Reiknigreind

Reikniverkefni 1

Hörður Freyr Yngvason

1 Inngangur

Í verkefninu skoðum við *back-propagation* á margra-laga tauganet fyrir flokkunarverkefni. Við höfum gefna útfærslu á slíku reikniriti, þar sem skekkjan er metin út frá 2-normi fjarlægðar frá réttum gildum, og viljum breyta kóðanum þannig að notast verði við *cross-entropy* fyrir skekkjumat.

Til að prófa útfærsluna höfum við síðan safn af 624 svarthvítum andlitsmyndum [1] sem unnt er að flokka á ýmsan hátt; t.d. eru 311 myndanna af viðfangsefni með sólgleraugu. Í seinni hluta verkefnisins reynum við að læra að flokka myndirnar með tauganeti af þessari gerð.

2 Fræðileg umgjörð

Táknum með $h : \mathbb{R} \to \mathbb{R}$ þjálthough einhalla vaxandi fall sem er takmarkað að ofan og neðan, köllum það *yfirfærslufall*. Við byrjum með með fjölskyldu para inntaks- og úttaksgilda (x_i, t_i) og setjum upp endanlegt stefnt net hnúta i með gildi a_i og z_i, og leggja $ji \equiv i \to j$ með þyngdir w_{ij}, sem tengjast með formúlunum

$$a_j = \sum_{i \to j} w_{ji} z_i$$

og

$$z_j = h(a_j),$$

þar sem $i \to j$ þýðir að við höfum legg með þyngd w_{ji} frá i til j (getum aldrei farið afturábak). Inntaksgildin x_i eru einu gildin z_i þannig að ekki er til tenging $j \to i$ og úttaksgildin eru þau gildi $y_k := z_k$ þannig að ekki sé til tenging $k \to j$. Netið skiptist síðan upp í *lög*, sem eru jafngildisflokkar hnútanna m.t.t. fjölda skrefa frá inntaki, og úttaksgildin y_k þurfa öll að vera í sama laginu. Við viljum síðan lágmarka gefna skekkju $E(\mathbf{w})$ með tilliti til þyngdanna w_{ji} yfir öll hugsanleg inntaksgildi x_i (en við látum okkur nægja að lágmarka fyrir úrtakið).

Nálgunin, sem tekin er í tauganetinu okkar, er að beita *gradient descent*, til að finna staðbundið lággildi á E m.t.t. $\mathbf{w}$ þar sem afleiðurnar $\partial E / \partial w_{ji}$ eru reiknaðar afturábak með *back-propagation*: Skilgreinum hjálparstærðir

$$\delta_j := -\frac{\partial E}{\partial a_j}$$

fyrir alla hnúta j. Gerum síðan ráð fyrir að við getum reiknað gildin δ_k fyrir alla úttakshnúta k. Af keðjureglunni fæst að

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

og með því að sjóða saman allt að ofan fæst

$$\delta_i = -\frac{\partial E}{\partial a_j} = -\sum_{i\to j}\frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial a_i} = h'(a_i)\sum_{i\to j} w_{ji}\delta_j$$

sem gefur okkur leið til að meta δ_i út frá öllum δ_j þannig að $i \to j$; þ.e.a.s. við getum metið skekkjuna á hverju lagi út frá skekkjunni á laginu fyrir ofan (þ.e. laginu sem er nær úttakinu).

3 Cross-entropy

Cross-entropy skekkjufallið er gefið með

$$E(\mathbf{w}) = -\sum_k (t_k \log(y_k) + (1 - t_k)\log(1 - y_k)).$$

Til að stinga þessu falli inn í *back-propagation* reikniritið nægir okkur nú að reikna frumstillingarnar

$$\delta_k = -\frac{\delta E}{\delta a_k};$$

munum að $y_k = h(a_k)$ svo $\partial y_k/\partial a_k = h'(a_k)$ og fáum

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{t_k}{y_k}h'(a_k) - \frac{1 - t_k}{1 - y_k}h'(a_k) = h'(a_k)\frac{t_k - y_k}{y_k(1 - y_k)}$$

Í okkar tilfelli notum við *logistic sigmoid* fallið

$$h(x) := \frac{1}{1 + e^{-x}}$$

með

$$h'(x) = h(x)(1 - h(x))$$

svo að

$$\delta_k = t_k - y_k.$$

Breytt útgáfa af skjalinu `vbp.m`, sem notar cross-entropy reiknað á þennan hátt, er í viðauka.

3.1 XOR-prófið

Þekkt er að XOR-prófið er ekki línulega sundurgreinanlegt, svo það er tilvalið í að bera almenn tauganet saman við stakar taugar. Til að sannreyna að forritið virki enn eftir breytinguna má keyra kóðabútinn í viðauka B á blaðsíðu 13; dæmi um þróun skekkjunnar má sjá á mynd 1; þar fékkst útkomuvigurinn

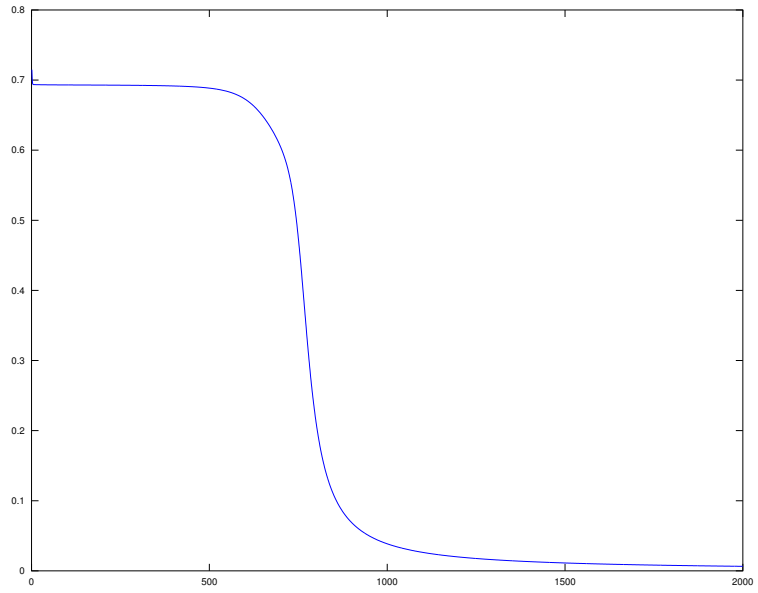
```
0.0055121   0.9942149   0.9942143   0.0086114
```

á móti réttu XOR-gildunum

```
0   1   1   0
```

á inntaksgildin

```
(0,0)  (0,1)  (1,0)  (1,1)
```



Mynd 1: Skekkjan sem fall af fjölda ítrana í XOR-prófinu

4 Myndgreining

Forritin fyrir þennan hluta voru skrifuð að mestu í C, örlítið í Octave og límt saman í Bash. Til að byrja með flokkum við myndirnar í forritunum með gildum úr $[0, 1]$, þannig að 0 er nei og 1 er já fyrir gefinn flokk.

4.1 Að þekkja fólk með sólgleraugu

Munum að við höfum 624 myndir; þar af eru 311 með sólgleraugu og 313 án sólgleraugna. Við sjáum að fjöldi mynda í hverjum flokki m.t.t. sunglasses/open skiptingarinnar (tafla 1 bls. 4) er nokkuð jafn (gerum ráð fyrir að notandinn skipti ekki máli). Allt bendir því til að tauganetið fái jafna möguleika á því að læra hvern flokk.

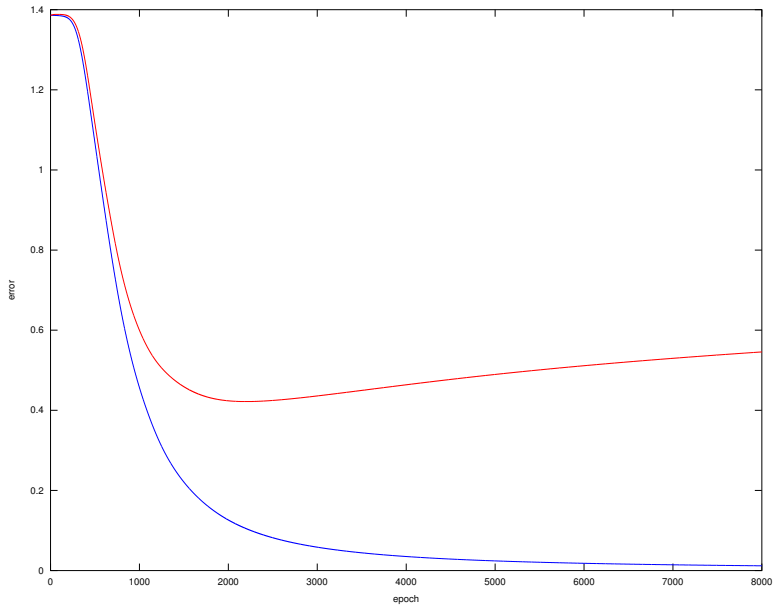
Þjálfum nú tauganetið við eftirfarandi skilyrði:

- Inntak er myndir í lægstu upplausn, hæðin er 30 punktar og breiddin er 32 punktar.

	sunglasses	open
straight	78	78
left	78	79
right	78	77
up	77	79
neutral	78	80
happy	77	78
sad	78	78
angry	78	77

Tafla 1: Dreifing á sólgleraugum milli flokka

- Úttakið er einn flokkur, þ.e. hvort einstaklingur er með sólgleraugu
- Eitt falið lag og með tveimur hnútum
- 8000 ítranir, lærdómshraði $\eta = 0.1$ og skriðþungi $\mu = 0.3$.
- Þjálfunargögnin eru um 70% af inntakinu, þ.e. 437 myndir; hinar 187 eru notaðar í prófun.



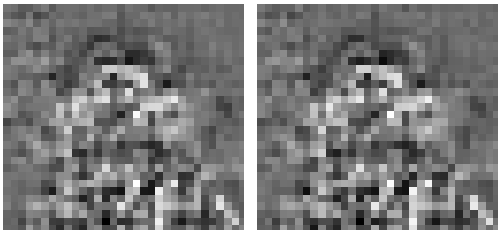
Mynd 2: Þjálfunar- (blá) og prófunarskekkjan (rauð) þegar við látum netið læra að þekkja flokkana *sunglasses* og *open* með eitt tveggja-hnúta millilag, $\eta = 0.1$ og $\mu = 0.3$; 8000 umferðir. Skekkjan dettur í lokin niður í um 0.01. Sjáum auk þess að að prófunarskekkjan fer að vaxa upp úr 1000 ítrunum; það þýðir að netið fer að *oflæra* gögnin upp úr þeim fjölda ítrana.

Hnútarnir í millilaginu virðast taka mjög svipuð gildi og úttakshnúturinn fyrir ekki-sólgleraugu, en úttaksgildin þjappast frekar upp að endapunktunum 0 og 1 í ákvörðunarbilinu $[0, 1]$. Til dæmis fékkst fyrir millilagið (slembið prófunarúrtak) í sömu keyrslu og þeirri sem gaf mynd 3:

```
Node activations:   0.010169 0.008465
output activations: 0.998648 0.001352
Target activations: 1.000000 0.000000

Node activations:   0.014515 0.010832
output activations: 0.998579 0.001421
Target activations: 1.000000 0.000000

Node activations:   0.923234 0.936433
output activations: 0.000576 0.999425
Target activations: 0.000000 1.000000
```



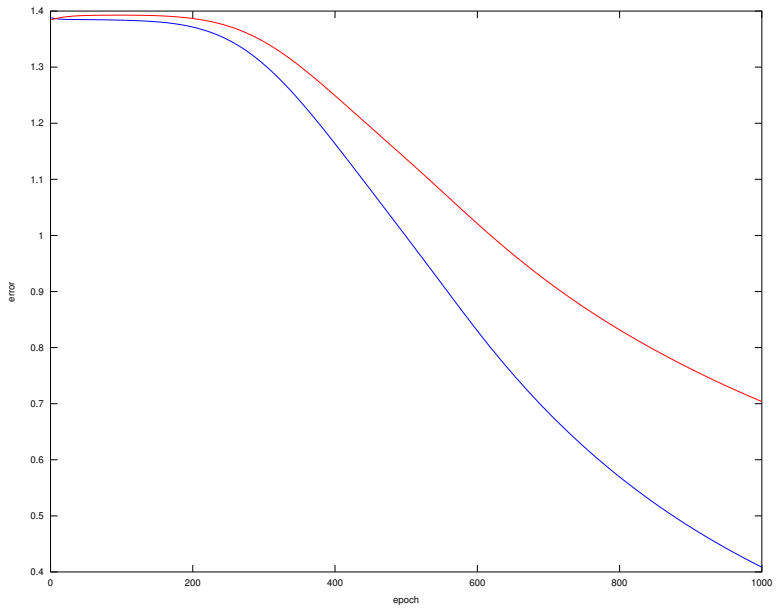
Mynd 3: Þyngdirnar frá inntaki yfir á millilag (fyrri myndin er fyrri línuvigurinn) þegar læra skal flokkana *sunglasses* og *open*. Mjög dökkir litir svara til neikvæðra þyngda og ljósir litir svara til jákvæðra þyngda, en meðalgráir litir eru líklega þyngdir í grennd við 0. Veitum því sérstaklega athygli að sólgleraugun fá ekkert frekar jaðarþyngdir (þ.e. hvít/svört gildi) en aðrir punktar;

Af grafinu í mynd 2 er nokkuð ljóst að við erum að *oflæra* gögnin. Af þyngdunum á mynd 3 sést að netið tekur greinilega tillit til fleiri atriða en sólgleraugna eingöngu. Prófum þess í stað að hætta eftir 1000 ítranir. Þróun villunar má þá sjá á mynd 4 og tilheyrandi þyngdir má sjá á mynd 5.

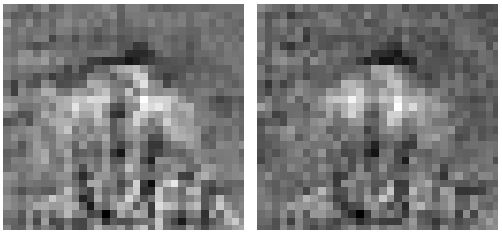
4.2 Að læra alla 10 flokkana

Reynum nú að láta tauganetið læra alla 10 flokkana í einu. Fyrsta tilraun verður þannig:

- Lægsta upplausn, 30 raðir og 32 dálkar í mynd.
- Eitt 10-hnúta millilag.
- 8000 ítranir.
- $\eta = 0.1$, $\mu = 0.3$.
- 70% úrtaks, 437 myndir, til þjálfunar; 30%, 187 myndir, til prófunar.



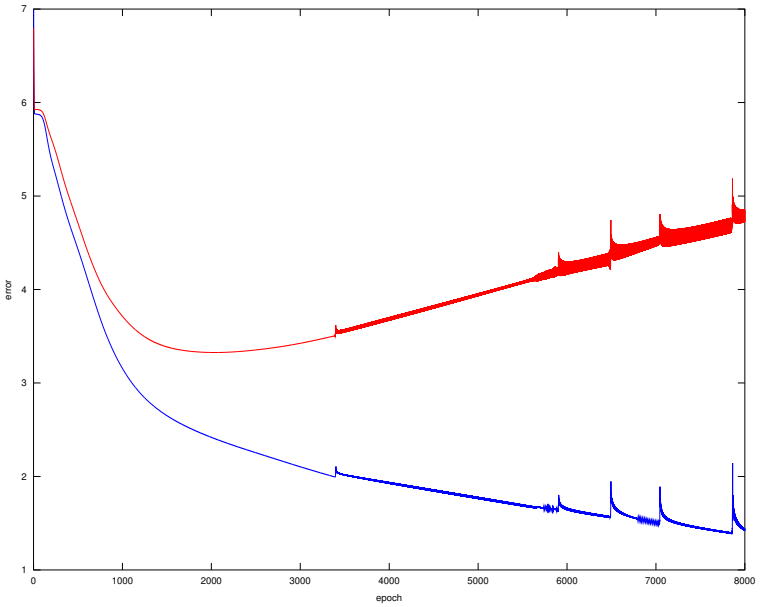
Mynd 4: 1000 ítranir, $\eta = 0.1, \mu = 0.3$. Flokkarnir *sunglasses* og *open*



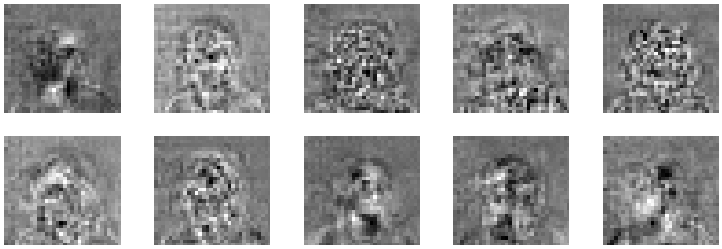
Mynd 5: 1000 ítranir, $\eta = 0.1, \mu = 0.3$. Flokkar *sunglasses* og *open*. Í ljósi þess að við viljum þekkja sólgleraugu, þá er þetta eðlilegra en þegar teknar voru 8000 ítranir (ljósir blettir við augun).

Þróun skekkjunnar má sjá á mynd 6 og þyngdirnar á mynd 7. Rétt eins og í síðustu undirgrein sjáum við að við fórum allt of margar ítranir, því prófunarskekkjan staðnar greinilega (fyrir utan sveiflur) eftir um 1000 ítranir, og þyngdirnar bera merki þess að reyna að læra of flókin mynstur (t.d. fá axlir vægi, sem er óeðlilegt).

Prófum nú á móti að fara 1000 ítranir við annars sömu skilyrði. Niðurstöðurnar eru á myndum 8 og 9.



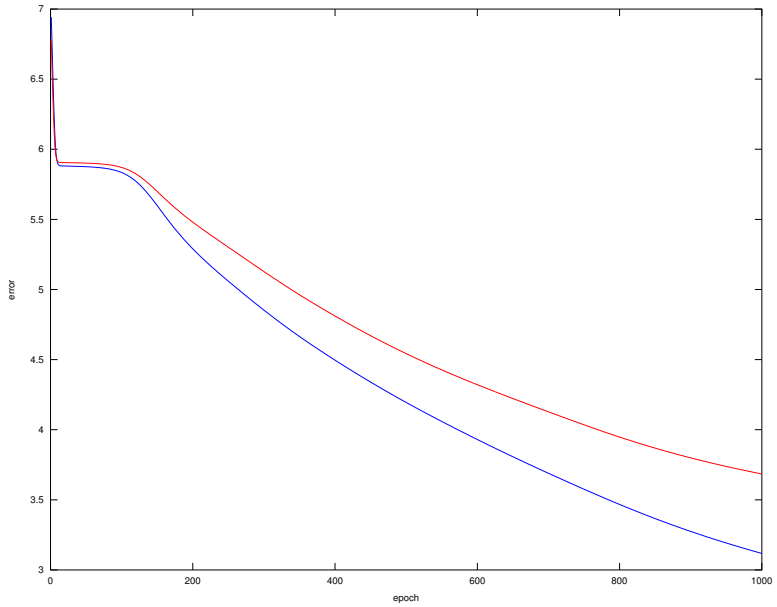
Mynd 6: 8000 ítranir, $\eta = 0.1, \mu = 0.3$. Allir flokkar. Sjáum að við byrjum að oflæra strax upp úr 1000 ítrunum.



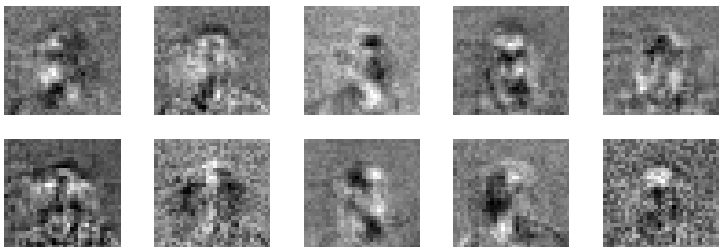
Mynd 7: 8000 ítranir, $\eta = 0.1, \mu = 0.3$. Allir flokkar. Gera má sér í hugarlund að myndin efst til vinstri gefi þeim sem horfa til vinstri meira vægi, og eins að myndin hægra megin við hana, gefi þeim vægi sem horfa til áfram. Mynstrin eru þó frekar óskýr og netið er greinilega að oflæra.

4.3 Túlkun á niðurstöðum

Með því að skoða þyngdirnar á myndum 4 og 8 nánar þá sjáum við að netið virðist fyrst og fremst læra öfgapunkta, þ.e. mjög ljósa og mjög dökka punkta, í myndunum sem uppfylla jafnframt það skilyrði að vera frekar svipaðir í gegn um dæmin, með auknu vægi á þá punkta sem hafa fylgni með réttum-úttaksgildum. Til dæmis er húðliturinn almennt ljós og afmarkar útlínur höfuðsins, sem er fylgist að með stefnu höfuðs, svo við fáum almennt eitthvað sem líkist höfði á



Mynd 8: 1000 ítranir, $\eta = 0.1, \mu = 0.3$. Allir flokkar.



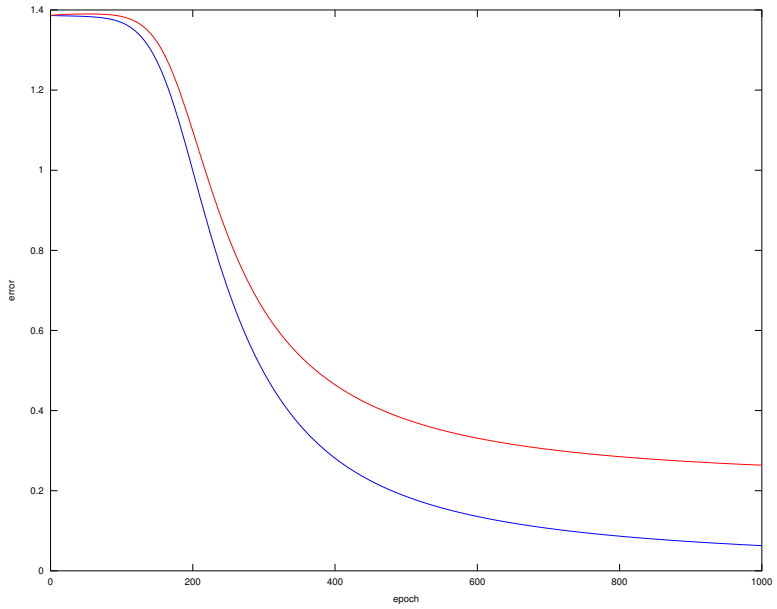
Mynd 9: 1000 ítranir, $\eta = 0.1, \mu = 0.3$. Allir flokkar.

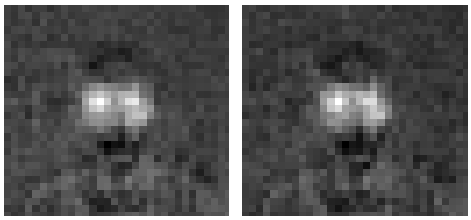
þyngdirnar.

Eins, þá eru sólglerugun áberandi dökk miðað við annars frekar ljóst svæði við gleraugnalaus augu, svo við ættum að fá frekar sterkan mun milli augnsvæðisins og umhverfisins. Það er vissulega svo á mynd 4, en sú mynd líður þó fyrir það að *sólgleraugun færast til með stefnu höfuðsins*. Til að undirstrikra þessa fylgni, sjá mynd 10, sem sýnir hversu mikið auðveldara verkefnið verður þegar við takmörkum sólgleraugnaverkefnið við andlit sem snúa beint áfram. Reyndar bendir þetta líka til þess að við gætum e.t.v. notað sólgleraugnaflokkinn til að hjálpa netinu að læra hvernig höfuðið á myndinni snýr (því hann kemur svo sterkur og staðbundinn inn).

Að lokum, athugum að netinu gengur afskaplega illa að læra svipbrigðin á myndunum; við því var þó að búast vegna þess að upplausnin í myndunum

er lítil, gildin í punktunum með svipbrigðum dreifast yfir stóran hluta $[0, 1]$, og netið lærir óháð röð punkta í myndinni (svo okkur gengur alveg eins hvernig sem við umröðum punktunum, svo lengi sem breytingni varðveitist milli mynda).





Mynd 10: Sólgleraugnaverkefnið þar sem úrtakið er takmarkað við andlit sem snúa beint áfram. Netið er fljótara að læra (bara 1000 ítranir að komast niður fyrir 0.1 í þjálfunarskekkju) og það gefur eðlilegri þyngdir m.t.t. verkefnisins (nánast bara sólgleraugun fá vægi).

4.4 Lágmörkun á prófunarskekkju

Athugum að það, að reyna að tákna myndirnar með gildum úr $[0, 1]$ þannig að 0 er nei og 1.0 er já, er ekki endilega ekki. Við höfum nefnilega einhalla færslufallið h sem uppfyllir $h(x) \to 0$ þegar $x \to -\infty$ og $h(x) \to 1$ þegar $x \to +\infty$; svo til að komast sem næst þeim gildum gætum við lent í að klemma gildin í $\mathbf{w}$ við $\pm\infty$. Með því að nota annað skekkjumat en cross-entropy kæmi til greina að láta gildin dreifast á bilið $[0.1, 0.9]$, því það eru gildi sem h getur náð í endanlegum stærðum. Með cross-entropy kæmi það hins vegar ekki til greina, því við erum

háð h í frumstillingu δ_k. Reyndar vill líka svo til að þyngdirnar okkar verða ekki svo stórar í dæmunum að ofan (þrátt fyrir hættuna á því). Önnur leið væri að búa til nýtt skekkjumat

$$G(\mathbf{w}) = \alpha E(\mathbf{w}) + \beta M(\mathbf{w})$$

þar sem $M(\mathbf{w})$ er svokallaður *regularizer*, sem gegnir því hlutverki að halda þyngdunum í skefjum. Ekki gafst tími til að kanna þessar hugmyndir nánar.

4.5 Varðandi önnur gildi á η, μ o.fl.

Fleiri gildi á öllum stikum voru prófuð við gerð verkefnisins en niðurstöðurnar ekki geymdar og því ekki settar fram hér. Almennt virðist sem hærri gildi á η gefi hraðari samleitni upp að vissu marki, en örar og háar sveiflur koma fram í bæði þjálfunar- og prófunarskekkju eftir því sem η eykst. Að lokum hættir netið að geta lært og skekkjan helst föst í upphafsgildi. Þannig er hægt að ná fram svipaðri samleitni og í 8000 ítrunum fyrir $\eta = 0.1, \mu = 0.3$ að ofan, í 2000 ítrunum fyrir svolítið hærri gildi á η, en samleitnin verður óstöðug á köflum. Skriðþunginn μ virðist hjálpa til við samleitni í tilfellum þegar η er mjög lítið og jafnvel geta minnkað óreiðukenndar sveiflur í skekkju, en að sama skapi getur hann gert illt verra fyrir stærri η. Varðandi millilög netsins, þá fengust bestu niðurstöðurnar með því að hafa eitt millilag með jafnmörgum hnútum og flokkarnir sjálfir; annars gekk ágætlega að leysa t.d. sólgleraugnaverkefnið með engu millilagi. Ekki var betra að bæta við fleiri lögum, en þar kenni ég þó um skorti á *regularizer* fyrir skekkjuna (fallið M úr seinustu undirgrein); enda hefur tauganet með fleiri lögum meira *minni*, þ.e. getur lært fleiri og flóknari hluti.

A `vbp.m` með cross-entropy

```
function [net,ep] = vbp(A, T, n, noepochs, eta, mu, oldnet)
%% VBP
%   vanilla backpropagation batched training a feed forward multilayer
%   neural network using the logistic sigmoid transfer function.
%
%% usage
%   [net,error] = vbp(A, T, n, noepochs, eta, mu, oldnet);
%
%% where
%   A        : input layer activation
%   T        : output layer activation
%   n        : number of neurons in hidden layers (vector)
%   noepochs : number of iterations through all training patterns
%   eta      : learning rate \in [0 2]
%   mu       : momentum term \in [0 1]
%
%% XOR example (one hidden layer with two neurons):
%   X         = [0 0;0 1;1 0;1 1]';
%   Y         = xor(X(1,:),X(2,:));
%   [net,ep]  = vbp(X,Y,[2],2000,1.3,0);
%   plot(ep); shg;
```

```
%   [Yest, A] = vbp(X,net);
%   Yest, Y
%
%% AND example (no hidden layers):
%   X         = [0 0; 0 1; 1 0; 1 1]';
%   Y         = X(1,:) & X(2,:);
%   [net,ep]  = vbp(X,Y,[],2000,1.3,0);
%   plot(ep); shg;
%   [Yest, A] = vbp(X,net);
%   Yest, Y

% Copyleft - tpr 2004

    if (nargin == 2)
        % Forward sweep for previously trained neural network (ie use that
        % network, T, to estimate some classification function.) The
        % roles of variables net and T are reversed in this case.
        for i = 1:size(A,2)
            T(1).a = A(:,i);       % init the input, then do forward sweep
            for l = 2:length(T(1).n)
                T(l).a = logistic_sigmoid( T(l).W * [ 1; T(l-1).a ] );
            end
            net(:,i)  = T(end).a;  % the neural network output
            ep(i).net = T;         % for showing hidden node activations (DEMO)
        end
        return; % Bail out!
    end

    n   = [size(A,1) n size(T,1)]; % number of input/output activations added
    L   = length(n);               % number of layers
    ell = size(A,2);               % number of training samples

    if (ell ~= size(T,2))
        error('number of X patterns does not match Y');
    end

    %% initialize weight matrices
    net(1).n = n;
    for l=2:L
        if (nargin == 7)
            % use last network for initialization
            net(l).W      = oldnet(l).W;
            net(l).dWlast = oldnet(l).dWlast; % last weight update for momentum
        else
            % fresh start
            net(l).W      = 2.0*rand(n(l), n(l-1)+1) - 1.0;
            net(l).dWlast = zeros(n(l), n(l-1)+1);
        end
    end
```

```
    %% start training
    for epoch = 1:noepochs
        % zero the change in weight
        for l = 2:L
            net(l).dW = zeros(size(net(l).W));
        end
        %% loop through each pattern
        for p = 1:ell
            % do a complete forward sweep
            net(1).a = A(:,p);
            for l = 2:L
                net(l).netsum = net(l).W * [1; net(l-1).a];
                net(l).a      = logistic_sigmoid(net(l).netsum);
            end
            %% calculate errors
            t   = T(:,p);
            cross_entropy(p) = -( t * log(net(L).a) + (1-t) * log(1-net(L).a) );
            % signal errors delta (backprop)
            net(L).delta   = t - net(L).a;
            delta          = net(L).delta;
            for l = (L-1):-1:2
                tmp_dls      = dlogistic_sigmoid([0; net(l).netsum]);
                tmp_sum      = sum( (delta*ones(1,n(l)+1)) .* net(l+1).W, 1 )';
                delta        = tmp_dls .* tmp_sum;
                delta        = delta(2:end);
                net(l).delta = delta;
            end
            %% cumulate the weight changes (batched learning)
            for l = 2:L
                net(l).dW = net(l).dW + net(l).delta * [1; net(l-1).a]';
            end
        end
        %% update the network weights
        for l = 2:L,
            net(l).dW     = eta * net(l).dW / ell + mu * net(l).dWlast;
            net(l).W      = net(l).W + net(l).dW;
            net(l).dWlast = net(l).dW;
        end
        %% store mean error per epoch
        %ep(epoch) = mean(rms_error);
        ep(epoch) = mean(cross_entropy);
    end

function [a] = logistic_sigmoid(n)
% logistic sigmoid transfer function
    a = 1 ./ (1+exp(-n));

function [d] = dlogistic_sigmoid(n)
% logistic sigmoid transfer derivative function
    a = logistic_sigmoid(n);
```

```
    d = a .* (1-a);
```

B XOR-prófið

```
%% XOR example from vbp.m (one hidden layer with two neurons)
X         = [0 0;0 1;1 0;1 1]';
Y         = xor(X(1,:),X(2,:));
[net,ep]  = vbp(X,Y,[2],2000,1.3,0);
[Yest, A] = vbp(X,net);
Yest, Y

plot(ep, 'b', 'LineWidth', 2);
print('xor_example.eps', '-color');
```

C VBP útfært í C

Virkar í GCC-4.4 og ICC-12.0.0 á nýlegu Ubuntu. Þarf útfærslu á BLAS [2] til að keyra.

C.1 Skilgreiningaskjalið `hfy_vbp.h`

```
/*
 *
 *
 * TYPEDEFS
 *
 *
 */

typedef struct _vbp_neural_layer {
    double *a;        // node activations
    double *w;        // (nodes x (nodes_in+1)) weight matrix
    double *dw;       // (nodes x (nodes_in+1)) delta weight matrix
    double *dwlast;   // (nodes x (nodes_in+1)) last delta weight matrix

    int nodes;        // number of non-bias nodes in this layer
    int nodes_in;     // number of non-bias nodes in the previous layer

    // NOTE: every layer has a bias-node, namely node number 0; we always have
    // a[0]=1.0 and w[0] is always the bias.

    double *delta;    // error deltas
    double *netsum;   // network activation

} vbp_neural_layer;

typedef struct _vbp_neural_network {
    int number_of_layers;
    vbp_neural_layer *layers;
```

```
} vbp_neural_network;

/*
 *
 *
 * MEMORY MANAGEMENT
 *
 *
 */

/*
 * Allocates memory to the contents of a single, given, layer. Useful when
 * allocating an array of neural layer structs.
 */
void vbp_allocate_layer_contents( int nodes, int nodes_in,
        vbp_neural_layer *layer);

/*
 * Allocates memory for the contents of the neural network `net` with
 * `number_of_layers` layers, such that layer `i` has `nodes[i]` nodes.
 * Useful when allocating an array of neural network structs.
 */
void vbp_allocate_net_contents(int number_of_layers,
        int *nodes, vbp_neural_network *net);

/*
 * Allocates memory for a new network and returns a pointer to the memory
 */
vbp_neural_network *vbp_allocate_net(int number_of_layers, int *nodes);

/*
 * Free the memory held by the layer `layer` and its contents, assuming `layer`
 * is not the first layer in every network containing it.
 */
void vbp_free_layer_contents(vbp_neural_layer *layer);

/*
 * Free the memory held by the network `net` and its contents, assuming the
 * network to be well formed.
 */
void vbp_free_net(vbp_neural_network *net);

/*
 *
 *
 * NEURAL NETWORK FUNCTIONS
 *
 *
 */
```

```
/*
 * Perform a single forward sweep on the network ‘net‘ with input ‘x‘, and, if
 * y!=NULL, copy the output to y.
 */
void vbp_forward_sweep( double *x, double *y, vbp_neural_network *net);

/*
 * Back-propagate the weights for the network ‘net‘, which _must_ by fully
 * allocated on entry.
 *
 * If use_old_weights != 0 then we continue using the weights already set in
 * the network (this, of course, requires them to be well defined). Otherwise,
 * train using new weights.
 *
 */
void vbp(double **input,
        double **target,
        int number_of_patterns,
        int number_of_layers,
        int number_of_epochs,
        double eta,
        double mu,
        vbp_neural_network *net,
        double *net_errors,
        int use_old_weights);

double logistic_sigmoid(double x);
double dlogistic_sigmoid(double x);
double cross_entropy(int n, double *target, double *output);
double norm(int n, double *target, double *output);


/*
 *
 *
 * UTILITY FUNCTIONS
 *
 *
 */

/*
 * Generate a random permutation using Fisher-Yates shuffle
 */
void vbp_perm(int n, int* perm);
```

C.2 Útfærsluskjalið `hfy_vbp.c`

```
// #include <stdio.h>
#include <stdlib.h>
```

```
#include <stdarg.h>
#include <math.h>
#include <time.h>
#include <cblas.h>

#include "hfy_vbp.h"

void vbp_allocate_layer_contents( int nodes, int nodes_in,
        vbp_neural_layer *layer)
{
    int asize = (1+nodes);
    int wsize = nodes*(nodes_in+1);
    int dsize = nodes;

    layer->nodes = nodes;
    layer->nodes_in = nodes_in;

    // use calloc so that all matrices and vectors start at 0.0
    layer->a = (double*)calloc(asize, sizeof(double));
    layer->w = (double*)calloc(wsize, sizeof(double));
    layer->dw = (double*)calloc(wsize, sizeof(double));
    layer->dwlast = (double*)calloc(wsize, sizeof(double));

    layer->delta = (double*)calloc(dsize, sizeof(double));
    layer->netsum = (double*)calloc(dsize, sizeof(double));

    (layer->a)[0] = 1.0;
}

void vbp_allocate_net_contents(int number_of_layers, int *nodes,
        vbp_neural_network *net)
{
    // local variables
    int i;

    net->number_of_layers = number_of_layers;
    // allocate layers
    vbp_neural_layer *layer = (vbp_neural_layer*)
        calloc(number_of_layers, sizeof(vbp_neural_layer));
    net->layers = layer;

    // allocate the contents of the first layer
    // (the first layer is always different)
    layer->a = (double*)calloc((1+nodes[0]), sizeof(double));
    (layer->a)[0] = 1.0;
    layer->nodes = nodes[0];
    layer->nodes_in = 0;
    layer->w = layer->dw = layer->dwlast = NULL;

    // allocate the contents of the rest of the layers
```

```
    for(i = 1; i < number_of_layers; ++i) {
        vbp_allocate_layer_contents(nodes[i], nodes[i-1], layer+i);
    }
}

vbp_neural_network *vbp_allocate_net(int number_of_layers, int *nodes) {
    vbp_neural_network *net =
        (vbp_neural_network*)calloc(1,sizeof(vbp_neural_network));
    vbp_allocate_net_contents(number_of_layers, nodes, net);
    return net;
}

void vbp_free_layer_contents(vbp_neural_layer *layer) {
    free(layer->a);
    free(layer->delta);
    free(layer->w);
    free(layer->dw);
    free(layer->dwlast);
    free(layer->netsum);
}

void vbp_free_net(vbp_neural_network *net) {
    vbp_neural_layer *layer = net->layers;
    const vbp_neural_layer *END = layer + (net->number_of_layers);
    // Free the first layer contents
    free(layer->a);
    // Free the contents of the other layers
    for(++layer; layer < END; ++layer) {
        vbp_free_layer_contents(layer);
    }
    // Memory allocated to the structs themselves
    free(net->layers); // all the layer structs were allocated together
    free(net);
}

double logistic_sigmoid(double x) {
    return 1.0 / (1.0 + exp(-x));
}

double dlogistic_sigmoid(double x) {
    double y = logistic_sigmoid(x);
    return y * (1.0 - y);
}

double cross_entropy(int n, double *target, double *output) {
    int i;
    double result;
    result = 0.0;
    for(i = 0; i < n; ++i) {
        result -= target[i]*log(output[i]) + (1-target[i])*log(1-output[i]);
```

```
    }
    return result;
}

double norm(int n, double *target, double *output) {
    int i;
    double sqsum, diff;
    sqsum = 0.0;
    for(i = 0; i < n; ++i) {
        diff = target[i]-output[i];
        sqsum += diff*diff;
    }
    return sqrt(sqsum);
}

void vbp_forward_sweep( double *x, double *y, vbp_neural_network *net) {
    // local variables
    int i,j;
    vbp_neural_layer *layer = net->layers;
    vbp_neural_layer *END = (net->layers) + (net->number_of_layers);

    // (layer->a)[1..] = x
    cblas_dcopy(layer->nodes, x, 1, (layer->a)+1, 1);
    layer->a[0] = 1.0; // for the bias node

    // forward sweep
    for(++layer; layer < END; ++layer) {
        cblas_dgemv(CblasColMajor, CblasNoTrans,
                layer->nodes, (layer->nodes_in)+1,
                1.0, layer->w, layer->nodes, (layer-1)->a, 1,
                0.0, layer->netsum, 1);

        // evaluate layer activations
        for(i = 0; i < layer->nodes; ++i) {
            (layer->a+1)[i] = logistic_sigmoid((layer->netsum)[i]);
            // printf("%f ", (layer->a+1)[i]);
        }
        // printf("\n");
        layer->a[0] = 1.0; // for the bias node
    }
    --layer;
    if(y != NULL) {
        cblas_dcopy(layer->nodes, (layer->a)+1, 1, y, 1);
    }
}

void vbp(double **input, double **target,
        int number_of_patterns,
        int number_of_layers,
        int number_of_epochs,
```

```
        double eta, double mu,
        vbp_neural_network *net,
        double *net_errors,
        int use_old_weights)
{
    // local variables
    int i,j, epoch, pattern, wsize;
    double *t;
    const double ZERO [] = { 0 };
    vbp_neural_layer *layer;
    vbp_neural_layer *END;

    // init the network
    END = (net->layers) + number_of_layers;
    if(use_old_weights == 0) {
        // initalize weights to random small numbers
        for(layer = net->layers+1; layer < END; ++layer) {
            wsize = (layer->nodes)*((layer->nodes_in)+1);
            for(i = 0; i < wsize; ++i) {
                (layer->w)[i] = 0.05*((2.0*rand())/RAND_MAX - 1.0);
                (layer->dwlast)[i] = 0.0;
            }
        }
    }

    // start training
    for(epoch = 0; epoch < number_of_epochs; ++epoch) {

        net_errors[epoch] = 0.0;

        // zero the change in weight
        for(layer = net->layers+1; layer < END; ++layer) {
            wsize = (layer->nodes)*((layer->nodes_in)+1);
            cblas_dcopy(wsize, ZERO, 0, layer->dw, 1);
        }

        // loop through each pattern
        for(pattern = 0; pattern < number_of_patterns; ++pattern) {
            t = target[pattern];

            // do a complete forward sweep
            vbp_forward_sweep(input[pattern], NULL, net);

            // top layer (output nodes)
            layer = END-1;

            // cross-entropy
            net_errors[epoch] += cross_entropy(layer->nodes, t, layer->a+1);

            // top layer deltas (cross-entropy)
```

```
            cblas_dcopy(layer->nodes, t, 1, layer->delta, 1);
            cblas_daxpy(layer->nodes, -1.0, (layer->a)+1, 1, layer->delta, 1);

            // back-propagate the lower layer deltas
            for(--layer; layer > net->layers; --layer) {
                for(i = 0; i < layer->nodes; ++i) {
                    (layer->delta)[i] =
                        dlogistic_sigmoid((layer->netsum)[i])
                        * cblas_ddot((layer+1)->nodes,
                            (layer+1)->delta, 1,
                            (layer+1)->w + (i+1)*((layer+1)->nodes), 1);
                }
            }
            // cumulate the weight changes (batched learning)
            for(++layer; layer < END; ++layer) {
                cblas_dger(CblasColMajor,
                        layer->nodes, (layer->nodes_in)+1,
                        1.0, layer->delta, 1, (layer-1)->a, 1,
                        layer->dw, layer->nodes);
            }
        }
        // update the network weights
        for(layer = net->layers+1; layer < END; ++layer) {
            wsize = (layer->nodes)*((layer->nodes_in)+1);

            // update dw
            cblas_dscal(wsize, eta/number_of_patterns, layer->dw, 1);
            cblas_daxpy(wsize, mu, layer->dwlast, 1, layer->dw, 1);

            // update w
            cblas_daxpy(wsize, 1.0, layer->dw, 1, layer->w, 1);

            // update dwlast
            cblas_dcopy(wsize, layer->dw, 1, layer->dwlast, 1);
        }
        // mean error per epoch
        net_errors[epoch] /= number_of_patterns;
    }
}

void vbp_perm(int n, int *perm) {
    int i;
    srand(time(NULL));

    for(i = n-1; i > 0; --i) {
        perm[i] = rand()%(i+1);
    }
}
```

D Keyrsluforrit fyrir andlitsprófin

Til að keyra forritin þarf nýlega útgáfu libnetpbm [3] auk `hfy_vbp` ofan. Forritin má t.d. þýða með

```
cc -O3 hfy_loadfaces.c hfy_vbp.c -lblas -lnetpbm -o loadfaces
```

fyrir gcc, clang o.fl. á móti atlas í ubuntu; eða

```
icc -fast hfy_loadfaces.c hfy_vbp.c -mkl -lnetpbm -o loadfaces
```

fyrir icc á móti mkl.

D.1 Útfærsluskjalið `hfy_loadfaces.c`

```
#include <pam.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>

#include <getopt.h>

#include "hfy_vbp.h"

/*
 * NOTE: There are a couple of types, such as 'sample' and 'tuple' which are
 * imported from 'pam.h'
 */

/*
 * Classify the first n paths into classes, such that path p is in class c if c
 * is a substring of the filename in p
 */
void classify_paths(const int N, int number_of_classes,
        char **classes, char **paths, double **target)
{
    int i, j;
    char *start;
    for(i = 0; i < N; ++i) {
        // init path
        start = strrchr(paths[i] , '/');
        if(start == NULL)
            start = paths[i];

        for(j = 0; j < number_of_classes; ++j) {
            if(strstr(start, classes[j]) != NULL)
                target[i][j] = 1.0;
            else
                target[i][j] = 0.0;
        }
    }
```

```
}

/*
 * Reads n image files with paths in 'paths'; the images should only be
 * pgm-files and they should all have the same dimensions. The arrays maxvals,
 * imdata and imnormal should already point to storage reserved for the output
 * data.
 */
void loadfaces(char **paths, int n, int *maxvals,
        unsigned long **imdata, double **imnormal)
{
    // Local variables
    FILE *buf;

    int i, row, column, pos;
    unsigned long sample_val;

    struct pam imstruct;
    tuple **imtuple;

    // read the files
    for(i = 0; i < n; ++i) {
        // open image at path[i]
        buf = fopen(paths[i], "r");
        // allocate space for the next image and retrieve it from buf
        imtuple = pnm_readpam(buf, &imstruct, PAM_STRUCT_SIZE(tuple_type));
        // close the image
        fclose(buf);

        // from the image tuple to the output
        maxvals[i] = imstruct.maxval; // value of white
        pos = 0; // position within imdata[i] and imnormal[i]
        for(row = 0; row < imstruct.height; ++row) {
            for(column = 0; column < imstruct.width; ++column) {
                sample_val = imtuple[row][column][0];
                imdata[i][pos] = sample_val;
                imnormal[i][pos] = ((double)sample_val)/maxvals[i];
                ++pos;
            }
        }

        // free the allocated space
        pnm_freepamarray(imtuple, &imstruct);
    }
}

/*
 * -n, --nodes n1,...,nN
 *  Comma separated list of nodes per hidden layer, NO SPACES ALLOWED.
 *  The number of layers is deduced from the number of arguments given to this
```

```
 *  option.
 *
 * -E, --epochs E
 *  Number of epochs.
 *
 * -e, --eta eta
 *  Learning rate, a floating point value between 0.0 and 2.0
 *
 * -m, --mu mu
 *  Learning momentum, a floating point value between 0.0 and 1.0
 *
 * -f, --files f1,...,fN
 *  Comma separated list of files, NO SPACES ALLOWED. Each filename in path f
 *  must contain the classes c that the it is part of (and no other classes).
 *
 * -c, --classes c1,...,cN
 *  Comma separated list of classes, NO SPACES ALLOWED.
 *
 * -x, --training-error f
 * -y, --testing-error g
 *  Write training/testing errors to files f/g
 *
 * -w, --weight-img f
 *  Write transfer weights from input to the next layer as an image file f.
 *  This works because each transfer can be visualized using images by
 *  reshaping the column-vectors of the first weight-matrix and scaling the
 *  values of each vector between 0 and 255.
 *
 */
int main(int argc, char *argv[]) {
    // insignificant variables
    int i, j, k, n;
    int *perm; // permutation
    void *tmp;

    // gnu getline stuff
    char short_options [] = "n::E:e:m:f:c:x:y:w:";
    struct option long_options [] = {
        { "nodes" ,         optional_argument, 0, 'n' },
        { "epochs",         required_argument, 0, 'E' },
        { "eta" ,           required_argument, 0, 'e' },
        { "mu",             required_argument, 0, 'm' },
        { "files",          required_argument, 0, 'f' },
        { "classes",        required_argument, 0, 'c' },
        { "training-error", required_argument, 0, 'x' },
        { "testing-error",  required_argument, 0, 'y' },
        { "weight-img",     required_argument, 0, 'w' },
        {0, 0, 0, 0}
    };
    int c, option_index;
```

```
FILE *buf;
struct pam imstruct;
tuple **imtuple;


// user configurable through command line arguments (argv)
int *nodes;
int number_of_layers;
int number_of_epochs;
double eta; // learning rate
double mu; // momentum
char **paths; // input files
char **classes; // actual class strings
char *trainerror_path;
char *testerror_path;
char *weight_img_path;

// other variables
int imcount; // total number of images
int imsize; // total pixel count of each image (constant)
int number_of_classes; // total number of classes

int *maxvals; // values of white for the images
unsigned long **imdata;
double **imnormal;
double **target; // target classifications

vbp_neural_network *net;

// training and testing splits
int training_size;
double **training_input;
double **training_target;
double *training_error;

int testing_size;
double **testing_input;
double **testing_target;
double *testing_output;
double *testing_error;

char tmpstr[1000];

// init netpbm
pm_init(argv[0], 0);

// read command line arguments (black magic)
while(1) {
    c = getopt_long(argc, argv, short_options, long_options, &option_index);
```

```
if(c == -1)
    break;

switch(c) {

    case 0:
        break;

    case 'f':
        // count the paths
        i = 0; // position within optarg
        imcount = 1; // number of files is 1 + number of commas
        while(optarg[i] != ' ' && optarg[i] != '\0') {
            if(optarg[i] == ',')
                imcount++;
            i++;
        }
        // imcount = number of files
        paths = (char**)malloc(imcount*sizeof(char*));

        // read the paths
        i = 0; // position within optarg
        j = 0; // length of path[k-1]
        k = 0; // number of paths read so far
        while(k != imcount) {
            j = strcspn(optarg+i, ", ");
            paths[k] = (char*)malloc((j+1)*sizeof(char));
            strncpy(paths[k], optarg+i, j);
            paths[k][j] = '\0';
            i += j+1;
            k++;
        }
        break;

    case 'c':
        // count the classes
        i = 0; // position within optarg
        number_of_classes = 1; // number of classes is 1 + number of commas
        while(optarg[i] != ' ' && optarg[i] != '\0') {
            if(optarg[i] == ',')
                number_of_classes++;
            i++;
        }
        classes = (char**)malloc(number_of_classes*sizeof(char*));

        // read the paths
        i = 0; // position within optarg
        j = 0; // length of class[k-1]
        k = 0; // number of classes read so far
        while(k != number_of_classes) {
```

```
            j = strcspn(optarg+i, ", ");
            classes[k] = (char*)malloc((j+1)*sizeof(char));
            strncpy(classes[k], optarg+i, j);
            classes[k][j] = '\0';
            i += j+1;
            k++;
        }
        break;

    case 'n':
        number_of_layers = 2;

        if(optarg != NULL) {
            // count the hidden layers
            i = 0; // position within optarg
            number_of_layers = 3; // number of layers is 3 + number of commas
            // since we always have input and output
            while(optarg[i] != ' ' && optarg[i] != '\0') {
                if(optarg[i] == ',')
                    number_of_layers++;
                i++;
            }
        }

        nodes = (int*)calloc(number_of_layers,sizeof(int));

        if(optarg != NULL) {
            // read the number nodes per hidden layer
            i = 0; // position within optarg
            j = 0; // number of digits in the number
            k = 1; // number of layers so far
            while(k != number_of_layers-1) {
                j = strcspn(optarg+i, ", ");
                strncpy(tmpstr, optarg+i, j);
                tmpstr[j] = '\0';
                nodes[k] = atoi(tmpstr);
                i += j+1;
                k++;
            }
        }
        break;

    case 'E':
        // read the number of epochs
        j = strcspn(optarg, " "); // length of the representation
        strncpy(tmpstr, optarg, j);
        tmpstr[j] = '\0';
        number_of_epochs = atoi(tmpstr);
        break;
```

```
            case 'e':
                // read eta
                j = strcspn(optarg, " "); // length of the representation
                strncpy(tmpstr, optarg, j);
                tmpstr[j] = '\0';
                eta = atof(tmpstr);
                break;

            case 'm':
                // read mu
                j = strcspn(optarg, " "); // length of the representation
                strncpy(tmpstr, optarg, j);
                tmpstr[j] = '\0';
                mu = atof(tmpstr);
                break;

            case 'x':
                // training error output path
                j = strcspn(optarg, " "); // length of the representation
                trainerror_path = (char*)malloc((j+1)*sizeof(char));
                strncpy(trainerror_path, optarg, j);
                trainerror_path[j] = '\0';
                break;

            case 'y':
                // testing error output path
                j = strcspn(optarg, " "); // length of the representation
                testerror_path = (char*)malloc((j+1)*sizeof(char));
                strncpy(testerror_path, optarg, j);
                testerror_path[j] = '\0';
                break;

            case 'w':
                // weight images output path
                j = strcspn(optarg, " "); // length of the representation
                weight_img_path = (char*)malloc((j+1)*sizeof(char));
                strncpy(weight_img_path, optarg, j);
                weight_img_path[j] = '\0';
                break;

            default:
                break;
        }
    }
    // generate a permutation
    perm = (int*)calloc(imcount,sizeof(int));
    vbp_perm(imcount, perm);
    // shuffle the paths; later on, this will allow us to split the data into
    // 70% training values and 30% testing values by simply using the first
    // 70% of each array for training and the rest as testing.
```

```
for(i = 0; i < imcount; ++i) {
    tmp = (void*)paths[i];
    paths[i] = paths[perm[i]];
    paths[perm[i]] = (char*)tmp;
}

// extract the image size from a sample image
buf = fopen(paths[0], "r");
imtuple = pnm_readpam(buf, &imstruct, PAM_STRUCT_SIZE(tuple_type));
fclose(buf);
imsize = imstruct.width * imstruct.height;
pnm_freepamarray(imtuple, &imstruct);

// input and output nodes
nodes[0] = imsize;
nodes[number_of_layers-1] = number_of_classes;

// allocate memory
maxvals = (int*)calloc(imcount, sizeof(int));
imdata = (unsigned long**)calloc(imcount, sizeof(unsigned long*));
imnormal = (double**)calloc(imcount, sizeof(double*));

target = (double**)calloc(imcount, sizeof(double*));

// training split
training_size = (int) round(0.7*imcount);
testing_size = imcount - training_size;

training_input = (double**)calloc(training_size, sizeof(double*));
training_target = (double**)calloc(training_size, sizeof(double*));
training_error = (double*)calloc(number_of_epochs, sizeof(double));

testing_input = (double**)calloc(testing_size, sizeof(double*));
testing_target = (double**)calloc(testing_size, sizeof(double*));
testing_output = (double*)calloc(number_of_classes, sizeof(double));
testing_error = (double*)calloc(number_of_epochs, sizeof(double));

net = vbp_allocate_net(number_of_layers, nodes);

for(i = 0; i < imcount; ++i) {
    imdata[i] = (unsigned long*)calloc(imsize, sizeof(unsigned long));
    imnormal[i] = (double*)calloc(imsize, sizeof(double));
    target[i] = (double*)calloc(number_of_classes, sizeof(double));

    // training and testing arrays share memory with the original data
    if(i < training_size) {
        training_input[i] = imnormal[i];
        training_target[i] = target[i];
    }
    else {
```

```
        testing_input[i-training_size] = imnormal[i];
        testing_target[i-training_size] = target[i];
    }
}

// load and classify faces
loadfaces(paths, imcount, maxvals, imdata, imnormal);
classify_paths(imcount, number_of_classes, classes, paths, target);

// train neural network
printf("Input data size is %d\n", imsize);
printf("Training with %d samples over %d epochs\n", training_size, number_of_epochs);
printf("Testing against %d samples\n", testing_size);
printf("Output classes are %d\n", number_of_classes);

// initial run to set the weights
vbp(training_input, training_target,
            training_size, number_of_layers, 0,
            eta, mu, net, training_error, 0);
// continue training
for(i = 0; i < number_of_epochs; ++i) {
    // train network for epoch number i
    // the training error is written in training_error[i]
    vbp(training_input, training_target,
            training_size, number_of_layers, 1,
            eta, mu, net, training_error+i, 1);

    // test the network against all testing data
    testing_error[i] = 0.0;

    for(j = 0; j < testing_size; ++j) {
        vbp_forward_sweep(testing_input[j], testing_output, net);
        testing_error[i] +=
            cross_entropy(number_of_classes, testing_target[j], testing_output);

        // check activations in the first non-input layer
        if(i == number_of_epochs-1) {
            // print node activations
            printf("Node activations:   ");
            for(k = 0; k < nodes[1]; ++k) {
                printf("%f ", (net->layers[1]).a[k+1]);
            }
            printf("\n");
            printf("output activations: ");
            for(k = 0; k < number_of_classes; ++k) {
                printf("%f ", testing_output[k]);
            }
            printf("\n");
            // print target vector for this input
            printf("Target activations: ");
```

```
            for(k = 0; k < number_of_classes; ++k) {
                printf("%f ", testing_target[j][k]);
            }
            printf("\n\n");
        }
    }
    testing_error[i] /= testing_size;
}

// output errors
buf = fopen(trainerror_path, "w");
for(i = 0;i < number_of_epochs; ++i) {
    fprintf(buf, "%f ", training_error[i]);
}
fprintf(buf, "\n");
fclose(buf);
buf = fopen(testerror_path, "w");
for(i = 0;i < number_of_epochs; ++i) {
    fprintf(buf, "%f ", testing_error[i]);
}
fprintf(buf, "\n");
fclose(buf);

// draw images of weights into the second layer
//
// we rely on 'imstruct' previously initialized when gathering information
// from  the sample image
//
// all images are written into the same file on the filesystem; since they
// can easily be split up later (using pamsplit from netpbm)
buf = fopen(weight_img_path, "w");
imstruct.plainformat = 1;
imstruct.maxval = 255;
imtuple = pnm_allocpamarray(&imstruct);
imstruct.file=buf;
for(k = 0; k < nodes[1]; ++k) {
    double *w = net->layers[1].w;
    double minw, maxw;
    // find the minimum and maximums of w for scaling
    minw = maxw = w[k + nodes[1]];
    for(i = 1; i <= nodes[0]; ++i) {
        if(minw > w[k + i*nodes[1]]) minw = w[k + i*nodes[1]];
        if(maxw < w[k + i*nodes[1]]) maxw = w[k + i*nodes[1]];
    }
    printf("minw=%f, maxw=%f\n", minw,maxw);
    // scale weights to take values in 0..255 and write the output to an
    // image
    n = 1;
    for(i = 0; i < imstruct.height; ++i) {
        for(j = 0; j < imstruct.width; ++j) {
```

```
                imtuple[i][j][0] = (sample)
                    round( 255.0 * ( (w[k + n*nodes[1]]-minw)/(maxw-minw) ) );
                ++n;
            }
        }
        // write the image
        pnm_writepam(&imstruct, imtuple);
    }
    // clean up
    pnm_freepamarray(imtuple, &imstruct);
    fclose(buf);

    // free memory
    for(i = 0; i < imcount; ++i) {
        free(paths[i]);
        free(imdata[i]);
        free(imnormal[i]);
        free(target[i]);
    }
    free(paths);
    free(perm);

    free(testerror_path);
    free(trainerror_path);
    free(weight_img_path);

    for(i = 0; i < number_of_classes; ++i) {
        free(classes[i]);
    }
    free(classes);

    free(imdata);
    free(imnormal);
    free(target);
    free(maxvals);
    free(nodes);

    free(training_input);
    free(training_target);
    free(training_error);

    free(testing_input);
    free(testing_target);
    free(testing_output);
    free(testing_error);

    vbp_free_net(net);

    return 0;
}
```

Heimildir

[1] http://kdd.ics.uci.edu/databases/faces/faces.html

[2] http://netlib.org/blas

[3] http://netpbm.sf.net