DSP 2 Labortagebuch

Inhalt

Übung 1

1. Farbräume

Die wichtigsten Farbräume sind RGB, YUV, HSV, CMYK und LAB.

2. Farbbilddarstellung in Matlab

a. Import eines Bildes und Betrachtung von Bildvariablen

Mit der Funktion imread kann in Matlab ein Bild eingelesen werden. Als Rückgabewert erhält man ein mehrdimensionales Array mit Integerwerten (Auflösung x 3 x 8 Bit). Ein Pixel enthält 3 Farbkomponenten (R, G, B) à 8 Bit, das zurückgegebene Array ergibt sich also dimensionsmäßig durch horizontale x vertikale Auflösung x 3 x 8 Bit.
Wählt man im Workspace die Variable aus, in die das Bild eingelesen wurde und wählt im Kontextmenü „imagesec" wird das Bild dargestellt.

b. Auftrennung der Farbinformationen im Bild nach R, G und B

Über die einzelnen Elemente des mehrdimensionalen Arrays (siehe a.) kann auf die R-, G- und B-Komponenten zugegriffen werden. Die dritte Dimension gibt den Farbanteil (R, G oder B) an.

```
img = imread('C:\Dokumente und Einstellungen\All Users\Dokumente\Eigene
Bilder\Beispielbilder\Sonnenuntergang.jpg');
r = img(:,:,1);
g = img(:,:,2);
b = img(:,:,3);
```

Durch die Doppelpunkte werden alle Bildpunkte erfasst. Der Index für R, G und B ergibt sich durch die Reihenfolge, in der Matlab das Array befüllt (vgl. a.).

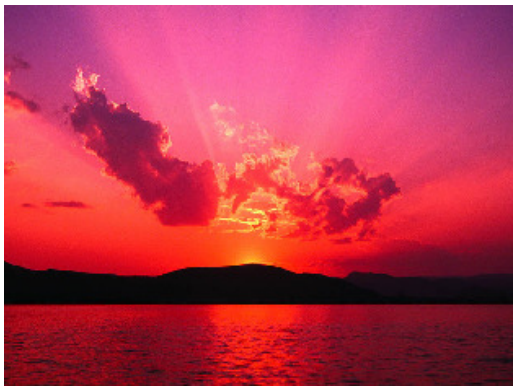
c. Graustufeneffekt bei der Darstellung der Farbanteile

Der Plot der einzelnen Farbanteile (r, g, b) wird nur schwarzweiß angezeigt, da die Werte in der Matrix nur zwischen 0 und 255 (jeder Anteil separat) sind – Matlab interpretiert diese Werte zwischen 0 und 255 als Graustufen.

```
subplot(221); imshow(r);
subplot(222); imshow(g);
subplot(223); imshow(b);
subplot(224); imshow(img);
```

Die Plots zeigen die R-, G- und B-Anteile des in b. geladenen Bildes Wie bereits erwähnt werden die Bilder nur in Graustufen angezeigt.



d. Richtiges Einfärben der extrahierten Farbanteile

Möchte man die Bilder, die die einzelnen Farbanteile zeigen, korrekt einfärben, muss eine neue Matrix/ein neues Bild erzeugt werden, die/das z.B. den roten Farbanteil (8 Bit) und zwei Mal 0 (auch jeweils wieder 8 Bit) enthält. Für die Nullmatrizen/-vektoren wird ein „leerer" Vektor A benötigt, der mit Nullen gefüllt wird (Funktion zeros). Mit der Funktion cat kann der passende Integerwert „erzeugt" werden, indem der Farbanteil an der korrekten Position (R an erster Stelle, G an zweiter, B an dritter) eingefügt und der Rest mit Nullen aus dem Nullvektor aufgefüllt wird.

```
A = uint8(zeros(600, 800));
R = cat(3, r, A, A);
G = cat(3, A, g, A);
B = cat(3, A, A, b);

subplot(221); imshow(R);
subplot(222); imshow(G);
subplot(223); imshow(B);
subplot(224); imshow(img);
```

Wird die zusammengesetzte Matrix dargestellt, wird der Rotanteil des Bildes korrekt angezeigt. Im folgenden (v.o.n.u, v.l.n.r: R-, G-, B- Anteil, Original)




3. 2D-Filter

a. Zweck von Kantenfiltern in der digitalen Bildverarbeitung

Kantenfilter dienen zur Kantenextraktion bzw. zum Erkennen von Objekten.

i. Kantenfilter

Die wichtigsten Kantenfilter sind (richtungsunabhängig) Gauß- und Laplace- sowie (richtungsabhängig) Sobel- und Prewitt-Filter.

ii. Kantenerkennung an einem einfärbigem Rechteck

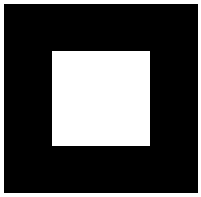
Zur Demonstration wird ein Testbild generiert, das komplett schwarz ist und ein weißes Quadrat in der Mitte beherbergt. Dazu wird ein Array mit Nullen aufgefüllt und in der Mitte (Pixel 6-15) auf eins gesetzt.
Anschließend wird über alle Pixel iteriert und ein horizontales Sobelfilter

$$s = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

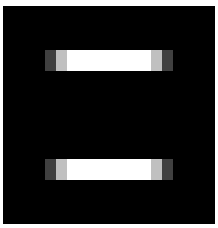
zur Extraktion der horizontal liegenden Kanten angewandt. Dies funktioniert wie folgt: die umliegenden Pixel werden mit den jeweiligen Koeffizienten des Sobelfilters multipliziert und schließlich aufaddiert. Der erhaltene Wert ergibt dann den gefilterten Pixelwert. Zu beachten ist, dass durch die negativen Koeffizienten negative Pixelwerte entstehen können. Dies kann durch das Bilden der Absolutwerte korrigiert werden. Durch die beiden Zweien in den Filterkoeffizienten des Sobelfilters können im Extremfall Werte zwischen -4 und 4 als Summe entstehen. Durch eine Division durch 4

können die Werte wieder in den ursprünglichen Wertebereich gebracht werden (vorherige Anwendung der Absolutfunktion vorausgesetzt). Der resultierende Wert kann als Grauwert interpretiert werden und zeigt an den Kanten weiße Linien:

```
A = zeros(20);
A(6:15, 6:15) = ones(10);
imshow(A);
B = A;
```

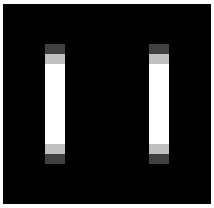


```
sobel_x = [1, 2, 1 ; 0, 0, 0 ; -1, -2, -1];
sobel_y = sobel_x';

for x = 2:19
    for y = 2:19
            sum = 0;
            for i = -1:1
                for j = -1:1
                    sum = sum + A(x + i,y + j) * sobel_x(i + 2, j + 2);
                end
            end
            B(x, y) = abs(sum) / 4;
    end
end
pause;
imshow(B);
```



Anmerkung: die Bilder wurden zur besseren Erkennbarkeit vergrößert. Da beim Filtern die Pixel rund um den zu filternden Bildpunkt berücksichtigt werden müssen, laufen i und j von -1 bis 1. Um die Indizes für die Sobelmatrix verwenden zu können muss daher 2 addiert werden (Indizes in Matrix zwischen 1 und 3). Zudem wird das Filter an den äußersten Bildpunkten (1 Pixel-Rand) angewandt, da keine Bildpunkte außerhalb des Bildes verfügbar sind und nicht extrapoliert wird.
Wird das Bild mit sobel_y anstatt mit sobel_x gefiltert (vertikales Sobelfilter), werden die vertikalen Kanten sichtbar (Anweisung in innerster for-Schleife ersetzen):

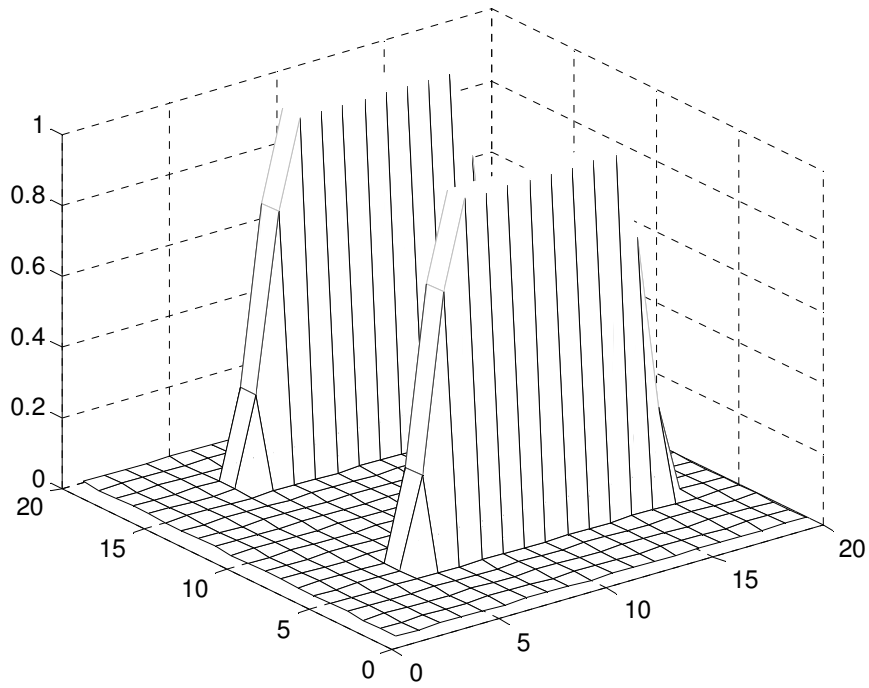
```
sum = sum + A(x + i,y + j) * sobel_y(i + 2, j + 2);
```



Durch die Funktion mesh können die Grauwerte des Bildes als dreidimensionale Funktion dargestellt werden, was die Kanten sehr schön zeigt. Die Übergänge zeigen Werte zwischen weiß und schwarz, die Ränder an diesen Stellen enden und das Filter durch

seine unterschiedliche Koeffizientengewichtung eine der Farben (schwarz/weiß) mehr bevorzugt, was zu einem dunkleren oder helleren Grauwert führt.

```
mesh(B);
```

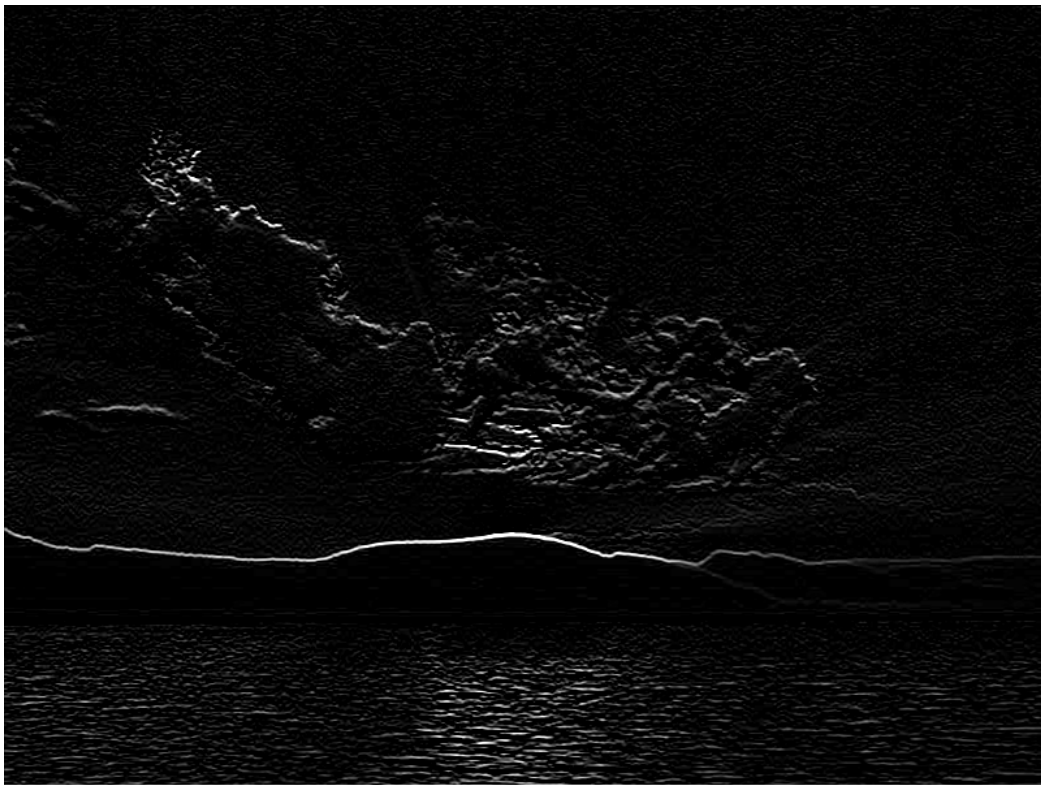


iii. Evaluierung der Matlabfunktionen imfilter und fspecial

Die Funktion imfilter kann ein Filter (mehrdimensionales Koeffizientenarray) auf ein Bild anwenden. Es kann beispielsweise das Sobelfilter aus ii. auf das Beispielbild anwenden und liefert dessen horizontale (horizontales Sobelfilter) bzw. vertikale (vertikales Sobelfilter) Kanten. Zusätzlich können Parameter angegeben werden, die das Filterverhalten an den Rändern definieren.
Zur Demonstration der Funktion imfilter wird das aus 1. bekannte Bild verwendet und durch ein Sobelfilter gefiltert. Um das Sobelfilter anwenden zu können, muss das Bild in Graustufen umgerechnet werden, da die einzelnen Farbanteile durch imfilter nicht berücksichtigt werden können. Das erledigt die Funktion rgb2gray.

```
A = imread('C:\Dokumente und Einstellungen\All Users\Dokumente\Eigene
Bilder\Beispielbilder\Sonnenuntergang.jpg');
A = rgb2gray(A);
imshow(A);
```

```
B = A;

sobel_x = [1, 2, 1 ; 0, 0, 0 ; -1, -2, -1];
sobel_y = sobel_x';

B = imfilter(A, sobel_x);

pause;
imshow(B);
```



```
B = imfilter(A, sobel_y);

pause;
imshow(B);
```

Das obere Bild zeigt ein horizontales, das untere ein vertikales Sobelfilter.
Die Funktion fspecial erzeugt eine Koeffizientenmatrix für den angegebenen Filternamen. Anwendungsbeispiele siehe b.

b. Filter zur Bildkorrektur

Unsharp-Filter können zur Bildschärfung, Medianfilter zur Entfernung von Peaks verwendet werden. Die Matlabfunktion fspecial bietet die Möglichkeit, diese Filter zu „erzeugen".

i. Verrauschte Bilder entrauschen

Mit imnoise kann ein Bild verrauscht werden. Dabei können verschiedene Rauscharten als Parameter angegeben werden. Die mit fspecial „erzeugten" Filter (vgl. b.) können dabei zum Entrauschen verwendet werden.
Aus Zeitgründen konnte nur versucht werden, mit der Rauschart „salt & pepper" verrauschte Bilder zu entrauschen. Das Originalbild ist in 3.a.iii zu sehen.

```
A = imread('C:\Dokumente und Einstellungen\All Users\Dokumente\Eigene
Bilder\Beispielbilder\Sonnenuntergang.jpg');
A = rgb2gray(A);

B = imnoise(A, 'salt & pepper');
median = fspecial('average', [2, 2]);
imshow(B);

B = imfilter(B, median);
imshow(B);
```

Wie man erkennen kann, sorgt der angewandte avarage-Filter dafür, dass das Bild etwas unscharf wird – das Rauschen kann nur verkleinert, nicht aber entfernt werden. Die Anwendung eines echten 2D-Medianfilters verbessert das Resultat erheblich – das Bild sieht fast exakt wie das Original aus:

```
B = medfilt2(B);
imshow(B);
```

Möchte man das Original mit dem gefilterten Bild vergleichen, kann man die beiden Bilder voneinander subtrahieren (Matrizensubtraktion) und erhält das Differenzbild:

```
C = A - B;
imshow(C);
```

Man erkennt im Bild kaum andere Werte als schwarz. Bei genauerem Hinsehen fallen bei den feinen Kanten Abweichungen auf – diese sind dadurch zu erklären, dass das Medianfilter die höchsten Werte entfernt. Diese minimalen Abweichungen dürften für das menschliche Auge als vernachlässigbar eingestuft werden.

c. Analyse imadjust

Es soll analysiert werden, was die Funktion imadjust im folgenden Matlabcode bewirkt:

```
[X,map] = imread('forest.tif');
I = ind2gray(X, map);
J = imadjust(I, [], [], 0.5);
imshow(I);
```

```
figure, imshow(J)
```

i. Wirkung imadjust

Wie aus den Bildern deutlich zu erkennen ist, wird das Bild einerseits leicht aufgehellt und andererseits im Kontrast verändert (Details sind besser erkennbar).

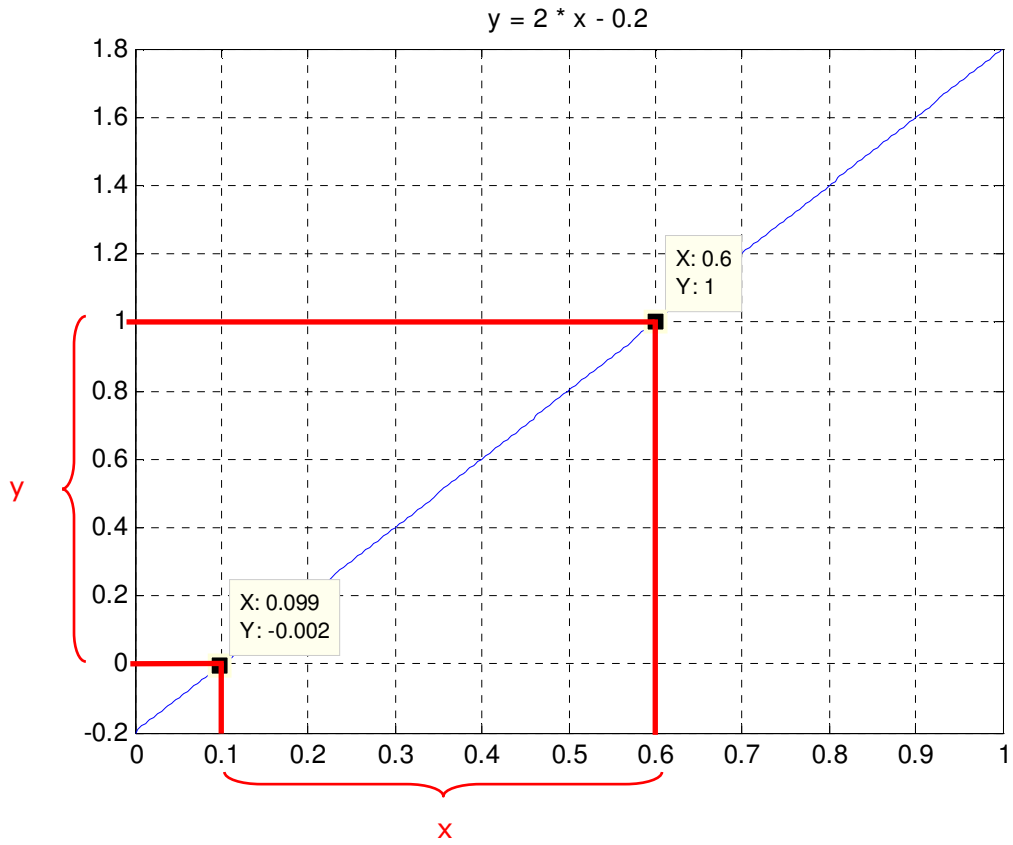
Übung 2

4. Gammakorrektur

Eine Gammafunktion kann verwendet werden, um die Helligkeit und den Kontrast eines Bildes anzupassen. Sie stellt die Helligkeitswerte am Ausgang gegen die Helligkeitswerte am Eingang dar.

a. Kontrastanpassung (linear)

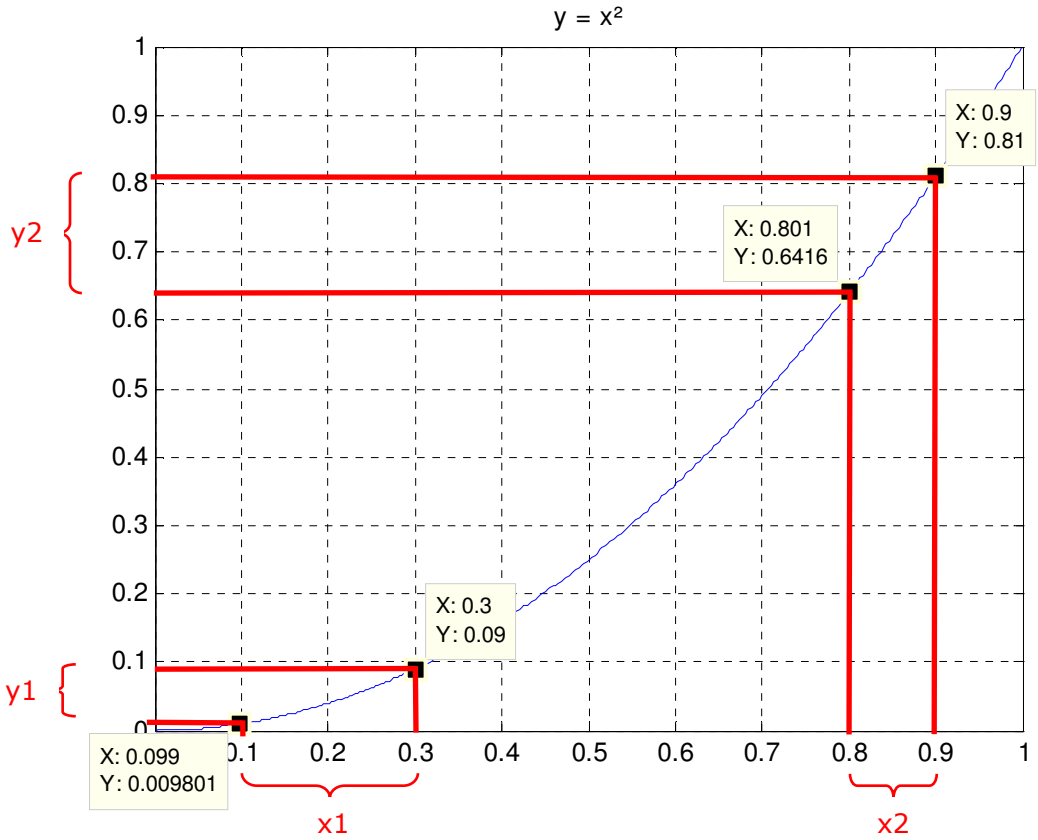
Um den Kontrast zu erhöhen wird eine lineare Gammafunktion erstellt, d.h. eine simple Gerade ($y = kx + d$). Um den Kontrast zu erhöhen, wird ein k größer 1 (steile Gerade) gewählt, damit die Helligkeitswerte am Ausgang breiter gefächert sind, d.h. durch die Gammafunktion „aufgespreizt" werden. Das d wird kleiner 0 gewählt (Gerade schneidet die x-Achse (Eingangshelligkeit) bei einer Helligkeit größer 0), was dafür sorgt, dass neben der Spreizung auch eine leichte Verschiebung stattfindet, sodass dunkle Teile des Eingangsbildes nicht berücksichtigt werden.
Die folgende Grafik veranschaulicht das anhand der Geraden $y = 2x - 0{,}2$:



Wie man erkennen kann, wird der Bereich zwischen 0,1 und 0,6 (x) im Eingangssignal auf das komplette Ausgangssignal (y, zwischen 0 und 1) aufgespreizt. Werte unter 0,1 bzw. über 0,6 im Eingangssignal werden hierbei nicht berücksichtigt bzw. auf 0 bzw. 1 gesetzt.
Beim Experimentieren mit dem in b. erläuterten Code wurden für die Geraden k = 2.3 und d = -0,6 als optimal erachtet, da das Bild bei dieser Geraden subjektiv am besten korrigiert wirkte.

b. Helligkeitsanpassung (nichtlinear)

Um eine Helligkeitsanpassung vorzunehmen wird ebenfalls eine Gammafunktion verwendet. Im Vergleich zur Kontrastanpassung ist diese bei der Helligkeitsanpassung allerdings nichtlinear (in diesem Fall x^r). Durch die Nichtlinearität (im Fall eines positiven Exponenten) bewirken große Helligkeitsunterschiede im dunklen Bereich des Eingangssignals kleine Helligkeitsunterschiede im dunklen Bereich des Ausgangssignals. Umgekehrt bewirken kleine Helligkeitsunterschiede im hellen Bereich des Eingangssignals große Helligkeitsunterschiede im hellen Bereich des Ausgangssignals.
Die folgende Grafik veranschaulicht das anhand des Parabelasts $y = x^2$:

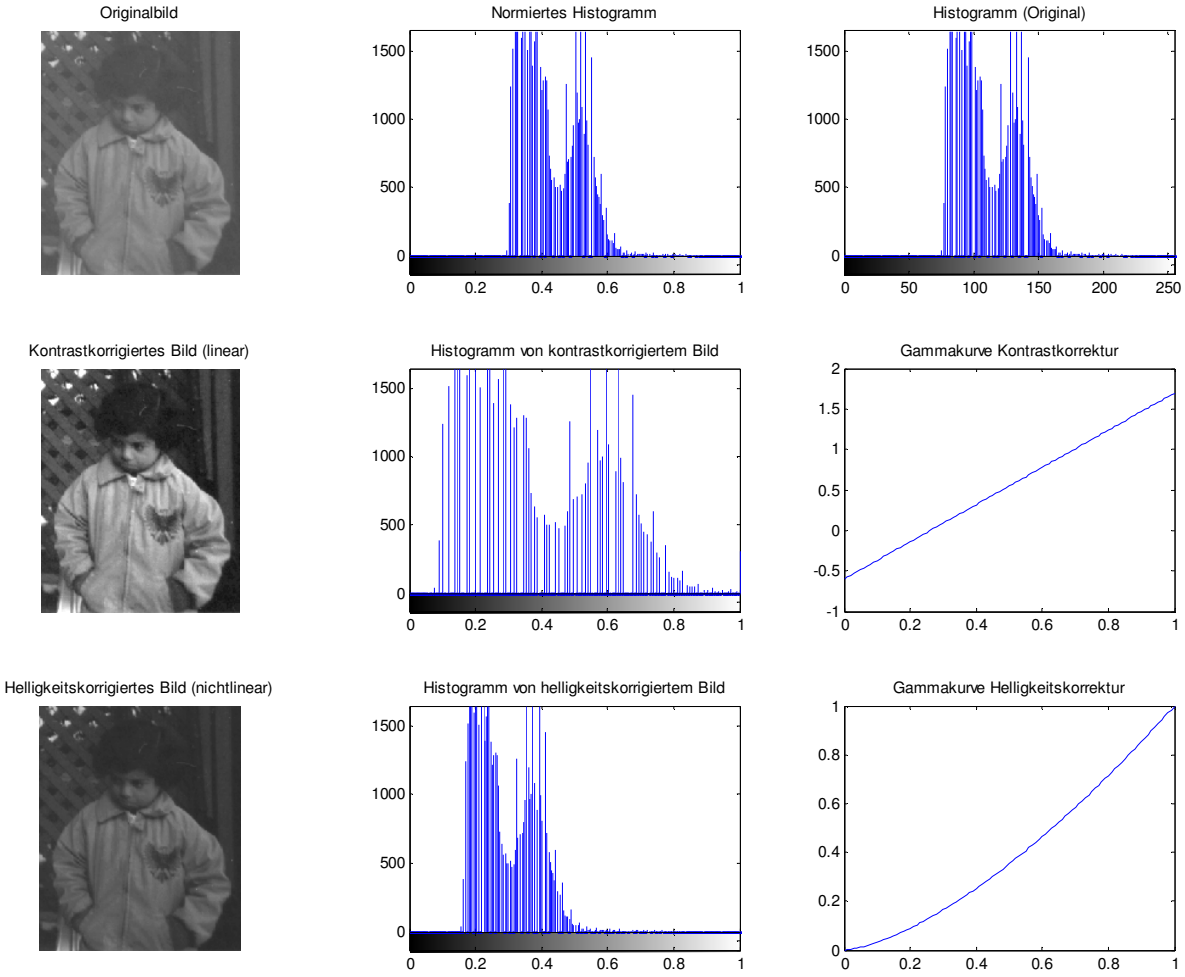


Deutlich zu erkennen ist, dass der große x1-Bereich auf einen kleinen y1-Bereich und der kleine x2-Bereich auf einen großen y2-Bereich abgebildet wird. Beim Experimentieren im unten erläuterten Code hat sich ein r von 1.5 als subjektiv beste Lösung ergeben.

Die Realisierung der Helligkeits- und der Kontrastanpassung in Matlab funktioniert wie folgt: zuerst wird das Originalbild (pout.tif) in ein Array geladen und in einen Wertebereich zwischen 0 und 1 (reell, daher Double) transformiert. Danach wird jeder Ausgangswert (Pixel) berechnet, indem der entsprechende Pixel des Originalbild durch die Gammafunktion angepasst wird.

```
original = imread('pout.tif'); %Bild laden
img = double(original); %In Double umwandelt (=> reelle Zahlen für
Normierung)
img_norm = img ./ 256; % Normierung auf Werte zwischen 0 und 1

subplot(331); imshow(original); title('Originalbild');
subplot(332); imhist(img_norm); title('Normiertes Histogramm');
subplot(333); imhist(original); title('Histogramm (Original)');

img_kontrast = 2.3 * img_norm - 0.6; %Gammafunktion für Kontrasterhöhung
```

```
subplot(334); imshow(img_kontrast); title('Kontrastkorrigiertes Bild
(linear)');
subplot(335); imhist(img_kontrast); title('Histogramm von
kontrastkorrigiertem Bild');
x = 0:0.01:1; y = 2.3 * x - 0.6; %Gammafunktion
subplot(336); plot(x, y); title('Gammakurve Kontrastkorrektur');

img_hell = img_norm .^ 1.5;
subplot(337); imshow(img_hell); title('Helligkeitskorrigiertes Bild
(nichtlinear)');
subplot(338); imhist(img_hell); title('Histogramm von
helligkeitskorrigiertem Bild');
x = 0:0.01:1; y = x .^ 1.5; %Gammafunktion
subplot(339); plot(x, y); title('Gammakurve Helligkeitskorrektur');
```

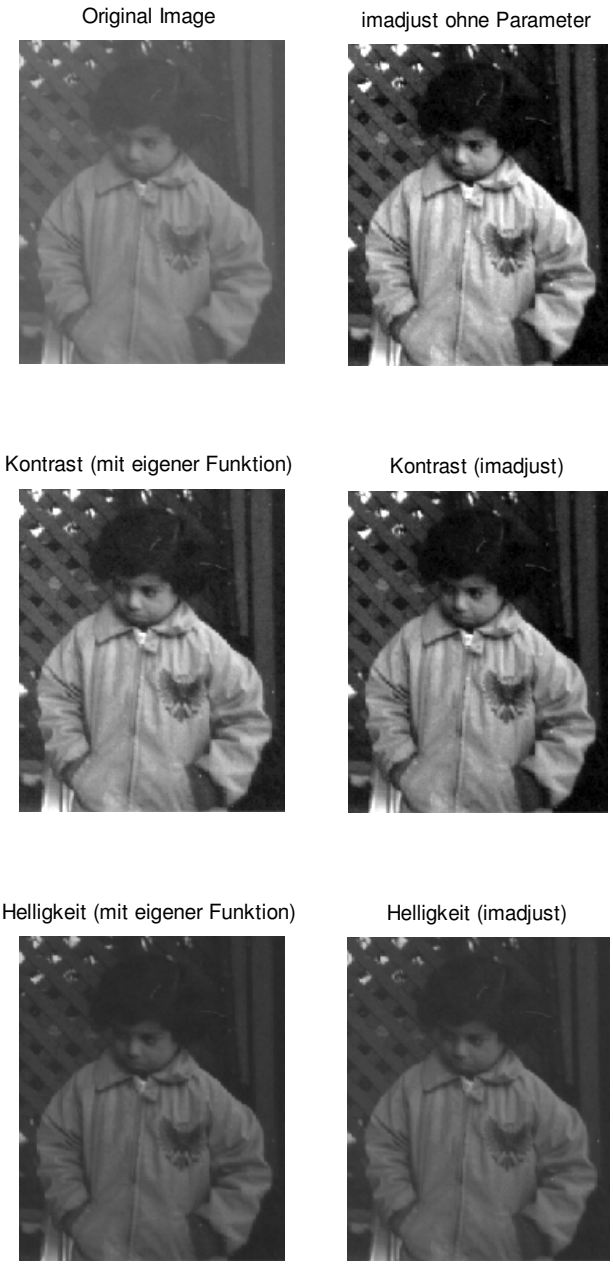


Man erkennt das im Vergleich zum Original leicht nach rechts (heller) verschobene Histogramm nach der Helligkeitsanpassung, sowie das verschobene und gespreizte Histogramm nach der Kontrastanpassung.
Matlab bietet mit der Funktion imadjust ebenfalls eine Möglichkeit an, über eine Gammafunktion Helligkeit und Kontrast anzupassen. Da die Parameter schwer mit k, d und r vergleichbar sind, wurden aus Zeitgründen Parameter gewählt, die in etwa ähnliche Resultate liefern wie die oben definierten eigenen Gammafunktionen.

```
original = imread('pout.tif'); %Bild laden
img = double(original); %In Double umwandelt (=> reelle Zahlen für
Normierung)
img_norm = img ./ 256; % Normierung auf Werte zwischen 0 und 1
```

```
adjusted = imadjust(original);
subplot(321); imshow(original); title('Original Image');
subplot(322); imshow(adjusted); title('imadjust ohne Parameter');

contrast = 2.3 * img_norm - 0.6;
subplot(323); imshow(contrast); title('Kontrast (mit eigener Funktion)');
adjusted = imadjust(original, [0.3 0.7], []);
subplot(324); imshow(adjusted); title('Kontrast (imadjust)');

hell = img_norm .^ 1.6;
subplot(325); imshow(hell); title('Helligkeit (mit eigener Funktion)');
adjusted = imadjust(original, [], [], 1.5);
subplot(326); imshow(adjusted); title('Helligkeit (imadjust)');
```

Die folgende Grafik versucht, die Ergebnisse von imadjust mit denen der selbst definierten Gammafunktionen zu vergleichen:



5. Histogramm Equalisation

Treten in einem Bild bestimmte Helligkeitswerte bzw. Gruppen von Helligkeitswerten häufig auf und sorgen dadurch für ein zu dunkles oder zu helles Bild, kann die Bildhelligkeit durch eine Histogramm-Equalisation-Funktion korrigiert werden. Der hier implementierte, relativ einfach gehaltene Algorithmus wird im Folgenden erläutert:

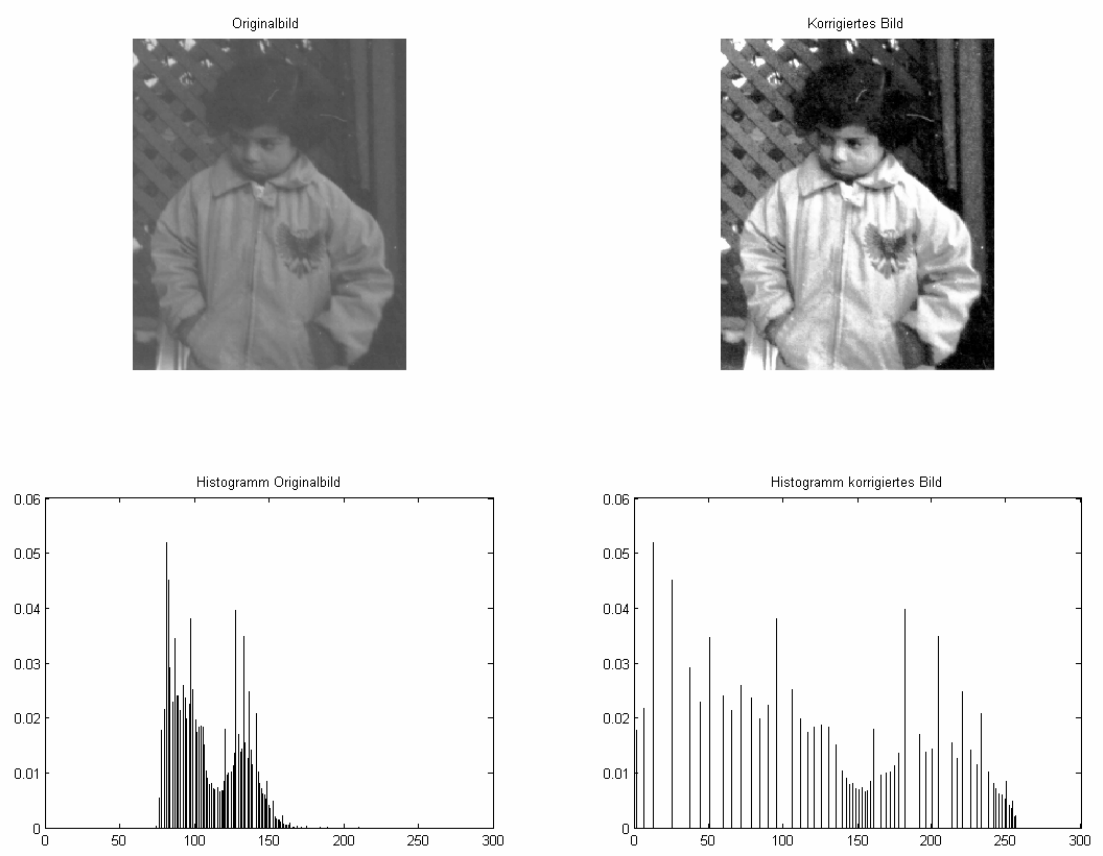
Im ersten Schritt wird das Histogramm des zu korrigierenden Bildes berechnet und normiert (sodass die Werte auf der y-Achse in % angegeben sind). Danach werden all jene Helligkeitswerte, deren y-Wert im Histogramm ungleich 0 ist auf das gesamte korrigierte Histogramm aufgeteilt.
Der erste Wert im korrigierten Histogramm bekommt somit die Intensität (y-Achsen-Wert im Histogramm) des ersten Helligkeitswerts ungleich 0 im Originalhistogramm zugewiesen. Dieser Zusammenhang zwischen altem und neuem Wert muss zwischengespeichert werden, damit die Werte später zurückgeändert werden können das korrigierte Bild die richtigen Helligkeiten aufweist. Dies wird über eine Verweis-Tabelle (alter Wert → neuer Wert) realisiert.
Nach dem neu gesetzten Wert im korrigierten Histogramm werden nun so viele nachfolgende Helligkeitswerte auf null gesetzt, dass die Anzahl der Nullwerte (inklusive dem neu gesetzten Wert) prozentuell den gleichen Anteil an 256 (Anzahl möglicher Grauwerte) haben wie die Intensität des neu gesetzten Werts an 100%. Der neue Wert wird quasi über die x-Achse in Form von Nullen im Histogramm ausgeweitet. Dieser Schritt wird für jeden Helligkeitswert ungleich 0 im Originalhistogramm wiederholt.
Zu guter letzt müssen im Originalbild alle Werte durch die neu zugeordneten Werte (siehe Verweis-Anmerkung oben) ersetzt werden, um das korrigierte Bild zu ergeben.

Die Implementation in Matlab sieht wie folgt aus:

```
img = imread('pout.tif');
number_of_pixels = 0;
hist_orig = zeros(256); %Original-Histogramm
hist_neu = zeros(256); %Korrigiertes Histogramm
verweis = zeros(256); %Tabelle zur Verknüpfung zwischen Helligkeitswerten
im Original- und korrigierten Histogramm

%Bildbreite/-höhe berechnen
[size_x, size_y] = size(img);

%Histogramm berechnen
for i = 1:size_x
    for j = 1:size_y
        hist_orig(img(i, j)) = hist_orig(img(i, j)) + 1;
        number_of_pixels = number_of_pixels + 1;
    end
end

%Histogramm normalisieren (um % zu erhalten)
for i = 1:256
    hist_orig(i) = hist_orig(i) / number_of_pixels;
end

%Histogramm
subplot(2, 2, 3);
bar(1:256, hist_orig); title('Histogramm Originalbild');

hist_neu_position = 0;

%Histogramm Equalisation
for i = 1:256
```

```
    if (hist_orig(i) == 0) %Nullwerte überspringen
        continue;
    end
    hist_neu_position = hist_neu_position + 1; %Nächster Wert im neuen
Histogramm wird befüllt
    hist_neu(hist_neu_position) = hist_orig(i); %Alten Wert an neue
Position schreiben
    verweis(i) = hist_neu_position; %Neue Position für Zuordnung speichern
    hist_neu_position = hist_neu_position + round(hist_orig(i) * 256) - 1;
%Nullen einfügen (bzw. Position nach hinten setzen)
end

%Histogramm neu
subplot(2, 2, 4);
bar(1:256, hist_neu); title('Histogramm korrigiertes Bild');

%Originalbild anzeigen
subplot(2, 2, 1);
imshow(img); title('Originalbild');

%Werte im Bild laut verweis-Array austauschen
for i = 1:size_x
    for j = 1:size_y
        img(i, j) = verweis(img(i, j));
    end
end

%Korrigiertes Bild anzeigen
subplot(2, 2, 2);
imshow(img); title('Korrigiertes Bild');
```

Der Code liefert folgenden Output:



Übung 3

1. XOR-Verknüpfung mit einem neuronalen Netz

Ein neuronales Netz soll so trainiert werden, dass es die an den beiden Eingängen anliegenden, binären Signale XOR-verknüpft. Dazu wird mittels der Matlabfunktion newff ein neues neuronales Netz mit zwei Eingängen, einem Ausgang und 3 Hidden Layers (Anzahl als Versuchswert angenommen) erzeugt. Die Wertebereiche der Eingänge müssen zwischen 0 und 1 definiert werden, damit die Gewichte, die in einem neuronalen Netz beim Lernen angepasst werden, mit einem gültigen Zufallswert initialisiert werden können. Als Transferfunktion wird eine Logarithmus-Sigmoid-Funktion verwendet (Vorgabe).

```
eingang =  [0 1 ;
            0 1];
net = newff(eingang, [3 1], {'logsig' 'logsig' 'logsig'});
```

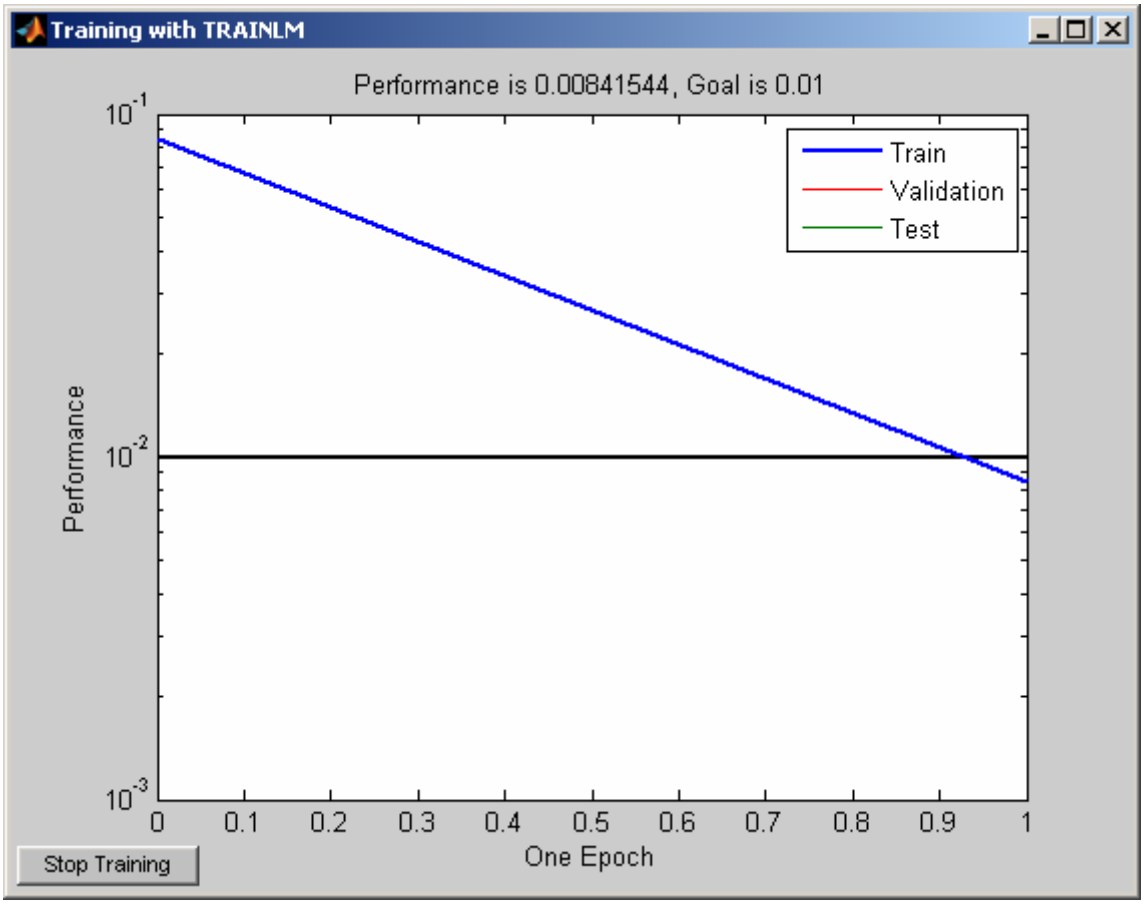
Über den goal-Parameter des von newff zurückgelieferten neuronalen Netzes kann angegeben werden, bis zu welcher Fehlerrate (mittlerer quadratischer Fehler) das Training mit den weiter unten beschriebenen Werten durchgeführt werden soll. Mit dem epochs-Parameter kann eine maximale Anzahl von Versuchen für das Training angegeben werden. Die tatsächliche Anzahl der Versuche, die benötigt werden, bis das neuronale System unter die angegebene Fehlerrate kommt ist zufällig, da auch die Startgewichte zufällig sind.

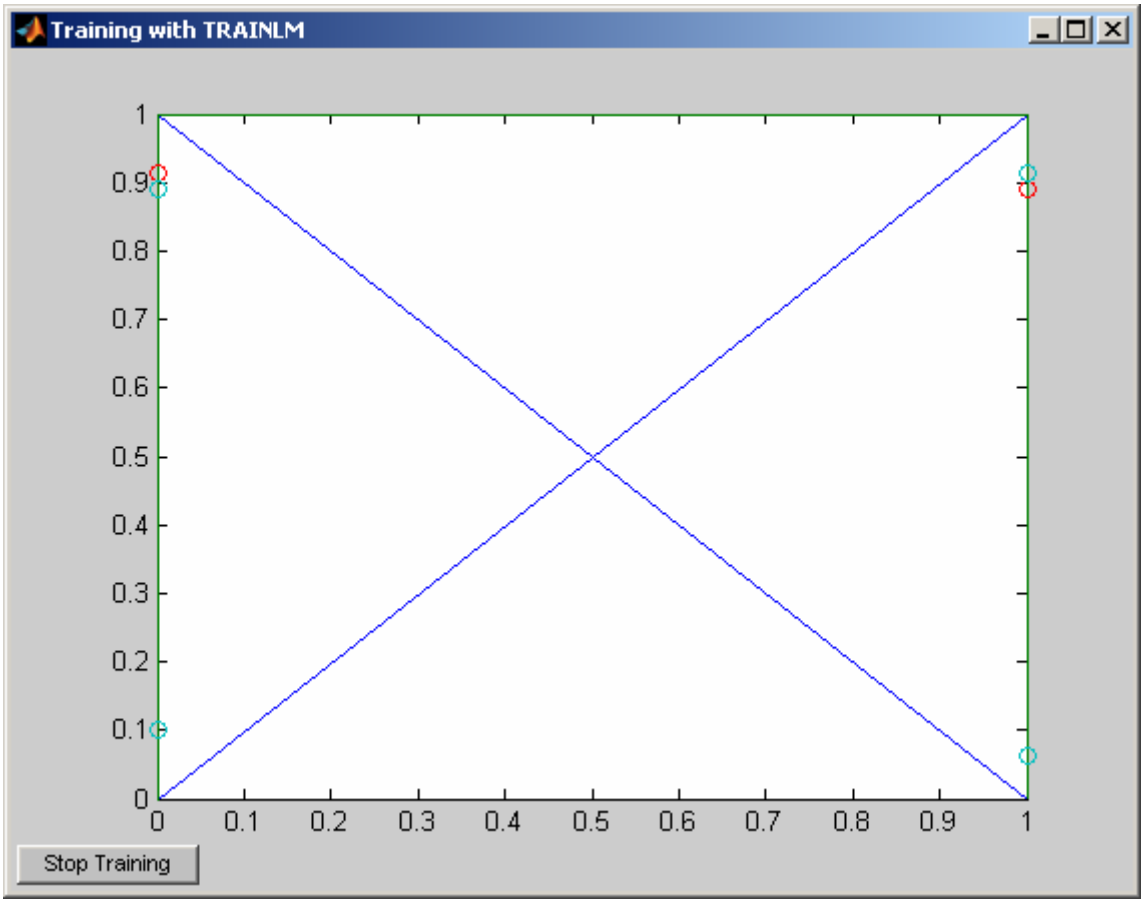
```
net.trainParam.goal = 0.01;
net.trainParam.epochs = 50;
```

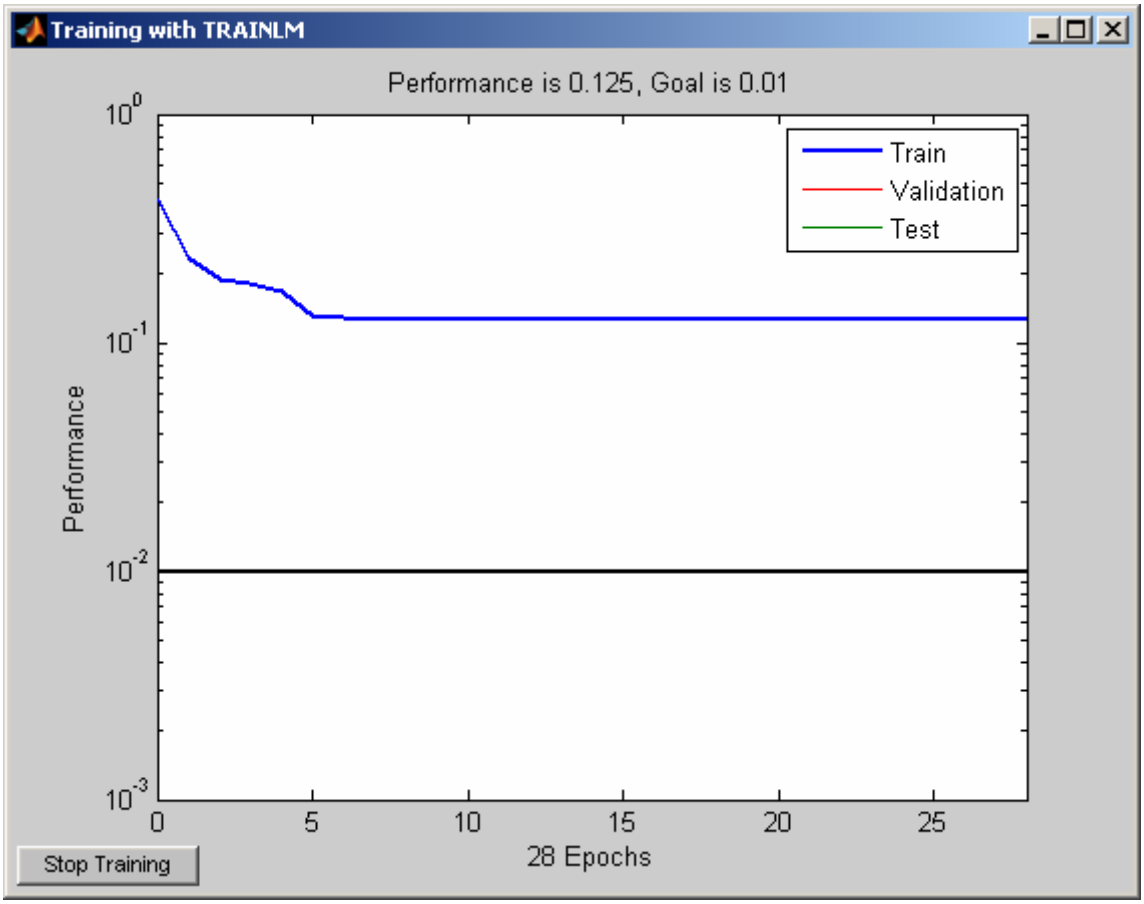
Das eigentliche Training läuft wie folgt ab: in zwei Matrizen werden vordefinierte Eingangswerte und die dazugehörigen Ausgangswerte spezifiziert und der Methode train übergeben. Das neuronale Netz wird dann so lange trainiert, bis – wie oben bereits beschrieben – die angegebene Fehlerrate unterschritten wird. Im diesem Beispiel wird die komplette XOR-Tabelle in Form von zwei Matrizen an die Trainingsroutine übergeben:

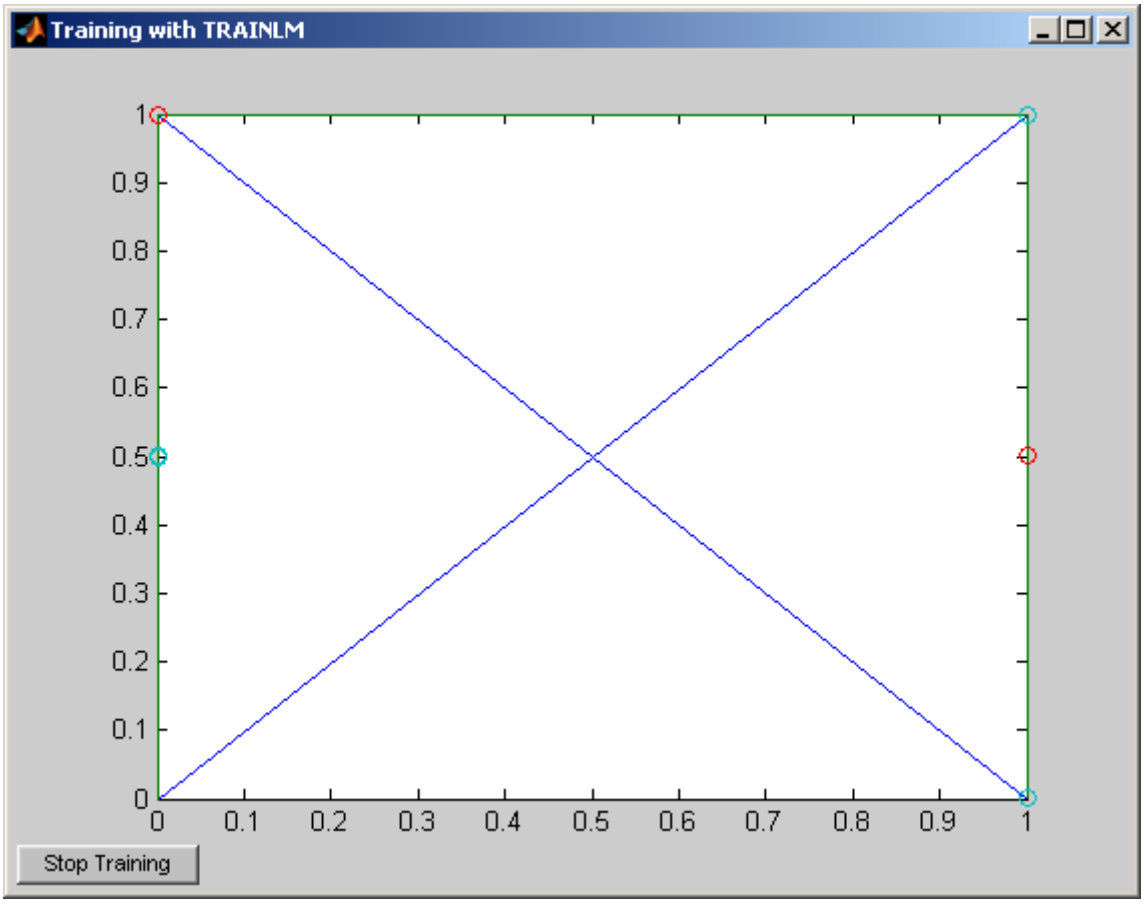
```
moeglichkeiten = [0 1 0 1;
                  0 0 1 1];

ergebnisse = [0 1 1 0];

net = train(net, moeglichkeiten, ergebnisse);
plot(moeglichkeiten, ergebnisse, moeglichkeiten, Y, 'o')
```

Der Plot visualisiert anschließend die korrekten und die durch das neuronale Netzwerk berechneten Ausgangszustände. Je nach Startgewichten können die korrekten (grün) und die berechneten Ausgangszustände (rot) mehr oder weniger weit auseinander liegen. Da die Startgewichte zufällig gewählt werden, dauert das Training unterschiedlich lang. So kann beispielsweise bereits nach einem Trainingsdurchlauf eine Fehlerrate von unter einem Prozent erreicht werden (siehe die ersten beiden Grafiken). Es kann allerdings auch passieren, dass die Fehlerrate aufgrund ungünstiger Startgewichte nach insgesamt 50 Versuchen nicht unter ein Prozent sinkt. Im letzten Fall werden die Gewichte nicht richtig justiert, wie man im zweiten Plot (letzte beiden Grafiken) erkennen kann.


Training with TRAINLM
Performance is 0.00841544, Goal is 0.01
Train
Validation
Test
Performance
One Epoch
Stop Training


Training with TRAINLM
Stop Training


Training with TRAINLM
Performance is 0.125, Goal is 0.01
Train
Validation
Test
Performance
28 Epochs
Stop Training


Training with TRAINLM
Stop Training

Anschließend soll überprüft werden, ob das Netz korrekt arbeitet. Dazu wird wieder mit der XOR-Tabelle als Eingang simuliert:

```
Y = sim(net, moeglichkeiten);
```

Einige Versuchen zeigten, dass das neuronale Netz – falls es nach den 50 Versuchen unter ein Prozent Fehlerrate sinken konnte – die richtigen Lösungen wiedergab. Zudem wurde versucht, die Wertepaare in einer anderen Reihenfolge als der antrainierten anzugeben, was nicht immer die richtigen Ergebnisse lieferte.

```
Y = sim(net,[0; 0]);
Y = floor(Y + 0.1);
```

Das neuronale Netz rechnet mit double-Werten. Die floor-Funktion rundet den zurückgegebenen Wert ab und sorgt dafür, dass der Rückgabewert als 0 oder 1 angezeigt wird.

2. Einfluss des goal-Parameters

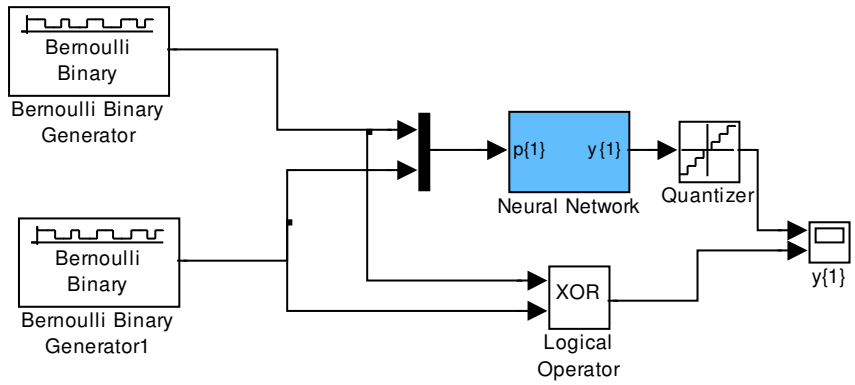
Der goal-Parameter gibt – wie in 1. erwähnt – die Fehlerrate an, die erreicht werden soll. Eine niedrige Fehlerrate verringert die Anzahl der Versuche, die notwendig sind, um die Rate zu erreichen, erhöht aber auch die Wahrscheinlichkeit, dass das neuronale Netz nicht die gewünschten Ausgangszustände liefert. Da die Anzahl der Versuche beschränkt ist, führt eine sehr niedrige Fehlerrate folglich auch öfter zu erfolglosen Trainings.

3. Erstellung eines Simulink-Modells

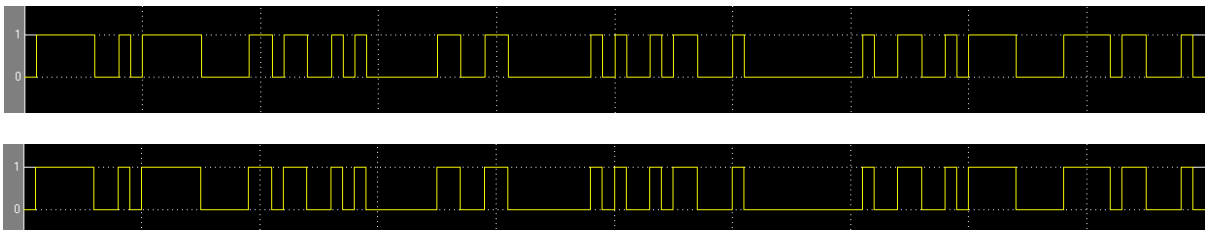
Aus dem in 1. erstellten neuronalen Netz soll ein Simulink-Modell generiert werden, das mithilfe einiger Bauteile erweitert werden soll, um die Funktionalität des neuronalen Netzes zu demonstrieren. Dazu wird das neuronale Netz in einem ersten Schritt mithilfe der Funktion gensim in ein Simulink-Modell umgewandelt:

```
gensim(net)
```

Anschließend wird das Modell um zwei Bernoulligeneratoren erweitert, die binäre Zufallsfolgen erzeugen. Da die Generatoren pseudozufällig sind, müssen die Seeds der beiden Zufallsgeneratoren verschieden sein, damit sie nicht idente Signale erzeugen. Es reicht daher, den Seed eines Zufallsgenerators zu verändern.
Zusätzlich wird ein Muxer benötigt, da das neuronale Netzwerk zwei Eingangssignale benötigt. Die Visualisierung wird mittels eines Scopes realisiert, das die Ergebnisse einer XOR-Verknüpfung mit denen des neuronalen Netzes vergleicht:



Wie man aus der Grafik erkennen kann, stimmen die Signale oben (neuronales Netz) und unten (XOR-Verknüfung) überein. Werden die Werte mit einem Quantisierungsintervall von 1 quantisiert, ergibt sich oben und unten der exakt gleiche Verlauf. Ohne Quantisierer ergäbe sich eine Abweichung, die in den Gewichten begründet ist (vgl. 1.).



4. Obsterkennung mit einem neuronalen Netz

Es soll ein neuronales Netz erstellt werden, das anhand der vier Eingangsgrößen Rotanteil, Blauanteil, Grünanteil und Rundheit (jeweils in Prozent, Bereich zwischen 0 und 1) entscheiden soll, um welche Frucht es sich handelt. Dazu definieren wir: 00 = Apfel, 01 = Birne, 10 = Orange.

```
eingang =  [0 1 ;
            0 1 ;
            0 1 ;
            0 1];

net = newff(eingang, [3 2], {'logsig' 'logsig' 'logsig'});
net.trainParam.goal = 0.01;
net.trainParam.epochs = 50;
```

Wir definieren außerdem rote und grüne Äpfel, Birnen und Orangen anhand der folgenden Eingabewerte:

```
moeglichkeiten = [0.8 0.1 0.2 0.9; %roter Apfel
                  0.1 0.8 0.2 0.9; %grüner Apfel
                  0.3 0.7 0.1 0.3; %Birne
                  0.8 0.4 0.3 0.9]'; %Orange

ergebnisse = [0 0 0 1;
              0 0 1 0];
```

Die vorgegebenen Ausgänge sind – wie oben angegeben – zwei Äpfel, eine Birne und eine Orange. Nun wird das Netz trainiert:

```
net = train(net, moeglichkeiten, ergebnisse);
```

Abschließend wird getestet, wie das Programm auf zuvor nicht definierte Obstsorten reagiert:

„Grüne Birne“:
Ergebnis: 0 0,9219 → richtig als Birne erkannt
„Blaugrüner Apfel“:
Ergebnis:beide Werte < 10^{-4} → richtig als Apfel erkannt

Abschlussübung(en)

In den letzten Übungen soll eine Erkennung der gesprochenen Laute A, E, I und O mittels eines neuronalen Netzwerks auf einem DSP implementiert werden.

1. Vorbereitung eines Simulink-Modells

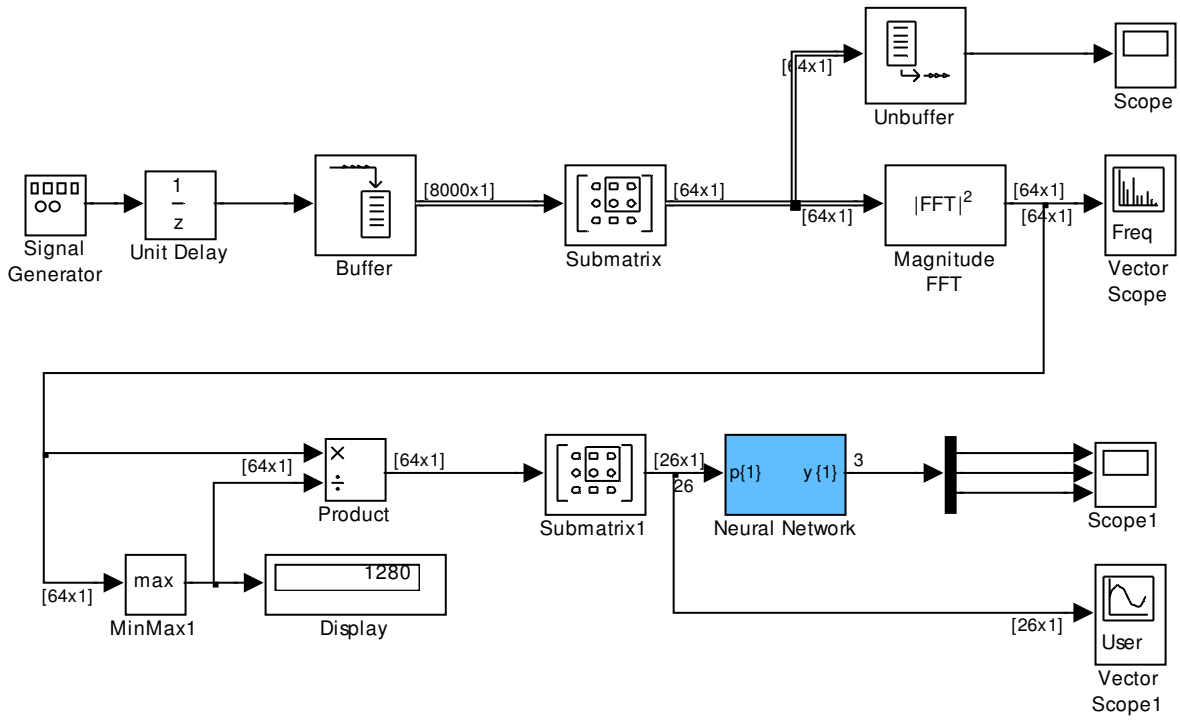
Ausgangspunkt für die Lauterkennung sind ein Simulink-Modell und ein Matlab-Skript, die ein neuronales Netzwerk implementieren, das anhand eines in den Frequenzbereich transformierten Audiosignals niedrige, mittlere und hohe Frequenzen (bis voneinander unterscheidet. Nachfolgend soll das Matlab-Skript erläutert werden:

```
% Beispiel
% Backpropagation Netzwerk
% Klassifizierung von Tönen in Tief - Mittel - Hoch
% in Abhängigkeit vom Spektrum:
% 26 Spektrallinien, Start bei 375Hz, Abstand 125Hz, Ende bei 3500Hz
% (FFT von 64 Werten, fa=8kHz, Weglassen der 3 ersten und 3 letzten Werte)
% (c) G. Jöchtl

% Eingangsvektoren (jew. 26 Werte)
%     T   T   T   T   T    M   M   M   M   M    H   H   H   H   H
P = [1.0 1.0 0.8 0.7 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0; %1
(375Hz)
     0.3 0.5 0.7 1.0 0.5  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0; % ->
Tief
     0.2 0.1 0.3 0.3 0.9  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0; %3
(625Hz)
     0.0 0.0 0.0 0.0 0.0  0.8 0.5 0.1 0.0 0.0  0.0 0.0 0.0 0.0 0.0; %4
(750Hz)
     0.0 0.0 0.0 0.0 0.0  0.2 0.6 0.8 0.0 0.0  0.0 0.0 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.3 0.3 0.2 0.0  0.0 0.0 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.1 0.4 0.0  0.0 0.0 0.0 0.0 0.0; % ->
Mittel
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 1.0 0.3  0.0 0.0 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.3 0.8  0.0 0.0 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.7  0.0 0.0 0.0 0.0 0.0; %10
(1500Hz)
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.7 0.0 0.0 0.0 0.0; %11
(1625Hz)
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.3 0.0 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.1 0.2 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.5 0.0 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.4 0.1 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.2 0.3 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.8 0.0 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.5 0.0 0.0; % ->
Hoch
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.1 0.2 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.4 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.8 0.0;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.4 0.1;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.2 0.3;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.5;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.8;
     0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.0  0.0 0.0 0.0 0.0 0.4
];%26(3500Hz)

% gewünschter Ausgang
```

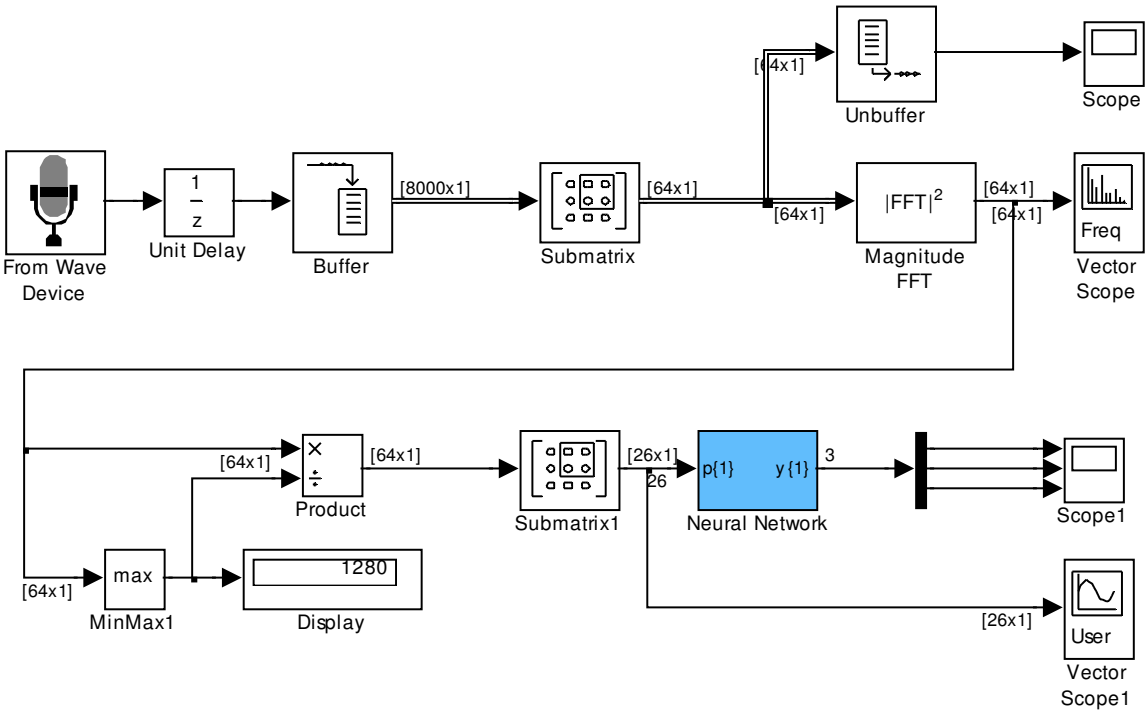
```
T = [ 1   1   1   1   1    0   0   0   0   0    0   0   0   0   0;   % Tief
      0   0   0   0   0    1   1   1   1   1    0   0   0   0   0;   %
Mittel
      0   0   0   0   0    0   0   0   0   0    1   1   1   1   1 ];  %
Hoch


inputs=[0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; %
26
        0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1];%
Werte
% Netzstruktur: 26 inputs 6 hidden units 3 output units, alle Ausgänge log.
Sigmoidfkt.
net = newff(inputs,[6 3],{'logsig' 'logsig'});
net.trainParam.epochs = 500;
net.trainParam.goal=0.001;
net = train(net,P,T);

% Test mit erlernten Werten
Y = sim(net,P)

% Test mit neuen Werten, z.B Ton mit 1250 Hz (->Mittel) :
Ptest = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]';
Y = sim(net,Ptest)
```

Das neuronale Netzwerk erwartet 26 Eingangswerte (26 Werte der FFT, siehe unten), besteht aus 6 hidden Units und 3 Output Units, um anzuzeigen, ob ein niederfrequentes, mittelfrequentes oder hochfrequentes Signal transformiert wurde.
P definiert die Trainingsdaten für das Netzwerk. Nach dem Training wird das Netzwerk mit einem neuen Wert trainiert, den es einwandfrei als mittelfrequent erkennt (Y enthält die Werte [0.0116, 0.9932, 0.0106])

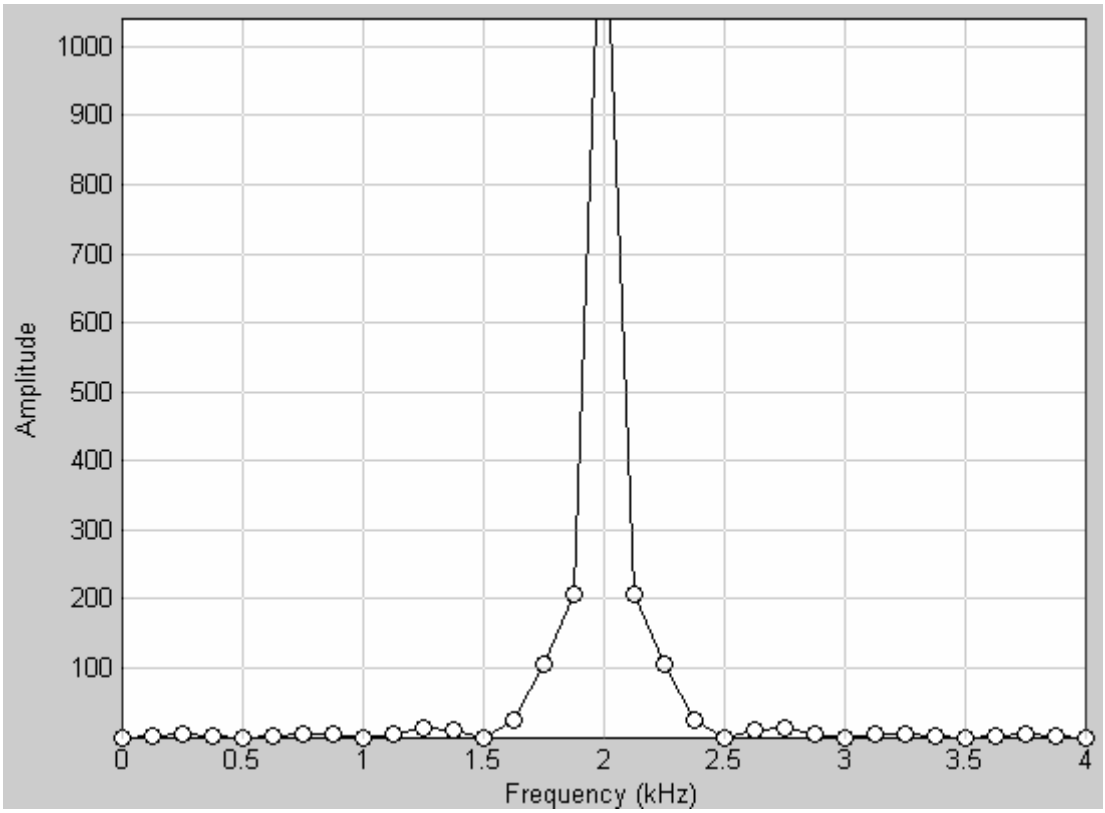
Im Anschluss soll das Simulink-Modell, in das das aus dem Matlab-Skript beschriebene neuronale Netzwerk eingebettet wurde, erläutert werden:



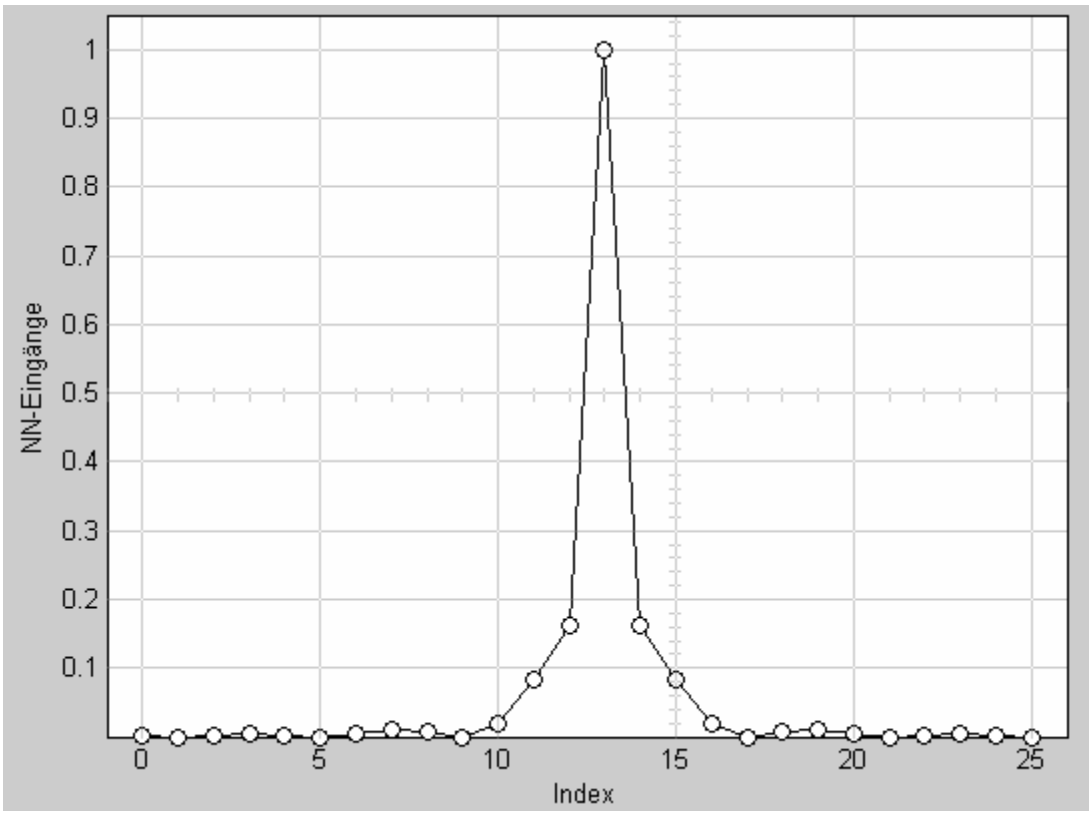
Das Signal aus dem Sinus-Generator wird mit einer Sample- und Hold-Schaltung mit 8 kHz abgetastet, was mehr als dem doppelten der maximal auftretende Frequenz entspricht. Die 8000 so in einer Sekunde abgetasteten Werte werden zur Weiterverarbeitung in einen Puffer schreiben. Da eine FFT mit 8000 Werten auf einem DSP schwer umsetzbar ist, werden die Werte in 64 Samples große Blöcke unterteilt (aus dem Puffer „geschnitten") und danach mittels FFT transformiert.
Zur Verarbeitung der Samples im neuronalen Netzwerk ist eine Normierung der Amplituden auf 1 notwendig. Dazu wird das Maximum der 64 Werte gebildet – alle anderen 63 Werte werden durch diesen Maximalwert dividiert. Dadurch kann erreicht werden, dass der Maximalwert zu 1 wird und alle anderen Werte im Intervall [0, 1] liegen.
Aus den 64 Werten werden anschließend die für die Spracherkennung relevanten herausgeschnitten, um die Anzahl der Eingänge des neuronalen Netzwerks zu minimieren. Sehr tiefe Töne werden dabei vernachlässigt, ebenso wie sehr hohe. Werte überhalb der halben Abtastfrequenz sind irrelevant da redundant.
Durch das neuronale Netzwerk wird anschließend (wie oben erläutert) eine Klassifizierung (nieder-, mittel-, hochfrequent) vorgenommen.
Zur Überprüfung der Funktionalität des neuronales Netzwerks werden nieder-, mittel- und hochfrequente Sinusschwingungen mittels einem Frequenzgenerators erzeugt und die Resultate beobachtet. Sie waren für mittel- und hochfrequente Schwingungen korrekt.
Eine erneute Überprüfung wurde mittels eines physischen Signalgenerators durchgeführt. Dazu wurde die Signalquelle durch den Mikrofoneingang ersetzt und der Signalgenerator mittels eines BNC-Klinke-Steckers auf den LineIn-Eingang verbunden. In Simulink können die auf dem LineIn-Eingang anliegenden Signale durch eine FromWaveDevice-Komponente eingelesen werden. Die Samplingrate dieser Komponente wurde auf 1 Sample pro frame gesetzt, da das eigentliche Sampling ohnehin bereits durch Mikrofon realisiert wird. Bei der erneuten Überprüfung kann aus demselben Grund die Sample- und Hold-Schaltung entfernt werden.
Versuche ergaben, dass eine Änderung des Bereichs beim Ausschneiden der frequenztransformierten Werte (SubMatrix1-Komponente) auf den Bereich zwischen 2 und 27 die Erkennung tiefer Frequenzen deutlich verbessert, da mehr tiefe Frequenzen berücksichtigt werden.
Die Simulink-Schaltung der angepassten Überprüfung mit dem Signalgenerator sieht nun wie folgt aus:



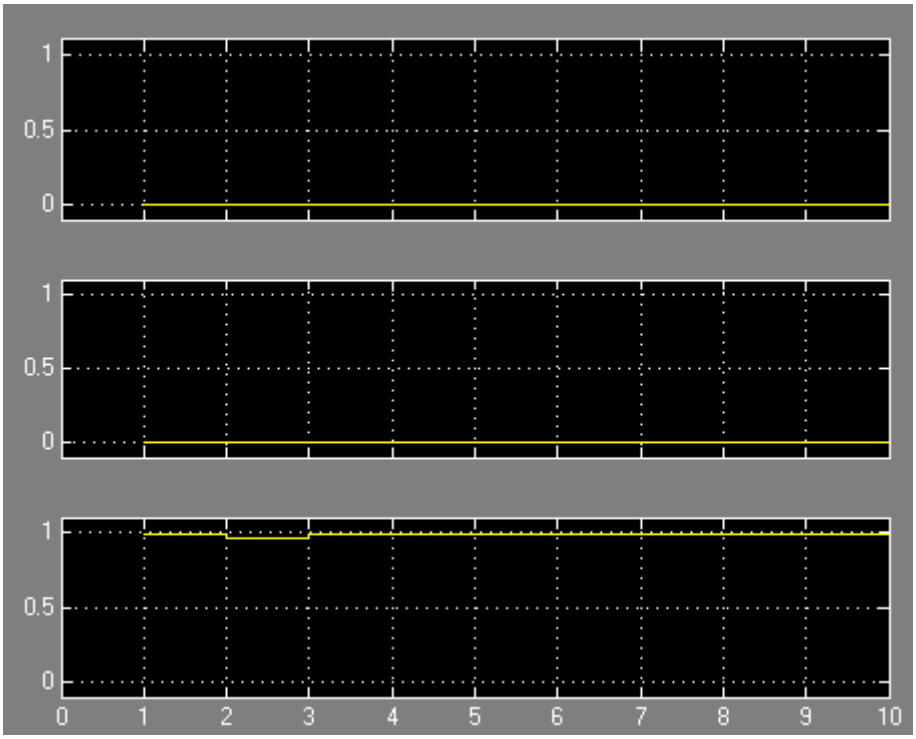
Nachfolgend der erfolgreiche Test einer hochfrequenten 2kHz-Sinusschwingung:



Frequenzbereich des Signals (FFT)

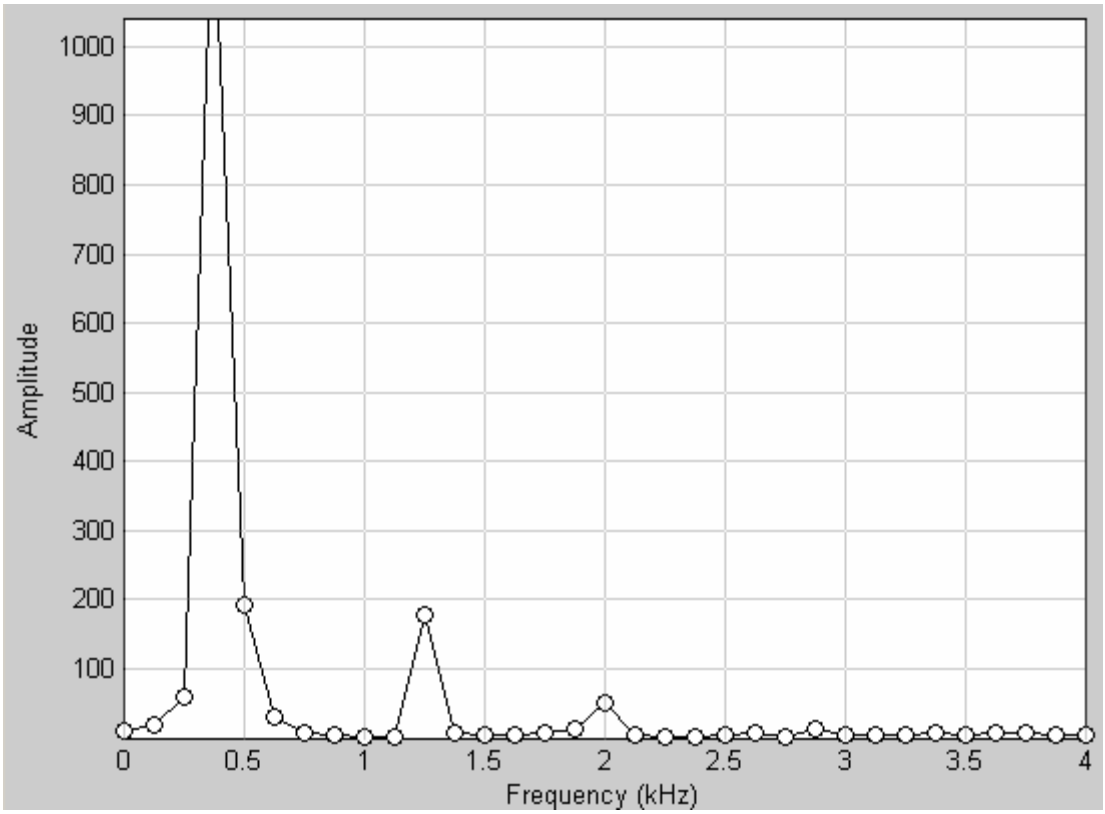


FFT der beschnittenen (26) Samples

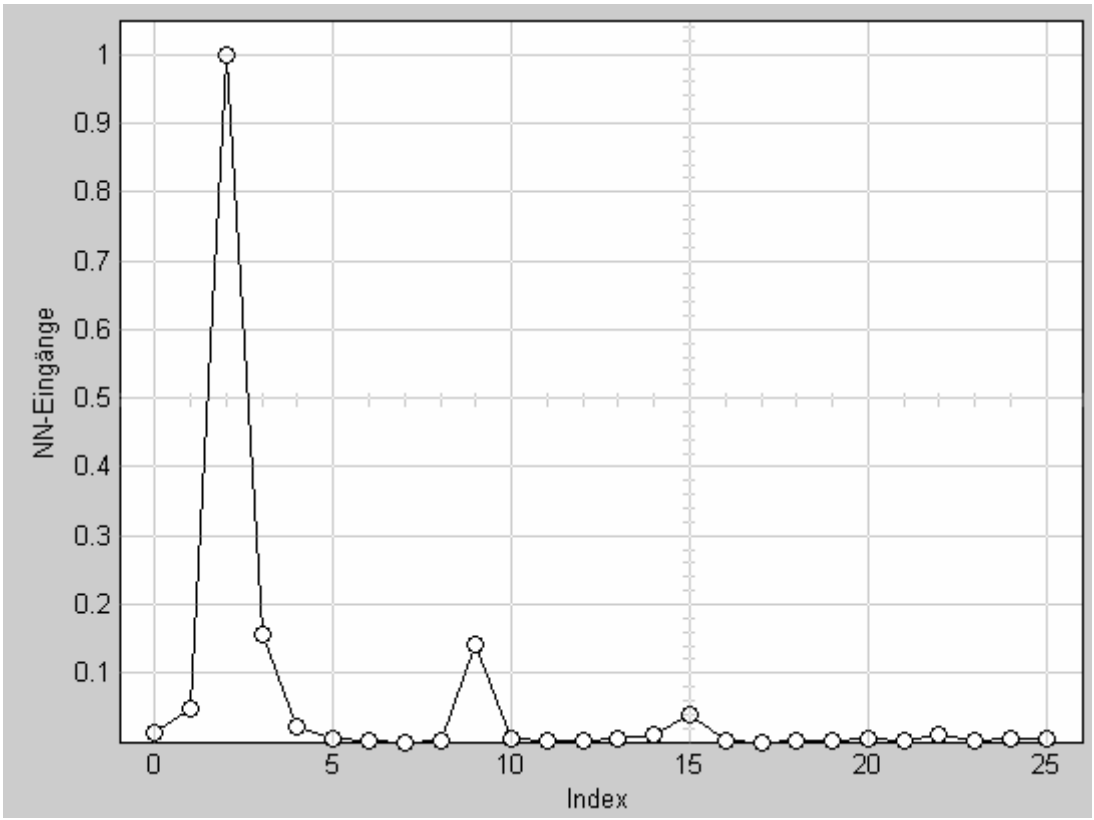


Klassifikation durch das neuronale Netzwerk (niedrig-, mittel-, hochfrequent v.o.n.u.)

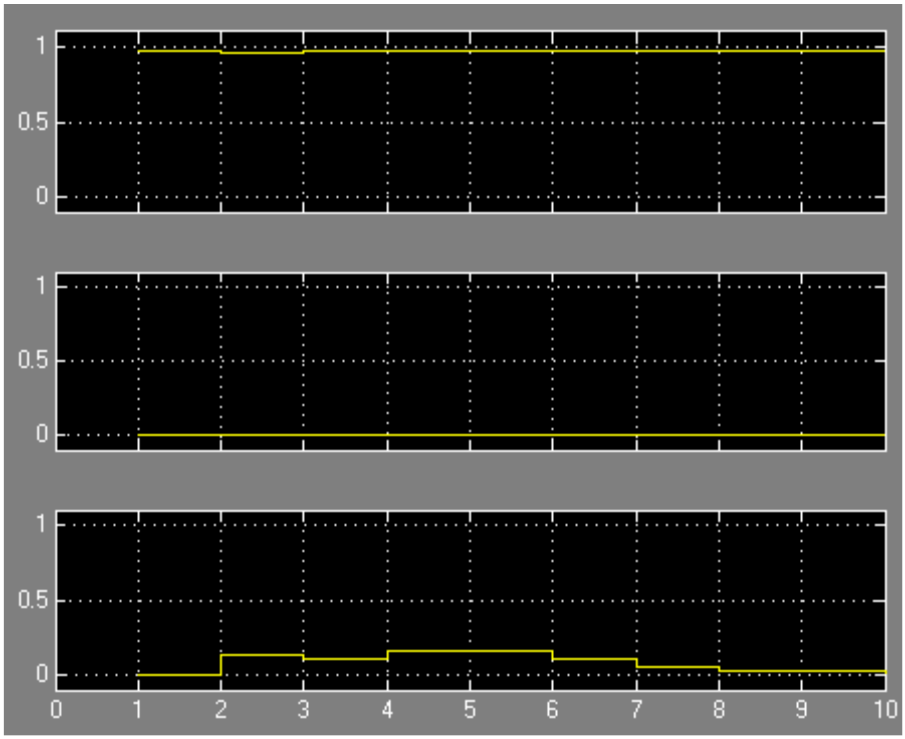
Nachfolgend auch der erfolgreiche Test einer mittelfrequenten 410Hz-Sinusschwingung:



Frequenzbereich des Signals (FFT). Durch Sample und Hold entstehen Oberwellen



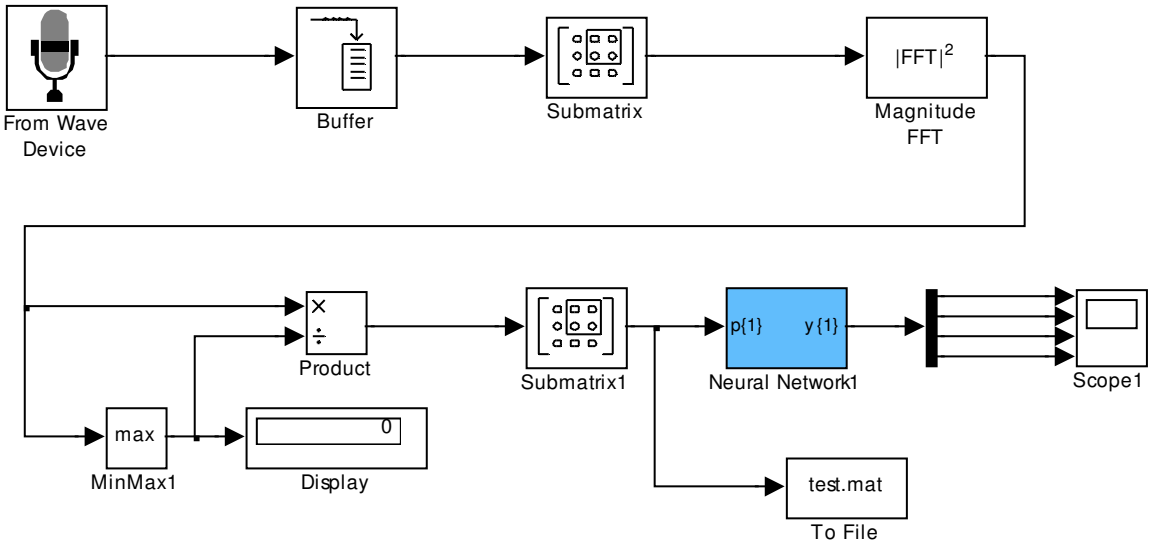
FFT der beschnittenen (26) Samples



Klassifikation durch das neuronale Netzwerk (niedrig-, mittel-, hochfrequent v.o.n.u.)

2. Lauterkennung mittels eines neuronalen Netzwerks

Das in 1. beschriebene Matlab-Skript sowie das Simulink-Modell sollen nun angepasst werden, um die Laute A, E, I und O zu erkennen.
Das Simulink-Modell wird nun dahingehend erweitert als dass As, Es, Is und Os mit einem Mikrofon aufgenommen und als Array in File gespeichert werden, um Trainingsdaten für das neuronale Netzwerk zu erhalten.
Anschließend werden mehrere Minuten lang As in das Mikrofon gesprochen und in der Datei a.mat gespeichert. Dieser Vorgang wird für E, I und O entsprechend wiederholt. Die erste Reihe der dabei erstellten Arrays wird im Anschluss gelöscht, da diese nur fortlaufende Nummern darstellt. „Stille" wird ebenfalls aufgezeichnet, um dem Netzwerk den Ausgangs-Sollzustand (0,0,0,0) zu lernen. Das Simulink-Modell sieht nun wie folgt aus:

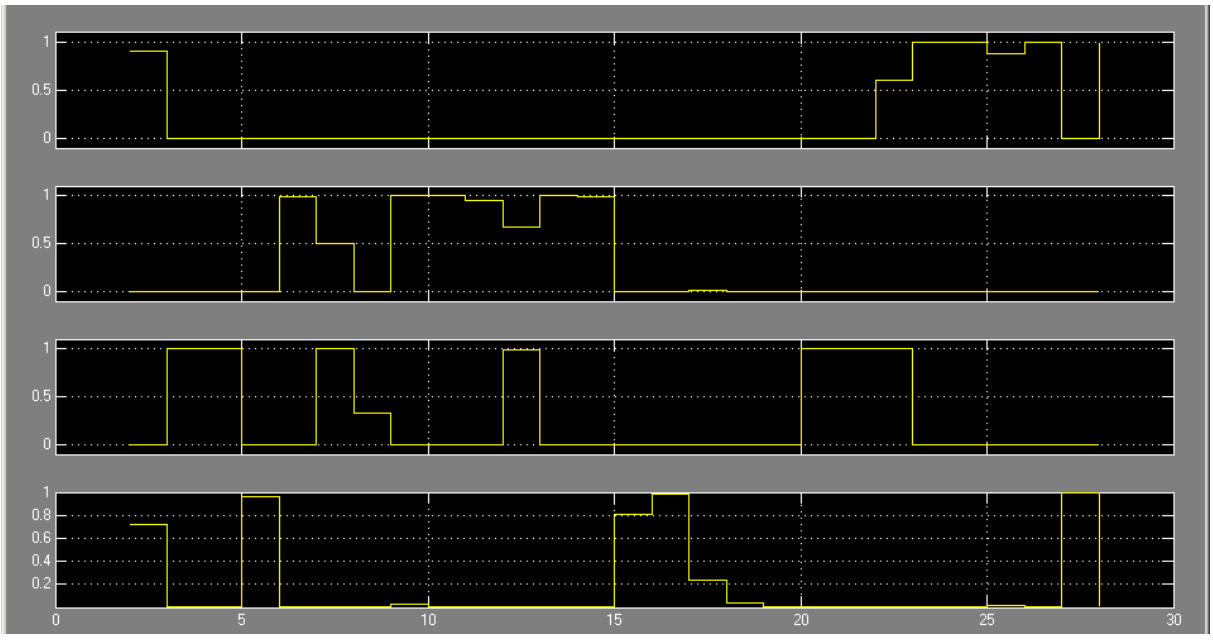


Im Matlab-Skript muss nun das neuronale Netzwerk so modifiziert werden, dass es 4 Ausgänge (einer für jeden der 4 Laute) aufweist. Als Trainingdaten werden nun die mit dem Simulink-Modell gespeicherten Arrays verwendet. Letztere sind in den entsprechenden mat-Dateien gespeichert und können mit der Funktion load in den Workspace geladen werden, um so mit den Variablennamen a, e, i und o (mit denen sie gespeichert wurden) angesprochen werden zu können. Die Soll-Ergebnisse des Trainings sind durch die Länge der Arrays vorgegeben. Sind beispielsweise 30 As, 40 Es, 40 Is und 50 Os trainiert worden, ist die Ergebnismatrix überall null, abgesehen von (Zeile, Spalte) den Stellen (1, 1-30) (A aufgenommen), (2, 31-70) (E aufgenommen), (3, 71-110) (I aufgenommen) und (4, 111-160) (O aufgenommen).
Das Matlab-Skript sieht nun wie folgt aus:

```
%Arrays laden
a = load('C:\Dokumente und Einstellungen\dsp\Desktop\lauterkennung\a.mat');
a = a.a;
e = load('C:\Dokumente und Einstellungen\dsp\Desktop\lauterkennung\e.mat');
e = e.e;
i = load('C:\Dokumente und Einstellungen\dsp\Desktop\lauterkennung\i.mat');
i = i.i;
o = load('C:\Dokumente und Einstellungen\dsp\Desktop\lauterkennung\o.mat');
o = o.o;
n = load('C:\Dokumente und
Einstellungen\dsp\Desktop\lauterkennung\null.mat'); %Stille
n = n.null;
```

```
%Arrays aneinanderhängen
P = cat(2, a, e, i, o, n);

%Gewünschte Ausgänge setzen
T = zeros(4, length(P(1,:)));

T(1, 1:length(a(1,:))) = 1; %As
for j = 1:1:length(e(1,:))
    T(2, length(a(1,:)) + j) = 1; %Es
end
for j = 1:1:length(i(1,:))
    T(3, length(a(1,:)) + length(e(1,:)) + j) = 1; %Is
end
for j = 1:1:length(o(1,:))
    T(4, length(a(1,:)) + length(e(1,:)) + length(i(1,:)) + j) = 1; %Os
end

%Wertebereiche der Inputs (0-1)
inputs=[0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1;
%26
        0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0
1];%Werte

% Netzstruktur: 26 Input Units, 10 Hidden Units 4 Output units, alle
Ausgänge log. Sigmoidfkt.
net = newff(inputs, [10 4], {'logsig', 'logsig', 'logsig'});
net.trainParam.epochs = 500;
net.trainParam.goal = 0.0001;
net = train(net, P, T); %Netz trainieren

% Test mit erlernten Werten
Y = sim(net, P);

%Simulinkmodell aus NN generieren
gensim(net)
```

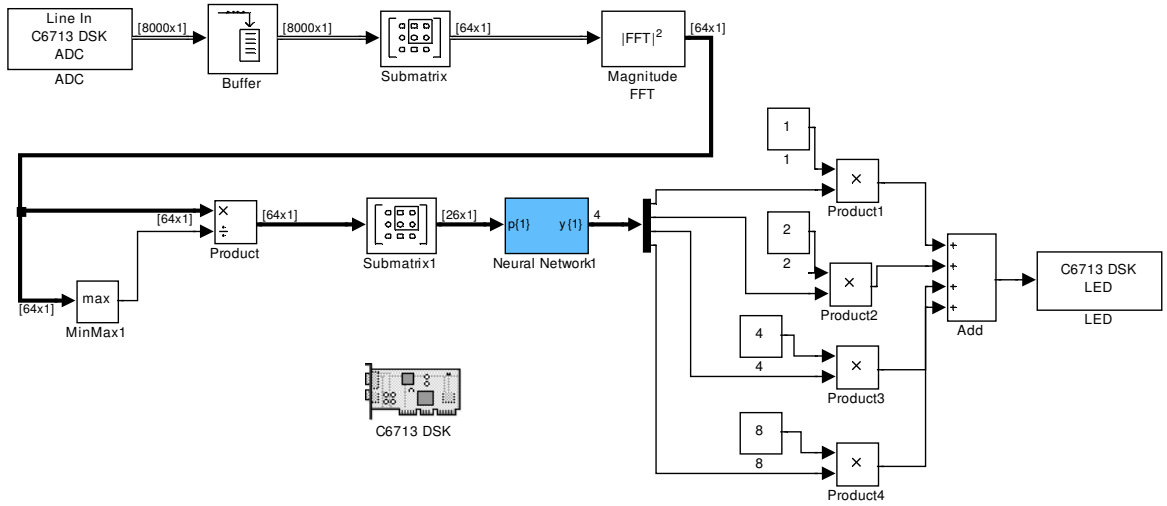
Das hier mit gensim erzeugte Modell wird zurück in die Simulinkschaltung integriert. Durch Ausprobieren wurden 10 Hidden Units und 4 Output-Units als zum Testen geeignet erachtet.



Die obenstehende Grafik zeigt das Scope (von oben nach unten: A, E, I, O) aus der obigen Schaltung während folgender Simulation (ein Aufzählungspunkt entspricht einer Sekunde im Scope, grün bedeutet korrekt erkannt, rot bedeutet nicht korrekt erkannt):

- A wird gesprochen (A und O erkannt)
- I wird gesprochen
- I wird gesprochen
- O wird gesprochen
- E wird gesprochen
- I wird gesprochen (E und I erkannt)
- I wird gesprochen
- E wird gesprochen
- E wird gesprochen
- E wird gesprochen
- E wird gesprochen (E und I erkannt)
- E wird gesprochen
- E wird gesprochen
- O wird gesprochen
- O wird gesprochen
- Stille
- I wird gesprochen
- I wird gesprochen (A und I erkannt)
- A wird gesprochen
- A wird gesprochen
- A wird gesprochen
- A wird gesprochen
- O wird gesprochen

Die Erkennungsrate ist relativ gut. Daher wird nun versucht, das Modell so zu erweitern, dass es auf ein DSP-Board gespielt und ausgeführt werden kann.



Neu hinzugekommen ist das Target (TI C6713) zur Erstellung des Codes für den DSP. Das Scope wurde durch ein LED ersetzt. Da das LED-Objekt Werte zwischen 0 und 15 erwartet werden die 4 Ausgänge des neuronalen Netzes mit Potenzen von 2 multipliziert und aufaddiert. Das sorgt dafür, dass jeder Ausgang einem LED entspricht.
Zusätzlich wurde der Audio-Input durch den entsprechenden Audio-Eingang am DSP ersetzt.
Abschließend wurde das Modell mittels Simulink auf den DSP gespielt. Es zeigt sich, dass der DSP leider eine zu schwache CPU hatte und daher nicht weiter getestet werden konnte. Wir vermuten allerdings, dass – eine entsprechende CPU vorausgesetzt – das neuronale Netz die selben Ergebnisse geliefert hätte wie in der Simulinksimulation.