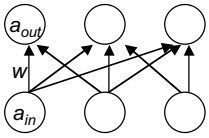
- An Epoch, consists of a single presentation of all training examples
  - Here there are 4 such examples
- A Sweep, is a presentation of a single training example
  - So, 250 epochs consists of 1000 sweeps

Summary

- Connectionism is inspired by information processing in the brain
- Models typically contain several layers of processing units
  - Units correspond to a neuron (or group of neurons)
  - Units sum weighted inputs from previous layers, and compute activation
  - Output activation is passed to units of the next layer
- An input stimulus causes a “patter of activation” on the first layer
  - Activations are then propagated through the network
  - The influence of one unit upon another is determined by the weight
  - The output response is the “pattern of activation” on the final layer
- Learning aims to reduce the discrepancy between actual and desired output patterns of activation
  - The Delta rule iteratively changes the weights of successive epochs
  - Training is complete when error is sufficiently reduced

"Perceptrons" [Rosenblatt 1958]

- Perceptron: a simple, one-layer, feed-forward network:



$$\text{netinput}_{out} = \sum_{in} w \cdot a_{in}$$

- Binary threshold activation function:

$$a_{out} = 1 \text{ if netinput}_{out} > \theta$$
$$= 0 \text{ otherwise}$$

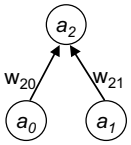
- Learning: the perceptron convergence rule
  - Two parameters can be adjusted:
    - The threshold
    - The weights

$$\text{The error, } \delta = (t_{out} - a_{out})$$
$$\Delta\theta = -\varepsilon\delta$$
$$\Delta w = \varepsilon\delta a_{in}$$

Learning OR

- Consider the following simple perceptron:
  - Recall the convergence rule:

$$\text{The error, } \delta = (t_{out} - a_{out})$$
$$\Delta\theta = -\varepsilon\delta$$
$$\Delta w = \varepsilon\delta a_{in}$$



Classification problem

a_0	a_1	a_2
0	0	0
0	1	1
1	0	1
1	1	1

- We want to train this to learn boolean OR:
  - Note: changes have opposite signs
    - E.g if activity is less than target, ∂ is positive
      - Threshold is decreased
      - Weight is increased
  - If ∂ is non-zero, threshold is always changed
    - But if a_{in} is zero, the weight is not changed
  - The changes can be calculated straight-forwardly, but do they lead to convergence on a solution to a problem?

Learning OR continued …

- Recall the convergence rule:

The error, $\delta = (t_{out} - a_{out})$

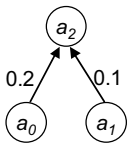
$\Delta\theta = -\varepsilon\delta$

$\Delta w = \varepsilon\delta a_{in}$

And the net:

$\theta = 1$

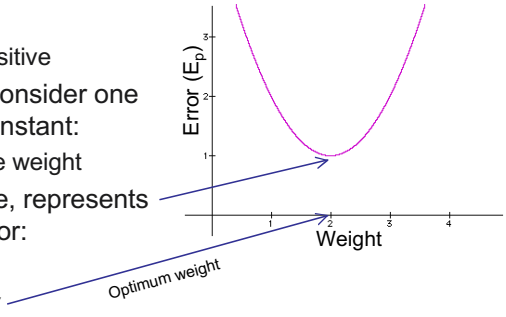
$\varepsilon = 0.5$



In	w_{20}	w_{21}	θ	a_2	t_2	δ	$\Delta\theta$	Δw_{20}	Δw_{21}
0 0	.2	.1	1.0	0	0	0	0	0	0
1 0	.2	.1	1.0	0	1	1.0	-0.5	0.5	0
0 1	.7	.1	0.5	0	1	1.0	-0.5	0	0.5
1 1	.7	.6	0.0	1	1	0	0	0	0
0 0	.7	.6	0.0	0	0	0	0	0	0
0 1	.7	.6	0.0	1	1	0	0	0	0
1 0	.7	.6	0.0	1	1	0	0	0	0
1 1	.7	.6	0.0	1	1	0	0	0	0

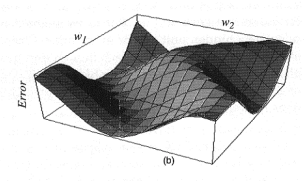
Gradient descent

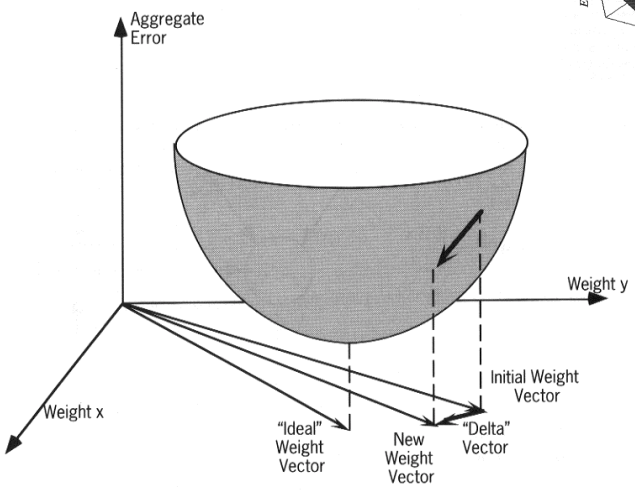
- Let's define the error on the outputs as: $E_p=(t_{out}-a_{out})^2$
  - Recall: $a_{out} = \sum w\, a_{in}$
  - This means E_p is always positive
- For a single layer net, if we consider one weight, holding the others constant:
  - Plot Error versus varying the weight
- The lowest point on the curve, represents the minimum error possible for:
  - For pattern *p*
  - By varying a given weight *w*



- Learning: the network is always at some point on the error curve
  - Use the *slope of the curve* to change the weights in the right direction
  - If slope is positive, then decrease the weight
  - If slope is negative, increase the weight

Visualising the error „surface“





Gradient descent continued

- We need calculus to allow us to determine how the error varies when a particular weight is varied:

$$\Delta w = -\varepsilon \frac{\partial E}{\partial w}$$

Slope: Rate of change of *E*, with *w*

$$\Delta w = -\varepsilon \frac{\partial (t_{out} - a_{out})^2}{\partial w}$$

Error = $(t_{out} - a_{out})^2$

$$\Delta w = -\varepsilon \frac{\partial [t_{out} - F(\sum_{in} w \cdot a_{in})]^2}{\partial w}$$

Differential of the activation function with respect to w, i.e. its slope

$$\Delta w = 2\varepsilon [t_{out} - F(\sum_{in} w \cdot a_{in})] \cdot F'(\sum_{in} w \cdot a_{in}) \cdot a_{in}$$

$$\Delta w = 2\varepsilon \delta F^* a_{in}$$

$$\left\lfloor \begin{array}{l} \delta = (t_{out} - a_{out}) \\ F^* = \text{slope of the activation function} \end{array} \right\rfloor$$

Gradient descent and the delta rule

- The perceptron convergence rule: $\Delta w = \varepsilon \delta a_{in}$
- Our revised learning rule, based on gradient descent is:

$$\Delta w = 2\varepsilon \delta F * a_{in}$$

  - where F^* is the slope of the activation function
- If the activation function is linear, the slope is constant:

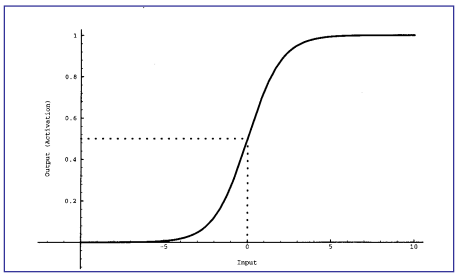
$$\Delta w = k \delta a_{in}$$

  - where k is a constant representing the learning rate and slope
- This corresponds to the original Delta rule:
  - It is straight-forward to calculate
  - Performs gradient descent to the bottom of an the error curve
  - Δw is proportional to $(t_{out}-a_{out})$, so changes get smaller as error is reduced
  - In 2-layer networks, there is a single minimum which gradient descent learning is guaranteed to find a solution, it one exists.

Learning with the Sigmoid activation function

- Networks with linear activation functions:
  - have mathematically well-defined learning capacities
  - they are known to be limited in the kinds of problems they can solve
- The logistic, or sigmoid, function is:
  - Non-linear, more powerful
  - More neurologically plausible
  - Less well-understood, more difficult to analyse mathematically
- Recall:

$$a_i = f(net_i) = \frac{1}{1+e^{net_i}}$$

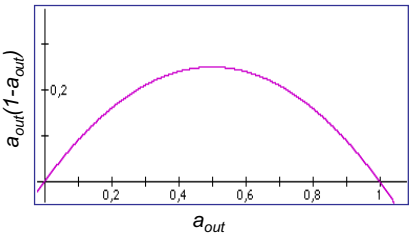


Behaviour of the logistic function

- Deriving the slope of the logistic function:

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

$$F^* = f'(net_i) = a_{out}(1-a_{out})$$

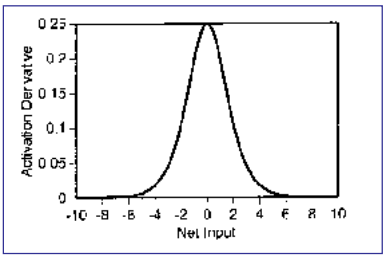


- The Delta rule, assuming the logistic function:

$$\Delta w = 2\varepsilon\delta F^* a_{in}$$

or

$$\Delta w = 2\varepsilon(t_{out} - a_{out})a_{out}(1-a_{out})a_{in}$$



Training a network

- The training phase involves
  - Presenting an input pattern, and computing the output for the network using the current connection weights: $a_{out}=f(\sum_{in} w_{out,in} \times a_{in})$
  - Calculating the error between the desired and the actual output $(t_{out} - a_{out})$
  - Using the Delta rule (appropriate for the activation function):

$$\Delta w = \eta(t_{out} - a_{out})a_{out}(1-a_{out})a_{in}$$

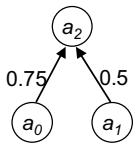
- One such cycle is called a sweep
- A sweep through each patter is called an epoch
- We can define the global error of the network, as the average error across all input patterns, *k:*
  - One common measure is the square root of mean error

$$\text{rms error} = \sqrt{\frac{\sum_k (\vec{t}_k - \vec{o}_k)^2}{k}}$$

  - Squaring avoids positive and negative error cancelling each other out

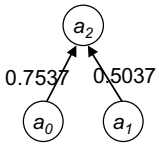
Training: an example

- Consider the simple feedforward network:
  - Assume an input pattern: 1 1
  - Assume a learning rate of 0.1
  - Assume a sigmoid activation
  - Desired output is: 1



- Determine the weight changes for 1 sweep:

$$a_2 = f(1\times 0.75 + 1\times 0.5) = 0.77$$
$$\delta_2 = (t - a_2) f'(0.77) = 0.23\times 0.16 = 0.037$$
$$\Delta w_{20} = \eta \delta_2 o_0 = 0.1\times 0.037\times 1 = 0.0037$$
$$\Delta w_{21} = \eta \delta_2 o_1 = 0.1\times 0.037\times 1 = 0.0037$$



The dynamics of weight changes

- Learning rate: predetermined constant
- The error: large error = large weight change
- The slope of the activation function:
  - The derivative of the logistic is largest for netinputs around 0, and for activations around .5
  - Small netinputs co-occur with small weights
  - Small weights tend to occur early in training
  - The result: bigger changes during early stages of learning
    - More resilience in older network: harder to teach new tricks!
- The momentum:
  - This parameter determines how much of the previous weight change affects the current weight change

Calculating Error

- Consider a simple network for learning the AND operation
- After training (1000 sweeps, 250 epochs), we can calculate the global (RMS) error as follows:

Input	Target	Output	(t-o)^2
0 0	0	0,147	0,022
0 1	0	0,297	0,088
1 0	0	0,334	0,112
1 1	1	0,552	0,201
		RMS:	0,325

- Observe how error steadily falls during training

Summary

- Learning rules:
  - Perceptron convergence rule
  - Delta rule
    - Depends on the (slope of the) activation function
  - For 2-layer networks using these rules:
    - A solution will be found, if it exists
  - How do we know if network has learned successfully?
- Error:
  - For learning, we use $(t_{out} - a_{out})$ to change weights
  - To characterise the performance of the network as a whole, we need a measure of global error:
    - Across all outputs
    - Across all training patterns
  - One possible measure is RMS
    - Another is entropy: doesn't really matter, since we only need to know if performance is improving or deteriorating on a relative basis