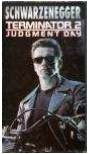
# ML (cont.): Perceptrons and Neural Networks

CS540 Bryan R Gibson University of Wisconsin-Madison

Slides adapted from those used by Prof. Jerry Zhu, CS540-1

# Terminator 2 (1991)




JOHN: Can you learn? So you can be... you know. More human. Not such a dork all the time.

**TERMINATOR:** My CPU is a neural-net processor... a learning computer. But **Skynet** presets the switch to "read-only" when we are sent out alone.

. . .

We'll learn how to **set** the neural net

**TERMINATOR** Basically. (starting the engine, backing out) The **Skynet** funding bill is passed. The system goes on-line August 4th, 1997. Human decisions are removed from strategic defense. **Skynet** begins to learn, at a geometric rate. It becomes **self-aware** at 2:14 a.m. eastern time, August 29. In a panic, they try to pull the plug.

SARAH: And **Skynet** fights back.

**TERMINATOR:** Yes. It launches its ICBMs against their targets in Russia.

SARAH: Why attack Russia?

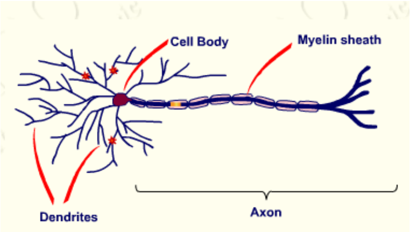
**TERMINATOR:** Because **Skynet** knows the Russian counter-strike will remove its enemies here.

# Outline

- Perceptron: a single neuron
  - Linear perceptron
  - Non-linear perceptron
  - Learning in a single perceptron
  - The power of a single perceptron
- Neural Network: a network of neurons
  - Layers, hidden units
  - Learning in a neural-network: backpropogation
  - The power of neural networks
  - Issues

- Everything revolves around gradient descent

# Biological Neurons




- Human brain: around a hundred trillion neurons
- Each neuron receives input from 1,000's of others
- Impulses arrive simultaneously
- Then they're added together
  - an impulse can either increase or decrease the possibility of a nerve pulse firing
- If sufficiently strong, a nerve pulse is generated
- The pulse becomes and input to other neurons

# Example: ALVINN

steering direction

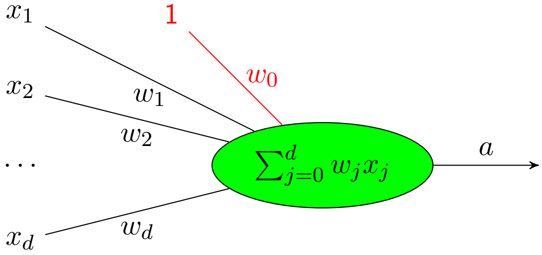
[Pomerleau, 1995]

# Linear Perceptron

- Perceptron: a math model for a single neuron
- Input: $x_1, \ldots, x_d$ (signals from other neurons)
- Weights: $w_1, \ldots, w_d$ (dendrites, can be negative!)
- We sneak in a constant (bias term) $x_0$, with weight $w_0$
- Activation function: linear (for now)

$$a = w_0 x_0 + w_1 x_1 + \ldots + w_d x_d$$

- This $a$ is the output of a linear perceptron



# Learning in a Linear Perceptron

- First, Regression: Training data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
- $\mathbf{x}_1$ is a vector $[x_{11}, \ldots x_1 d]$, same with $\mathbf{x}_2, \ldots, \mathbf{x}_n$
- $y$ is a real-valued output
- Goal : learn the weights $w_0, \ldots, w_d$ so that:
  given input $\mathbf{x}_i$, the output of the perceptron $a_i$ is close to $y_i$
- Need to define "close":

$$E = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

- $E$ is the "error": Given the training set, $E$ is a function of $w_0, \ldots, w_d$
- Want to minimize $E$: unconstrained optimization
  over variables $w_0, \ldots, w_d$

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

$$w_d \leftarrow w_d - \alpha \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

$$w_d \leftarrow w_d - \alpha \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

- Repeat until $E$ converges

# Learning in a Linear Perceptron (cont.)

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$
- $\alpha$ is a small constant, "learning rate" $=$ step size
- The gradient descent rule:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

$$w_d \leftarrow w_d - \alpha \sum_{i=1}^{n} (a_i - y_i) x_{id}$$

- Repeat until $E$ converges
- $E$ is convex in $\mathbf{w}$: there is a unique global minimum!

# The (Limited) Power of a Linear Perceptron

- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$

# The (Limited) Power of a Linear Perceptron

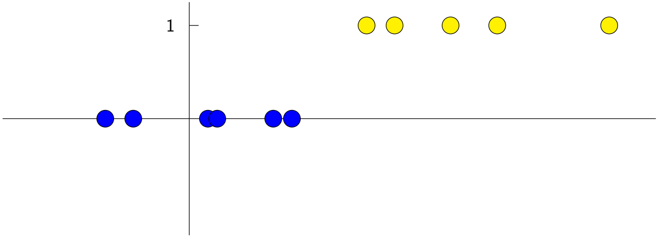
- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$
- it can represent any linear function in $d + 1$-dimensional space
  but that's it!

# The (Limited) Power of a Linear Perceptron

- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$
- it can represent any linear function in $d + 1$-dimensional space
  but that's it!
- In particular, it won't be a nice fit for binary classification (y={0,1})

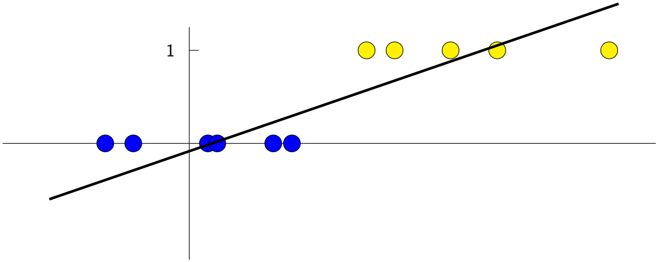
# The (Limited) Power of a Linear Perceptron

- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$
- it can represent any linear function in $d+1$-dimensional space
  but that's it!
- In particular, it won't be a nice fit for binary classification (y={0,1})



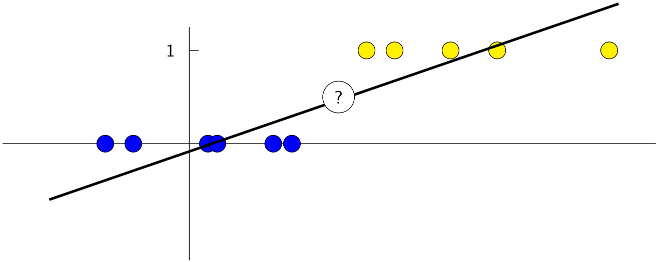
# The (Limited) Power of a Linear Perceptron

- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$
- it can represent any linear function in $d+1$-dimensional space
  but that's it!
- In particular, it won't be a nice fit for binary classification (y={0,1})



# The (Limited) Power of a Linear Perceptron

- Linear perceptron is just $a = \mathbf{w}'\mathbf{x}$
  where $\mathbf{x}$ is the input vector, augmented by $x_0 = 1$
- it can represent any linear function in $d + 1$-dimensional space
  but that's it!
- In particular, it won't be a nice fit for binary classification (y={0,1})

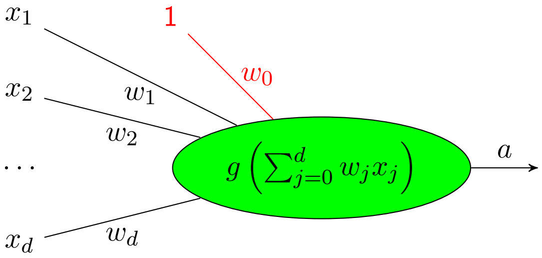


# Non-Linear Perceptron

- Change the activation function: use a step function

$$a = g(w_0 x_0 + w_1 x_1 + \ldots + w_d x_d)$$
$$g(h) = \mathbb{1}\{h \geq 0\}$$

- This is called a Linear Threshold Unit (LTU)

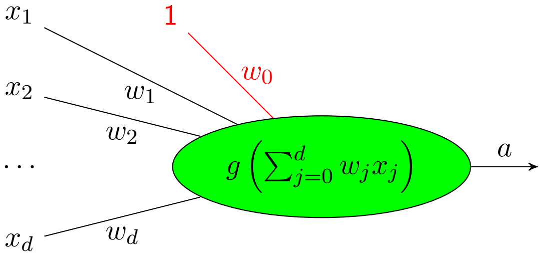



- Can you see how to make logical AND, OR, NOT functions using this perceptron?

# Non-Linear Perceptron: Linear Threshold Unit (LTU)

- Using a step function

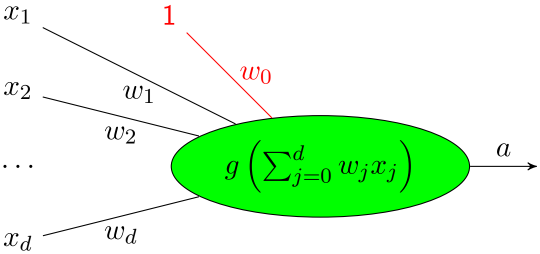
$$g(h) = \mathbb{1}\{h \geq 0\}$$



# Non-Linear Perceptron: Linear Threshold Unit (LTU)

- Using a step function

$$g(h) = \mathbb{1}\{h \geq 0\}$$

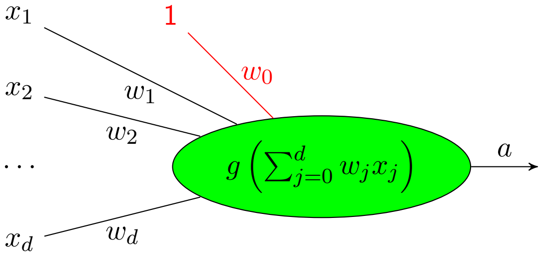



- AND: $w_1 = w_2 = 1,\ w_0 = -1.5$

# Non-Linear Perceptron: Linear Threshold Unit (LTU)

- Using a step function

$$g(h) = \mathbb{1}\{h \geq 0\}$$

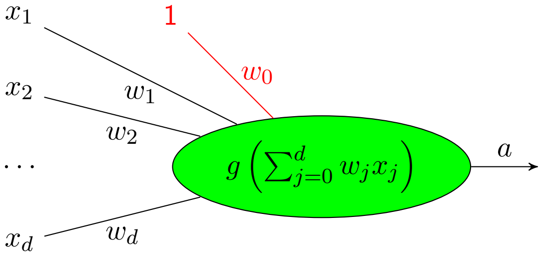



- AND: $w_1 = w_2 = 1,\ w_0 = -1.5$
- OR: $w_1 = w_2 = 1,\ w_0 = -0.5$

# Non-Linear Perceptron: Linear Threshold Unit (LTU)

- Using a step function

$$g(h) = \mathbb{1}\{h \geq 0\}$$



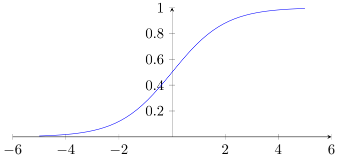

- AND: $w_1 = w_2 = 1,\ w_0 = -1.5$
- OR: $w_1 = w_2 = 1,\ w_0 = -0.5$
- NOT: $w_1 = -1,\ w_0 = 0.5$

# Non-Linear Perceptron: Sigmoid Activation Function

- The problem with LTU: step function is discontinuous, cannot use gradient descent!
- Change the activation function (again): use a sigmoid function

$$g(h) = \frac{1}{\left(1 + e^{(-h)}\right)}$$

- Exercise: $g'(h) = \ ?$

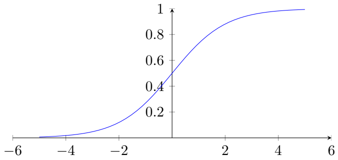


# Non-Linear Perceptron: Sigmoid Activation Function

- The problem with LTU: step function is discontinuous, cannot use gradient descent!
- Change the activation function (again): use a sigmoid function

$$g(h) = \frac{1}{\left(1 + e^{(-h)}\right)}$$

- Exercise: $g'(h) = g(h)\,(1 - g(h))$



# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(a_i - y_i)^2$$

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} (a_i - y_i)^2$$

- But now $a_i = g(\sum_d w_d x_{id})$ so we get

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(a_i - y_i)^2$$

- But now $a_i = g(\sum_d w_d x_{id})$ so we get

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(a_i - y_i)^2$$

- But now $a_i = g(\sum_d w_d x_{id})$ so we get

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

- The sigmoid perceptron update rule is then

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(a_i - y_i)^2$$

- But now $a_i = g(\sum_d w_d x_{id})$ so we get

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

- The sigmoid perceptron update rule is then

$$w_d \leftarrow w_d - \alpha\sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

# Learning in a Non-Linear Perceptron

- Again, we want to minimize the error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(a_i - y_i)^2$$

- But now $a_i = g(\sum_d w_d x_{id})$ so we get

$$\frac{\partial E}{\partial w_d} = \sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

- The sigmoid perceptron update rule is then

$$w_d \leftarrow w_d - \alpha\sum_{i=1}^{n}(a_i - y_i)a_i(1 - a_i)x_{id}$$

- Again, $\alpha$ is a small constant, the step size or learning rate
- Repeat until $E$ converges

# The (Limited) Power of a Non-Linear Perceptron

- Even with a non-linear sigmoid function, the only decision boundary a perceptron can produce is still linear.

# The (Limited) Power of a Non-Linear Perceptron

- Even with a non-linear sigmoid function, the only decision boundary a perceptron can produce is still linear.

- AND, OR, NOT revisited

# The (Limited) Power of a Non-Linear Perceptron

- Even with a non-linear sigmoid function, the only decision boundary a perceptron can produce is still linear.
- AND, OR, NOT revisited
- How about XOR

# The (Limited) Power of a Non-Linear Perceptron

- Even with a non-linear sigmoid function, the only decision boundary a perceptron can produce is still linear.

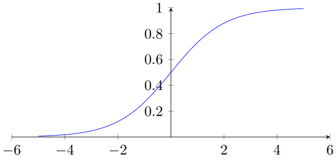
- AND, OR, NOT revisited

- How about XOR

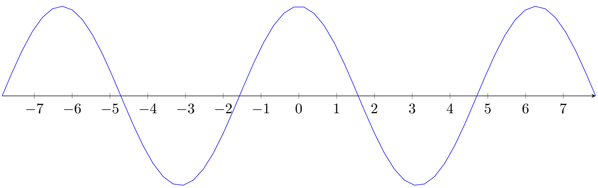
- This contributed to the first AI winter

# (Multi-Layer) Neural Networks

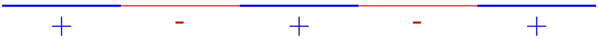
- Given sigmoid perceptrons ...




- can you produce output like ...

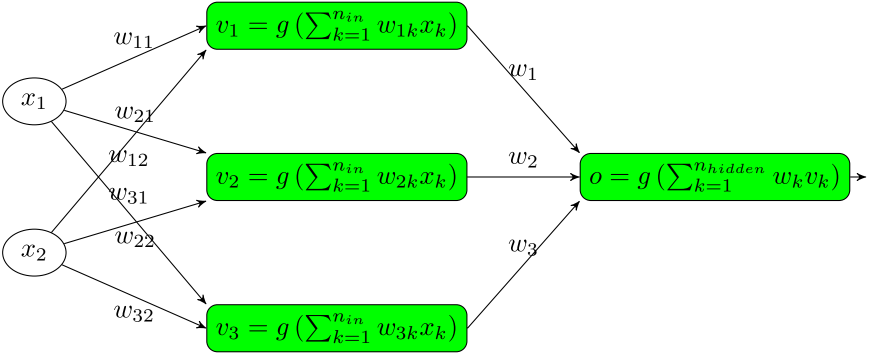



- which has a non-linear decision boundary?



# Mulit-Layer Neural Networks

- There are many ways to make a network of perceptrons.
- One standard way is multi-layer neural nets.
- 1 hidden layer (we can't see the output); 1 output layer

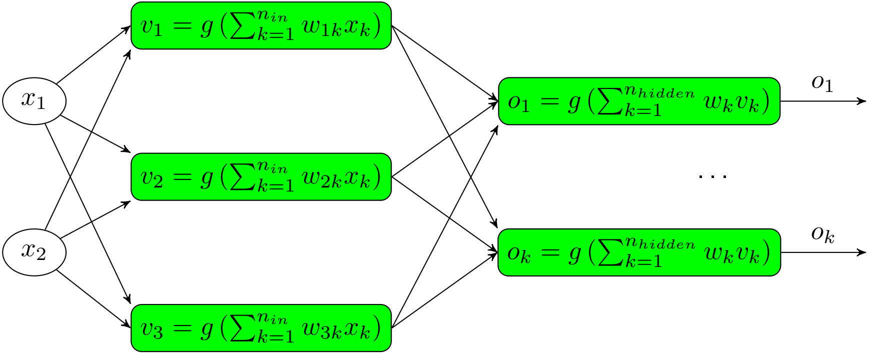


# The (Unlimited) Power of Neural Networks

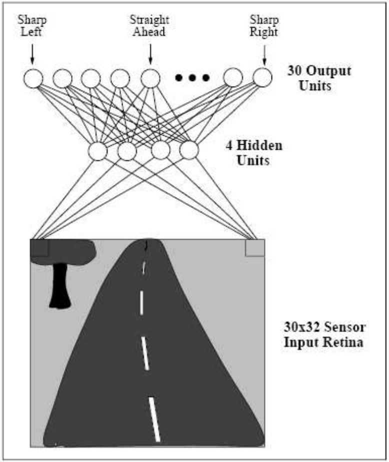
- In theory:
  - we don't need too many layers
  - 1 hidden-layer with enough hidden units can represent any continuous function of the inputs, with arbrirary accuracy
  - 2 hidden-layers can even represent discontinuous functions

# Neural Net for $k$-way Classification

- Use $k$ output units. During training:
  encode a label $y$ by an indicator vector with $k$ entries
- class1 = [1,0,...,0]', class2 = [0,1,0,...,0]', ...
- During test (decoding): choose the class corresponding to the output unit with largest activation

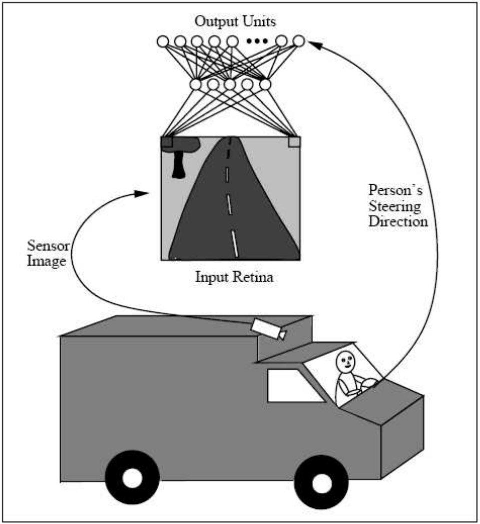


# Example Y encoding




[Pomerleau, 1995]

# Obtaining training data




[Pomerleau, 1995]

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{k}(o_{ic} - y_{ic})^2$$

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{k} (o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{k}(o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{k} (o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point
- $y_{ic}$: the $c$-th element of the $i$-th label indicator vector

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{k} (o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point
- $y_{ic}$: the $c$-th element of the $i$-th label indicator vector
- Our variables are all weights $w$ on all edges

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{k}(o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point
- $y_{ic}$: the $c$-th element of the $i$-th label indicator vector
- Our variables are all weights $w$ on all edges
  - Problem? : We don't know the "correct" output of hidden units

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{k}(o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point
- $y_{ic}$: the $c$-th element of the $i$-th label indicator vector
- Our variables are all weights $w$ on all edges
  - Problem? : We don't know the "correct" output of hidden units
  - Turns out to be ok: we can still do gradient descent.
    The trick is to use the chain rule

# Learning in a Neural Network

- Again, we minimize the error (for $k$ outputs):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{k} (o_{ic} - y_{ic})^2$$

- $i$: the $i$-th training point
- $o_{ic}$: the $c$-th output for the $i$-th training point
- $y_{ic}$: the $c$-th element of the $i$-th label indicator vector
- Our variables are all weights $w$ on all edges
  - Problem? : We don't know the "correct" output of hidden units
  - Turns out to be ok: we can still do gradient descent.
    The trick is to use the chain rule
- The algorithm: back-propogation

# The Back-Propogation Algorithm (Part 1)

BACKPROPOGATION(training set, $d$, $k$, $\alpha$, $n_{hid}$)

- Training set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
  $\mathbf{x}_i$: a feature vector of size $d$
  $y_i$: an output vector of size $k$
  $\alpha$: learning rate (step size)
  $n_{hid}$: number of hidden units

# The Back-Propogation Algorithm (Part 1)

BACKPROPOGATION(training set, $d$, $k$, $\alpha$, $n_{hid}$)

- Training set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
  $\mathbf{x}_i$: a feature vector of size $d$
  $y_i$: an output vector of size $k$
  $\alpha$: learning rate (step size)
  $n_{hid}$: number of hidden units
- Create neural network: $d$ inputs, $n_{hid}$ hidden units, $k$ outputs.

# The Back-Propogation Algorithm (Part 1)

BACKPROPOGATION(training set, $d$, $k$, $\alpha$, $n_{hid}$)

- Training set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
  $\mathbf{x}_i$: a feature vector of size $d$
  $y_i$: an output vector of size $k$
  $\alpha$: learning rate (step size)
  $n_{hid}$: number of hidden units
- Create neural network: $d$ inputs, $n_{hid}$ hidden units, $k$ outputs.
- Initialize weights to small random numbers (e.g. in $[-0.05, 0.05]$)

# The Back-Propogation Algorithm (Part 1)

BACKPROPOGATION(training set, $d$, $k$, $\alpha$, $n_{hid}$)

- Training set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
  $\mathbf{x}_i$: a feature vector of size $d$
  $y_i$: an output vector of size $k$
  $\alpha$: learning rate (step size)
  $n_{hid}$: number of hidden units
- Create neural network: $d$ inputs, $n_{hid}$ hidden units, $k$ outputs.
- Initialize weights to small random numbers (e.g. in $[-0.05, 0.05]$)
- Repeat (Part 2) until termination condition is met . . .

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network
- $\leftarrow$ : Propogate errors backward through the network

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network
- $\leftarrow$ : Propogate errors backward through the network
  - for each output unit $c$, compute its error term $\delta_c$

$$\delta_c \leftarrow (o_c - y_c)o_c(1 - o_c)$$

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network
- $\leftarrow$ : Propogate errors backward through the network
  - for each output unit $c$, compute its error term $\delta_c$

    $$\delta_c \leftarrow (o_c - y_c) o_c (1 - o_c)$$

  - for each hidden unit $h$, compute its error term $\delta_h$

    $$\delta_h \leftarrow \left( \sum_{i \in \mathsf{succ}(h)} w_{ih} \delta_i \right) o_h (1 - o_h)$$

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network
- $\leftarrow$ : Propogate errors backward through the network
  - for each output unit $c$, compute its error term $\delta_c$

    $$\delta_c \leftarrow (o_c - y_c) o_c (1 - o_c)$$

  - for each hidden unit $h$, compute its error term $\delta_h$

    $$\delta_h \leftarrow \left( \sum_{i \in \mathsf{succ}(h)} w_{ih} \delta_i \right) o_h (1 - o_h)$$

  - update each weight $w_{ji}$

    $$w_{ji} \leftarrow w_{ji} - \alpha \delta_j x_{ji}$$

# The Back-Propogation Algorithm (Part 2)

For each training example $(\mathbf{x}, y)$:

- $\rightarrow$ : Propogate input forward through the network:
  Input $\mathbf{x}$ and compute output $o_u$ for every unit $u$ in network
- $\leftarrow$ : Propogate errors backward through the network
  - for each output unit $c$, compute its error term $\delta_c$

    $$\delta_c \leftarrow (o_c - y_c) o_c (1 - o_c)$$

  - for each hidden unit $h$, compute its error term $\delta_h$

    $$\delta_h \leftarrow \left( \sum_{i \in \mathsf{succ}(h)} w_{ih} \delta_i \right) o_h (1 - o_h)$$

  - update each weight $w_{ji}$

    $$w_{ji} \leftarrow w_{ji} - \alpha \delta_j x_{ji}$$

    - $x_{ji}$: input from unit $i$ into $j$
      ($o_i$ if $i$ is hidden unit; $\mathbf{x}_i$ if $i$ is an input)
    - $w_{ji}$: weight from unit $i$ to unit $j$

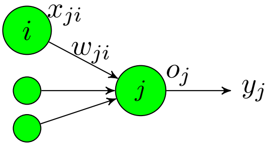
# Derivation of Back-Propagation

- For simplicity, assume online learning (vs. batch learning): 1-step grad. descent after seeing ea. training example $(\mathbf{x}, y)$
- For each $(\mathbf{x}, y)$ the error is

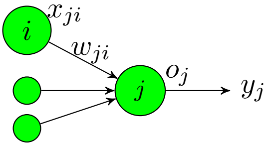
$$E(\mathbf{w}) = \frac{1}{2}\sum_{c=1}^{k}(o_c - y_c)^2$$

  - $o_c$: the $c$-th output unit (when input is $\mathbf{x}$)
  - $y_c$: the $c$-th element of label indicator vector
- Use grad. descent to change all weights $w_{ji}$ to minimize error.
- Separate into two cases:
  - Case 1: $w_{ji}$ when $j$ is output unit
  - Case 2: $w_{ji}$ when $j$ is hidden unit

# Case 1: Weights for Output Unit



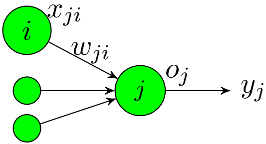
# Case 1: Weights for Output Unit




$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}\left(g\left[\sum_m w_{jm} x_{jm}\right] - y_j\right)^2}{\partial w_{ji}}$$

$$= (o_j - y_j) o_j (1 - o_j) x_{ji}$$

# Case 1: Weights for Output Unit

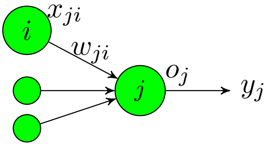



$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}\left(g\left[\sum_m w_{jm} x_{jm}\right] - y_j\right)^2}{\partial w_{ji}}$$

$$= (o_j - y_j) o_j (1 - o_j) x_{ji}$$

- $o_c$: the $c$-th output unit (when input is $\mathbf{x}$)

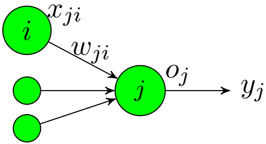
# Case 1: Weights for Output Unit




$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}\left(g\left[\sum_m w_{jm} x_{jm}\right] - y_j\right)^2}{\partial w_{ji}}$$
$$= (o_j - y_j) o_j (1 - o_j) x_{ji}$$

- $o_c$: the $c$-th output unit (when input is $\mathbf{x}$)
- $y_c$: the $c$-th element of label indicator vector

# Case 1: Weights for Output Unit

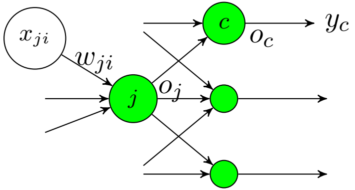



$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}\left(g\left[\sum_m w_{jm}x_{jm}\right] - y_j\right)^2}{\partial w_{ji}}$$
$$= (o_j - y_j)o_j(1 - o_j)x_{ji}$$

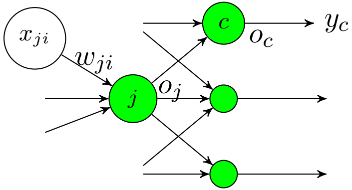
- $o_c$: the $c$-th output unit (when input is $\mathbf{x}$)
- $y_c$: the $c$-th element of label indicator vector
- grad. descent: to min. error, head away from part. derivative:

$$w_{ji} \leftarrow w_{ji} - \alpha \frac{\partial E}{\partial w_{ji}} = w_{ji} - \alpha(o_j - y_j)o_j(1 - o_j)x_{ji}$$

# Case 2: Weights for Hidden Unit

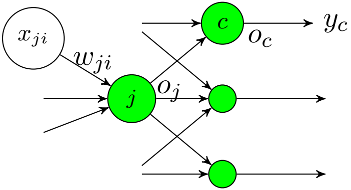


# Case 2: Weights for Hidden Unit




$$\frac{\partial E}{\partial w_{ji}} = \sum_{c \in \mathsf{succ}(j)} \left[ \frac{\partial E_c}{\partial o_c} \cdot \frac{\partial o_c}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}} \right]$$

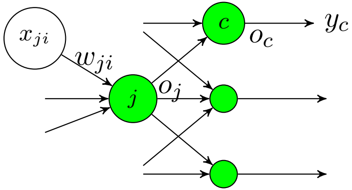
# Case 2: Weights for Hidden Unit




$$\frac{\partial E}{\partial w_{ji}} = \sum_{c \in \textsf{succ}(j)} \left[ \frac{\partial E_c}{\partial o_c} \cdot \frac{\partial o_c}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}} \right]$$

$$= \sum_{c \in \textsf{succ}(j)} \left[ (o_c - y_c) \cdot \frac{\partial g\left(\sum_m w_{cm} x_{cm}\right)}{\partial x_{cm}} \cdot \frac{\partial g\left(\sum_n w_{jn} x_{jn}\right)}{\partial w_{ji}} \right]$$

# Case 2: Weights for Hidden Unit




$$\frac{\partial E}{\partial w_{ji}} = \sum_{c \in \mathsf{succ}(j)} \left[ \frac{\partial E_c}{\partial o_c} \cdot \frac{\partial o_c}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}} \right]$$

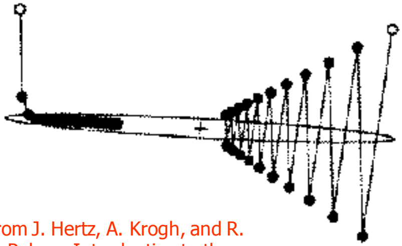
$$= \sum_{c \in \mathsf{succ}(j)} \left[ (o_c - y_c) \cdot \frac{\partial g\left(\sum_m w_{cm} x_{cm}\right)}{\partial x_{cm}} \cdot \frac{\partial g\left(\sum_n w_{jn} x_{jn}\right)}{\partial w_{ji}} \right]$$

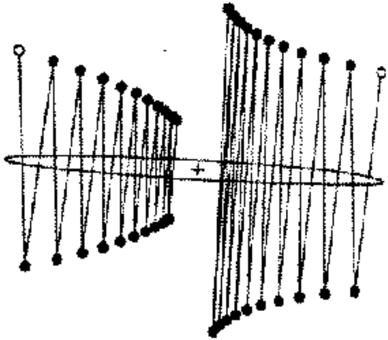
$$= \sum_{c \in \mathsf{succ}(j)} \left[ (o_c - y_c) \cdot o_c(1 - o_c) w_{cj} \cdot o_j(1 - o_j) x_{ji} \right]$$

# Neural Network Learning Issues: Weights

- When to terminate back-prop.? Overfitting and early-stopping
  - After fixed number of iterations (ok)
  - When training error less than a threshold (not ok!)
  - When holdout set error starts to go up (ok)
- Local Optima:
  - Weights will converge to local minimum
- Learning Rate:
  - Convergence sensitive to learning rate
  - Weight learning can be slow

# Sensitivity to learning rate





From J. Hertz, A. Krogh, and R. G. Palmer. Introduction to the Theory of Neural Computation. Addison-Wesley, 1994.

FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of $\eta$, which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

# Neural Network Learning Issues: Weights (cont.)

- Use "momentum" (a heuristic?) to dampen grad. descent

$$\Delta \mathbf{w}^{(t-1)} = \text{ previous change to } \mathbf{w}$$
$$\Delta \mathbf{w}^{(t)} = -\alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} + \beta \Delta \mathbf{w}^{(t-1)}$$
$$\mathbf{w} \leftarrow \mathbf{w} + \Delta w^{(t)}$$

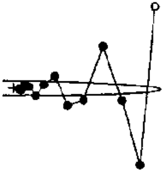


**FIGURE 6.3** Gradient descent on the simple quadratic surface of Fig. 5.10. Both trajectories are for 12 steps with $\eta = 0.0476$, the best value in the absence of momentum. On the left there is no momentum ($\alpha = 0$), while $\alpha = 0.5$ on the right.

- Alternatives to gradient descent:
  Newton-Raphson, Conjugate Gradient

# Neural Network Learning Issues: Structure

- How many hidden units?
- How many layers?
- How to connect units?

- Cross validation