The two questions were worth 3 points each.

On many of your homeworks, I commented that for problem 2, ‘reconstruct an image from the set of learned features’ meant to run the LCA on a sample image patch rather than reconstruct the whole image. It seems the question was unclear as many of you did some form of full image reconstruction. So please ignore my remarks and ignore the image reconstructions in the below solution sets.

Attached are excellent problem set solutions from two of your classmates, Alex Huth and William Coulter, both Caltech-ers.
Problem 1.

(a) Instead of using Hebb or Oja or Sanger, I just ran PCA on the lines. Here’s the code and the result:

```matlab
p=0.2;
N=64;
batch_size=100;
X=gen_lines(N,batch_size,p);
[c,s,l] = princomp(X');
```

Output:

![Graph showing percentage of variance explained vs. principal component number](image-url)
Interestingly, it appears that the first 16 principal components contain much more information than the rest, which is as it should be. Otherwise, total failure. Included (above) are the first four principal components.

I used the same WTA code as for the last lab, but with the input as above. Here’s the result (a bit better than PCA, but not perfect):
(b) Pertinent code:

```plaintext
for t=1:num_trials
    % generate data for this batch
    X=gen_lines(N,batch_size,p);
    % compute outputs
    Y=sparsify_f(X,Q,W,theta);
    % compute statistics for this batch
    muy=mean(Y,2);
    Cyy=Y*Y'/batch_size;
    Cyx=Y*X'/batch_size;
    % update lateral weights
    dW= -alpha * (Y*Y' - p^2); % dW rule here
    W=W+dW;
    W=W-diag(diag(W)); % zero out diagonal
    W(find(W>0))=0;  % rectify weights
    % update feedforward weights
    dQ= beta * sum(shiftdim(repmat(Y',[1 1 N]),1).* (shiftdim(repmat(X,[1 1 M]),2) - repmat(Q,[1 1 batch_size]))),3);
```
% dQ rule here
Q=Q+dQ;
% update thresholds
dtheta= sum(gamma * (Y-p),2); % dtheta rule here (theta =
t in Foldiak’s paper)
theta=theta+dtheta;
% compute moving averages of muy and Cyy
Y_ave=(1-eta_ave)*Y_ave + eta_ave*muy;
Cyy_ave=(1-eta_ave)*Cyy_ave + eta_ave*Cyy;
% display network state and activity statistics
show_network(theta,Y_ave,W,Cyy_ave,Q);
end

Output with 16 units:

Output with 8 units:
It does not converge nicely with 8 units, with some units describing one line, some a few lines, and some none at all.

**Problem 2.**

Higher $\lambda$ values yield sparser basis sets, with more curves and blobs that could be composed of a few simpler functions. Here are 64 basis functions learned with $\lambda = 1.5$. Some basis functions were not learned at all, presumably because they were never made active.
Here are 64 basis functions learned with $\lambda = 0.1$: 
Here are 256:
Here are 128:
It appears that the overcomplete sets have more plaid-like structures than the regular (64x64) sets.

Here are 32:
Here are 16:
Here are 9:
And here are 2:
The undercomplete sets, however, seem to contain only the most important blob-like Gabor filters.
Problem 1

Part a)

A linear hebbian network cannot be entrained with this network because they pick up various checkered patterns (these explain the greatest variance), instead of lines. A WTA network will be phenomenally unconstrained on this data and move towards a single data point, again resulting in a checkerboard rather than lines. Plots are shown in figure 1 (a better job might have been done with the weights, but it just doesn’t work anyway).

Part b)

$\beta$ weight learning rates should be slowest because the lateral weights $\alpha$ and the threshold $\gamma$, need “time” to stabilize. Having a moving input would make this unstable. As in the paper, the first 100 iterations were run training only the threshold. After that, the following weights were used: $\alpha = 0.2$, $\beta = 0.05$, and $\gamma = 0.02$. See figure 2 for the weight progression and figure 3. We are able to pick out the 16 lines perfectly as basis functions.

Part c)

Here, I dried 8 and 28 units, respectively. The 8 unit system finds some (but not all) of the single lines and also has some crossed patterns (see figure 4). The 28-unit system finds all the same 16 lines that were originally found, and additional combinations / crosses (see figure 5). It seems that overcomplete systems recover the sparse causes, but then generate mostly non-sensical other basis functions – the same effect as for 28 units was seen for larger numbers (not shown). Being undercomplete is crippling in terms of finding the underlying sources.
Figure 1: 1.a: Oja’s rule hebbian learning doesn’t work :-(
Figure 2: 1.b: Trial by trial weight values. Learning is fast.
Figure 3: 1.b: Final output of Bruno’s display_network function. Autocorrelation is along diagonal, as desired.
Figure 4: 1.c: Trial by trial weight values for 8-unit network.
Figure 5: 1.c: Trial by trial weight values for 28-unit network.
Problem 2

Part a)

The $\eta$ parameter was decreased exponentially (simulated annealing-style) according to $\eta = \eta_0 e^{-\beta t}$ starting with $\eta_0 = 0.5$ and using a characteristic time constant of $\beta = 0.01$. 1000 trials were used. One set of basis functions are shown in figure 6. Reconstructions are shown for $\lambda = 0.1$ (figure 7) and for $\lambda = 1$ (figure 8). The higher $\lambda$ is, the worse the reconstruction is, but the more sparse the reconstruction tends to be for each image patch investigated.

Part b)

Using a larger network of 128 units yields far more basis functions (and also takes longer to run :-)). One set of basis functions are shown in figure 9. As before, reconstructions are shown for $\lambda = 0.1$ (figure 10) and for $\lambda = 1$ (figure 11). Again, the higher $\lambda$ is, the worse the reconstruction is, but the more sparse the reconstruction tends to be for each image patch investigated. In contrast with 64 basis functions, these images exhibit two characteristics:

1. They tend to use slightly more basis functions per patch (although the distribution is smoother – probably because there are more to choose from).

2. They tend to have sharper edge expression and features pop out more clearly. This is easiest to see by looking at individual patches (not shown).
Figure 6: 2.a: One set of basis functions (the ones that were used for reconstruction).
Figure 7: 2.a: Reconstruction for $\lambda = 0.1$. The reconstruction is quite good, but the sparsity is not.
Figure 8: 2.a: Reconstruction for $\lambda = 1$. The reconstruction is worse, but the coefficients are much more sparse.
Figure 9: 2.b: One set of basis functions (the ones that were used for reconstruction).
Figure 10: 2.b: Reconstruction for $\lambda = 0.1$. The reconstruction is quite good, but the sparsity is not.
Figure 11: 2.b: Reconstruction for $\lambda = 1$. The reconstruction is worse, but the coefficients are much more sparse.
Matlab Code

Problem 3:

hebb.m

% hebb.m - script to do hebbian learning
%
% tic
if ~exist('D1','var')
end

batch_size =100;
num_trials = 1000;

% number of inputs
N=64;

% number of outputs
M=16;

% target output firing rate
p=0.2;

% initialize weights (start with same ICs)
state = floor(rand * 10000);

% Go through both data sets
for data_idx = 1:1
X = gen_lines(N, batch_size, p);

% Take the mean out, since we have no bias term.
X = X - repmat(mean(X,2),1,size(X,2));
[N K]=size(X);

% Do all three calculations at once in each iteration. Use the same random
% ICs each time.
randn('state',state)
w=randn(N,M);
w_hebb = w(1:N,1:M);  % Unconstrained Hebbian
w_oja = w(1:N,1:M);  % Constrained Hebbian
w_sanger = w;  % Sanger's rule.

eta=1/(K*N);  % Normalize by the number of data points and the dimensions.

for t=1:num_trials

% compute neuron output for all data (can be done as one line)
y_hebb = w_hebb' * X;  %see HKP p199
y_oja = w_oja’ * X;
y_sanger = w_sanger’ * X; %this yields two neurons.

% compute dw: Hebbian learning rule
% Order is changed from lecture to make matrix multiplication work.
dw_hebb = eta * X * y_hebb’;
dw_oja = eta * ( X - w_oja * y_oja ) * y_oja’;

% This one’s more complicated.
dw_sanger = zeros(size(w_sanger));
% Things are kind of backwards — i is the column and j is the row.
% That is to say, the convention with sanger’s rule is that j’s are the
% inputs and i’s are the outputs.
for i = 1:size(w_sanger,2)
    for j = 1:size(w_sanger,1)
        % In order for the first column of this to be equivalent to
        % w_oja, we need to take the transpose of what we’d expect (i.e.
        % in the case of one output (j=1), we want a column vector).
        dw_sanger(j,i) = eta * ((X(j,:) ... 
            - w_sanger(j,1:i) * y_sanger(1:i,:) ) * y_sanger(i,:)’); 
    end
end

% update weight vector by dw
w_hebb = w_hebb + dw_hebb;
w_oja = w_oja + dw_oja;
w_sanger = w_sanger + dw_sanger;

if mod(t,floor(num_trials/10)) ==0
    % Update the plots.
    showrfs(w_oja’)
drawnow
    pause(0.1)
    % toc
end
end
showrfs(w_oja’)

end % data_idx

printfullpagepdf(‘vs298ps4p1a.pdf’)

foldiak.m

% foldiak.m — simulates Foldiak’s sparse coding circuit

%Set rand to its default initial state:
%rand(’twister’, 5489);

batch_size =100;
num_trials = 1000;

% number of inputs
N=64;

% number of outputs
M=16;

% target output firing rate
p=0.2;

% Initialize network parameters (comment these lines out if you want % to restart the script from where you left off in a previous run) %

% feedforward weights
Q=rand(M,N);
Q=diag(1./sqrt(sum(Q.*Q,2)))*Q;

% horizontal connections
W=zeros(M);

% thresholds
theta=ones(M,1);

% learning rates
alpha=0.2;
beta=0.05;
gamma=0.02;

% rate parameter for computing moving averages
eta_ave=0.1;

Y_ave=p;
Cyy_ave=p^2;

for t=1:num_trials

% generate data for this batch
X=gen_lines(N,batch_size,p);

% compute outputs
Y=sparsify_f(X,Q,W,theta);

% compute statistics for this batch
muy=mean(Y,2);
Cyy=Y*Y'/batch_size;
Cyx=Y*X'/batch_size;

if t<100

%just update the threshold
dtheta=.1*(muy-p);% dtheta rule here (theta = t in Foldiak's paper)
theta=theta+dtheta;
end
else

% update lateral weights
dW = -alpha*(Cyy-p.^2); % dW rule here
W = W + dW;
W = W - diag(diag(W)); % zero out diagonal
W(W>0)=0; % rectify weights

% update feedforward weights
dQ = beta*(Cyx-repmat(muy,1,64).*Q);
if 0
for dQ_idx = 1:size(Q,1)
    dQ(dQ_idx,:) = dQ(dQ_idx,:).*
                -beta*Y(dQ_idx,:)*repmat(Q(dQ_idx,:),size(X,2),1)./batch_size;
end
end
Q = Q + dQ;

% update thresholds
dtheta = gamma*(muy-p); % dtheta rule here (theta = t in Foldiak's paper)
theta = theta + dtheta;
end

% compute moving averages of muy and Cyy
Y_ave = (1-eta_ave)*Y_ave + eta_ave*muy;
Cyy_ave = (1-eta_ave)*Cyy_ave + eta_ave*Cyy;

% display network state and activity statistics
if t == 1 || mod(t,num_trials/5) == 0;
    show_network(theta,Y_ave,W,Cyy_ave,Q);
    figure(2);
    if t == 1
        clf
        subplot(3,2,1)
    else
        subplot(3,2,t/num_trials*5+1)
    end
    showrfs(Q)
    title(sprintf('Q at trial %1.0f',t))
end

end

printfullpagepdf('vs298ps4p1_endscreen.pdf',1)
printfullpagepdf('vs298ps4p1_bytrial.pdf',2)

Problem 4:

sparsenet.m

% sparsenet.m - Olshausen & Field sparse coding algorithm
% Before running you must first load the training data array IMAGES

num_trials = 1000;
batch_size = 100;

if ~exist('IMAGES', 'var')
    load IMAGES
end

[imsize imsize num_images] = size(IMAGES);
BUFF = 4;

% number of outputs
M = 128;

% number of inputs
N = 64;
sz = sqrt(N);

% initialize basis functions (comment out these lines if you wish to
% pick up where you left off)
Phi = randn(N, M);
Phi = Phi * diag(1 / sqrt(sum(Phi * Phi)));

% learning rate (start out large, then lower as solution converges)
% Start with this eta
eta_0 = .5;
beta = 0.01;

% lambda
lambda = 0.1;

a_var = ones(M, 1);
var_eta = .1;

I = zeros(N, batch_size);
display_every = 50;
display_network(Phi, a_var);

for t = 1:num_trials

    % choose an image for this batch
    eta = eta_0 * exp(-beta * t);
    imi = ceil(num_images * rand);

    % extract subimages at random from this image to make data array I
    for i = 1:batch_size
        r = BUFF + ceil((imsize - sz - 2*BUFF) * rand);
    end
end
c = BUFF + ceil((imsize - sz - 2*BUFF) * rand);
    I(:, i) = reshape(IMAGES(r:r+sz-1, c:c+sz-1, imi), N, 1);
end

% calculate coefficients for these data via LCA
ahat = sparsify(I, Phi, lambda);

% calculate residual error
R = I - Phi * ahat;

% update bases
    dPhi = eta * R * ahat';  % learning rule here
    Phi = Phi + dPhi;
    Phi = Phi * diag(1./sqrt(sum(Phi.*Phi)));  % normalize bases

% accumulate activity statistics
a_var = (1 - var_eta) * a_var + var_eta * mean(ahat.^2, 2);

% display
    if (mod(t, display_every) == 0)
        display_network(Phi, a_var);
        [eta t]
    end

end

printfullpagepdf(sprintf('vs298ps4p2_basis_M=%3.0f.pdf', M))

process_image.m

% Make clear that this must me a multiple of 8
options.demosize = 8*64;

% patch_width = 8;
    step = patch_width / 1;
    image_num = 1;
    lambda = .1;
% This is from the sparsenet output
    basis = Phi;

% load sim_0.mat.mat basis neIMAGES
    img = IMAGES(1:options.demosize, 1:options.demosize, image_num);
    [ys, xs] = size(img);
img_noise = img + noise * randn(size(img));

% TODO
Io = zeros(ys, xs);
Ir = zeros(ys, xs);
Ic = zeros(ys, xs);

count = 0;
for x = 1:step:(xs − patch_width +1)
    for y = 1:step:(ys − patch_width +1)
        count = count + 1;
    end
end

% The size of this is the number of patches that will be needed
used = zeros(count, 1);

stopped_at_maxiter = [];

count = 0;
for x = 1:step:(xs − patch_width +1)
    for y = 1:step:(ys − patch_width +1)
        % This is the patch-sized image segment we are fitting. And it's 1D now.
        X = reshape(img_noise(x:(x+patch_width−1),y:(y+patch_width−1)), ... 
                   patch_width*patch_width, 1);

        patch = X;

        coefficients = sparsify(patch, Phi, lambda);

        % How many patches have we filled so far?
        count = count + 1;

        % How many coefficients of the basis functions are non-zero?
        used(count) = length(find(coefficients));

        % Reconstructed patch.
        Xr = basis * coefficients;

        % Reconstructed image.
        Ir(x:(x+patch_width−1),y:(y+patch_width−1)) = Ir(x:(x+patch_width−1), ... 
                                                   y:(y+patch_width−1)) ... 
                                                   + reshape(Xr, patch_width , patch_width);

        % Update the count of how many times a pixel has been included in a patch.
        Ic(x:(x+patch_width−1),y:(y+patch_width−1)) = Ic(x:(x+patch_width−1), ... 
                                                   y:(y+patch_width−1)) + 1;
end
end

% Normalize the image.
Ir = Ir ./ Ic;

figure(1); clf

subplot(1,2,1);
colormap gray
imagesc(img, 'EraseMode', 'none', [-1 1]);
axis image off
title 'Original'

subplot(1,2,2);
colormap gray
imagesc(Ir, 'EraseMode', 'none', [-1 1]);
axis image off
title 'Reconstruction'

figure(130);

subplot(2,2,1);
colormap gray
imagesc(img, 'EraseMode', 'none', [-1 1]);
axis image off
title 'original'

subplot(2,2,2);
colormap gray
imagesc(Ir, 'EraseMode', 'none', [-1 1]);
axis image off
title(sprintf('Reconstruction \(\lambda=%2.2f\)', lambda))

subplot(2,2,3);
colormap gray
imagesc(Ir-im, 'EraseMode', 'none', [-1 1]);
axis image off
title('error wrt. original')

subplot(2,2,4);
hist(used);
title('used basis functions per patch')

printfullpagepdf(sprintf('vs298ps4p2b_%2.2f.pdf', lambda))