Artificiella neuronnät och andra lärande system,
2D1432

Laboration 1:
Framåtkopplade nät —
Deltaregeln och Backpropagation

1 Mål

Denna laboration handlar om framåtkopplade nätverk och hur de kan tränas med felkorrigerande inlärningsmetoder. Efter att ha genomfört laborationen skall du förstå:

- hur nätverk kan användas för *klassificering*, *funktionsanpassning* och *generalisering*
- begränsningarna för vilka klassificeringsuppgifter ett enlagersnät kan klara av
- hur ett flerlagersnät kan genomföra generell klassificering och funktionsanpassning
- hur *deltaregeln* och den *generaliserade deltaregeln* kan användas för att träna ett nätverk

2 Introduktion

I denna laboration ska du implementera en enlagers- och en tvålagersperceptron samt studera deras egenskaper. Eftersom beräkningarna med fördel uttrycks i vektor/matris-form är det naturligt att implementera det hela i Matlab[1].

Vi ska här begränsa oss till två inlärningsalgoritmer: *deltaregeln* (för enlagersperceptronen) och den *generaliserade deltaregeln* (för tvålagersperceptronen). Den generaliserade deltaregeln brukar ofta även kallas *error backpropagation* eller helt enkelt *BackProp*. Det är en av de mest använda övervakade inlärningsalgoritmerna för neuronnät och kan användas på flera olika sätt. I denna laboration

[1] Det går också utmärkt att använda Octave, den fria varianten av Matlab.

ska du få prova på att använda den för *klassificering*, *komprimering* och *funktionsapproximering*.

2.1 Representation av data

Låt oss först bestämma hur mönstren som ska läras in ska representeras. Eftersom det handlar om övervakad inlärning betyder det att träningsdata består av par av inmönster och utmönster. Vi väljer att göra s.k. *batch-inlärning*, vilket innebär att man behandlar alla mönster i träningsmängden samtidigt (parallellt) istället för att gå igenom dem ett efter ett och succesivt uppdatera vikterna. Batch-inlärning passar bättre för en matrisrepresentation och blir också mycket effektivare i Matlab. I batch-inlärning brukar varje genomgång av hela mönstermängden kallas en *epok* (eng. *epoch*). Genom en lämplig representation kan en hel epok utföras med några få matrisoperationer.

Vi måste dessutom bestämma oss för ifall vi ska använda en binär (0/1) eller en symmetrisk (−1/1) representation av mönstren. Det är praktiskt med det symmetriska intervallet −1 till 1, eftersom fler av formlerna blir enklare och alla trösklingar kan göras vid noll.

Vi väljer att lagra inmönstren och motsvarande utmönster som kolumner i två matriser: X respektive T. Med vår representation skulle till exempel XOR-problemet beskrivas genom:

$$X = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

Detta läses kolumnvis och betyder att mönstret $(-1, -1)$ ska ge resultat -1, $(1, -1)$ ska ge 1, o.s.v.

Enlagersperceptronen fungerar så att man beräknar summan av de viktade insignalerna och adderar en *bias*-term. Slutligen trösklas svaret. Om man har mer än ett utvärde använder man helt enkelt separata viktuppsättningar för de olika utsignalerna.

Dessa beräkningar blir mycket enkla i matrisform. För det första ser vi till att bli av med bias-termen genom att lägga till en extra insignal som alltid har värdet ett. I vårt exempel får vi alltså:

$$X_{\texttt{input}} = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

Vikterna beskrivs med hjälp av en matris W med lika många kolumner som längden på inmönstren och lika många rader som längden på utmönstren. Nätets utmönster för *alla* inmönster kan då beräknas genom en enkel matrismultiplikation, följt av en tröskling vid noll. Inlärning med hjälp av deltaregeln syftar,

med vår valda representation, till att hitta den viktuppsättning W som ger den bästa approximationen:

$$W \cdot \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \approx \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

Tyvärr råkar XOR-problemet vara ett av de klassiska problem där en enlagersperceptron inte kan klassificera alla mönstren rätt.

3 Datormiljö för kursen

Om du inte redan har gjort det, börja med att anpassa din datormiljö för kursen. Detta sker med kommandot `course join` och är nödvändigt för att labbarna skall fungera enligt anvisningarna. Om du inte registrerat dig på kursen (med `res checkin`) är det också hög tid att göra det nu!

```
> res checkin ann06
> course join ann06
```

Du behöver bara ge dessa kommandon en gång. Däremot kanske din labbpartner också vill passa på att göra samma sak, enligt nedan. Efter att man gjort `course join` måste man logga ut och sedan in igen innan ändringarna får effekt.

```
> ksu labbpartner
> res checkin ann06
> course join ann06
> kdestroy
> exit
```

Det finns ingen färdig kod att hämta för denna laboration, utan du kommer att skriva all kod själv, förslagsvis i en katalog enligt nedan:

```
> mkdir -p ~/ann06/lab1
> cd ~/ann06/lab1
```

4 Klassificering med en enlagersperceptron

4.1 Generera träningsdata

Börja med att generera lite testdata som kan användas för klassificering. För att göra det enkelt att grafiskt titta på resultaten nöjer vi oss med punkter

i två dimensioner. Vi låter indata bestå av två klasser, var och en med en normalfördelad spridning kring sin mittpunkt.

Det är lämpligt att skapa en fil (t.ex. `sepdata.m`) med alla Matlab-satser som behövs för att skapa datamängden. På så sätt kan man använda kommandot `sepdata` för att generera nya träningsdata.

Tal som är normalfördelade kring origo kan man generera med hjälp av Matlab-funktionen `randn`. Denna funktion skapar en matris med normalfördelade slumptal med medelvärdet noll och standardavvikelsen ett. Två klasser med hundra punkter vardera kan genereras med genom:

```
classA(1,:) = randn(1,100) .* 0.5 + 1.0;
classA(2,:) = randn(1,100) .* 0.5 + 0.5;
classB(1,:) = randn(1,100) .* 0.5 - 1.0;
classB(2,:) = randn(1,100) .* 0.5 + 0.0;
```

Vi har här skalat och translaterat punkterna så att klass A har sin mittpunkt vid $(1.0, 0.5)$ och klass B vid $(-1.0, 0.0)$. Båda har standardavvikelsen 0.5. Matlab-operatorn `.*` betecknar elementvis multiplikation (till skillnad från matrismultiplikation).

Nu återstår bara att sätta samman de två klasserna till en enda datamängd. Man kan utnyttja Matlabs matrisoperationer för att sätta ihop alla punkterna till en enda matris, `patterns`, med 200 kolumner:

```
patterns = [classA, classB];
```

Vi måste också skapa en matris `targets` med de rätta svaren. Låt klass A motsvaras av värdet 1 och klass B av värdet -1. Formulera själv en Matlab-sats som sätter samman en lämplig `targets`-matris. Tips: du kan utnytta Matlab-funktionen `ones`.

Det är kanske inte så naturligt att punkterna i datamängden är ordnade så att alla ur den ena klassen kommer först och de andra efteråt. Punkterna borde istället komma i slumpmässig ordning[2]. Detta åstadkommer man enkelt genom att utnyttja funktionen `randperm`, som skapar en slumpmässig permutationsvektor. Permutationsvektorn kan användas för att kasta om ordningen för kolumnerna i både `patterns` och `targets`:

```
permute = randperm(200);
patterns = patterns(:, permute);
targets = targets(:, permute);
```

[2]Vid batch-inlärning spelar det egentligen ingen roll vilken ordning mönstren har eftersom alla bidrag ändå summeras före viktuppdateringen, men vid sekvensiell inlärning kan ordningen påverka resultatet.

Om du vill kan du titta på hur dina datapunkter hamnade genom kommandot:

```
plot (patterns(1, find(targets>0)), ...
      patterns(2, find(targets>0)), '*', ...
      patterns(1, find(targets<0)), ...
      patterns(2, find(targets<0)), '+');
```

4.2 Implementera deltaregeln

Vi ska nu implementera själva deltaregeln. Skriv gärna dessa Matlab-satser i en egen fil, `delta.m`, så att du kan använda kommandot `delta` för att köra algoritmen om och om igen. Utgå ifrån att träningsdata finns i de globala variablerna `patterns` respektive `targets`. Det gör det enkelt att senare prova samma algoritm med andra träningsdata.

Själva deltaregeln kan skrivas som

$$\Delta w_{j,i} = -\eta x_i \left(\sum_k w_{j,k} x_k - t_j \right)$$

där $\bar{x}$ är inmönstret, $\bar{t}$ är det önskade utmönstret och $w_{j,i}$ är kopplingen från x_i till t_j. Detta kan vi skriva kompaktare på matrisform:

$$\Delta W = -\eta (W\bar{x} - \bar{t})\bar{x}^T$$

Ovanstående formel beskriver hur vikterna ska ändras, baserat på *ett* träningsmönster. För att få den totala ändringen från en hel epok ska bidragen från *alla* mönstren summeras. Eftersom vi lagrar mönstren som kolumner i X och T får vi denna summering "på köpet" när vi multiplicerar matriserna. Den totala viktändringen från en hel epok kan därför skrivas på detta kompakta sätt:

$$\Delta W = -\eta (WX - T)X^T$$

Skriv själv de Matlab-satser som implementerar denna beräkning. Lägg märke till att X inte direkt motsvarar innehållet i variabeln `patterns` eftersom denna inte innehåller den rad med ettor som krävs för att bias-termen ska komma med i beräkningen.

Skriv din kod så att inlärningen enligt formeln ovan upprepas `epochs` gånger (där ett lämpligt värde på `epochs` är 20). Se till att ditt program fungerar för godtyckliga storlekar av indata och utdatamönstren och antal träningsexempel. Steglängden η bör sättas till något lagom litet värde, t.ex. 0.001.

Tips: man kan enkelt plocka fram dimensionerna för de aktuella matriserna genom satserna:

```
[insize, ndata] = size(patterns);
[outsize, ndata] = size(targets);
```

Låt dock inte detta fresta dig att använda `for`-loopar för att gå igenom enskilda element i matriserna. Utnyttja Matlabs förmåga att hantera vektorer och matriser istället.

Innan själva inlärningen kan sätta igång måste vikterna ha något startvärde. Det normala är att man startar med små slumpmässiga värden. Hitta själv på de Matlab-satser som behövs för att skapa en initial viktmatris, t.ex. genom att utnyttja `randn`-funktionen. Tänk på att viktmatrisen måste ha rätt storlek.

Titta på separationslinjen under inlärningens gång

För att det ska vara någon mening med programmet du nu skrivit måste det kompletteras med satser som på något lämpligt sätt presenterar resultatet. I princip skulle man kunna studera värdena i W-matrisen efter inlärningen, men det är inte speciellt överskådligt. Ett bättre alternativ är att samla någon form av statistik över hur många mönster som klassas rätt.

Vi ska dock utnyttja det faktum att vi har tvådimentionella data. Detta gör det möjligt att grafiskt visa hur inlärningen lyckas placera separationslinjen mellan de två klasserna A och B. Följande Matlab-satser ritar ut separationslinjen samt själva datapunkterna i indatarummet:

```
p = w(1,1:2);
k = -w(1, insize+1) / (p*p');
l = sqrt(p*p');
plot (patterns(1, find(targets>0)), ...
      patterns(2, find(targets>0)), '*', ...
      patterns(1, find(targets<0)), ...
      patterns(2, find(targets<0)), '+', ...
      [p(1), p(1)]*k + [-p(2), p(2)]/l, ...
      [p(2), p(2)]*k + [p(1), -p(1)]/l, '-');
drawnow;
```

Dessa rader kan användas efter att inlärningen är klar, men det är roligare att lägga in dem i `for`-loopen för inlärningsepokerna så att man ser hur linjen succesivt flyttas medan inlärningen pågår. För att undvika att Matlab ändrar skalan medan "animeringen" pågår kan man sätta en fix skala med hjälp av:

```
axis ([-2, 2, -2, 2], 'square');
```

Sätt nu ihop ovanstående delar till ett fungernade program och provkör. Experimentera gärna med olika värden på η och `epochs`.

4.3 Icke linjärt separerbara data

Nu ska vi generera data för ett klassificeringsproblem som inte är fullt så enkelt. Följande Matlab-rader skapar två klasser som inte är linjärt separerbara.

```
classA(1,:) = [randn(1,50) .* 0.2 - 1.0, ...
               randn(1,50) .* 0.2 + 1.0];
classA(2,:) = randn(1,100) .* 0.2 + 0.3;
classB(1,:) = randn(1,100) .* 0.3 + 0.0;
classB(2,:) = randn(1,100) .* 0.3 - 0.1;
```

Skapa en ny fil `nsepdata.m` med dessa rader samt det som behövs för att få till variablerna `patterns` och `targets` igen. Prova vad som nu händer med deltaregeln.

Ett tips för fortsättningen är att rensa bort gamla variabler med Matlab-kommandot `clear` innan varje ny körning. Det gör det lättare att felsöka din kod.

5 Klassificering med tvålagersperceptroner

5.1 Implementera en tvålagersperceptron

Nu har vi kommit fram till den viktigaste delen av denna labb, nämligen att implementera den generaliserade deltaregeln, även kallad BackProp. Denna algoritm kommer du att använda för flera olika experiment så det lönar sig att göra det hela generellt. Se speciellt till att antalet noder i det gömda lagret enkelt kan varieras, t.ex. genom att ändra värdet på en global variabel. Låt även parametrar som steglängd och antalet iterationer styras på detta sätt.

För att ett flerlagersnät ska tillföra någonting extra är det nödvändigt att använda en icke-linjär överföringsfunktion, ofta betecknad φ. Normalt väljer man en funktion som har en derivata som är enkel att beräkna, t.ex.

$$\varphi(x) = \frac{2}{1 + e^{-x}} - 1$$

som har derivatan

$$\varphi'(x) = \frac{[1 + \varphi(x)][1 - \varphi(x)]}{2}$$

Lägg märke till att man gärna uttrycker derivatan som en funktion av $\varphi(x)$ eftersom denna storhet ändå måste beräknas.

I BackProp består varje epok av tre delar. Först utförs det s.k. framåtpasset då nodernas aktivitet beräknas, lager för lager. Därefter kommer bakåtpasset då en felsignal δ beräknas för varje nod. Eftersom δ-värdena beror av δ-värdena

i efterföljande lager måste denna beräkning utgå från utlagret och succesivt arbeta sig bakåt, lager för lager (därav namnet "backpropagation"). Slutligen sker själva viktuppdateringen. Vi börjar med att titta på hur framåtpasset kan implementeras.

5.1.1 Framåtpasset

Låt x_i beteckna aktivitetsnivån i nod i i inlagret och låt h_j vara aktiviteten i nod j i det gömda lagret. Utsignalen h_j blir nu

$$h_j = \varphi(h_j^*)$$

där h_j^* betecknar den summerade insignalen till nod j, d.v.s.

$$h_j^* = \sum_i w_{j,i} x_i$$

Därefter sker samma sak för nästa lager, vilket ger oss det slutliga utmönstret i form av vektorn $\bar{o}$.

$$o_k = \varphi(o_k^*)$$
$$o_k^* = \sum_j v_{k,j} h_j$$

Precis som för enlagersperceptronen kan dessa beräkningar med fördel skrivas på matrisform, vilket också innebär att beräkningen sker samtidigt över alla mönster.

$$H = \varphi(WX)$$
$$O = \varphi(VH)$$

Överföringsfunktionen φ ska här appliceras på alla element i matrisen, oberoende av varandra.

Vi har så här långt hoppat över en liten men viktig detalj, den s.k. *bias*-termen. För att algoritmen ska fungera krävs det att man lägger till en insignal i varje lager som alltid har värdet ett[3]. I vårt fall måste alltså matriserna X och H kompletteras med en rad med ettor på slutet.

I Matlab kan framåtpasset uttryckas såhär

```
hin = w * [patterns ; ones(1,ndata)];
hout = [2 ./ (1+exp(-hin)) - 1 ; ones(1,ndata)];

oin = v * hout;
out = 2 ./ (1+exp(-oin)) - 1;
```

[3]Vissa författare väljer att låta *bias*-termen stå utanför summan i formlerna, men det innebär att den måste specialbehandlas hela vägen genom algoritmen. Både formlerna och implementationen blir enklare ifall man hela tiden ser till att det tillkommer en extra insignal med värdet 1 i varje lager.

Här använder vi variablerna `hin` för H^*, `hout` för H, `oin` för O^* samt `out` för O. Observera användningen av Matlab-operatorn `./` som betecknar elementvis division (till skillnad från matrisdivision). Motsvarande operator `.*` har vi redan använt för att få elementvis multiplikation.

5.1.2 Bakåtpasset

Bakåtpasset går ut på att beräkna de generaliserade felsignaler δ som används för viktuppdateringarna. För utlagrets noder beräknas δ som felet i utdata multiplicerat med överförinsfunktionens derivata (φ'), alltså:

$$\delta_k^{(o)} = (o_k - t_k) \cdot \varphi'(o_k^*)$$

För att nu beräkna δ i nästa lager utnyttjar man de nyss beräknade $\delta^{(o)}$:

$$\delta_j^{(h)} = \left(\sum_k v_{k,j} \delta_k^{(o)} \right) \cdot \varphi'(h_j^*)$$

Låt oss genast uttrycka detta på matrisform

$$\delta^{(o)} = (O - T) \odot \varphi'(O^*)$$

$$\delta^{(h)} = (V^T \delta^{(o)}) \odot \varphi'(H^*)$$

(där $\odot$ betecknar elementvis multiplikation).

Nu kan vi uttrycka även detta som Matlab-kod

```
delta_o = (out - targets) .* ((1 + out) .* (1 - out)) * 0.5;
delta_h = (v' * delta_o) .* ((1 + hout) .* (1 - hout)) * 0.5;
delta_h = delta_h(1:hidden, :);
```

Den sista raden har bara till uppgift att ta bort den extra raden som vi lade till i framåtpasset för att ta hand om *bias*-termen. Vi har här förutsatt att variabeln `hidden` innehåller antalet noder i det gömda lagret.

5.1.3 Viktuppdateringen

Efter bakåtpasset är det dags att utföra själva viktuppdateringen. Grundformeln för viktuppdateringen är

$$\Delta w_{j,i} = -\eta x_i \delta_j^{(h)}$$

$$\Delta v_{k,j} = -\eta h_j \delta_k^{(o)}$$

vilket vi som vanligt snabbt gör om till matrisform

$$\Delta W = -\eta \delta^{(h)} X^T$$
$$\Delta V = -\eta \delta^{(o)} H^T$$

Innan vi implementerar detta som Matlab-kod ska vi dock se till att införa en förbättring av algoritmen som brukar kallas *tröghet* (eng. *momentum*). Det innebär att man inte uppdaterar vikterna direkt med ovanstående värden, utan med ett glidande medelvärde för att undvika alltför snabba svängningar. Detta gör att man kan använda större steg vilket i sin tur innebär att algoritmen konvergerar snabbare. En skalär faktor α styr hur mycket av den gamla förändringsvektorn man behåller. Ett lämpligt värde på α brukar vara t.ex. 0.9. Den nya uppdateringsregeln blir nu (på matrisform):

$$\Theta = \alpha\Theta - (1-\alpha)\delta^{(h)} X^T$$
$$\Psi = \alpha\Psi - (1-\alpha)\delta^{(o)} H^T$$
$$\Delta W = \eta\Theta$$
$$\Delta V = \eta\Psi$$

Samma sak uttryckt som Matlab-satser blir

```
dw = (dw .* alpha) - (delta_h * pat') .* (1-alpha);
dv = (dv .* alpha) - (delta_o * hout') .* (1-alpha);
w = w + dw .* eta;
v = v + dv .* eta;
```

Vi har nu gått igenom alla centrala delar av algoritmen. Vad som återstår är att sätta samman alltihop. Glöm inte att framåtpasset, bakåtpasset samt viktuppdateringen ska göras för varje epok. Använd lämpligen en `for`-loop för att succesivt få bättre och bättre vikter.

Ett speciellt problem är hur du ska kunna se vad algoritmen gör! Det är inte lika enkelt som för enlagersperceptronen att rita ut separationslinjen ens när algoritmen används för klassificering. Det bästa är nog att räkna antalet noder med fel klassning i alla mönster för varje epok och succesivt spara dessa värden i en vektor som man kan rita vid inlärningens slut. Tänk på att algoritmen, så som den nu har beskrivits, ger graderade värden i matrisen `out`. När man vill se hur nätet klassificerar indatamönstren bör man tröskla vid noll. Detta gör man enklast med Matlab-funktionen `sign`.

Det totala antalet felaktiga noder för alla mönster räknar man ut genom en dubbelsumma över alla element, t.ex. på detta sätt:

```
error(epoch) = sum(sum(abs(sign(out) - targets)./2));
```

Vi har här antagit att det finns en vektor `error` av tillräcklig längd där vi kan lagra resultatet.

5.2 Icke-linjärt separerbara data (igen)

Nu är vi redo att återgå till det tidigare problemet med icke-linjärt separerbara klasser. Kör den nya algoritmen och kontrollera att den klarar av att lösa problemet. Observera att det inte är säkert att *alla* mönster kommer att klassificeras rätt ifall man inte har väldigt många gömda noder. Datamängden är ju slumpmässigt genererad så det är troligt att några datapunkter hamnar väldigt snett. Undersök hur många noder som behövs i det gömda lagret.

5.3 Encoder-problemet

Encoder-problemet är ett klassiskt exempel på hur man kan tvinga ett nätverk att hitta en kompakt kodning för glesa data. Det går till så att man använder ett timglasformat nätverk, d.v.s. antalet noder i det gömda lagret är mycket mindre än in- och utlagret. Nätet tränas med identiska in- och utmönster vilket tvingar nätet att hitta en kompakt mellanrepresentation för det gömda lagret.

Vi ska här studera det klassiska 8–3–8 problemet. Man låter indata (och alltså även utdata) bestå av mönster där endast ett element av åtta är "på". Med representationen i intervallet $(-1, 1)$ innebär det mönster av typen

$$\begin{bmatrix} -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix}^T.$$

Det finns totalt åtta sådana mönster. Genom att låta det gömda lagret ha bara tre noder kan vi tvinga fram en representation där de åtta mönstren representeras med endast tre värden. Din uppgift här är att studera vilken typ av representation som uppstår i det gömda lagret.

Vi kan utnyttja Matlab-funktionen `eye`, som genererar en enhetsmatris, för att skapa våra mönster. Observera att in- och utmönstren ska vara identiska. Matlab-koden för att skapa mönstren blir helt enkelt:

```
patterns = eye(8) * 2 - 1;
targets = patterns;
```

Använd din implementation av generaliserade deltaregeln från ovan för att träna nätet tills klassningen blir korrekt. Lyckas nätet alltid med detta? Undersök sedan hur den interna koden blev. Detta gör man enklast genom att titta på viktmatrisen för det första lagret. Lägg särskilt märke till ifall det finns någon struktur i vikternas tecken.

6 Funktionsanpassning

Hittills har vi bara utnyttjat perceptronens förmåga att klassificera data men en minst lika viktig egenskap är förmågan att approximera en godtycklig kontinuerlig funktion. Här ska vi studera hur man kan lära ett tvålagersnät (d.v.s. ett nät med ett gömt lager) att approximera en funktion, given i form av exempel. För att göra det enkelt att titta på funktionen och dess approximation väljer vi en funktion av två variabler, med ett reellt värde som resultat.

6.1 Generera funktionsdata

Som exempelfunktion väljer vi den välkända Gauss-klockan[4]

$$f(x, y) = e^{-(x^2+y^2)/10} - 0.5$$

Följande Matlab-satser skapar argumentvektorerna x och y samt resultatmatrisen z.

```
x=[-5:1:5]';
y=x;
z=exp(-x.*x*0.1) * exp(-y.*y*0.1)' - 0.5;
```

Vi kan använda Matlab-funktionen `mesh` för att se hur funktionen ser ut:

```
mesh (x, y, z);
```

Den lagringsform vi nu har för in- och utdata passar utmärkt för att visa grafiskt, men för att kunna använda mönstren som träningsdata måste de formas om till mönstermatriser med rätt form. Här har man nytta av Matlabfunktionerna `reshape` och `meshgrid`. Följande satser sätter samman de två matriserna `patterns` och `targets` som vi behöver för träningen.

```
targets = reshape (z, 1, ndata);
[xx, yy] = meshgrid (x, y);
patterns = [reshape(xx, 1, ndata); reshape(yy, 1, ndata)];
```

(`ndata` är här antalet mönster, d.v.s. produkten av antalet element i x och i y).

[4] Vi låter funktionen variera mellan -0.5 och $+0.5$ för att vara säkra på att noden i utlagret kan producera de värden som behövs.

6.2 Rita resultatet

Genom att göra motsvarande transformation åt andra hållet kan man titta på nätets approximation grafiskt. Satserna för att att göra detta blir

```
zz = reshape(out, gridsize, gridsize);
mesh(x,y,zz);
axis([-5 5 -5 5 -0.7 0.7]);
drawnow;
```

Här har vi antagit att `gridsize` är antalet element i x eller y (d.v.s. `ndata` = `gridsize`$\cdot$`gridsize`). Variablen `out` är resultatet från framåtpasset i nätet när alla träningsdata presenteras som indata.

Funktionen `drawnow` används för att tvinga Matlab att visa diagrammet trots att beräkningarna inte är slutförda. Detta innebär att vi kan stoppa in ovanstående rader i den loop som används för att succesivt förändra vikterna och på så sätt få en animering av själva inlärningen.

6.3 Träna nätet

Sätt nu samman alla delar så att du får ett program som genererar funktionsdata och sedan under inlärningens gång efter varje epok visar hur funktionsapproximationen ser ut. När allt fungerar bör du se en animerad funktion som succesivt blir mer och mer lik en Gauss-klocka.

Experimentera med olika många noder i det gömda lagret för att få en uppfattning om hur det påverkar den slutliga representationen.

7 Generalisering

En viktig egenskap för neurala nätverk är deras förmåga att generalisera, d.v.s. att ge vettiga resultat även för indata som inte har förekommit under träningen. Vi ska här modifiera experimentet ovan genom att endast träna nätet med ett begränsat urval datapunkter men fortfarande titta på funktionsapproximationen i alla punkter som tidigare.

För att åstadkomma detta kan man göra en slumpmässig permutation av vektorerna `patterns` och `targets` och endast träna med de n första mönstren (för olika värden på n). Programmet kommer då att behöva göra två olika framåtpass; ett för träningspunkterna och ett för samtliga punkter. Endast det första passet är kopplat till viktuppdateringen. Resultatet av det andra används

för att se hur väl nätverket generaliserar. I vårt fall innebär en god generalisering att nätverket återskapar hela funktionen, trots att det endast har fått "se" några få punkter.

Prova med $n = 10$ och $n = 25$. Variera antalet noder i det gömda lagret och försök se någon tendens. Vad händer när du har väldigt få gömda noder (färre än 5)? Vad händer när du har många (fler än 20)? Kan du förklara dina iakttagelser?