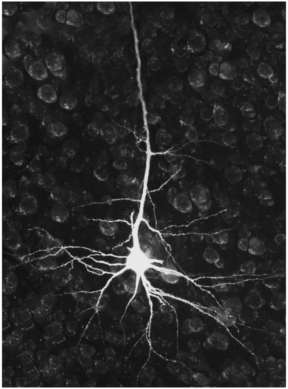
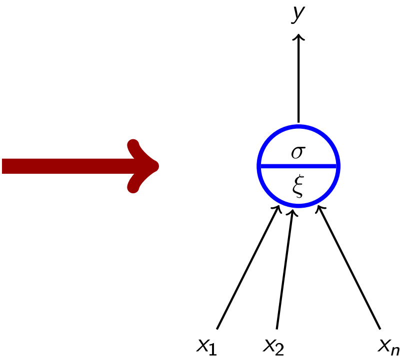
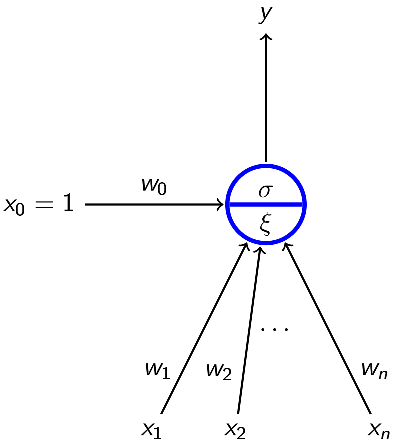
(Primitive) Mathematical Model of Neuron



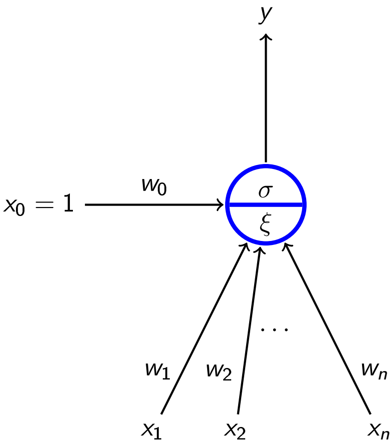


Formal neuron



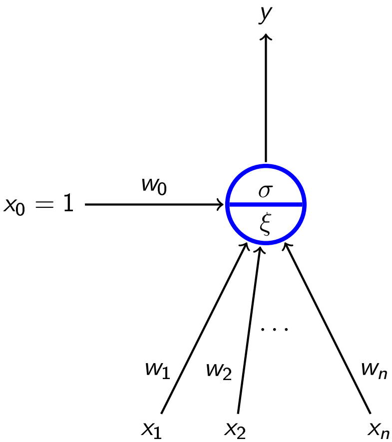
- $x_1, \ldots, x_n$ real *inputs*

Formal neuron



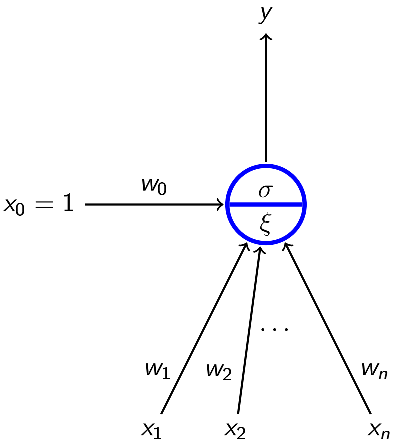
- $x_1, \dots, x_n$ real *inputs*
- x_0 special input, always 1

Formal neuron



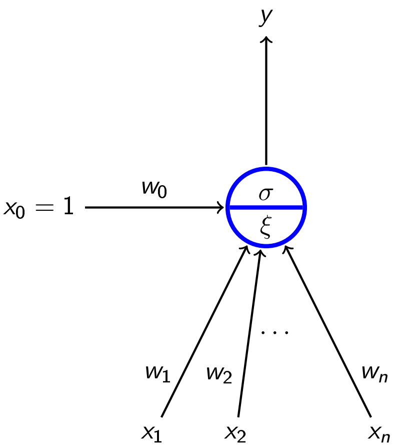
- $x_1, \dots, x_n$ real *inputs*
- x_0 special input, always 1
- $w_0, w_1, \dots, w_n$ real *weights*

Formal neuron



- $x_1, \dots, x_n$ real *inputs*
- x_0 special input, always 1
- $w_0, w_1, \dots, w_n$ real *weights*
- $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$ *inner potential*;
  In general, other potentials are considered (e.g. Gaussian), more on this in PV021.

Formal neuron



- $x_1, \ldots, x_n$ real *inputs*
- x_0 special input, always 1
- $w_0, w_1, \ldots, w_n$ real *weights*
- $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$ *inner potential*;
  In general, other potentials are considered (e.g. Gaussian), more on this in PV021.
- y *output* defined by $y = \sigma(\xi)$ where σ is an *activation function*.
  We consider several activation functions.

  e.g. *linear threshold function*

$$\sigma(\xi) = sgn(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Formal Neuron vs Linear Models

Both linear classifier and linear (affine) function are special cases of the formal neuron.

- If σ is a linear threshold function

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

  we obtain a linear classifier.
- If σ is identity, i.e. $\sigma(\xi) = \xi$, we obtain a linear (affine) function.

Formal Neuron vs Linear Models

Both linear classifier and linear (affine) function are special cases of the formal neuron.

- If σ is a linear threshold function

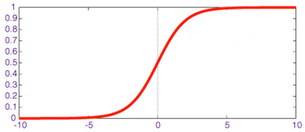
  $$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

  we obtain a linear classifier.
- If σ is identity, i.e. $\sigma(\xi) = \xi$, we obtain a linear (affine) function.

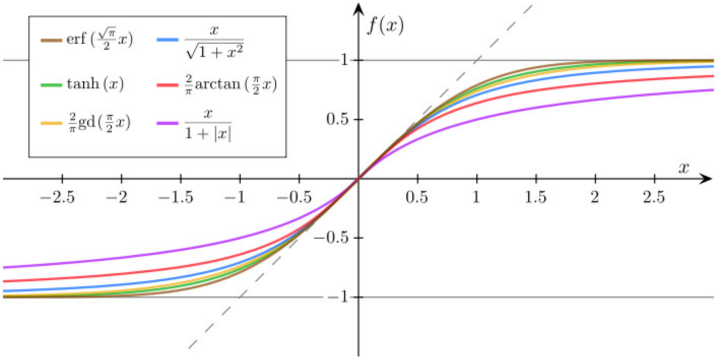
Many more activation functions are used in neural networks!

Sigmoid Functions

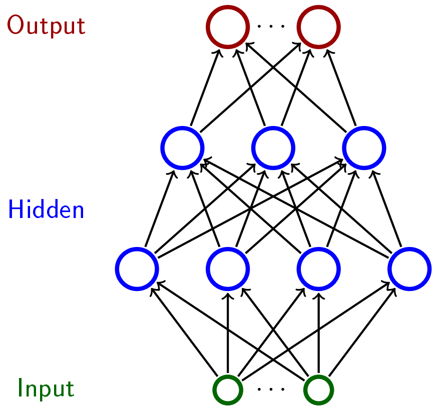
Logistic sigmoid $\sigma(\xi) = \dfrac{1}{1 + e^{-\xi}}$



Others ...

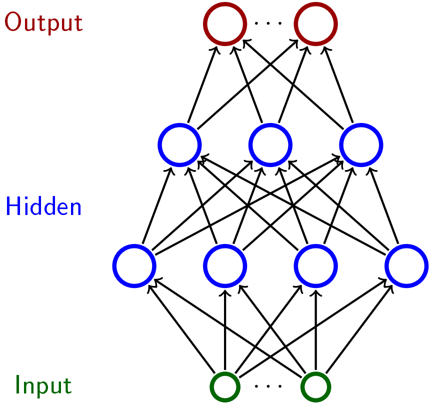


Multilayer Perceptron (MLP)



- Neurons are organized in *layers* (input layer, output layer, possibly several hidden layers)
- Layers are numbered from 0; the input is 0-th
- Neurons in the ℓ-th layer are connected with all neurons in the $\ell + 1$-th layer

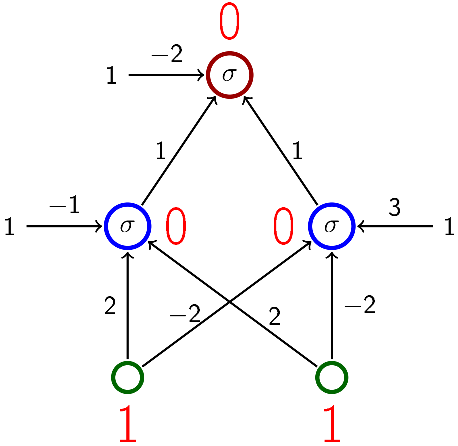
Multilayer Perceptron (MLP)



- Neurons are organized in *layers*
  (input layer, output layer, possibly several hidden layers)
- Layers are numbered from 0;
  the input is 0-th
- Neurons in the ℓ-th layer are connected with all neurons in the $\ell+1$-th layer

Intuition: The network computes a function as follows: Assign input values to the input neurons and 0 to the rest. Proceed upwards through the layers, one layer per step. In the ℓ-th step consider output values of neurons in $\ell-1$-th layer as inputs to neurons of the ℓ-th layer. Compute output values of neurons in the ℓ-th layer.

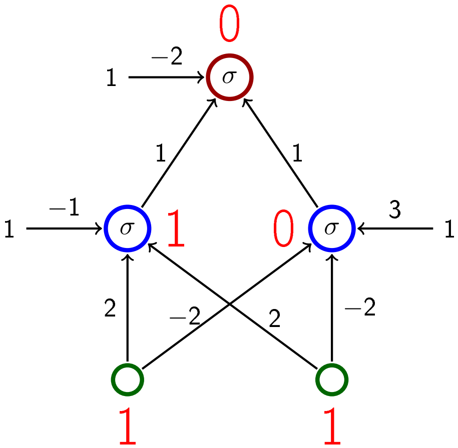
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

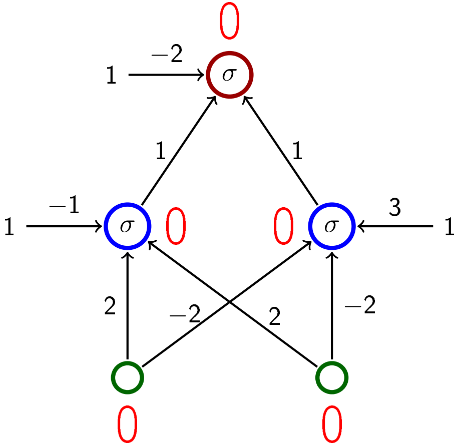
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

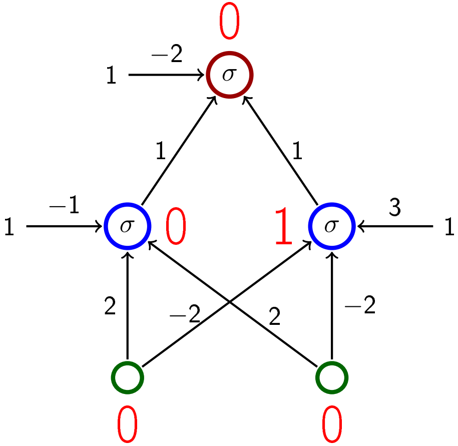
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

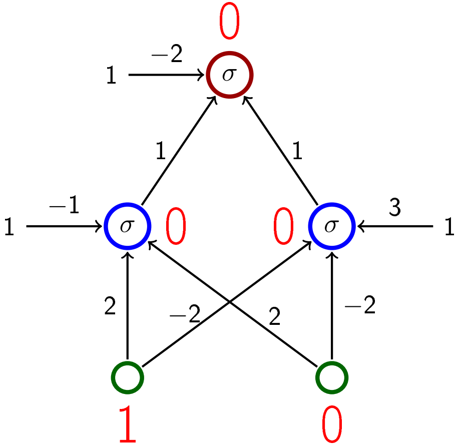
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

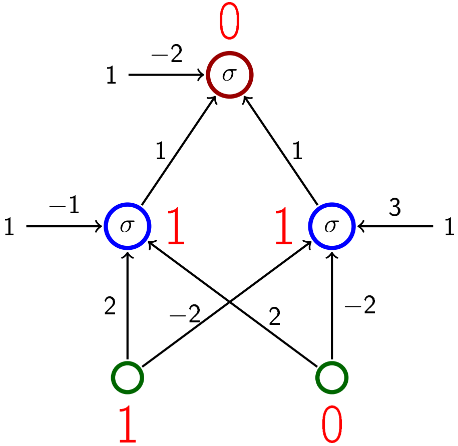
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

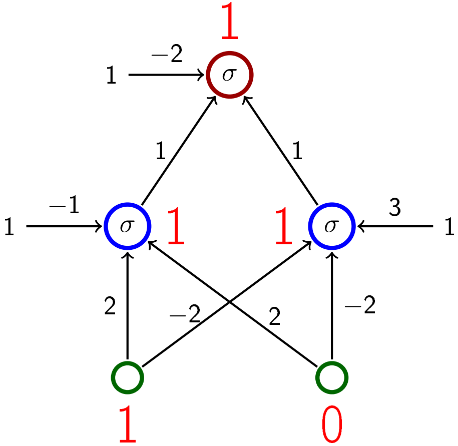
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

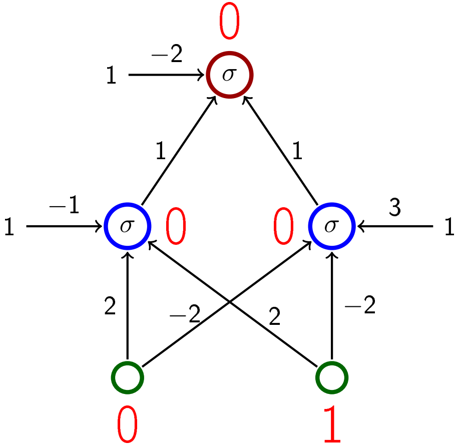
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

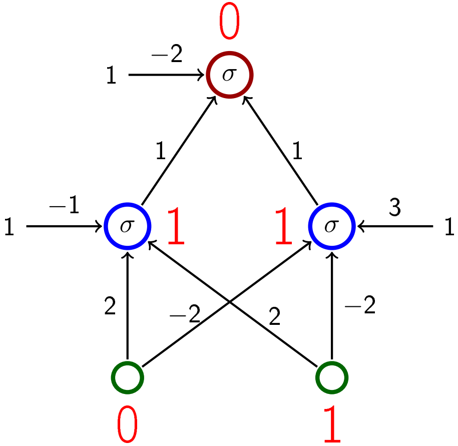
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

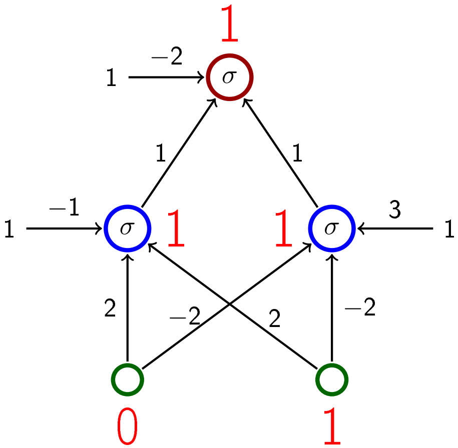
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

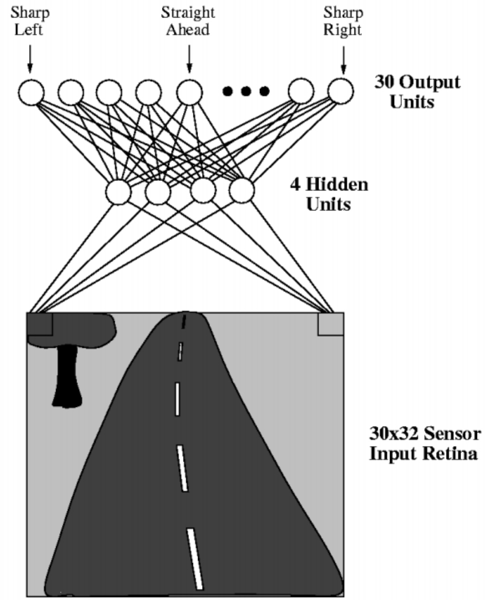
Example



- Activation function: linear threshold

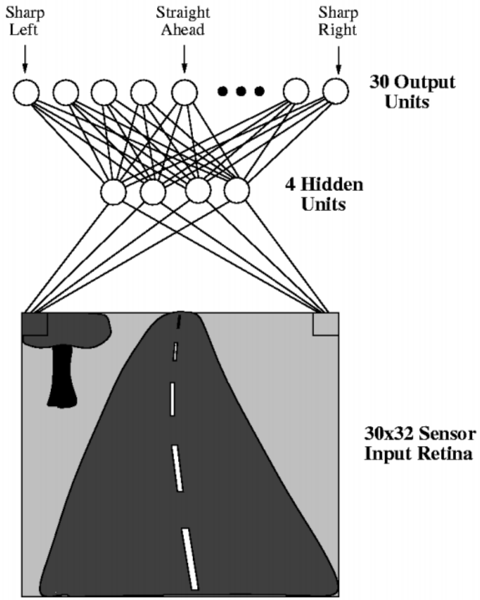
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Classical Example – ALVINN



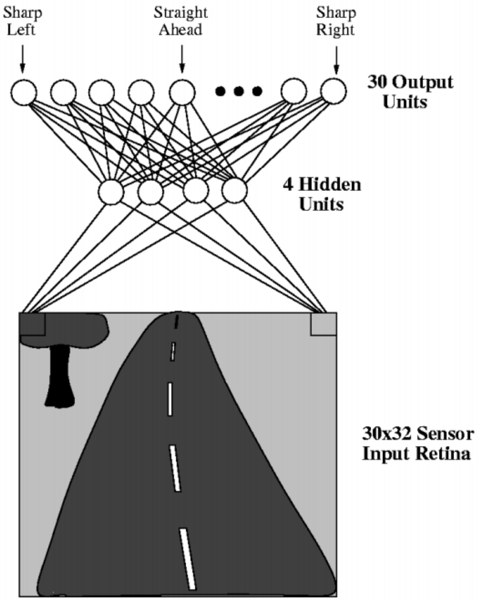
- One of the first autonomous car driving systems (in 90s)
- ALVINN drives a car

Classical Example – ALVINN



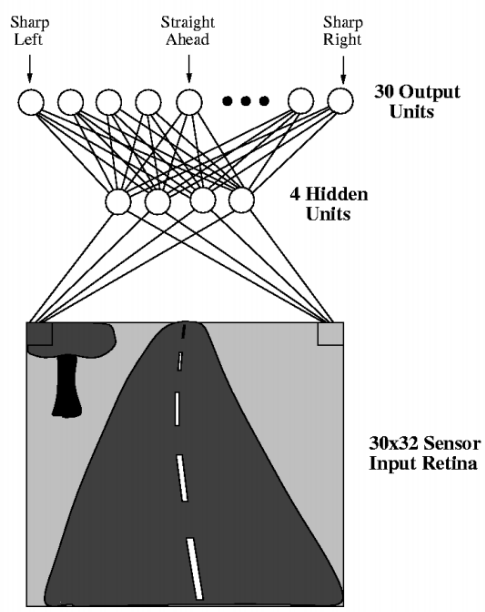
- One of the first autonomous car driving systems (in 90s)
- ALVINN drives a car
- The net has $30 \times 32 = 960$ input neurons (the input space is $\mathbb{R}^{960}$).

Classical Example – ALVINN



- One of the first autonomous car driving systems (in 90s)
- ALVINN drives a car
- The net has $30 \times 32 = 960$ input neurons (the input space is $\mathbb{R}^{960}$).
- The value of each input captures the shade of gray of the corresponding pixel.

Classical Example – ALVINN

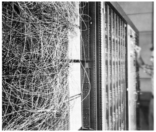


- One of the first autonomous car driving systems (in 90s)
- ALVINN drives a car
- The net has $30 \times 32 = 960$ input neurons (the input space is $\mathbb{R}^{960}$).
- The value of each input captures the shade of gray of the corresponding pixel.
- Output neurons indicate where to turn (to the center of gravity).

Source: `http://jmvidal.cse.sc.edu/talks/ann/alvin.html`

A Bit of History

- Perceptron (Rosenblatt et al, 1957)

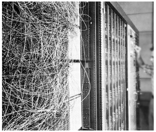


- Single layer (i.e. no hidden layers), the activation function *linear threshold*

  (i.e., a bit more general linear classifier)
- Perceptron learning algorithm
- Used to recognize numbers

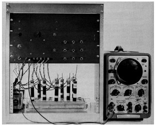
A Bit of History

- Perceptron (Rosenblatt et al, 1957)



  - Single layer (i.e. no hidden layers), the activation function *linear threshold*

    (i.e., a bit more general linear classifier)
  - Perceptron learning algorithm
  - Used to recognize numbers
- Adaline (Widrow & Hof, 1960)



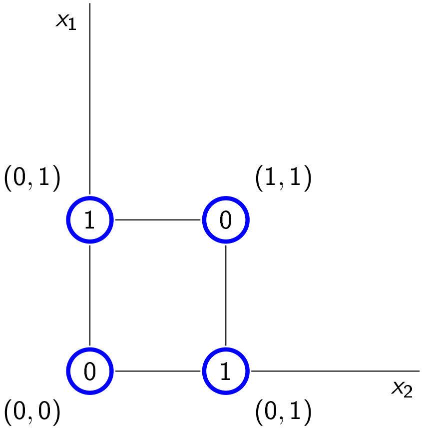
  - Single layer, the activation function *identity*

    (i.e., a bit more linear function)
  - Online version of the gradient descent
  - Used a new circuitry element called *memristor* which was able to "remember"history of current in form of resistance

In both cases, the expressive power is rather limited – can express only linear decision boundaries and linear (affine) functions.

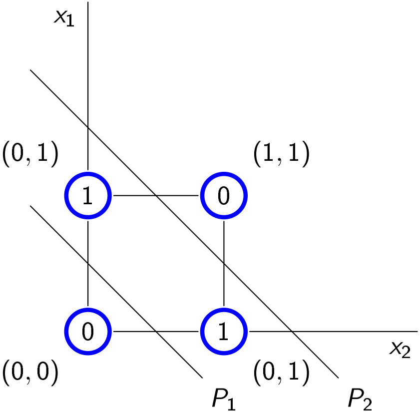
A Bit of History

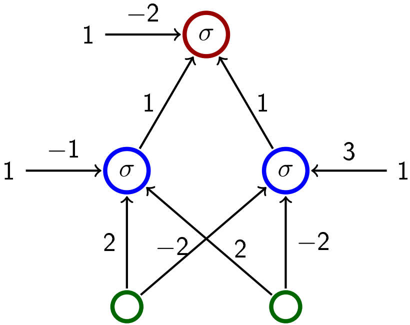
One of the famous (counter)-examples: XOR



No perceptron can distinguish between ones and zeros.

XOR vs Multilayer Perceptron





(Here σ is a linear threshold function.)

$P_1 : -1 + 2x_1 + 2x_2 = 0$ $\qquad$ $P_2 : 3 - 2x_1 - 2x_2 = 0$

The output neuron performs an intersection of half-spaces.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function with range $(0, 1)$.

Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function with range $(0, 1)$.

  Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

So multi-layer perceptrons are suffuciently powerful for any application.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function with range $(0, 1)$.

  Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

So multi-layer perceptrons are suffuciently powerful for any application.

But for a long time, at least throughout 60s and 70s, nobody well-known knew any efficient method for training multilayer networks!

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function with range $(0, 1)$.

  Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

So multi-layer perceptrons are suffuciently powerful for any application.

But for a long time, at least throughout 60s and 70s, nobody well-known knew any efficient method for training multilayer networks!

... then the **backpropagation** appeared in 1986!

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)
- w_{ji} is the weight of the arc **from** the neuron i **to** the neuron j.

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)
- w_{ji} is the weight of the arc **from** the neuron i **to** the neuron j.
- $j_{\leftarrow}$ is the set of all neurons from which there are edges to j
  (i.e. $j_{\leftarrow}$ is the layer directly below j)

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)
- w_{ji} is the weight of the arc **from** the neuron i **to** the neuron j.
- $j_{\leftarrow}$ is the set of all neurons from which there are edges to j
  (i.e. $j_{\leftarrow}$ is the layer directly below j)
- $j^{\rightarrow}$ is the set of all neurons to which there are edges from j.
  (i.e. $j^{\rightarrow}$ is the layer directly above j)

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- for simplicity, the activation function of every neuron will be the logistic sigmoid $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$.
  (We may of course consider logistic sigmoids with different steepness paramaters, or other sigmoidal functions, more in PV021.)

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- for simplicity, the activation function of every neuron will be the logistic sigmoid $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$.

  (We may of course consider logistic sigmoids with different steepness paramaters, or other sigmoidal functions, more in PV021.)

- A value of a non-input neuron $j \in Z \setminus X$ when the computation is finished is $y_j = \sigma(\xi_j)$

  (y_j is determined by weights $\vec{w}$ and a given input $\vec{x}$, so it's sometimes written as $y_j[\vec{w}](\vec{x})$)

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- for simplicity, the activation function of every neuron will be the logistic sigmoid $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$.

  (We may of course consider logistic sigmoids with different steepness paramaters, or other sigmoidal functions, more in PV021.)

- A value of a non-input neuron $j \in Z \setminus X$ when the computation is finished is $y_j = \sigma(\xi_j)$

  (y_j is determined by weights $\vec{w}$ and a given input $\vec{x}$, so it's sometimes written as $y_j[\vec{w}](\vec{x})$)

- Fixing weights of all neurons, the network computes a function $F[\vec{w}] : \mathbb{R}^{|X|} \to \mathbb{R}^{|Y|}$ as follows: Assign values of a given vector $\vec{x} \in \mathbb{R}^{|X|}$ to the input neurons, evaluate the network, then $F[\vec{w}](\vec{x})$ is the vector of values of the output neurons.

  Here we implicitly assume a fixed orderings on input and output vectors.

MLP – Learning

- Given a set D of training examples:

$$D = \left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $\vec{d}_k \in \mathbb{R}^{|Y|}$. We write d_{kj} to denote the value in $\vec{d}_k$ corresponding to the output neuron j.

MLP – Learning

- Given a set D of training examples:

$$D = \left\{ \left(\vec{x}_k, \vec{d}_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $\vec{d}_k \in \mathbb{R}^{|Y|}$. We write d_{kj} to denote the value in $\vec{d}_k$ corresponding to the output neuron j.

- **Least Squares Error Function:** Let $\vec{w}$ be a vector of all weights in the network.

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} (y_j[\vec{w}](\vec{x}_k) - d_{kj})^2$$

MLP – Learning Algorithm

Batch Learning – Gradient Descent:

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \ldots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0

MLP – Learning Algorithm

Batch Learning – Gradient Descent:

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \ldots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0
- in the step $t+1$ (here $t = 0, 1, 2\ldots$) is $\vec{w}^{(t+1)}$ computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

MLP – Learning Algorithm

Batch Learning – Gradient Descent:

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \ldots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$) is $\vec{w}^{(t+1)}$ computed as follows:

  $$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

  where

  $$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

  is the weight change w_{ji} and $0 < \varepsilon(t) \leq 1$ is the learning speed in the step $t+1$.

MLP – Learning Algorithm

Batch Learning – Gradient Descent:

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \ldots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$) is $\vec{w}^{(t+1)}$ computed as follows:

  $$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

  where

  $$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

  is the weight change w_{ji} and $0 < \varepsilon(t) \leq 1$ is the learning speed in the step $t+1$.

Note that $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$ is a component of ∇E, i.e. the weight change in the step $t+1$ can be written as follows: $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

MLP – Gradient Computation

For every weight w_{ji} we have (obviously)

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

So now it suffices to compute $\frac{\partial E_k}{\partial w_{ji}}$, that is the error for a fixed training example $(\vec{x}_k, d_k)$.

MLP – Gradient Computation

For every weight w_{ji} we have (obviously)

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

So now it suffices to compute $\frac{\partial E_k}{\partial w_{ji}}$, that is the error for a fixed training example $(\vec{x}_k, d_k)$.

It holds that

$$\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$$

where

$$\delta_j = y_j - d_{kj} \qquad \text{pro } j \in Y$$

$$\delta_j = \sum_{r \in j^{\rightarrow}} \delta_r \cdot y_r(1 - y_r) \cdot w_{rj} \qquad \text{pro } j \in Z \smallsetminus (Y \cup X)$$

(Here $y_r = y[\vec{w}](\vec{x}_k)$ where $\vec{w}$ are the current weights and $\vec{x}_k$ is the input of the k-th training example.)

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Compute all $\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$ for every training example $(\vec{x}_k, \vec{d}_k)$:

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Compute all $\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$ for every training example $(\vec{x}_k, \vec{d}_k)$:

- Evaluate all values y_i of neurons using the standard bottom-up procedure with the input $\vec{x}_k$.

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Compute all $\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$ for every training example $(\vec{x}_k, \vec{d}_k)$:

- Evaluate all values y_i of neurons using the standard bottom-up procedure with the input $\vec{x}_k$.
- Compute δ_j using *backpropagation* through layers top-down :
  - Assign $\delta_j = y_j - d_{kj}$ for all $j \in Y$

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Compute all $\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$ for every training example $(\vec{x}_k, \vec{d}_k)$:

- Evaluate all values y_i of neurons using the standard bottom-up procedure with the input $\vec{x}_k$.
- Compute δ_j using *backpropagation* through layers top-down :
  - Assign $\delta_j = y_j - d_{kj}$ for all $j \in Y$
  - In the layer ℓ, assuming that δ_r has been computed for all neurons r in the layer $\ell + 1$, compute

    $$\delta_j = \sum_{r \in j^{\rightarrow}} \delta_r \cdot y_r(1 - y_r) \cdot w_{rj}$$

    for all j from the ℓ-th layer.

Example

Assume $w_{30}^{(0)} = w_{50}^{(0)} = w_{41}^{(0)} = w_{42}^{(0)} = w_{54}^{(0)} = 1$ and $w_{40}^{(0)} = w_{31}^{(0)} = w_{32}^{(0)} = w_{53}^{(0)} = -1$. Consider a training set $\{((1,0),1)\}$.

Then

$y_1 = 1,$

$y_2 = 0,$

$y_3 = \sigma(w_{30} + w_{31}^{(0)} y_1 + w_{32}^{(0)} y_2) = 0.5,$

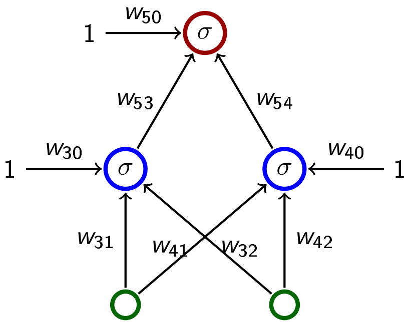
$y_4 = 0.5,$

$y_5 = 0.731058.$

$\delta_5 = y_5 - 1 = -0.268942,$

$\delta_4 = \delta_5 \cdot y_5 \cdot (1 - y_5) * w_{54}^{(0)} = -0.052877,$

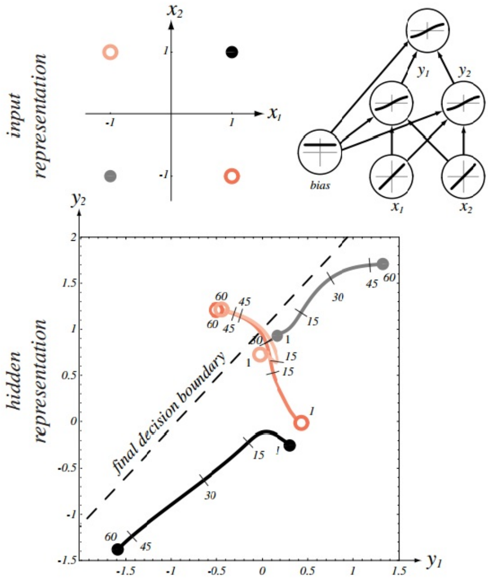
$\delta_3 = 0.052877.$



$$\frac{\partial E_1}{\partial w_{53}} = \delta_5 \cdot y_5 \cdot (1 - y_5) \cdot y_3 = -0.026438,$$
$$\frac{\partial E_1}{\partial w_{54}} = \delta_5 \cdot y_5 \cdot (1 - y_5) \cdot y_4 = -0.026438,$$
$$\frac{\partial E_1}{\partial w_{30}} = \delta_3 \cdot y_3 \cdot (1 - y_3) \cdot 1 = 0.01321925,$$

....

Illustration of Gradient Descent – XOR



Source: Pattern Classification (2nd Edition); Richard O. Duda, Peter E. Hart, David G. Stork

Comments on Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.

Comments on Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- In practice, does converge to low error for many large networks on real data.

Comments on Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- In practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.

Comments on Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- In practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (random restarts).
  - Take results of trial with lowest training set error.
  - Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).

There are many more issues concerning learning efficiency (data normalization, selection of activation functions, weight initialization, training speed, efficiency of the gradient descent itself etc.) – see PV021.

Hidden Neurons Representations

Trained hidden neurons can be seen as newly constructed features.
E.g., in a two layer network used for classification, the hidden layer transforms the input so that important features become explicit (and hence the result may become linearly separable).

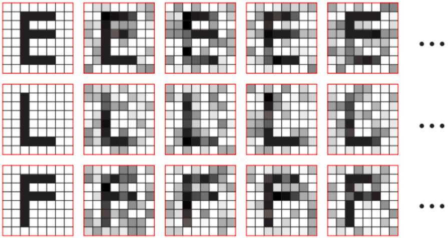
Hidden Neurons Representations

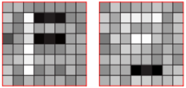
Trained hidden neurons can be seen as newly constructed features.

E.g., in a two layer network used for classification, the hidden layer transforms the input so that important features become explicit (and hence the result may become linearly separable).

Consider a two-layer MLP, 64-2-3 for classification of letters (three output neurons, each corresponds to one of the letters).

sample training patterns

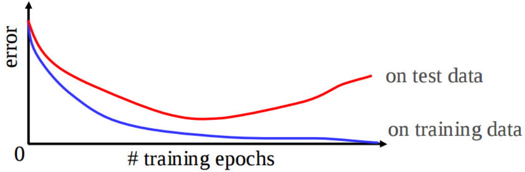




learned input-to-hidden weights

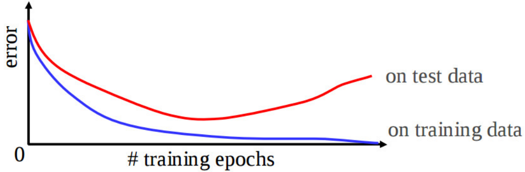
Overfitting

- Due to their expressive power, neural networks are quite sensitive to overfitting.



Overfitting

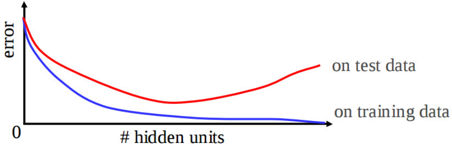
- Due to their expressive power, neural networks are quite sensitive to overfitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase the validation error.

Overfitting – The Number of Hidden Neurons

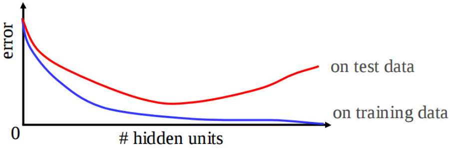
- Too few hidden neurons prevent the network from adequately fitting the data.
- Too many hidden units can result in overfitting.
  (There are advanced methods that prevent overfitting even for rich models, such as regularization, where the error function penalizes overfitting – see PV021.)



Overfitting – The Number of Hidden Neurons

- Too few hidden neurons prevent the network from adequately fitting the data.
- Too many hidden units can result in overfitting.

  (There are advanced methods that prevent overfitting even for rich models, such as regularization, where the error function penalizes overfitting – see PV021.)



- Use cross-validation to empirically determine an optimal number of hidden units.

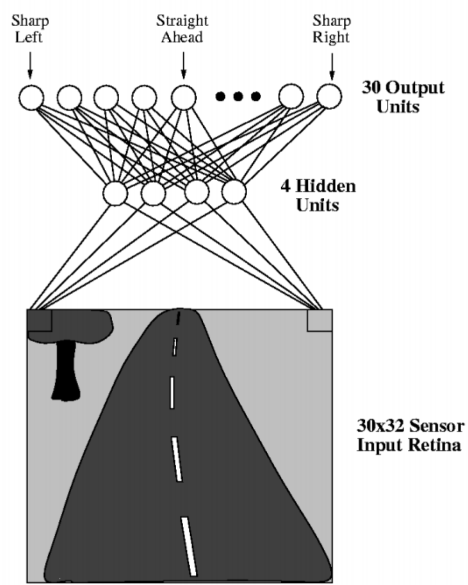
  There are methods that automatically construct the structure of the network based on data, they are not much used though.

Applications

- Text to Speech and vice versa
- Fraud detection
- finance & business predictions
- Game playing (backgammon is a classical example, AlphaGo is the modern one)
- Image recognition

  This is the main area in which the current state-of-the-art deep networks excel.
- (artificial brain and intelligence)
- ...

ALVINN


Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina

ALVINN

- Two layer MLP, $960 - 4 - 30$ (sometimes $960 - 5 - 30$)

ALVINN

- Two layer MLP, $960 - 4 - 30$ (sometimes $960 - 5 - 30$)
- Inputs correspond to pixels.
- Sigmoidal activation function (logistic sigmoid).

ALVINN

- Two layer MLP, $960 - 4 - 30$ (sometimes $960 - 5 - 30$)
- Inputs correspond to pixels.
- Sigmoidal activation function (logistic sigmoid).
- Direction corresponds to the center of gravity.

  I.e., output neurons are considered as points of mass evenly distributed along a line. Weight of each neuron corresponds to its value.

ALVINN – Training

Trained while driving.

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second
- Training examples $(\vec{x}_k, \vec{d}_k)$ where

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second
- Training examples $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second
- Training examples $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road
  - $\vec{d}_k \approx$ corresponding direction of the steering wheel set by the driver

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second
- Training examples $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road
  - $\vec{d}_k \approx$ corresponding direction of the steering wheel set by the driver
- the values $\vec{d}_k$ computed using Gaussian distribution:

$$d_{ki} = e^{-D_i^2/10}$$

where D_i is the distance between the i-th output from the one that corresponds to the real direction of the steering wheel.

(This is better than the binary output because similar road directions induce similar reaction of the driver.)

Selection of Training Examples

Naive approach: just take images from the camera.

Selection of Training Examples

Naive approach: just take images from the camera.

Problems:

Selection of Training Examples

Naive approach: just take images from the camera.

Problems:

- A too good driver never teaches the network how to solve deviations from the right track. Couple of harsh solutions:

Selection of Training Examples

Naive approach: just take images from the camera.

Problems:

- A too good driver never teaches the network how to solve deviations from the right track. Couple of harsh solutions:
    - turn the learning off for a moment, deviate from the right track, then turn on the learning and let the network learn how to solve the situation.

Selection of Training Examples

Naive approach: just take images from the camera.

Problems:

- A too good driver never teaches the network how to solve deviations from the right track. Couple of harsh solutions:
  - turn the learning off for a moment, deviate from the right track, then turn on the learning and let the network learn how to solve the situation.
  - let the driver go crazy! (a bit dangerous, expensive, unreliable)

Selection of Training Examples

Naive approach: just take images from the camera.

Problems:

- A too good driver never teaches the network how to solve deviations from the right track. Couple of harsh solutions:
  - turn the learning off for a moment, deviate from the right track, then turn on the learning and let the network learn how to solve the situation.
  - let the driver go crazy! (a bit dangerous, expensive, unreliable)
- Images are very similar (the network basically sees the road from the right lane), may be overtrained.

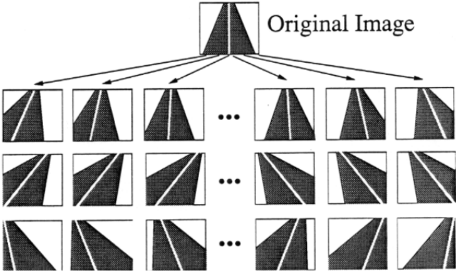
Selection of Training Examples

Problem with too good driver were solved as follows:

Selection of Training Examples

Problem with too good driver were solved as follows:

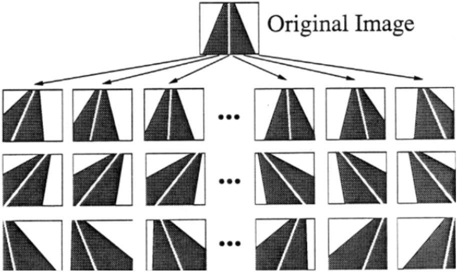
- every image of the road has been has been transformed to 15 slightly different copies



Selection of Training Examples

Problem with too good driver were solved as follows:

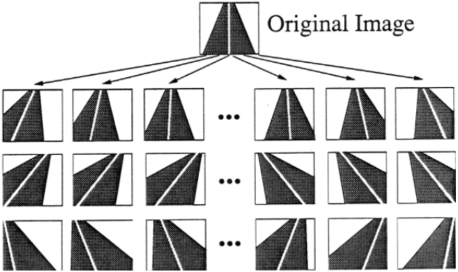
- every image of the road has been has been transformed to 15 slightly different copies



Selection of Training Examples

Problem with too good driver were solved as follows:

- every image of the road has been has been transformed to 15 slightly different copies

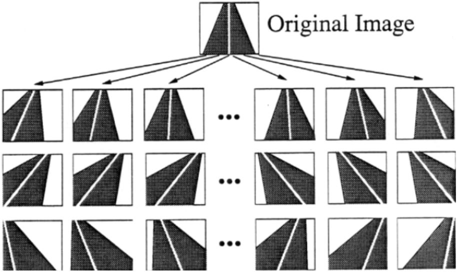


Repetitiveness of images was solved as follows:

Selection of Training Examples

Problem with too good driver were solved as follows:

- every image of the road has been has been transformed to 15 slightly different copies



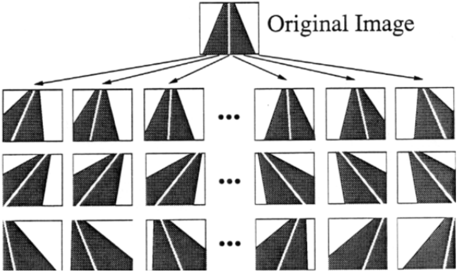
Repetitiveness of images was solved as follows:

- the system has a buffer of 200 images (including the 15 copies of the current one), in every round trains on these images

Selection of Training Examples

Problem with too good driver were solved as follows:

- every image of the road has been has been transformed to 15 slightly different copies



Repetitiveness of images was solved as follows:

- the system has a buffer of 200 images (including the 15 copies of the current one), in every round trains on these images
- afterwards, a new image is captured, 15 copies made, and these new 15 substitute 15 selected from the buffer (10 with the smallest training error, 5 randomly)

ALVINN – Training

- standard backpropagation

ALVINN – Training

- standard backpropagation
- constant speed of learning (possibly different for each neuron – see PV021)

ALVINN – Training

- standard backpropagation
- constant speed of learning (possibly different for each neuron – see PV021)
- some other optimizations (see PV021)

ALVINN – Training

- standard backpropagation
- constant speed of learning (possibly different for each neuron – see PV021)
- some other optimizations (see PV021)

Výsledek:

- Training took 5 minutes, the speed was 4 miles per hour
  (The speed was limited by the hydraulic controller of the steering wheel not the learning algorithm.)

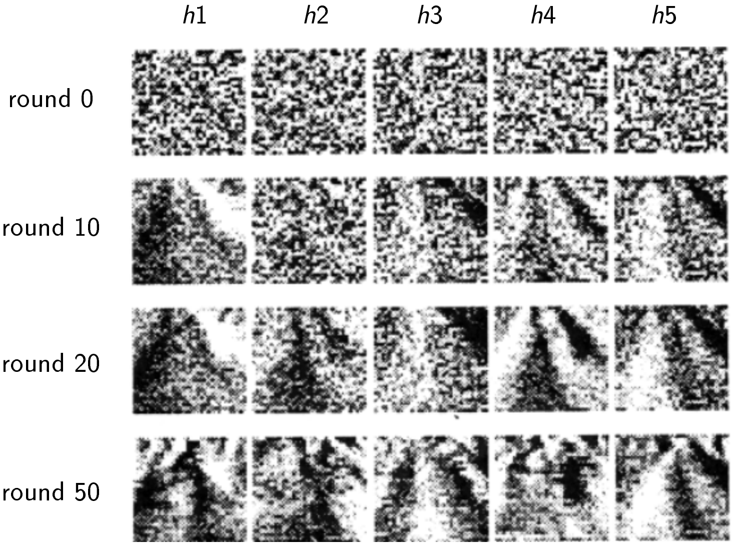
ALVINN – Training

- standard backpropagation
- constant speed of learning (possibly different for each neuron – see PV021)
- some other optimizations (see PV021)

Výsledek:

- Training took 5 minutes, the speed was 4 miles per hour
  (The speed was limited by the hydraulic controller of the steering wheel not the learning algorithm.)
- ALVINN was able to go through roads it never "seen" and in different weather

ALVINN – Weight Learning



Here $h1, \ldots, h5$ are values of hidden neurons.

Extensions and Directions (PV021)

- Other types of learning inspired by neuroscience – Hebbian learning

Extensions and Directions (PV021)

- Other types of learning inspired by neuroscience – Hebbian learning
- More biologically plausible models of neural networks – spiking neurons

  This goes into the direction of HUGE area of (computational) neuroscience, only very lightly touched in PV021.

Extensions and Directions (PV021)

- Other types of learning inspired by neuroscience – Hebbian learning
- More biologically plausible models of neural networks – spiking neurons

  This goes into the direction of HUGE area of (computational) neuroscience, only very lightly touched in PV021.
- Unsupervised learning – Self-Organizing Maps

Extensions and Directions (PV021)

- Other types of learning inspired by neuroscience – Hebbian learning
- More biologically plausible models of neural networks – spiking neurons

  This goes into the direction of HUGE area of (computational) neuroscience, only very lightly touched in PV021.
- Unsupervised learning – Self-Organizing Maps
- Reinforcement learning
  - learning to make decisions, or play games, sequentially
  - neural networks have been used – temporal difference learning

Deep Learning

- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.

Deep Learning

- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.
- Later, it has been shown (experimentally) that deep networks (with many layers) have better represenational properties.

Deep Learning

- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.
- Later, it has been shown (experimentally) that deep networks (with many layers) have better represenational properties.
- ... but how to train them? The backpropagation suffers from so-called vanishing gradient, intuitively, updates of weights in lower layers are *very* slow.

Deep Learning

- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.
- Later, it has been shown (experimentally) that deep networks (with many layers) have better represenational properties.
- ... but how to train them? The backpropagation suffers from so-called vanishing gradient, intuitively, updates of weights in lower layers are *very* slow.
- In 2006 a solution was found by Hinton et al:
  - Use *unsupervised* methods to initialize the weights so that they capture important features in data.
    More precisely: The lowest hidden layer learns patterns in data, second lowest learns patterns in data transformed through the first layer, and so on.

Deep Learning

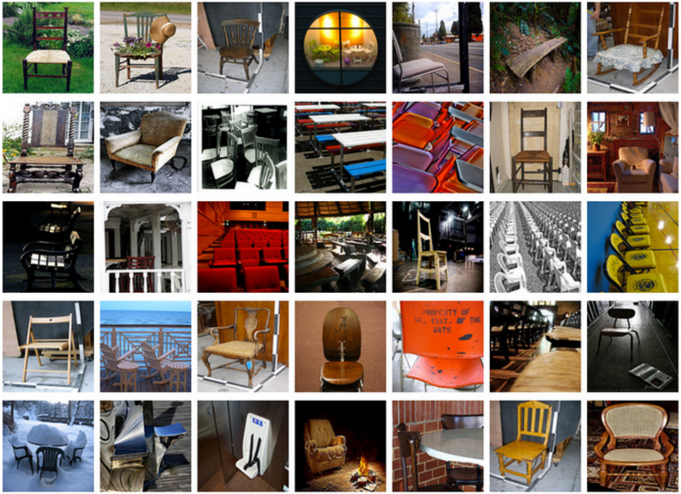
- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.
- Later, it has been shown (experimentally) that deep networks (with many layers) have better represenational properties.
- ... but how to train them? The backpropagation suffers from so-called vanishing gradient, intuitively, updates of weights in lower layers are *very* slow.
- In 2006 a solution was found by Hinton et al:
  - Use *unsupervised* methods to initialize the weights so that they capture important features in data.

    More precisely: The lowest hidden layer learns patterns in data, second lowest learns patterns in data transformed through the first layer, and so on.
  - Then use a supervised learning algorithm to only *fine tune* the weights to the desired input-output behavior.

A rather heavy machinery is needed to develop this, but you will be rewarded by insight into a *very* modern and expensive technology.

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

ImageNet database (16,000,000 color images, 20,000 categories)



ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

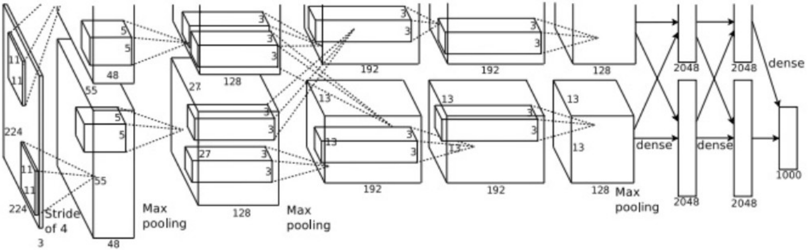
Competition in classification over a subset of images from ImageNet.

In 2012: Training se 1,200,000 images, 1000 categories. Validation set 50,000, Test set 150,000.

Many images contain several objects $\rightarrow$ typical rule is top-5 highest probability assigned by the net.

KSH síť

ImageNet classification with deep convolutional neural networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton (2012).



Trained on two GPUs (NVIDIA GeForce GTX 580)

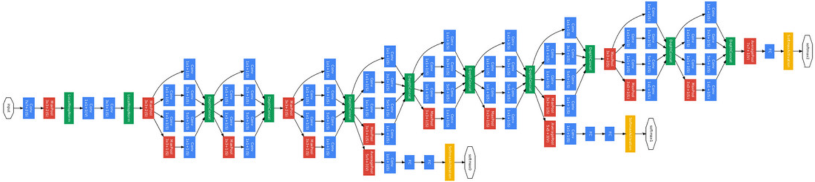
Results:

- Accuracy 84.7% in top-5 (second best alg. at the time: 73.8%)
- 63.3% in "perfect" classification (top-1)

ILSVRC 2014

The same set of images as in 2012, top-5 criterium.

GoogLeNet: deep convolutional net, 22 layers



Results:

- 93.33% in top-5

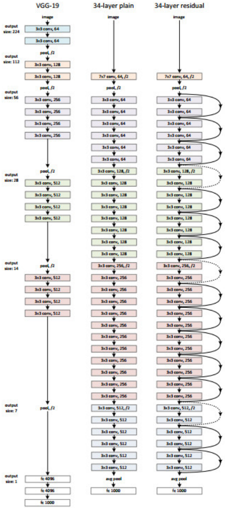
Superhuman power?

Superhuman GoogLeNet?!

Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought we could recruit paid undergrads. Then I organized a labeling party of intense labeling effort only among the (expert labelers) in our lab. Then I developed a modified interface that used GoogLeNet predictions to prune the number of categories from 1000 to only about 100. It was still too hard - people kept missing categories and getting up to ranges of 13-15% error rates. In the end I realized that to get anywhere competitively close to GoogLeNet, it was most efficient if I sat down and went through the painfully long training process and the subsequent careful annotation process myself... The labeling happened at a rate of about 1 per minute, but this decreased over time... Some images are easily recognized, while some images (such as those of fine-grained breeds of dogs, birds, or monkeys) can require multiple minutes of concentrated effort. I became very good at identifying breeds of dogs... Based on the sample of images I worked on, the GoogLeNet classification error turned out to be 6.8%... My own error in the end turned out to be 5.1%, approximately 1.7% better.

ILSVRC 2015

- Microsoft network ResNet: 152 layers, complex architecture
- Trained on 8 GPUs
- 96.43% **accuracy** in top-5



ILSVRC

ilsvrc.png

Deeper Insight into the Logistic Sigmoid

Consider a perceptron (that is a linear classifier):

$$\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

and $y = sgn(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$

Deeper Insight into the Logistic Sigmoid

Consider a perceptron (that is a linear classifier):

$$\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

and $y = sgn(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$

Recall, that the *signed distance* from the decision boundary determined by $\xi = 0$ is (here $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{w} = (w_1, \ldots, w_n)$)

$$\frac{w_0 + \sum_{i=1}^{n} w_i \cdot x_i}{\sqrt{\sum_{i=1}^{n} w_i^2}} = \frac{\xi}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

This value is positive for $\vec{x}$ on the side of $\vec{w}$ and negative on the opposite.

Deeper Insight into the Logistic Sigmoid

Consider a perceptron (that is a linear classifier):

$$\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

and $y = sgn(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$

Recall, that the *signed distance* from the decision boundary determined by $\xi = 0$ is (here $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{w} = (w_1, \ldots, w_n)$)

$$\frac{w_0 + \sum_{i=1}^{n} w_i \cdot x_i}{\sqrt{\sum_{i=1}^{n} w_i^2}} = \frac{\xi}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

This value is positive for $\vec{x}$ on the side of $\vec{w}$ and negative on the opposite.

For simplicity, assume that $\sqrt{\sum_{i=1}^{n} w_i^2} = 1$, and thus that the potential ξ is *equal to the signed distance of $\vec{x}$ from the boundary*.

Deeper Insight into the Logistic Sigmoid

Assume that training examples $(\vec{x}, c(\vec{x}))$ are randomly generated.

Deeper Insight into the Logistic Sigmoid

Assume that training examples $(\vec{x}, c(\vec{x}))$ are randomly generated.

Denote:

- ξ^1 mean signed distance from the boundary of points classified 1.
- ξ^0 mean signed distance from the boundary of points classified 0.

Deeper Insight into the Logistic Sigmoid

Assume that training examples $(\vec{x}, c(\vec{x}))$ are randomly generated.

Denote:

- ξ^1 mean signed distance from the boundary of points classified 1.
- ξ^0 mean signed distance from the boundary of points classified 0.

It is not unreasonable to assume that

- conditioned on $c = 1$, the signed distance ξ is normally distributed with the mean ξ^1 and variance (for simplicity) 1,
- conditioned on $c = 0$, the signed distance ξ is normally distributed with the mean ξ^0 and variance (for simplicity) 1.

(Notice that ξ may be negative, which means that such point is on the wrong side of the boundary (the same for $\xi > 0$).)

Deeper Insight into the Logistic Sigmoid

Assume that training examples $(\vec{x}, c(\vec{x}))$ are randomly generated.

Denote:

- ξ^1 mean signed distance from the boundary of points classified 1.
- ξ^0 mean signed distance from the boundary of points classified 0.

It is not unreasonable to assume that

- conditioned on $c = 1$, the signed distance ξ is normally distributed with the mean ξ^1 and variance (for simplicity) 1,
- conditioned on $c = 0$, the signed distance ξ is normally distributed with the mean ξ^0 and variance (for simplicity) 1.

(Notice that ξ may be negative, which means that such point is on the wrong side of the boundary (the same for $\xi > 0$).)

Now, can we decide what is the probability of $c = 1$ given a distance?

Deeper Insight into the Logistic Sigmoid

For simplicity, assume that $\xi^1 = -\xi^0 = 1/2$.

$$\begin{aligned} P(1 \mid \xi) &= \frac{p(\xi \mid 1)P(1)}{p(\xi \mid 1)P(1) + p(\xi \mid 0)P(0)} \\ &= \frac{LR}{LR + 1/clr} \end{aligned}$$

where

$$LR = \frac{p(\xi \mid 1)}{p(\xi \mid 0)} = \frac{\exp(-(\xi - 1/2)^2/2)}{\exp(-(\xi + 1/2)^2/2)} = \exp(\xi)$$

and

$$clr = \frac{P(1)}{P(0)} \text{ which we assume (for simplicity) } = 1$$

Deeper Insight into the Logistic Sigmoid

For simplicity, assume that $\xi^1 = -\xi^0 = 1/2$.

$$P(1 \mid \xi) = \frac{p(\xi \mid 1)P(1)}{p(\xi \mid 1)P(1) + p(\xi \mid 0)P(0)} = \frac{LR}{LR + 1/clr}$$

where

$$LR = \frac{p(\xi \mid 1)}{p(\xi \mid 0)} = \frac{\exp(-(\xi - 1/2)^2/2)}{\exp(-(\xi + 1/2)^2/2)} = \exp(\xi)$$

and

$$clr = \frac{P(1)}{P(0)} \text{ which we assume (for simplicity) } = 1$$

So

$$P(1 \mid \xi) = \frac{\exp(\xi)}{\exp(\xi) + 1} = \frac{1}{1 + e^{-\xi}}$$

Thus the logistic sigmoid applied to $\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$ gives *the probability* of $c = 1$ given the input!

Deeper Insight into the Logistic Sigmoid

So if we use the logistic sigmoid as an activation function,

and turn the neuron into a classifier as follows:

$$\text{classify a given input } \vec{x} \text{ as 1 iff } y \geq 1/2$$

Deeper Insight into the Logistic Sigmoid

So if we use the logistic sigmoid as an activation function,

and turn the neuron into a classifier as follows:

$$\text{classify a given input } \vec{x} \text{ as 1 iff } y \geq 1/2$$

Then the neuron basically works as the *Bayes classifier!*

Deeper Insight into the Logistic Sigmoid

So if we use the logistic sigmoid as an activation function,

and turn the neuron into a classifier as follows:

$$\text{classify a given input } \vec{x} \text{ as 1 iff } y \geq 1/2$$

Then the neuron basically works as the *Bayes classifier!*

This is the basis of logistic regression.

Deeper Insight into the Logistic Sigmoid

So if we use the logistic sigmoid as an activation function,

and turn the neuron into a classifier as follows:

classify a given input $\vec{x}$ as 1 iff $y \geq 1/2$

Then the neuron basically works as the *Bayes classifier!*

This is the basis of logistic regression.

Given training data, we may compute the weights $\vec{w}$ that maximize the likelihood of the training data (w.r.t. the probabilities returned by the neuron).

An extremely interesting observation is that such $\vec{w}$ maximizing the likelihood coincides with the minimum of least squares for the corresponding linear function (that is the same neuron but with identity as the activation function).