Administration

- HW4 will be out next week!
- Midterm is close!
  - Come to sections with questions

Administration

Questions

- HW4 will be out next week!
- Midterm is close!
  - Come to sections with questions

Administration

- HW4 will be out next week!
- Midterm is close!
  - Come to sections with questions
- Projects proposals are due on Friday 10/16/15

Questions

Projects

- Projects proposals are due on Friday 10/16/15
- Within a week we will give you an approval to continue with your project along with comments and/or a request to modify/augment/do a different project. There will also be a mechanism for peer comments.
- We allow team projects – a team can be up to 3 people.

- Please start thinking and working on the project.
- Your proposal is limited to 1-2 pages, but needs to include references and, ideally, some of the ideas you have developed in the direction of the project (maybe even some preliminary results).
- Any project that has a significant Machine Learning component is good.
- You can do experimental work, theoretical work, a combination of both or a critical survey of results in some specialized topic.
- The work *has* to include some reading. Even if you do not do a survey, you must read (at least) two related papers or book chapters and relate your work to it.
- Originality is not mandatory but is encouraged.
- Try to make it interesting!

Neural Networks

Neural Networks

- **Robust** approach to approximating **real-valued**, **discrete-valued** and **vector valued** target functions.
- Among the most effective **general purpose** learning method currently known.

Neural Networks

- **Robust** approach to approximating **real-valued**, **discrete-valued** and **vector valued** target functions.
- Among the most effective **general purpose** learning method currently known.
- Effective especially for **complex and hard to interpret input data** such as real-world sensory data

Neural Networks

- **Robust** approach to approximating **real-valued**, **discrete-valued** and **vector valued** target functions.
- Among the most effective **general purpose** learning method currently known.
- Effective especially for **complex and hard to interpret input data** such as real-world sensory data
- The **Backpropagation algorithm** for neural networks has been shown successful in many practical problems
  - handwritten character recognition, spoken words recognition, face recognition

Neural Networks

$$\mathrm{N}: X \rightarrow Y$$

- $= [0,1]^n$, *or* $\{0,1\}^n$ *and* $Y = [0,1], \{0,1\}$

Neural Networks

- $0,1\ n\ 0,1\ 0,1\ 0,1\ 0,1\ n\ nn\ 0,1\ n$, or $\{0,1\}\ n\ \{0,1\}\ \{0,1\}\ n\ nn\ \{0,1\}\ n$ and $YY = 0,1\ 0,1\ 0,1$, $\{0,1\}$
- $N: XX \rightarrow YY$
  - where $X = [0,1]^n$, *or* $\{0,1\}^n$ *and* $Y = [0,1], \{0,1\}$
  - $= [0,1]^n$, *or* $\{0,1\}^n$ *and* $Y = [0,1], \{0,1\}$

Neural Networks

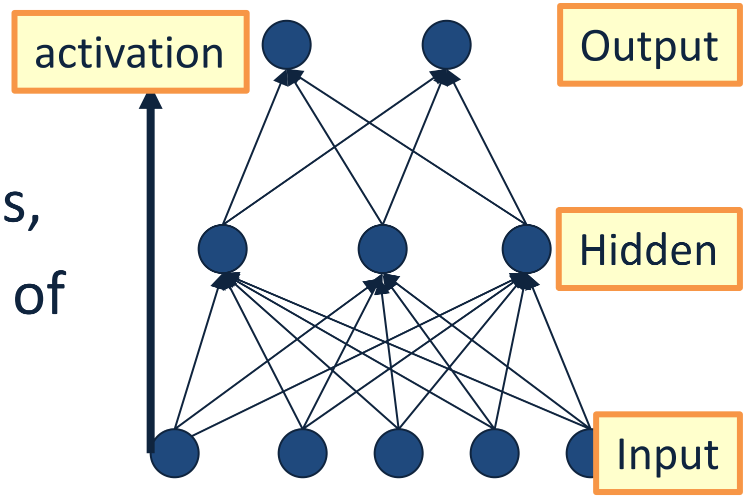
- $0,1 \ n \ 0,1 \ 0,1 \ 0,1 \ \ 0,1 \ n \ nn \ 0,1 \ n$, or $\{0,1\} \ n \ \{0,1\} \ \{0,1\} \ n \ nn \ \{0,1\} \ n$ and $YY = 0,1 \ 0,1 \ 0,1$, $\{0,1\}$
- $N: XX \rightarrow YY$
- NN can be used as an approximation of a target classifier
  - In their general form, NN can approximate any function

Neural Networks

- $0,1 \quad n \quad 0,1$ $0,1$ $0,1$ $0,1 \quad n$ nn $0,1 \quad n$, or $\{0,1\}$ n $\{0,1\}$ $\{0,1\}$ n nn $\{0,1\}$ n and $YY = 0,1$ $0,1$ $0,1$, $\{0,1\}$
- $N:XX \rightarrow YY$
- NN can be used as an approximation of a target classifier
  - In their general form, NN can approximate any function
- Algorithms exist that can learn a NN representation from labeled training data (e.g., Backpropagation).

Multi-Layer Neural Networks

- Multi-layer network were designed to overcome the computational (**expressivity**) limitation of a single threshold element.
- The idea is to **stack** several layers of threshold elements, each layer using the output of the previous layer as input.



- Multi-layer networks **can represent arbitrary functions**, but building effective learning methods for such network was [thought to be] difficult.

Motivation for Neural Networks

- Inspired by **biological systems**, the best examples we have of **robust** learning systems
  - Used to model biological systems (so we understand how they learn)
- Massive **parallelism** that may allow for computational efficiency

Motivation for Neural Networks

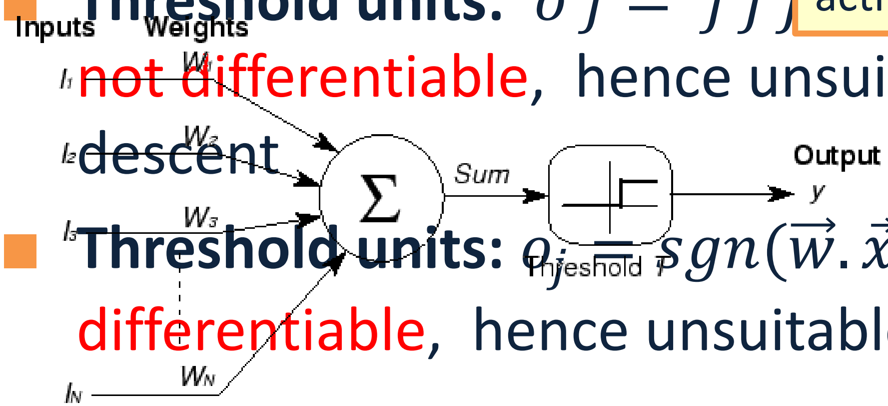
- Inspired by **biological systems**, the best examples we have of **robust** learning systems
  - Used to model biological systems (so we understand how they learn)
- Massive **parallelism** that may allow for computational efficiency
- Graceful degradation due to **distributed representation** that spread the representation of knowledge among the computational units.
- Intelligent behavior **“emerges” from large number of simple units** rather than from explicit symbolically encoded rules.

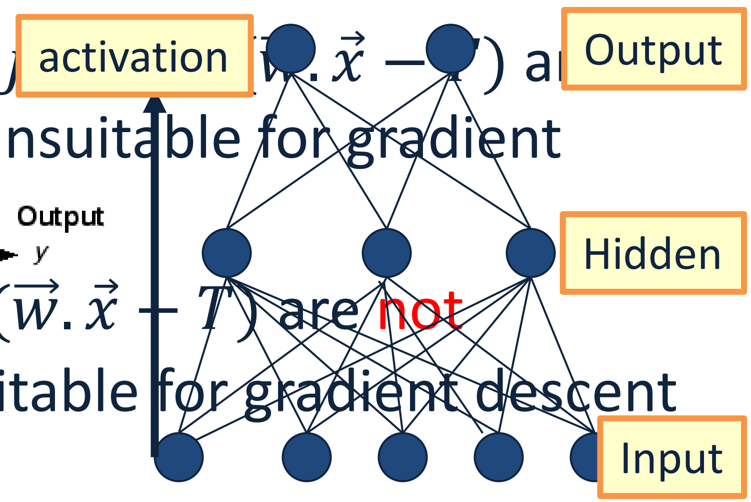
Neural Speed Constraints

- Neuron "switching time" is **O(milliseconds)**, compared to nanosecond for transistors.
  - However, biological systems can perform significant cognitive tasks (vision, language understanding) in fractions of a second.
- Even for limited abilities, current AI systems require orders of magnitude more steps.
- Human brain has approximately 10^10 neurons, each connected to 10^4; must explore massive parallelism (but there's more…)

Basic Unit in Multi-Layer Neural Network

- ***ssggnn*(*w ww w* . *x xx x* –*TT*) are not differentiable, hence unsuitable for gradient descent**
- ***w ww w* . *x xx x*** multiple layers of linear functions produce linear functions. We want to represent nonlinear functions.
- **Threshold units:** $o\ j = \ j\ j\ j$ ... $\vec{w}.\vec{x} - T$) a... not differentiable, hence unsuitable for gradient descent

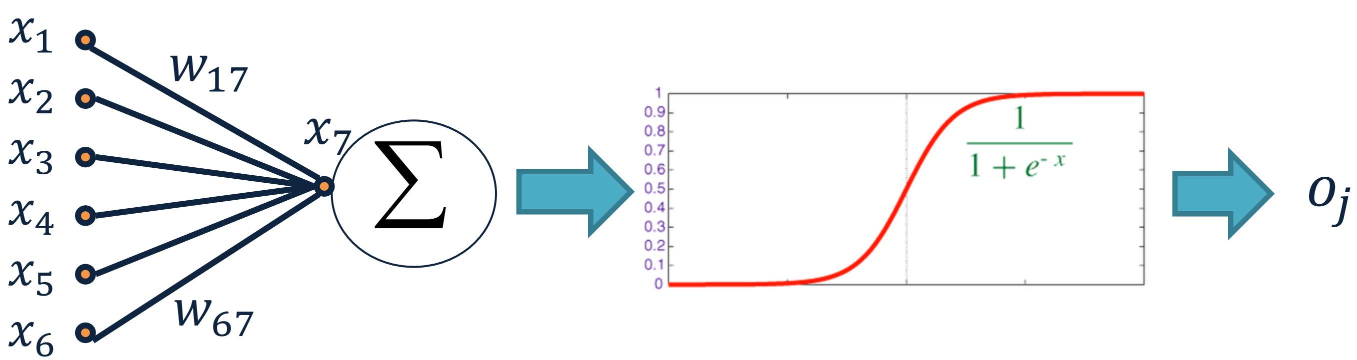




- **Threshold units:** $o_j = sgn(\vec{w}.\vec{x} - T)$ are not differentiable, hence unsuitable for gradient descent

Model Neuron (Logistic)

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.

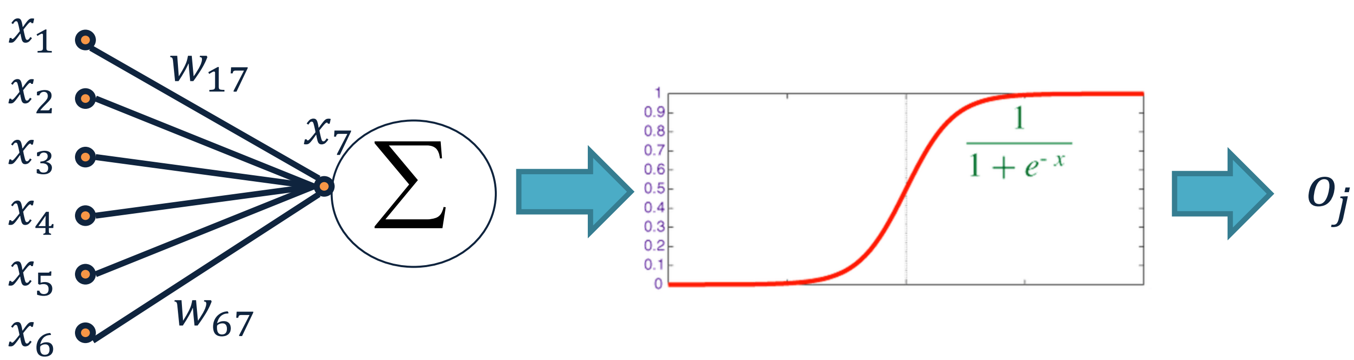


  - Use a non-linear, differentiable output function such as the sigmoid or logistic function

Model Neuron (Logistic)

Neuron Definition

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.

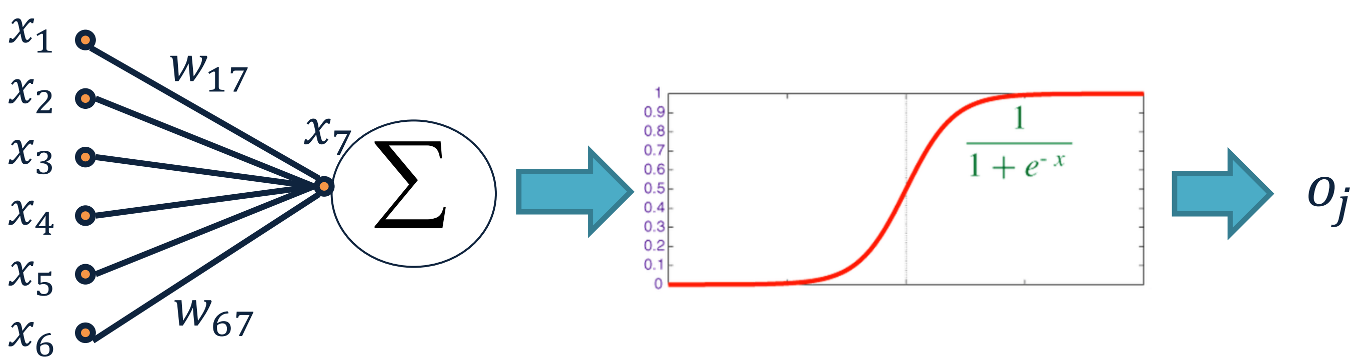


  - Use a non-linear, differentiable output function such as the sigmoid or logistic function
  - Net input to a unit is defined as: $\text{net}_j = \sum w_{ij} . x_i$

Model Neuron (Logistic)

Neuron Definition

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.



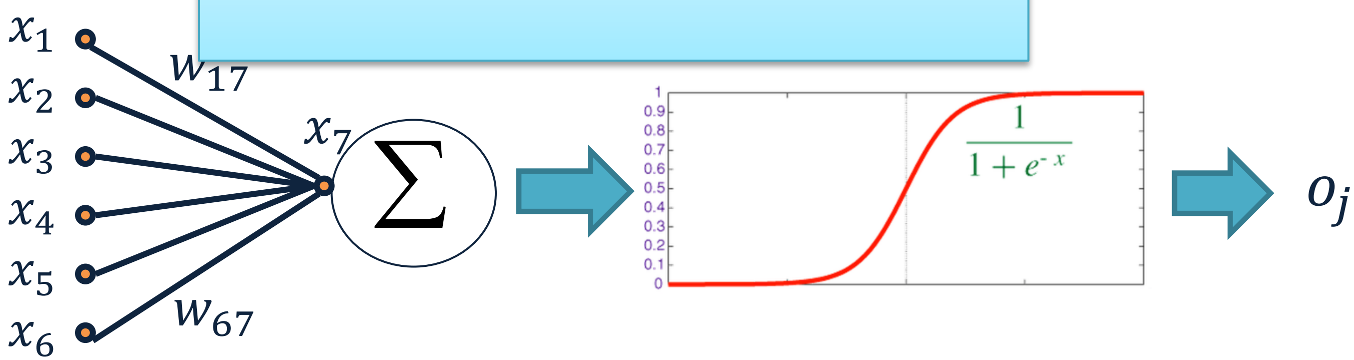
- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as: $\text{net}_j = \sum w_{ij} . x_i$
- Output of a unit is defined as: $o_j = \dfrac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$

Model Neuron (Logistic)

Neuron Definition

- Neuron is modeled by a unit j connected by weighted

The parameters so far?



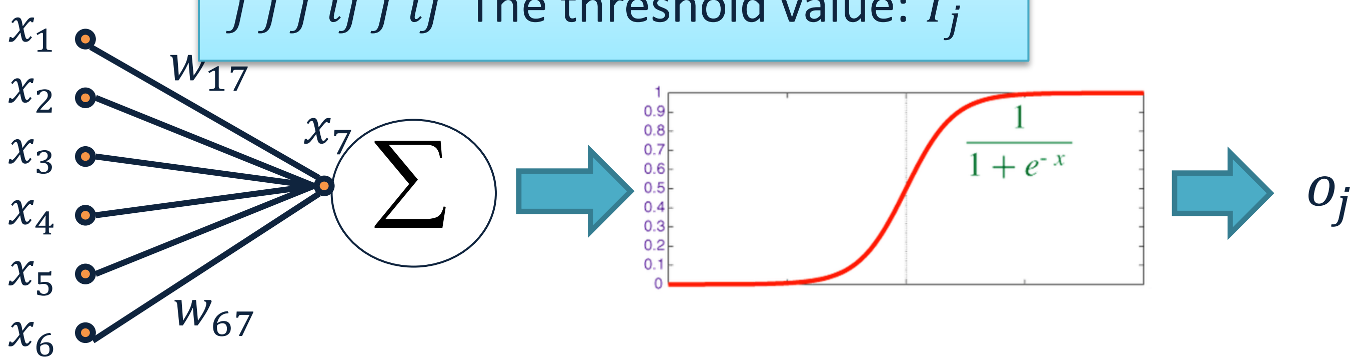
- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as: $\text{net}_j = \sum w_{ij} . x_i$
- Output of a unit is defined as: $o_j = \dfrac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$

Model Neuron (Logistic)

- Neuron i The parameters so far? j connected by weighted

The set of connectThe threshold value: $T\ j$

$j\ j\ j\ ij\ j\ ij$ The threshold value: T_j



- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as: $\text{net}_j = \sum w_{ij}.x_i$
- Output of a unit is defined as: $o_j = \dfrac{1}{1+\exp\big(-(\text{net}_j - T_j)\big)}$

Neuron Definition

Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:**
  - **OR:**
  - **NOT:** One input =1, the input to be inverted have negative weight

Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:**
  - **OR:**
  - **NOT:** One input =1, the input to be inverted have negative weight

Remember:

$$\text{net}_j = \sum w_{ij}.x_i$$

$$o_j = \frac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$$

Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:** $w_{ij} = T_j/n$
  - **OR:**
  - **NOT:** One input =1, the input to be inverted have negative weight

Remember:

$$\text{net}_j = \sum w_{ij}.x_i$$

$$o_j = \frac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$$

Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:** $w_{ij} = T_j/n$
  - **OR:** $w_{ij} = T_j$
  - **NOT:** One input =1, the input to be inverted have negative weight

Remember:

$$\text{net}_j = \sum w_{ij}.x_i$$

$$o_j = \frac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$$

Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:** $w_{ij} = T_j/n$
  - **OR:** $w_{ij} = T_j$
  - **NOT:** One input =1, the input to be inverted have negative weight
- Can build arbitrary logic circuits, finite-state machines and computers given these basis gates.
- Can specify any Boolean function using two layer network (w/ negation)
  - DNF and CNF are universal representations

Remember:

$$\mathrm{net}_j = \sum w_{ij}.x_i$$

$$o_j = \frac{1}{1 + \exp\left(-(\mathrm{net}_j - T_j)\right)}$$

Learning Rules

Examples of update rules

- Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase: $w_{ij} = w_{ij} + R o_i o_j$
  - R and is a constant called **the learning rate**
  - Supported by physiological evidence

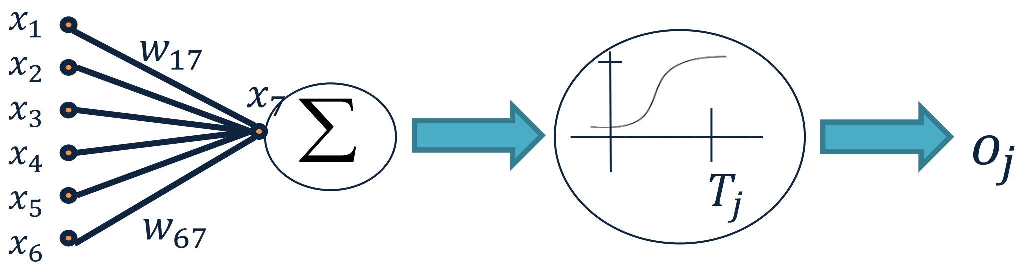
Learning Rules

Examples of update rules

- Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase: $w_{ij} = w_{ij} + R o_i o_j$
  - R and is a constant called **the learning rate**
  - Supported by physiological evidence
- Rosenblatt (1959) suggested that when a target output value is provided for a single neuron **with fixed input**, it can **incrementally change weights** and learn to produce the output using the **Perceptron learning rule**.
  - assumes **binary output** units; single linear threshold unit

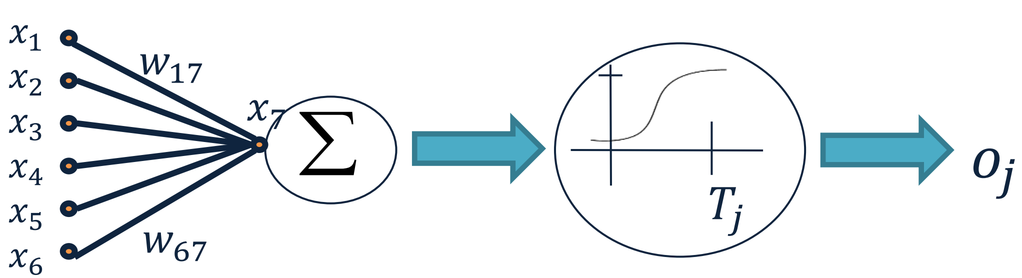
Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$



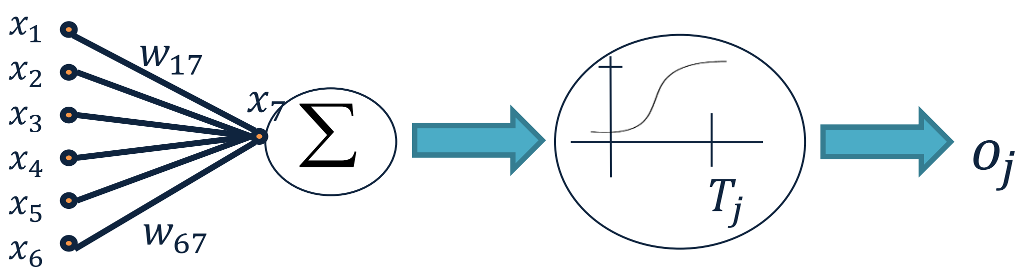
Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R\left(t_j - o_j\right)x_i$
  - If output is **correct**, don't change the weights



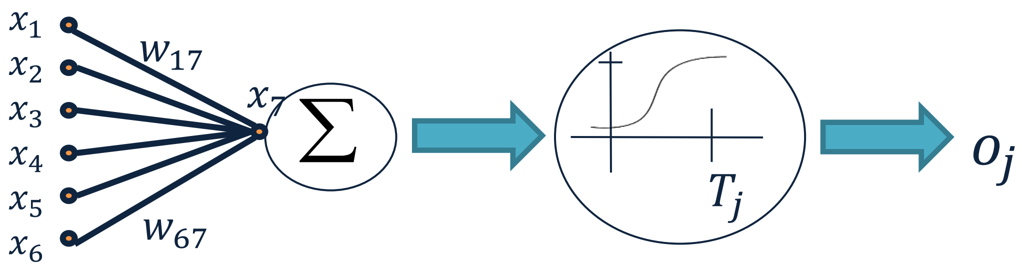
Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$
  - If output is **correct**, don't change the weights



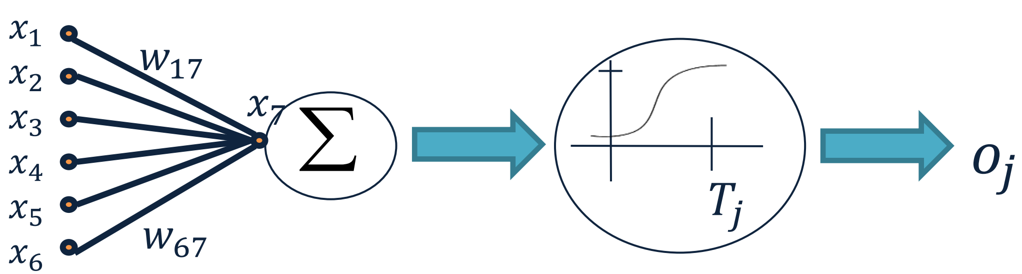
Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$
  - If output is **correct**, don't change the weights
  - If output is **wrong**, change weights for all inputs which are 1



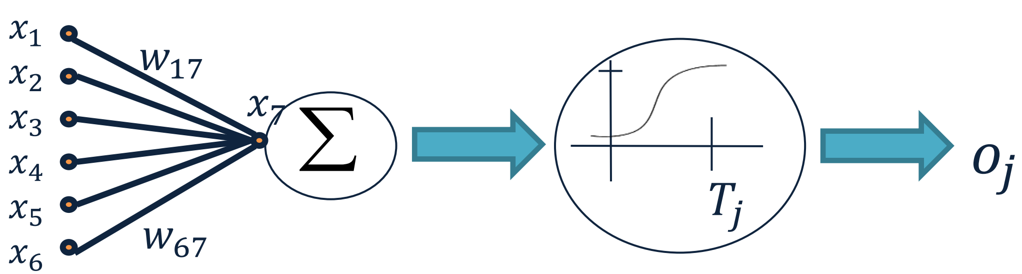
Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$
  - If output is **correct**, don't change the weights
  - If output is **wrong**, change weights for all inputs which are 1
    - If output is low (0, needs to be 1) increment weights



Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$
  - If output is **correct**, don't change the weights
  - If output is **wrong**, change weights for all inputs which are 1
    - If output is low (0, needs to be 1) increment weights
    - If output is high (1, needs to be 0) decrement weights



Perceptron Learning Algorithm

- Repeatedly iterate through examples adjusting weights according to the perceptron learning rule until all outputs are corrected

$$w_{ij} = w_{ij} + R(t_j - o_j)x_i$$

1. Initialize all weights to zero (or randomly)
2. Until outputs for all training examples are correct
   1. For each training example j, do
      1. compute the current output o
      2. compare it to the target t and update the weights according to the perceptron learning rule

Perceptron Learning Algorithm

- Repeatedly iterate through examples adjusting weights according to the perceptron learning rule until all outputs are corrected

$$w_{ij} = w_{ij} + R(t_j - o_j)x_i$$

1. Initialize all weights to zero (or randomly)
2. Until outputs for all training examples are correct
    1. For each training example j, do
        1. compute the current output o
        2. compare it to the target t and update the weights according to the perceptron learning rule

When will this algorithm terminate (converge) ?

Perceptron Convergence

Perceptron Guarantees

- **Perceptron Convergence Theorem:**

Perceptron Convergence

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are **consistent** with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge

Perceptron Guarantees

Perceptron Convergence

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are **consistent** with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - How long would it take to converge ?

Perceptron Guarantees

Perceptron Convergence

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are **consistent** with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - How long would it take to converge ?
- **Perceptron Cycling Theorem:**

Perceptron Guarantees

Perceptron Convergence

Perceptron Guarantees

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are **consistent** with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - How long would it take to converge ?
- **Perceptron Cycling Theorem:**
- If the training data is not linearly separable the perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.

Perceptron Convergence

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are **consistent** with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - How long would it take to converge ?
- **Perceptron Cycling Theorem:**
- If the training data is not linearly separable the perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.
  - How to provide robustness, more expressivity ?

Perceptron Learnability

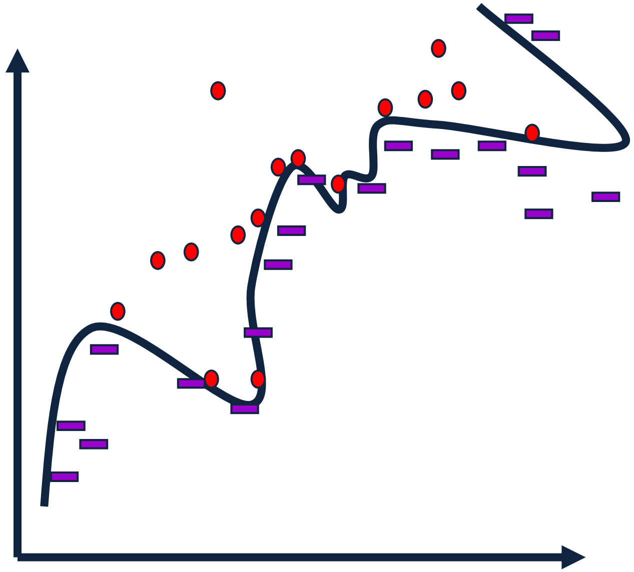
- Obviously **cannot learn what it cannot represent**
  - Only linearly separable functions

Perceptron Learnability

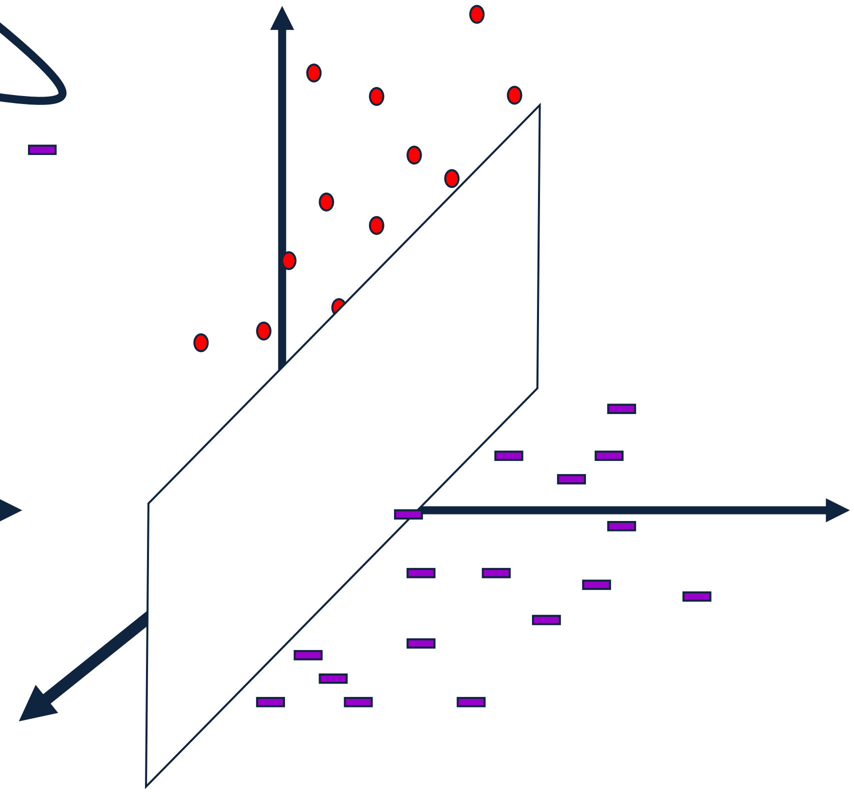
- Obviously **cannot learn what it cannot represent**
  - Only linearly separable functions
- Minsky and Papert (1969) wrote an influential book in which they demonstrate the representational limitations of Perceptron
  - Parity functions cannot be learned (generalization of XOR)
  - In visual pattern recognition, if patterns are represented using local features, perceptron cannot represent properties like Symmetry, Connectivity

Perceptron Learnability

- Obviously **cannot learn what it cannot represent**
  - Only linearly separable functions
- Minsky and Papert (1969) wrote an influential book in which they demonstrate the representational limitations of Perceptron
  - Parity functions cannot be learned (generalization of XOR)
  - In visual pattern recognition, if patterns are represented using local features, perceptron cannot represent properties like Symmetry, Connectivity
- These observations discouraged research on neural network for years.
- But, Rosenblatt (1959) asked: "What pattern recognition problems can be transformed so as to become linearly separable"



$(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$



$y_1 \wedge y_2$

Widrow-Hoff Rule

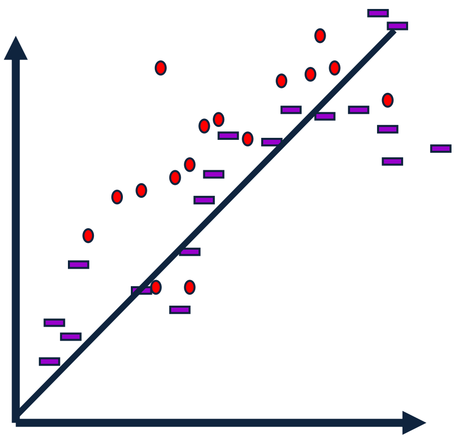
- This incremental update rule provides an approximation to the goal:
  - Find the best linear approximation of the data

$$Err\left(\vec{w}^{(j)}\right) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

  - where:

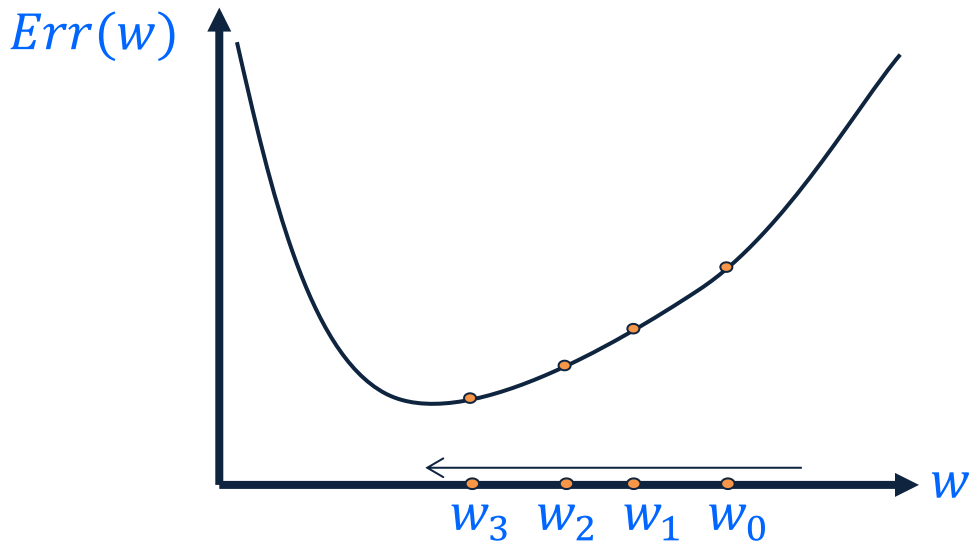
$$o_d = \sum_i w_{ij} . x_i = \vec{w}^{(j)} . \vec{x}$$

    output of linear unit on example *d*
  - t_d = Target output for example d



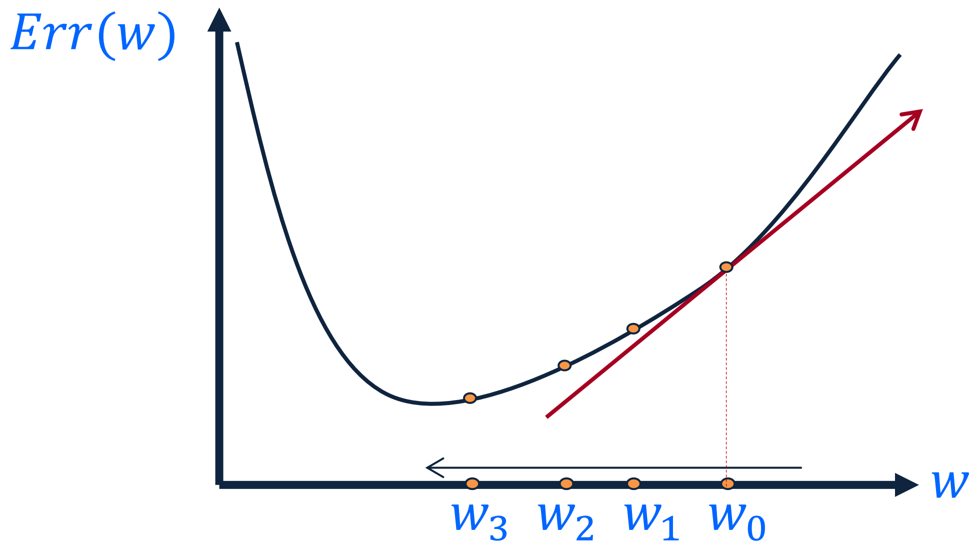
Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



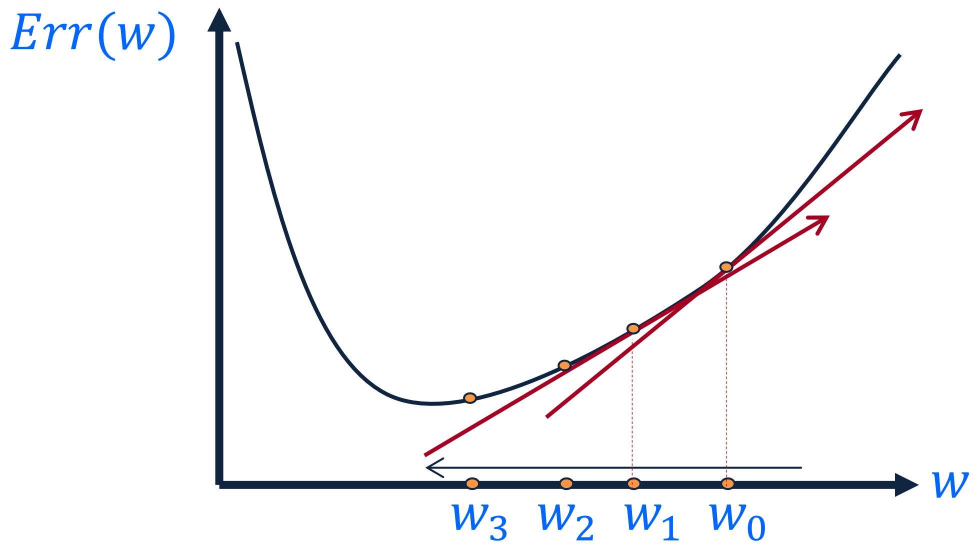
Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



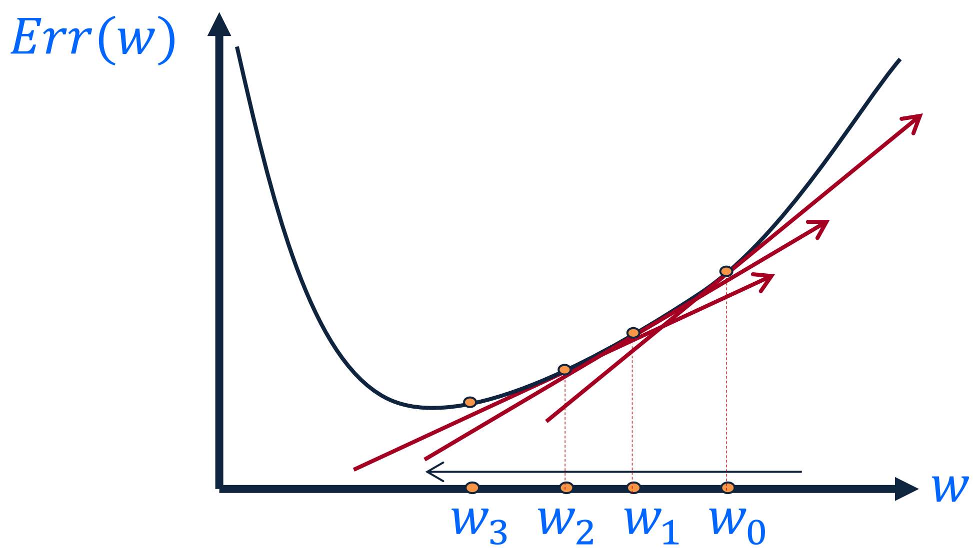
Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



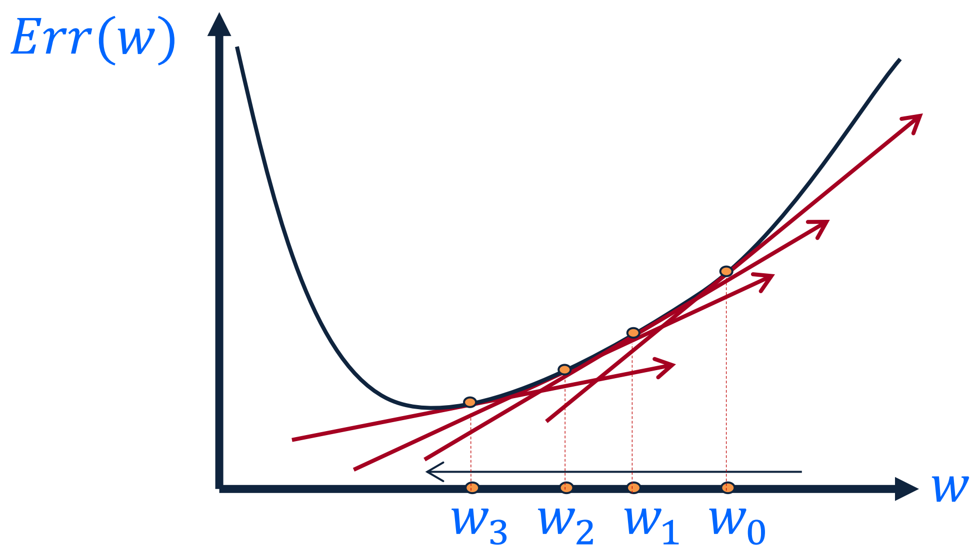
Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



Gradient Descent

To find the best direction in the **feature space** we compute the gradient of E with respect to each of the components of $\vec{w}$

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

- This vector specifies the direction the produces the steepest increase in E;
- in the direction of $-\nabla E(\vec{w})$

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

- Where:

$$\Delta\vec{w} = -\mathrm{R}\,\nabla E(\vec{w})$$

Gradient Descent

$w\,ww\,w = -R\,\nabla\nabla EE\;\; w\;\;\; w\,ww\,w\;\;\; w$

$w\,ww\,w + \Delta\, w\,ww\,w$

in the direction of $-\nabla\nabla EE\;\; w\;\;\; w\,ww\,w\;\;\; w$

To find the best direction in the **feature space** we compute the gradient of E with respect to each of the components of $\vec{w}$

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_n}\right]$$

- This vector specifies the direction the produces the steepest increase in E;

$$\Delta\vec{w} = -R\,\nabla E(\vec{w})$$

- Where:

$$\Delta\vec{w} = -R\,\nabla E(\vec{w})$$

Gradient Descent – LMS

- We have: $Err\left(\vec{w}^{(j)}\right) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$
- Therefore:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

Gradient Descent – LMS

- We have: $Err\left(\vec{w}^{(j)}\right) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$
- Therefore:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 =$$

Gradient Descent – LMS

- We have: $Err\left(\vec{w}^{(j)}\right) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$
- Therefore:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d \in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 =$$

$$= \frac{1}{2}\sum_{d \in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}.\vec{x}_d) \quad =$$

Gradient Descent – LMS

- We have: $Err\left(\vec{w}^{(j)}\right) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$
- Therefore:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 =$$

$$= \frac{1}{2}\sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}.\vec{x}_d) \quad =$$

$$= \sum_{d\in D}(t_d - o_d)(-x_{id})$$

$$\vec{x}_d = [x_{1d}, \ldots, x_{nd}]$$
$$\vec{w} = [w_1, \ldots, w_n]$$

Gradient Descent – LMS

- Weight update rule: $\Delta w_i = R \sum_{d \in D} (t_d - o_d) \vec{x}_{id}$
- Gradient descent algorithm for training linear units:

1. Start with an initial random weight vector
2. For every example d with target value t_d:
   1. Evaluate the linear unit $o_d = \sum_i w_i . x_d = \vec{w}^{(j)} . \vec{x}_d$
3. update $\vec{w}$ by adding Δw_i to each component
4. Continue until E below some threshold

Convergence?

Gradient Descent – LMS

- Weight update rule: $\Delta w_i = R\sum_{d \in D}(t_d - o_d)\vec{x}_{id}$
- Gradient descent algorithm for training linear units:
  1. Start with an initial random weight vector
  2. For every example d with target value t_d:
     1. Evaluate the linear unit $o_d = \sum_i w_i . x_d = \vec{w}^{(j)} . \vec{x}_d$
  3. update $\vec{w}$ by adding Δw_i to each component
  4. Continue until E below some threshold
- Because the surface contains only a single global minimum the algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable

Convergence?

Gradient Descent – LMS

Weight update rule: $\Delta \;= R$

Convergence

Gradient Descent – LMS

Weight update rule: $\Delta = R$

Convergence

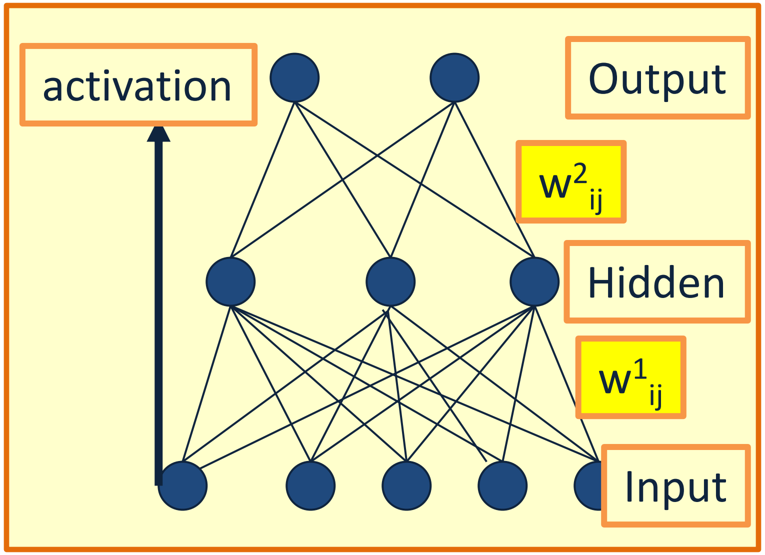
- In general does not converge to global minimum
- Robbins-Monro: Decreasing R with time, guarantees convergence.

Summary: Single Layer Network

- Variety of update rules
  - Multiplicative
  - Additive
- **Batch** and **incremental** algorithms
- Various convergence and efficiency conditions
- There are other ways to learn linear functions
  - Linear Programming (general purpose)
  - Probabilistic Classifiers (some assumption)
- Although simple and restrictive -- linear predictors perform very well on many realistic problems
- However, the representational restriction is limiting in many applications

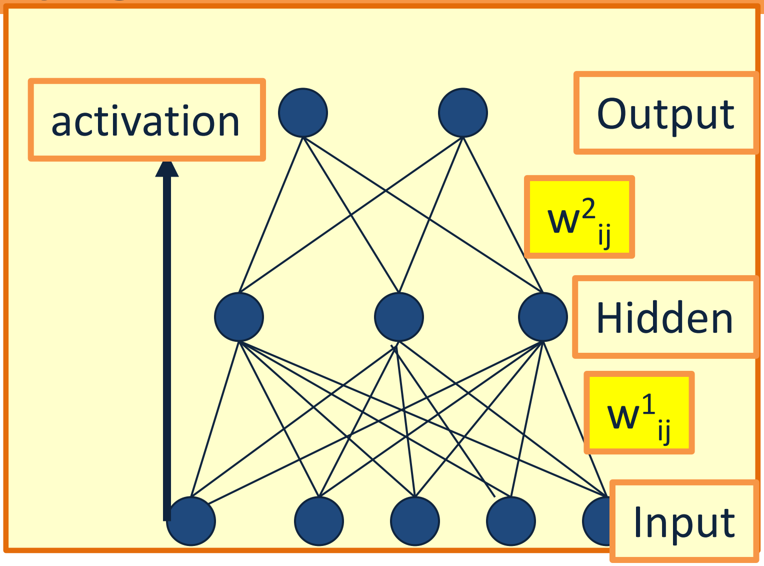
Learning with a Multi-Layer Perceptron

- It's easy to learn the top layer – it's just a linear unit.
- Given feedback (truth) at the top layer, and the activation at the layer below it, you can use the Perceptron update rule (more generally, gradient descent) to updated these weights.
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).



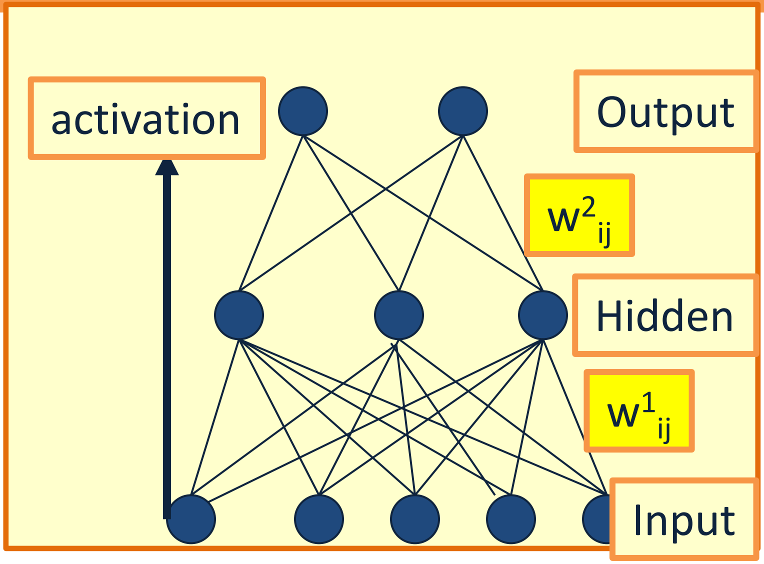
Learning with a Multi-Layer Perceptron

- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).



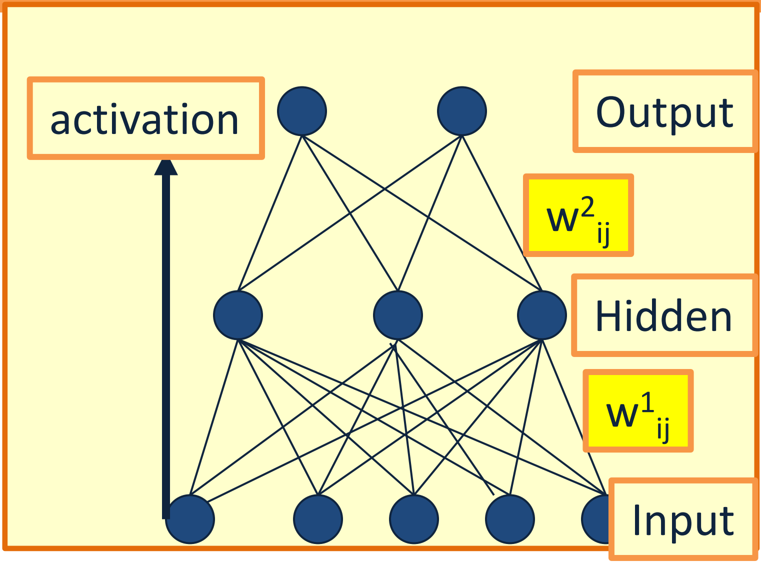
Learning with a Multi-Layer Perceptron

- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation



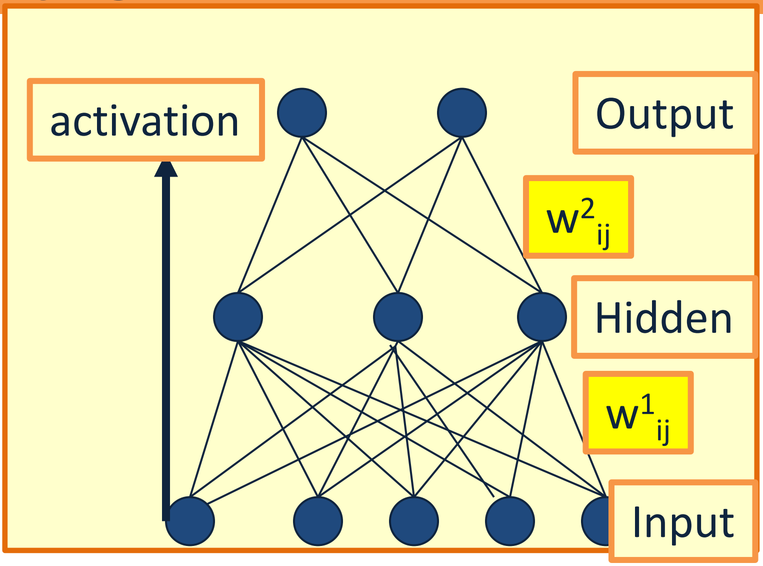
Learning with a Multi-Layer Perceptron

- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then

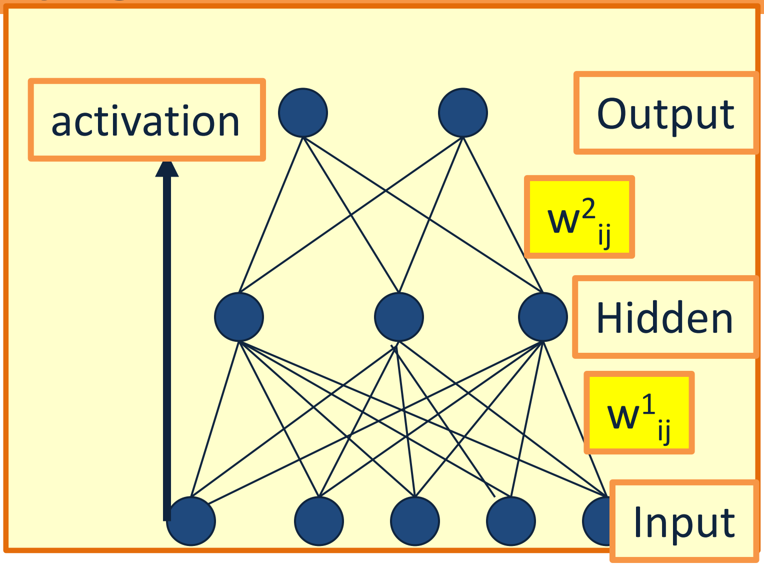


Learning with a Multi-Layer Perceptron

- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also

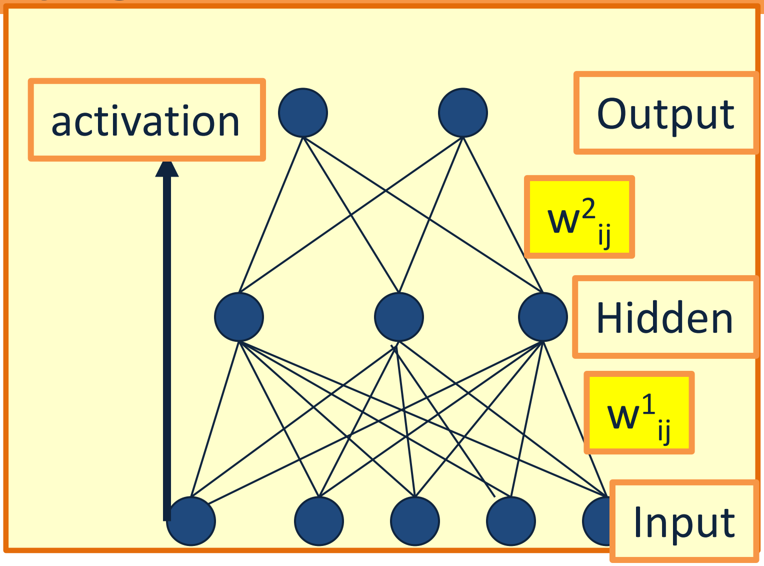


Learning with a Multi-Layer Perceptron



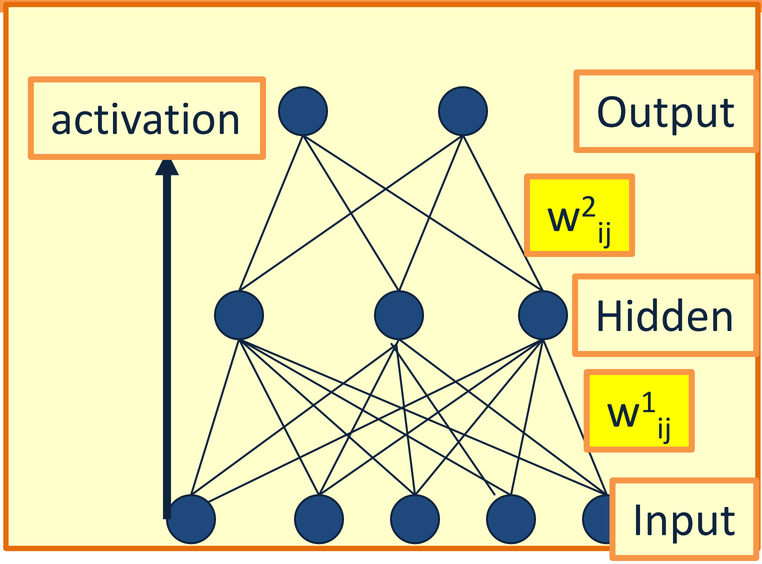
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also a differentiable function of the input and weights in the network.

Learning with a Multi-Layer Perceptron



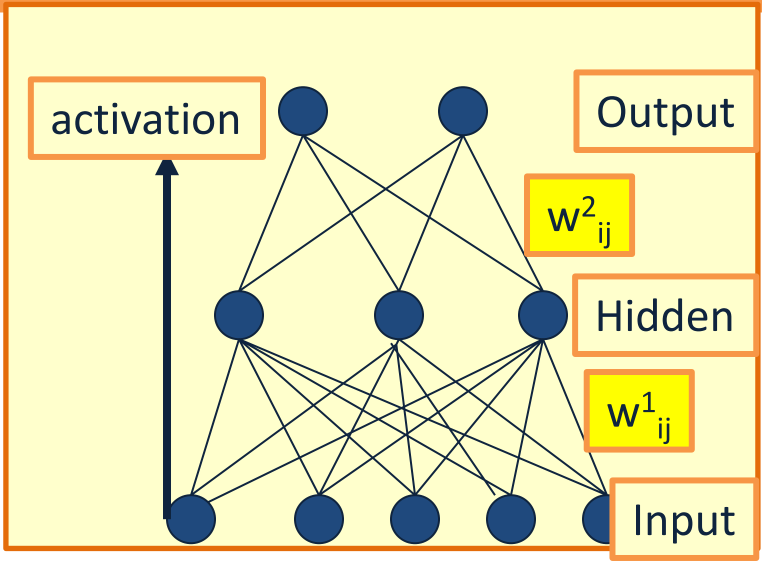
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also a differentiable function of the input and weights in the network.
- Define an error function (e.g., sum of squares) that is a differentiable function of the output, i.e. this error function is also a differentiable function of the weights.

Learning with a Multi-Layer Perceptron



- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also a differentiable function of the input and weights in the network.
- Define an error function (e.g., sum of squares) that is a differentiable function of the output, i.e. this error function is also a differentiable function of the weights.
- We can then evaluate the derivatives of the error with respect to the weights, and use these derivatives to find weight values that minimize this error function. This can be done, for example, using gradient descent (or other optimization methods).

Learning with a Multi-Layer Perceptron



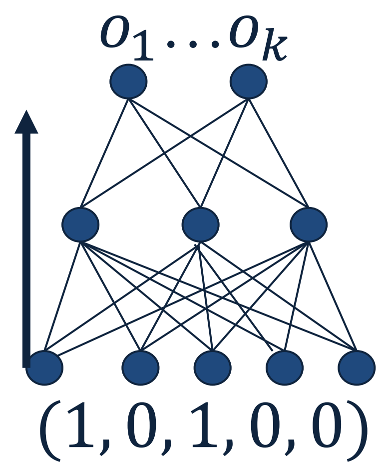
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also a differentiable function of the input and weights in the network.
- Define an error function (e.g., sum of squares) that is a differentiable function of the output, i.e. this error function is also a differentiable function of the weights.
- We can then evaluate the derivatives of the error with respect to the weights, and use these derivatives to find weight values that minimize this error function. This can be done, for example, using gradient descent (or other optimization methods).
- This results in an algorithm called back-propagation.

Backpropagation Learning Rule

- Since there could be multiple output units, we define the **error** as the sum over all the network output units.

$$Err(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in K} \quad (t_{kd} - o_{kd})^2$$

  - where D is the set of training examples,
  - K is the set of output units



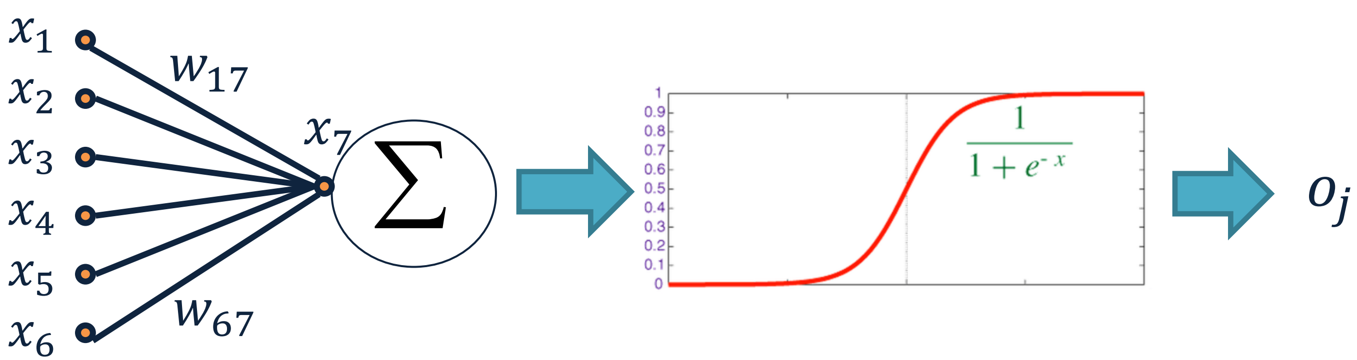
- This can be used to derive the (global)
- learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

$$\Delta w_{ij} = -R\,\frac{\partial E}{\partial w_{ij}}$$

Reminder: Model Neuron (Logistic)

Neuron Definition

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.



  - Use a non-linear, differentiable output function such as the sigmoid or logistic function
  - Net input to a unit is defined as: $\text{net}_j = \sum w_{ij} . x_i$
  - Output of a unit is defined as: $o_j = \dfrac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$

Derivation of Learning Rule

Propagation error to earlier layer

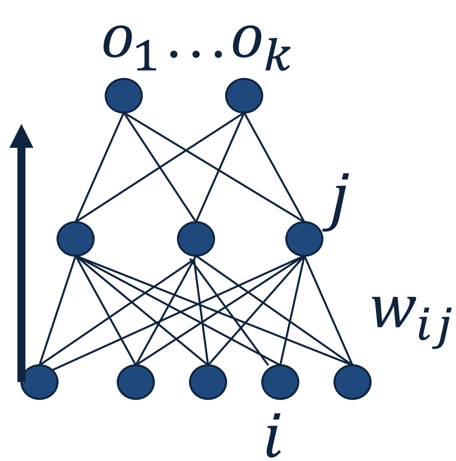
The weights are updated incrementally; the error is computed **for each example** and the weight update is then derived.

- $Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$
- influences the output only through net_j

$$\mathrm{net}_j = \sum w_{ij}.x_{ij}$$

- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$



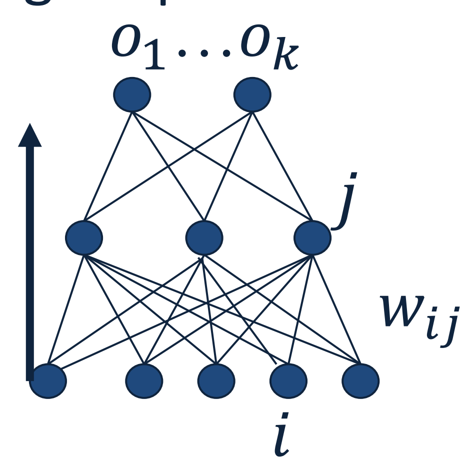
Derivation of Learning Rule

Propagation error to earlier layer

et net j jj net j $=\sum w$ ij ww w ij $iijj$ w ij . x ij xx x ij $iijj$ x ij

influences the output only through net j net net j jj net j

The weights are updated incrementally; the error is computed **for each example** and the weight update is then derived.



- $Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$

$$\text{net}\, j \; = \sum w_{ij}.x_{ij}$$

$$\text{net}_j = \sum w_{ij}.x_{ij}$$

- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

Derivation of Learning Rule

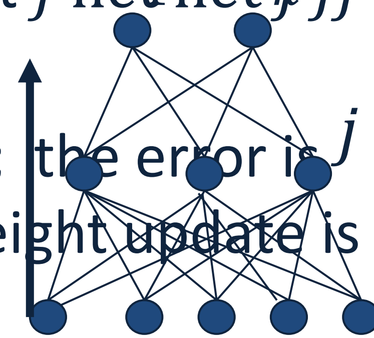
Propagation error to earlier layer

∂ E d ∂ net j ∂∂ E d EE E d dd E d ∂ E d ∂ net j ∂ ∂ net j net net j jj net j ∂ E d ∂ net j ∂ net j ∂ w ij ∂∂ net j net net j jj net j ∂ net j ∂ w ij ∂∂ w ij ww w ij iijj w ij ∂ net j ∂ w ij

et net j jj net j =∑ w ij ww w ij iijj w ij . x ij xx x ij iijj x ij

influences the output only through net j net net j jj net j

The weights are updated incrementally; the error is computed **for each example** and the weight update is then derived.



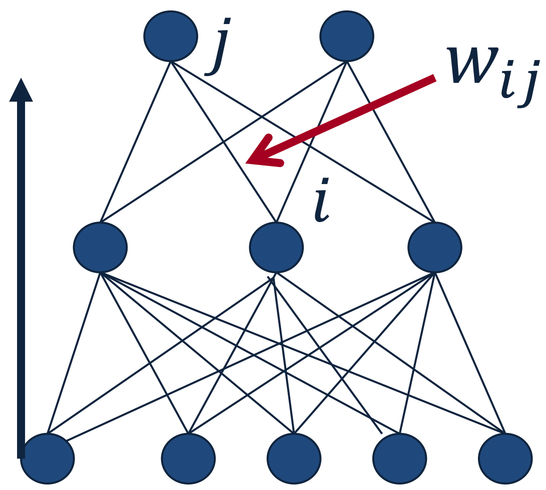
- $Err_d(\vec{w}) = \frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$
- Therefore:

∂ E d ∂ w ij

Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

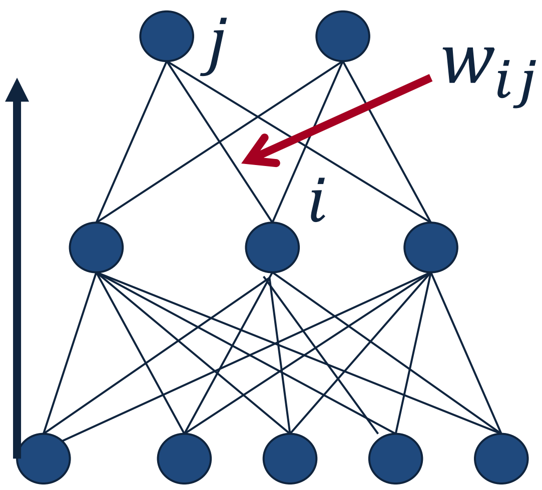


Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial o_j} \quad \frac{\partial o_j}{\partial \mathrm{net}_j} \quad \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$



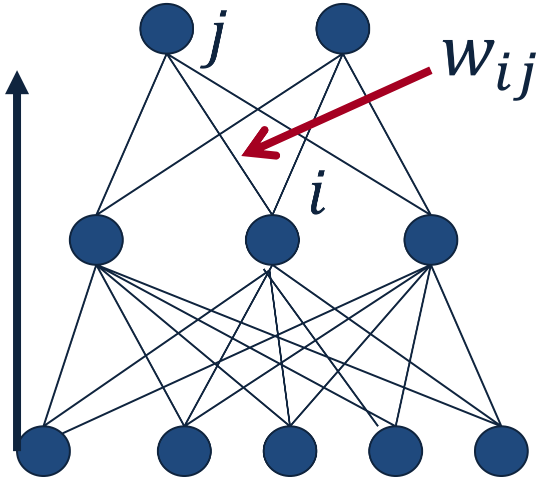
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial \mathrm{o}_j}\frac{\partial o_j}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= -\left(t_j - o_j\right)\ o_j\left(1 - o_j\right)\ \ x_{ij}$$



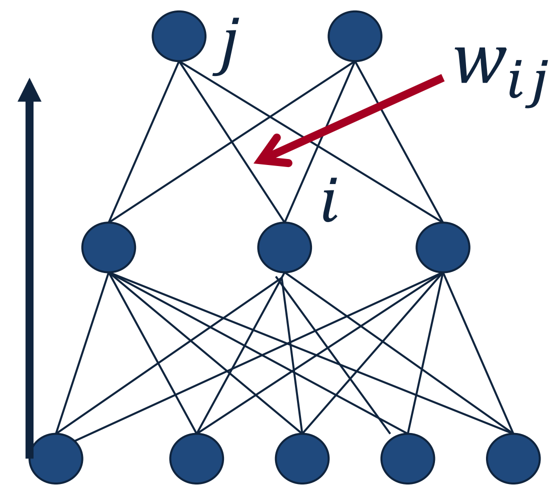
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \boxed{\frac{\partial E_d}{\partial \text{o}_j}} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \boxed{-(t_j - o_j)} \; o_j(1 - o_j) \quad x_{ij}$$



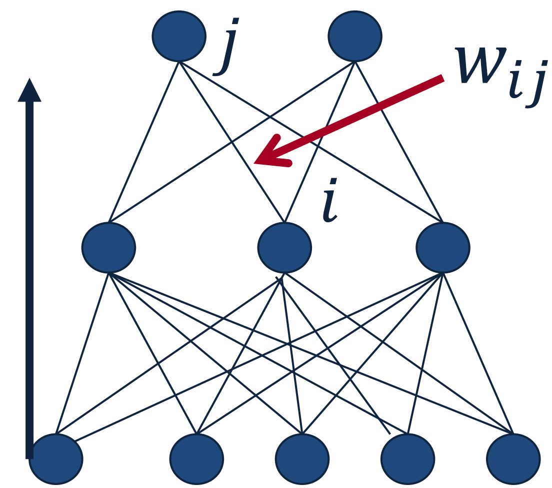
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\ \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \boxed{\frac{\partial E_d}{\partial o_j}}\quad \frac{\partial o_j}{\partial \mathrm{net}_j}\quad \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \boxed{-(t_j - o_j)}\quad o_j(1 - o_j)\quad x_{ij}$$



$$Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$$

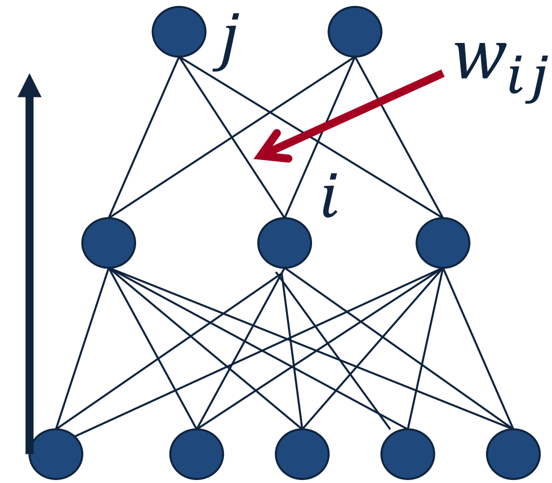
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial \mathrm{o}_j} \quad \frac{\partial o_j}{\partial \mathrm{net}_j} \quad \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= -\left(t_j - o_j\right) \quad o_j\left(1 - o_j\right) \quad x_{ij}$$



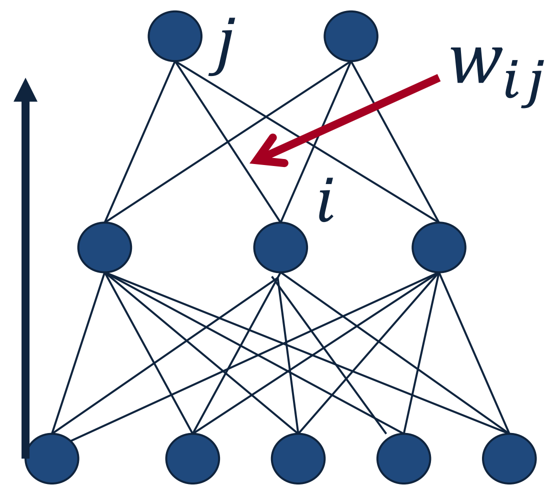
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial o_j} \boxed{\frac{\partial o_j}{\partial \text{net}_j}} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= -\left(t_j - o_j\right) \boxed{o_j\left(1 - o_j\right)} \; x_{ij}$$



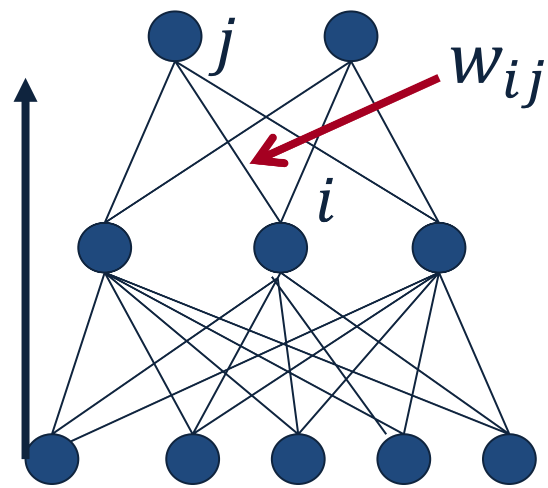
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial o_j}\boxed{\frac{\partial o_j}{\partial \mathrm{net}_j}}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= -(t_j - o_j)\,\boxed{o_j(1 - o_j)}\,x_{ij}$$



$$\frac{\partial o_j}{\partial \mathrm{net}_j} = o_j(1 - o_j)$$

$$o_j = \frac{1}{1 + \exp\{-(\mathrm{net}_j - T_j)\}}$$

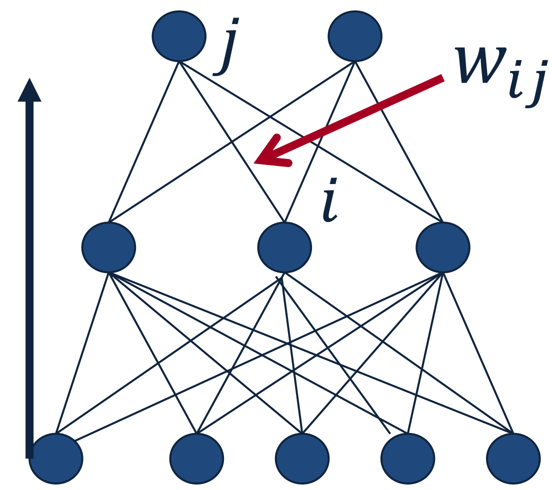
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial \mathrm{o}_j} \quad \frac{\partial o_j}{\partial \mathrm{net}_j} \quad \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= -\left(t_j - o_j\right) \quad o_j\left(1 - o_j\right) \quad x_{ij}$$



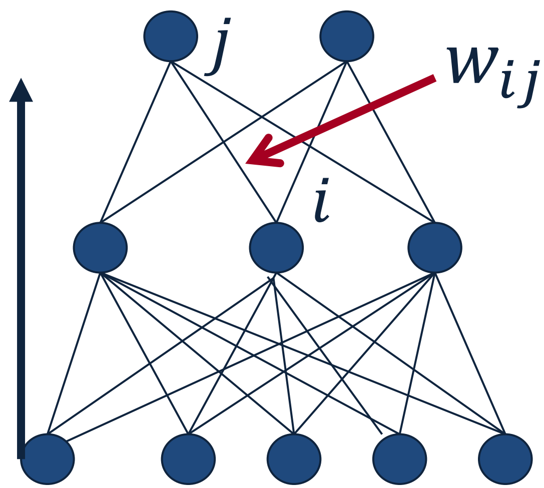
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\ \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial \mathrm{o}_j}\ \frac{\partial o_j}{\partial \mathrm{net}_j}\ \boxed{\frac{\partial \mathrm{net}_j}{\partial w_{ij}}}$$

$$= -\left(t_j - o_j\right)\ o_j\left(1 - o_j\right)\ \boxed{x_{ij}}$$



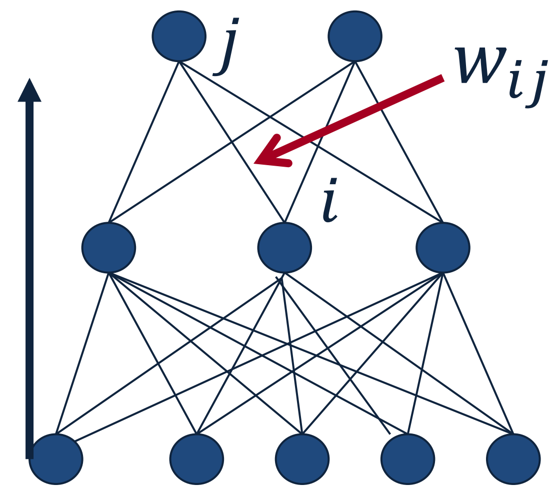
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\ \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial \text{o}_j}\quad \frac{\partial o_j}{\partial \text{net}_j}\quad \boxed{\frac{\partial \text{net}_j}{\partial w_{ij}}}$$

$$= -\left(t_j - o_j\right)\ \ o_j\left(1 - o_j\right)\ \ \boxed{x_{ij}}$$



$$\boxed{\text{net}_j = \sum w_{ij} . x_{ij}}$$

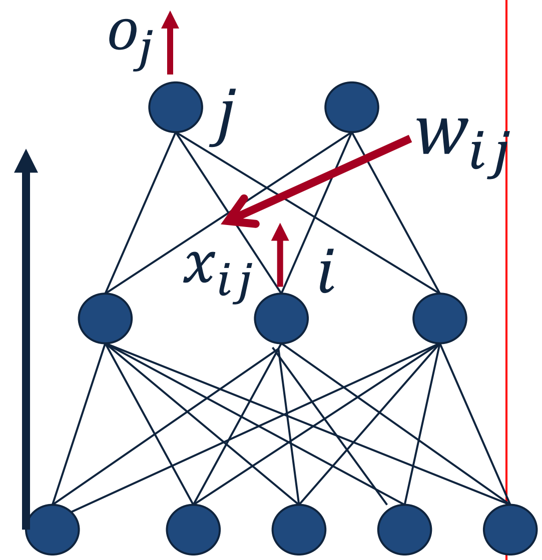
Derivation of Learning Rule (3)

- Weights of output units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = R(t_j - o_j)o_j(1 - o_j)x_{ij}$$
$$= R\delta_j x_{ij}$$

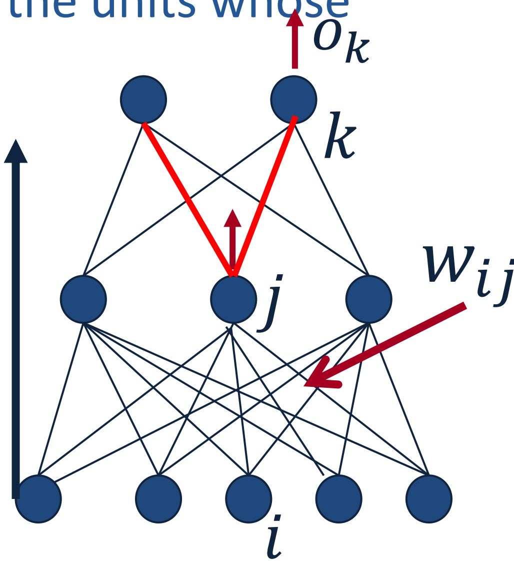
where

$$\delta_j = (t_j - o_j)o_j(1 - o_j)$$



Derivation of Learning Rule (4)

- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j



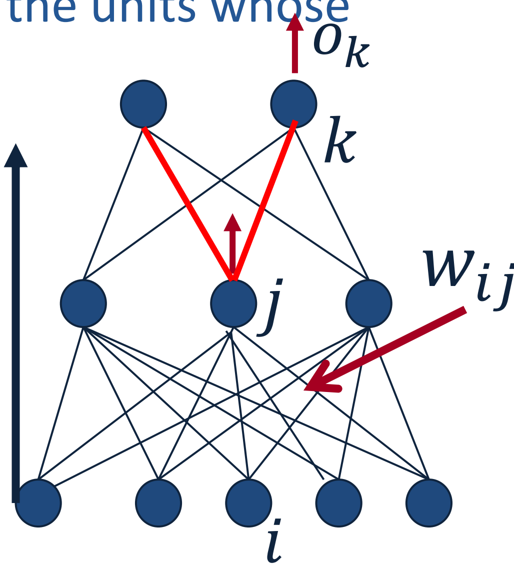
$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}} =$$

Derivation of Learning Rule (4)

- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\,\frac{\partial \text{net}_j}{\partial w_{ij}} =$$

$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k}\,\frac{\partial \text{net}_k}{\partial \text{net}_k}\, x_{ij}$$



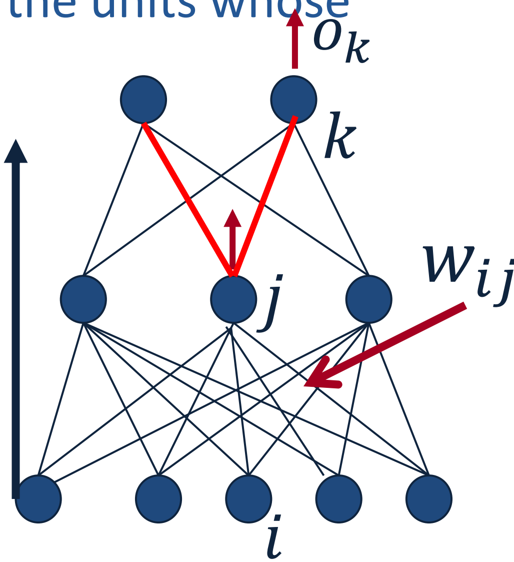
Derivation of Learning Rule (4)

- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\mathrm{net}_j = \sum w_{ij} . x_{ij}$$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j} \boxed{\frac{\partial \mathrm{net}_j}{\partial w_{ij}}} =$$

$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \mathrm{net}_k} \frac{\partial \mathrm{net}_k}{\partial \mathrm{net}_k} \boxed{x_{ij}}$$



Derivation of Learning Rule (4)

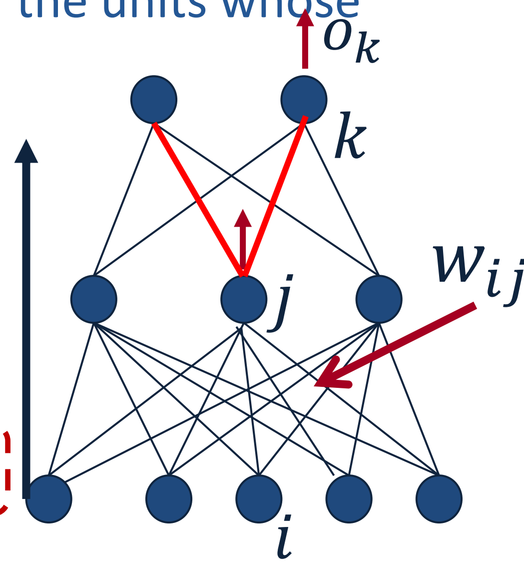
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\mathrm{net}_j = \sum w_{ij}.x_{ij}$$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}} =$$

$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \mathrm{net}_k}\frac{\partial \mathrm{net}_k}{\partial \mathrm{net}_k} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \mathrm{net}_k}{\partial \mathrm{net}_k} x_{ij}$$



Derivation of Learning Rule (4)

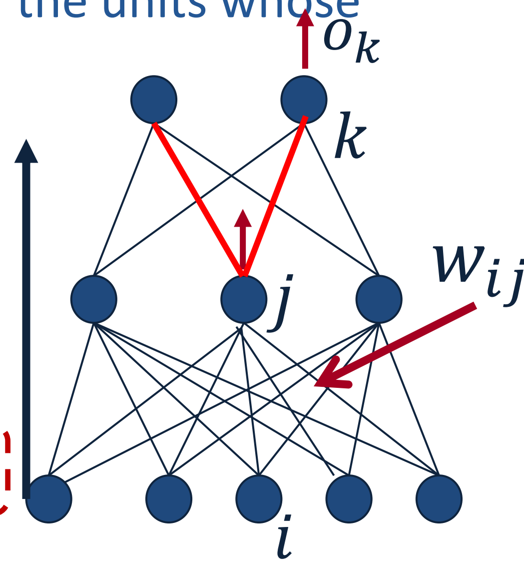
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\text{net}_j = \sum w_{ij} . x_{ij}$$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \boxed{\frac{\partial \text{net}_j}{\partial w_{ij}}} =$$

$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_k} \boxed{x_{ij}}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_k} x_{ij}$$



Derivation of Learning Rule (4)

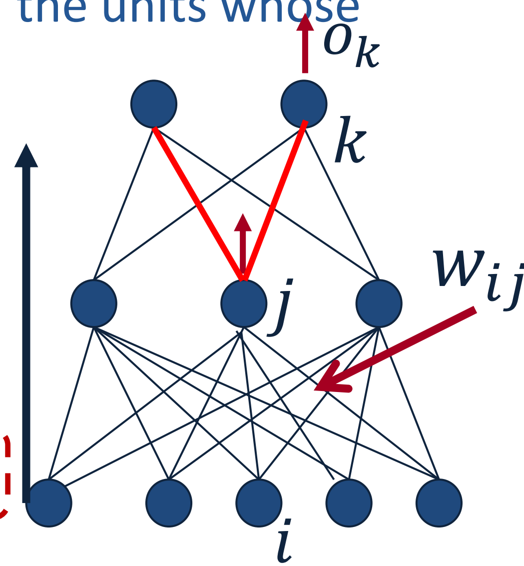
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$\text{net}_j = \sum w_{ij} . x_{ij}$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \boxed{\frac{\partial \text{net}_j}{\partial w_{ij}}} =$$

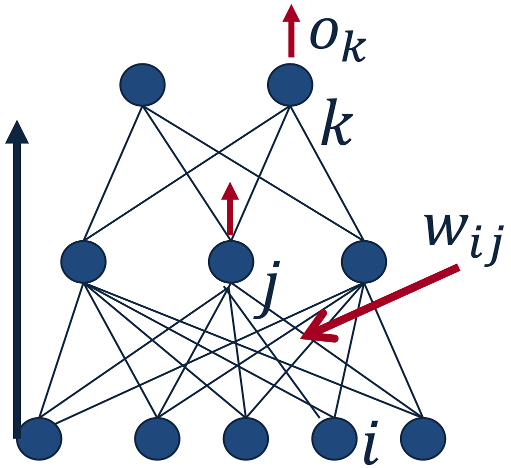
$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_k} \boxed{x_{ij}}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_k} x_{ij}$$



Derivation of Learning Rule (5)

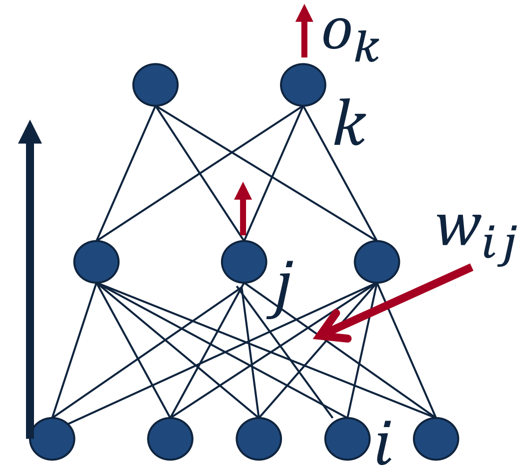
- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_k} x_{ij} =$$

Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j

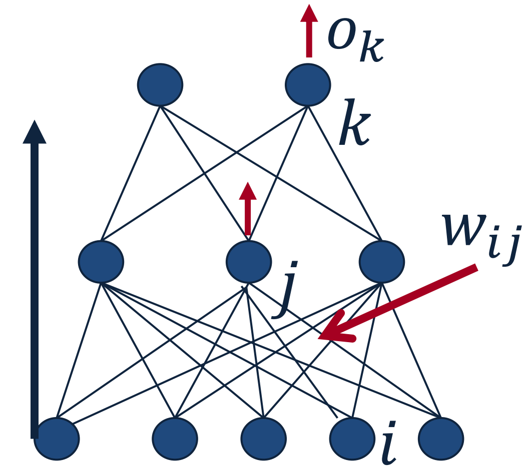


$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \mathrm{net}_k}{\partial \mathrm{net}_k} x_{ij} =$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \mathrm{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net}_j} x_{ij}$$

Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



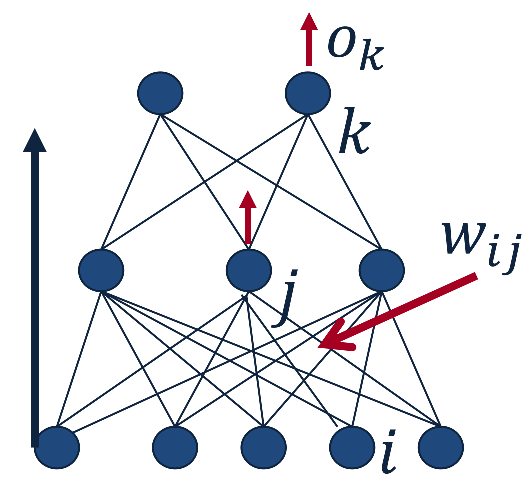
$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \ \frac{\partial \mathrm{net}_k}{\partial \mathrm{net}_k} x_{ij} =$$

$$= \sum_{k \in downstream(j)} -\delta_k \ \frac{\partial \mathrm{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net}_j} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \ w_{jk} \ o_j(1-o_j) \quad x_{ij}$$

Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



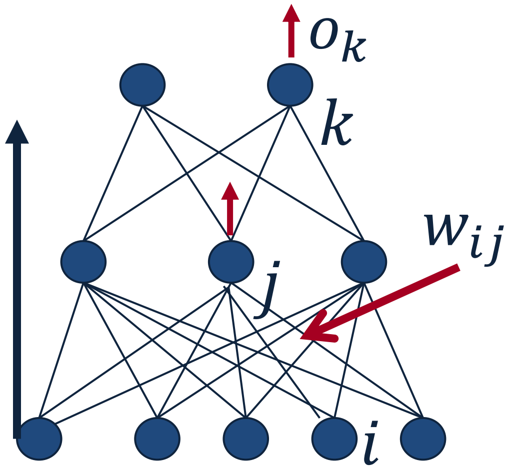
$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_k} x_{ij} =$$

$$= \sum_{k \in downstream(j)} -\delta_k \boxed{\frac{\partial \text{net}_k}{\partial o_j}} \frac{\partial o_j}{\partial \text{net}_j} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \boxed{w_{jk}}\, o_j(1 - o_j) \quad x_{ij}$$

Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_k} x_{ij} =$$

$$= \sum_{k \in downstream(j)} -\delta_k \boxed{\frac{\partial \text{net}_k}{\partial o_j}} \boxed{\frac{\partial o_j}{\partial \text{net}_j}} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \boxed{w_{jk}} \boxed{o_j(1-o_j)} \; x_{ij}$$

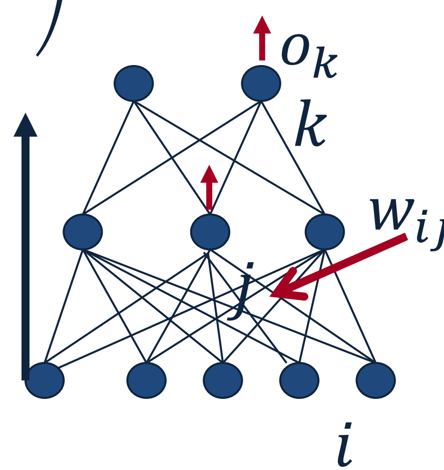
Derivation of Learning Rule (6)

Delta Rule

- Weights of hidden units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = R\, o_j(1 - o_j).\left(\sum_{k \in downstream(j)} -\delta_k \; w_{jk}\right) x_{ij}$$

$$= R\delta_j x_{ij}$$



- Where

$$\delta_j = o_j(1 - o_j).\left(\sum_{k \in downstream(j)} -\delta_k \; w_{jk}\right)$$

- First determine the error for the output units.
- Then, backpropagate this error layer by layer through the network, changing weights appropriately in each layer.

The Backpropagation Algorithm

- Create a fully connected three layer network. Initialize weights.
- Until all examples produce the correct output within ϵ (or other criteria)

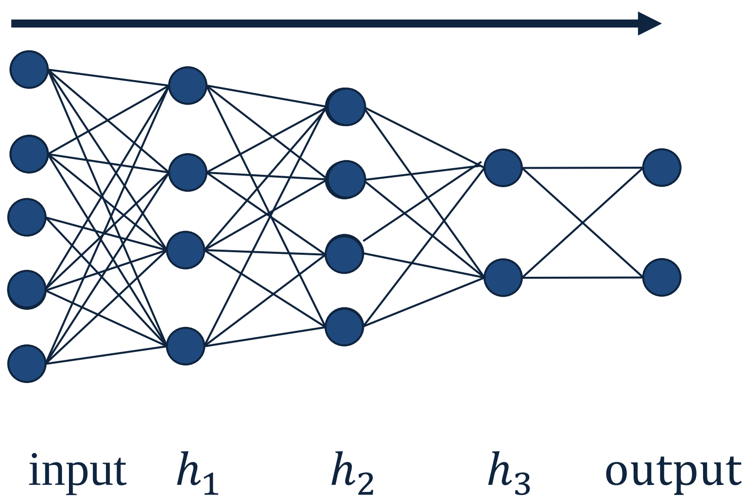
For each example in the training set do:

1. Compute the network output for this example
2. Compute the error between the output and target value
$$\delta_k = (t_k - o_k) o_k (1 - o_k)$$
1. For each output unit k, compute error term
$$\delta_j = o_j(1 - o_j). \sum_{k \in downstream(j)} -\delta_k \; w_{jk}$$
1. For each hidden unit, compute error term:
$$\Delta w_{ij} = R \delta_j x_{ij}$$
1. Update network weights

End epoch

More Hidden Layers

- The same algorithm holds for more hidden layers.



Comments on Training

- **No guarantee of convergence**; may **oscillate** or reach a local minima.
- In practice, many large networks can be trained on **large amounts of data** for realistic problems.

Comments on Training

- **No guarantee of convergence**; may **oscillate** or reach a local minima.
- In practice, many large networks can be trained on **large amounts of data** for realistic problems.
- **Many epochs** (tens of thousands) may be needed for adequate training. Large data sets may require many hours of CPU

Comments on Training

- **No guarantee of convergence**; may **oscillate** or reach a local minima.
- In practice, many large networks can be trained on **large amounts of data** for realistic problems.
- **Many epochs** (tens of thousands) may be needed for adequate training. Large data sets may require many hours of CPU
- **Termination criteria**: Number of epochs; Threshold on training set error; No decrease in error; Increased error on a validation set.
- To **avoid local minima**: several trials with different random initial weights with majority or voting techniques

Over-training Prevention

- Running too many epochs may **over-train** the network and result in over-fitting. (improved result on training, decrease in performance on test set)

Over-training Prevention

- Running too many epochs may **over-train** the network and result in over-fitting. (improved result on training, decrease in performance on test set)
- Keep an **hold-out validation** set and test accuracy after every epoch
- Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.

Over-training Prevention

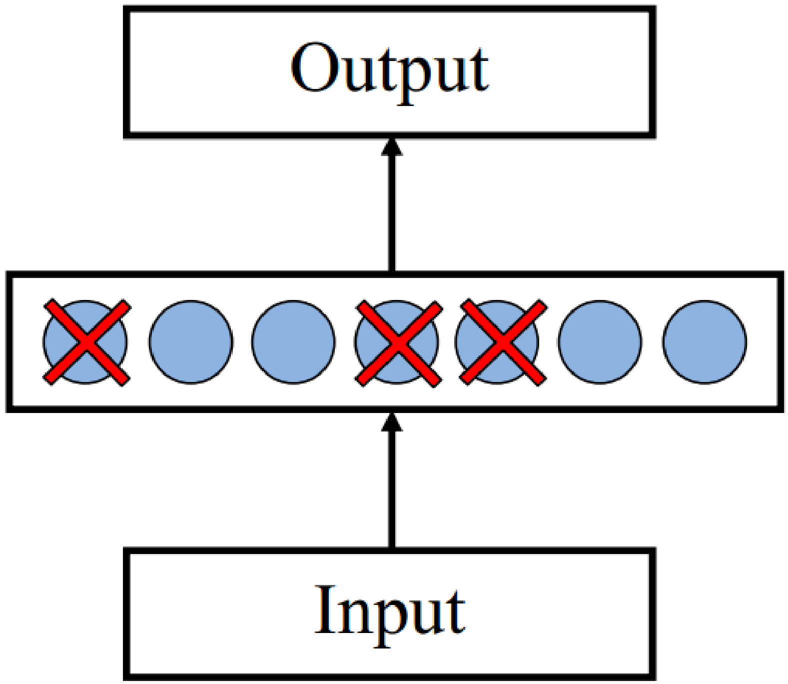
- Running too many epochs may **over-train** the network and result in over-fitting. (improved result on training, decrease in performance on test set)
- Keep an **hold-out validation** set and test accuracy after every epoch
- Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
- To avoid losing training data to validation:
  - Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance
  - Train on the full data set using this many epochs to produce the final results

Over-fitting prevention

- **Too few hidden units** prevent the system from adequately fitting the data and learning the concept.
- **Using too many hidden units** leads to over-fitting.
- Similar cross-validation method can be used to determine an appropriate number of hidden units. (general)
- Another approach to prevent over-fitting is weight-decay: all weights are multiplied by some fraction in (0,1) after every epoch.
  - Encourages smaller weights and less complex hypothesis
  - Equivalently: change Error function to include a term for the sum of the squares of the weights in the network. (general)

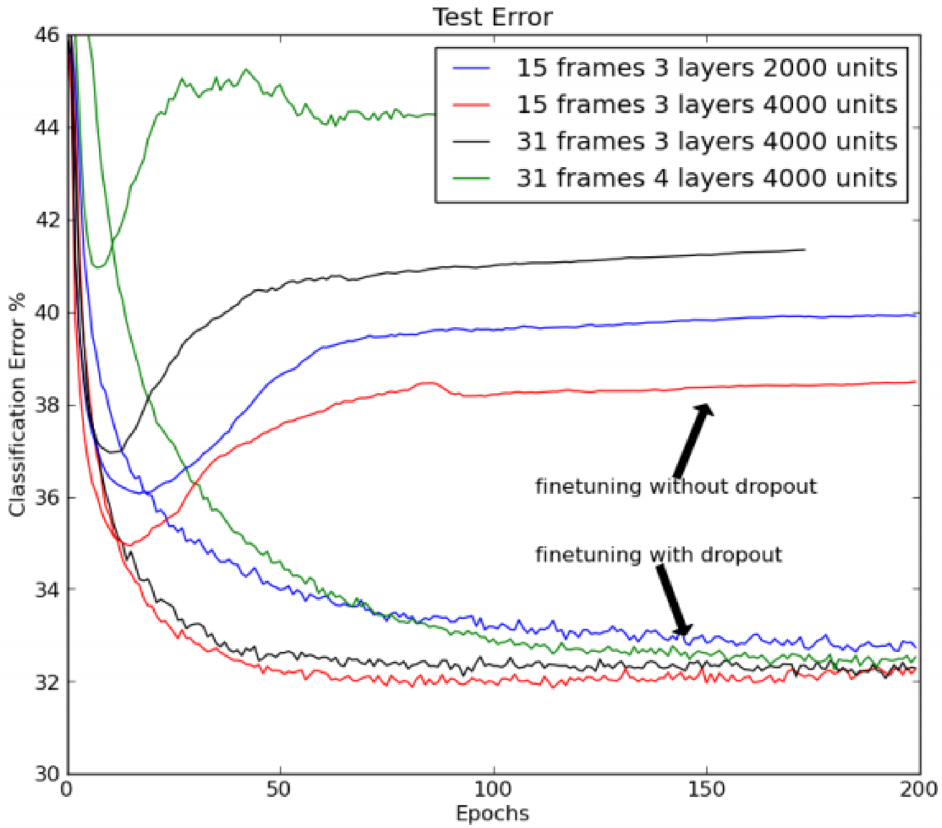
Dropout training

- Proposed by (Hinton et al, 2012)



- Each time decide whether to delete one hidden unit with some probability *p*

Dropout training



- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

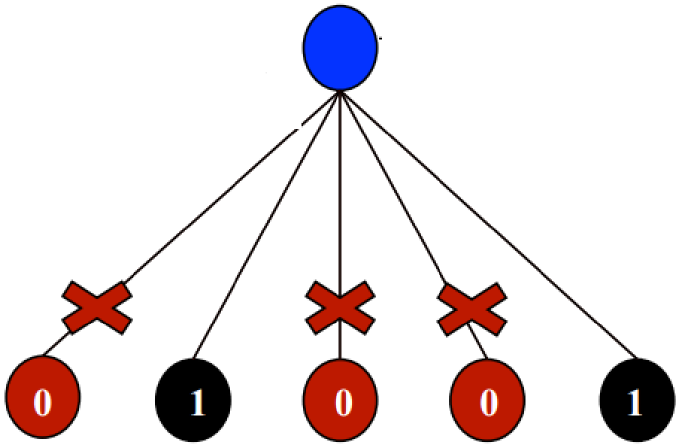
Dropout training

- Model averaging effect
  - Among 2^H models, with shared parameters
    - H: number of units in the network
  - Only a few get trained
  - Much stronger than the known regularizer

Dropout training

- Model averaging effect
  - Among 2^H models, with shared parameters
    - *H*: number of units in the network
  - Only a few get trained
  - Much stronger than the known regularizer

- What about the input space?
  - Do the same thing!



Input-Output Coding

- Appropriate coding of inputs and outputs can make learning problem easier and improve generalization.
- Encode each binary feature as a separate input unit;
- For multi-valued features include one binary unit per value rather than trying to encode input information in fewer units.
- For disjoint categorization problem, best to have one output unit for each category rather than encoding N categories into log N bits.

Representational Power

- The Backpropagation version presented is for networks with a single hidden layer,

But:

- Any Boolean function can be represented by a **two layer** network (simulate a two layer AND-OR network)
- Any **bounded continuous function** can be approximated with **arbitrary small error** by a **two layer** network.
- Sigmoid functions provide a set of **basis function** from which arbitrary function can be composed.
- **Any function** can be approximated to arbitrary accuracy by a **three layer** network.

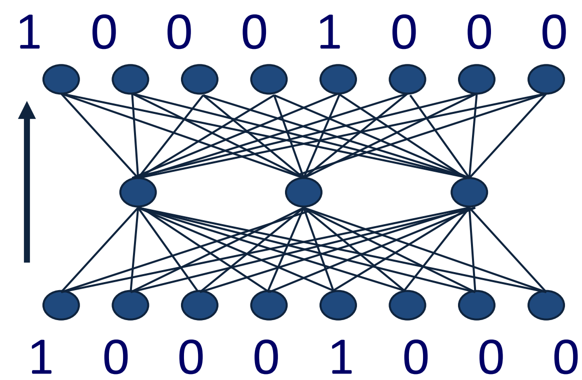
Hidden Layer Representation

- Weight tuning procedure sets weights that define whatever hidden units representation is most effective at minimizing the error.
- Sometimes Backpropagation will define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.
- Trained hidden units can be seen as newly constructed features that re-represent the examples so that they are linearly separable

Auto-associative Network

- An auto-associative network trained with 8 inputs, 3 hidden units and 8 output nodes, where the output must reproduce the input.
- When trained with vectors with only one bit on

INPUT	HIDDEN
1 0 0 0 0 0 0 0	.89 .40 0.8
0 1 0 0 0 0 0 0	.97 .99 .71
….	
0 0 0 0 0 0 0 1	.01 .11 .88



- Learned the standard 3-bit encoding for the 8 bit vectors.
- Illustrates also data compression aspects of learning