Die gesamte individuelle Lernprozedur besteht aus den folgenden Verarbeitungsschritten:

- Wähle vorläufige Werte für die Koeffizientenmatrizen $\mathbf{W}^1, \mathbf{W}^2, \ldots, \mathbf{W}^H$ aller Schichten
- Eine neue Beobachtung $[\mathbf{x}'_j, \hat{\mathbf{y}}_j]$ sei gegeben
- Berechne die Diskriminierungsfunktion $\hat{\mathbf{y}}_j$ aus der gegebenen Beobachtung $\mathbf{x}'_j$ und den momentanen Gewichtsmatrizen (Vorwärtsrechnung)
- Berechne den Fehler zwischen Schätzung $\hat{\mathbf{y}}$ und Zielvektor $\mathbf{y}$:

$$\Delta\mathbf{y} = \hat{\mathbf{y}} - \mathbf{y}$$

- Berechne den Gradienten $\nabla\mathbf{J}$ bzgl. aller Perceptron-Gewichte (error backpropagation). Berechne dazu zuerst δ_j^H der Ausgangsschicht gemäß:

$$\delta_j^H = -\frac{\partial J}{\partial s_n^H} = -\frac{\partial J}{\partial \hat{y}_n^H} \cdot \frac{\partial \hat{y}_n^H}{\partial s_n^H} = (y_j - \hat{y}_j) f'(s_j^H)$$

- und berechne daraus rückwärts alle Werte der niederen Schichten gemäß:

$$\delta_j^{r-1} = f'(s_j^{r-1}) \left[\sum_k w'^r_{kj} \delta_m^k \right] \quad \text{für} \quad r = H, H-1, \ldots, 2$$

- unter Beachtung der ersten Ableitung der Sigmoidfunktion $f(x) = \psi(x)$:

$$f'(x) = \frac{d\psi}{dx} = \psi(x)(1 - \psi(x))$$

- Korrigiere (parallel) alle Perceptron-Gewichte gemäß:

$$w'^r_{nm} \leftarrow w'^r_{nm} - \alpha \frac{\partial J}{\partial w'^r_{nm}} = w'^r_{nm} - \alpha \delta_n^r y_m^{r-1} \quad \text{für} \quad \begin{array}{l} h = 1, 2, \ldots H \\ n = 1, 2, \ldots N^H \\ m = 1, 2, \ldots M^H \end{array}$$

- Beim individuellen Lernen werden die Gewichte $\{w'^h_{nm}\}$ mit $\nabla\mathbf{J}$ für jede Stichprobe korrigiert, wohingegen beim kumulativen Lernen die gesamte Lernstichprobe durchgearbeitet werden muss, um den gemittelten Gradienten $\nabla\mathbf{J}$ aus der Sequenz der $\{\nabla\mathbf{J}\}$ zu ermitteln, bevor die Gewichte korrigiert werden können gemäß:

$$w'^r_{nm} \leftarrow w'^r_{nm} - \sum_i \alpha \delta_n^r(i) y_m^{r-1}(i) \quad \text{für} \quad \begin{array}{l} h = 1, 2, \ldots H \\ n = 1, 2, \ldots N^H \\ m = 1, 2, \ldots M^H \end{array}$$

Eigenschaften:

- Der Backpropagation-Algorithmus ist einfach zu implementieren, aber sehr rechenaufwendig, insbesondere wenn die Koeffizientenmatrix groß ist, was zur Folge hat, dass auch die Lernstichprobe entsprechend groß sein muss. Nachteilig ist weiterhin die Abhängigkeit des Verfahrens von den Startwerten der Gewichte, dem Korrekturfaktor α und die Reihenfolge, in der die Stichproben abgearbeitet werden.
- Wie beim rekursiven Trainieren des Polynomklassifikators bleiben lineare Abhängigkeiten in den Merkmalen unberücksichtigt.
- Positiv zu vermerken ist, das der Gradientenalgorithmus auch für sehr große Probleme verwendet werden kann.
- Die Dimension der Koeffizientenmatrix **W** ergibt sich aus den Dimensionen des Eingangsvektors, der verdeckten Schichten, sowie des Ausgangsvektors $(N,N^1,...,N^{H-1},K)$ zu:

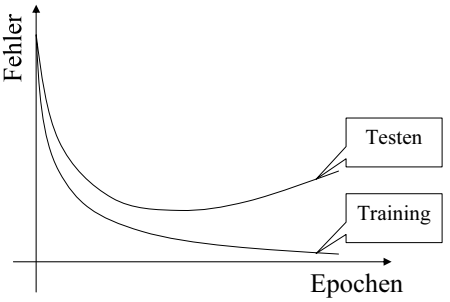
$$T = \dim(\mathbf{W}) = \sum_{h=1}^{H} (N^{h-1}+1)N^h \quad \text{mit } N^0 = N \text{ und } N^H = K$$

$N = \dim(\mathbf{x})$ Merkmalsraum

$K = \dim(\hat{\mathbf{y}})$ Anzahl der Klassen

Zur Dimensionierung des Netzes

- Die Überlegungen bei dem Entwurf eines mehrschichtigen Perceptrons mit Schwellwertfunktionen (σ bzw. sign) geben eine gute Vorstellung, wieviele Hidden-Layer und wieviele Neuronen man für ein MLPC verwenden sollte für das Backpropagation-Lernen (vorausgestzt, man hat eine gewisse Vorstellung über die Verteilung der Cluster).
- Das Netz sollte so einfach wie nur möglich gewählt werden. Mit höherer Dimension steigt die Gefahr des overfitting, gepaart mit einem Verlust an Generalisierungsfähigkeit und zugleich werden viele Nebenmaxima zugelassen, wo der Algorithmus hängen bleiben kann!
- Bei einem vorgegebenen Stichprobenumfang, sollte die Lernphase bei einem bestimmten Punkt abgebrochen werden. Danach wird das Netz zu sehr an die vorhandenen Daten angepasst (overfitting) und verliert an Generalisierungsfähigkeit.



Zur Berechnungskomplexität des Backpropagation-Algorithmus

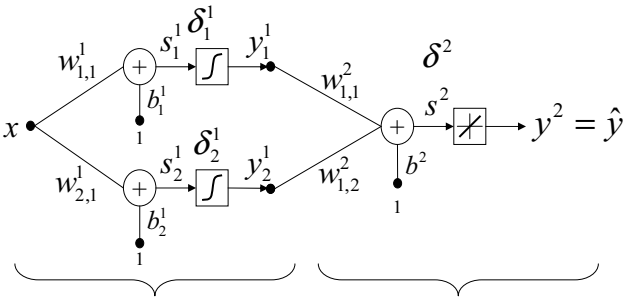
- Wenn T=dim(**W**) die Anzahl der Gewichte und Biasterme ist, so läßt sich einfach nachvollziehen, daß *O(T)* Berechnungsschritte für die Vorwärtssimulation benötigt werden, *O(T)* für das Backpropagieren des Fehlers und ebenso *O(T)* Operationen für die Korrektur der Gewichte, also erhält man insgesamt einen lineare Komplexität in der Anzahl der Gewichte: *O(T)*.
- Würde man den Gradienten durch finite Differenzen experimentell bestimmen (dazu ändert man jedes Gewicht inkrementell und ermittelt die Wirkung auf das Gütemaß durch Vorwärtsrechnung) durch Berechnung eines Differenzenquotienten gemäß:

$$\frac{\partial J}{\partial w_{ji}^r} = \frac{J(w_{ji}^r + \varepsilon) - J(w_{ji}^r)}{\varepsilon} + O(\varepsilon)$$

- so ergäben sich T Vorwärtsrechnungen der Komplexität *O(T)* und damit insgesamt eine Gesamtkomplexität von: $O(T^2)$

Backpropagation zum Trainieren eines zweischichtigen Netzwerks zur Funktionsapproximation

(Matlab-Demo: „Backpropagation Calculation)



$$\mathbf{y}^1 = \boldsymbol{\psi}(\mathbf{W}^1 x + \mathbf{b}^1) = \boldsymbol{\psi}(\mathbf{W}'^1 \mathbf{x}'^1) = \boldsymbol{\psi}(\mathbf{s}^1) \qquad \hat{y} = y^2 = (\mathbf{W}^2 \mathbf{y}^1 + b^2)$$

mit: $\mathbf{W}^1 = \begin{bmatrix} w_{1,1}^1 \\ w_{2,1}^1 \end{bmatrix}$ und $\mathbf{b}^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$ mit: $\mathbf{W}^2 = \begin{bmatrix} w_{1,1}^2 & w_{1,2}^2 \end{bmatrix}$

- Gesucht wird eine Approximation der Funktion

$$y(x) = 1 + \sin(\tfrac{\pi}{4} x) \quad \text{für} \quad -2 \le x \le 2$$

Backpropagation zum Trainieren eines zweischichtigen Netzwerks zur Funktionsapproximation

- Backpropagation: Für die Ausgangsschicht oder zweite Schicht mit linearer Aktivierungsfunktion ergibt sich wegen $f'=1$:

$$\delta^2 = (y - \hat{y})$$

- Backpropagation: Für die erste Schicht mit Sigmoid-Funktion ergibt sich: :

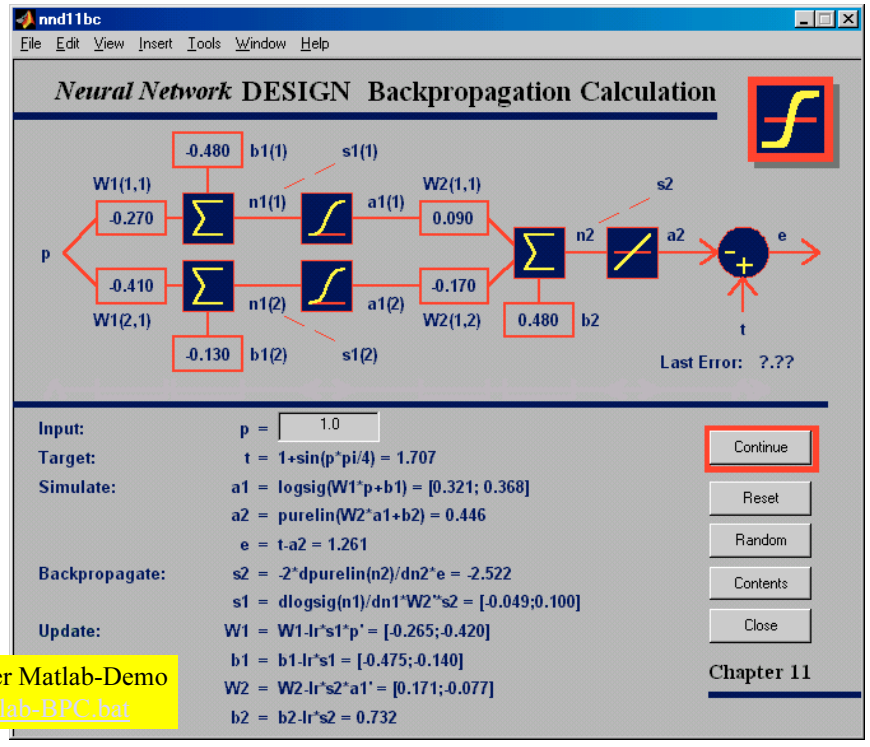
$$\delta_j^1 = f'(s_j^1)\left[w_{1,j}^{\prime 2}\delta^2\right] = s_j^1(1-s_j^1)\left[w_{1,j}^2\delta^2\right] \quad \text{für} \quad j=1,2$$

- Korrektur der Gewichte in der Ausgangsschicht:

$$\Delta w_{1,j}^2 = \alpha\delta^2 y_j^1 \quad \text{und} \quad \Delta b^2 = \alpha\delta^2 \quad \text{für} \quad j=1,2$$

- Korrektur der Gewichte in der Eingangsschicht:

$$\Delta w_{j,1}^1 = \alpha\delta_j^1 x \quad \text{und} \quad \Delta b_j^1 = \alpha\delta_j^1 \quad \text{für} \quad j=1,2$$



Start der Matlab-Demo
matlab-BPC.bat

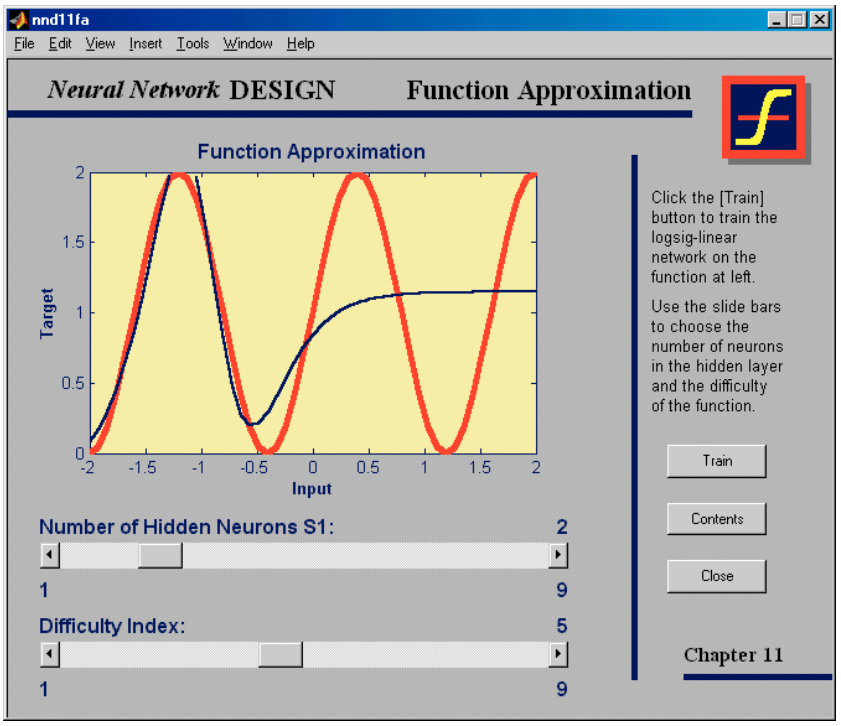
Backpropagation zum Trainieren eines zweischichtigen Netzwerks zur Funktionsapproximation

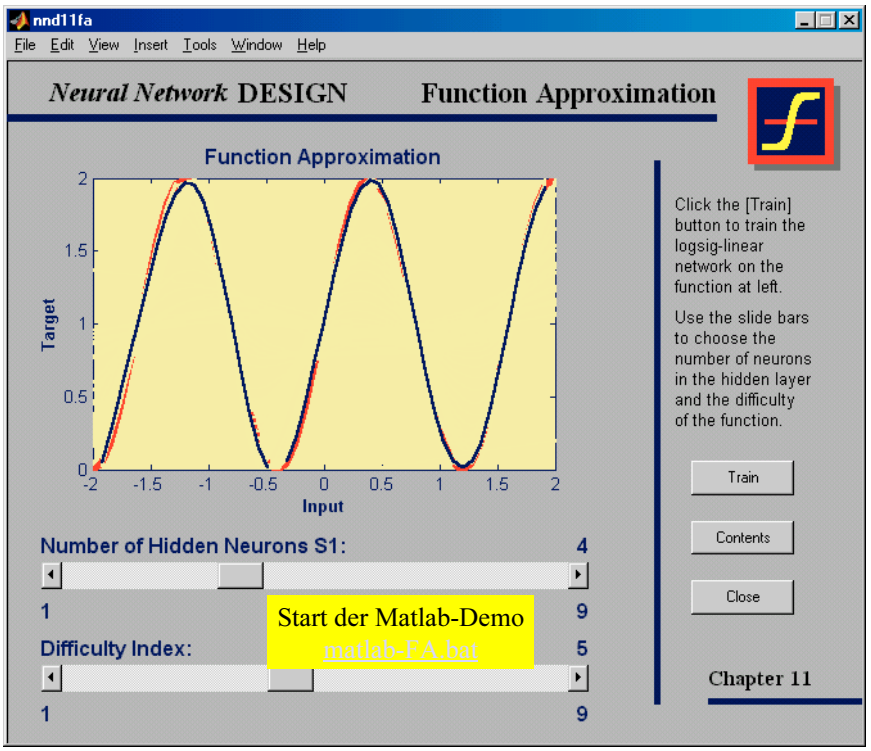
(Matlab-Demo: „Function Approximation)

- Gesucht wird eine Approximation der Funktion

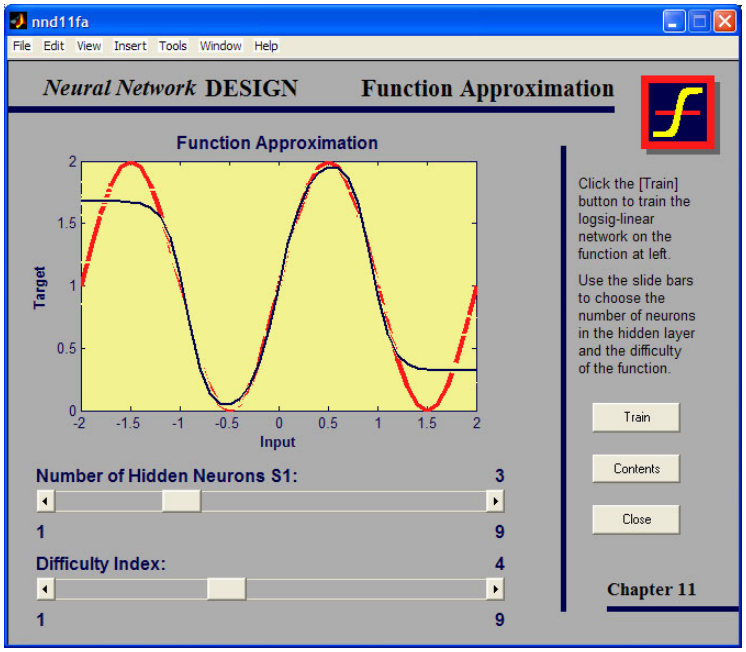
$$y(x) = 1 + \sin(i \cdot \tfrac{\pi}{4} x) \quad \text{für} \quad -2 \le x \le 2$$

- Mit zunehmenden Wert von i (difficulty index) steigen die Anforderungen an das MLPC-Netzwerk. Die auf der Sigmoid-Funktion aufbauende Approximation benötigt mehr und mehr Schichten, um mehrere Perioden der Sinus-Funktion darzustellen.
- Problem: Konvergenz zu lokalen Minima, selbst in Fällen, wo das Netzwerk groß genug gewählt wird, um die Funktion
  - Das Netzwerk wird zu klein gewählt, so dass eine Approximation nur schlecht gelingt, d.h. es wird zwar das globale Minimum erreicht, aber dieses liefert noch keine gute Approximationsgüte (i=8 und Netzwerk 1-3-1)
  - Das Netzwerk wird groß genug gewählt, so dass eine gute Approximation möglich ist, aber es konvergiert nur gegen ein lokales Minimum (i=4 und Netzwerk 1-3-1)





Modell groß genug, aber nur lokales Minimum



Generalisierungsfähigkeit des Netzwerkes

- Wir gehen davon aus, dass das Netzwerk für 11 Abtastpunkte trainiert wird

$$\{y_1, x_1\}, \{y_2, x_2\}, \ldots \{y_{11}, x_{11}\}$$

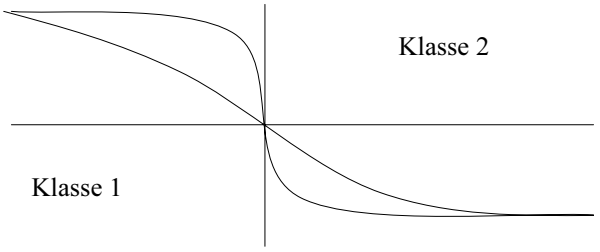
- Die Frage ist nun, wie gut das Netzwerk die Funktion für nicht gelernte Abtastpunkte approximiert in Abhängigkeit von der Komplexität des Netzwerkes

Start der Matlab-Demo (Hagan 11-21)
matlab-GENERAL.bat

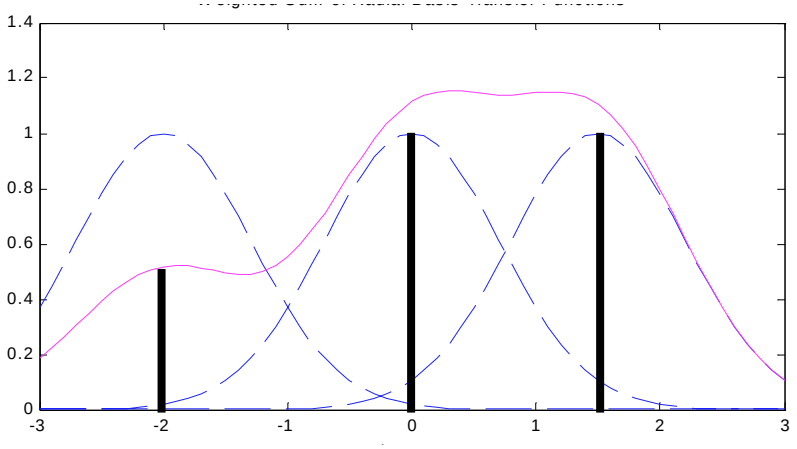
- Regel: Das Netzwerk sollte weniger Parameter haben als die zur Verfügung stehenden Ein-/Ausgangspaare

Verbesserung der Generalisierungsfähigkeit eines Netzes durch Hinzufügen additiver Störungen

- Stehen nur wenige Stichproben zur Verfügung, so kann die Generalisierungs-fähigkeit eines NN verbessert werden, indem man den Stichprobenumfang durch z.Bsp. normalverteilte Störungen verbreitert. D.h. man fügt weitere Stichproben hinzu, welche mit hoher Wahrscheinlichkeit in der unmittelbaren Nachbarschaft anzutreffen sind.
- Dadurch werden die Intraklassenbereiche verbreitert und die Grenzen zwischen den Klassen schärfer ausgebildet.

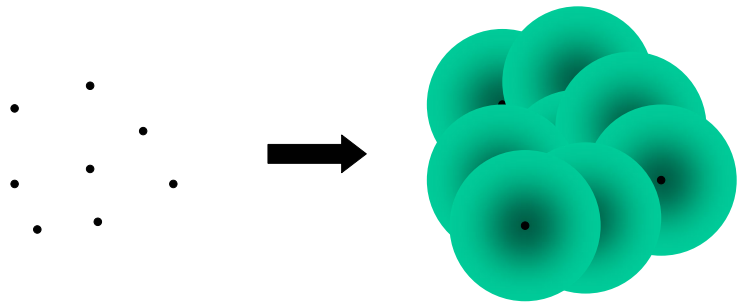


Interpolation durch Überlagerung von Normalverteilungen

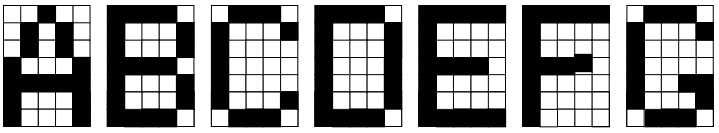


$$y(x) = \sum \alpha_i f_i(x - t_i)$$

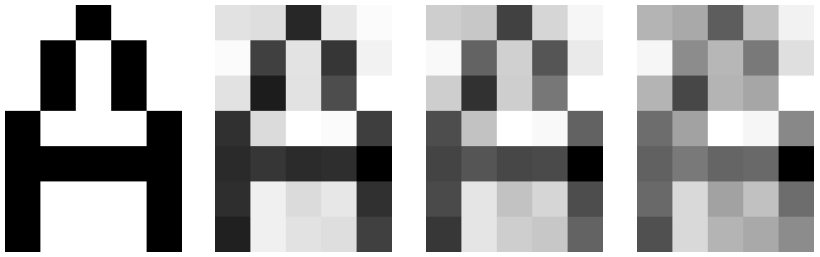
Interpolation durch Überlagerung von Normalverteilungen



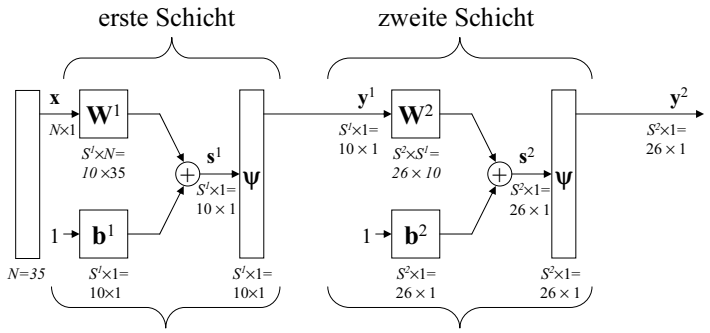
Zeichenerkennungsaufgabe für 26 Buchstaben der Größe 7×5



- mit graduell ansteigenden additiven Störungen:



Zweischichtiges Neuronales Netz für die Zeichenerkennung mit 35 Eingangswerten (Pixel), 10 Neuronen in der verdeckten Schicht und 26 Neuronen in der Ausgangsschicht



$\mathbf{y}^1=\mathbf{f}^1(\mathbf{W}^1\mathbf{x}+\mathbf{b}^1)$ $\mathbf{y}^2=\mathbf{f}^2(\mathbf{W}^2\mathbf{y}^1+\mathbf{b}^2)$

$$\mathbf{y}^2= \psi(W^2\, \psi(W^1 x+\mathbf{b}^1)+\mathbf{b}^2)$$

Demo mit MATLAB

- Öffnen von Matlab
  - Demo in Toolbox Neural Networks
  - „Character recognition“ (command line)
  - Es werden zwei Netzwerke trainiert
    - Netzwerk 1 ohne Störungen
    - Netzwerk 2 mit Störungen
  - Auf einem unabhängigen Testdatensatz liefert Netzwerk 2 bessere Ergebnisse als Netzwerk 1

Start der Matlab-Demo
matlab-CR.bat

Möglichkeiten zur Beschleunigung der Adaption

Es handelt sich bei der Adaption von NN um ein allgemeines Parameteroptimierungsproblem. Demgemäss können im Prinzip eine Vielzahl weiterer Optimierungsmethoden aus der numerischen Mathematik eingesetzt werden. Dies kann zu erheblichen Verbesserungen führen (Konvergenzgeschwindigkeit, Konvergenzbereich, Stabilität, Berechnungsaufwand).

I.a. lassen sich jedoch nur sehr schwer allgemeingültige Aussagen machen, da die Ergebnisse von Fall zu Fall sehr verschieden sein können und in der Regel Kompromisse in den Eigenschaften zu akzeptieren sind!

- Heuristische Verbesserungen des Gradientenalgorithmus (Schrittweitenkontrolle)
  - Gradientenalgorithmus (Steepest descent) mit Momentum-Term (update der Gewichte nicht nur abhängig vom Gradient, sondern auch vom vorherigen update)
  - Verwendung eines adaptiven Lernfaktors
- Konjugierter Gradientenalgorithmus

- Newton und Quasi-Newton-Algorithmen (Verwendung der zweiten Ableitungen der Fehlerfunktion z.B. in Form der Hesse-Matrix oder deren Schätzung)
  - Quickprop
  - Levenberg-Marquardt-Algorithmus

Taylorentwicklung der Gütefunktion in $\mathbf{w}$:

$$J(\mathbf{w}+\Delta\mathbf{w}) = J(\mathbf{w}) + \nabla\mathbf{J}^T\Delta\mathbf{w} + \tfrac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} + \text{Terme höherer Ordnung}$$

mit: $\nabla\mathbf{J} = \dfrac{\partial J}{\partial \mathbf{w}}$ Gradientenvektor

und: $\mathbf{H} = \left\{ \dfrac{\partial^2 J}{\partial w_i \partial w_j} \right\}$ Hesse-Matrix

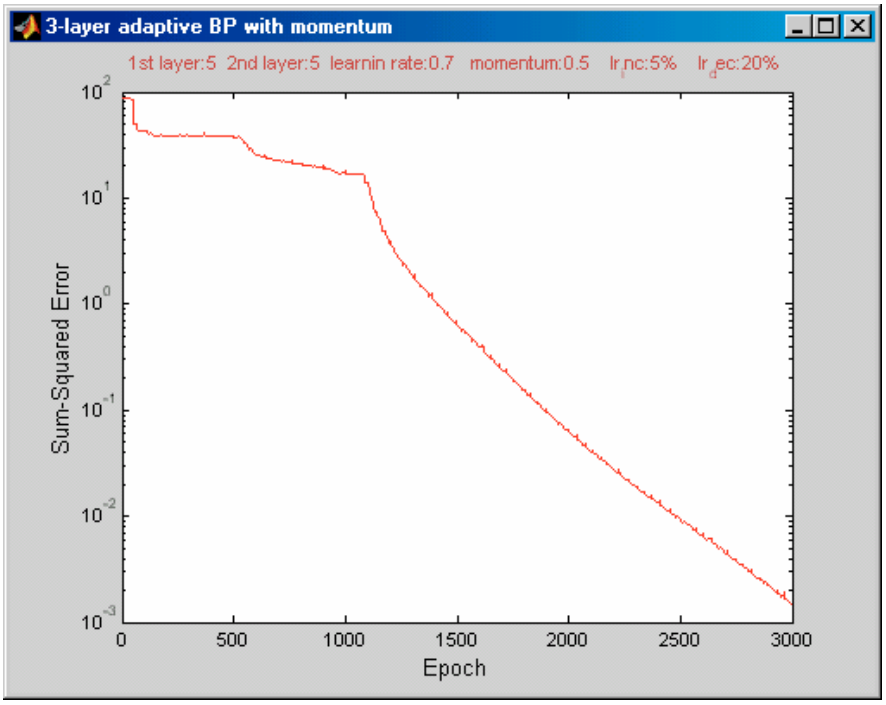
- Pruning-Techniken. Man startet mit einem hinreichend großen Netzwerk und nimmt dann wieder Neuronen aus dem Netz, welche keinen oder nur geringen Einfluß auf die Gütefunktion haben und reduziert damit das Overfitting der Daten.

Demo mit MATLAB

(Demo von Theodoridis)

- Öffnen von Matlab
  - C:\...\matlab\PR\startdemo
  - Example 3 mit dreischichtigem Netzwerk [5,5] (3000 Epochen, learning rate 0,7, momentum 0,5)

Start der Matlab-Demo
matlab-theodoridis.bat


3-layer adaptive BP with momentum
1st layer:5 2nd layer:5 learnin rate:0.7 momentum:0.5 lr_inc:5% lr_dec:20%
Sum-Squared Error
Epoch
10^2
10^1
10^0
10^-1
10^-2
10^-3
0
500
1000
1500
2000
2500
3000