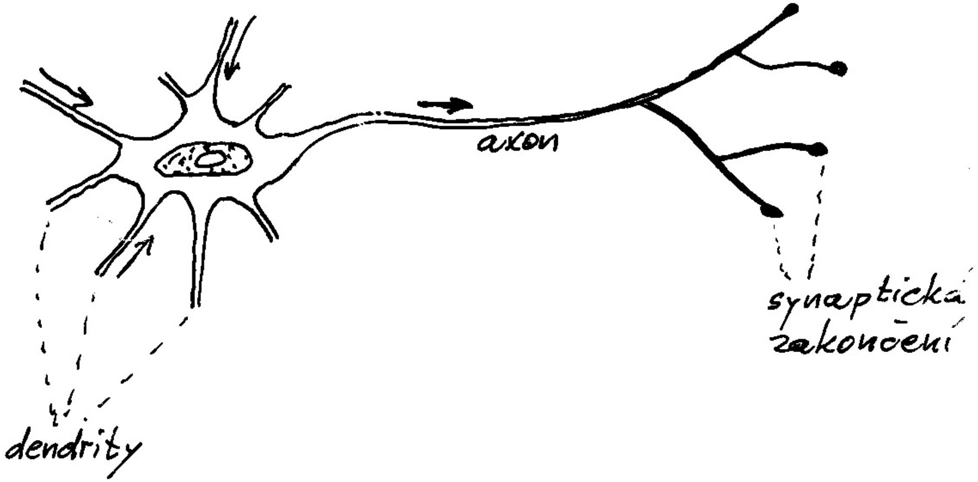
NEURON

Umělý perceptron je inspirován svým biologickým protějškem – neuronem, který je jedním z prvků nervové soustavy a tvoří základ mozku.

Neexistuje „typický neuron", avšak následující obrázek schematicky znázorňuje vlastnosti sdílené mnoha neurony:



Dendrity tvoří vstupy neuronu, axon je výstupem. Synaptická zakončení tvoří spoj k jiným neuronům či efektorům.

Aktivita receptorů modifikuje membránový potenciál dendritů a těla buňky. Elektrický impuls membránového potenciálu se šíří podél axonu a aktivuje synaptická zakončení, která následně modifikují membránový potenciál dalších neuronů či svalových vláken (efektorů).

Potenciálový rozdíl se šíří skrze membránu obalující nervovou buňku, a se vzdáleností zaniká. Pokud ovšem napětí převýší jistou hodnotu zvanou „práh", pak se elektrický signál může dále šířit bez poklesu své velikosti.

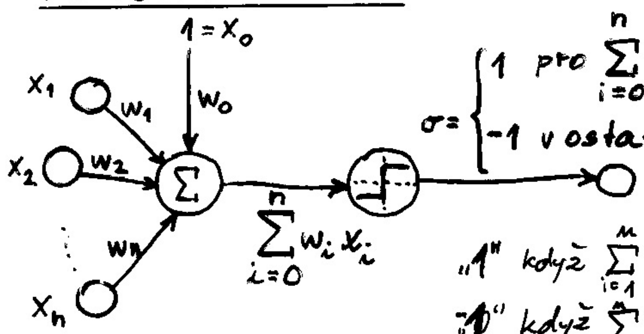
Většina synapsí má chemickou povahu. Elektrický signál, který dosáhne synapsi, způsobí uvolnění molekul zvaných transmitéry (přenašeče), které mohou mít buď excitační nebo inhibiční účinek. Excitační účinek „posune" neuron směrem k jeho prahu, zatímco inhibiční naopak.

Koncept umělého neuronu pochází z r. 1943 (autoři McCuloch a Pitts). Tehdejší umělý neuron byl popsán jako binární diskrétní (časově) element. Jeho výstup (log. „0" nebo „1") je ovlivněn hodnotou součtu vstupů – pokud součet převýší určitou prahovou hodnotu, na výstupu umělého neuronu se objeví „1", jinak „0".

V r. 1957 rozšířil Rosenblatt model umělého neuronu na tzv. perceptron zejména tím, že zavedl tzv. váhy spojů mezi neurony. V principu každá váha určuje sílu vztahu mezi dvojicí neuronů. Změna hodnoty váhy umožňuje se neuronu adaptovat na změněné podmínky a tím také se učit.

Spojením neuronů do sítě vznikne struktura schopná učení. Existuje mnoho typů sítí a učících metod.

PERCEPTRONY

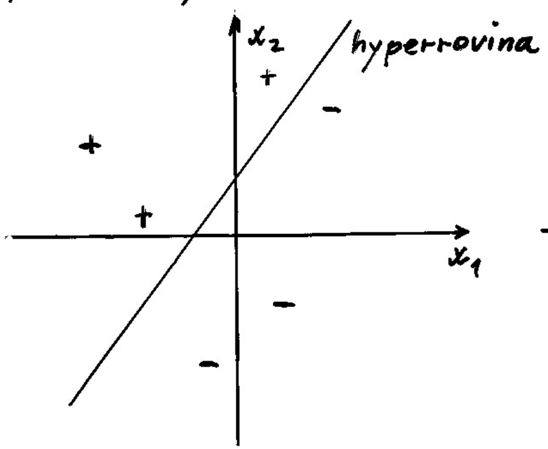


$$\sigma = \begin{cases} 1 & \text{pro } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{v ostatních případech} \end{cases}$$

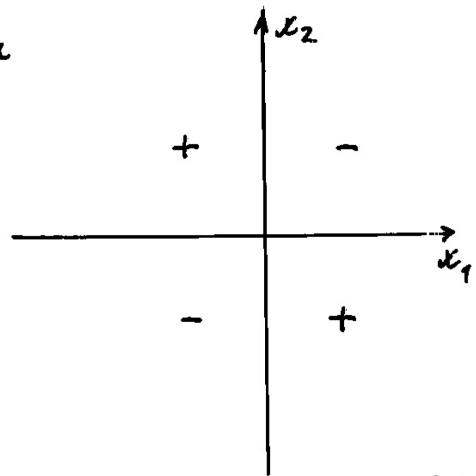
„1" když $\sum_{i=1}^{n} x_i w_i > t$ ... práh

„-1" když $\sum_{i=1}^{n} x_i w_i \leq t$ $(w_0 = -t)$

Perceptron representuje rozhodovací hyperrovinu v n-rozměrném instančním prostoru (instance = bod). Výstupem perceptronu je 1 pro instance nacházející se na jedné straně hyperroviny a -1 pro instance na druhé straně:

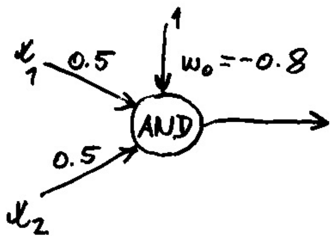


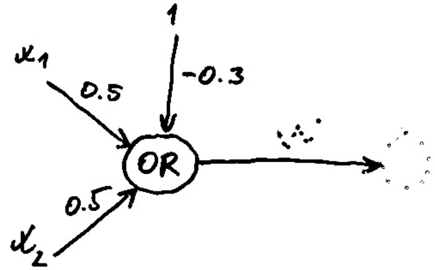
a) soubor trénovacích instancí a hyperrovina perceptronu korektně klasifikující instance



b) soubor trénovacích instancí, které nejsou lineárně separabilní (XOR)

Jednoduchý perceptron lze využít pro realizaci mnoha booleovských funkcí. Je-li TRUE = 1 a FALSE = -1, pak např. nastavení vah $w_0 = -0.8$, $w_1 = w_2 = 0.5$ realizuje AND. Změnou $w_0 = -0.3$ dosáhneme OR.





Perceptrony AND a OR jsou speciálními případy tzv. m-z-n funkcí (nejméně m z n vstupů musí být TRUE aby výstup byl TRUE). AND odpovídá $m = n$, OR odpovídá $m = 1$.

Perceptrony (mohou) representovat primitivní booleovské funkce AND, OR, NAND ($\neg$AND), NOR ($\neg$OR).

Některé funkce však representovat NELZE, např. XOR (exklusivní OR: výstup je 1 pouze když $x_1 \neq x_2$).

Schopnost representace AND, OR, NAND, NOR je důležitá proto, že KAŽDOU booleovskou funkci lze realizovat nějakou sítí propojených základních jednotek (stačí dvouvrstvá síť). Lze použít např. DNF (disjunktivní normální formu) jako disjunkci (OR) konjunkcí (AND) vstupů a jejich negací (negace AND ... změna znaménka w_i).

Umělý perceptron: representace boolských funkcí

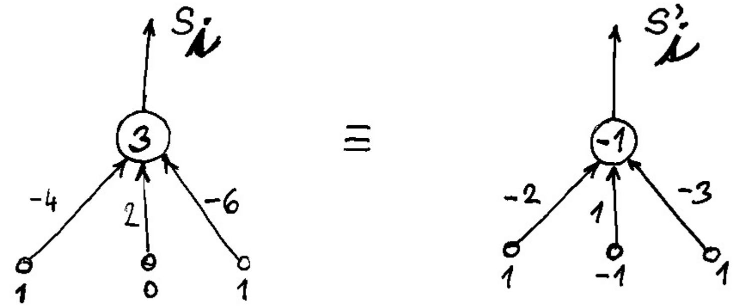
+1 ... true (logická "1")
−1 ... false (−"− "0")
(0 ... neznámo)

} možný způsob representace boolských hodnot

+1 ... T
0 ... F

} jiný možný (standardní) způsob

Obě formy jsou ekvivalentní:



$w'_{ij} = \frac{1}{2} w_{ij}$ (verze bez posuvu)

$w'_{i0} = w_{i0} + \sum_{j=1}^{n} w'_{ij} = w_{i0} + \frac{1}{2} \sum_{j=1}^{n} w_{ij}$

} přepočet vah (a posuvu)

$(-4 \cdot 1) + (2 \cdot 0) + (-6 \cdot 1) = -10$ $\qquad$ $(-2 \cdot 1) + (1 \cdot (-1)) + (-3 \cdot 1) = -6$

$-10 + 3 = \underline{\underline{-7}}$ $\qquad$ $-6 - 1 = \underline{\underline{-7}}$

$$\underline{\underline{S_i = S'_i}}$$

Pozn.: pokaždé, když změníme v systému {0, 1} vstup z F na T (0 → 1), zvýšíme S_i o w_{ij}. V systému {−1, 1} je S'_i zvýšeno o $2w_{ij}$. Z požadavku na $S_i = S'_i$ plyne výše uvedené.

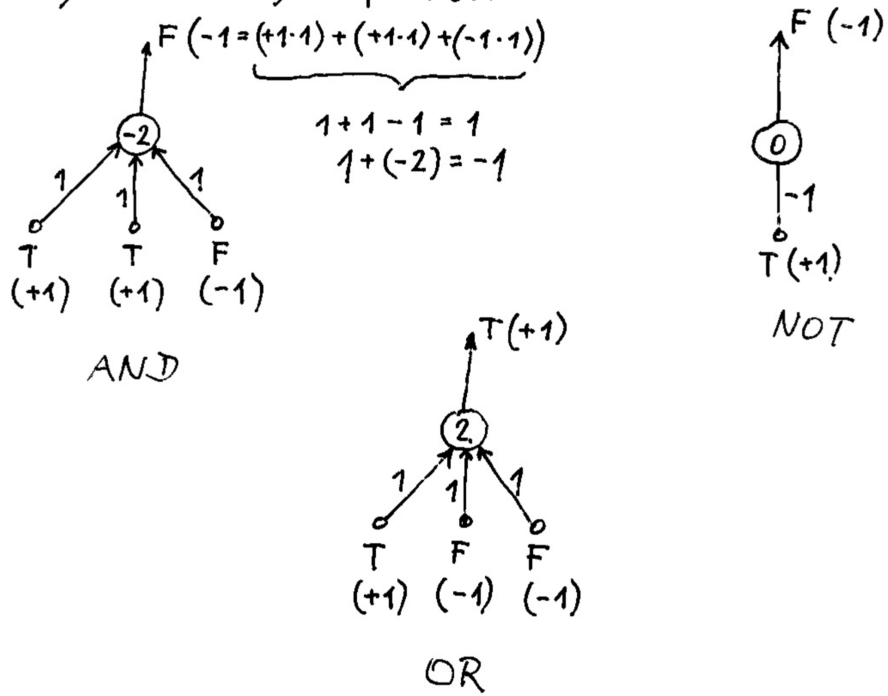
Obvykle se dává přednost systému {−1, +1}, protože v tomto případě probíhá trénování (sítě) perceptronů rychleji:

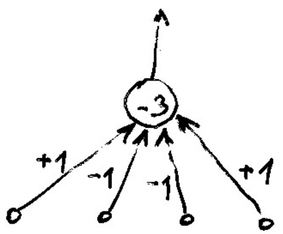
Je-li vstup $u_j = 0$ v systému {0, 1}, pak nedojde k žádné opravě váhy (násobení nulou). V systému {−1, +1} je však $u_j = -1$, takže w_{ij} se změní. To obvykle vede k rychlejší konvergenci.

Je-li zapotřebí v systému {0, 1} pracovat s hodnotou "neznámo", lze použít 1/2.

Modely s jednou jednotkou

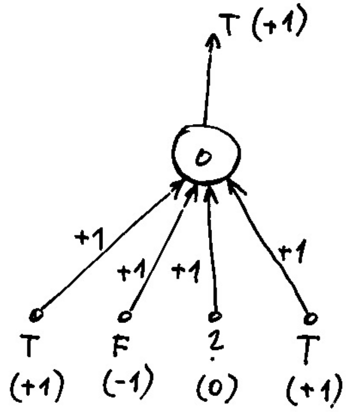
Tyto lineární modely mohou počítat většinu běžných boolských funkcí:





SELEKTOR

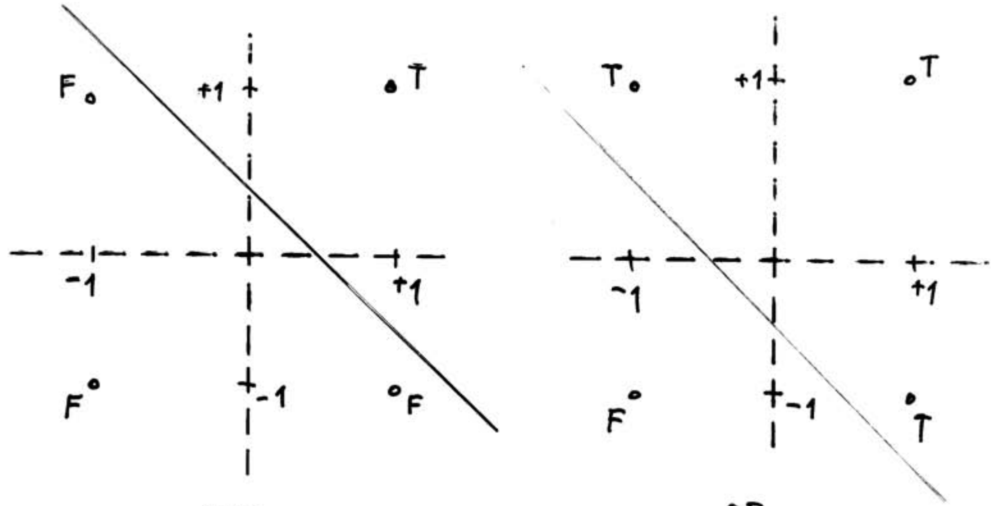
Na výstupu je +1 právě pro jedinou vstupní kombinaci (zde +1, −1, −1, +1).



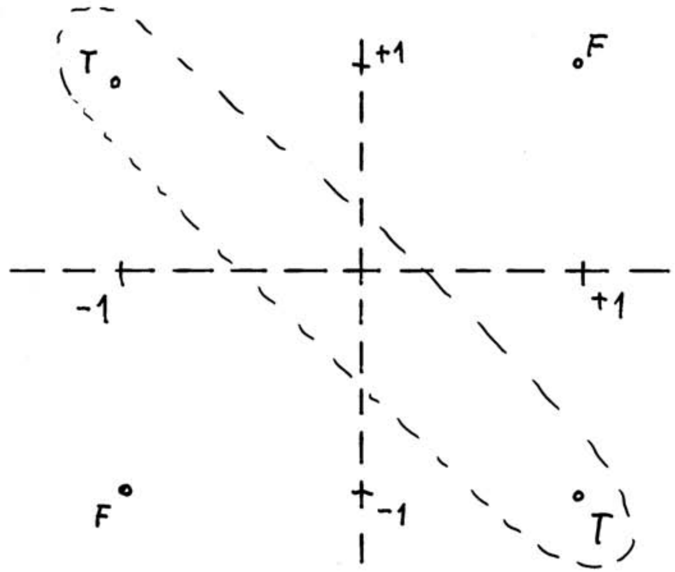
VĚTŠINA

Na výstupu je T (+1) tehdy, když většina vstupů je T (více T než F), F(−1) když je více F než T, a ? (neznámo, 0) pokud je stejně T i F.

<u>Důležité:</u> Lineární model s 1 jednotkou neumí počítat <u>všechny</u> boolské funkce. Umí počítat pouze tzv. <u>lineárně separabilní</u> funkce.



AND → lineárně separovatelné ← OR



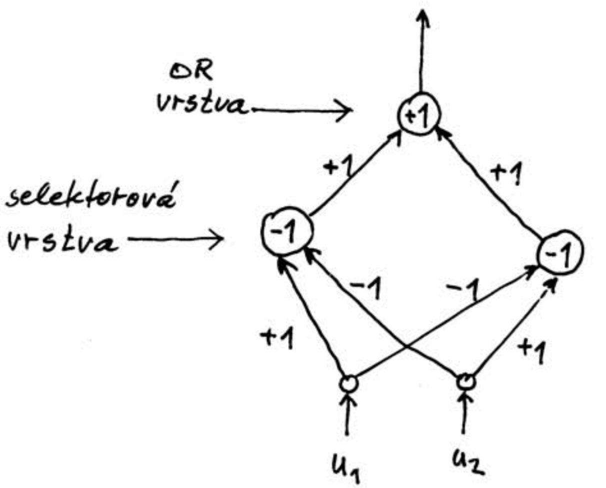
XOR → nelze zkonstruovat přímku oddělující F a T
(XOR není lineárně separovatelná fce)

XOR je z hlediska počtu vstupů nejjednodušší lineárně neseparovatelná funkce.

Tzv. paritní funkce (zobecněná XOR) je T pokud počet T-vstupů je lichý, jinak je F.

Representace libovolné boolské funkce

Vícevrstevný perceptron umí representovat libovolnou boolskou funkci f:



Perceptronová síť pro výpočet hodnoty XOR pomocí selektorů.
Výstup je T pro vstupy {+1, -1} a {-1, +1}, jinak je F.

Věta: Jakákoliv boolská funkce s konečným počtem vstupů je representovatelná pomocí vícevrstvého lineárního perceptronu.

Důkaz: Ve struktuře OR-selektory způsobí každý vstup, že právě 1 nebo žádná selektorová jednotka bude mít na výstupu +1, v závislosti na tom, zda požadovaný výstup je T nebo F. To zaručuje, že výstup jednotky OR bude odpovídat požadovanému výstupu.

Důsledek: Je-li dán soubor nekontradiktorických trénovacích příkladů mající boolské atributy a klasifikace, pak vždy existuje vícevrstevný lineární perceptron produkující korektní výstup pro všechny trénovací příklady.

Praktická limitace: existuje mnoho funkcí s výstupem T přibližně ~~1/2 OT~~ pro 1/2 vstupů (možných vstupních vzorů), takže selektorová konstrukce by vyžadovala cca 2^{p-1} jednotek pro representaci těchto funkcí (p je počet vstupů). Např. pro pouhých 10 vstupů to je $2^{10-1} = 2^9 = 512$. V praxi lze často očekávat $p \sim 10^n$, $n \geq 2$. Dále, např. pro $p = 100$ a počet trénovacích vzorků = 1000 sice sestavíme vhodnou síť s $\leq 10^3$ selektory, ale tato síť poskytne na výstupu T pouze když vstup bude duplikovat jeden z T-trénovacích příkladů → malá robustnost.

TRÉNOVACÍ PRAVIDLO PRO PERCEPTRON

Problém učení lze definovat jako problém stanovení vektoru vah tak, aby perceptron dával na výstupu korektně hodnoty ±1 pro každou z daných trénovacích instancí.
Je známo několik algoritmů, zde uvedeme dva: perceptronové pravidlo a delta pravidlo.

Jednou z možností, jak získat přijatelný vektor vah $\vec{w}$, je začít s váhami s náhodně přiřazenými hodnotami a iterativně použít perceptron na každý trénovací příklad – bude-li klasifikován chybně, váhy je nutno modifikovat.
Tento proces se opakuje tak dlouho, dokud nejsou všechny trénovací příklady klasifikovány správně. Modifikace vah probíhá podle perceptronového trénovacího pravidla:

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta (t - \sigma) x_i$$

$$\eta > 0$$

t je očekávaná hodnota
σ je skutečná hodnota na výstupu
η je tzv. učicí konstanta

Koeficient η moderuje stupeň velikosti změn vah v každém kroku. Obvykle se η nastaví na nějakou malou hodnotu, např. $\eta = 0.1$. Se vzrůstajícím počtem iteračních kroků se většinou nechává vliv η postupně vymizet.

Proč uvedené perceptronové pravidlo konverguje směrem k vytvoření váhového vektoru?

Předpokl., že ve speciálním případě je daná instance klasifikována korektně: $(t - \sigma) = 0 \Rightarrow \Delta w_i = 0$, tzn. váhy nejsou modifikovány.
Objeví-li se místo +1 na výstupu −1, je nutno váhy změnit (pro $x_i > 0$ se zvýší w_i a tím se perceptron přiblíží korektní klasifikaci daného příkladu).
w_i se zvýší, neboť $(t - \sigma) > 0$, $\eta > 0$, $x_i > 0$, např. $x_i = 0.8$, $\eta = 0.1$, $t = 1$, $\sigma = -1$:

$$\Delta w_i = \eta (t - \sigma) x_i = 0.1 (1 - (-1)) \cdot 0.8 = 0.16$$

Na druhé straně bude-li $t = -1$ a $\sigma = +1$, pak váhy příslušející pozitivním x_i budou sníženy (nikoliv zvýšeny).

V r. 1969 dokázali Minsky a Papert, že za předpokladu lineární separability trénovacích příkladů a dostatečně malého η perceptronové trénovací pravidlo konverguje. Nejsou-li data lineárně separabilní, nelze konvergenci zaručit.

Pravidlo delta a gradientní sestup

Perceptronové pravidlo může selhat v případě nesplnění podmínky lineární separability (oddělitelnosti) dat. Pravidlo delta umožňuje tuto obtíž překonat: zaručí konvergenci k <u>nejlépe se hodící aproximaci</u> cílového konceptu.

<u>Podstata</u>: pro prohledávání prostoru možných vah se použije tzv. gradientní sestup k nalezení vah, které umožní co nejlepší aproximaci.

<u>Pozn.</u>: tzv. BACK-PROPAGATION algoritmus pro trénování umělých neuronových sítí vzniklých propojením perceptronů je založen na pravidlu delta.

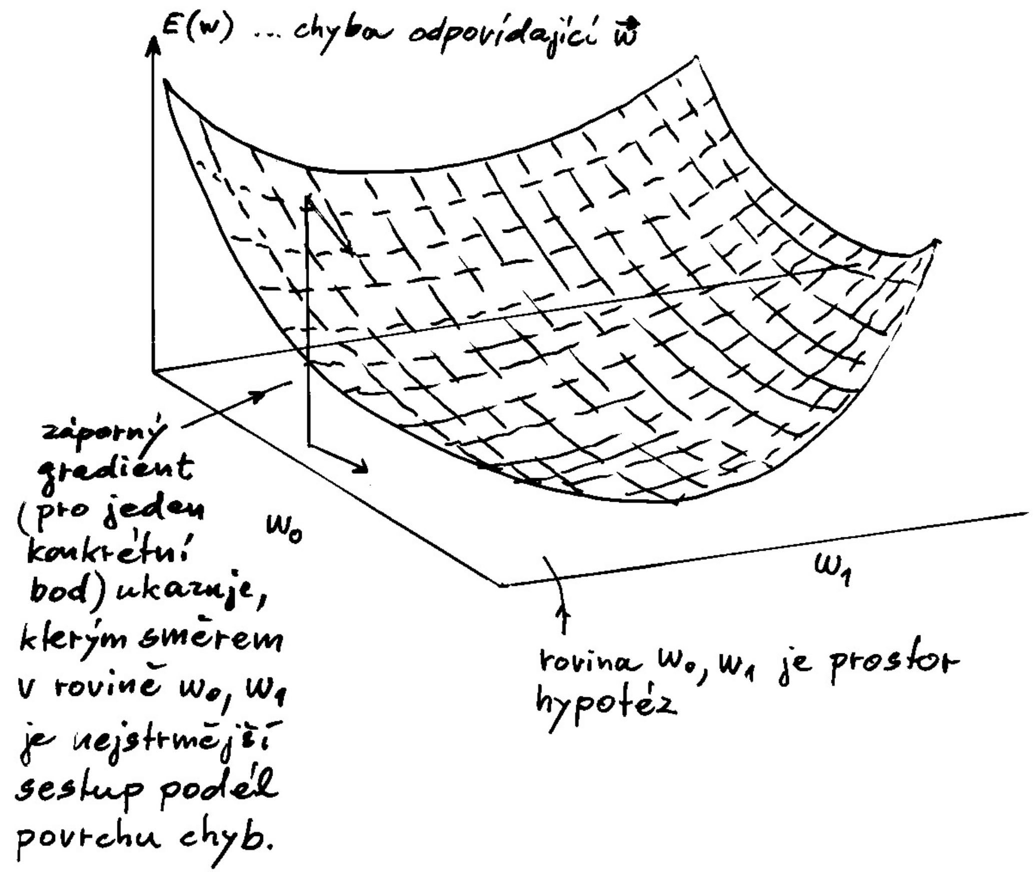
Předpokládejme lineární jednotku (perceptron bez prahu), jejíž výstup je dán vztahem

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Je nutno stanovit nějakou míru trénovací chyby vzhledem k trénovacím příkladům. Nejčastěji se používá výhodný uztah

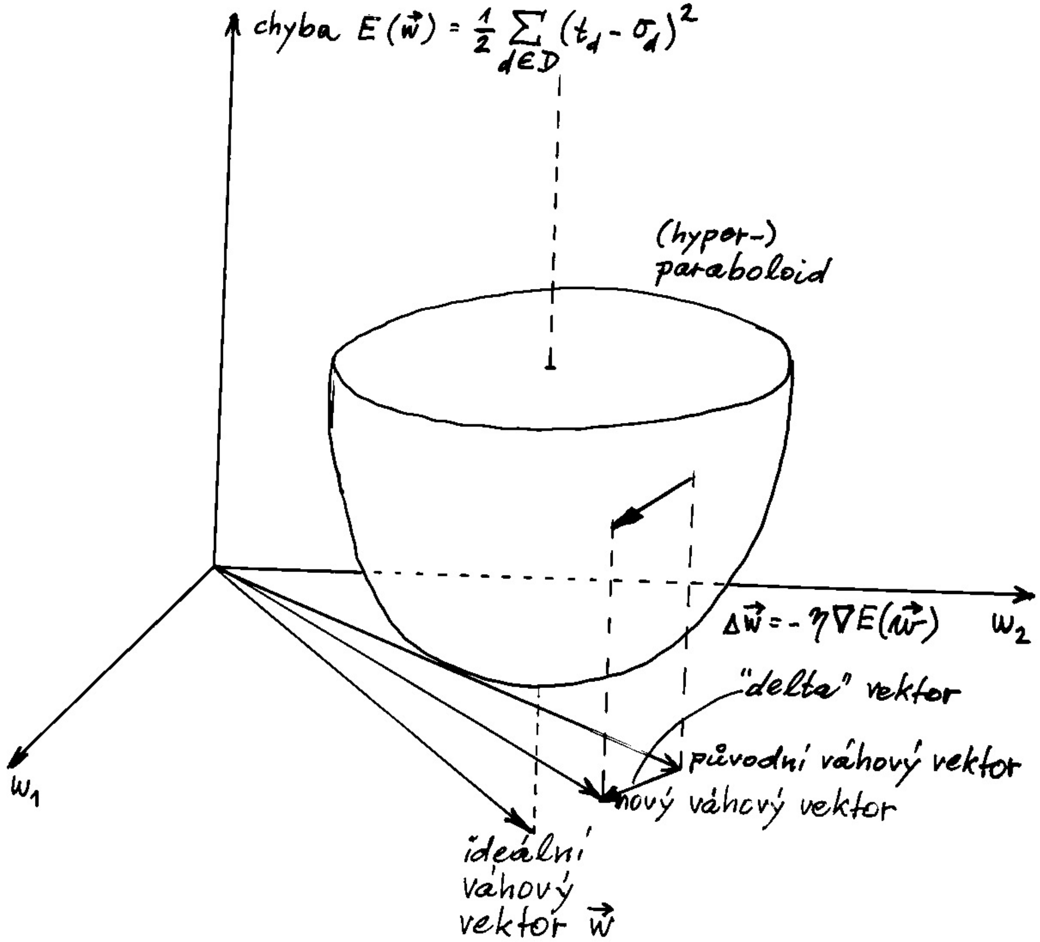
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

kde D je množina trénovacích příkladů, t_d je správný výstup pro trénovací příklad d, o_d je skutečný výstup získaný pro instanci d.



Cílem je najít hypotézu s minimální chybou. Pro lineární jednotky je (vzhledem k definici E) chybový povrch parabolický s jediným globálním minimem. (Parabola samozřejmě závisí na trénovací množině!)

Gradientní sestup pomáhá určit váhový vektor $\vec{w}$, jenž minimalizuje E: začne se s libovolným počátečním vektorem vah a tento je opakovaně modifikován v malých krocích. V každém kroku je $\vec{w}$ měněn ve směru, který dává nejstrmější sestup podél chybového povrchu. Konec: dosažení glob. minima.



Pravidlo "delta" posunuje váhový vektor tak, aby jeho projekce na (hyper-)paraboloid minimální chyby se pohybovala směrem negativního gradientu.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad \Delta\vec{w} = -\eta\nabla E$$

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

Výpočet směru nejstrmějšího sestupu po chybovém povrchu: počítá se derivace E vzhledem ke každé složce vektoru $\vec{w}$. Tento derivovaný vektor se nazývá gradient E vzhledem k $\vec{w}$, tj. $\nabla E(\vec{w})$:

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n}\right]$$

$-\nabla E(\vec{w})$ je rovněž vektor, specifikující směr nejstrmějšího sestupu na E. ($+\nabla E(\vec{w})$ udává směr nejstrmějšího vzestupu).

Trénovací pravidlo tedy bude:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \quad \text{kde } \Delta\vec{w} = -\eta\nabla E(\vec{w})$$

$\eta > 0$ je opět učící konstanta ovlivňující velikost kroku při hledání pomocí gradientního sestupu.

Jiný zápis: $\Delta w_i = -\eta \dfrac{\partial E}{\partial w_i}$, $w_i \leftarrow w_i + \Delta w_i$

tj. nejstrmější sestup nalezneme změnou každé složky (w_i) vektoru $\vec{w}$ úměrně $\partial E / \partial w_i$.

Nalezení gradientu v každém kroku není obtížné:

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

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

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

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

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

kde x_{id} označuje ~~jeden~~ jednu vstupní hodnotu trénovací instance d.

Takže trénovací pravidlo pro gradientní sestup je:

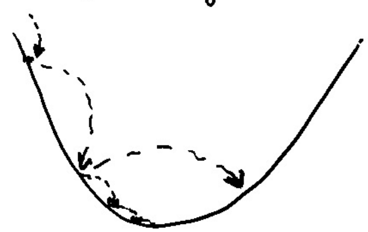
$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

Postup při trénování perceptronu:

- vytvoř náhodně inicializovaný malými hodnotami vektor vah;
- aplikuj lineární jednotku na všechny trénovací instance;
- vypočítej Δw_i pro každou váhu w_i;
- každou váhu w_i aktualizuj přičtením Δw_i;
- celý proces opakuj.

Vzhledem k existenci jediného globálního minima algoritmus konverguje k vektoru vah pro minimální chybu bez ohledu na to zda trénovací příklady jsou lineárně separovatelné nebo ne (předpokládá se malé η, např. 0.5, 0.1 apod.)

η: je-li příliš velké, pak hrozí riziko přeskočení minima (místo jeho dosažení):



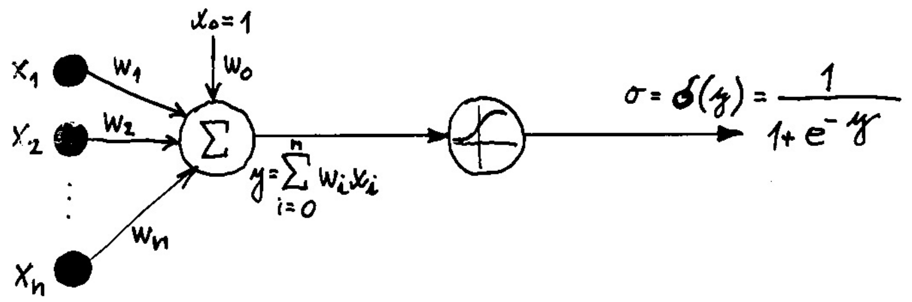
Proto se obvykle volí postupné snižování hodnoty η se vzrůstem počtu kroků.

Vícevrstvé sítě a trénovací algoritmus BP

Bylo ukázáno, že jednoduchý perceptron umožňuje pouze lineární rozhodovací hyperroviny.
Pro nelineární rozhodovací plochy je zapotřebí vytvořit z perceptronů tzv. vrstvené sítě.

Jaké jednotky jsou pro konstrukci sítě vhodné?
Lineární jednotky, pro něž bylo odvozeno Δ pravidlo, mohou vytvářet v kaskádách opět pouze lineární funkce, přičemž zapotřebí je, aby síť uměla reprezentovat rozeznávat nelineární závislosti.
Perceptron s nespojitým prahem neumožňuje výpočet derivace nutný pro stanovení gradientu.
Je proto zapotřebí vytvořit jednotku, jejíž ~~vstu~~ výstup je nelineární funkcí vstupů a zároveň jejíž výstup je diferencovatelnou funkcí vstupů.

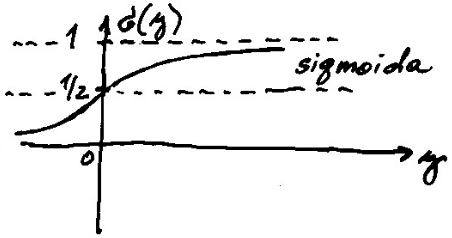
Používaným nejběžnějším řešením je tzv. sigmoidální jednotka - velmi podobná perceptronu, avšak založená na hladké diferencovatelné prahové funkci:



Zobrazená „sigmoidální jednotka" podobně jako perceptron napřed spočítá lineární kombinaci svých vstupů a pak na výsledek aplikuje práh (prahovou funkci).
Sigmoidální jednotka používá pro stanovení výstupu spojitou funkci zvanou sigmoida:

$$o = \sigma(\vec{w} \cdot \vec{x})$$

$$\sigma(y) = \frac{1}{1+e^{-y}}$$



(tato funkce se někdy také nazývá „logistická funkce").
Výstup je mezi 0 a 1, roste monotónně, mapuje velmi široký rozsah vstupní domény do úzkého výstupního rozsahu („squashing function").
Derivace je snadno vyjádřitelná v termínech vstupu:

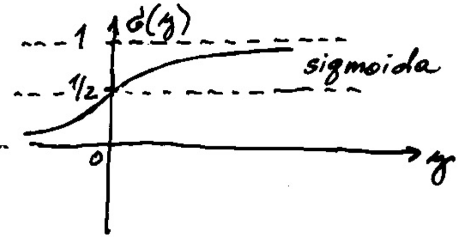
$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

Tato derivace je tedy využitelná pro výpočet gradientu.
Pozn.: někdy se používají jiné funkce, např. člen e^{-y} bývá nahrazován e^{-ky} ($k > 0$ ovlivňuje strmost prahu). Alternativou bývá místo $\sigma(y)$ také $\tanh(y)$.

Zobrazená „sigmoidální jednotka" podobně jako perceptron napřed spočítá lineární kombinaci svých vstupů a pak na výsledek aplikuje práh (prahovou funkci).
Sigmoidální jednotka používá pro stanovení výstupu spojitou funkci zvanou sigmoida:

$$o = \sigma(\vec{w}\cdot\vec{x})$$

$$\sigma(y) = \frac{1}{1+e^{-y}}$$



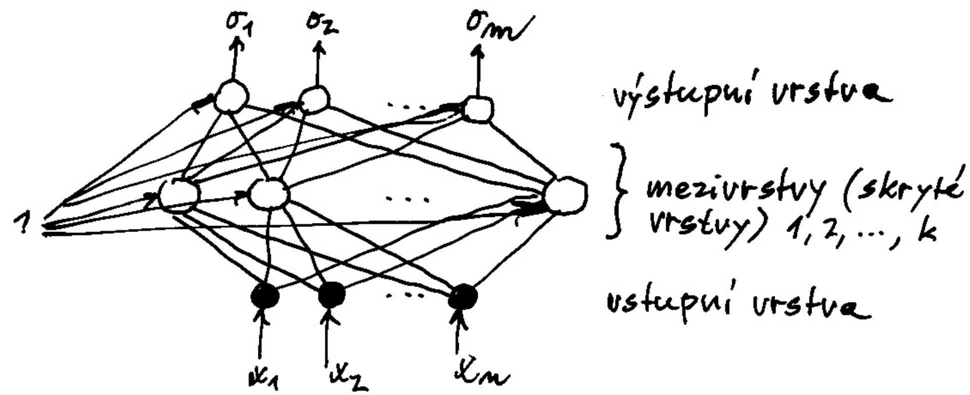
(tato funkce se někdy také nazývá „logistická funkce"). Výstup je mezi 0 a 1, roste monotónně, mapuje velmi široký rozsah vstupní domény do úzkého výstupního rozsahu („squashing function").
Derivace je snadno vyjádřitelná v termínech vstupu:

$$\frac{d\sigma(y)}{dy} = \sigma(y)\cdot(1-\sigma(y))$$

Tato derivace je tedy využitelná pro výpočet gradientu.
Pozn.: někdy se používají jiné funkce, např. člen e^{-y} bývá nahrazován e^{-ky} ($k>0$ ovlivňuje strmost prahu). Alternativou bývá místo $\sigma(y)$ také $\tanh(y)$.

Trénovací algoritmus Back Propagation (BP) (zpětné šíření)

Pomocí BP se síť učí potřebné hodnoty vah za předpokladu, že architektura sítě (počet jednotek a spojů) je neměnná.
Pro minimalizaci kvadrátu chyby (odchylky) mezi skutečným výstupem sítě a požadovaným se používá gradientní sestup.



Protože nyní uvažujeme síť s vícenásobným výstupem, definujeme chybu E jako součet chyb přes všechny výstupy jednotek:

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in \text{výstupy}} (t_{kd} - o_{kd})^2$$

kde výstupy je množina výstupních jednotek sítě, t_{kd} a o_{kd} jsou hodnoty požadované a dosažené pro <u>k</u>-tou výst. jednotku a trénovací příklad <u>d</u>.

Učící problém pro algoritmus BP znamená vyhledat v rozsáhlém prostoru hypotéz (definovaném všemi možnými hodnotami vah) hypotézu korektní vzhledem k trénovací množině.
Pro minimalizaci E se použije gradient sestupu.

V případě mnohovrstvé sítě může mít chybová plocha mnoho lokálních minim (na rozdíl od parabolické plochy ukázané dříve). Znamená to, že gradientní sestup je garantován pouze ve smyslu konvergence k nějakému <u>lokálnímu</u> minimu, <u>nikoliv</u> vůči minimu <u>globálnímu.</u> Navzdory tomuto problému byl BP užitečný v mnoha reálných aplikacích.

Algoritmus BP

BP(trénovací_příklady, η, n_{in}, n_{out}, n_{hidden})
/* Každý trénovací příklad je tvořen dvojicí $\langle \vec{x}, \vec{t} \rangle$, kde $\vec{x}$ je vektor vstupních hodnot a $\vec{t}$ je vektor správných výstupů.
η je „učící konstanta" (např. 0,5)
n_{in} je počet vstupních jednotek
n_{hidden} je počet jednotek skryté vrstvy
n_{out} je počet výstupních jednotek
Vstup z jednotky <u>i</u> do <u>j</u> je označen x_{ij}
Váha spoje mezi jednotkou <u>i</u> a <u>j</u> je w_{ij}
*/

- Vytvoř dopřednou síť s n_{in} vstupy, n_{hidden} ~~výstupy~~ skrytými jednotkami a n_{out} výstupy.
- Inicializuj všechny váhy malými náhodně generovanými hodnotami $\in [-0.05, +0.05]$ např.
- Dokud není splněna ukončovací podmínka
  DO {
  - Pro každou dvojici $\langle \vec{x}, \vec{t} \rangle$ z trénovací_příklady
    DO {

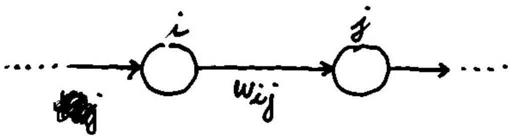
    /* Propaguj vstup směrem vpřed sítí: */
    1. Poskytni na vstup $\vec{x}$ a vypočítej o_u, tj. výstup každé jednotky <u>u</u> v síti.

    /* Propaguj chybu směrem zpět sítí: */
    2. Pro každou výstupní jednotku sítě <u>k</u> vypočti její chybu δ_k:
    $$\delta_k \longleftarrow o_k(1-o_k)(t_k - o_k)$$
    3. Pro každou skrytou jednotku <u>h</u> vypočti její chybu δ_h:
    $$\delta_h \longleftarrow o_h(1-o_h)\sum_{k \in výstupy} w_{kh} \cdot \delta_k$$
    4. Aktualizuj každou váhu sítě w_{ij}:
    $$w_{ij} \longleftarrow w_{ij} + \Delta w_{ij}$$
    $$\Delta w_{ij} = \eta \delta_j x_{ij}$$

    }

  }

Popsaný algoritmus je pro vrstvenou dopřednou síť s 1 skrytou vrstvou a 1 výstupní vrstvou (dvouvrstvá síť).



vstup z $\underline{i}$ do j (x_{ij})

$$\delta_n = -\frac{\partial E}{\partial y_n}$$ je chyba náležející jednotce $\underline{n}$

Hlavní smyčka (cyklus) algoritmu opakovaně iteruje přes trénovací příklady: pro každý příklad se spočítá chyba výstupu, dále gradient vzhledem k chybě daného příkladu, a pak se aktualizují váhy (všechny).

Iterace probíhá dostatečně dlouho (např. mnoho tisíc krát) za použití týchž příkladů, dokud není výkonnost sítě při klasifikaci dostatečně dobrá.

Chyba $(t-o)$ z Δ pravidla je nahrazena složitějším výrazem δ_k pro každou výstupní jednotku $\underline{k}$. δ_k je $(t_k - o_k)$ násobené faktorem $o_k(1-o_k)$, což je derivace sigmoidy.

Chyba pro jednotku h skryté vrstvy se počítá jako suma chyb δ_k pro každou výstupní jednotku ovlivněnou jednotkou $\underline{h}$. δ_k je váhováno w_{kh}, tj. váhou svoje z $\underline{h}$ do $\underline{k}$. Tato váha charakterizuje stupeň „odpovědnosti" $\underline{h}$ za chybu jednotky $\underline{k}$.

Popsaný algoritmus aktualizuje váhy inkrementálně, v závislosti na předkládání trénovacích příkladů.

Ukončovací podmínka:

- zastavení po předem daném počtu iterací
- snížení chyby pod nějakou stanovenou hodnotu
- splnění nějakého chybového kritéria pro testovací množinu příkladů (různých od trénovacích)

Volba ukončovací podmínky je důležitá:

- příliš málo iterací nesníží dostatečně chybu
- příliš mnoho iterací vede k „přetrénování" (bude diskutováno později).

Přidání tzv. momenta (urychlení konvergence)

Nejrozšířenější variací BP je změna způsobu aktualizace váhy tak, aby se míra změny v n-té iteraci odvíjela nějak z předchozích n-1 iterací:

$$\Delta w_{ij}(n) = \eta \delta_j x_{ij} + \overbrace{\alpha \Delta w_{ij}(n-1)}^{\text{momentový člen}}$$

$\Delta w_{ij}(n)$ je aktualizovaná hodnota v n-tém kroku, $0 \le \alpha \le 1$ je konstanta zvaná momentum. Efekt momenta: analogie setrvačnosti udržující kutálení míčku stejným směrem mezi dvěma iteracemi (proběhnutí lok. minima; udržení pohybu v ploché oblasti).

Učení v libovolných acyklických sítích

Popsaný učící algoritmus BP lze zobecnit na sítě libovolné hloubky – mění se jen procedura výpočtu δ:

$$\delta_r = \sigma_r (1-\sigma_r) \sum_{s \in \text{vrstva } m+1} w_{rs} \delta_s$$

tj. chyba r-té jednotky m-té vrstvy se počítá z hodnot chyb δ v m+1 vrstvě.

Odvození tréninkového algoritmu BP

BP tréninkový algoritmus upravuje hodnoty vah spojů mezi uzly sítě tak, aby byla minimalizována chyba výstupů sítě vzhledem k prvkům tréninkové množiny. Gradientní sestup zahrnuje iterace postupně přes jednotlivé trénovací pří~~kazy~~klady $\underline{d}$.

Pro každý trénovací pří~~k~~klad $\underline{d}$ je každá váha w_{ij} aktualizována přičtením opravy Δw_{ij}:

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}}$$

$\underline{E_d}$ je chyba pro příklad $\underline{d}$ přes všechny vstupní jednotky:

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in \text{výstupy}} (t_k - \sigma_k)^2$$

Zde výstupy je množina výstupních jednotek sítě, t_k je cílová hodnota (~~[illegible]~~ požadovaná h.) pro jednotku $\underline{k}$ pro trénovací příklad $\underline{d}$.

σ_k je skutečná hodnota výstupu jednotky $\underline{k}$ pro $\underline{d}$.

V dalším bude použito značení:

x_{ij} … i-tý vstup do j-té jednotky
w_{ij} … váha asociovaná s tímto vstupem
net_j … $\sum_i w_{ij} x_{ij}$ (váhovaný součet vstupů pro jedn. j)
σ_j … vypočítaný výstup jednotky j
t_j … požadovaný výstup jednotky j
σ … sigmoidální fce

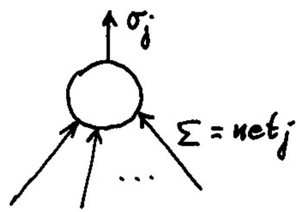
Klíčové je odvození výrazu pro $\frac{\partial E_d}{\partial w_{ij}}$, aby bylo možno implementovat gradientní sestup.
Váha w_{ij} ovlivňuje síť prostřednictvím net_j. Proto lze použít řetězcové pravidlo a napsat:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j}\,\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j}\cdot x_{ij}$$

Nyní je třeba odvodit $\partial E_d / \partial net_j$. Rozlišují se 2 případy: ① jednotka j je výstupní a ② jednotka j je ve vnitřní vrstvě.

① Výstupní jednotka

net_j může ovlivnit zbytek sítě pouze přes σ_j. Proto lze psát:



$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial \sigma_j}\,\frac{\partial \sigma_j}{\partial net_j}$$

$$\frac{\partial E_d}{\partial \sigma_j} = \frac{\partial}{\partial \sigma_j}\cdot\frac{1}{2}\sum_{k \in \text{výstupy}}(t_k - \sigma_k)^2$$

Derivace $(\partial/\partial\sigma_j)(t_k - \sigma_k)^2 = 0$ pro všechny případy vyjma $k = j$, proto lze psát:

$$\frac{\partial E_d}{\partial \sigma_j} = \frac{\partial}{\partial \sigma_j}\cdot\frac{1}{2}(t_j - \sigma_j)^2 =$$

$$= \frac{1}{2}\cdot 2\,(t_j - \sigma_j)\,\frac{\partial(t_j - \sigma_j)}{\partial \sigma_j} =$$

$$= -(t_j - \sigma_j)$$

Druhý člen vztahu $\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial \sigma_j}\,\frac{\partial \sigma_j}{\partial net_j}$

Protože $\sigma_j = \sigma(net_j)$ tak derivace je pouze derivace sigmoidální fce, což je $\sigma(y)(1-\sigma(y))$, proto:

$$\frac{\partial \sigma_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = \sigma_j(1-\sigma_j)$$

Dosazením do původního vztahu $\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j}\,x_{ij}$:

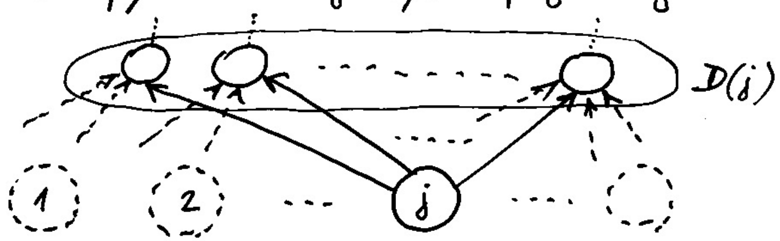
$$\frac{\partial E_d}{\partial net_j} = -(t_j - \sigma_j)\,\sigma_j\,(1-\sigma_j)$$

$$\Delta w_{ij} = -\eta\,\frac{\partial E_d}{\partial w_{ij}} = \eta\,(t_j - \sigma_j)\,\sigma_j\,(1-\sigma_j)\,x_{ij}$$

---

② Vnitřní (skrytá) jednotka

Vnitřní jednotka může ovlivňovat výstupy sítě nepřímo. Označme $D(j)$ všechny jednotky, jejichž přímé vstupy zahrnují výstup j-té jednotky.



Lze psát:

$$\frac{\partial E_d}{\partial net_j} = \sum_{k\in D(j)} \frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} =$$

chyba k-té jednotky

$$= \sum_{k\in D(j)} -\delta_k \frac{\partial net_k}{\partial net_j} = \sum_{k\in D(j)} -\delta_k \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j} =$$

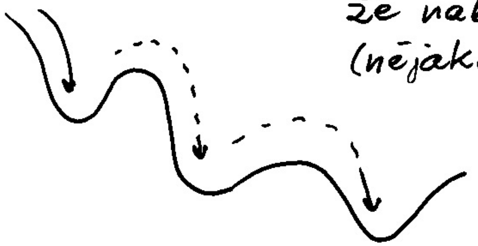
$$= \sum_{k\in D(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} = \sum_{k\in D(j)} -\delta_k w_{kj} o_j (1-o_j)$$

$$\delta_j = -\frac{\partial E_d}{\partial net_j}$$

$$\delta_j = o_j(1-o_j)\sum_{k\in D(j)} \delta_k w_{kj}, \quad \Delta w_{ij} = \eta \delta_j x_{ij}$$

Konvergence a lokální minima

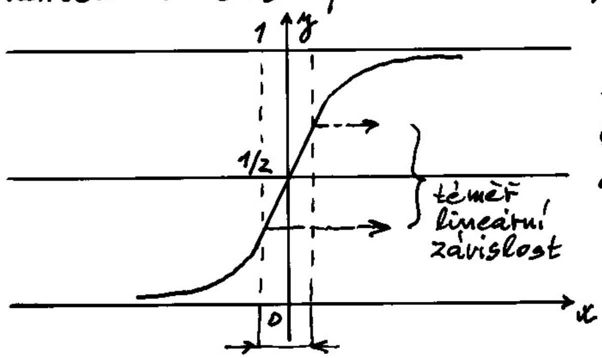
BP iterativně redukuje chybu E mezi skutečnou a požadovanou hodnotou na výstupu sítě. Povrch hyperplochy representující chybu jako funkci vah může obecně obsahovat velké množství lokalních extrémů (minim), do nichž se může gradientní sestup „chytit". Proto je u BP pouze zaručeno, že nalezne lokální minimum (nějaké), není záruka, že bude nalezeno globální minimum.



V praxi se ovšem BP ukazuje jako velmi efektivní pro mnoho aplikací (v podstatě funguje jako aproximátor funkcí s určitou minimalizací odchylky). Problém lokálních minim se ukázal jako nikoliv fatální.

Síť s velmi mnoho váhami odpovídá chybové hyperploše v mnohorozměrném prostoru (1 rozměr na váhu). Chytí-li se gradientní sestup do lokálního minima vzhledem k jedné z vah, nemusí to ještě znamenat, že bude v lokálním minimu vzhledem k ostatním váhám. Čím více vah v síti existuje, tím více dimenzí poskytuje „únikovou cestu" pro gradientní sestup, aby mohl utéci z lokálního minima, kde uvázl v jedné dimenzi (uviznutí se pozná podle toho, že přestane klesat chyba - síť se neučí).

Další zajímavé hledisko je způsob, jímž se váhy vyvíjejí spolu se vzrůstajícím počtem iterací. Jsou-li váhy inicializovány hodnotami poblíž nuly, pak na počátku gradientního sestupu representuje síť velmi hladkou funkci (lineární vzhledem ke vstupním hodnotám). Pouze poté, co váhy měly dostatek času patřičně vzrůst, lze očekávat nelineární (silně) průběh závislosti. Existuje tedy oprávněná naděje, že než dojde



k velkému vzrůstu hodnot vah do kladných a záporných hodnot, síť se dostatečně přiblíží oblasti globálního minima natolik, že i uvíznutí v blízkém (vůči globálnímu) minimu bude přijatelné.

Bohužel není známo, jak s jistotou předpovědět, kdy lokální minima způsobují obtíže. Existují obecné heuristiky pomáhající překonat problém lokálních minim:

- přidání tzv. momenta v mnoha případech umožní gradientnímu sestupu dostat se z úzkého lokálního minima (ALE! v principu může způsobit i to, že se gr. sestup dostane i z úzkého <u>globálního</u> minima do jiného lokálního minima)
- pomocí týchž trénovacích dat učit různé sítě (různě náhodně inicializované). Pokud výsledky učení povedou k různým lokálním minimům, vybere se síť, která dává nejlepší výsledky na základě testovacího souboru dat ≠ trénovacího.

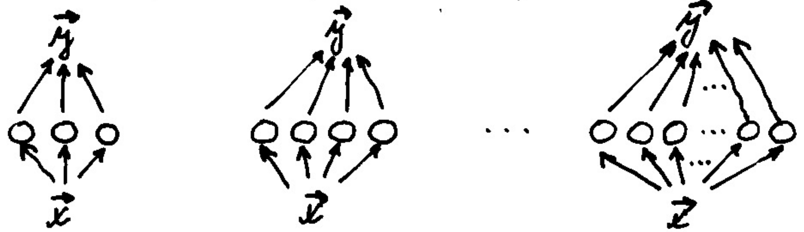
Representační schopnosti dopředných sítí

Jaký soubor funkcí lze pomocí dopředných sítí representovat? To mj. záleží na šířce a hloubce sítě. I když o tomto problému není mnoho známo, jsou k dispozici následující 3 výsledky:

- Booleovské funkce: každou bool. fci lze přesně representovat nějakou dvojvrstvou sítí, ačkoliv s rostoucím počtem vstupů poroste počet uzlů skryté vrstvy exponenciálně: Pro každý možný vstupní vektor se vytvoří 1 skrytá jednotka různá od ostatních, aktivovaná pouze tehdy, když se na vstupu objeví právě „její" vstupní vektor. Výstupní jednotka je typu OR, aktivovaná právě pro požadované vstupní hodnoty.
- Spojité funkce: každá ohraničená spojitá fce může být aproximována s libovolně malou chybou (konečné velikosti) sítí se 2 vrstvami (dokázáno matematicky v r. 1989). Týká se sítí se sigmoidálními jednotkami ve skryté vrstvě a lineárními jednotkami bez prahů ve výstupní vrstvě. Počet jednotek skryté vrstvy závisí na dané funkci.
- Libovolné funkce: mohou být aproximovány s libovolnou přesností trojvrstvou sítí (důkaz r. 1988). Obě skryté vrstvy jsou složeny ze sigmoidálních jednotek, výstupní vrstva z lineárních. Obecně není známo, jak určit počet jednotek skryté vrstvy. (Důkaz spočívá v tom, že se napřed ukáže, že libovolnou funkci lze aproximovat lineární kombinací mnoha lokálních funkcí s hodnotou = 0 všude kromě malých oblastí, a dále se ukáže, že 2 vrstvy sigmoidálních jednotek postačují k vytvoření dobrých lokálních aproximací).

Hledání efektivní architektury umělých neuronových sítí

a) Zvyšováním počtu ~~j~~ jednotek skryté vrstvy



- stanoví se počáteční minimální počet
- síť se natrénuje
- zjistí se chyba vůči testovací množině
- je-li chyba nižší než v předchozím cyklu (netýká se prvního trénování), ~~a~~ dosažená architektura se zapamatuje
- roste-li chyba, hledání se ukončí a použije se architektura s nejnižší chybou

b) Snižováním počtu jednotek skryté vrstvy

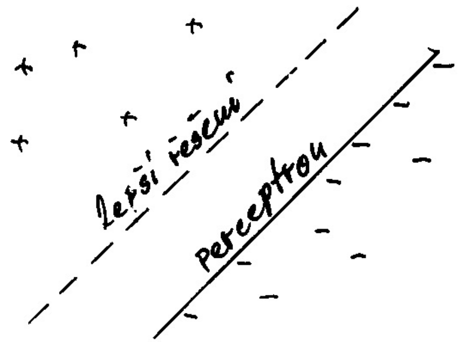
- postupuje se opačně, od nějakého maximálního počtu jednotek; počet se postupně snižuje a sleduje se chyba na testovací množině.

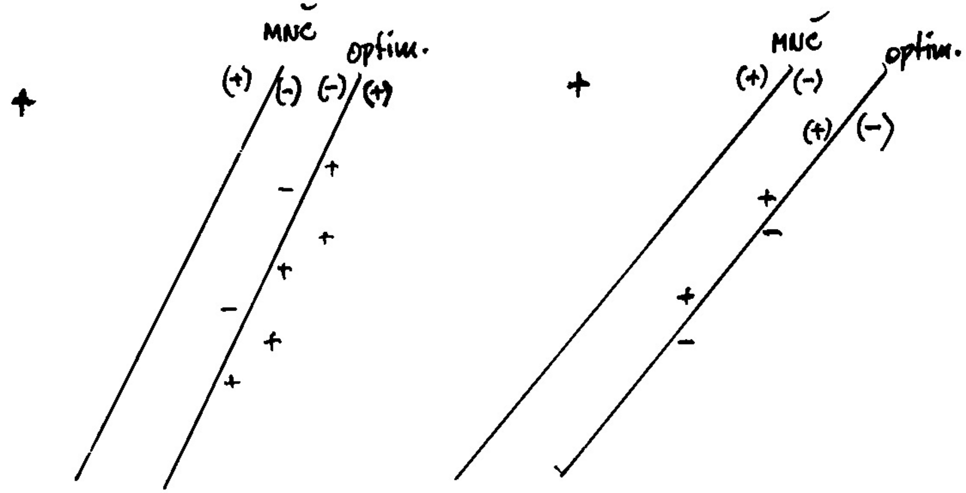
Vždy je vhodné sledovat i dobu trénování.
Někdy bývá výhodnější obětovat nepříliš významný pokles chyby za jednodušší architekturu.
Vždy je nutno postupovat stejně (stejná trénovací a testovací data, změna konstanty učení apod.).

Poznámka k chybové funkci MNČ

Výhoda: její derivace vzhledem k vahám existuje všude, tj. lze aplikovat gradientní sestup.

Nevýhoda:





Některé významnější aplikace

Algoritmus BP je nejpopulárnější učící metodou pro vícevrstvé neuronové sítě. BP a jeho variace byly použity pro řešení široké škály problémů:

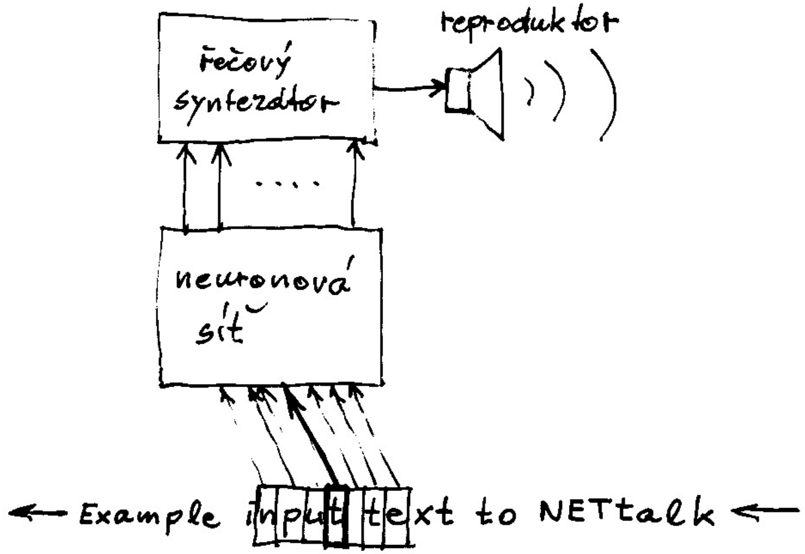
- rozpoznávání vzorů
- zpracování signálů
- komprese obrazů
- modelování nelineárních systémů
- rozpoznávání řeči
- lékařská diagnostika
- předpovědi
- řízení procesů

Nejpřitažlivější vlastností je schopnost adaptace, umožňující modelování složitých procesů pomocí učení se na základě vzorků měření či příkladů. Není nutná znalost specifických matematických modelů, ani expertní znalost.

NETtalk

Jednou z prvních aplikací bylo natrénování sítě pro konverzi anglického textu na řeč (r. 1987). Systém, známý jako NETtalk, se skládal ze dvou modulů:

- mapovací síť
- komerční řečový syntezátor



Mapovací síť měla 80 jednotek ve skryté vrstvě a 26 jednotek ve výstupní vrstvě. Výstup tvořil kód 1-z-26 pro kódování fonémů. Výstup sítě byl zároveň vstupem řečového syntetizéru, který generoval zvuky asociované se vstupními fonémy.

Vstupem do sítě byl 203-rozměrný binární vektor, který kódoval „okno" složené ze 7 následných písmen (29 bitů pro každý ze 7 znaků včetně punktuace; každý znak je kódován za použití kódu 1-ze-29 binárně).

Požadovaným výstupem byla hláska, resp. její kód poskytující výslovnost písmena nacházejícího se uprostřed okna.

Při trénování s použitím 1024 slov ze souboru příkladů anglických hlásek byl NETtalk schopen po 10 trénovacích cyklech poskytovat srozumitelnou řeč. Po 50 cyklech činila přesnost 95%. Síť byla schopna rozeznávat hranice mezi slovy a jak se postupně učila dále, silně připomínala dítě učící se mluvit. Síť uměla rozlišit samohlásky a souhlásky a při testování na novém odlišném textu dosahovala přesnosti 78%. Přidáním náhodného šumu k hodnotám vah či odstraněním několika neuronů klesala výkonnost sítě kontinuálně, nikoliv – jak je obvyklé u sériových digitálních systémů – náhle.

Obdobné komerční zařízení (DEC-talk) založené na bázi pravidel a využívající expertní systém s „ručně" zakódovanými lingvistickými pravidly je lepší. Význam NETtalku ovšem spočívá ve velmi krátké době vývoje (NETtalk se jednoduše učil z omezeného souboru příkladů, zatímco DEC-talk využíval pravidel, která vznikla jako výsledek mnohaleté analýzy mnoha jazykovědců.

Uvedená aplikace ilustruje snadnost (relativní vůči expertním systémům), jak lze pomocí umělé neuronové sítě vyvinout systém dokonce i tehdy, nerozumíme-li příliš či úplně řešenému problému.

Glove-Talk

Systém je založen na neuronové síti jako adaptivním rozhraní pro mapování gesta → řeč.
Využívá se 5 dopředných sítí. Gesta jsou popsána 16 parametry (x, y, z, natočení, směr vzhledem k pevné referenci + 10 úhlů prstů). Parametry jsou měřeny každou 1/60 vteřiny.

Systém je trénován tak, že napřed je vytvořeno slovo pomocí gesta, potom pohybem vpřed či vzad v jednom ze 6 směrů se určí zakončení slova (nahoru -s [plurál], k osobě -ed, od osoby -ing, ~~k osobě~~ doprava -er, doleva -ly, dolů nic).
~~Každá~~ Např. síť „Hand Shape" má 80 skrytých jednotek plně propojených na 66 výstupních neuronů (kódování 1-z-66). Tím se vytváří 66 „kořenových" slov.

Glove-Talk je schopen konvertovat 203 gest na slova s 99% přesností.

ZIP-Code Recognition

Rozeznávání PSČ. Viz obrázek.
Trénováno na 7291 příkladech, testováno na 2007 příkladech.
Síť má celkem 1000 jednotek, 64,660 spojů.
Přesnost ≈ 99%.

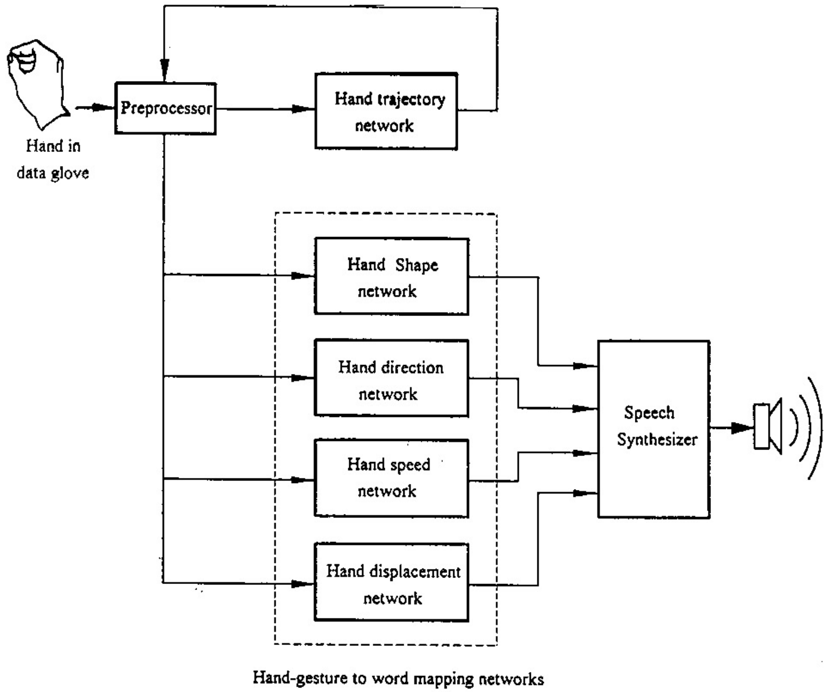


Figure 5.3.2
Glove-Talk: A neural network–based system that maps hand gestures to speech. (From S. S. Fels and G. E. Hinton, Glove-Talk: A neural network interface between a data-glove and a speech synthesizer, *IEEE Transactions on Neural Networks*, 4(1):2–8, 1993; © 1993 IEEE.)

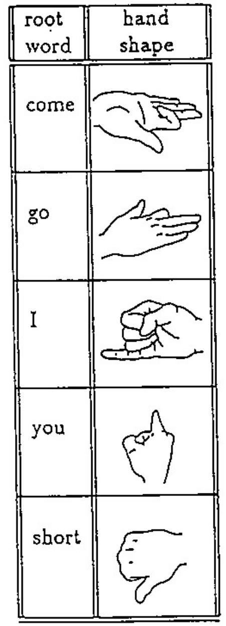


Figure 5.3.3
Examples of root words for several hand gestures (Adopted from S. S. Fels and G. E. Hinton, Glove-Talk: A neural network interface between a data-glove and a speech synthesizer, *IEEE Transactions on Neural Networks*, 4(1):2–8, 1993; © 1993 IEEE.)

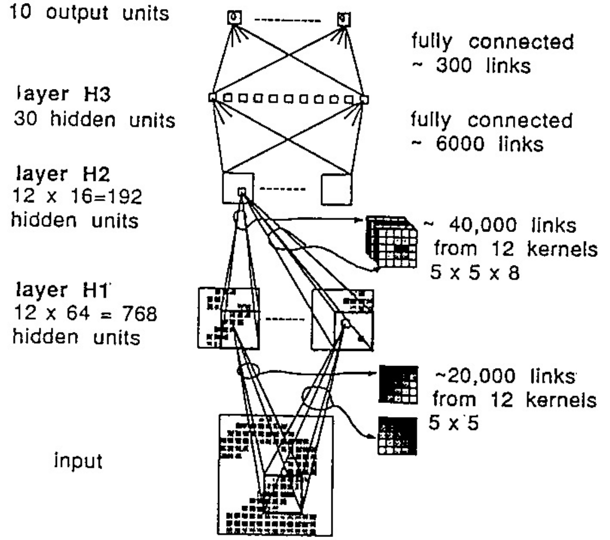


Figure 5.3.5
Network architecture for the handwritten ZIP code recognition neural network (From Y. Le Cun et al., 1989, with permission of the MIT Press.)

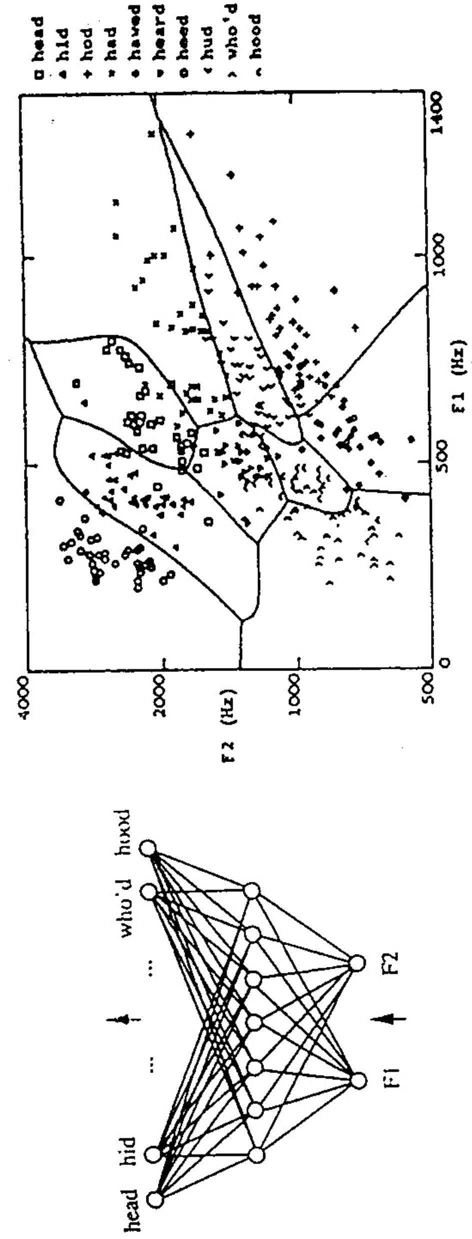
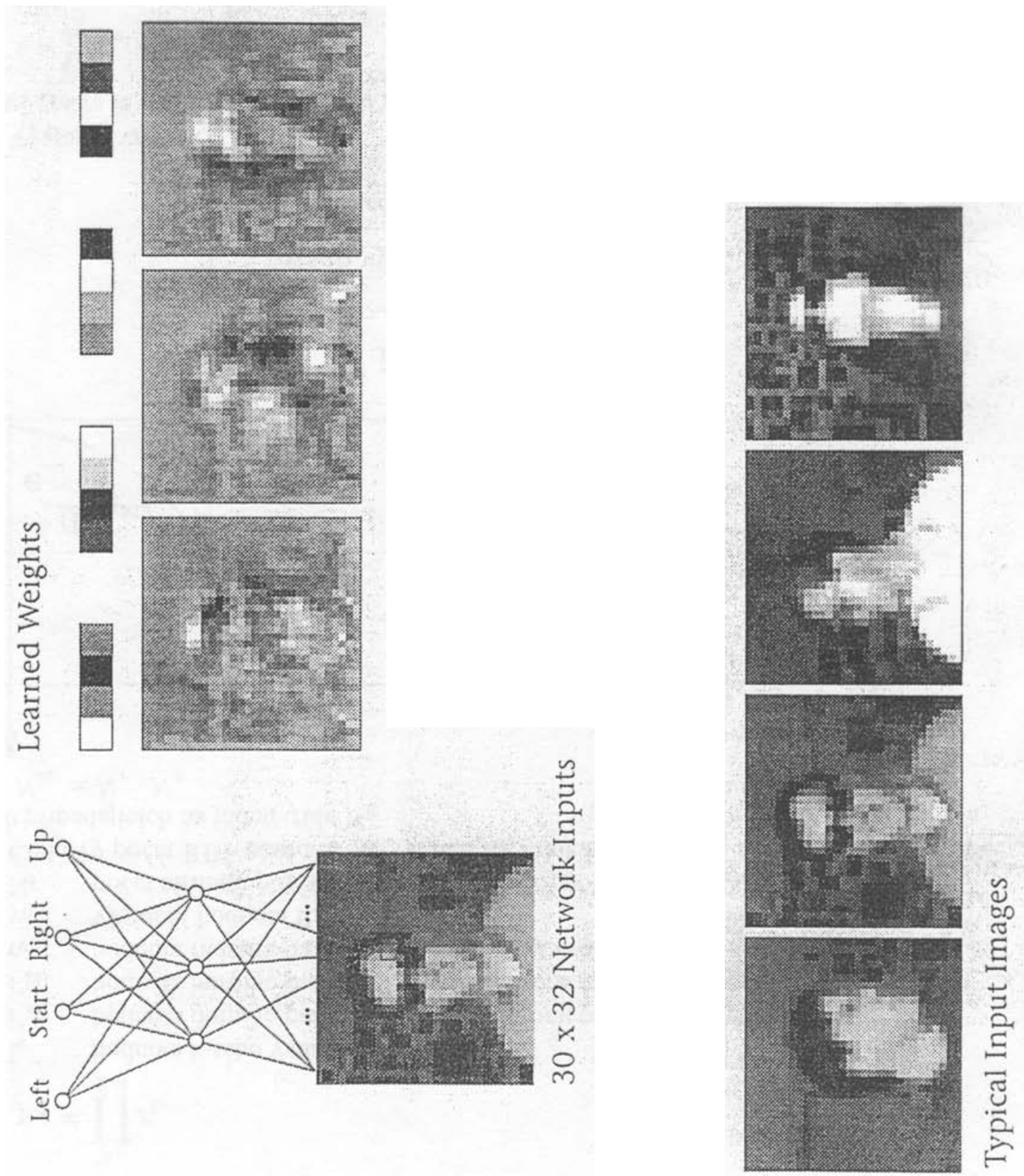


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). **The network** input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. **The network prediction is the** output whose value is highest. The plot on the right illustrates **the highly** nonlinear decision surface represented by the learned network. Points shown on the plot are **test** examples distinct from the examples used to train the network. (Reprinted by permission from **Haung** and Lippmann (1988).)



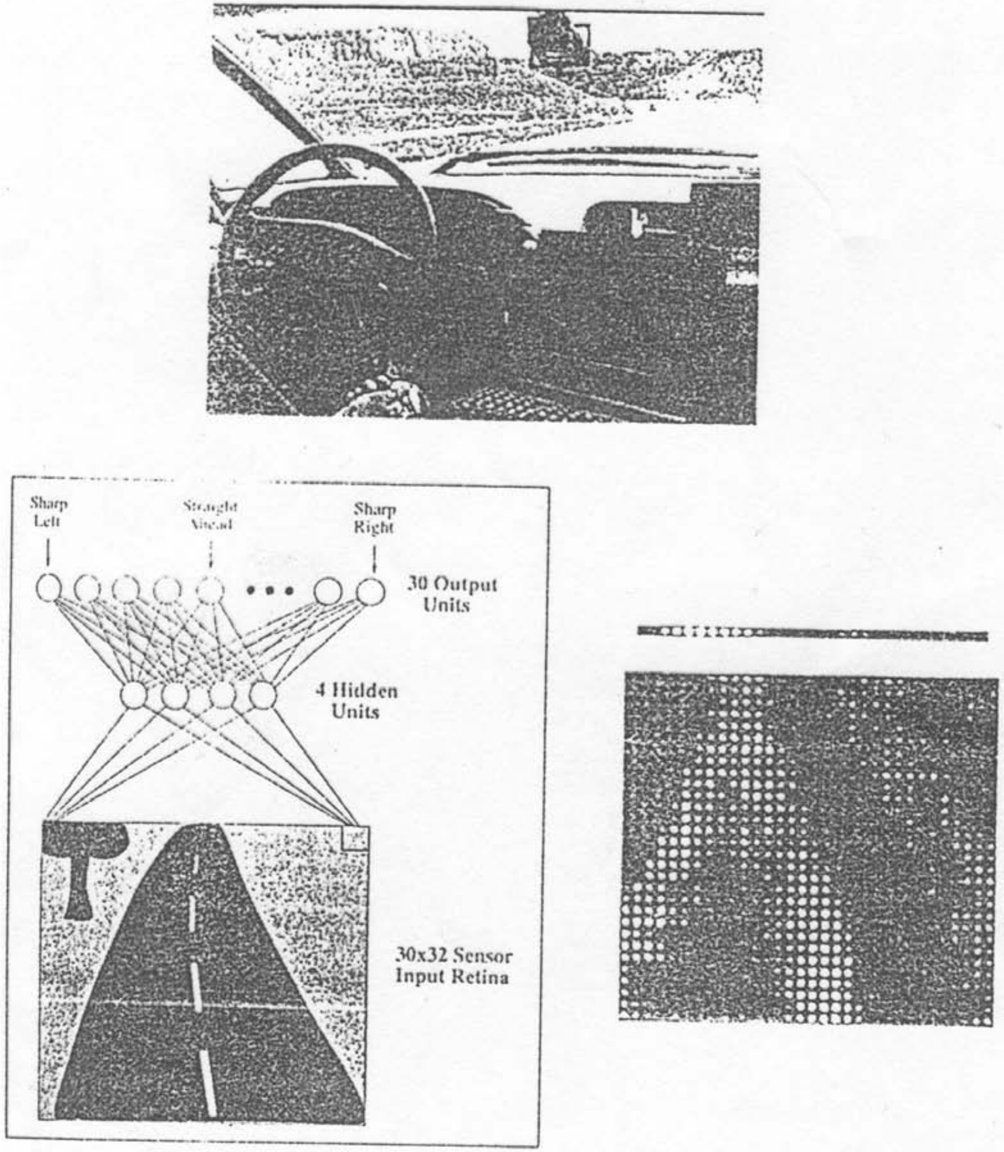


FIGURE 4.1

Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30×32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.