Debreceni Egyetem

Informatikai Kar

Neurális hálózatok MATLAB programcsomagban

Témavezető:

Dr. Fazekas István

Egyetemi tanár

Készítette:

Horváth József

Programtervező informatikus

Debrecen

2011

Tartalomjegyzék

1. Fejezet 5

Előrecsatolt neurális hálózatok 5

Visszacsatolás 8

Eltolás-súly 8

Tanulás 10

Neurális Hálózatok Struktúrái 11

Többrétegű perceptron 12

A többrétegű perceptron tanítása 14

Kötegelt tanítás 16

Szekvenciális tanítás 17

MATLAB példaprogram - XOR probléma 18

Neuronok a MATLAB szoftverben 26

Előrecsatolt hálózatok tanítása: A DE-LM algoritmus 28

Differential Evolution 28

A Levenberg-Marquardt algoritmus 31

A DE-LM algoritmus 32

2. Fejezet 36

RBF Hálózatok 36

Általánosított RBF hálózatok 38

Középpontok 39

Particle Swarm Optimization 43

PSO algoritmus lépései 45

MATLAB: Függvény approximáció RBF hálózatokkal 47

3. Fejezet 57

Szeparáló hipersíkok 57

Lineárisan nem szeparálható adathalmazok 59

Magfüggvények 60

Esettanulmány - DNS szekvenciák osztályozása 61

SM-SVM (*Soft Margin Support Vector Machine)* 63

SMO – Szekvenciális Optimalizáció [16] 64

MATLAB: Legkisebb Négyzetes Tartóvektor Gépek [13] 65

MATLAB: kétváltozós függvények approximációja 68

Összegzés 70
Irodalomjegyzék 72
Ábrák jegyzéke 74
Köszönetnyilvánítás 76

Bevezetés

Az utóbbi évtizedekben föllendült a mesterséges intelligencia (MI) kutatása. Dolgozatomban az ezen kutatási területnek csak egy kis szeletével fogunk foglalkozni a *neurális hálózatok*kal. A számítástechnikában használatos neurális hálózatok biológiai indíttatásúak. Megalkotásuk célja valószínűleg az emberi agy bizonyos funkcionalitásának gépi alkalmazása lehetett. A biológiai, illetve mesterséges neurális hálózatok között számos párhuzam és ellentét megfigyelhető, de funkcionalitásukat tekintve többé-kevésbé ekvivalens módon viselkednek. Az MI ezen ága főleg osztályozási, approximációs (függvényközelítés) problémákkal foglalkozik, vagyis olyan feladatokkal, amelyekre nem léteznek egzakt algoritmusok.

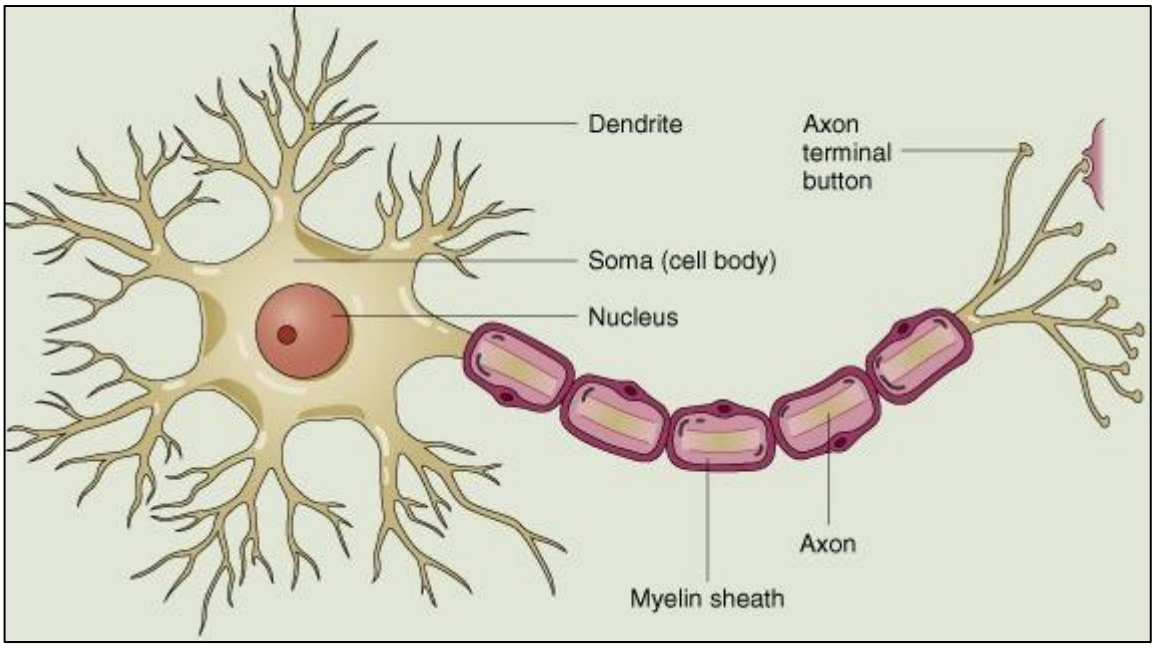
A neurális hálózatokat eredetileg bizonyos képek felismerésére alkották meg. Ezen feladat tipikusan egy osztályozási problémának feleltethető meg. Napjainkra a neurális hálózatok túlnőttek a velük szemben kezdetben támasztott követelményeken, és számos egyéb dologra is alkalmazhatóak, mint például idősor előrejelzés, tőzsdei árfolyam változásának becslése, de neurális hálózatok alkalmazhatóak a képfeldolgozás területén is.

Mi is az a neurális hálózat? A dolgozat bevezető részében a neurális hálózatokról általánosságban fogunk beszélni, megpróbálunk rájuk egyfajta közelítő definíciót adni. Megadjuk a főbb különbségeket és hasonlóságokat a biológiai, illetve a mesterséges neurális hálózatok között. Ezt követően a dolgozat második részében a neurális hálózatok négy különböző osztályával fogunk megismerkedni; az előrecsatolt [3],[6], az RBF, az SVM [7],[8] illetve az evolúciós neurális hálózatokkal [4],[5] egy speciális matematikai szoftver, a MATLAB [9] vonatkozásában. A dolgozat során a hangsúly főleg ezen hálózatok tanításán lesz, amelyhez olyan algoritmusok kerülnek bemutatásra, amelyek a napjainkban felmerülő problémákra is könnyedén alkalmazhatóak.

1. Fejezet

Előrecsatolt neurális hálózatok

A neurális hálózatok tárgyalásának alapja, hogy megismerjük ezek felépítését. Bármely neurális hálózatról elmondható, hogy neuronokból épül föl. Ezen neuronok olyan agysejtek, amelyek alapfeladata valamilyen elektromos jelek továbbítása [1] más neuronok felé a kapcsolataikon keresztül. Valószínűsíthető, hogy az agy információ feldolgozó képessége és kapacitása e neuronok valamilyen hálózatából alakul ki.

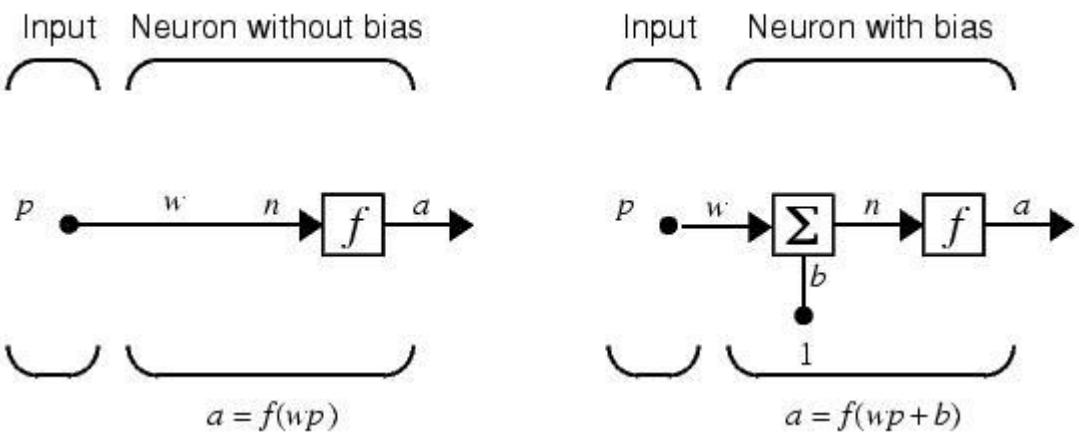


1. ábra Biológiai Neuron sematikus rajza. – Wikipedia illusztráció

Az 1. ábra egy egyszerűsített biológiai neuront ábrázol. Könnyedén leolvashatóak a neuron főbb részei. A továbbiakban azon részeit fogjuk bemutatni a fenti neuronnak, amelyek fontosak a dolgozatunk további részeinek megértése céljából. A dendrite a görög *déndron* szóból származik, melynek magyar megfelelője fát jelent. Az agysejtek eme részét könnyen beazonosíthatjuk az 1. ábra bal oldalán látható. Nevéhez hűen tényleg hasonlít egy fához.

Feladatát tekintve a dendritek tekinthetőek a neuron bemeneteinek; elektrokémiai folyamatok során valamilyen elektromos jeleket továbbítanak a sejtmag, azaz a nucleus felé. Fontos azonban megjegyezni, hogy a neuronok nem a dendritjeiken keresztül csatlakoznak egymáshoz, hanem a dendriteken található szinaptikus kapcsolatokon keresztül. Ezen szinaptikus kapcsolatok pontosan két neuron között jelátvivő szerepet töltenek be. Miután a neuron sejtmagja feldolgozta az elektromos „kisülésként" kapott információt a sejttesten keresztül (szóma) az axon felé egy valamilyen mértékű elektromos jelet küld. Ezen elektromos jel a sejt Axonján át jut el egy axon-terminalnak nevezett részre, amely funkcionalitását tekintve azonos a dendritekkel, azzal a különbséggel, hogy információküldő szerepet játszik, hasonlóan szinaptikus kapcsolatokon keresztül más neuronok dendritjeivel.

Látható, hogy egy biológiai neuron három fő részből tevődik össze így bemenet, központi egység, illetve kimenet. 1943-ban McCullock és Pitts (lásd [1]) megalkottak egy mesterséges neuront, amely funkcionalitását tekintve azonos egy biológiai neuronnal. Ezt nevezzük *mesterséges neuron*nak. A mesterséges neuron vagy más néven *Perceptron* egy egyszerűsített biológiai neuron viselkedését szimulálja. A 2. ábrán egy ilyen perceptront láthatunk, ahogyan azt a MATLAB [9] matematikai szoftverben implementálták Rosenblatt-féle modell [3] szerint.



2. ábra Rosenblatt-féle Perceptron modell, ahogyan azt a MATLAB [9] szoftverben használhatjuk.

A 2. ábra két részből tevődik össze, de még mielőtt rátérnénk, e különbségek tárgyalására ismerkedjünk meg a mesterséges neurális hálózat alapegységeivel, a mesterséges neuronokkal részletesebben.

Minden mesterséges neurális háló irányított kapcsolatokkal összekötött neuronokból épül föl. Ezen kapcsolatokat *szinaptikus kapcsolat*oknak vagy egyszerűen *szinapszis*oknak nevezzük. Ekkor jelölje $\omega_{i,j}$ a j-edik neurontól az i-edik neuronig vezető *szinapszis*t. Ezen *szinapszis*ok rendelkeznek egy tulajdonságnak, amit a kapcsolat *súlyának* nevezünk. Ezen súly meghatározza a kapcsolat erősségét és előjelét [1],[2].

$$In_i = \sum_{k=0}^{m} \omega_{i,k} * p_k \, . \tag{1}$$

Minden neuron első lépésként a bemeneteinek súlyozott összegét állítja elő. Ez azt jelenti, hogy adott *p* bemeneti vektor, továbbá *p* minden értékéhez egy ω súly. Fontos megjegyezni, hogy az (1)-es képlet megadható az alábbi módon:

$$\langle \omega, p \rangle \, . \tag{2}$$

Itt a $\langle \, , \rangle$ a belső szorzat, T pedig a transzponálás művelete. Ezek alapján a (2)-es képlet az alábbi formában is felírható.

$$\langle \omega, p \rangle = \omega^T * p = \sum_{k=0}^{m} \omega_{i,k} * p_k \, .$$

Az (1), illetve (2) képletek egymással ekvivalensek. A következő fázisban a neuronnak valamilyen módon reagálnia kell az éppen feldolgozott inputra. Azaz hasonlóan a biológiai neuronhoz, amely egy elektromos jelet küld a kimenetei felé, a mesterséges neuronnak is egy kimenetet kell küldeni a hozzá kapcsolódó neuronoknak. A kimenetet úgy kapjuk meg, hogy egy f(.) *aktivációs függvény*t alkalmazunk az (1) (2) képletek alapján kapott In_i súlyozott összegre:

$$O_i = f(In_i) = f\left(\sum_{k=0}^{m} \omega_{i,k} * p_k\right). \tag{3}$$

Az aktivációs függvénytől általában két dolgot várnak el [1]. Az egyik, hogy amennyiben *p* „igaz” bemenetet jelent, az aktivációs függvény egy „+1”-hez közeli értéket generáljon, míg „hamis” bemenetre „0” (vagy „-1” értéket). Az utóbbi számos tényezőtől függ. A neurális hálózatok alkalmazására nem létezik konkrét séma, amelyet tetszőleges problémára alkalmazhatnánk. Minden problémára más és más neurális hálót kell létrehoznunk. Az aktivációs függvénytől egy másik dolgot is elvárunk, mégpedig azt, hogy ne legyen lineáris.

ez elsősorban azért fontos, mert a neurális hálók egyfajta függvényt valósítanak meg, amely függvény közelíti azon adatokat, amelyekre a hálót betanítjuk. Ha lineáris aktivációs függvényeket használnánk, a neurális hálózatunk egy lineáris függvényt valósítana meg és lineáris függvényt illesztene a tanulóadatokra. Ez sok esetben azt eredményezi, hogy a hálózat hibája nem csökken le kellőképpen. A numerikus matematikában ezt nevezzük alulillesztésnek.

A perceptron tanítása

Fontos megjegyezni, hogy egyetlen perceptron önmagában is képes bizonyos feladatok megoldására, így például egyszerű osztályozási feladatok végrehajtására. Mielőtt megadnánk a perceptron tanulásának folyamatát tisztáznunk kell két fontos fogalmat, a *visszacsatolás*t és az *eltolás-súly*t.

A továbbiakban jelölje x(n) a neuron *p* bemeneti vektorát az n-edik időpillanatban.

Visszacsatolás

Tetszőleges neuron egy x(n) bemenet hatására generál egy y(n) kimenetet. Azon adatokat, amelyek esetében {x(n), d(n)} ismert, tanulóadatoknak nevezzük, ahol x(n) a neuron bemenete d(n) pedig az a kimenet, amit elvárunk a neurontól.

$$(d(n) - y(n))^2 \to 0\,. \tag{4}$$

A (4)-es összefüggés által definiált mennyiséget a neuron négyzetes hibájának nevezzük. Ezen hibát csökkentve y(n) és d(n) közötti különbség is csökkenni fog. Ezt úgy tudjuk elérni, hogy a neuron bemeneti súlyait valamilyen módon megváltoztatjuk. Ezt a folyamatot nevezzük *visszacsatolás*nak.

Eltolás-súly

Ahhoz, hogy megértsük az eltolás-súly jelentőségét tekintsük a következő példát. Legyen adott a következő halmaz:

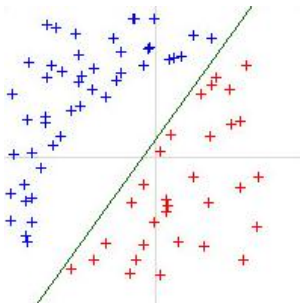
$$\{(x(0), d(0)), (x(1), d(1)),, \dots, \big(x(m), d(m)\big), \big(x(m+1), d(m+1)\big), \dots, \big(x(n), d(n)\big)\} \tag{5}$$

Amely halmazban jelölje x(0), x(1), ... ,x(m) az egy osztályba tartozó pontokat, illetve x(m+1),..., x(n) egy másik osztályba tartozó pontokat. Azt mondjuk, hogy e két ponthalmaz elválasztható egy hipersíkkal, amelyet *szeparáló hipersík*nak nevezünk olyan módon, hogy x(k), k=0,1,...,m, a szeparáló hipersík jobb, x(l), l=m+1,m+2,...,n, a szeparáló hipersík bal oldalán találhatóak. Tetszőleges szeparáló hipersík megadható a (2) összefüggéssel (pontosabban szólva $\langle \omega, p \rangle = 0$ egyenlettel). Ezen összefüggés egyetlen hátránya, hogy az így definiált hipersík átmegy az origón. Az adathalmazunk szempontjából azonban egyáltalán nem garantált, hogy egy origón áthaladó hipersík szeparálni tudja, ezért el kell azt tolnunk valamilyen irányban. Azt a vektort, amellyel eltoljuk a szeparáló hipersíkot *eltolás-súly*nak nevezzük.

Vegyük észre, hogy ez esetben mind az (1), mind a (2), illetve (3) összefüggések megváltoznak. Ezen összefüggések a 2. ábra bal oldali perceptronját írják le. Ahhoz, hogy a jobb oldali perceptront leírjuk meg kell adnunk ezt az eltolás-súlyt.

$$In_i = \sum_{k=0}^{m} \omega_{i,k} * p_k + b \ . \tag{6}$$

Az (1) összefüggés azt írta le, amelyet egy neuron első lépésként végrehajt minden bemenetén kapott vektorokkal. Vegyük észre, hogy a (6) összefüggésben már helyet kapott a b eltolás-súly paraméter is. Ennek hatását a 3. ábrán figyelhetjük meg.



3. ábra Az (5) halmaz pontjai Descartes féle koordinátarendszerben ábrázolva. Kék/Piros jelzik a két osztályt amelyet a neuronnak szeparálnia kell a két halmazt elválasztó egyenes megtalálásával.

Az 3. ábrán látható, hogy a szeparáló egyenes nem az origón halad át, hanem el lett tolva b értékével. Az is könnyen látható, hogy a két adathalmaz b eltolás-súly használata nélkül nem lenne szeparálható, mert lennének olyan pontok, amelyek nem a megfelelő osztályba kerülnének besorolásra. Ennek következtében a (4) összefüggés által definiált hibát bővíthetjük m-dimenziós esetre.

$$E(n) = \sum_{i=0}^{m} (d_i(n) - y_i(n))^2. \quad (7)$$

Az ilyen módon definiált hibát *Mean Squared Error (MSE)* –nak nevezzük [4]. Amennyiben 3. ábrának megfelelő esetben nem használunk eltolás súlyt a (7) összefüggés által definiált hiba értéke nem fog 0-hoz konvergálni, hanem megragad egy valamely $e \in \mathbb{R}$ értéknél. Ezt az okozza, hogy a perceptron nem tud olyan szeparáló egyenest konstruálni, amely elválasztaná egymástól a két ponthalmazt olyan módon, hogy a „piros" pontok az egyenes jobb míg a „kék" pontok az egyenes bal oldalán helyezkedjenek el.

Egy másik fontos paramétert kell még tisztáznunk. Mit jelent a $\omega_{k,j}$ paraméter a szeparáló hipersík vonatkozásában. Az eltolás-súly megadja, hogy milyen mértékben toljuk el a szeparáló hipersíkot az origóhoz képest, ahogyan azt a 3. ábrán láthatjuk. A szinaptikus súly e szeparáló hipersík egy normálvektorát jelenti. Vagyis a szinaptikus súly megadja, hogy a szeparáló hipersík milyen meredek illetve, hogy melyik irányba „dől".

Tanulás

Az egységes kezelés érdekében érdemes $\omega_0 = b$ és $x_0 = 1$ jelölést használni (azaz a súlyvektor első koordinátája b, az inputé pedig 1). Legyen a két halmaz A1 és A2. A két halmazt szeparáló hipersík ω normálvektorára legyen $\langle x(n), \omega \rangle > 0$, ha $x(n) \in A2$ és $\langle x(n), \omega \rangle < 0$, ha $x(n) \in A1$. Az ilyen módon leírt tanítási folyamat az alábbi lépések legfeljebb véges sokszori végrehajtásával írható le.

1. lépés: Kezdeti ω vektor paramétereinek megválasztása. Jó döntés lehet, ha egyenletes eloszlású véletlen számokkal töltjük fel a (0,1) nyílt intervallumból.
2. Adott tanulóadatra a perceptron kimenetének meghatározása.
3. Ha $\langle x(n), \omega(n) \rangle < 0$, de $x(n) \in A2 \rightarrow x(n)$ irányába fordítjuk el a súlyvektort, vagyis $\omega(n+1) = \omega(n) + \eta(n) * x(n)$.
4. Ha $\langle x(n), \omega(n) \rangle > 0, de\ x(n) \in A1 \rightarrow x(n)$-nel ellentétes irányban fordítjuk el a súlyvektort, vagyis $\omega(n+1) = \omega(n) - \eta(n) * x(n)$.
5. Ha $\langle x(n), \omega(n) \rangle < 0\ és\ x(n) \in A1\ vagy\ \langle x(n), \omega(n) \rangle > 0\ és\ x(n) \in A2$, akkor $x(n)$ pontot jól osztályozta a perceptron, így $\omega(n+1) = \omega(n)$.

Itt $\eta(n)$ tanulási paraméter.

A tanulópontok száma véges. Ekkor belátható, hogy véges sok tanulópont esetén véges sok lépésben meghatározható a súlyvektor. Azonban a súlyvektor meghatározásához nincs szükségünk minden tanulópontra. Elegendőek azon tanulópontok a fenti eljárás végrehajtásához, amelyeket a hálózat rosszul osztályoz. Ezen pontokat *korrekciós pont*oknak [10] nevezzük. Ha a súlyokat a fenti eljárás szerint beállítjuk $\omega_{i,j}$ szerint minimalizáljuk (7) összefüggést. Véges sok lépés után a perceptron helyesen fogja osztályozni a tanulópontokat.

Neurális Hálózatok Struktúrái

Önmagában a perceptron képes egyszerű osztályozási feladatok ellátására. A gyakorlatban azonban nem minden osztályozási feladat olyan egyszerű, hogy egy perceptron egyedül képes legyen megfelelő szeparáló hipersíkot találni. Ahhoz, hogy a bonyolultabb feladatok is megoldhatóak legyenek, a neuronokat valamilyen analógia szerint össze kell kapcsolni más neuronokkal. Az összekapcsolás módja szerint kétféle hálózatot különböztetünk meg:

1. Előrecsatolt (feed-forward) neurális hálózat [1],[2],[4],[10]. Az előrecsatolt neurális hálózat egyetlen ismeretlen paramétere a hálózatban szereplő szinaptikus kapcsolatok súlyaiból alkotott súlyvektor. A hálózat adott $x(n)$ bemeneti adatra ugyanazon $y(n)$ kimenetet fogja adni bármennyi iteráció után.
2. Asszociatív/Visszacsatolt/Rekurrens (recurrent) neurális hálózat [1],[4],[5],[6]. Az asszociatív neurális hálózatok jellegzetessége, hogy a szinaptikus kapcsolataik nem csak más neuronok bemeneteire vannak rákapcsolva, de saját bemenetükre is. Ezen hálózati struktúra egy gráffal szemléltethető a legjobban. Az asszociatív háló egy olyan gráf, melyben tetszőleges hosszúságú köröket találunk. Miután e hálózatokban tetszőleges neuron kimenete kapcsolódhat egy másik neuron bemenetére, ezért a hálózat aktuális kimenete már nem csak a hálózat szinaptikus kapcsolatainak súlyai által generált vektortól, de a hálózat korábbi állapotától is függ [4]. Ennek köszönhetően a rekurrens hálózatokban kialakul egyfajta memória.

A dolgozat következő részében az előrecsatolt neurális hálózatokkal fogunk foglalkozni, illetve ezek speciális változataival. A hangsúly végig ezen hálózatok tanításán lesz, hogy az

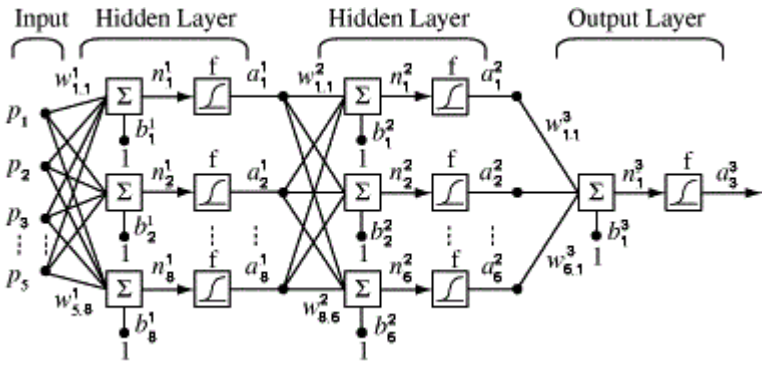
milyen elven zajlik le a MATLAB matematikai szoftverben. Ezt követően megismerkedünk két olyan tanulóalgoritmussal, amelyek nem találhatóak meg a MATLAB-ban, de ezek lényegesen javítják az előrecsatolt hálózatok approximáló képességeit. Így a következőkben bemutatandó előrecsatolt hálózatok a következők lesznek:

- Többrétegű perceptron (Multi Layer Perceptron – MLP) [4],[10]. Ahhoz, hogy tárgyalni tudjuk a többrétegű hálózatok viselkedését illetve tanításuk módszereit, először is tisztáznunk kell a rétegek fogalmát. Ellentétben a Perceptronnal az többrétegű perceptronban a neuronok rétegelt architektúrát alkotnak.
- Radial Basis Function (RBF) hálózatok: Gyakran találkozhatunk olyan problémákkal, amelyeknél az adataink nem lineárisan szeparálhatóak az adott térben, amelyet meghatároz ezen adatok reprezentációja. Ilyenkor az adatokat egy magasabb dimenziójú térbe transzformáljuk ahol ezen adatok lineárisan szeparálhatóak. Ezen probléma megoldására alkották meg az RBF hálózatokat. Az ilyen magasabb térbe való transzformálás egy $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ $(n \ll m)$ leképezés.
- Support Vector Machines (SVM)[7][8]: Az RBF hálózatokhoz hasonló $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ $(n \ll m)$ leképezésen alapulnak, vagyis tetszőleges $x(n) \rightarrow f(x(n))$ adatokhoz keres szeparáló hipersíkot. A föntebb említett megoldásokkal közös „hátránya" az, hogy két lineárisan szeparálható adathalmazhoz nagyon sok szeparáló hipersík megadható. Az SVM nagy előnye, hogy az optimális hipersíkot találja meg.

Többrétegű perceptron

Többrétegű perceptronról akkor beszélünk, ha a bemeneti, illetve kimeneti rétegek mellett a hálózatban megjelenik egy vagy több rejtett réteg is. Egy ilyen hálózatot ábrázol a 4. ábra a MATLAB Neural Network Toolbox [9] megvalósítás szerint. Látható, hogy a hálózatban egy rétegben tetszőleges számú neuron helyezhető el. Arra a kérdésre, hogy egy probléma megoldásához rétegenként hány neuron szükséges, még ma sem tudunk pontos választ adni [4]. A többrétegű hálózatok közös jellemzője, hogy adott k-adik réteg neuronjainak kimeneti szinaptikus kapcsolatai csak a (k+1)-edik réteg neuronjainak bemeneti szinaptikus kapcsolatai lehetnek.

Általában, ha egy probléma megoldása során a hálózat hibája nem csökken az adott korlát alá a rejtett neuronok számát igyekszünk növelni. Személyes kutatásaim azt mutatják, hogy egy rejtett réteg neuronjainak száma nem feltétlen javítja a háló approximáló képességét, ha meghaladja az adott réteget megelőző neuronok számának, illetve a következő réteg neuronjai számának, a szorzatának a felét. Ha ez az eset állna elő, célszerűbb egy új réteget létrehozni a hálózatban az adott réteg után. Fontos továbbá megjegyezni, hogy az előrecsatolt neurális hálózatok esetében a választott tanítóeljárás megszabja a hálózat elvi felépítését. A legtöbb tanítás egy tetszőleges hibafüggvény minimalizálását jelenti. Egy függvényt minimalizálni pedig kétféle módon tudunk. Ha analitikus módon határozzuk meg egy minimum helyét, akkor gyakran lokális minimumba érünk. A másik megoldás, ha a minimumhelyet valamilyen heurisztika alapján keressük. Ezek az eljárások gyakran globális optimumot adnak [4],[5]. A megoldások bármelyike megszabja a hálózat neuronjainak vagy az egész hálózatnak a felépítését. A MATLAB első neurális hálózatot tanító eljárása a *backpropagation* tanítás, amellyel később részletesebben fogunk foglalkozni. Ennél a tanítóeljárásnál az a lényeg, hogy a hálózat kimenetén keletkező hibát a kimenet felől a bemenet felé áramoltatjuk. Ezt a folyamatot nevezzük hiba visszaterjesztésnek.



4. ábra Többrétegű perceptron a MATLAB Neural Network Toolbox megvalósításában.

A 4. ábrán ez azt jelenti, hogy a hálózat kimenetén meghatározzuk a hibát, majd ezt visszafelé áramoltatjuk a hálózaton a p bemeneti vektorig. Globális optimalizációs eljárások esetén a hálózatnak nem kell garantálnia ezt a fajta viselkedést; a hibát ugyanis nem fogjuk visszafelé áramoltatni a hálózatban.

A többrétegű perceptron tanítása

A 4. ábra egy háromrétegű többrétegű perceptront szemléltet. Ezen hálózat tanulóalgoritmusa hasonló a korábban bemutatott perceptron tanulóalgoritmusához azzal a különbséggel, hogy a kimenet nem garantált, hogy egy $x \in \mathbb{R}$ valós szám, lehet, hogy $x \in \mathbb{R}^n$ valós értékű vektor. Ebben az esetben kell alkalmaznunk a (7) összefüggésben megadott *Mean Squared Error* hibafüggvényt, amelyet az egyszerűség kedvéért most eljelölünk $h_\omega(x)$-szel. A kimeneti rétegre így az alábbi összefüggést kapjuk:

$$h_\omega(x) = \frac{1}{2} * \sum_{i=0}^{m} \left(d_i(n) - y_i(n)\right)^2 . \tag{8}$$

Amely összefüggésben $d_i(n)$ jelöli az adott tanulópontra elvárt kimenetet, míg $y_i(n)$ az adott tanulópontra elvárt tényleges kimenetet jelöli. Továbbá tudjuk, hogy $y_i(n)$-re a (3), illetve (6) összefüggésekből következik, hogy:

$$y_i(n) = \varphi\left(\sum_{j=1}^{m} \omega_{i,j} * x_j + b\right). \tag{9}$$

ahol $\varphi(.)$ jelöli az i-edik neuron aktivációs függvényét. Ekkora a (8) összefüggésbe behelyettesítve a (9) $y_i(n)$-re megadott képletét az alábbi összefüggéshez jutunk:

$$h_\omega(x) = \frac{1}{2} * \sum_{i=0}^{m} \left(d_i(n) - \varphi_i\left(\sum_{j=0}^{m} \omega_{i,j} * x_j\right)\right)^2 . \tag{10}$$

Többrétegű perceptron tanításánál is a feladatunk a hálózat paramétereinek helyes beállítása. Ahogy azt már korábban említettük előrecsatolt hálózati architektúra esetén a hálózat egyetlen paraméterhalmazzal rendelkezik, ami a neuronokat összekötő szinaptikus kapcsolatok súlyaiból álló vektort jelenti. A feladat a hibafüggvény minimalizálása $\omega_{i,j}$ szerint:

$$\min_{\omega_{j,i}} \frac{1}{2} * \sum_{i=0}^{m} \left(d_i(n) - \varphi_i\left(\sum_{j=0}^{m} \omega_{i,j} * x_j\right)\right)^2 . \tag{11}$$

Ahhoz, hogy meg tudjuk határozni, milyen $\omega_{j,i}$ paraméterek szükségesek a hibafüggvény minimumának eléréséhez, be kell látnunk, hogy a hiba visszaterjesztése egy kétlépcsős folyamat [4]. A (10) összefüggés által definiált $h_\omega(x)$ ugyanis csak a kimeneti rétegre lett definiálva. Azt ugyanis, hogy a rejtett rétegekben a neuronoktól milyen kimenetet várunk el nem tudjuk megmondani. A kimeneti rétegben a súly változtatásának mértékét az alábbi összefüggés alapján számolhatjuk ki (Gradiens módszer). A gradiens:

$$\frac{\partial h_\omega(x)}{\partial \omega_{i,j}} = -\big(d_i(n) - y_i(n)\big) * \frac{\partial y_i(n)}{\partial \omega_{i,j}} = -\big(d_i(n) - y_i(n)\big) * \frac{\partial \varphi(In_i)}{\partial \omega_{i,j}} =$$

$$-\big(d_i(n) - y_i(n)\big) * \varphi'(In_i) * \frac{\partial In_i}{\partial \omega_{i,j}} =$$

$$-\big(d_i(n) - y_i(n)\big) * \varphi'(In_i) * \frac{\partial}{\partial \omega_{i,j}} * \left(\sum_k \omega_{i,k} * y_k(n)\right) =$$

$$-\big(d_i(n) - y_i(n)\big) * \varphi'(In_i) * y_j(n)\,.$$

Vagyis rétegenként a súlyok módosítása a fenti levezetés eredményeképpen az alábbi módon adható meg:

$$\omega_{i,j} \leftarrow \omega_{i,j} + \mu * \Big(\big(d_i(n) - y_i(n)\big) * \varphi'(In_i) * y_j(n)\Big) = \omega_{i,j} + \mu * \delta_i * y_j(n). \quad (12)$$

Itt $\mu > 0$ tanulási paraméter. Ez a képlet közvetlenül számolható a kimeneti rétegben. A többi rétegben viszont támaszkodnunk kell az azt következő rétegre.

$\delta_i = \delta_i(n) = -\frac{\partial h_\omega(x)}{\partial In_i}$ a lokális gradiens a már korábban kiszámolt lokális gradiensek segítségével (a láncszabály alapján) számolható ki:

$$\delta_i = \varphi_i'(In_i) * \sum_k \delta_k * \omega_{k,i}$$

az összegzés az i-edik neuron utáni layer összes, k-val jelölt neuronjára kiterjed.

Az általános súlymódosítási formula tehát a következőképpen írható föl:

$$\omega_{i,j} \leftarrow \omega_{i,j} + \mu * \delta_i * y_j(n)$$

Kötegelt tanítás

Kötegelt tanítás esetén a tanulópontokat egyszerre használjuk, azaz minden tanulópont esetén meghatározzuk a hibát, majd ezen hibák összegének átlagát próbáljuk meg csökkenteni, amely hiba a (20) összefüggés alapján számítható. Vegyük észre, hogy ez a folyamat nagyon lassú és számításigényes. Ennek belátására, vezessük be a következő $\Delta\omega_{j,i}$ mennyiséget:

$$\Delta\omega_{j,i}(n) = -\mu * \frac{\partial h_{\omega}(x)}{\partial \omega_{j,i}}. \tag{17}$$

Definiálja $\Delta\omega_{j,i}(n)$ a (17) összefüggés által leírt mennyiséget. Kötegelt tanítás esetén minden tanulóadatra meghatározzuk az alábbi halmazt.

$$\{\Delta\omega_{j,i}(0), \Delta\omega_{j,i}(1), \Delta\omega_{j,i}(2), \Delta\omega_{j,i}(3), \dots, \Delta\omega_{j,i}(n)\} \tag{18}$$

azaz első lépésben minden tanulópontra meghatározzuk a $\Delta\omega_{j,i}(k)$ értékét, majd ezt követően a súlyokat (19) képletnek megfelelően írjuk felül:

$$\Delta\omega_{j,i} = \Delta\omega_{j,i}(0) + \Delta\omega_{j,i}(1) + \Delta\omega_{j,i}(2) + \cdots + \Delta\omega_{j,i}(n)\,. \tag{19}$$

Belátható, hogy az így megadott eljárás lassú és számításigényes. Személyes tapasztalatom az, hogy az ilyen módon tanított hálózatok a tanulóadatokat sem minden esetben tudják megfelelő mértékben közelíteni ellenben a hálózat approximációs képessége még így is megfelelő egyszerűbb osztályozási feladatok megoldásához.

Az eljárást a szakirodalmak szerint célszerű olyan módon inicializálni, hogy a súlyoknak önkényesen adunk egy tetszőleges értéket [4],[10]. A személyes tapasztalataim azt mutatják, hogy a konvergencia jelentősen felgyorsulhat, ha a súlyokat nem önkényesen választjuk meg hanem a [0,1] intervallumból generálunk egy egyenletes eloszlású véletlen számokat tartalmazó vektort; és ezen vektort használjuk a hálózat súlyparamétereiként. Ezt követően ezen súlyparamétereket minden epoch végén módosítani fogjuk egészen addig, amíg a gradiens vektor elég kicsivé nem válik [10], vagy amíg az egy epoch alatt kapott átlagos hiba egy adott korlát alá nem csökken [10].

Szekvenciális tanítás

Az előzőekben bemutatott tanítás hátrányosságát próbálja meg kiküszöbölni. Gyorsabb tanítás és alacsonyabb számítási kapacitás elérése volt a cél, amikor megalkották a szekvenciális tanítás módszerét, amelyet ma is széles körben alkalmazunk a gradiens alapú minimalizáló eljárásokkal párhuzamosan. Ezen tanítási módszer lényege, hogy a tanulópontokat egyenként használva minden tanulópontra meghatározzuk a hálózat hibáját, majd ezen hibát minden lépésben minimalizáljuk. Egyszerűen megfogalmazva a tanulópontok átáramoltatása után a hálózat súlyait módosítani fogjuk. Ezen eljárás esetén fogjuk megadni a klasszikus többrétegű perceptron tanítási algoritmusát, ahogyan azt a MATLAB szoftverben implementálták.

1. lépés: Az adott tanulópontot végigáramoltatjuk a hálózaton, és meghatározzuk a (10) képlet alapján a hálózat mintára generált hibáját.
2. lépés: A hálózat kimeneti rétegében a súlyokat egyből módosíthatjuk a (12) összefüggés felhasználásával.
3. lépés: Minden kimeneti rétegeket megelőző rétegben először meghatározzuk a réteg által generált hiba mértékét (δ_j) a (15) összefüggésnek megfelelően.
4. lépés: Az adott réteg bemeneti súlyait felülírjuk a (16) összefüggés alábbi egyszerűsített változatával: $\omega_{j,i} \leftarrow \omega_{j,i} + \mu * \delta_j * y_i(n)$, ahol $y_i(n)$ az adott rétegbeni i-edik neuron kimenetét jelenti.

Ezen négy lépés véges sokszori alkalmazásával a hálózat által generált átlagos hibát olyan módon csökkentjük, hogy az átlagos hiba összes hibatagját csökkentjük. Az átlagos hibát az alábbi összefüggéssel tudjuk leírni:

$$\varepsilon(n) = \sum_{p} h_{\omega}(p) \tag{20}$$

A következőkben bemutatjuk, hogy milyen módon tudunk létrehozni egy a 4. ábrának megfelelő architektúrájú hálózatot MATLAB-ban, illetve ezt hogyan tudjuk betanítani szekvenciális hiba-visszaterjesztéses tanulóalgoritmussal egy egyszerű osztályozási problémára, a XOR problémára.

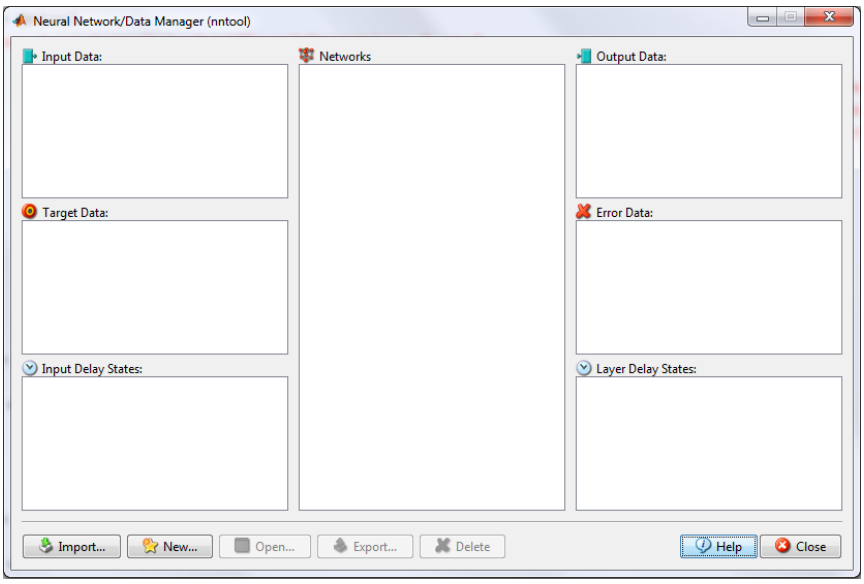
Ezt követően ugyanezen típusú hálózatokhoz megismerkedünk egy olyan tanulóalgoritmussal, amely jelenleg még nem érhető el a MATLAB Neural Network Toolbox

programcsomagjában. Az általam bemutatott eljárás egy genetikus algoritmusból, illetve a nem lineáris Levenberg Marquardt optimalizációs eljárás ötvözéséből alakult ki.

MATLAB példaprogram - XOR probléma

A MATLAB szoftverben kétféle módon tudunk létrehozni neurális hálózatokat. Az első és egyben nehezebb megvalósítási koncepció, hogy a program *Command Window* panelját használva parancsnyelven hozzuk létre azt a hálózati struktúrát, amire éppen szükségünk van. A továbbiakban mi egy egyszerűbb megoldást fogunk előnyben részesíteni. A *Neural Network Toolbox [9]* megjelenésével a MATLAB-ban megjelent egy grafikus eszköz neurális hálózatok kezelése céljából. Az 5. ábra ezt a grafikus felületet szemlélteti. A neurális hálózat/adat kezelő indításához a MATLAB *Command Window* paneljában a következő parancsot kell kiadnunk:

```
>>nntool
```



5. ábra A MATLAB NNTool nevű grafikus Neurális Hálózat kezelő csomagja.

A fönti parancs kiadása után az 5. ábrát fogjuk magunk előtt látni. Tekintsük át, hogy mit is jelentenek számunkra a dolgozat szempontjából érdekes részek. A továbbiakban így az alábbi paneleket fogjuk áttekinteni:

- Input Data – Bemeneti adatok
- Target Data – A hálózat bemeneti adataira elvárt kimeneti adatok
- Networks – A neurális hálózatok
- Output Data – A hálózat szimulációja során kapott aktuális kimeneti értékek
- Error Data – A hálózat hibái

Természetesen lehetőségünk lenne mindezeket a MATLAB parancsnyelvén létrehozni és erre lehetőségünk is van. Amennyiben külső már parancsnyelven létrehozott objektumokat akarunk felhasználni az nntool csomagon belül akkor ezeket az *Import...* gombra való kattintással tudjuk hozzáadni a megfelelő panelekhez. Fontos azonban megjegyezni, hogy az Output Data, illetve Error Data objektumokat nem magunknak kell létrehoznunk, ugyanis ezeket a MATLAB fogja létrehozni. Az előbbit a hálózat szimulációja után, míg az utóbbit a hálózat tanítása során tölti föl a program.

Továbbá másik fontos dolog, hogy amennyiben a hálózatot parancsértelmezőn keresztül hozzuk létre, úgy a következő parancsformát kell használnunk:

```
>>[nev] = newff(PR,[S1 S2 S3 ... Sn],{F1 F2 ... Fn},BTF,BLF,PF);
```

ahol az egyes paraméterek jelentése a következő:

PR – Egy $R \times 2$ –es mátrix, amely az egyes neuronok bemeneti értékeinek minimumát és maximumát jelentik.

Si – Az i-edik réteg neuronjainak száma. Annyi rétegű lesz a hálózat, ahány elemet megadunk ennek a vektornak, és az egyes rétegek pontosan annyi neuront fognak tartalmazni, amennyi az i-edik eleme ezen vektornak.

Fi – Az egyes rétegek aktivációs függvényeit adja meg. Itt fontos megjegyeznünk, hogy a MATLAB szoftverben a neurális hálózatok aktivációs függvényei nem a neuronokban vannak letárolva, hanem rétegszinten. Ennek főként az az oka, hogy minél kevesebb memóriát foglaljon le a hálózatunk. Hasonló hatást elérhetünk azon magas szintű programozási nyelvekben, amelyek képesek kezelni a függvénymutatókat. Így csupán egyetlen helyen tároljuk a függvényt, és ha használnunk kell, csupán e függvény címére hivatkozunk. Ezen paraméterek alapértelmezetten 'tansig' aktivációs függvényt valósítanak meg.

$$tansig(x) = \frac{2}{1 + e^{-2*x}} - 1 \tag{21}$$

Látható, hogy ezen aktivációs függvény nem a [0,1] hanem a [-1,1] intervallumra képez le, ahogyan azt a dolgozat korábbi szakaszában olvashattuk. Az aktivációs függvényeket is a megoldandó problémához kell igazítanunk. A XOR probléma megoldható mind a (21) aktivációs függvény segítségével mind pedig egy olyan aktivációs függvénnyel, amely a [0,1] intervallumra képezi le a paraméterét.

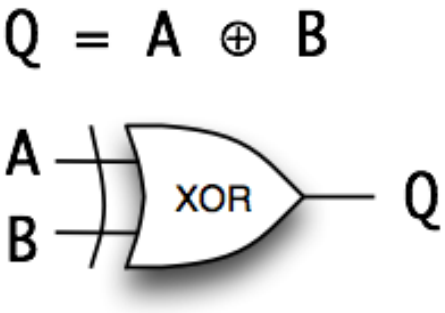
BTF – a backpropagation hálózat tanító algoritmusát adja meg. Ezen eljárás szerint tanítjuk be a hálózatot a tanulóadatok segítségével. Ezen paraméter alapértelmezett értéke ’traingdx’ amely az angol *Gradient descent with momentum and adaptive learning rate backpropagation* megnevezést takarja. Az algoritmus lényege, hogy a legmeredekebb irány mentén indulunk el a hibafüggvény minimalizálása során.

BLF – a backpropagation súly/eltolás-súly tanuló algoritmusát takarja. Azaz, hogy a hálózat milyen algoritmussal fogja megváltoztatni a saját súlyvektorának paramétereit. Ezen tulajdonság alapértelmezett értéke ’learngdm’, amely az angol *Gradiens descent with momentum weight and bias learning* algoritmus rövidítése.

PF – az angol *Performance Function* mozaikszó rövidítése. Azt szablya meg, hogy milyen hibafüggvényt akarunk minimalizálni. Ezzel lényegében azt is megmondjuk a hálózatnak, hogy kötegelt vagy szekvenciális tanítást akarunk végrehajtani. Az alapértelmezett érték ’mse’ amely ugyanazon képlet szerint kerül meghatározásra, amelyre mi a dolgozatban (7) összefüggést alkalmaztuk.

A továbbiakban az nntool [9] alkalmazást fogjuk használni, hogy a hálózatot betanítsuk a XOR problémára. A XOR probléma alatt a logikai XOR műveletet megvalósító neurális hálózatra kell gondolnunk. A 6. ábra a XOR logikai művelet igazságtábláját szemlélteti, jobb oldalon pedig a XOR műveletet megvalósító logikai kaput láthatjuk.

XOR		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



6. ábra A XOR logikai művelet igazságtáblája, illetve az ezt megvalósító logikai kapu sematikus rajza.

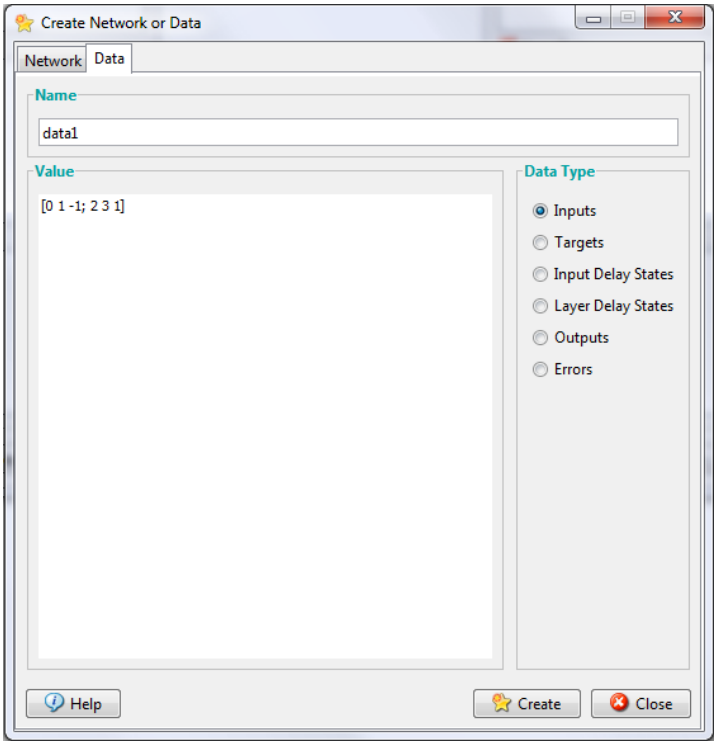
Ezen probléma neurális hálózattal történő megvalósításához a következőket kell észrevennünk. A XOR egy bináris logikai művelet, azaz két bemeneti adatból egy kimeneti adat fog előállni. Az egyszerűség kedvért, illetve mivel kezdetleges algoritmust fogunk használni a hálózat tanítására, egy kétrétegű hálózatot fogunk használni. Mivel két bemenő adatunk van, ezért a hálózat első rétegében két neuront fogunk létrehozni, míg a második rétegben egyet. A tanító adataink a fenti táblázat oszlopai lesznek.

Jelölje a tanítóadatok halmazát T, az elvárt kimenetek halmazát pedig D. Ekkor a két halmaz az alábbi elemeket tartalmazza:

$$T := \{[0\ 0\ 1\ 1], [0\ 1\ 0\ 1]\} \tag{22}$$

$$D := \{[0\ 1\ 1\ 0]\} \tag{23}$$

Láthatjuk, hogy mind T mind pedig D halmazok vektorokat tartalmaznak. Ezeket az alábbi módon fogjuk létrehozni. Az *nntool* panelen kattintsunk a *New...* gombra majd válasszuk ki a *Data* panelt, amelyre kattintva a 7. ábrán látható ablakot fogjuk látni.



7. ábra Adatobjektumok létrehozása varázsló

Ezen panel három fő részből épül föl. A felső vízszintes részben az adat-obkejtumunk nevét adhatjuk meg. Adjunk meg ennek most az *„inputs”* nevet. A jobb oldalon azt választhatjuk ki, hogy milyen típusú adatobjektumot akarunk létrehozni. Alapértelmezésben az *Inputs* van kiválasztva, és mivel bemeneti adatokat, azaz a T halmazt fogjuk megadni, hagyjuk ezen típust bejelölve. A baloldalon láthatunk a *Value* felirat alatt egy példabemenetet. Cseréljük ki ezt a T halmaznak megfelelően a következőre:

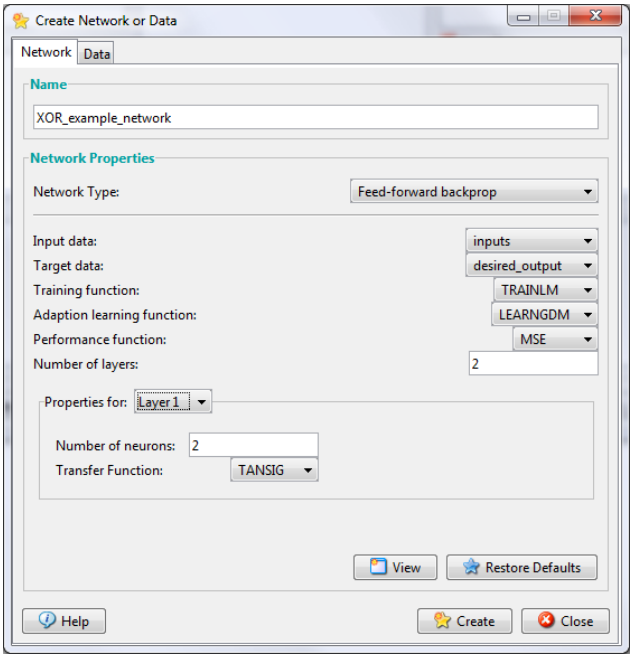
$$[0\ 0\ 1\ 1; 0\ 1\ 0\ 1]\,. \tag{24}$$

Majd kattintsunk a *Create* feliratú gombra. Ezután ugyanezen ablakot fogjuk látni. Ezúttal az elvárt kimeneteinket fogjuk létrehozni, vagyis a D halmazt. A jobb oldali listából válasszuk ki a *Targets* lehetőséget, majd a bal oldali szövegdobozba adjuk meg az alábbi vektort:

$$[0\ 1\ 1\ 0]\,. \tag{25}$$

Névnek pedig gépeljük be az *„desired_output”* azaz elvárt kimenet szöveget. Ha ezzel elkészültünk, a *Data* panelra tovább nincs szükségünk. A következőkben létrehozzuk a neurális hálózat objektumunkat magát, amelyet képesek leszünk betanítani. A hálózat létrehozásához nincs szükség további vektorok létrehozására. A *Network* panel felépítését a 8.

ábra szemlélteti, ahol már beállításra kerültek a szükséges opciók. A 8. ábrának megfelelően állítsuk, be a panelünket, majd kattintsunk a *Create* feliratú gombra, majd zárjuk be ezen ablakot.

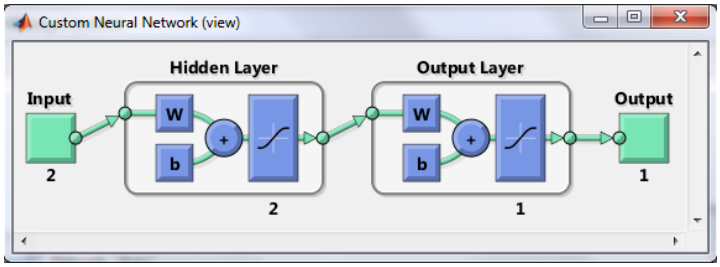


8. ábra Network panel a szükséges beállításokkal.

Számunkra a *Network Properties* érdekes. A hálózat típusa amint látjuk előrecsatolt hiba-visszaterjesztéses hálózat. Ez azért fontos, mert a hibát a kimenettől a bemenet felé visszafelé áramoltatjuk. Az *Input data* értéke az általunk létrehozott *inputs* objektum, amelyet a (22) halmaz alapján a (24) MATLAB vektor ír le. A *Target data* az előzővel ekvivalens módon jött létre a (23) halmaz alapján a (25) vektor-objektum létrehozásával. A választott tanító algoritmus a Levenberg-Marquardt eljárás lesz, 'learngdm' súly illetve eltolás-súly tanuló algoritmus szerint. A hibafüggvényünk pedig a szekvenciális tanításhoz felhasznált MSE (7). A hálózatunk rétegjeinek számát állítsuk kettőre, valamint a neuronok számát is kettőre kell állítanunk. Az aktivációs vagy MATLAB terminológia szerint transzfer függvénynek megfelelő az alapértelmezett tansig, amelyet a (21) összefüggés szerint definiáltak a MATLAB 7-es verziójában.

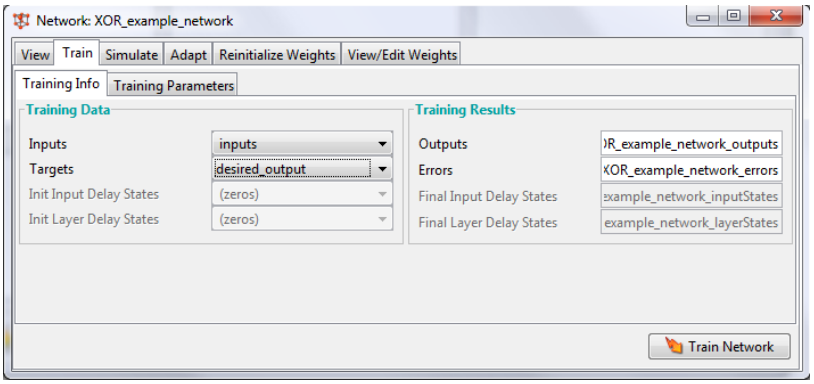
Ha kíváncsiak vagyunk, hogy néz ki vizuálisan a létrehozott hálózatunk azt a *View* gombra kattintva tekinthetjük meg, melynek kimenete, ha minden jól csináltunk a 9. ábrán látott hálózat lesz. Látható, hogy két bemeneti pontja van a hálózatnak, két rejtett neuronnal és egy

kimeneti neuronnal. Az eltolás-súly értéke alapértelmezés szerint nulla, így ezzel a hálózat nem számol a továbbiakban.



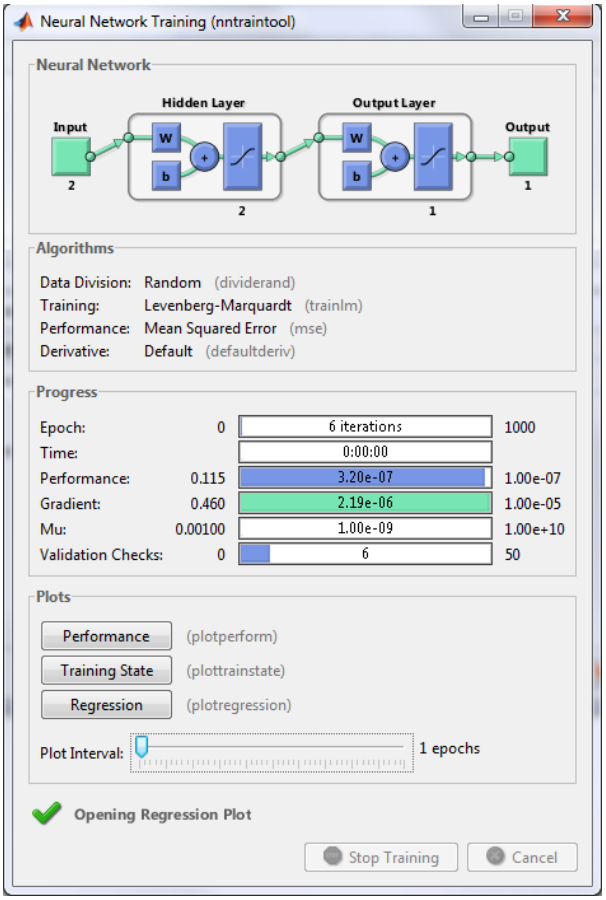
9. ábra A létrehozott előrecsatolt hiba-visszaterjesztéses neurális hálózat sematikus ábrája, ahogyan azt a MATLAB 7 verziójában láthatjuk.

A hálózat tanításához a 5. ábra szerinti *Networks* panelen létrejött *XOR_example_network* nevére kétszer kattintva egy új ablak jelenik meg. Ezen az ablakon válasszuk ki a *Train* panelt, és az *Inputs* valamint *Targets* paraméterek értékeinek adjuk, meg az általunk létrehozott objektumokat majd kattintsunk a *Train Network* gombra.



10. ábra Az elkészült Neurális hálózat tanítási paramétereinek beállításai

Ha rákattintunk, a gombra egy újabb ablak nyílik, meg amely a hálózat tanulásának állását hivatott a felhasználóval közölni. Ezen ablakot a 11. ábra szemlélteti. A továbbiakban áttekintjük ezen ablak főbb összetevőit, illetve azok jelentéseit.



11. ábra Az elkészült neurális hálózat tanulásának fázisait szemléltető panel. A panel felső részén a hálózat sematikus rajzát láthatjuk. Ezt követően a tanító algoritmusra vonatkozó tulajdonságok láthatóak, majd a tanítás jelenlegi állását követhetjük nyomon

Amint azt a 11. ábra magyarázata mutatja a tanulási fázist mutató panel három fő részből tevődik össze. A negyedik részét csak a tanulás befejezte után érhetjük el. Tekintsük most át e részek pontosan mit is mutatnak, és mit jelentenek a mi XOR problémát megoldó hálózatunk szempontjából.

A panel első része a *Neural Network* nevet kapta. Itt az általunk létrehozott hálózat sematikus rajzát láthatjuk. Ezen rajzon fel van tüntetve, hogy hány bemeneti csomópontunk van, rétegenként hány neuront hoztunk létre illetve, hogy hány kimeneti csomóponttal rendelkezik a hálózat. A rétegekben csupán egyetlen neuron elvi felépítését láthatjuk, de ebből is elmondható, hogy milyen módon épülnek fel a neuronok a MATLAB programban.

Neuronok a MATLAB szoftverben

A MATLAB speciális módon kezeli a neuronokat. Minden neuronhoz tárol egy vektor-objektumot, amelyekben a szinaptikus kapcsolatok súlyparaméterei, illetve egy valós paramétert, amely az eltolás-súlyt reprezentálja. Fontos megjegyezni, hogy a MATLAB-ban nem a neuron felelős saját kimenete aktiválásáért, hanem a neuront tartalmazó réteg felelős a benne található összes neuron kimenetének aktiválásáért. Ennek elsősorban helytakarékossági okai vannak. Működését tekintve a MATLAB neuronok kimenetit a (6) összefüggés alapján határozzák meg. A súlyvektorok a neuronokkal együtt tárolt vektorok. Ennek elsősorban az az oka, hogy sok tanítóeljárás köztük az általunk is alkalmazott Levenberg-Marquardt optimalizáció $Ax = b$ alakú lineáris vagy nem-lineáris egyenletrendszerek megoldását végzik el ,ahol x az ismeretlen vektort jelenti. Esetünkben az $Ax = b$ egyenletrendszer x komponense a súlyok alkotta ω vektor.

A 11. ábrán szemléltetett panel második része a tanításhoz felhasznált algoritmusokról ad általános információt. Számunkra két fontos paraméter található ezen a részen:

1. Training: A hálózat hibafüggvényének minimalizálásához felhasznált eljárás neve. Esetünkben ez Levenberg-Marquardt optimalizáció, amelyet a matlab 'trainlm' konstanssal nevez meg. Ez egy nem lineáris kvázi newton módszeren alapul, amely megoldását az alábbi képlet alapján keresi:

$$\Delta\omega \leftarrow [J^T(\omega) * J(\omega) + \mu * I]^{-1} * J^T(\omega)\varepsilon \qquad (26)$$

   Ahol $J(.)$ a jacobi mátrix, I az egységmátrix, $\varepsilon, \mu > 0$ valós számok, a $J^T(\omega) * J(\omega)$ pedig a Hesse mátrix numerikus közelítésére szolgál bizonyos feltételek mellett.

2. Performance: megszabja, hogy milyen hibaszámítási módszert használunk, és ezzel megadja a MATLAB-nak, hogy milyen tanítást hajtson végre a hálózaton. Kötegelt tanításhoz a hálózat létrehozásánál az 'SSE' azaz *'Sum Squared Error'* hibaszámítási módot kell választanunk, míg szekvenciális tanításhoz a választott 'MSE' vagyis *'Mean Squared Error'* a megfelelő választás. A MATLAB rendelkezik egy harmadik típusú hibaszámítási módszerrel, de ezt dolgozatunk során nem fogjuk érinteni. A

későbbiekben egy teljesen más hibaszámítási módszert fogunk látni amit ’CEE’ vagyis *’Cross Entropical Error’*-nak nevezünk.

A harmadik része a panelünknek a leglényegesebb a tanulás folyamán. Az itt lévő lényeges pontokat részletesen bemutatom a következőkben. Előtte azonban tisztáznunk kell egy már korábban említett fogalmat, az *epoch* fogalmát. Egy *epoch* alatt az összes tanulópont hálózaton történő átáramoltatást értjük [10]. A MATLAB az iteráció és az epoch fogalmát összemossa, de vegyük észre, hogy programozói szemszögből ez nem túl szerencsés. Tanácsosabb, ha egy iteráció alatt egy tanulópont hálózaton történő átáramoltatását értjük, míg egy epochot a föntebbi definíció szerint értelmezünk. Ebben a megközelítésben egy epoch nem feltétlenül egy iteráció. A legtöbb tanító eljárás *tanulási-szabálya* epochra vonatkozik, nem pedig iterációra. A tanulási-szabály nem más, mint a tanulási algoritmust terminálo feltétel megnevezése. Ezen fogalmak bevezetése után lássuk mi is a jelentése a számunkra lényeges tulajdonságoknak a panel harmadik szekciójában:

- Epoch: Megmutatja, hogy a tanítási folyamat során hányadszor áramoltatjuk át a tanulóadatok teljes halmazát a hálózaton.
- Time: Azt mutatja meg, hogy a tanulási folyamat kezdete óta mennyi idő telt el.
- Performance: A hálózat átlagos hibafüggvényének értékét mutatja meg. Fontos megjegyezni, hogy ez az érték csak epochonként változik meg!
- Gradient: A gradiens vektor méretét adja vissza. A MATLAB a gradiens vektor hosszát euklideszi norma szerint határozza meg, azaz az alábbi összefüggés alapján:

$$\|v\|_{\infty} = \sum_{i=0}^{n} \sqrt{v_i^2}. \tag{27}$$

Ezek a számunkra lényeges tulajdonságok a harmadik panelen. Korábban már láttuk, hogy a tanítás kétféle módon állhat le. Vagy túl kicsivé válik a gradiens vektor, vagy pedig két egymást követő epoch során a hálózat hibája meghatározott korlát alá csökken. A MATLAB ettől különböző nézetet vall. Ha túl kicsivé válik a gradiens vektor a tanulás terminálni fog, de a hiba alsó korlátot elveti. Helyette inkább egy maximális epoch számot definiál, amit elérve a tanulási folyamat megszakad. Ez nem túl szerencsés hiszen, ha nem körültekintően választjuk

meg a tanuláshoz a maximális epochok számát, akkor lehet, hogy azelőtt megszakítjuk a minimalizálási folyamatot mielőtt egy minimumot elérnénk. Tanácsos nagy maximális epoch számot választani, mert így garantáljuk, hogy a tanulási folyamat konvergenciával terminál és nem áll fönn annak a veszélye, hogy azelőtt terminálna mielőtt a hiba értéke konvergálna.

Ez azonban veszélyes is ugyanis nem tudjuk megmondani, hogy adott adathalmazra kapott hibafüggvény minimumát hány iteráció alatt érjük el. A gradiens vektor méretével való terminálás lényege az, hogy ha túl sok epochot adunk meg a hálózat tanítására és azt amint a fenti ábra is mutatja hat epoch alatt a hálózat képes be tanulni a mintákat, fölösleges várni még 1000-6 epoch végigszámolására. Ezen a ponton ugyanis a gradiens vektorunk, amint azt az ábra is szemlélteti $2.19e-06$ méretűre csökkent, vagy elértünk egy lokális minimumot, innen nem fogunk elmozdulni lényegtelen, hogy hány epoch van még hátra.

Ezzel a XOR problémát megoldó MATLAB példaprogramunk végére értünk. A következő részben egy alternatív hibrid tanító algoritmus kerül bemutatásra, amely igen széles körben alkalmazható a napjainkban felmerülő problémák bármelyikére.

Előrecsatolt hálózatok tanítása: A DE-LM algoritmus

A következőkben nem osztályozási feladatot, hanem nem-lineáris függvény approximációs feladatot fogunk megoldani *Differential Evolution* (DE) és *Levenberg Marquardt* (LM) algoritmusok egy kombinált változatával [11]. Ahhoz, hogy tárgyalni tudjuk e kombinált algoritmus működését, először meg kell ismerkednünk az algoritmust alkotó két optimalizációs technikával.

Differential Evolution

A következőkben a *Differential Evolution* megnevezés helyett a *DE* rövidítést fogjuk használni. Az eljárás a magyar szakirodalomban sajnos nem kapott megfelelő fordítást ám mivel működése nagyban hasonlít a szimulált hűtéséhez és mivel *Genetic Annealing* néven is gyakran hivatkozzák magyar megfelelője akár a Genetikus Hűtés is lehetne. Maga az

algoritmus, ahogyan a neve is mutatja egy genetikus algoritmus ezen algoritmuscsalád minden előnyével illetve hátrányával.

Ezen algoritmus alkalmazható n-változós függvények minimumhelyének megtalálására egy fix előre definiált méretű keresési térben [11]. Hasonlóan a részecskerendszer módszerhez [4][5] itt is populáció alapú keresésről van szó. A populáció egyedei jelen esetben a minimalizálandó függvény értelmezési tartományának elemeiként vannak definiálva.

$P \in R_f$, ahol R_f az $f(x)$ függvény értelmezési tartományának halmaza.

Legyen adott $f(x_1, x_2, \dots, x_n)$ függvény melynek keressük $x_1^*, x_2^*, x_3^*, \dots, x_n^*$ helyen értelmezett f^* helyettesítési értékét melyre teljesül, hogy

$$\forall i \; f^*(x_1^*, x_2^*, \dots, x_n^*) \leq f\left(x_1^i, x_2^i, \dots, x_n^i\right) \tag{28}$$

ahol az $[x_1, x_2, \dots, x_n]$ n-dimenziós vektort a populáció egyedeinek nevezzük, míg a teljes populáció egy $\mathbb{R}^{m\,x\,n}$mátrix, amely a következőképpen van definiálva:

$$P \coloneqq \begin{bmatrix} x_1^1 & \cdots & x_n^1 \\ \vdots & \ddots & \vdots \\ x_1^m & \cdots & x_n^m \end{bmatrix} \in \mathbb{R}^{m\,x\,n}. \tag{29}$$

Ekkor $[x_1^1, x_2^1, \dots, x_n^1]$ jelöli a teljes populáció első egyedét, míg $[x_1^m, x_2^m, \dots, x_n^m]$ a teljes populáció m-edik egyedét reprezentálja. Ezen egyedek közvetett módon rendelkeznek egy rátermettségi mutatóval, amely nem más, mint az f függvény helyettesítési értéke az adott egyed által reprezentált pontban.

A DE eljárásban definiált az egyedek között egy speciális egyed amelyet G_p-vel jelölünk. Ezen egyed rendelkezik a legjobb rátermettségi mutatóval a populáció összes egyede közül.

$$\min\{ f(p_i) \}$$

ahol p_i a P mátrix i-edik sora által értelmezett egyedet jelenti.

Az eljárás során minden egyedhez a mutációs operátor segítségével egy próbavektor kerül generálásra. A mutációs operátor két paraméterrel rendelkezik, amely két paraméter a G_p, illetve az aktuális egyed vektora. Ezen próbavektort a (30) összefüggés alapján határozzuk meg.

$$v_{i,G+1} := x_{r1,G} + F * \left(x_{r2,G} - x_{r3,G}\right) \tag{30}$$

Ahol r_1, r_2 illetve r_3 egymástól különböző egyedek P mátrixbeli sorindexeit jelölik, valamint G az aktuális generáció indexe. Generáció alatt ezen eljárásra vonatkoztatva azt értjük, hogy a P mátrix egyedei maximálisan hányszor kerültek megváltoztatásra. Az F paraméter egy tetszőleges valós szám, amelyet a felhasználó ad meg.

Továbbá az eljárás generál egy véletlen számot (jelölje ezt a számot $Rand_j$), majd ezen generált véletlen számot összehasonlítva egy a felhasználó által megadott véletlen számmal *(CR)* eldönti, hogy az aktuális egyedvektor a (G+1)-edik generációban a (30) képlettel meghatározott próbavektor legyen –e; vagy maradjon az eredeti.

$$x_{j,G+1} := \begin{cases} v_{j,G+1} \; ha \; Rand_j \; \leq CR \\ x_{j,G+1} \; ha \; Rand_j > CR \end{cases} \tag{31}$$

Vegyük észre, hogy a CR paraméter az eljárás futása során nem változik meg, azt a felhasználó maga definiálja az eljárás hívásakor. Ezzel ellentétben $Rand_j$ minden próbavektor generálása után újra meghatározásra kerül. A próbavektor ilyen módú alkalmazásának köszönhetően kiküszöbölhető a túl korai konvergencia, amely nagyban növeli egy lokális minimumban történő megrekedést, ugyanakkor az eljárás teljes mértékben érzéketlen arra, hogy a próbavektor által definiált egyed rátermettségi mutatója (azaz az f függvény helyettesítési értéke a próbavektor által definiált helyen) alacsonyabb –e, mint az előző. Azonban fontos megjegyeznünk, hogy a DE algoritmus futása során egyáltalán nem garantált a konvergencia; azaz, az algoritmus rendelkezik a genetikus algoritmusok bizonytalan viselkedési jellemzőivel is.

Formálisan az algoritmus az alábbi elven működik: Legyen $f: E^n \rightarrow \mathbb{R}$ a rátermettségi mutatót megadó függvény, amelynek egyetlen argumentuma a populáció egy egyede, amelyet egy n elemű valós számokat tartalmazó vektor szemléltet. Az f függvény gradienstere ismeretlen; a cél egy olyan m egyed előálltása amelyre telesül, hogy $f(m) \leq f(p)$ a keresési tér minden p elemére. Ez azt jelenti, hogy az m egyed az f függvény globális minimumának közelében van.

A neurális hálózatok genetikus algoritmussal való tanítása nem napjaink kérdése. Az első ilyen alkalmazásról Haykin [4] könyvében is olvashatunk ahol *rekurrens* hálózat súlyvektorának beállítására használják a genetikus algoritmusokat. A következő algoritmus, amelyet bemutatok a *Levenberg Marquardt* féle nemlineáris optimalizációs eljárás.

A Levenberg-Marquardt algoritmus

A Levebmerg-Marquardt algoritmus talán az egyik legszélesebb körben használatos optimalizációs technika. Sokkal hatékonyabb, mint az egyszerű gradient descent algoritmusok, és mint a legtöbb konjugált gradiens módszer, számos probléma tekintetében. A probléma, amelyre az LM algoritmus megoldást nyújt a Nemlineáris legkisebb négyzetek módszere. Ez formálisan a (32) képlet által definiált függvény minimalizáció végrehajtását jelenti.

$$F(x) = \frac{1}{2} * \sum_{j=1}^{m} \big(f_i(x)\big)^2 \rightarrow MIN \tag{32}$$

A Levenberg-Marquardt algoritmus egy olyan iterációs eljárás, amely egy β vektor által definiált irány mentén keres minimumot [16]. Első lépésben akár a gradient descent alapú algoritmusok esetében meg kell határoznunk egy induló β vektort amelyet az eljárás $\beta + \delta$-ra próbál javtani, az alábbi módon:

$$f_i(\beta + \delta) \approx f(\beta) + J_f(\delta) \tag{33}$$

ahol J_f az f függvény Jacobi mátrixa. Továbbá ismeretes az, hogy a minimalizálandó F függvénynek adott X pontban akkor van minimuma, ha $\nabla_\delta F(X) = 0$. Innen meghatározható δ-ra az alábbi összefüggés:

$$\Big(J_{f_i}^T(X_i) * J_{f_i}(X_i)\Big) * \delta = -J_{f_i}^T(X_i) * f_i(X_i) \tag{34}$$

Amely összefüggésből δ-t $\Big(J_{f_i}^T(X_i) * J_{f_i}(X_i)\Big)$ mátrix invertálásával kaphatjuk meg. Az eljárás további sajátossága, hogy a a fenti mátrixot felskálázza valamilyen λ vektorral.

$$\big(J_{f_i}^T(X_i) * J_{f_i}(X_i) + \lambda_i * I\big)\delta_i = -J^T * F(X_i) \tag{35}$$

ahol *I* az egységmátrix.

A fentieknek megfelelően valamint mivel az eljárás iterációs jellegű a feladat az X vektor finomtása a (35) képlet megoldásával. Ezt az alábbi formula szerint fogjuk megtenni:

$$x_{i+1} = x_i - \big(\nabla^2 F(x_i)\big)^{-1} * \nabla F(x_i) \tag{36}$$

ahol $\nabla^2 F(X_i)$ az $F(.)$ függvény másodrendű parciális deriváltjai által alkotott Hesse mátrix; amelyet az eljárás a $\left(J_{f_i}^T(X_i) * J_{f_i}(X_i)\right)$ mátrixal közelít. Ezen közelítés alkalmazhatóságához egyéb speciális feltételeknek kell teljesülnie amelyektől most eltekintünk.

A korábban bemutatott MATLAB neurális hálózatot ezen algoritmust felhasználva tanítottuk be. Természetesen az elmúlt években ezen algoritmusnak is születtek variánsai, javításai. Az eredeti algoritmust Marquardt mutatta be, amelyben egy θ paraméter irányította a keresést. Az algoritmus számos javítása közül a Powell féle úgynevezett *Dog Leg [12]* azaz kutyaláb algoritmus terjedt el hasonló mértékben, mint maga az LM optimalizációs eljárás.

A következőkben egy alternatív tanulóalgoritmus kerül bemutatásra. Ezen algoritmus az előzőekben bemutatott DE, illetve LM algoritmusok kombinációjából jött létre. A DE algoritmusa a korábban már említett módon keresi az ideális súlyvektorokat, amelyekre teljesül, hogy csökkentik a hálózat végén megjelenő átlagos hibát. Az LM algoritmusa a Hessian mátrixot közelítve mind az elsőrendű, mind pedig a másodrendű parciális deriváltakat felhasználva közelíti egy (32) alakú függvény minimumhelyét. A továbbiakban a (7) pontban definiált *Mean Sqared Error*-t fogjuk alkalmazni és e függvényt fogjuk minimalizálni.

A DE-LM algoritmus

Egy előrecsatolt neurális hálózat kimenete a hálózat szinaptikus kapcsolatai erősségének és a hálózat bemenetének függvénye, vagyis $y = f(x, \omega)$. Adott kimeneti érték esetén a $(d - y)^2$ eltérést a hálózat hibájának nevezzük. A hibát ezúttal E-vel fogjuk jelölni. Vegyük észre, hogy a hálózat hibája egy kétváltozós függvény, amely a hálózat elvárt kimenetétől és az aktuális kimenettől függ. A hálózat hibája tehát $E(d, f(x, \omega))$. A (7) pontban definiált *Mean Squared Error* definícióját lecseréljük a következőre:

$$E = \frac{1}{N} * \sum_{k=1}^{N} \left(d_k - f\left(x_k, \omega_{k,j}\right)\right)^2 \tag{37}$$

Itt N a tanulóadatok halmazának számosságát, vagyis a tanulóadatok számát jelenti. A tanítás célja az E függvény minimalizálása, a hálózat szinaptikus kapcsolatai erősségének

optimalizálásával ($\omega = [\omega_1, \omega_2, \omega_3, \dots, \omega_N]$). A minimalizálást a következő eljárás hajtja végre:

1. lépés: A populáció egyedeinek inicializálása. Generáljunk egyenletes eloszlású véletlen számokat a [0,1] intervallumból pontosan olyan dimenziószámmal amennyi súly szerepel a hálózatban.

   $P_G = \left[\omega_{1,G}, \omega_{2,G}, \omega_{3,G}, \dots, \omega_{N,G}\right]^T, G = 1,2,3, \dots, G_{MAX}$ - G_{MAX} a DE algoritmus által maximálisan elérhető Generációt jelenti.

   $\omega_{i,G} = \left[\omega_{1,i,G}, \omega_{2,i,G}, \omega_{3,i,G}, \dots, \omega_{D,i,G}\right], i = 1,2,3, \dots, N$ - N a hálózat súlyparamétereinek száma.

2. lépés: Minden egyedre határozzuk meg E értékét a (37) összefüggés alapján.
3. lépés: Minden egyedvektorhoz válasszunk ki véletlenszerűen $r_1, r_2, r_3 \in \{1,2,3, \dots, N\}$ indexeket úgy, hogy az aktuális egyedvektor, illetve r_1, r_2, r_3 különbözőek legyenek!
4. lépés: Alkalmazzuk a DE-nél megadott mutációs operátort a (30) összefüggésnek megfelelően a harmadik lépésben kiválasztott r_1, r_2, r_3, egyedvektorokra így létrehozva egy (G+1)-edik generációs próbavektort.
5. lépés: A (31) összefüggésnek megfelelően keresztezzük a populáció vektorait a negyedik lépésben létrehozott próbavektorokkal. A (31) összefüggésben szereplő CR konstans egy [0,1] intervallumba eső egyenletes eloszlású véletlen szám legyen. Az itt létrehozott vektorokat jelölje $x_{i,G+1}$.
6. lépés: A következő lépés a szelekció. Ezt nem adtuk meg a DE algoritmusnál, mert probléma specifikus függvényről van szó, azaz problémához kell igazítani. A szelekció lényege, hogy új populációt hozzon létre attól függően, hogy milyen rátermettségi mutatóval rendelkeznek az egyes egyedek. A szelekciós függvény a mi esetünkben az alábbi módon adható meg:

$$\omega_{i,G+1} := \begin{cases} x_{i,G+1}, & ha\ E\left(d, f\left(x, \omega_{i,G+1}\right)\right) \leq E\left(d, f\left(x, \omega_{i,G}\right)\right) \\ \omega_{i,G}, & egyébként \end{cases}$$

   Ahol az $x_{i,G+1}$ az 5. lépésben létrehozott vektorokat jelenti. Ez lényegében a DE algoritmus leírásánál megadott P populációs mátrix egy sorvektora.

7. lépés: Egy adott K iteráció után kapott ω vektorral inicializáljuk az LM algoritmus súlymátrixát. Ezen súlymátrixhoz már ismerjük E értékét, mivel a DE algoritmus meghatározta azt.
8. lépés: Határozzuk meg $J(\omega)$ Jacobi mátrixot.
9. lépés: Határozzuk meg $\Delta\omega$ értékét az alábbi egyenlőség kiszámításával:

$$\Delta\omega = [J^T(\omega) * J(\omega) + \mu * I]^{-1} * J^T(\omega) * E$$

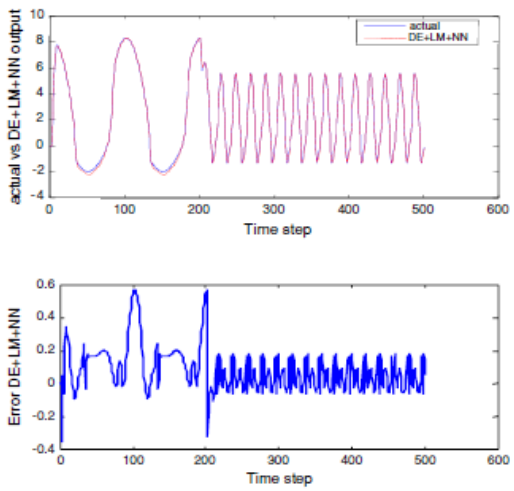
Ezen számítási mód hasonlít az L-M algoritmus közelítési képletéhez [12].

10. lépés: Számoljuk újra E értékét $(\omega + \Delta\omega)$ mellett. Ha az új E értéke kisebb mint a 7. lépésben meghatározott E értéke csökkentsük μ-t és térjünk vissza az 1. lépéshez.
11. Az algoritmus akkor konvergál, ha gradiens vektor egy tetszőlegesen választott normája (legyen p-norma) $\|\nabla E\|_p = \|J^T(\omega) * d - f(x, \omega)\|$ egy adott érték alá csökken, vagy a hálózat átlagos hibája csökken egy adott korlát alá.

$$\|\nabla E\|_p \leq \gamma$$

ahol $\gamma > 0$ valós szám jelöli ezt a korlátot.

Ezen algoritmus hatásfokát tekintve jelentősen növeli az előrecsatolt neurális hálózatok approximáló képességét a függvény approximációs feladatok megoldásában. A 12. ábrán egy DE-LM algoritmussal tanított hálózatkimenete, illetve annak hiba-grafikonja látható. Ezen tesztadatokat a [12] cikkben közölték. A cél egy nemlineáris rendszer közelítése volt.



12. ábra A DE-LM algoritmussal tanított előrecsatolt hálózat függvény approximációs képessége illetve hibájának alakulása[12]

Ezen nemlineáris rendszert a következő összefüggés írja le:

$$y_p(k+1) = \frac{y_p(k) * [y_p(k-1)+2] * [y_p(k)+2.5]}{8.5 + [y_p(k)]^2 + [y_p(k-1)]^2} + u(k)$$

Ezen modellt közelítették a szerzők, Bidyadhar Subudhi és Debashisha Jena a 2008-ban közzétett publikációjukban a DE-LM algoritmussal tanított előrecsatolt neurális hálózattal. Amint látható az ezen eljárással tanított háló approximáló képessége jelentősen megnőtt az általánosan alkalmazott algoritmusokhoz képest. Kutatásaim szerint ezen algoritmus, hasonló hatékonysággal alkalmazható idősor előrejelzésre és az osztályozási feladatok egy részére is. Az általam vizsgált problémák, időjárás előrejelzési kísérletek voltak, amelyhez az adatokat Metnet.hu időjárási portálról tudtam XML formátumban beszerezni illetve a tényleges eredmények is ugyanilyen formában csak későbbi időpontban kerültek lementésre. A másik terület ahol ezen algoritmus hatékonyságát vizsgáltam, az az egyes pénznemek árfolyam változásait próbálta meg előre jelezni.

2. Fejezet

RBF Hálózatok

Az alapvetően rosszul felállított problémák megoldására Tyihonov a regularizációt javasolta. A regularizáció szerint ezen problémákra is adható helyes válasz. Ezt úgy érhetjük el, hogy nem negatív segédfüggvényekkel próbáljuk meg stabilizálni a megoldást. Regularizáció egy olyan $F: \mathbb{R}^m \to \mathbb{R}$ alakú leképezés keresését jelenti, amely jól illeszkedik az (x_i, d_i) adatainkra azaz $F(x_i) \approx di,\ i = 1,2,..,N$. Ekkor az illesztés hibája az alábbi összefüggés által adható meg:

$$\varepsilon(F) = \frac{1}{2} * \sum_{i=1}^{N} \big(d_i - F(x_i)\big)^2 + \lambda * \frac{1}{2} * \|DF\|^2. \tag{38}$$

Ezen összefüggésben szereplő tagok a következőket jelentik:

- $\frac{1}{2} * \sum_{i=1}^{N} \big(d_i - F(x_i)\big)^2$ jelenti a sztenderd hibát. Ezen hibát kell minimalizálnunk a hálózat tanítása folyamán. Jelölje $\varepsilon_s(F)$ ezen hiba értékét.
- $\frac{1}{2} * \|DF\|^2$ tagot, regularizációs tagnak nevezzük. Ezen regularizációs tagban D egy Hilbert-téren értelmezett lineáris operátor, amely a problémáról előzetes információt tartalmaz. Ezen lineáris operátor feladata a megoldás stabilizálása. Szokásos elnevezés még a b*üntető függvény* is. Jelöljük el a hibafüggvény ezen tagját $\varepsilon_c(F) - szel.$

Tehát az (39) összefüggés által leírt képlet az alábbi formában is megadható:

$$\varepsilon(F) = \varepsilon_s(F) + \lambda * \varepsilon_c(F), \tag{39}$$

amelyet minimalizálnunk kell, és ezen függvény minimumhelyét $F_\lambda(x)$-szel jelöltük. Ekkor λ-t regularizációs paraméternek nevezzük, amely egy pozitív valós szám. Két fontos tulajdonsággal rendelkezik:

1. Ha λ kicsi, akkor az adatok a meghatározóak.
2. Ha λ nagy, akkor az előzetes információk lesznek meghatározóak.

A feladat tehát a (39) összefüggésben szereplő $\varepsilon(F)$ függvény minimumhelyének kiszámolása, melyre az alábbi összefüggés adódik:

$$F_\lambda(x) = \sum_{i=1}^{N} \omega_i * G(x, x_i)\,. \tag{40}$$

Ezen képletből az látszik, hogy a minimumproblémára konkrét pontos matematikai megoldás adható az RBF hálózatok esetén. Az összefüggésben szereplő $G(x, x_i)$-t Green-függvénynek nevezzük. A fenti $F_\lambda(x)$ −re kapott kifejezés speciális esetben átírható az alábbi formára:

$$F_\lambda(x) = \sum_{i=1}^{N} \omega_i * G(\|x - x_i\|). \tag{41}$$

Ezen kifejezés egyetlen ismeretlen paraméterét, ω_i −t az alábbi módon fejezhetjük ki:

$$\omega_i = \frac{1}{\lambda} * \left(d_i - F_\lambda(x_i)\right). \tag{42}$$

Green-függvénynek pedig választhatjuk a többdimenziós normális eloszlás sűrűségfüggvény tetszőleges skalárszorosát: $\mathcal{N}_{m0}(x_i, \sigma_i^2 * \mathrm{I})$. Az ilyen módon megadott függvényt, az I-hez tartozó Green-függvénynek nevezzük.

$$G(x, x_i) = \exp\left\{-\frac{1}{2 * \sigma_i^2} * \|x - x_i\|^2\right\}. \tag{43}$$

Az ilyen módon létrehozott regularizációs hálózatokat RBF (*Radial Basis Function*) hálózatoknak nevezzük. Ezen hálózatok problémája az, hogy túl sok neuron van a rejtett rétegben. Pontosan annyi ahány tanulópontunk van. Ennek belátására tekintsük a következőket. Ismeretes, hogy [4]

$$F_\lambda(x) = \sum_{i=1}^{N} \omega_i * G(x, x_i), \tag{44}$$

$$\omega = [\omega_1, \omega_2, \dots, \omega_N]^T = (G + \lambda * I)^{-1} * d, \tag{45}$$

ahol d a d_i mennyiségekből alkotott vektor, G pedig a Green-függvény segítségével adott, NxN-es mátrix:

$$G = \begin{bmatrix} G(x_1, x_1) & \cdots & G(x_1, x_N) \\ \vdots & \ddots & \vdots \\ G(x_N, x_1) & \cdots & G(x_N, x_N) \end{bmatrix}.$$

Látható, hogy G egy kvadratikus mátrix, azaz a hálózat ω paramétereit egy $(G + \lambda * I) * \omega = d$ lineáris egyenletrendszer megoldásával kapjuk. Könnyen belátható, hogy ilyen esetben a ω paramétervektor kiszámítása rendkívül műveletigényes. Tetszőleges $M \in \mathbb{R}^{NxN}$ mátrix inverzének meghatározásához N^3 műveletvégzés szükséges. Ez a probléma a már korábban vázolt regularizációs hálózat rejtett rétegbeli neuronjainak számából fakad. N tanulópont esetén a rejtett rétegben N darab neuron lesz. Ezen probléma megoldására a hálózatot redukáljuk, amennyire csak lehet olyan módon, hogy az a problémára még elfogadható választ adjon. Az ilyen módon kapott regularizációs hálózatot általánosított RBF hálózatnak nevezzük. A továbbiakban megismerkedünk az általánosított RBF hálózattal, illetve annak tanítási módszerével. Ezt követően egy egyszerű függvény approximációs feladat kerül megoldásra, ezúttal a MATLAB parancsnyelvét felhasználva.

Általánosított RBF hálózatok

A korábban említett RBF hálózatok problémája, hogy a hálózat súlyparamétereinek meghatározásához egy NxN-es mátrix inverzére volna szükségünk, amelyet csak N^3 művelet elvégezésével tudnánk előállítani. Ennek megoldására a korábban bemutatott RBF hálózat rejtett neuronjainak számát redukálni fogjuk, amennyire csak lehet olyan módon, hogy a hálózat aktuális problémára adott megoldása még elfogadható legyen. Ezeket a hálózatokat nevezzük általánosított RBF hálózatoknak. Jelen helyzetben a korábban (39) kifejezés által definiált közelítés hibáját az alábbi módon írhatjuk föl:

$$\varepsilon(F^*) = \frac{1}{2} * \sum_{i=1}^{N} \left(d_i - \sum_{j=1}^{m-1} \omega_j * G(x_i, t_j) \right)^2 + \frac{1}{2} * \lambda * \|DF^*\|^2. \qquad (46)$$

Látható, hogy az előzőekben N darab rejtett rétegbeli neuronok számát m-1 darabra redukáljuk. Ezen közelítési hiba tagjai az előzőekben tárgyaltakkal ekvivalens módon

értelmezhetőek, vagyis a cél jelen helyzetben is $\varepsilon(F^*)$ minimalizálása, amelyet $F_\lambda(x)$ helyett $F^*(x)$ −el fogunk jelölni és az alábbi módon definiáljuk:

$$F^*(x) = \sum_{j=1}^{m-1} \omega_j * G(x, t_j) \tag{47}$$

Mind (46) mind pedig (47) összefüggésekben új t_k ismeretlen is szerepel. Ezt a t_k mennyiséget középpontnak nevezzük. Egy általánosított RBF feladat megoldása az alábbi mennyiségek ismeretét feltételezi:

- $\{(x_1, d_1), (x_2, d_2), \dots, (x_N, d_N)\}$ – A tanulópontok nem üres véges halmaza.
- $\{t_1, t_2, \dots, t_{m-1}\}$ – A középpontok nem üres véges halmaza.
- $G(\circ,\circ)$ – Green-függvény, amelyet ez esetben is választhatunk a (43) összefüggésben definiált függvénynek.

Középpontok

Tetszőleges t_i középponton a $G(\circ, t_i)$ Green-függvény középpontját értjük. Általánosított RBF hálózatok esetén ezen t_i paraméterek is befolyásolják a tanulást a súlyparamétereken kívül. A középpontok meghatározására három módszer létezik:

1. A középpontok véletlenszerű kiválasztása
2. A középpontok önszerveződő kiválasztása
3. A középpontok felügyelt kiválasztása

A középpontok ilyen módon történő kiválasztásai megszabják a hálózat tanulási stratégiáját. Tekintsük most át a fenti három módszer által meghatározott tanulási stratégiákat egyenként. Ezt követően megadjuk az RBF hálózatok tanulási algoritmusát, majd konkrét függvény approximációs feladatot oldunk meg MATLAB-ban, de ezúttal nem a grafikus *nntool* eszközt fogjuk használni, hanem a parancsértelmezős felületet.

Véletlenszerűen választott, de aztán rögzített középpontok

Green-függvényként használjuk a (43) –beli $G(\|x - t_i\|^2)$ függvényt. A

$\{X_1, X_2, X_3, X_4, \dots, X_N\}$ tanulópontok közül válasszuk véletlenszerűen a $\{t_1, t_2, t_3, t_4, \dots, t_{m1}\}$ pontokat. Legyen $d_{MAX} \in \mathbb{R}$ a t_i pontok közötti legnagyobb távolság.

Legyen konkrétan $G(\|x - t_i\|^2)$ az alábbi:

$G(\|x - t_i\|^2) = \exp\left\{-\frac{m1}{d_{MAX}^2} * \|x - t_i\|^2\right\}$ Ez nem más, mint $\mathcal{N}_{m_0}\left(t, \frac{d_{MAX}^2}{2*m_1}\right)$ sűrűségfüggvényének számszorosa.

Ha speciálisan $\lambda = 0$, akkor

$$\varepsilon(F^*) = \frac{1}{2} * \|d - G * \omega\|^2 .$$

Ezen $\varepsilon(F^*)$ minimumát keressük, melynek megoldását $(G^T G)\omega = G^T d$ normálegyenlet megoldásával kaphatjuk meg.

Látható, hogy a véletlenszerűen kiválasztott középpontok esetén a feladat megoldása egy numerikus minimalizációs feladat megoldását jelenti. Ez a módszer rendkívül könnyen számolható, de számos feladat esetében igen nagymértékben ronthatja az RBF hálózat approximáló képességét. Ezen gyöngeség kiküszöbölésére, de az alacsony számításigény megtartására egyéb a gépi tanulásban használatos eljárásokat szokás használni. Az ilyen módon elvégzett kiválasztási módszert nevezik önszerveződő kiválasztásnak. Leggyakrabban a k-közép (KNN) algoritmust szokták alkalmazni ezen módszer esetén, mert könnyen implementálható és igen gyors, más hasonló algoritmusokhoz képest.

A középpontok önszerveződő kiválasztása

Ez egy összetett folyamat, amelyet két egymástól jól elkülönülő részre bonthatjuk:

1. A rejtett rétegbeli pontok kiválasztása – A középpontokat a klaszterezés k-közép módszerével választjuk ki. A klaszterezés olyan speciális osztályozást jelent, amely

osztályozásnál az osztályokat is mi magunk alakítjuk ki. Ezen eljárás lépései a következők:

1.1. $t_1(0), t_2(0), \dots, t_{m-1}(0)$: Kezdeti középpontok véletlenszerű kiválasztása olyan módon, hogy minden $t_i(0)$ egymástól különböző legyek.

1.2. Az n-edik lépésben a hálózat bementén az X_n tanulópont jelenik meg.

1.3. k(x) = annak a $t_k(n)$ –nek az indexe amely X_n –hez a legközelebb van. Azaz minimális távolságra. A távolságot euklideszi norma alapján számolhatjuk.

1.4. Az új klaszterközéppont kiválasztása: $t_{k(x)}(n+1) = t_{k(x)}(n) + \eta * \left(x(n) - t_{k(x)}(n)\right)$.

1.5. $n \leftarrow n+1$ és térjünk vissza 1.2 pontra, amíg a $t_{k(x)}$ középpontnál változás tapasztalható. Ha ezen változás egy adott érték alá csökken állítsuk le az eljárást.

2. A kimeneti rétegbeli súlyok meghatározása.

Az fent közölt eljárás tehát egy klaszterező algoritmust használ fel a középpontok kiválasztására. Minden n-edik lépésben csak egyetlen $t_{k(x)}$ kerül megváltoztatásra. Az eljárás egyetlen paramétere az $\eta \in (0,1)$ egyenletes eloszlású véletlen szám. Fontos megjegyezni, hogy a tanulópontokat nem feltétlenül csak egyszer áramoltathatjuk át a hálózaton, de akár többször is. Ekkor a fenti eljárás kiegészíthető egy feltételvizsgálattal, amely csak akkor cseréli le a már meglévő klaszterközéppontot, ha az jobb $\varepsilon(F^*)$ értéket eredményez. Ezen feltételvizsgálat azonban azt feltételezi, hogy minden iterációban meghatározzuk $\varepsilon(F^*)$ értékét, amely lineárisan növeli az eljárás számításigényét.

A középpontok felügyelt kiválasztása

Az általánosított regularizációs hálózat hibafüggvényének minimum helyére legyen adott a következő összefüggés:

$$F^*(x) = \sum_{i=1}^{m-1} \omega_i * G\left(\|x - t_i\|_{C_i}\right)$$

amely összefüggésben ω_i, t_i, c_i ismeretlen paraméterek. Ekkor a hálózat hibáját az alábbi módon definiálhatjuk:

$$\mathcal{E} = \frac{1}{2} * \sum_{j=1}^{m-1} \left(d_j - F^*(x_j)\right)^2$$

Az így definiált $\mathcal{E}$ hibát kell minimalizálnunk ω,t és K szerint, ahol K-t az alábbi módon kaptuk:

$$\|C_i * (x - x_i)\|^2 = (x - t_i)^T * C_i^T * C_i * (x - t_i) = (x - t_i)^T * K^{-1} * (x - t_i)$$

A minimalizálás gradiens módszer szerint vezetjük le, de alkalmazható rá a korábban bemutatott DE algoritmus is. A későbbiekben bemutatunk egy másik populáció alapú minimumkereső eljárást, amelyet alkalmazhatunk az $\mathcal{E}$ függvény minimumának megkeresésére, a *Particle Swarm Optimization* (PSO) [4][5] algoritmust.

Az $\mathcal{E}$ függvény minimalizálását tehát ω_i, t_i, K paraméterek szerint kell elvégezni. Gradiens módszerrel való megoldás szerint ehhez három parciális deriváltat kell meghatároznunk, amelyek rendre a következők:

1) $\frac{\partial \mathcal{E}(n)}{\partial \omega_i(n)}$ 2) $\frac{\partial \mathcal{E}(n)}{\partial t_i(n)}$ 3) $\frac{\partial \mathcal{E}(n)}{\partial K_i^{-1}(n)}$

Végezzük ez ezen parciális deriválásokat:

Végezzük el az 1) parciális deriválást. Vegyük észre, hogy ezen deriválás eredménye az $\mathcal{E}$ függvény ω szerinti minimalizálását jelenti tehát ezen parciális deriválás eredménye a regularizációs hálózat súlyvektorának módosításáért felel.

$$\omega_i(n+1) = \omega_i(n) - \eta_1 \frac{\partial \mathcal{E}(n)}{\partial \omega_i(n)},$$

$$\frac{\partial \mathcal{E}(n)}{\partial \omega_i(n)} = -\sum_{j=1}^{N} \left(d_j - F^*(x_j)\right) * G\left(\left\|x_j - t_i(n)\right\|_{C_i}\right).$$

A 2) parciális derivált a középpontok megtalálásáért felelős. Ezen derivált az adott $\mathcal{E}$ függvényt t_i paraméter szerint minimalizálja, vagyis a hálózat által használt középpontokat módosítja. Vegyük észre, hogy a középpontok önszerveződő kiválasztásánál ezért volt felelős a k-közép algoritmus, amely klaszterezéssel oldotta meg ezen problémát.

$$t_i(n+1) = t_i(n) - \eta_2 \frac{\partial \mathcal{E}(n)}{\partial t_i(n)},$$

$$\frac{\partial \mathcal{E}(n)}{\partial t_i(n)} = \sum_{j=1}^{N} \left(d_j - F^*(x_j)\right) * \omega_i(n) * G'\left(\left\|x_j - t_i(n)\right\|_{C_i}\right) * 2 * K_i^{-1}\left(x_j(n) - t_i(n)\right).$$

A 3) és egyben utolsó parciális derivált az $\mathcal{E}$ hibafüggvény K^{-1} szerinti deriváltját jelenti, vagyis ezen paraméter szerint minimalizálja $\mathcal{E}$ −t. Ezen derivált értéke a K_i^{-1} értékét módosítja iterációról iterációra.

$$K_i^{-1}(n+1) = K_i^{-1}(n) - \eta_3 \frac{\partial \mathcal{E}(n)}{\partial K_i^{-1}(n)},$$

$$\frac{\partial \mathcal{E}(n)}{\partial K_i^{-1}(n)} = -\sum_{j=1}^{N} \left(d_j - F^*(x_j)\right) * \omega_i(n) * G'\left(\left\|x_j - t_i(n)\right\|_{C_i(n)}\right) * \left(x_j - t_i(n)\right) * \left(x_j - t_i(n)\right)^T.$$

Ezen három parciális deriváltakra kapott képlet felhasználásával $\mathcal{E}$ −t minimalizálhatjuk az adott három ismeretlen paraméter szerint. A következőkben tekintsük át a Particle Swarm Optimization algoritmusát, amely egy hatékony középpont kiválasztó algoritmus lehet alacsony számításigény mellett.

Particle Swarm Optimization

A PSO algoritmust 1995-ben Kennedy és Eberhart [4] mutatták be. Az algoritmus ötletét egy speciális megfigyelés, adta amely azt vizsgálta, hogy miként keres egy madárraj egy élelemforrást zárt térben. A két tudós a madarak megfigyelése után fejlesztette ki a *Particle Swarm Optimization* algoritmusát, amelynek az idők folyamán számos módosítása és javítása született. A továbbiakban a klasszikus PSO algoritmus kerül bemutatásra.

A PSO egy sztochasztikus populáció alapú optimalizáló eljárás, amely alkalmas folytonos és diszkrét függvények minimumhelyének keresésére. Az eljárásban részecskék mozognak adott szabályok szerint a keresési térben és ezen részecskék adott pozíciója reprezentálja a

minimalizálandó függvény helyettesítési értékét a részecske helyén. Jelölje $\mathcal{p}$ a részecskék halmazát a következőféleképpen:

$$\mathcal{p} = \{\mathcal{p}_1, \mathcal{p}_2, \mathcal{p}_3, \dots, \mathcal{p}_k\}$$

Az eljárás célja egy $f: \Theta \to \mathbb{R}\ (\Theta \subseteq \mathbb{R})$ adott függvény minimumhelyének megkeresése, amelyet az alábbi módon formalizálhatunk:

$$\Theta^* = \underset{\vec{\Theta} \in \Theta}{\arg min}\, f(\vec{\Theta}) = \{\vec{\Theta^*} \in \Theta : \mathrm{f}(\vec{\Theta^*}) \leq f(\vec{\Theta}), \forall \vec{\Theta} \in \Theta\}$$

ahol $\vec{\Theta}$ egy k-dimenziós vektor, amely a keresési tér eleme. Keresési tér alatt az $f(.)$ függvény értelmezési tartományának azon részhalmazát értjük, amely részhalmazban a keresés zajlik.

A keresőrészecskék két alapvető tulajdonsággal rendelkeznek, amely tulajdonságok a részecskék mozgását írják le a térben. Ennek megfelelően minden részecske rendelkezik egy aktuális pozícióval, amelyet a továbbiakban $\vec{x_i}(t)$ −vel, illetve egy sebességvektorral $\vec{v_i}(t)$ −vel jelölünk. Az algoritmus minden iterációban minden részecske ezen két tulajdonságát módosítja az alábbi szabályok szerint:

$$\vec{v_i}(t+1) = \omega * \vec{v_i}(t) + \varphi_1 * \vec{U_1}(t) * \left(\vec{b_i}(t) - \vec{x_i}(t)\right) + \varphi_2 * \vec{U_2}(t) * \left(\vec{l_i}(t) - \vec{x_i}(t)\right), \quad (48)$$

$$\vec{x_i}(t+1) = \vec{x_i}(t) + \vec{v_i}(t+1)\,. \quad (49)$$

Ezen összefüggések módosítják minden részecske sebességét a (48) illetve aktuális pozícióját az (49) összefüggés alapján. Ezen algoritmus elsősorban azért tudott igen nagy mértékben elterjedni, mert ezen két művelet iterációnkénti elvégzését igényli. Egyetlen hátránya a megállási feltétel meghatározása. Megállási feltételként a legtöbb szakirodalomban egy maximális iteráció számhoz kötik az algoritmus futását. Amennyiben az aktuális iteráció eléri ezen maximálisan megszabott iteráció számot [4], az algoritmus leáll. Ezen megközelítésnek van egy erős hátránya, mégpedig az, hogy ezen maximális iteráció szám elérése esetén az algoritmus részecskéi közötti távolság nem feltétlenül kicsi. Ez azt jelenti, hogy a részecskék a keresési térben még elszórtan helyezkednek el.

Ezen algoritmussal kapcsolatos kutatásaim azt mutatták, hogy amennyiben a PSO algoritmust a korábbiakban említett RBF hálózat középpontjainak kiválasztására használjuk föl, megadhatunk egy alternatív megállási feltételt az alábbiak szerint:

Értelmezzük két részecske közötti távolságot a következő p-normának megfelelően:

$$D(\mathcal{p}_i, \mathcal{p}_k) = \sqrt[p]{\sum_{l=1}^{k} (|p_{il}| - |p_{kl}|)^p} .$$

Valamint értelmezzük a teljes populáció minden egyede közötti összegzett távolságot ezen p-normák összegeként:

$$\Gamma_k = \sum_{i=1}^{\|\mathcal{p}\|} \sum_{j=1}^{\|\mathcal{p}\|} D(\mathcal{p}_i, \mathcal{p}_j).$$

Ekkor az algoritmust olyan módon kell módosítanunk, hogy minden iteráció végén határozza meg ezen Γ_k függvény értékét, és álljon meg a keresés ha két egymást követő iteráció során Γ_k illetve Γ_{k+1} abszolút értékes eltérése egy adott $\vartheta \in \mathbb{R}$ alá csökken. Ekkor az algoritmus megállási feltételét az alábbi módon formalizálhatjuk:

$$(|\Gamma_k - \Gamma_k + 1| \leq \vartheta) \supset STOP$$

A következőkben foglaljuk össze a klasszikus *Particle Swarm Optimization* algoritmus lépéseit. Magát az eljárást két részre fogjuk bontani [5]. Egy inicializáló, illetve egy fő ciklus részre. Az inicializációs rész feladata a populáció létrehozása. A fő ciklus feladata a populáció egyedeinek módosítása a (48), (49) összefüggések alapján.

PSO algoritmus lépései

Az algoritmus megadását kezdjük az inicializációs blokk feladatainak megadásával. Az eljárás ezen része felelős a populáció egyedeinek kezdeti tulajdonságainak beállításáért. A dolgok egyszerűsítése miatt jelölje Θ' a keresési teret.

1. lépés: A populáció egyedeinek $\mathcal{p}_i$ aktuális pozícióját állítsuk be egy k-dimenziós véletlen számokat tartalmazó vektorra. ($\overrightarrow{x_i} \in \Theta'$).
2. lépés: Minden $\mathcal{p}_i$ részecskének $\overrightarrow{v_i}$ tulajdonságát állítsuk nullára. $\forall_i$: $\overrightarrow{v_i} = 0$.
3. lépés: $\overrightarrow{b_i} = \overrightarrow{x_i}$, ahol $\overrightarrow{b_i}$ a populáció legrátermettebb részecskéje.

Ezen lépéseket az eljárás indítása előtt célszerű végrehajtani. Fontos megjegyezni, hogy amennyiben több egymástól független keresést hajtunk végre, ezen lépések végrehajtása nélkülözhetetlen. Másképpen fogalmazva, az inicializáló lépéseket pontosan annyiszor kell végrehajtani, ahány egymástól független keresést hajtunk végre. Ezt követően a részecskerendszerünk készen áll az adott $f(.)$ függvény szélsőértékének keresésére. A keresés ismét egy kétlépcsős folyamat. Az első lépésben minden részecske sebességét és pozícióját megváltoztatjuk. A második lépésben a populáció legrátermettebb egyedét választjuk ki. Ezen részecske szerepe a klasszikus PSO algoritmusban a megoldás megtalálása. Egy adott m iterációs lépés elérése után ezen legrátermettebb egyed tartalmazza a megoldást. Az algoritmus így egy lokális szélsőérték-kereső eljárássá redukálódik. Ha a részecskék közötti összesített távolság alapján terminálju az algoritmust, ezen részecske köré csoportosul a többi részecske.

1. lépés: $\overrightarrow{l_i}(t) = \arg\min_{\overrightarrow{b_j}(t)} f\left(\overrightarrow{b_j}(t)\right)$, amely tetszőleges $\wp_i$ részecske szomszédos részecskéi által az eddig elért legjobb pozíció.
2. lépés: Generáljunk $\overrightarrow{U_1}$ illetve $\overrightarrow{U_2}$ mátrixokat úgy, hogy $\overrightarrow{U_i} \in \mathbb{R}^{\|p\| x \|p\|}$ diagonális mátrix melynek főátlóját [0,1) intervallumból generált véletlen számok alkotják.
3. lépés: Módosítsuk $\overrightarrow{v_i}(t)$ −t a (48) összefüggésnek megfelelően.
4. lépés: Módosítsuk $\overrightarrow{x_i}(t)$ −t az (49) összefüggésnek megfelelően.

A fenti négy lépés felelős a populáció egyedei tulajdonságának módosításáért, a (48) és (49) összefüggések felhasználásával. A klasszikus PSO algoritmus tehát a $\overrightarrow{b_i}$ részecskét adja vissza megoldásként. Ezt az alábbi módon teszi:

1. lépés: Menjünk végig a teljes populáción, a fenti négy lépés befejezte után hajtsuk végre az alábbi műveletet:

$$\left(f\left(\overrightarrow{x_j}(t)\right) \leq f\left(\overrightarrow{b_j}(t)\right)\right) \supset \overrightarrow{b_j}(t) = \overrightarrow{x_j}(t)$$

Ezen lépések véges sokszori végrehajtását követően, egy egyszerű feltételvizsgálattal állíthatjuk meg az eljárást. A bemutatott klasszikus PSO értelmében ez azt jelenti, hogy amennyiben az aktuális iteráció szám elérte az általunk megadott felső korlátot, az algoritmust leállítjuk és az eredményt $\overrightarrow{b_j}(t)$ −ben találjuk.

A következőkben egy konkrét függvény approximációs feladatot fogunk megoldani MATLAB segítségével, az RBF hálózatok kétféle középpont-kiválasztási stratégiájával, valamint a most bemutatott PSO algoritmussal.

MATLAB: Függvény approximáció RBF hálózatokkal

A kitűzött cél egy adott $f(.)$ függvény approximációja. Az approximáció szó jelentése közelítés. Az approximáció ismeretlen mennyiségek közelítő pontossággal való meghatározására szolgáló eljárás. Tudjuk, hogy az RBF hálózat a nemlineáris legkisebb négyzetek módszerének megoldását állítja elő. Azon speciális esetben mikor a G mátrix NxN-es, vagyis nem általánosított hálózatról van szó, ez az egyszerű lineáris legkisebb négyzetek feladatának megoldását jelenti.

Definíció: A legkisebb négyzetek módszere az adott függvényosztályból azt az f függvényt határozza meg, amelyre függvényértékek különbségeinek négyzetösszege, vagyis a

$$\sum_{k=1}^{N} \left(f_k - f(x_k)\right)^2$$

kifejezés értéke minimális.

Tekintsük a következő három függvényt, amelyeket majd RBF hálózattal szeretnénk közelíteni. A közelítést a [-4,4] zárt intervallumon végezzük el. A mintavételezéshez, az eredeti közelítendő $f(.)$ függvény értelmezési tartományából válasszunk fix lépésközzel diszkrét valós értékeket 0.5 , 0.3 , majd végül 0.1 lépésközökkel. Az így kapott vektor minden eleméhez határozzuk meg ezen függvények helyettesítési értékét egy újabb vektorban, majd ezen adatokat felhasználva tanítsunk be egy RBF hálózatot a MATLAB Neural Network Toolbox [9] programcsomagját felhasználva. A feladat az alábbi három függvény közelítése:

1. $f_1(x) = \sin(x) + cos^2(x)$
2. $f_2(x) = \sin(5 * x) + x$
3. $f_3(x) = 2 * x^2 - \sin(x^2) + \frac{4*x}{2}$

Ezen feladatok megvalósításához tudnunk kell ezen függvények helyettesítési értékét adott pontokban. Ehhez az alábbi MATLAB parancsot fogjuk felhasználni:

```
>> <objektumnév> = [<kezdőérték> : <lépésköz> : <végérték>];
```

A parancs szintaxisa nagyon egyszerű. Szemantikáját tekintve egy (<végérték> - <kezdőérték>) / 2 elemű vektorobjektumot hoz létre, mely vektor elemei <kezdőérték>-től lépésköznyi differenciával <végértékig> követik egymást. Továbbá tudnunk kell RBF hálózatot létrehozni. Ezt az alábbi paranccsal tehetjük meg:

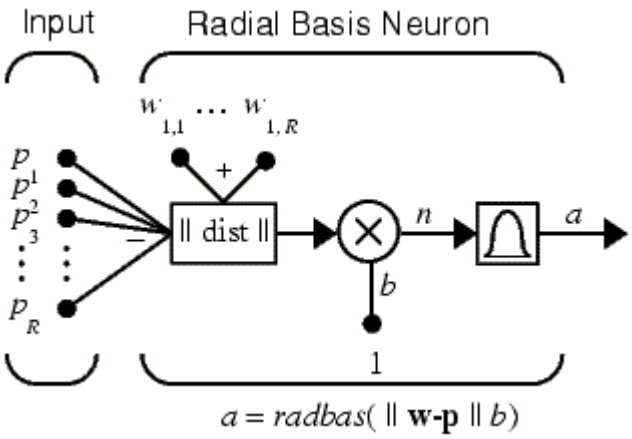
```
>> <hálózatnév> = newrb(P, T, goal, spread, MN, DF);
```

Tekintsük meg, mit is jelentenek a $newrb(.)$ függvény argumentumai. Könnyen észrevehető, hogy az első két argumentum ugyanaz, mint a $newff(.)$ függvény esetén voltak. Egyetlen megszorítás ezen két argumentumra, hogy a be és kimeneti vektorok dimenziószámának azonosnak kell lennie. A specifikáció szerint:

$$P \in \mathbb{R}^{RxQ} \text{ és } T \in \mathbb{R}^{SxQ}$$

vagyis P és T mátrixok, amely mátrix sorvektorainak dimenziója megegyezik, de a két mátrix sorvektorainak száma különbözhet. A spread paraméter alapértelmezett értéke 1.0, és feladata a közelítés simaságának meghatározása. Ez azt jelenti, hogy minél jobban növeljük a spread paramétert a függvényközelítés annál finomabb lesz, de egyúttal egy nagy spread paraméter eredménye az a mellékhatás is, hogy igen sok neuronra lesz szükség a függvény közelítéséhez. Azonban a spread paraméter túl alacsony értéke a hálózat általánosító képességének romlásához vezethet. Látható, hogy amíg MLP esetén a hálózat neuronjainak számát nehéz megmondani, addig RBF hálózatok esetén a spread paraméter értéke mutat ehhez hasonló viselkedést. A megoldás erre a problémára az, ha a $newrb(.)$ függvényt több spread paraméterrel is kipróbáljuk.

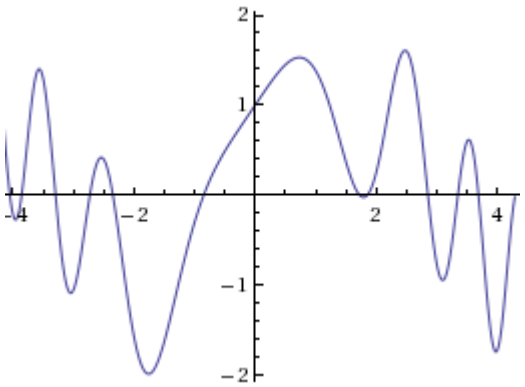
Számunkra a jelenlegi feladat megoldásához a $newrb(.)$ függvény ezen paramétereinek ismerete szükséges. A továbbiakban tekintsük meg, hogy hogyan is néz ki egy RBF hálózat MATLAB vizualizációban. A 13. ábra egy RBF neuront szemléltet annak bemeneti vektorával és belső felépítésével.



13. ábra Általánosított RBF hálózat MATLAB Neural Network Toolbox programcsomagjában.

Látható, hogy a MATLAB RBF [9] neuronjának implementációja az általánosított RBF hálózati modellt alkalmazza. Ezt onnan lehet észrevenni, hogy egy neuronnak egy R elemű vektor a bemenete (R,1) kapcsolat szerint, amely nem megfelelő a lineáris RBF hálózat (R,R) kapcsolati viszonyának.

MATLAB segítségével az RBF hálózat létrehozásakor a hálózat neuronjainak száma automatikusan meghatározódik és a tanulási fázis rögtön kezdetét veszi, nincs szükség explicit parancs kiadására. A továbbiakban az előzőekben megadott három függvény approximációja a feladat. Erre a célra három különböző módon tanítjuk be az RBF hálónkat. Az első függvényt a középpontok önszerveződő kiválasztásával és azon belül a KNN algoritmussal betanított RBF hálózattal fogjuk közelíteni. Ezen függvény ábrázolását a 14. ábra szemlélteti.



14. ábra Első közelítendő függvény - F1(x)

A feladat által megszabott mintavételezést a függvény értelmezési tartományának egy részhalmazán kell elvégezni 0.5-ös lépésközzel. Ezt az alábbi MATLAB parancs kiadásával tehetjük meg:

```
>> x = [-4.0 : 0.5 : +4.0];
>> f1x = sin(x) + cos(x.^2);
```

A következő dolgunk az x vektor által meghatározott pontokban az $f_1(x)$ függvény helyettesítési értékének meghatározása, amelyet a második parancs kiadásával kaphatunk meg. Ezzel eljutottunk arra a pontra, amikor rendelkezésünkre áll a $newrb(.)$ függvény első két argumentuma. A következő argumentum, amit meg kell adnunk a hiba értéke. A hálózat tanítása akkor fog véget érni, ha a hálózat által generált hiba értéke ezen goal paraméter értéke alá csökken. Ezen paraméter értékét nullának választjuk. A következő paraméter a már korábban tisztázott spread, amely értéke ezen függvény approximációjához a jelenlegi módszerrel maradhat alapértelmezett. A kiadandó parancs tehát a következő:

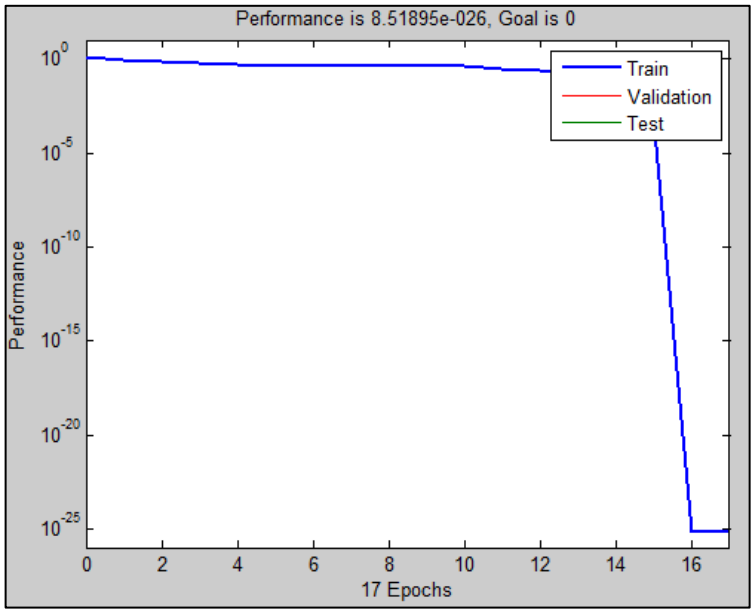
```
>> net_rb = newrb(x, f1x);
```

Ezen parancs kiadására létrejön az RBF hálózatunkat tároló objektum és megkezdődik a tanítás folyamata. A 15. ábrán a hálózat átlagos hibájának alakulását láthatjuk. Látható, hogy a hálózat hibája 10^{-25} −es nagyságrendűre csökkent alig 17 epoch alatt.

Ebből az látható, hogy a középpontok önszerveződő kiválasztásával kapott tanulási stratégia nagyon gyors, és a hálózat átlagos hibájának értéke is nullához közeli. Jelen helyzetben e tanulási stratégia által kapott általánosítási képességet akarunk tesztelni. Tehát a tanulás ugyan effektívnek nevezhető, de a cél most nem a legkisebb hiba elérése, hanem a 14. ábrán megadott függvény minél pontosabb közelítése.

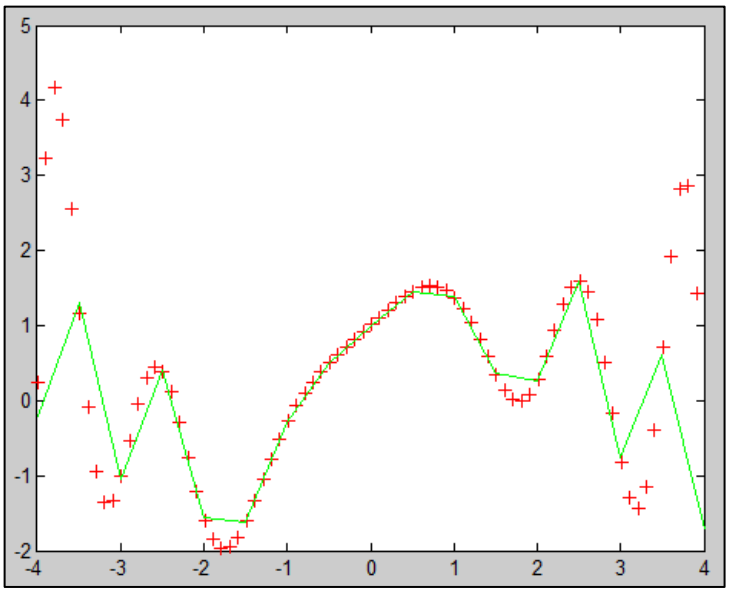
A továbbiakban azt kell megnéznünk, hogy a hálózat, hogyan tudja visszarajzolni nekünk az $f_1(x)$ függvényt, amelyet az előzőekben az alábbi módon definiáltunk:

$$f_1(x) = \sin(x) + cos^2(x)$$



15. ábra Az RBF hálózat átlagos hibájának változása F1(x) közelítendő függvény esetén.

Ahhoz, hogy a hálózat által generált függvény grafikonját megjeleníthessük, szimulálnunk kell a hálózatot. A szimulációt, mivel a hálózat approximációs és általánosító képességére vagyunk kíváncsiak, az x intervallum felére adjuk meg, de a hálózat kimenetét a teljes x intervallumon jelenítjük meg. Ezen kimenetet a 16. ábra szemlélteti. Látható, hogy a hálózat, approximációs képessége igen gyenge.



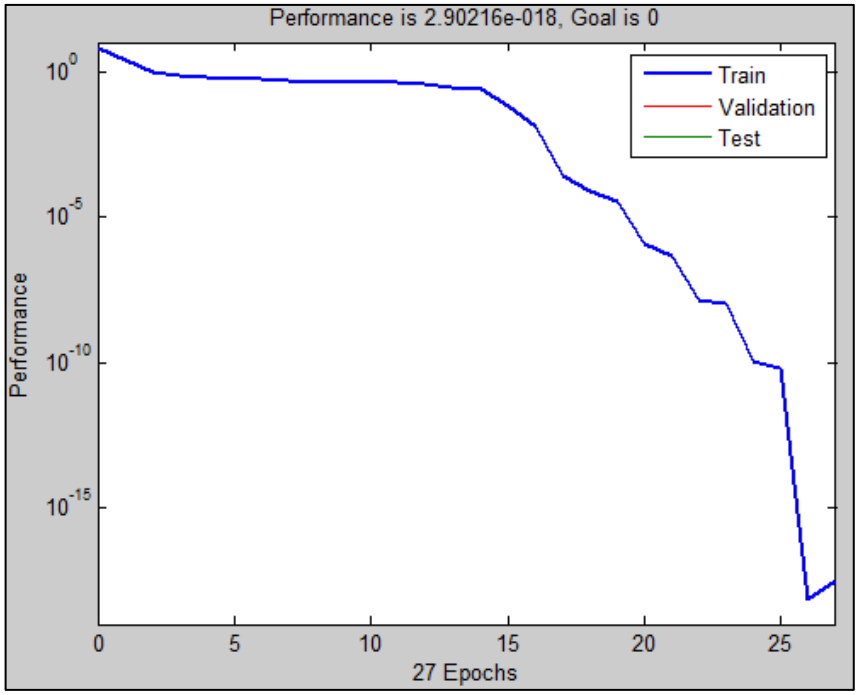
16. ábra Az RBF hálózat tanításának eredmény F1(x) függvényre, a középpontok önszerveződő kiválasztásával KNN algoritmus szerint.

A hálózat körülbelül azon az intervallumon közelíti jól $f_1(x)$ −et amely intervallumra szimuláltuk a hálót a tényleges $f_1(x_i)$ helyettesítési értékkel. Ezt követően a hálózat, a betanult adatok alapján próbát meg általánosítani a közelített adatok alapján. Az ábrán a piros pontok jelölik a közelítendő függvényt, míg zöld folyamatos vonallal került kirajzolásra az RBF hálózat által generált függvény képe. Látható, hogy ugyan a hálózat hibája nagyon alacsony, a hálózat általánosító képessége ezen tanulási módszer alkalmazásával nagyban függ attól, hogy a hálózatot milyen adatokra szimuláljuk alkalmazása előtt. Ezen módszer tehát nem a legalkalmasabb függvényközelítési feladatokra. Osztályozási feladatoknál e módszer sokkal jobban teljesít, ha a $newrb(.)$ függvény spread paraméterét körültekintően választjuk meg.

Tekintsük a következő példát, azaz $f_2(x)$ −et amelyet az alábbi módon definiáltunk:

$$f_2(x) = \sin(5 * x) + x$$

Ezen feladat esetében tanulási stratégiának a középpontok felügyelt kiválasztását fogjuk alkalmazni. Az ilyen módon tanított hálózat globális hibájának változását a 17. ábra szemlélteti.

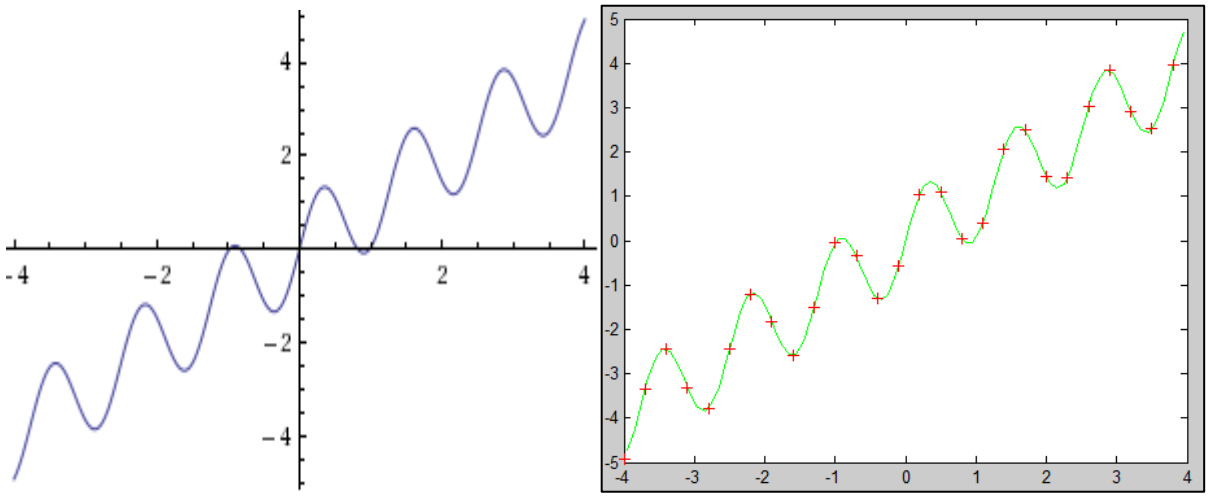


17. ábra Az RBF hálózat globális hibájának alakulása F2(x) közelítendő függvény esetén.

Látható, hogy ezen módszer lassabban konvergál, és a hiba értéke is csupán 10^{-15} –es nagyságrendbe csökkent le. Ebből következtethetnénk arra, hogy ezen tanítási módszer rosszabb, mint a középpontok önszerveződő kiválasztása, de tartsuk szem előtt, hogy a hálózat általánosítását vizsgáljuk.

A módszer ugyanaz lesz, amit az előző példánál alkalmaztunk. A hálózatot szimuláljuk a $[-2,+2]$ intervallumra a pontos $f_2(x_i)$ helyettesítési értékekre, illetve a $[-4,-2] \cup [+2,+4]$ intervallumon a hálózat által betanult közelítésre. Ezen közelítés grafikonját szemlélteti a 18. ábra. Látható, hogy a közelítés 0.75 lépésközzel is megfelelő a $[-4,-2] \cup [+2,+4]$ intervallumokon is, szemben az önszerveződő kiválasztással, amely esetén 0.1-es lépésközt használtunk a közelítés szimulációjára.

Látható tehát, hogy a középpontok felügyelt kiválasztása a hálózat globális hibáját ugyan nem csökkenti olyan mértékben, mint a KNN algoritmussal történő önszerveződő kiválasztás, de jelentősen növeli a hálózat approximációs képességét még akkor is, ha a hálózat szimulációjához sokkal nagyobb lépésközt használunk, mint annak tanításához.



18. ábra Bal oldalon a közelítendő függvény, jobb oldalon pedig a betanított RBF hálózat általánosítási képessége 0.75 lépésközzel F2(x) függvényre.

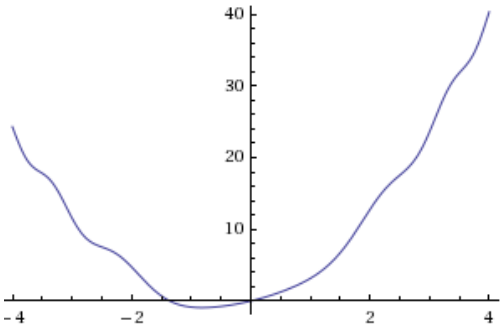
Az ilyen módú tanulási stratégia tehát rendkívül jól általánosít approximációs feladatok megoldásában, ám a tapasztalat azt mutatja, hogy osztályozási feladatokra a módszer sokkal lassabb, mint a középpontok önszerveződő kiválasztásával kapott tanulási stratégia, és a hálózat általánosítása sem annyival jobb, hogy megérje ezt a módszert alkalmazni.

Láthatjuk tehát, hogy mindkét módszernek megvannak az előnyei illetve hátrányai. Amíg a középpontok felügyelt kiválasztása rendkívül jól általánosít approximációs feladatokon, addig osztályozási feladatok megoldásánál nem éri meg alkalmazni a középpontok önszerveződő kiválasztásával szemben. Míg az utóbbi approximációs feladatoknál igen rosszul képes általánosítani, de osztályozási feladatok elvégzésében sokkal jobban alkalmazható. A következőkben bemutatjuk a csomópontok önszerveződő kiválasztásán alapuló tanulási stratégiát PSO algoritmussal alkalmazva. A PSO algoritmus megállási feltétele nem a klasszikus algoritmusban leírt maximális iteráció szám elérése, hanem a részecskék közötti teljes távolságra vonatkozott.

A hálózatot az alábbi módon definiált $f_3(x)$ közelítésének céljából hoztuk létre:

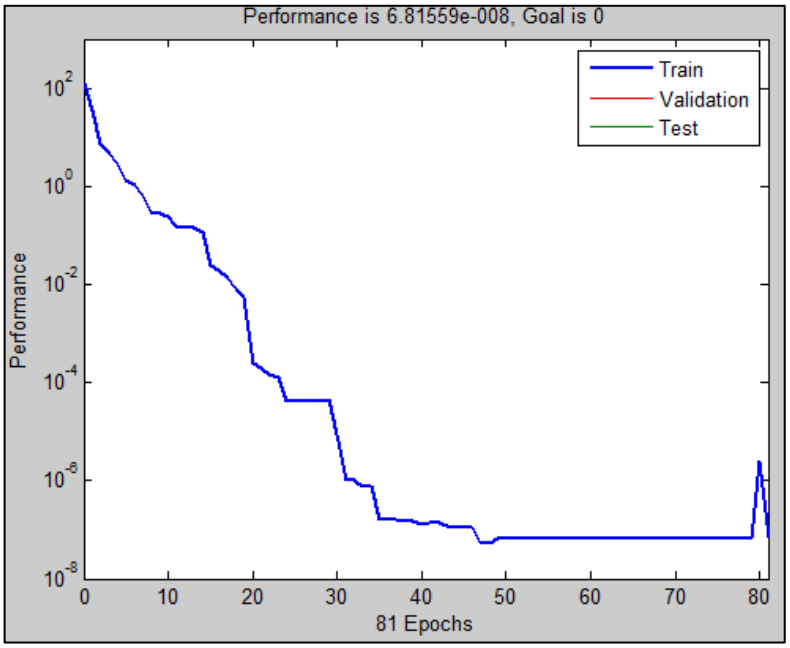
$$f_3(x) = 2 * x^2 - \sin(x^2) + \frac{4 * x}{2}$$

Ezen eljáráshoz felhasználtuk a MATLAB *Particle Swarm Optimization Toolbox* által elérhető PSO algoritmust. A függvényt a $[-4, +4]$ intervallumon közelítettük, 1.025-ös spread paraméter alkalmazása mellett. A közelítendő függvény grafikonját a 19. ábrán láthatjuk.



19. ábra A közelítendő F3(x) függvény grafikonja.

A hálózat tanítási stratégiája tehát a középpontok önszerveződő kiválasztásán alapul, de a korábbi KNN algoritmust lecseréltük PSO algoritmusra. Az ilyen módon tanított hálózat globális átlagos hibájának alakulását a 20. ábra szemlélteti.



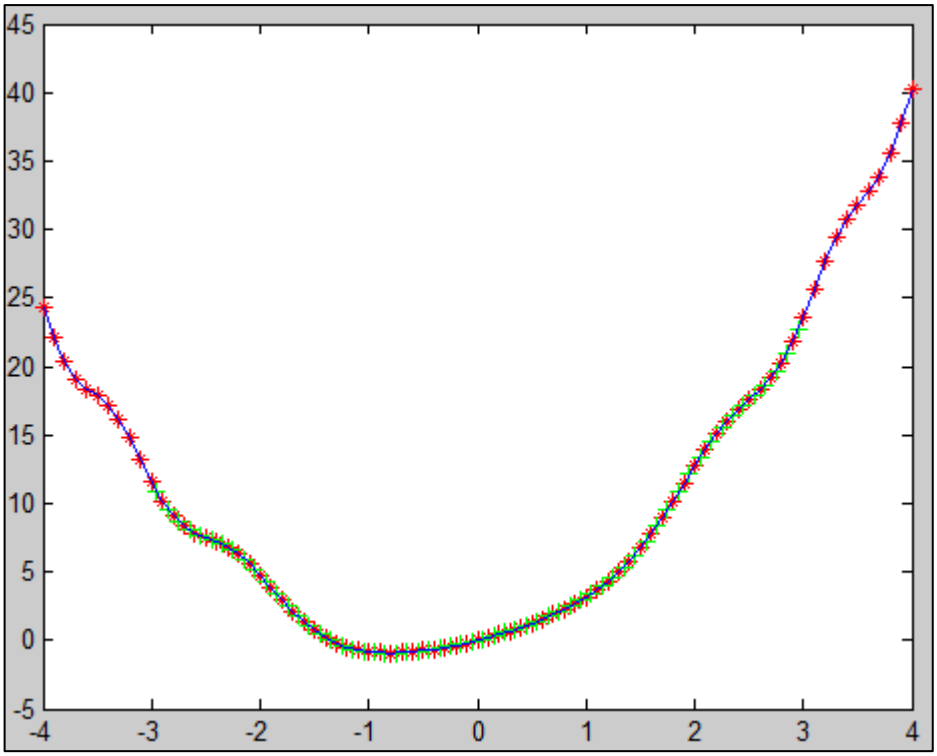
20. ábra A középpontok önszerveződő kiválasztása PSO algoritmussal - tanulási stratégiát használó RBF hálózat globális átlagos hibájának alakulása a tanítás folyamán.

Látható, hogy a hálózat tanulása 81 epochot vett igénybe és a globális átlagos hiba értéke 10^{-7} nagyságrendű. Azért kellett globális átlagos hibát figyelni, mert a PSO egy részecske alapú optimalizációs eljárás, így a teljes populáció egyedei eltérő globális hibát eredményeznek. Ennek elkerülése érdekében ezen globális hibák átlagolásra kerültek és ez lett a hálózat kimenetén megjelenő hiba értéke is.

A hálózat általánosításának tesztelésére ugyanazon elv a követendő, amelyek az előző két példában is alkalmazásra kerültek. Az így kapott hálózat kimenetét a 21. ábra szemlélteti. Az ábrán megjelenő piros pontok a tanításhoz felhasznált mintavételezési pontok voltak. A $[-3, +3]$ intervallumba eső zöld pontok a hálózat szimulációjánál kerültek alkalmazásra. A hálózat kimenetét pedig a kék folytonos vonal jelzi. Látható, hogy az illesztés tökéletes még a pontos szimulációs pontokon kívül is.

A PSO eljárást használó tanulási stratégiák tehát rendkívül jó approximáló képességgel rendelkeznek és jelentősen növelik a hálózatok általánosítási képességét. Ez elsősorban annak köszönhető, hogy nem minden esetben ugyanaz a részecske kerül kiválasztásra, mint a populáció legrátermettebb egyede és két egymást követő iterációban ugyanazon részecskét figyelemmel kísérve a részecske sebessége és aktuális pozíciója nem egyezik meg. Noha ezen eljárás konvergenciájára nincs semmiféle garancia a felmerülő problémák széles skálájára

alkalmazható. Az ilyen módon tanított hálózat függvény approximációs képessége kiváló. Az általam kipróbált feladatok azt mutatják, hogy ezen eljárás hasonló tulajdonságokkal rendelkezik az osztályozási feladatok megoldásánál is.



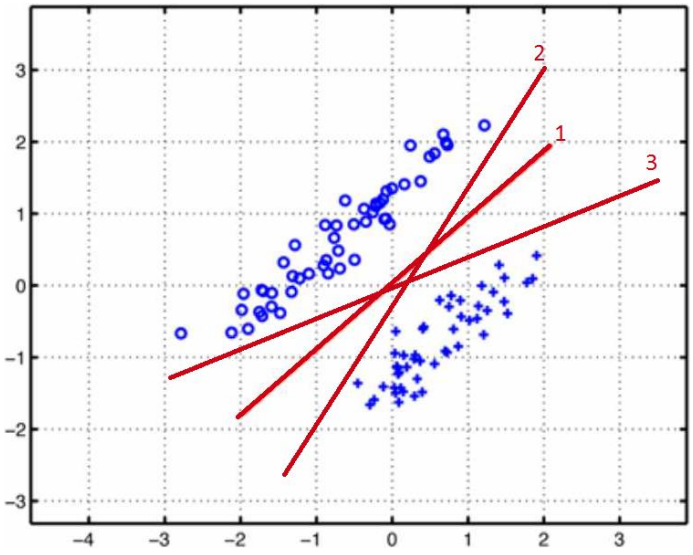
21. ábra A középpontok önszerveződő kiválasztása PSO algoritmussal az F3(x) függvény közelítésére.

Ezzel az RBF hálózatok képességét bemutató MATLAB példa végére értünk. A Következő témakörben megismerkedünk az SVM (Tartóvektor gép) „hálózatokkal", azok két típusával illetve tanítási algoritmusával.

3. Fejezet

Szeparáló hipersíkok

Az Tartóvektor gépek a korábban tárgyalt neurális hálózatok gyengeségére adnak egy lehetséges megoldást. Mindkét esetben adathalmazokat szeparálunk egy valamilyen hipersíkkal amelyet a $\langle \omega^T x \rangle + b$ összefüggéssel tudunk leírni. Ebben a formában ω −t az elválasztó hipersík normálvektorának nevezzük. Egy ilyen szeparálásnál az a legnagyobb probléma, hogy két adathalmazt $\mathcal{A}$ és $\mathcal{B}$ nem csak egy szeparáló hipersíkkal tudunk szétválasztani. Ezt szemlélteti a 22. ábra.



22. ábra Két adathalmazt szeparáló egyenesek.

Ekkor tetszőleges $a \in \mathcal{A} \cup \mathcal{B}$ pontról úgy tudjuk eldönteni, hogy melyik osztályba tartozik, ha meghatározzuk az (51) kifejezés értékét, amely egy adott x pontról eldönti, hogy melyik osztályba tartozik.

$$h_{\omega,b}(x) = sign(\omega^T x + b), \tag{50}$$

ha

$$\omega = \sum_i \alpha_i * y^{(i)} * x^{(i)}$$

alakú függvény. A probléma a 22. ábrán szemléltetett 1), 2), illetve 3) szeparáló egyenesekkel az, hogy adott $a \in \mathcal{A} \cup \mathcal{B}$ pont esetén a $h_{\omega,b}(x)$ értéke nem feltétlenül egyezik meg. Tekintsük például az (1.75,2.0) pontot. Ez a 2) szeparáló egyenest felhasználva $h_{\omega,b}(x)$ értékére a pontot a $\mathcal{B}$ osztályba sorolná be, míg az 1) illetve 3) szeparáló egyeneseket fölhasználva az $\mathcal{A}$ osztályba. A feladat SVM-nél tehát nagyon hasonló a klasszikus neurális hálózatoknál alkalmazott osztályozási feladatokhoz. Adott

$$\{([x_1, x_2, x_3, \dots, x_N]_1, y_1), ([x_1, x_2, x_3, \dots, x_N]_2, y_2), \dots, ([x_1, x_2, x_3, \dots, x_N]_K, y_K)\}$$

tanuló adathalmaz, amely halmaz elemei elem párok. Ezen elem párok első komponense egy $\mathbb{R}^N$-beli vektor, míg második komponense egy $\{-1, +1\}$ halmazbeli érték, amely azt jelzi, hogy az adott $[x_1, x_2, x_3, \dots, x_N]_i$ pont melyik osztályba tartozik.

Könnyen belátható, hogy két adathalmazt szeparáló hipersíkok halmazában található egy optimális hipersík, amely mindkét halmaz elemeitől maximális távolságra van. Ismeretes, hogy a feladatot az alábbi módon formalizálhatjuk:

$$\min \frac{1}{2} * \|\omega\|^2 \ \ feltéve, hogy\ y^{(i)}\left(\omega^T x^{(i)} + b\right) \geq 1.$$

A Kuhn-Tucker-tétel felhasználásával belátható, hogy alábbi problémát kell megoldani

$$\max \sum_i \alpha_i - \frac{1}{2} * \sum_i \sum_j y^{(i)} * y^{(j)} * \alpha_i * \alpha_j * \langle x^{(i)}, x^{(j)} \rangle, bizonyos\ lineáris\ feltételekkel$$

Az optimális hipersík normálvektora:

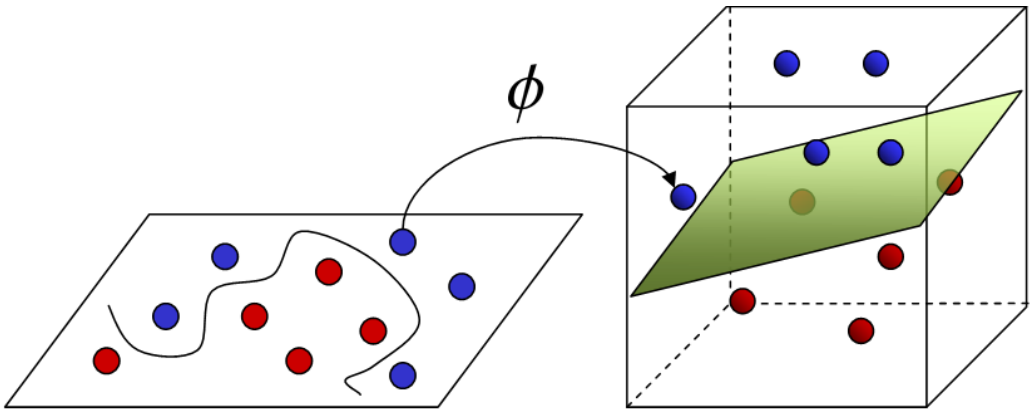
$$\omega = \sum_i \alpha_i * y^{(i)} * x^{(i)}$$

lesz, ahol az $\alpha_i^{(0)}$ a fenti feladat megoldása.

Ez egy kvadratikus programozási feladat [1]. Tehát azt a szeparáló hipersíkot kell megtalálnunk, amely maximális távolságra van mindkét adathalmaz egymáshoz legközelebb eső pontjaitól. Könnyen belátható, hogy a 22. ábrán az optimális hipersík a 1)-essel jelölt elválasztó egyenes.

Lineárisan nem szeparálható adathalmazok

A való életben felmerülő problémák gyakran nem lineárisan szeparálhatóak. Ilyen lineárisan nem szeparálható halmazokat szemléltet a 23. ábra bal oldalán szereplő adathalmaz. Látható, nem létezik szeparáló egyenes, amellyel elválaszthatóak lennének egymástól a két ponthalmaz tagjai [7]. Az ilyen jellegű szeparálást, amelynél az adott reprezentációs térben adott adathalmaz két osztályát nem tudjuk egy szeparáló hipersíkkal elválasztani, nemlineáris szeparálásnak nevezzük.



23. ábra Lineárisan nem szeparálható adathalmaz balra, illetve ezen adathalmaz lineáris szeparálása egy magasabb dimenziójú térben jobbra.

Ilyen lineárisan nem szeparálható adathalmazok esetén a halmazban szereplő adatokat transzformáljuk egy magasabb dimenziójú térbe. Ez elsősorban azért célszerű, mert ha az adott reprezentációs térben az adatok lineárisan nem szeparálhatóak, őket egy nagyobb dimenziójú térbe transzformálva már lineárisan szeparálható adatokat kaphatunk. A 22. ábrán szemléltetett $\Phi(x_i)$ függvény a síkbeli pontokat térbe transzformálja. Egy ilyen lehetséges leképezés a következő:

$$\Phi(x_1, x_2) = \left[x_1^2, x_2^2, \sqrt{2} * x_1 * x_2\right].$$

Így a pontokat a kapott 3-dimenziós térben már lineárisan szeparálni tudjuk. Úgy is mondhatjuk, hogy a Φ függvény egy $\Phi: \mathbb{R}^N \rightarrow \mathbb{R}^K$ leképezést valósít meg ahol $N \ll K$ feltétel teljesül. A feladatot tehát az alábbi módon fogalmazhatjuk át.

Magfüggvények

Legyen adott egy tetszőleges adathalmaz melynek elemei olyan rendezett elem kettesek, amelyek első komponense egy magfüggvény [8] által magasabb dimenziójú térbe transzformált pontot jelöl ki. A második komponensük pedig egy valós szám a $\{-1, +1\}$ halmazból, amely azt mondja meg, hogy az elem kettes első komponense melyik osztályba tartozik. Vegyük észre, hogy a $\Phi(.)$ leképezésnek lehet, hogy olyan térbe kell az adatokat transzformálnia, hogy ezen adatok számítógépes reprezentációja nem lehetséges. Ezen problémát az alábbi módon formalizálhatjuk:

$$\{(\Phi([x_1, x_2, x_3, \dots, x_N])_1, y_1), (\Phi([x_1, x_2, x_3, \dots, x_N])_2, y_2), \dots, (\Phi([x_1, x_2, x_3, \dots, x_N])_K, y_K)\}$$

Ezen formulában szereplő elem kettesek első komponense $\Phi([x_1, x_2, x_3, \dots, x_N])_i$ egy magasabb térbe transzformált vektor, amely vektor nem garantált, hogy ábrázolható vagy letárolható a számítógép memóriájában [8]. Ekkor mivel minden bemeneti adatot magasabb dimenziójú térbe transzformálunk az (52) összefüggésben szereplő $\langle x^{(i)}, x \rangle$ belső szorzat paramétereit is transzformálnunk kell a következőféleképp:

$$\forall_i \forall_j \langle x^{(i)}, x^{(j)} \rangle \rightarrow \langle \Phi(x^{(i)}), \Phi(x^{(j)}) \rangle = K(x^{(i)}, x^{(j)})$$

ahol $K(x^{(i)}, x^{(j)})$ jelöli a felhasznált magfüggvényt.

Mercer-tétel: *Legyen $K(x, z)$ adott magfüggvény. Ekkor, ha K Mercer-kernel vagyis $\exists \Phi$, hogy $K(x, z) = \langle \Phi(x), \Phi(z) \rangle$, akkor minden $\{x^{(1)}, \dots, x^{(m)}\}$ pontra $(m < \infty)$ a kernel-mátrix $(K \in \mathbb{R}^{NxM})$ pozitív szemi definit.*

Ezen tétel bizonyítását a következőképpen végezhetjük el. Tegyük fel, hogy K egy magfüggvény, továbbá, hogy adott $\{x^{(1)}, \dots, x^{(m)}\}$ és $(z \in \mathbb{R}^M)$. Ekkor a kernel mátrixa – magfüggvényekből álló mátrix - $(K \in \mathbb{R}^{NxM})$, amelynek jelölje K_{ij} a következőképpen megadott elemeit:

$$K_{ij} = K(x^{(i)}, x^{(j)})$$

A bizonyítás a következőképpen végezhető el:

$$z^T * K * z = \sum_{i=1}^{M}\sum_{j=1}^{M} z_i * K_{ij} * z_j = \sum_i \sum_j z_i * \Phi\left(x^{(i)}\right)^T * \Phi\left(x^{(j)}\right) * z_j$$

$$= \sum_i \sum_j z_j * \sum_k \left(\Phi\left(x^{(i)}\right)\right)_k * \left(\Phi\left(x^{(j)}\right)\right)_k * z_j = \sum_k \left(\sum_i z_i * \left(\Phi\left(x^{(i)}\right)\right)_k\right)^2 \geq 0$$

Vagyis $K \geq 0$ teljesül, amely a tétel feltevése volt.

A kérdés ezek után az, hogy milyen kernelfüggvényt érdemes választani az egyes feladatok megoldásához. Legyenek adottak $x \rightarrow \Phi(x)$, illetve $z \rightarrow \Phi(\mathrm{z})$. Ekkor $K(x,z)$ magfüggvényre az alábbi teljesül:

$$K(x,z) = \langle \Phi(\mathrm{x}), \Phi(\mathrm{z}) \rangle$$

Az ilyen módon definiált magfüggvénynek az alábbi két kritériumnak kell eleget tennie:

$$K(x,z) = \begin{cases} legyen\ nagy, ha\ x\ és\ z\ vektorok\ által\ reprezentált\ adatok\ hasonlóak \\ legyen\ kicsi\ egyébként \end{cases} \quad (51)$$

SVM-nél elsősorban azokat a magfüggvényeket alkalmazhatjuk hatékonyan, amelyek teljesítik mind a Mercer-tétel által kimondott tulajdonságot mind pedig az (51) összefüggés által leírt kritériumot [7],[8]. A gyakorlatban az alábbi három magfüggvényt alkalmazzák széles körben mind osztályozási és egyéb feladatokra:

RBF kernel	*Polinomiális kernel*	*MLP kernel*
$K(x,z) = \exp\left(-\frac{\|x-z\|^2}{2*\sigma^2}\right)$	$K(x,z) = (x^T z + c)^d$	$K(x,z) = \tanh(x^T z + b)$

24. ábra A gyakorlatban alkalmazott magfüggvények táblázata.

Esettanulmány - DNS szekvenciák osztályozása

Tekintsük a következő példát annak demonstrálására, hogy milyen módon kezelhetjük a magfüggvényeket. A feladatunk legyen az, hogy egy DNS szekvenciáról el kell döntenünk, hogy az adott DNS szekvencia adott részei által alkotott gének milyen feladatot látnak el. Ebben az esetben az osztályok a lehetséges feladatok lehetnek. Tekintsünk el most attól, hogy

milyen módon tudunk több osztályos osztályozást végezni az SVM-el. A feladat tehát egy adott DNS szekvencia:

```
catatgaatagtcaaggaaaaaatatacaaaaagagataagaagcatcaacaact
tacaaatataagtccatataacgcaattatccttgaaaaacttatacataagcta
aaacaaaccaatgaacaaatacggcttgttcaagaactccataaaaagcgagtaa
aatgacaaaagatacatcacaatgagtcataaatcaacgatggtcaggaaagaac
aaagaatcatcaacaattaacaaattatac
```

ezen DNS szekvenciáról akarjuk eldönteni, hogy milyen feladatokért felelős. A probléma elsősorban az, hogy minden DNS szekvencia általában eltérő hosszúságú és egyes részek hasonlóak, míg mások alapvetően különbözőek egymástól. A kérdés az, hogy ilyen esetben milyen módon adhatjuk meg az SVM tanításához szükséges magfüggvényt.

$$\Phi(\mathrm{x}) = ?$$

Erre egy lehetséges megoldás, amit Professor Andrew Ng. az Amerikai Stanford egyetem Informatikai Tudományok tanszékének professzora protein szekvenciák osztályozásához mutatott be nyolcadik Gépi Tanulás (CS 229) előadásán a következő:

> Állítsuk elő az (ACTG) betűk összes lehetséges 4 hosszúságú permutációját. Ezen, négyhosszúságú permutációként kapott karakterláncokat, keressük meg az eredeti DNS szekvenciában és számoljuk meg, hogy hányszor fordulnak elő. Az előfordulásokat tároljuk le független vektorban. Az így generált vektor legyen a DNS szekvenciához tartozó $\Phi(.)$ sajátságvektor.

Az így kapott sajátságvektor igen magas dimenziószámú $\left(\Phi(x) \in \mathbb{R}^{(k^4)}\right)$. Könnyen belátható, hogy ez igen alacsony k értékekre is kezelhetetlen méretű sajátságvektort eredményez. Ezen igen nagyméretű sajátságvektor letárolása probléma lehet még a mai modern számítógépeknek is. Ilyen esetekben célszerű ezen $\Phi(x)$ sajátságvektorokat az alábbi módon tárolnunk:

$$\Phi(x)^T \Phi(x)$$

Ilyen módon már kezelhető méretűvé válik a sajátságvektorunk.

SM-SVM (*Soft Margin Support Vector Machine)*

Az SVM-nek egy másik változata is széles körben elterjedt. Ezt nevezi a szakirodalom *Soft Margin SVM*-nek. Ezen változat lényege, hogy implementál egy az RBF hálózatnál már tárgyalt D lineáris operátorhoz hasonló büntető paramétert. Ezen paraméter a maximális margó nagyságát hivatott befolyásolni. Ezen elmélet szerint lehetnek olyan adott osztályhoz tartozó pontok, amelyek a meghatározott margón belül esnek. Ezért magyarul az SVM ezen típusát *lágy margó*jú SVM-nek nevezhetnénk. Ezt úgy kell érteni, hogy a margókra nem klasszikus megszorítást alkalmazzuk. Ezen megszorítás felírásához újra fel kell írnunk az SVM-nél már korábban megadott feladatot, de a büntető paraméter felhasználásával.

$$\min\left(\frac{1}{2} * \|\omega\|^2 + C * \sum_{i=1}^{m} \xi_i\right)$$

$$feltéve, hogy\ \xi_i > 0\ minden\ i\ esetén\ és\ y^{(i)}\left(\omega^T x^{(i)} + b\right) \geq 1 - \xi_i$$

Ezen két megszorítást nem szükséges explicit megadni. Megadhatjuk ezeket az optimalizálási feladat leírásában implicit módon. Ha ezt a megoldást választjuk, akkor az optimalizálási feladat az következőképpen adható meg:

$$\min_{\omega} \frac{1}{2} * \|\omega\|^2 + c * \sum_{i=1}^{m} \max\left(0, 1 - y^{(i)}\left(\omega^T x^{(i)} + b\right)\right) \tag{52}$$

Az így kapott optimalizálási feladat megoldása a célunk. Vegyük észre, hogy nem kötelező ω szerint minimalizálnunk, ugyanis ω az alábbi módon számolható:

$$\omega = \sum_{i=1}^{m} \alpha_i * \Phi(x_i) \tag{53}$$

Így a probléma egy konvex optimalizálási feladat megoldását jelenti és az (53) összefüggés által leírt megoldandó optimalizálási feladat feladatban (α_i, b) paramétereket keressük.

Tehát α_i egy pozitív valós szám, amely csak abban az esetben nem nullaértékű, ha az $x^{(i)}$ által definiált adat tartóvektor. Azonban tartsuk szem előtt azt is, hogy a tartóvektorok most nem csak azon adatokat jelentik, amelyek a margón helyezkednek el a reprezentációs térben,

de azon adatokat is amelyek a margón belül helyezkednek el és amelyek a szeparáló hipersík rossz oldalán van.

Így már meg tudjuk fogalmazni az SM-SVM általános feladatát. Legyenek adottak, $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ adatok egy halmaza, amely halmaz rendezett elempároknak az első komponenseire $\left(x_i \in \mathbb{R}^K\right)$; második komponenseire $(y_i \in \{-1, +1\})$. Feltételezzük, hogy az adathalmaz által leírt osztályok mindegyikében vannak nem az osztályba tartozó pontok is. Ilyen módon a klasszikus SVM nem tudná elvégezni a szeparálást még magfüggvények alkalmazásával sem kellő pontossággal. Ez nagyban hasonlít Tyihonov elméletére miszerint alapvetően hibás feladatokra is próbál helyes megoldást adni. Ennek ismeretében feltételezhetjük, hogy a *Soft Margin Support Vector Machine* alapvetően „hibás"adathalmazt is képes megfelelően szeparálni.

A következőkben az SVM tanítási módszerei közül az SMO (Sequential Minimal Optimization) eljárást fogjuk áttekinteni. Fontos azonban megjegyezni, hogy a tartóvektor gépeknél nem szerencsés szóhasználat a tanítás. Könnyedén belátható, hogy a gépi tanulásnak ezen módszere kevésbé flexibilis mint a neurális hálózatok. A tanítás jelen esetben egy függvény maximum helyének megkeresését jelenti, amelyből a tartóvektor gép futásidőben határozza meg ω paramétereit az (53) összefüggésnek megfelelően. A másik probléma a tartóvektor gépekkel, hogy tanítás után is szükségünk van azon adathalmazra, amellyel az (52) kifejezés által leírt optimalizálási feladatot megoldottuk. Erre azért van szükség, mert az SVM egy adott x pontról az (50) összefüggés által tudja megmondani, hogy az melyik osztályhoz tartozik. Ezen összefüggésben szereplő $x^{(i)}$, illetve $y^{(i)}$ paraméterek a tanulóadatok halmazának elemei. Ez a tulajdonság nagyban ronthatja az SVM hordozhatóságát, hiszen egy kisméretű program mellé esetlegesen igen nagyméretű tanuló adathalmazt kell csatolnunk, hogy az használható legyen más környezetben is. Mindezek mellett mivel az SVM az optimális szeparáló hipersíkot határozza meg minden esetben igen széles körben alkalmazzák.

SMO – Szekvenciális Optimalizáció [16]

Az SMO algoritmus lényege, hogy a tanuló adathalmaznak csupán egy részhalmazát optimalizáljuk, míg a többivel nem foglalkozunk. A legrosszabb esetben a teljes tanuló adathalmazt optimalizálnunk kell, de előfordulhat az is, hogy csupán egyetlen változót kell

optimalizálnunk és a többit nem. Általános esetben a teljes $\mathcal{T}$ tanuló adathalmaznak csupán egy $\mathcal{S}$ részhalmazára optimalizálunk. Ha $\mathcal{S}$ minden elemére optimalizáltunk, akkor egy $\mathcal{K}$ részhalmazt választunk olyan módon, hogy $\mathcal{K} \cap \mathcal{S} = \emptyset$ egészen addig, amíg konvergencia nem tapasztalható az SVM hibájában. Az algoritmust az alábbi lépések szerint kell végrehajtani:

1. lépés: Generáljuk olyan α értékeket, amelyre teljesül $\alpha_i = 0 \Rightarrow y^{(i)}\left(\omega^T x^{(i)} + b\right) \geq 1$.
2. lépés: Az 1. lépésben generált α értékek közül válasszunk ki két egymástól különböző α_i és α_j paramétereket, amelyek teljesítik $y^{(i)}\left(\omega^T x^{(i)} + b\right) < 1$ feltételt.
3. lépés: Határozzuk meg α_i új értékét az alábbi összefüggésnek megfelelően:

$$\alpha_i = \frac{1}{y^{(j)}} * \left(\left(\sum_{i \geq 3} \alpha_i * y^{(i)}\right) - y^{(j)} * \alpha_j\right)$$

MATLAB: Legkisebb Négyzetes Tartóvektor Gépek [13]

A továbbiakban tárgyalandó SVM-el elkészített feladataim bemutatásához elengedhetetlen, hogy bemutassam a MATLAB-hoz készített LS-SVM Toolboxot. A LibLS-SVM programcsomag segítségével az SVM-et nem csak osztályozási, de klaszterezési, approximációs és regressziós feladatokra is használni tudjuk.

Ezen programcsomag használatát tekintve az alábbiak szering épül föl:

Adathalmaz→Funkcionális/Objektum Orientált Interfész→tunelssvm→trainlssvm→sim/plot-lssvm

Az adathalmaz rendezett elem kettesesekből állhat, amely rendezett elem kettesek első komponense egy PxQ mátrix lehet, és második komponense szintén egy PxQ mátrix. Az általam készített feladatok megoldása során a Funkcionális interfészt használtam. Ez azt jelenti, hogy a teljes SVM „gépet” meg kellett adnom minden ezt igénylő függvény paramétereként. Az LS-SVM programcsomag az alábbi módon definiál egy SVM-et:

```
{X,Y,type,gam,sig2,kernel}
```

Egy ilyen SVM paramétereinek jelentése a következő. X jelenti a tanulóadatok rendezett elem kettesének első komponensét, azaz a bemenetet. Y paraméter ugyanezen rendezett elem kettes azt adja meg, hogy az ehhez tartozó bemenet mely osztályba tartozik. A type paraméter az SVM típusát határozza meg. Ezen programcsomagban kétféle típust különböztetünk meg:

- classification – Klasszikus osztályozási feladatot megvalósító SVM-hez valamint klaszterezéshez.
- function estimation – Függvény approximációhoz valamint regressziós feladatok megoldásához.

A gam paraméter és sig2 paraméterek kernelenként változnak. A későbbiekben bemutatásra kerülő feladatoknál mivel csak RBF kernelt használtam, így ezen paramétereken kívül nem lesz más. A gam paraméter a regularizációs paramétert jelenti, míg sig2 a 24. ábráról leolvasható RBF kernel σ^2 paramétere. Ezek a paraméterek nem feltétlenül a felhasználó által megadandóak, mivel a tunelssvm ezen paramétereket hivatott automatikusan meghatározni az adott (X,Y) paraméterekre.

A kernel paraméter három különböző értéket vehet föl amelyek rendre a 24. ábrán szemléltetett magfüggvények lehetnek. Így a lehetséges kernelek az ’RBF_kernel’ a ’poly_kernel’, ’MLP_kernel’ és egy negyedik magfüggvény is implementálásra került, ez pedig a ’lin_kernel’.

Egy tetszőleges feladat megoldása tehát az alábbi parancsokból építhető föl:

- tunelssvm – Meghatározza a gam és sig2 paramétereket specifikusan az adott adathalmazra történő vonatkozásban.
- trainlssvm – Betanítja a SVM-et SMO algoritmust felhasználva
- simlssvm – Az adott SVM-et és a kapott alfa és b paraméterek felhasználásával új ismeretlen adatokra határozza meg az SVM kimenetét.
- plotlssvm – Ábrázolja az SVM kimenetét. Egyik nagy hátránya, hogy csak kétdimenziós ábrákat tud készíteni így a többváltozós függvények közelítésének eredményét nem tudja effektíven megjeleníteni. Ezen feladatok megjelenítését a MATLAB $surf(.)$ nevű függvényével oldottam meg.

A továbbiakban bemutatok két függvény approximációs feladatot, amelyek során az alábbi két függvényt közelítettem LS-SVM –el.

$$f_1(x,y) = x * \exp(-x^2 - y^2)$$

$$f_2(x,y) = x^2 - y^2$$

Ezen két függvény approximációjához az tanulóadatok bemeneti adatait a $meshgrid()$ függvénnyel az alábbi módon hoztam létre:

```
[x,y] = meshgrid(-4.0:0.1:4.0, -4.0:0.1:4.0);
```

Ezt követően [x,y] mátrix minden pontjában meg kellett határoznom mind $f_1(x,y)$ mind pedig $f_2(x,y)$ függvények helyettesítési értékét. Így kaptam két azonos dimenziószámú mátrixot, amelyeket már könnyedén fel tudtam használni az SVM tanításához. Az approximáció hibáját az alábbi módon számoltam:

Miután az SVM-et betanítottam, generáltam egy újabb adathalmazt ugyanezen méretekkel a $meshgrid(.)$ függvényt felhasználva, majd a szimuláltam az SVM-et az új adathalmazra. Ennek eredményeképpen előállt egy újabb mátrix, amelyet kivontam a függvény helyettesítési értékeit tároló mátrixból, majd meghatároztam az így eredményül kapott mátrix euklideszi normáját. Ezzel megkaptam, hogy a két eredményül kapott mátrix milyen távol van egymástól és ezen értéket használtam föl az approximáció hibájaként.

A függvények helyettesítési értékét az alábbi módon hoztam létre:

```
for i=1:length(x)
    for j=1:length(y)
        F1xy(i,j) = x.*exp(-x.^2-y.^2);
        F2xy(i,j)=x^2-y^2;
    end
end
```

Az így létrejött két mátrixot fel tudtam használni az [x,y] mátrixszal. A következőkben az általam megoldott feladatok részletesebb ismertetésére kerül sor.

MATLAB: kétváltozós függvények approximációja

Célom az volt, hogy a korábban bemutatott LS-SVM programcsomagot többváltozós függvények approximációjára tudjam kipróbálni és összehasonlítani az eredeti függvénnyel. Majd a közelített függvény és eredeti függvény eltérését megadva definiálni a közelítés hibáját. A közelítendő két függvény a következő volt:

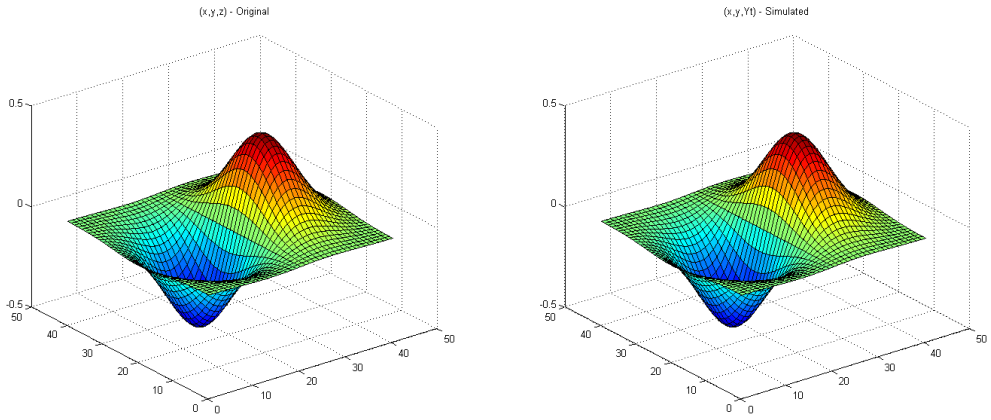
1. $f_1(x,y) = x * \exp(-x^2 - y^2)$
2. $f_2(x,y) = x^2 - y^2$

Az első függvény közkedvelt példája a különböző gradiens alapú szélsőérték kereső eljárásának demonstrálásánál, míg a második a klasszikus nyeregfüggvény, amely semmilyen szélsőértékkel nem rendelkezik. Mindkét függvényt ugyanazon az értelmezési tartományon értelmeztem, majd ugyanezen értelmezési tartományon hoztam létre az SVM szimulációjából kapott közelített függvényértékeket. Az így kapott mátrixot kivonva az eredeti helyettesítési értékeket tartalmazó mátrixból létrejött egy különbségmátrix, amelynek euklideszi normáját vettem a közelítés hibájának.

A 25. ábra az $f_1(x,y)$ függvény approximációja utáni kimenetet szemlélteti. Baloldalon az eredeti függvény ábrázolása látható, amelyet a $surf(.)$ beépített függvénnyel készítettem el; jobb oldalon pedig a betanított LS-SVM szimulációja során előállt mátrix került ábrázolásra.

Az LS-SVM önmaga határozta meg az RBF magfüggvényhez (24. ábra) tartozó paraméterét illetve a regularizációs paramétert az alábbi parancs végrehajtásával:

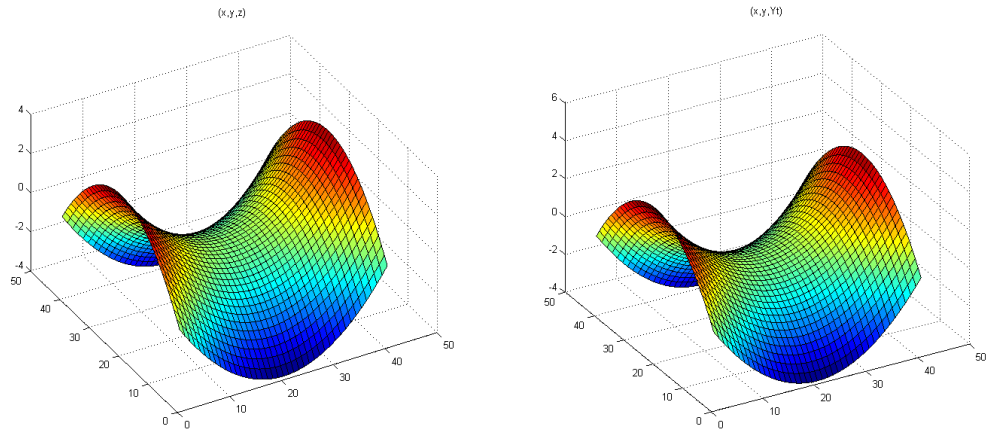
```
>> [gam, sig2] = tunelssvm({[x,y],Fxy,'f',[],[],'RBF_kernel'},
'simplex', 'leaveoneoutlssvm',{'mse'});
```



25. ábra x*exp(-x^2-y^2) függvény eredeti ábrázolása bal valamint LS-SVM-el közelített ábrázolása jobb -oldalon.

Az így betanított közelítés hibája 0.0028, ami igen jónak tekinthető, ha MLP vagy RBF hálózattal akarunk többváltozós függvényt közelíteni.

Az $f_2(x,y)$ függvényt ugyanezen az elven közelítettem teljesen analóg LS-SVM hálózattal. Ezen közelítés eredményét a 26. ábra szemlélteti, amely ábra bal oldalán az eredeti függvény ábrázolása jobb oldalon pedig az SVM szimulációja során előállt mátrix ábrázolása látható.



26. ábra x^2-y^2 függvény approximációja. Baloldalon az eredeti függvény képe, jobb oldalon a közelítés során előállt mátrix ábrázolása látható.

Ezen függvény approximációjának hibája 1.028e-006. Ebből arra lehet következtetni, hogy az exponenciális, illetve szögfüggvények megnehezítik az adott függvény approximációját még SVM esetében is.

A klasszikus SVM és az SM-SVM mellett talán az LS-SVM lehet a következő széles körben elterjedt variánsa a tartóvektor gépeknek. A függvény approximációs és regressziós tulajdonságaik mellett hasonlóan jól alkalmazhatóak klaszterezési feladatok elvégzésére így kibővítve a klasszikus SVM korlátolt képességeit.

Összegzés

Ezen három fejezetben lényegében áttekintettük a neurális hálózatok három legismertebb típusát. Megfigyelhető, hogy a neurális hálózat csak az előrecsatolt (és visszacsatolt) hálózatok esetén találó elnevezés. Ezen hálózattípusok még nagyban függnek attól, hogy az egyes neuronok között milyen kapcsolati viszony került kialakításra, míg a későbbiekben tárgyalt RBF, illetve SVM hálózatok már inkább matematikai modellek, mint hálózatok.

Mindhárom modellnek megvannak a maguk előnyei, illetve hátrányai, azt azonban fontos megjegyezni, hogy az utóbbi években egyre nagyobb mértékben bukkannak föl egyes területeken a visszacsatolt neuronhálók. Mivel ezen hálózatok tanítási fázisa az itt tárgyalt hálótípusokhoz képest sokszor nagyobb számítási kapacitást igényelt, így meg kellett várni, amíg a számítógépek processzorai képesek lettek ezeket elérhető időn belül végrehajtani.

Mivel a visszacsatolt neuronhálók tanítása sokkalta bonyolultabb, mint az általam bemutatott módszerek így azokra nem térek ki külön. Formálisan egy visszacsatolt hálózat tanítását úgy lehet elképzelni, mintha egy folyamatosan hullámzó vízfelszínen akarnánk megkeresni azt az egy pontot, amely ponton a vízszint folyamatosan a legalacsonyabb volt a megfigyelt időtartam alatt.

A klasszikus SVM hálókat inkább a legkisebb négyzetes SVM hálók kezdik leváltani főleg osztályozási feladatok megoldásában használatosak leginkább. Azt azonban fontos megjegyezni, hogy noha az SVM hálók optimális hipersíkot találnak mivel ezen hálózatok gyakorlati alkalmazásához elengedhetetlenek a hálózat tanítása során felhasznált tanulóadatok

közül azon adatok, amelyeket a hálózat tartóvektorokként definiált nem terjednek olyan mértékben, mint a neuronhálók.

A neuronhálók egyeduralmát elsősorban a genetikus algoritmusokban rejlő potenciál jelentheti. A genetikus algoritmusok azért közkedveltek ezen a területen, mert egyszerű számításokat kell elvégezni, és ha megfelelő terminálási feltételt szabunk meg garantálhatjuk, hogy egy fix méretű keresési térben nem akadunk föl egy lokális minimumban. Erre egyértelmű matematikai feltétel nem adható, csak különféle heurisztikák alkalmazhatóak. A szimulált hűtés például egy véletlen szám generálásával próbálja eldönteni, hogy a talált minimum helyet elfogadja-e globális minimumnak vagy nem. Ez a bizonytalanság kezelés már rontja egy amúgy sem pontos numerikus közelítés értékét, ami egyértelműen kedvez a feltörekvő genetikus algoritmusoknak.

Irodalomjegyzék

[1] Peter Norvig, Stuart J. Russell, *Mesterséges Intelligencia - Modern megközelítésben (2. kiadás)*, Panem Kiadó Kft. 2006

[2] Futó Iván, *Mesterséges Intelligencia*, Aula Kiadó 1999

[3] Simon Haykin, *Neural Networks and Learning Machines (3rd Edition)*, Prentice Hall; 3 edition (November 28, 2008)

[4] Konstantinos E. Parsopoulos, *Particle Swarm Optimization and Intelligence: Advances and Applications (Premier Reference Source)*, Information Science Publishing; 1 edition (February 28, 2010)

[5] Bipul Luitel, Ganesh Kumar Venayagamoorthy, *Training Simultaneous Recurrent Neural Networks Using the PSO-QI Algorithm to Learn MIMO Systems*, http://brain2grid.com/wp-content/uploads/2009/12/NN_Letter_PSOQI_SRN_edited_V2.pdf

[6] Anderson, J. A., *Introduction to Neural Networks*, Cambridge, MA: MIT Press 1995

[7] John Shawe-Taylor & Nello Cristianini, *Support Vector Machines and other kernel-based learning methods*, Cambridge University Press, 2000

[8] Nello Christianini, *Support Vector and Kernel Machines*, BIOwulf Technologies ICML 2001, http://pages.cs.wisc.edu/~shavlik/SVM_icml01_tutorial.pdf

[9] Mark Hudson Beale, Martin T. Hagan, Howard B. Demuth – Neural Network Toolbox™ 7 User's Guide, The Mathsworks Inc. 2010, http://www.mathworks.com/help/pdf_doc/nnet/nnet.pdf

[10] Fazekas István, *Neurális hálózatok*, mobiDIÁK könyvtár, 2003

[11] Bidyadhar Subudhi, Debashisha Jena, *Differential Evolution and Levenberg Marquard Trained Neural Network Scheme for Nonlinear System Identification*, Springer Science+Budiness Media LLC. 2008

[12] Manolis I.A. Lourakis, Antonis A. Argyros, *Is Levenberg.Marquardt the Most Efficient Optimization Algorithm for Implementing Bundle Adjustment?*, Institute of Computer Science, Foundation for Research and Technology

[13] K. de Brabanter, P. Karsmaker, F. Ojeda, C. Alzate, D. De Brabanter, K. Pelckmans, B. De Moor, J. Vandewalle, J.A.K. Suykens; *LS-SVMlab Toolbox User's Guide version 1.7;* Katholieke Universiteit Leuven Deparment of Electrical Engineering; 2010

[14] Annath Ranganathan, *The Levenberg-Marquardt Algorithm*, 8th June 2004

[15] J. Platt, Fast training of support vector machines using sequential minimal optimization, in Advances in Kernel Methods – Support Vector Learning (B. Scholkopf, C.J.C. Burges, and A.J. Smola, eds.), MIT Press, 185-208, 1999,

http://machinelearning123.pbworks.com/f/smo-algo+on+a+sheet.pdf

[16] Veress Krisztián, *A Newton és a Gauss-Newton módszerek alkalmazása egyenletrendszerek megoldására és nemlineáris optimalizálására*, 2007.július 10

http://www.inf.u-szeged.hu/~verkri/munka/newton_modszerek.pdf

Ábrák jegyzéke

1. ábra Biológiai Neuron sematikus rajza – Wikipedia illusztráció. *(5. oldal)*
2. ábra Rosenblatt-féle Perceptron modell, ahogyan azt a MATLAB [9] szoftverben használhatjuk. *(6. oldal)*
3. ábra Az (5) halmaz pontjai Descartes féle koordinátarendszerben ábrázolva. Kék/Piros jelzik a két osztályt amelyet a neuronnak szeparálnia kell a két halmazt elválasztó egyenes megtalálásával. *(9. oldal)*
4. ábra Többrétegű perceptron a MATLAB Neural Network Toolbox megvalósításában. *(14. oldal)*
5. ábra A MATLAB NNTool nevű grafikus Neurális Hálózat kezelő csomagja. *(20. oldal)*
6. ábra A XOR logikai művelet igazságtáblája illetve az ezt megvalósító logikai kapu sematikus rajza. *(23. oldal)*
7. ábra Adatobjektumok létrehozása varázsló *(24. oldal)*
8. ábra Network panel a szükséges beállításokkal. *(25. oldal)*
9. ábra A létrehozott előrecsatolt hiba-visszaterjesztéses neurális hálózat sematikus ábrája, ahogyan azt a MATLAB 7 verziójában láthatjuk. *(26. oldal)*
10. ábra Az elkészült Neurális hálózat tanítási paramétereinek beállításai *(26. oldal)*
11. ábra Az elkészült neurális hálózat tanulásának fázisait szemléltető panel. A panel felső részén a hálózat sematikus rajzát láthatjuk. Ezt követően a tanító algoritmusra vonatkozó tulajdonságok láthatóak, majd a tanítás jelenlegi állását követhetjük nyomon *(27. oldal)*
12. ábra A DE-LM algoritmussal tanított előrecsatolt hálózat függvény approximációs képessége illetve hibájának alakulása[12] *(28. oldal)*
13. ábra Általánosított RBF hálózat MATLAB Neural Network Toolbox programcsomagjában. *(51. oldal)*
14. ábra Első közelítendő függvény - F1(x) *(52. oldal)*
15. ábra Az RBF hálózat átlagos hibájának változása F1(x) közelítendő függvény esetén. *(53. oldal)*
16. ábra Az RBF hálózat tanításának eredmény F1(x) függvényre, a középpontok önszerveződő kiválasztásával KNN algoritmus szerint. *(54. oldal)*

17. ábra Az RBF hálózat globális hibájának alakulása F2(x) közelítendő függvény esetén. *(55. oldal)*
18. ábra Bal oldalon a közelítendő függvény, jobb oldalon pedig a betanított RBF hálózat általánosítási képessége 0.75 lépésközzel F2(x) függvényre. *(56. oldal)*
19. ábra A közelítendő F3(x) függvény grafikonja. *(57. oldal)*
20. ábra A középpontok önszerveződő kiválasztása PSO algoritmussal - tanulási stratégiát használó RBF hálózat globális átlagos hibájának alakulása a tanítás folyamán. *(57. oldal)*
21. ábra A középpontok önszerveződő kiválasztása PSO algoritmussal az F3(x) függvény közelítésére. *(58. oldal)*
22. ábra Két adathalmazt szeparáló egyenesek. *(59. oldal)*
23. ábra Lineárisan nem szeparálható adathalmaz balra, illetve ezen adathalmaz lineáris szeparálása egy magasabb dimenziójú térben jobbra. *(61. oldal)*
24. ábra A gyakorlatban alkalmazott Magfüggvények táblázata. *(64. oldal)*
25. ábra x*exp(-x^2-y^2) függvény eredeti ábrázolása bal valamint LS-SVM-el közelített ábrázolása jobb -oldalon. *(72. oldal)*
26. ábra x^2-y^2 függvény approximációja. Baloldalon az eredeti függvény képe, jobb oldalon a közelítés során előállt mátrix ábrázolása látható. *(72. oldal)*

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezető tanáromnak, dr. Fazekas Istvánnak, aki nélkül soha nem figyeltem volna föl a *Neurális Hálózatokra.* Valamint szeretnék köszönetet mondani az Informatikai Intézet tanárainak, akik munkájukkal remek alapot adtak a további szükséges ismeretek megértéséhez valamint elsajátításához illetve alkalmazásához. Külön köszönettel tartozom dr. Jeszenszky Péternek, aki rendkívül sok gyakorlati nehézségen segített át, valamint dr. Baran Ágnesnek a numerikus matematika alapjainak elsajátításában nyújtott munkájáért.