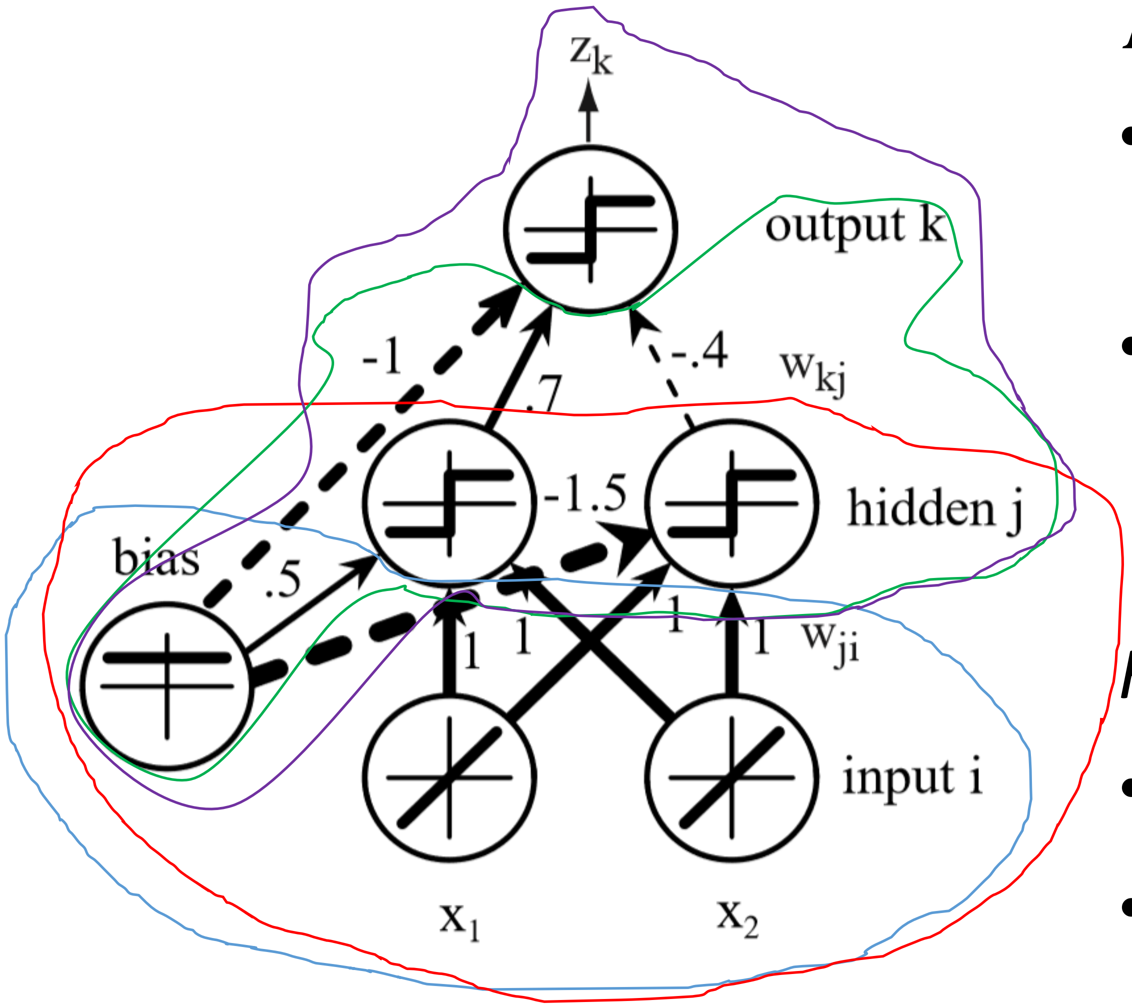
Multilayer Neural Networks

Pattern Classification by Richard O. Duda, Peter E . Hart and David G. Stork
Chapter 6 - Part 2 (training a network)

KERESTÉLY ÁRPÁD & MAJERCSIK LUCIANA 28.05.2019

Previously on Neural Networks …



Input to hidden layer:

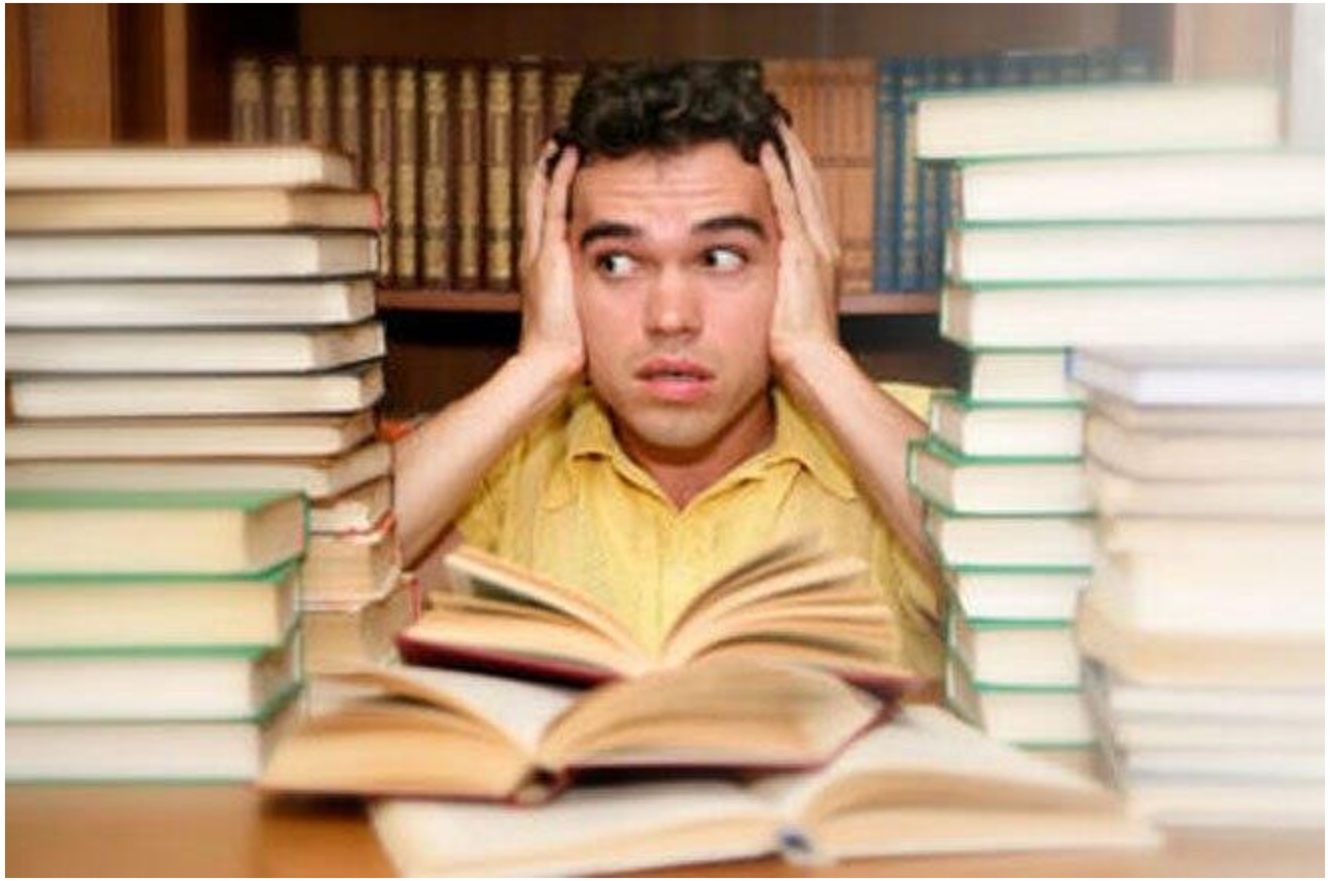
- $net_j = \sum_{i=1}^{d}\left(x_i w_{ji} + w_{j0}\right)$
  $net_j = \sum_{i=0}^{d}\left(x_i w_{ji}\right) = w_j^t x$
- $y_j = f(net_j)$

Hidden to output layer:

- $net_k = \sum_{j=0}^{n_h}\left(y_j w_{kj}\right) = w_k^t y$
- $z_k = f(net_k)$

- Did you notice, the weights in the previous example were given?
- In real life we don’t have this luxury ☺
- We need to find the proper weights somehow, more precisely, we need to make the network learn (i.e. find) them

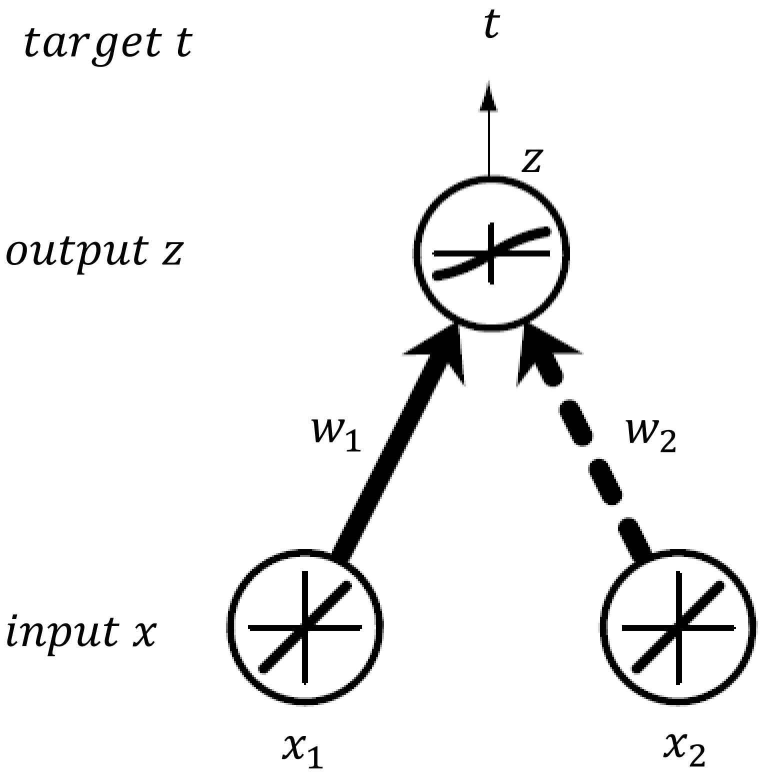
How exactly do they “learn” stuff?



How exactly do they “learn” stuff?

- In the same way that we **learn** from experience in our lives, neural networks require **data** to learn.
- In most cases, the more data that can be thrown at a neural network, the more accurate it will become.
- Think of it like any task you do over and over. Over time, you gradually get more efficient and make fewer mistakes.

How exactly do they “learn” stuff?



- We look for an algorithm which lets us find weights and biases, so that the output from the network, z (or z_k in the case of multiple outputs), approximates the real values, t (or t_k), for all training inputs.

- As with any approximation, to quantify how well we're achieving this goal we need to define a *cost function.*

- The cost function measures how far the overall results are from the truth.

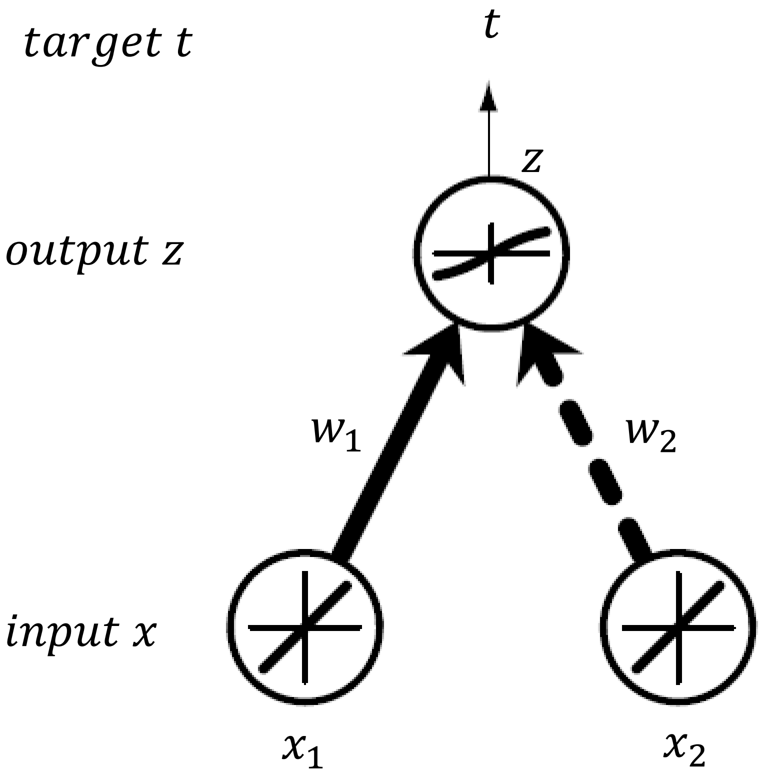
Backpropagation

- One of the simplest and most general methods for supervised training of multilayer neural networks is Backpropagation

- Other methods may be faster or have other desirable properties, but few are more instructive

- Backpropagation is the natural extension of the LMS (Least Mean Squares) algorithm for linear systems

Least Mean Squares



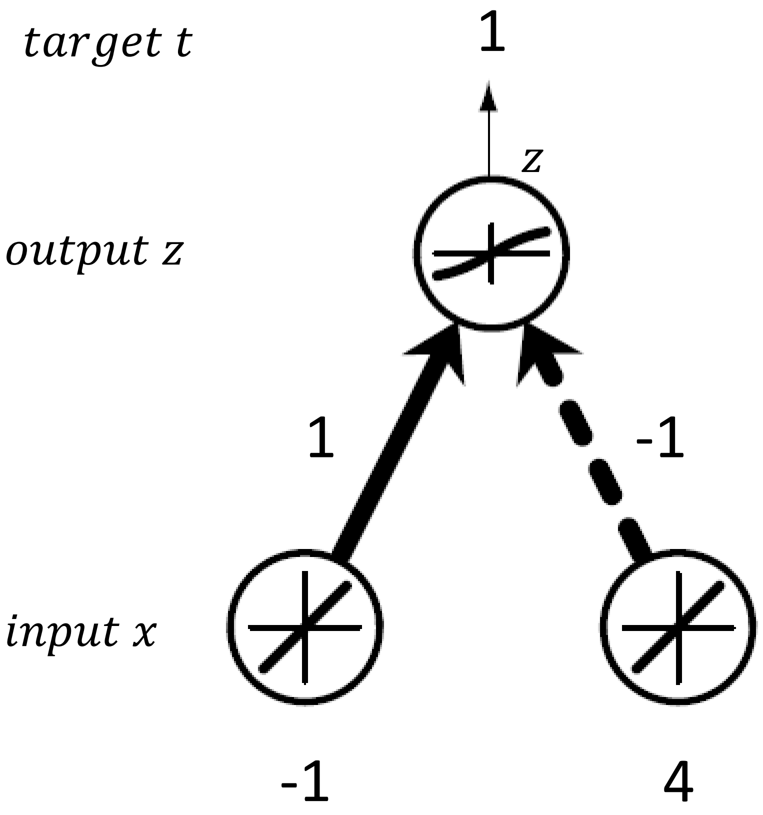
- The LMS algorithm works for two-layer systems because we have an error evaluated at the output unit
- For one sample from the training set, we have:

$$LMS\colon\ J(w_1, w_2) = (t - z)^2$$

Let’s have some fun!

With a nice little example.

The network

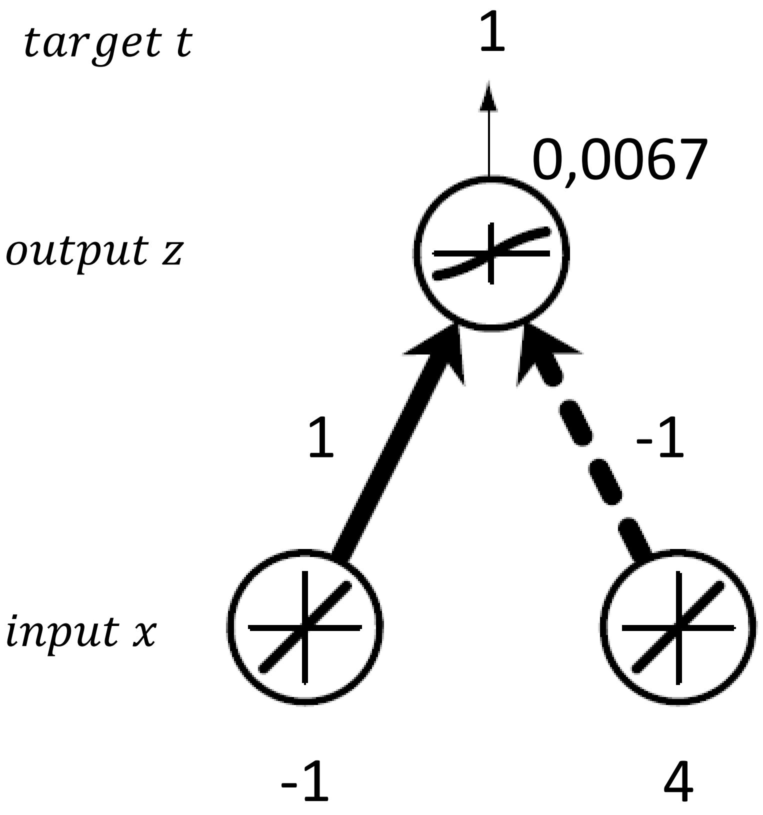


←The desired output

←Some random weights

←A sample form the training set

Feed-forward

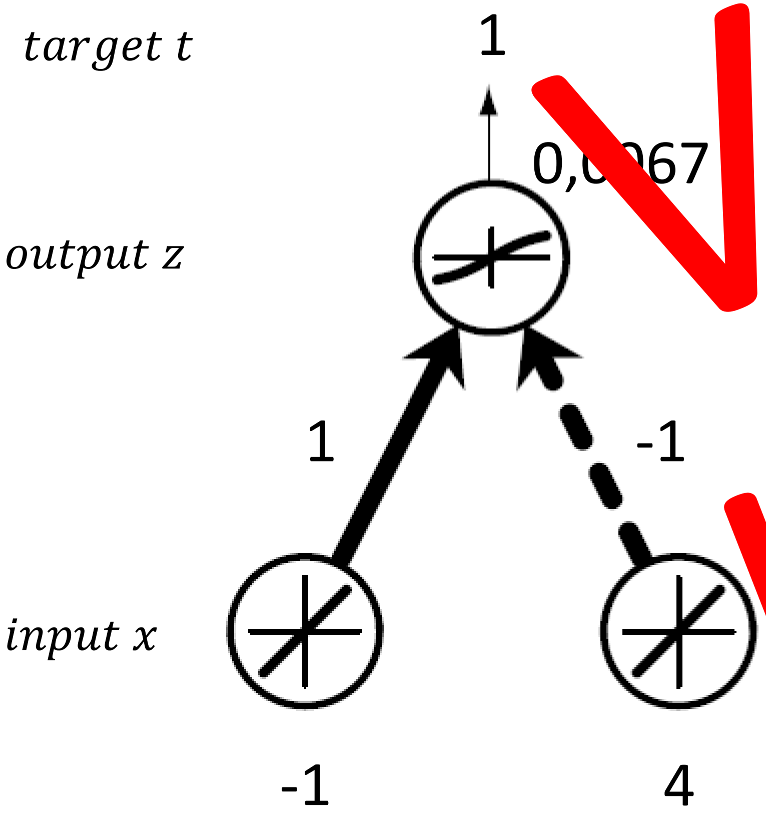


$$z = f(net), \quad f(x) = \frac{1}{1+e^{-x}}, \quad net = \sum_{i=1}^{2} x_i w_i$$

$$net = -1 \cdot 1 + 4 \cdot (-1) = -5$$

$$z = f(net) = \frac{1}{1+e^{-(-5)}} = 0.0067$$

How good is this?



$$J = LMS = (t - z)^2$$

$$J = (1 - 0.0067)^2 = (-0.9933)^2 = 0.9866$$

- J is the cost function
- In the case of LMS it's always positive
- And in this case is …

Very bad!

Getting better …

- We need to change (adjust) the weights
- By choosing different weights and seeing which gives the smallest error
- This could go forever …
- But the key to a better solution lies hidden in the previous solution
- **smallest error**
- Which means finding the minimum of the cost function J
- Which in turn depends on the two weights w_1 and w_2
- So we need the gradient of $J(w_1, w_2)$

$$\nabla J(w_1, w_2) = \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2} \right)$$

How to calculate the derivative with respect to $\boldsymbol{w_i}$

$$\frac{\partial J}{\partial w_i} = ?$$

Because J does not directly depend on w_i:

$$J = (t - z)^2 \rightarrow z = f(net) \rightarrow net = w_1 x_1 + w_2 x_2$$

We apply the chain rule for differentiation:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial z} \frac{\partial z}{\partial net} \frac{\partial net}{\partial w_i}$$

$$\frac{\partial J}{\partial w_i} = -2(t - z) f'(net) x_i$$

The perfect solution

- Theoretically, we could find the best weights by making the gradient equal to 0
- But the calculations get harder when increasing the number of weights

$$\frac{\partial J}{\partial w_i} = -2(t-z)f'(net)x_i = 0$$

$$z = f(net) = \frac{1}{1+e^{-net}} \Rightarrow f'(net) = \frac{e^{-net}}{(1+e^{-net})^2}$$

$$\frac{\partial J}{\partial w_i} = -2(t - \frac{1}{1+e^{-net}})\frac{e^{-net}}{(1+e^{-net})^2}x_i = 0$$

$$x_1 = -1, x_2 = 4, t = 1 \Rightarrow net = w_1 - 4w_2$$

$$(1 - \frac{1}{1+e^{w_1-4w_2}})\frac{e^{w_1-4w_2}}{(1+e^{w_1-4w_2})^2} = 0$$

What is the solution?

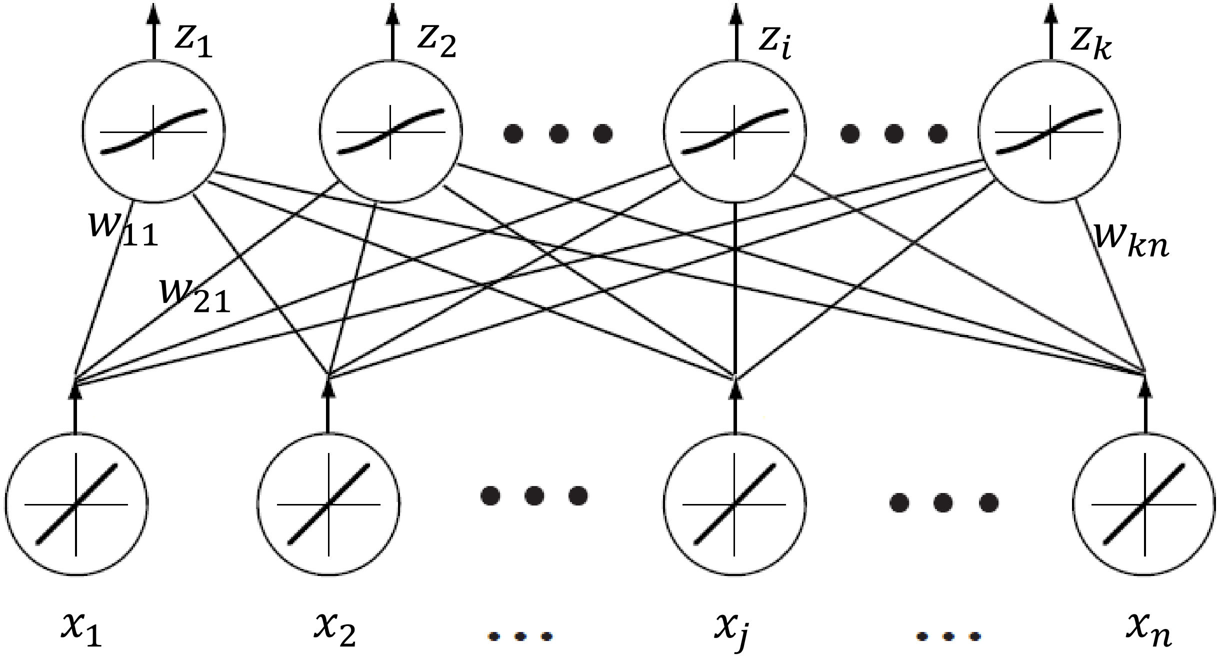
An approximation

- But we can be happy with some approximation of the best weights also, in which case we will move by a small portion of the gradient towards the minimum throughout several iterations

$$\Delta w_i = -\eta \frac{\partial J}{w_i}$$

Multiple outputs

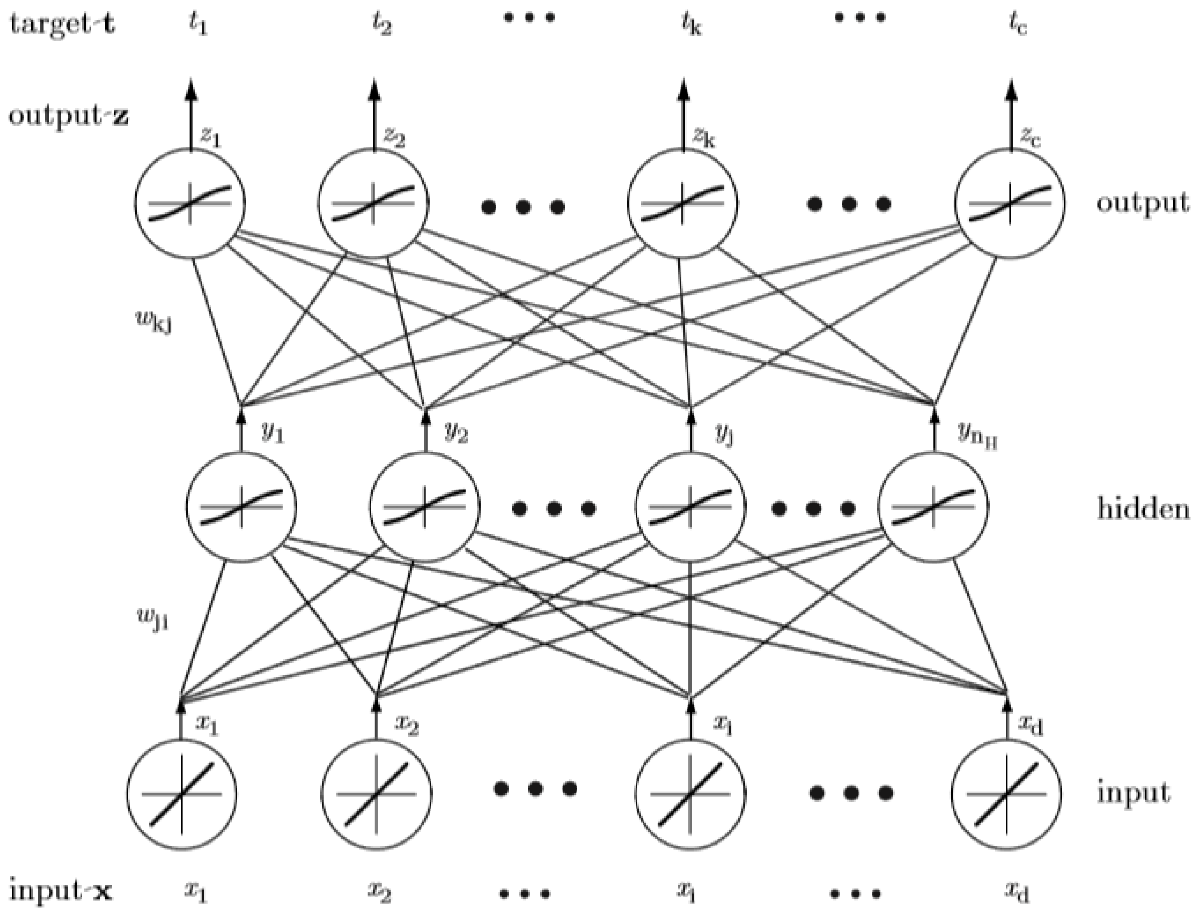
- In the case of multiple outputs, the calculation of the error function changes to the following form



$$J(w_1 \ldots w_{kn}) = \sum_{i=0}^{k} (t_i - z_i)^2$$

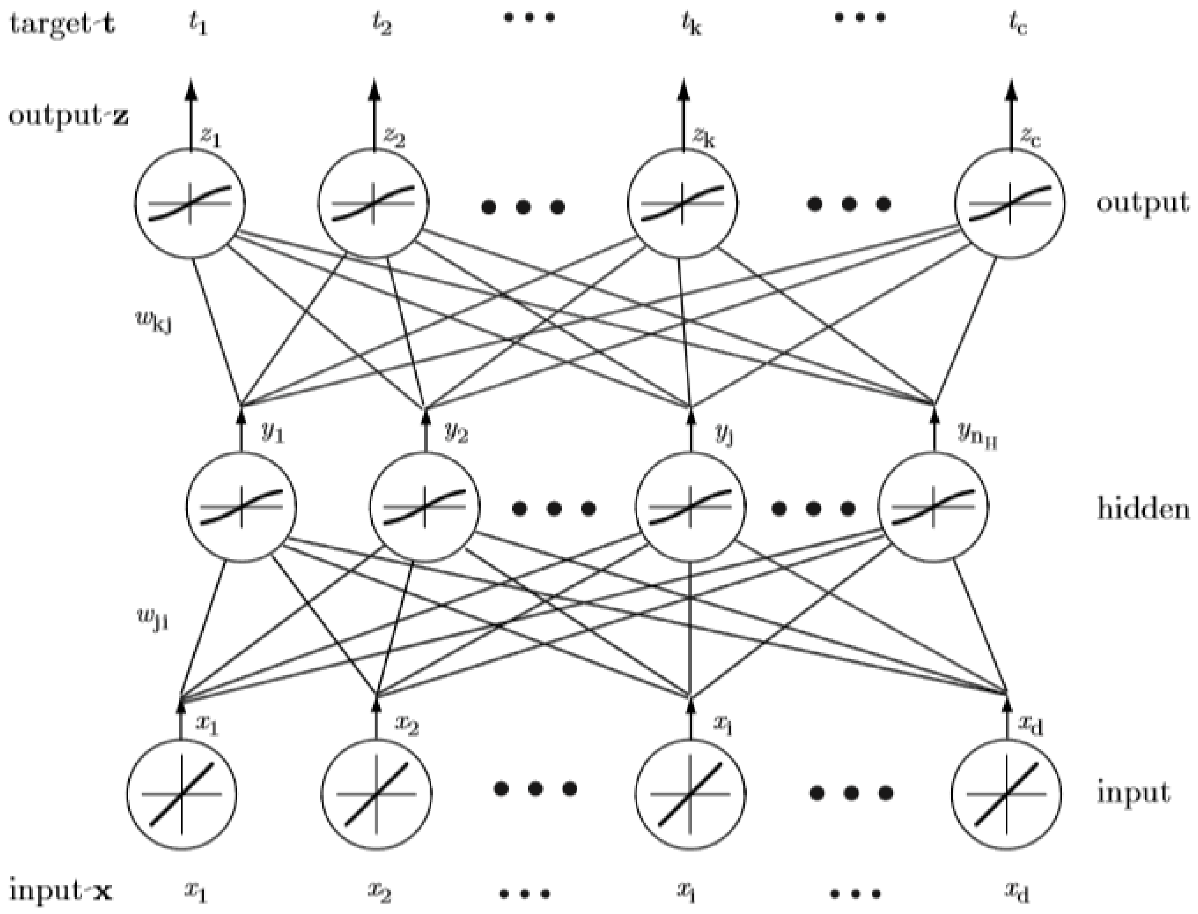
Where k is the number of outputs and n is the number of inputs

The case of a hidden layer



- What if the network has three layers (one hidden layer)?
- Then we have two sets of weights

Notation for weights



- We used w_{ji} to denote the weight between input unit i and hidden unit j
- We used w_{kj} to denote the weight between hidden unit j and output unit k

Getting better with 3 layered networks

- Works in a similar fashion to the 2 layered networks seen previously; modify the weights so the error gets as small as possible (i.e. apply the gradient descendent rule)
- Modifying the weights is done in two steps
- Calculating the change in the weights between the **hidden** and **output** layer is done by the same formula that we used for two layered networks
- But when calculating the change in the weights between the **input** and **hidden** layer, special care must be taken

How to calculate the error change with respect to $\boldsymbol{w_{kj}}$

- i.e. weights from hidden to output

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}}$$

$$= -2(t_k - z_k)\, f'(net_k)\, y_j$$

- We denote with $\boldsymbol{\delta_k}$ the sensitivity of unit $\boldsymbol{k}$

$$\delta_k = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k}$$

$$= 2(t_k - z_k)\, f'(net_k)$$

- The gradient descent (learning rule) for hidden to output weights

$$\Delta w_{kj} = -\eta \frac{\partial J}{\partial w_{kj}}$$

$$= \eta \delta_k y_j$$

$$= 2\eta(t_k - z_k)\, f'(net_k)\, y_j$$

How to calculate the error change with respect to $\boldsymbol{w_{ji}}$

- i.e. weights from input to hidden

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

$$\frac{\partial J}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\sum_{k=1}^{c}(t_k - z_k)^2\right]$$

$$= -2\sum_{k=1}^{c}(t_k - z_k)\ \frac{\partial z_k}{\partial y_j}$$

$$= -2\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}$$

$$= -2\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{kj}$$

- We denote with $\boldsymbol{\delta_j}$ the sensitivity of unit $\boldsymbol{j}$

$$\delta_j = f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

- The gradient descent (learning rule) for input to hidden weights

$$\Delta w_{ji} = -\eta\frac{\partial J}{\partial w_{ji}}$$

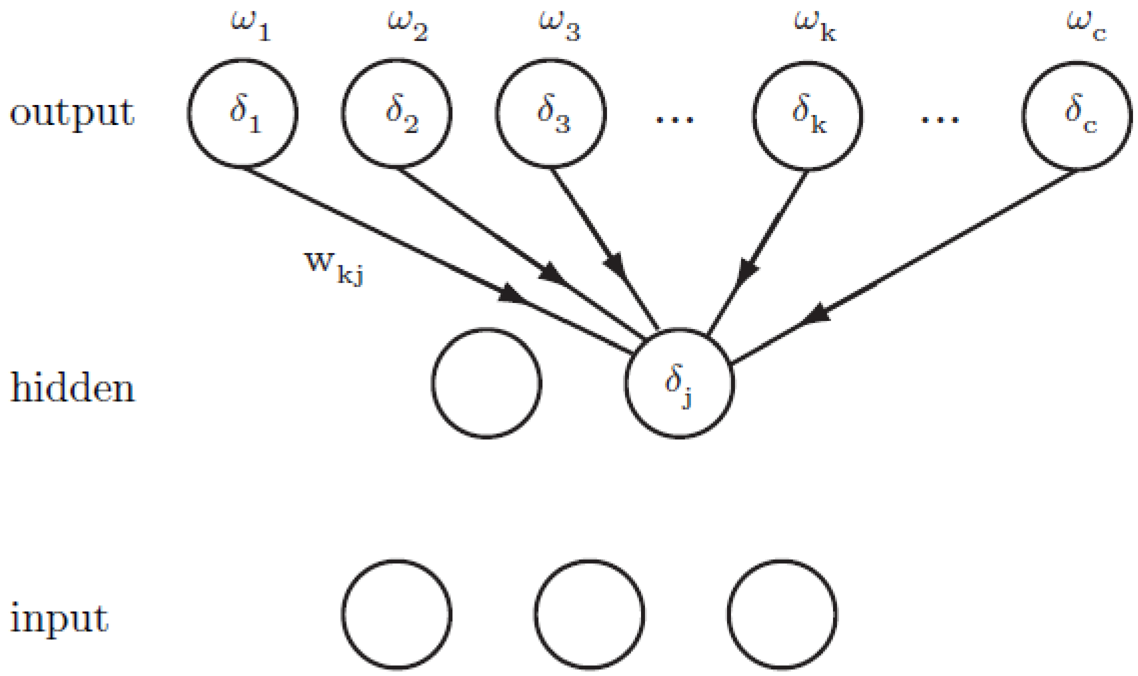
$$= \eta\delta_j x_i$$

$$= \eta\, x_i\, f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

What this really means?

- On the way back, the “backpropagation” is finding how much each weight is contributing to the overall “error”.

- The weights that contribute more to the overall “error” will have larger derivation values, which means that they will change more (when computing Gradient descent).



Learning from multiple samples

- With the formulas seen until now, the network can learn a single sample

- To learn from multiple samples, thus get better at generalizing, the network needs to adopt a *training protocol*

Training protocols

- There are a variety of training protocols
- The three most useful being:
  - Stochastic (or pattern training)
  - Batch
  - On-line
- One very interesting:
  - Learning with queries

- Most training protocols will need several ***epochs*** (number of presentations of the full training set) to learn

Stochastic training protocol

- Patterns are chosen randomly from the training set, and the network weights are updated for each pattern presentation
- Is called stochastic because the training data can be considered a random variable

Algorithm 1 (Stochastic backpropagation)

begin initialize network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, m \leftarrow 0$
    do $m \leftarrow m + 1$
        $\mathbf{x}^m \leftarrow$ randomly chosen pattern
        $w_{ij} \leftarrow w_{ij} + \eta \delta_j x_i; \quad w_{jk} \leftarrow w_{jk} + \eta \delta_k y_j$
    until $\nabla J(\mathrm{w}) < \theta$
return $\mathbf{w}$
end

On-line training protocol

- Each pattern is presented once and only once
- There is no memory for storing the patterns
  - Some on-line training algorithms are considered models of biological learning, where the organism is exposed to the environment and cannot store all input patterns for multiple "presentations."

Algorithm 1.1(On-line backpropagation)

<u>**begin**</u> <u>**initialize**</u> network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, m \leftarrow 0$
    <u>**do**</u> $m \leftarrow m + 1$
        $\mathbf{x}^m \leftarrow$ sequenctially choosen pattern
        $w_{ij} \leftarrow w_{ij} + \eta\delta_j x_i; \quad w_{jk} \leftarrow w_{jk} + \eta\delta_k y_j$
    <u>**until**</u> $\nabla J(\mathbf{w}) < \theta$
<u>**return**</u> $\mathbf{w}$
<u>**end**</u>

Batch training protocol

- All patterns are presented to the network before learning (weight update) takes place
- The network needs to see all the patterns multiple times to be able to minimize the error function

Algorithm 2 (Batch backpropagation)

<u>**begin**</u> <u>**initialize**</u> network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, r \leftarrow 0$
    <u>**do**</u> $r \leftarrow r + 1$ (increment epoch)
        $m \leftarrow 0;\ \Delta w_{ij} \leftarrow 0;\ \Delta w_{jk} \leftarrow 0$
        <u>**do**</u> $m \leftarrow m + 1$
            $\mathbf{x}^m \leftarrow$ select pattern
            $\Delta w_{ij} \leftarrow \Delta w_{ij} + \eta \delta_j x_i;\quad \Delta w_{jk} \leftarrow \Delta w_{jk} + \eta \delta_k y_j$
        <u>**until**</u> $m = n$
        $w_{ij} \leftarrow w_{ij} + \Delta w_{ij};\quad w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$
    <u>**until**</u> $\nabla J(\mathbf{w}) < \theta$
<u>**return**</u> $\mathbf{w}$
<u>**end**</u>

Learning with queries training protocol

- The output of the network is used to select new training patterns
- Such queries generally focus on points that are likely to give the most information to the classifier
- While this protocol maybe faster in many cases, its drawback is that the training samples are no longer independent, identically distributed
- This, in turn, generally distorts the effective distributions and may or may not improve recognition accuracy

In conclusion, what is Backpropagation?

- Equations:

$$\Delta w_{kj} = -\eta \frac{\partial J}{\partial w_{kj}} = \eta \delta_k y_j = 2\eta\, y_j\, f'(net_k)(t_k - z_k)$$

$$\Delta w_{ji} = -\eta \frac{\partial J}{\partial w_{ji}} = \eta \delta_j x_i = \eta\, x_i\, f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k$$

- together with training protocols such as described before, give the backpropagation algorithm or more specifically the “backpropagation of errors” algorithm

Practical Tips for Improving BP

- When creating a multilayer neural network classifier, the designers must make two major types of decision:
  - selection of the network architecture
  - selection of the network parameters

- While parameter adjustment is problem dependent, there are several rules of that emerged from an analysis of networks.

Practical Tips for Improving BP: Initializing weights

- Do not set the values of either w_{ji} or w_{kj} to 0. Setting the values of w_{kj} to 0 will lead to no update in the w_{ji} weights

- As a rule for the sigmoid function: choose random weights from the range

$$\frac{-1}{\sqrt{d}} < w_{ji} < \frac{1}{\sqrt{d}}$$

$$\frac{-1}{\sqrt{n_H}} < w_{kj} < \frac{1}{\sqrt{n_H}}$$

Practical Tips for Improving BP: Learning rate

- As any gradient descent algorithm, backpropagation depends on the learning rate η
- Suggestion: take the initial value $\eta = 0.1$
- However we can adjust η at the training time. Based on what? The objective function J should decrease during gradient descent
- If the values of J are oscillating, η is too large, we have to decrease it
- If the values of J are going down but very slowly, η is too small, so we have to increase it

Practical Tips for Improving BP: No. of hidden layers

- Networks with 1 hidden layer have the same expressive power as those with several hidden layers
- For some applications, having more than 1 hidden layer may result in faster learning and less hidden units overall
- However networks with more than 1 hidden layer are more prone to the local minima problem

Practical Tips for Improving BP: No. of hidden units

- The number of input units = number of features,
- The number of output units = number of classes.

- The, how to choose n_H, the number of hidden units?

- n_H determines the expressive power of the network
  - Too small n_H may not be sufficient to learn complex decision boundaries
  - Too large n_H may over fit the training data resulting in poor generalization

Practical Tips for Improving BP: No. of hidden units

- Choosing the best n_H is not a solved problem

- As a practical rule :
  - if total number of training samples is n, choose n_H so that the total number of weights is n /10
  - The total number of weights = (no of w_{ji}) + (no of w_{kj})

Thank you!