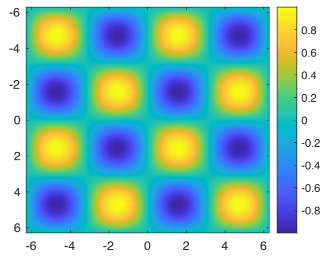
BE-130 Homework #4, Due Tuesday May 7.

1. (a) Hand code a small 3-layer neural network (2 layers of weights) and train it using backprop on a version of the classic XOR problem after randomly initializing the weight values. XOR mapping: [0,0]→0, [0,1]→1, [1,0]→1, [1,1]→0. Display its output after training (as an image) over a range of input values (from 0 to 1 continuously) that interpolates the trained values, and comment on the result. Also display its output over a wider range (-1 to 2, for both inputs), and comment on what the extrapolation looks like. **(b)** Compare the learning curve when there are 0 vs 2 vs 4 vs 8 hidden (layer 2) units. Comment on your findings.



2. Hand code a small 3 layer network to do function approximation for the function shown in the figure on the right. This is a continuous-space version of an XOR-like parity function generated by the product of two sine waves, like this:
`x1=-2*pi:0.1:2*pi; N=length(x1); for k1=1:N, for k2=1:N, f(k1,k2)=sin(x1(k1))*sin(x1(k2)); end; end`
OR like this: `x1=-2*pi:0.1:2*pi; [x,y]=meshgrid(x1); f=sin(x).*sin(y);`
And then plotted like this: `figure; imagesc(x1,x1,f); colorbar;`
Approximately how many hidden-layer units are required to get a decent convergence rate? Why might this be?

3. Analyze the relationship between amount of training data available and the performance level for a multilayer network image classifier. We can do this by making a few simple edits to MATLAB's deep network classifier tutorial: TrainABasicConvolutionalNeuralNetworkForClassificationExample.mlx First run this example and examine the output plot. Then click "save" → "save as" to save it as a regular m-file with a different name like myclass1.m to make it easier to manipulate (most of this new m-file will be comment lines). Instead of statically setting the training set size to 750 examples for each digit, encapsulate most of the code in a for loop and vary the training set size over a wide range NTrain_lst=[10, 20, 50, 100, 200, 500, 750] and then plot the relationship between training set size and cross-validated classification accuracy. To obtain the latter you can (assuming "k" is you loop variable) either change line 193 to:
`FinalAccuracy(k) = sum(YPred == YValidation)/numel(YValidation)`
Or instead, change line 183 to:
`[net, info1] = trainNetwork(imdsTrain,layers,options); FinalAccuracy(k)= info1.ValidationAccuracy(end)/100;`
And then once outside of the loop:
`figure; semilogy(NTrain_list, 1-FinalAccuracy); title ('Error rate a function to training set size')`

4. For the simple, single-state model of error-dependent learning discussed in class:

$$e(n) = x_{IDEAL}(n) - x(n)$$
$$x(n+1) = Ax(n) + Be(n)$$

(a) Assuming $x_{IDEAL}(n)$ is a constant (i.e. $x_{IDEAL}(n)=C$), derive the asymptotic learning level $x(\infty)=(B/[1-A+B])*C$, and the time constant for learning in terms of the model parameters, A, B, C.

Recall that e=error, x = adaptive state, i.e. adaptation/learning level, A = retention coefficient, B = learning rate, x_{IDEAL} = the ideal state to compensate the disturbance, & n = trial #. Recall also that the time constant of a decaying exponential function is the inverse of its rate constant k. And for a pure exponential function: $f(t) = K_1e^{-kt}$, the rate constant k is negative the ratio of the function's slope to its value: $-f'(t)/f(t)$. For an exponential function with an offset: $f(t) = K_1e^{-kt} + K_0$, k would be $-f'(t)/[f(t)-K_0]$. In the discrete case, $f'(t)$ would be, of course, be equivalent to $f(n+1)-f(n)$.
Thus, $k = -[f(n+1)-f(n)]/[f(n)-K_0]$ and the time constant would be $-[f(n)- K_0]/[f(n+1)-f(n)] = [K_0-f(n)]/[f(n+1)-f(n)]$.
So to find the time constant, first subtract the current state from the asymptotic value, and then take the ratio of that quantity to the change in state.

(b) A is usually close to but less than 1, and B is usually small and positive. Simulate how the adaptive state x(n) evolves for several different (A,B) value pairs. Plot these simulations on the same axis choosing a simulation duration that allows the asymptote to be reached. Include a legend that identifies which simulation is which.
You can use the following pairs for (A,B): { (0.9, 0.1), (0.9, 0.2), (0.9, 0.3), (0.8, 0.1), (0.95, 0.1), (0.99, 0.1) }
Comment briefly on the results, in particular on how the time constants and asymptotic errors are affected by the retention factor A and the learning rate B.

In MATLAB, you can create these simulations using a simple for loop over trial # (n), like this:

```
figure;
A=0.9; B=0.1; % assign the learning parameters
x(1)=0;           % assign the initial state to zero
x_ideal = [zeros(1,10), ones(1,50)]; N=length(xi) ; % the perturbation vector: x_ideal = the ideal state x on each trial
for n=1:N,
  e(n) = x_ideal(n) - x(n);      % compute the error for the current state, e(n)
  x(n+1) = A*x(n) + B*e(n);  % compute the next state, x(n+1)
end
x = x(1:N); % this isn't absolutely necessary, but here we trim x so it's the same length as xi and e
plot(x); grid on;
```

5. Consider the parallel two-state (2nd order) version of this trial-to-trial learning rule (Smith et al 2006).
$e(n) = x_{IDEAL}(n) - (x_F(n) + x_S(n))$
$x_F(n+1)=A_F x_F(n)+B_F e(n)$
$x_S(n+1)=A_S x_S(n)+B_S e(n)$

(a) Derive the asymptotic learning level, assuming $x_{IDEAL}(n)$ is a constant C.

(b) Simulate learning for this two-state model for 120 trials with A_F=0.6, A_S=0.99, B_F=0.2, B_S=0.02. Plot $x_F(n)$, $x_S(n)$, $x_{TOT}(n)$, and e(n) where $x_{TOT}(n) = x_F(n) + x_S(n)$ all on the same axis. Include a legend. Comment briefly on the results.

(c) **Bonus: Derive the asymptotic learning level for the parallel Nth order version of this learning rule.