ES 249 Problem Set 4

Andrew T. Sullivan

7 May 2019

1 Question 1: The XOR Problem

In this problem, we implement a simple 3-layer fully connected feedforward neural network and train it to compute the XOR problem, i.e. to return 0 if the two inputs are the same (1 and 1 or 0 and 0) and to return 1 if they are different (1 and 0 or 0 and 1). The first layer contains two neurons, which recieve the two inputs, and the third (output) layer contains a single neuron, corresponding to the output of the network. For each neuron j, the net input is the weighted sum of all the activations of the neurons in the previous layer, $\Sigma_i w_{ij} y_i$, and the activation of the neuron is given by passing its net input through a nonlinearity, such that $y_j = f(\Sigma_i w_{ij} y_i)$. We initially set the two weight matrices, w_{ij} and w_{jk} to be random values between 0 and 1, and train the network based on the backpropagation algorithm.

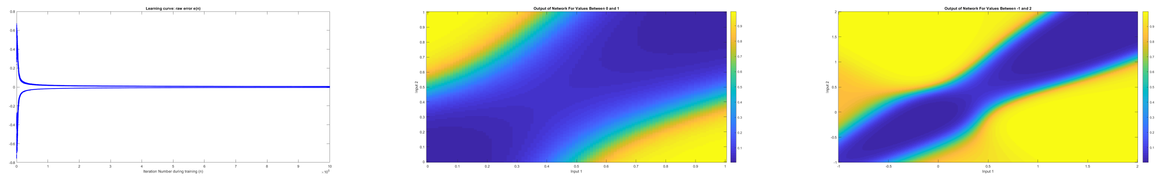
We use the sample code provided and extend it to include proper backprop and simulation of various hidden layer sizes. This code works by first initializing the two weight matrices with random numbers, accounting for an extra bias weight feeding in from the input and hidden layers. Thus, the first weight matrix is $3xN_{Hidden}$ and the second is $N_{Hidden} + 1x1$. We initially set the hidden layer to have 8 units, and further explore the dependence on this size later. The inputs and outputs are specified to match the XOR function, as defined above. Here, we use a nonlinearity of the form $f(x) = \frac{e^x}{1+e^x}$, such that the derivative (required for backprop) is given by $f(x)(1 - f(x))$. We initialize our learning rate as 0.1 and perform 10^6 learning steps. Finally, we initialize a vector which is used to store the error as a function of iteration number.

The backpropagation algorithm (backprop) is an extension of gradient descent applied to fully connected neural networks. The whole derivation is not reproduced here, but in brief, the error is computed as the difference between the output of the network and the target value. The output value is found by propagating the input through the network with its current weights. For the purposes of our three-layer network with bias units of 1 in the first and second layers, we first choose a random integer between 1 and 4 to choose the input. The input for the XOR function is augmented with the bias value. The output of the hidden layer is given by multiplying the transpose of the first weight matrix with the augmented input and passing these values through the nonlinearity. This output is then augmented to account for the bias unit in the hidden layer, and the process is repeated using the second weight matrix. The output of this last calculation is the output of the network. From this error, the gradient is computed for each neuron in the preceding layer as the product of the error, the activation of the neuron being updated, and the derivative of the nonlinearity evaluated for the output neuron, i.e. $\nabla_2 = (X_3 - X*) f'(W_2' * X_{2a}) X_{2a}$. As we are updating all the weights in the hidden layer, including that for the bias unit, the X_2 vectors here are the augmented versions.

In order to find the gradients for the first weight matrix, we must backpropagate the error to the previous layer. This backpropagation is accomplished by replacing the terminal X_2 with the weight matrix for the hidden layer (neglecting the bias weight since it cannot be backpropagated as it does not receive input from the input layer). We then perform element-wise multiplication with the derivative of the nonlinearity, now evaluated using the product of the first weight matrix and the augmented input. This element-wise multiplication yields an $8x1$ vector, which we then use to take the outer product with the activations of the input layer $(1x3)$ to obtain the required dimensionality for the updates, i.e. $\nabla_1 = (X_3 - X*) f'(W_2' * X_{2a}) W_{2,1:N\_Hidden} .* f'(W_1' * X_{1a}) .* X_{1a}$. As with the first layer, the gradient is the product of an error, the derivative of the nonlinearity evaluated using the summed inputs to each neuron in the layer, and the inputs of the previous layer, but in this case, our error is backpropagated one layer. With our gradients, we update the weight matrices by subtracting the gradient scaled by the learning rate.

1.1 a)

We train our network using the above scheme and evaluate the output from 0 to 1 in 0.01 steps and from -1 to 2 using the same steps. We also plot the trajectory of learning as the error as a function of iteration. We can see that using 8 hidden layer units is sufficient to allow for rapid learning within approximately 200,000 steps, and that the network properly learns to return the desired values at the four specified coordinates (the corners in Figure 1b) with a central saddle point. Extending the tested range shows that only a small region returns values below 0.5 and most of the function is dominated by larger outputs. However, it seems likely that this is a consequence of the random weight initialization and that the opposite case could arise in which the blue region was larger than the yellow region depending on the conditions of learning.



(a) Error vs. Learning Increment. (b) Values Between 0 and 1. (c) Values Between -1 and 2.

Figure 1: Learning Trajectory and Network Output for 3-Layer Neural Network with 8 Hidden Units for XOR Problem.

1.2 b)

We next explore the dependence of learning on the size of the hidden layer by simulating the same network with the same inputs but varying the size of the hidden layer from the values 0, 1, 2, 4, and 8. Our results highlight

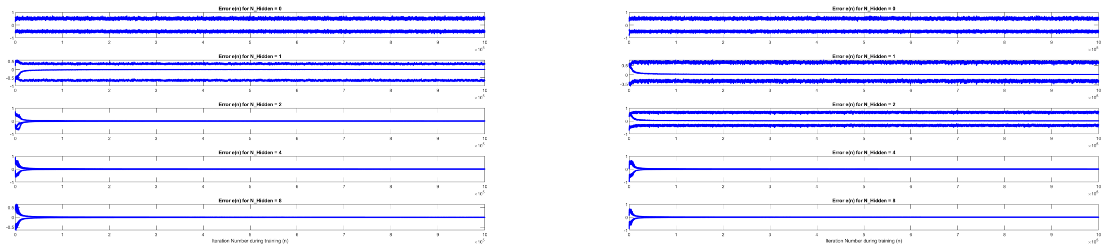


Figure 2: Learning Trajectory for Two Different Trials with Varying Hidden Layer Sizes.

the important dependence on hidden layer size in function approximation. Results from two trials with the same parameters are displayed in Figure 2. In both cases, networks without hidden layers or with only 1 hidden layer neuron (the top two plots) do not converge within 10^6 steps, and it seems likely that the network will never be able to learn the proper outputs (this has been confirmed for the perceptron case, which has no hidden layers and is unable to solve the XOR problem) regardless of the number of iterations. For hidden layers with four or eight neurons, the network learns proper function approximation rapidly for this simple case. For intermediate sizes (i.e. 2 neurons), the network performance varies, likely depending on the weight initialization and randomness of sample presentation. For the first trial, the network converges rapidly as in the larger network cases. For the second trial however, the network is unable to learn over the entire learning duration, similar to the smaller networks. Thus, it is important to properly initialize the network and validate over multiple trials to ensure proper learning and convergence.

2 Question 2: Continuous XOR-Like Parity Function

Here, we employ the same 3-layer network architecture discussed above to perform function approximation for the function $f(x, y) = sin(x) * sin(y)$ over the range $[-2\pi, 2\pi]$ for both variables. This function is plotted below for reference. As mentioned above, we begin by attempting to train the 3-layer network defined above, probing the influence of various conditions (hidden layer size, learning rate, resolution of the training set, and number of training iterations) on the performance of the network. The first presented results were an initial scan over a large range of hidden layer sizes, from 10 to 100 neurons. All simulations were trained for at least 10^6 iterations. Larger and smaller training lengths were tested (from 10^5 to 10^8) and this size seemed sufficient to reach a steady-state asymptotic learning level in all cases. From here, we can see that no network does a perfect job of completely capturing all the "bumps" in the function, even for the largest sizes tested (networks with 1000 and 2000 cells were also trained, but results were similarly unsatisfactory). The influence of learning rate (0.05 vs. 0.1) was also tested, with a network with 30 hidden neurons showing the closest performance for a smaller learning rate. Given that using a range of -2π to 2π with an increment of 0.1 in both input variables leads to a very large input space for the system to explore (15,876 points), the influence of increasing the increment size was also examined. In general, using an increment of $\frac{\pi}{4}$ was still sufficient to preserve the learning properties of the system, but any larger sizes could no longer capture the proper periodicity. Further reducing the spacing did not improve learning.

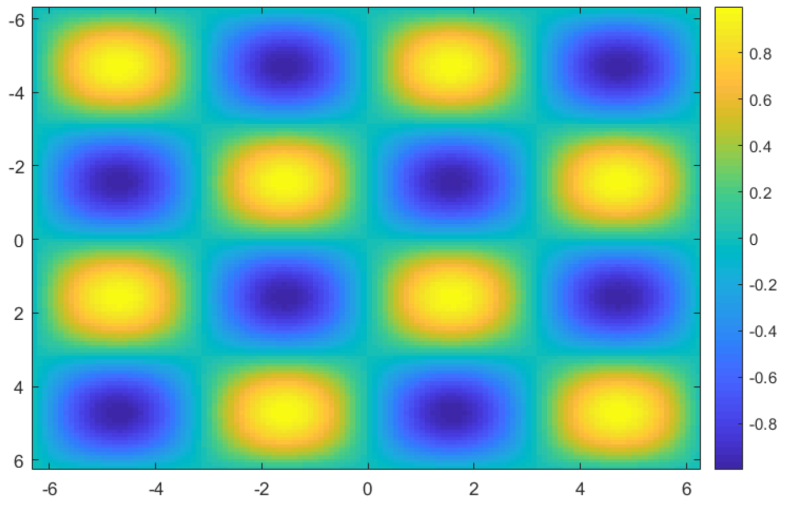
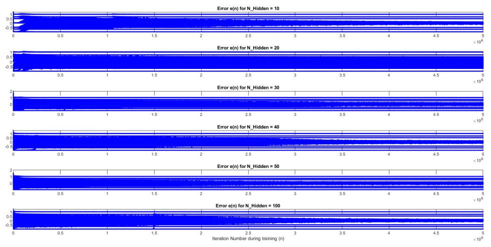
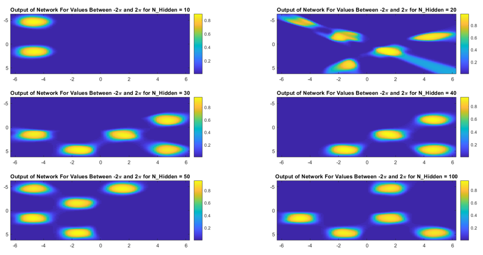


Figure 3: Target Function for Approximation: Product of Two Sines.

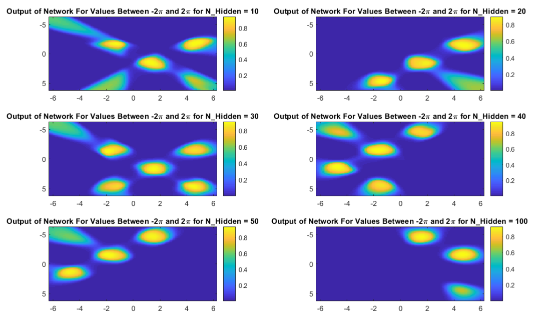


(a) Learning Trajectory.

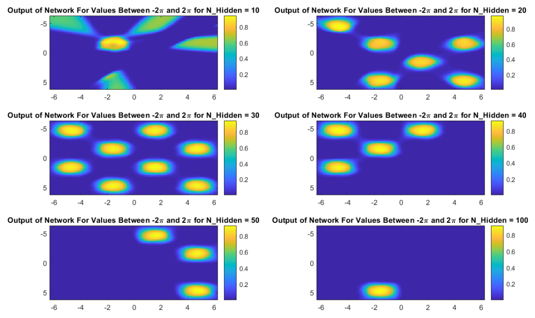


(b) Accuracy.

Figure 4: Training of 3-Layer NN for Various Hidden Layer Sizes.



(a) $\alpha = 0.1$.



(b) $\alpha = 0.05$.

Figure 5: Training of 3-Layer NN for Two Learning Rates.

Clearly, no combination of parameters was successful in properly capturing the function of interest, suggesting that there was a different intrinsic problem with the network. One observation which revealed what this issue was is that, while negative errors often decreased in magnitude toward zero, the same did not occur for positive errors in most cases of parameter combinations. Similarly, though some of the positive bumps are properly captured, none of the negative bumps are present in any of the tested conditions. Since the range of the function of interest is from -1

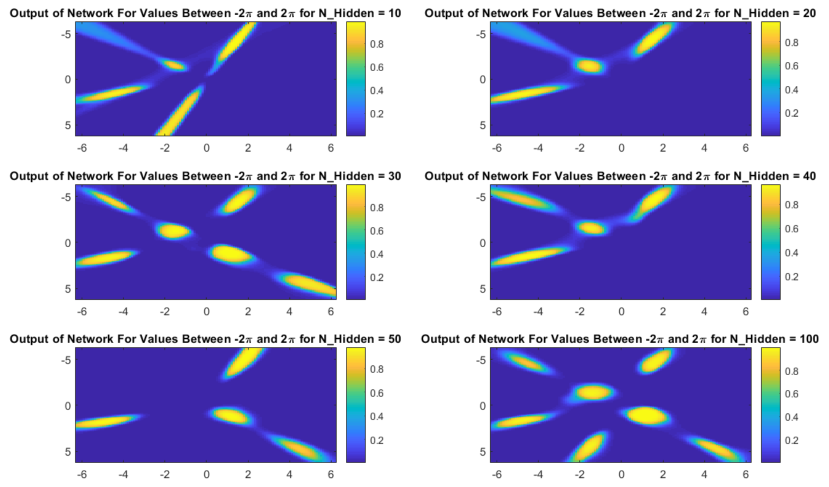
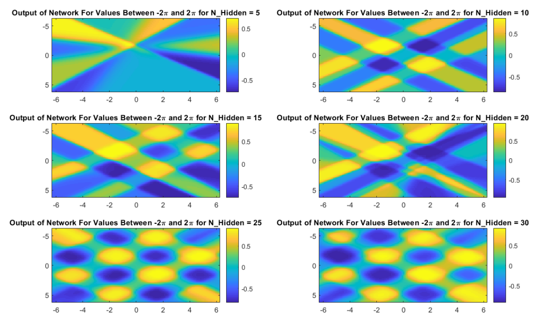
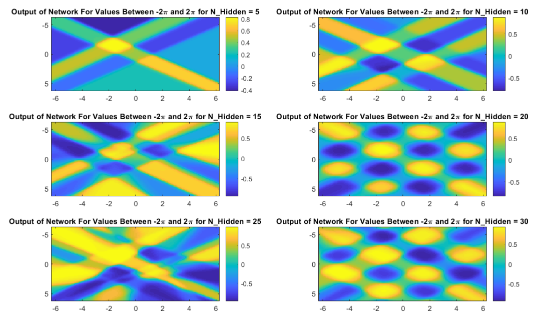


Figure 6: Larger Increment of $\pi/2$ to Reduce the Size of the Training Data.



(a) $\alpha = 0.05$.



(b) $\alpha = 0.1$.

Figure 7: Training of 3-Layer NN Using Tanh Nonlinearity.

to +1, this function is incompatible with the network described above, as the range of the nonlinearity being used (i.e. the exponential/Boltzmann function, and thus the output of the final neuron) is from 0 to 1, preventing any negative output. Therefore, the $tanh(x)$ nonlinearity was employed instead, another common nonlinearity but one which allows for negative outputs. Its derivative is $1 - tanh^2(x)$. Using this new nonlinearity, networks at varying size were once again trained, and though convergence was imperfect, networks with 20 and 25 hidden units did reasonably good jobs of capturing the behavior of the target funtion for increments of 0.1 and 0.05 in the training data, respectively. Given the larger dimensionality of the training data (15,000 points with a periodic structure vs. 4 points), it is rather unsurprising that this network took approximately 6 times more hidden layer neurons to qualitatively reproduce the desired output than that used for the XOR function. Interestingly, further increasing the learning iterations or the network size did not produce a resultant increase in performance. For the former parameter, it seems that the network has either found a local minimum and is stuck, which happens for non-convex optimization, or the asymptote is a consequence of a vanishing gradient as the minimum is approached. In the latter case, larger numbers of neurons in the hidden layer substantially increase the dimensionality of both weight matrices and the optimization problem as whole, further increasing the likelihood that a local minimum will cause the system to get stuck or requiring very large increases in the number of iterations required for proper training.

3 Question 3: Image Classifier Network

In this question, we consider the training of a convolutional neural network (CNN) with pre-specified architecture as in the Matlab live example TrainABasicConvolutionalNeuralNetworkForClassificationExample.mlx. Convolutional neural networks with similar architectures are often used in computer vision applications for image classification. The details are further specified as comments in the code, but briefly, the script extracts 10,000 images from the DigitDataset datastore corresponding to 1000 examples each of digits 0-9. Datasets like this and MNIST are commonly used for image classification tests. 20 examples are plotted for reference.

After separating the images in each class and specifying the class labels, a CNN is initialized with a typical archi-

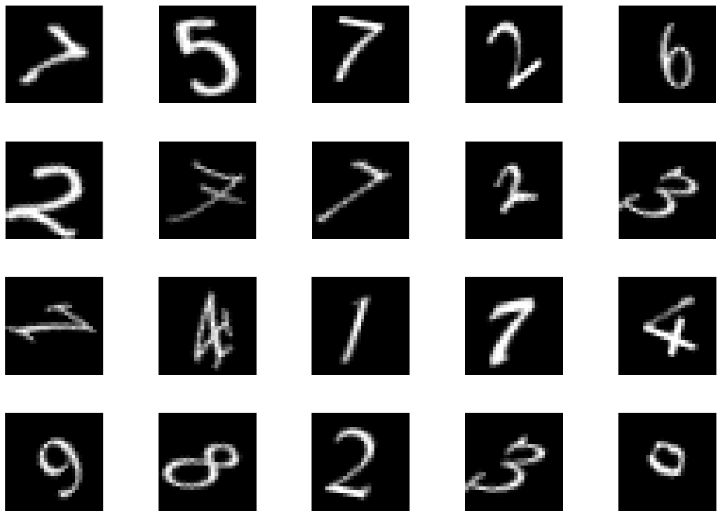
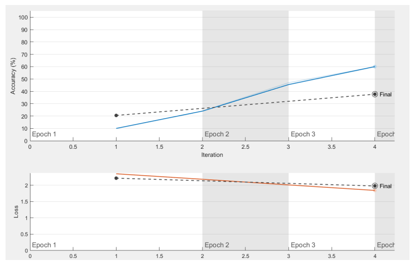
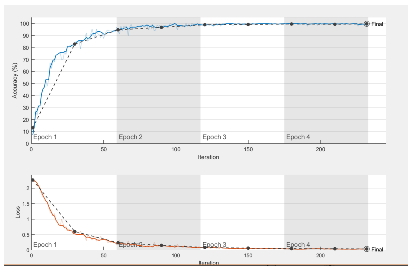


Figure 8: Examples from the Digits Dataset Used for Training and Validation.

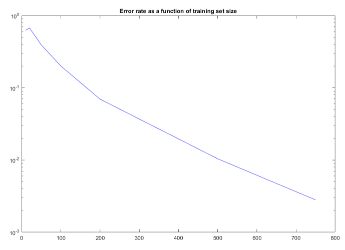
tecture. An input layer feeds in the value of each pixel in the image to the first convolutional layer, which scans over the image using 3x3 convolutional filters padded to form an output with the same size as the input. Output values are normalized and passed through a rectified linear unit (ReLU) nonlinearity, which sets all negative values to zero. A max pooling layer reduces the size by taking the maximum in 2x2 sections of the image. This process is repeated two more times with the final max pooling layer replaced by a fully connected layer, in which all input neurons project to all of 10 ouput neurons. A softmax nonlinearity is applied, which represents the probability of the image belonging to each of the 10 digit classes. The classification layer then selects the predicted label for the image. The network architecture is then trained on a randomly selected subset of a specified size from the dataset using stochastic gradient descent with momentum and validation from a sample not included in the training data every 30 steps. 4 epochs are used in training and the overall accuracy upon completion is assessed as the frequency of correct identifications across the whole validation dataset. We slightly modify this function to test the dependence on the size of the training set, testing network performance for training sets with 10, 20, 50, 100, 200, 500, and 750 images per digit. For each, we train the network and test accuracy over the validation dataset. We then plot the overall error (1-accuracy) as a function of training set size.



(a) 10 Training Examples per Digit.



(b) 750 Examples per Digit.



(c) Overall Error vs. Examples.

Figure 9: Learning Trajectory for Two Different Training Set Sizes and Overall Error vs. Training Set Size.

Unsurprisingly, we can see that the networks provided with larger training set sizes perform substantially better than those with smaller sizes, hence the general downward trend in the error as a function of training set size. There are a number of contributions to this trend. Most obviously, providing the network with larger numbers of examples will lead to better generalization and learning. More subtly, however, is that the validation dataset, i.e. the number of samples against which the network is validated, is larger for those with smaller datasets, which may also contribute to decreased overall accuracy. Finally, the number of epochs of training (number of times through which all of the samples are fed into the network and used to train the weights) is the same across training set sizes, but the number of total iterations (the number of training samples times the number of epochs) is lower for the smaller training sets. Therefore, it is possible that, given the same number of total trials rather than epochs, the networks with less available data would at least perform at a level closer to that of the networks with more data. However, it is also possible that this would lead to overfitting and produce the opposite effect.

4 Question 4: The Single-State Learning Model

In this question, we consider a simple single-state model for error-dependent learning, in which the error, $e(n) = x_{IDEAL}(n) - x(n)$ is used to update the state at each step by $x(n+1) = Ax(n) + Be(n)$, in which x_{IDEAL} is the target value for the state x, e is the error, n is the step number, and A and B are constants representing the retention and learning rates, respectively. We can find simple analytical solutions for the asymptotic learning level $x(\infty)$ and the time constant of learning τ if we make the assumption that the ideal learning level is a constant, $x_{IDEAL} = C$.

4.1 a)

Using this assumption, we can write the error as $e(n) = C - x(n)$ and the update expression as $x(n+1) = Ax(n) + B(C - x(n)) = (A - B)x(n) + BC$. If we subtract $x(n)$ from both sides, we can obtain an expression for the incremental change in the state variable, $\Delta x = x(n+1) - x(n) = (A - B - 1)x(n) + BC$. At the asymptotic value, the state does not change, so $\Delta x = 0 = (A - B - 1)x(\infty) + BC$. Rearranging, we obtain an expression for the asymptotic learning level, $x(\infty) = \frac{BC}{1-A+B}$.
The learning as a function of time can be represented by a (discrete) shifted expontential of the form $f(t) = K_1 e^{-kt} + K_0$, in which K_0 is the asymptotic value derived above. The rate constant, k, is given by $k = \frac{-f'(t)}{f(t)-K_0}$ for a shifted expontential, and the time constant, τ, is its reciprocal. For our discrete case, the derivative $f'(t)$ is given by the difference between consective points, $x(n+1) - x(n) = \Delta x = (A-B-1)x(n)+BC = -(1-A+B)x(n)+BC$, and $K_0 - f(t)$ is given by the difference between the asymptotic value and the current state, $\frac{BC}{1-A+B} - x(n)$. Therefore, the time constant $\tau = \frac{1}{k} = \frac{\frac{BC}{1-A+B} - x(n)}{BC-(1-A+B)x(n)}$. Factoring out the coefficient $1 - A + B$ from the denominator, we obtain the expression for the time constant $\tau = \frac{1}{1-A+B} \frac{\frac{BC}{1-A+B} - x(n)}{\frac{BC}{1-A+B} - x(n)} = \frac{1}{1-A+B}$.

4.2 b)

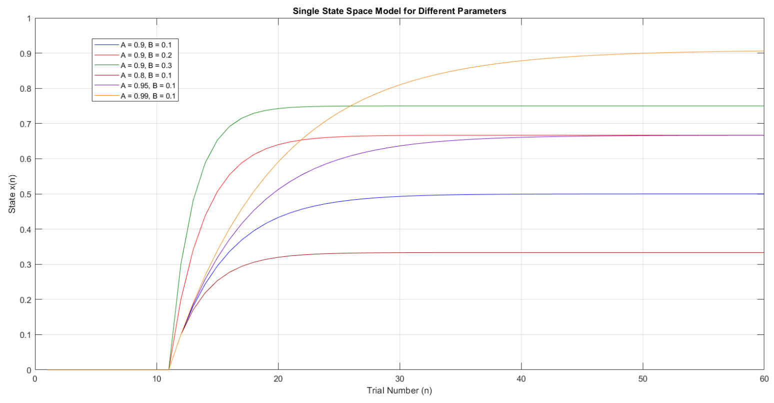


Figure 10: Single State Space Model with Different Retention and Learning Parameters.

We model the learning dynamics of the single state space model for varying values of the parameters for retention and learning, A and B. The specified parameter combinations are denoted in the legend. It is clear that different parameter choices significantly affect both the asymptotic error value and the time constant, as anticipated from the above derivations. If we set the ideal learning value, C, to 1 so that our state reflects relative learning, we can test the asymptotic value predictions with the simulations. For the first curve, we expect the asymptote to be at $B/(1-A+B) = 0.1/(1-0.9+0.1) = 0.5$, consistent with our simulation. Other asymptotic values are also correct. In terms of trends, we observe that, for fixed retention, increasing the learning rate increases the asymptotic value and decreases the time constant, as expected from our derivations. For a fixed learning rate, increasing the retention rate increases the asymptotic value and increases the time constant for learning. Therefore, a balance of learning and retention is required to maximize the asymptotic value and minimize the time constant, and in general, more effective learning appears to require longer times.

5 Question 5: The Parallel Two-State Learning Model

Instead of considering only one state variable in representing learning, it is more common to consider a two-state parallel model, in which learning is governed by the parallel action of fast and slow state variables, x_f and x_s. The total state is given by the sum of these two variables, so the total error is $e(n) = x_{IDEAL} - x(n) = x_{IDEAL} - (x_f(n) + x_s(n))$, and the two state variables are updated by analogous expressions to those introduced above, $x_f(n+1) = A_f x_f(n) + B_f e(n)$ and $x_s(n+1) = A_s x_s(n) + B_s e(n)$. We once again consider the case where the ideal state is constant, $x_{IDEAL} = C$.

5.1 a)

As before, the increment for each of the state variables is given by the expressions $\Delta x_f = (A_f - 1)x_f(n) + B_f e(n)$ and $\Delta x_s = (A_s - 1)x_s(n) + B_s e(n)$. Once again, at the asymptotic value, the increment of each state variable should be zero, so we obtain $(A_f - 1)x_f(\infty) + B_f e(\infty) = 0$ and $(A_s - 1)x_s(\infty) + B_s e(\infty)$. We can solve these for the state variables and insert these expressions into the equation for the asymptotic error, giving $e(\infty) = C - x_f(\infty) - x_s(\infty) = C - \frac{B_f e(\infty)}{1-A_f} - \frac{B_s e(\infty)}{1-A_s}$, or $e(\infty) = \frac{C}{1+\frac{B_f}{1-A_f}+\frac{B_s}{1-A_s}}$. Inserting this back into our expression for each asymptotic state variable, we have $x_f(\infty) = \frac{B_f}{1-A_f}\frac{C}{1+\frac{B_f}{1-A_f}+\frac{B_s}{1-A_s}}$ and $x_s(\infty) = \frac{B_s}{1-A_s}\frac{C}{1+\frac{B_f}{1-A_f}+\frac{B_s}{1-A_s}}$. The total learning level, $x(\infty)$, is the sum of these two terms, which is also equal to the difference between the ideal level, C, and the asymptotic error, $e(\infty)$. Then, $x(\infty) = \frac{C}{1+\frac{B_f}{1-A_f}+\frac{B_s}{1-A_s}}(\frac{B_f}{1-A_f} + \frac{B_s}{1-A_s})$.

5.2 b)

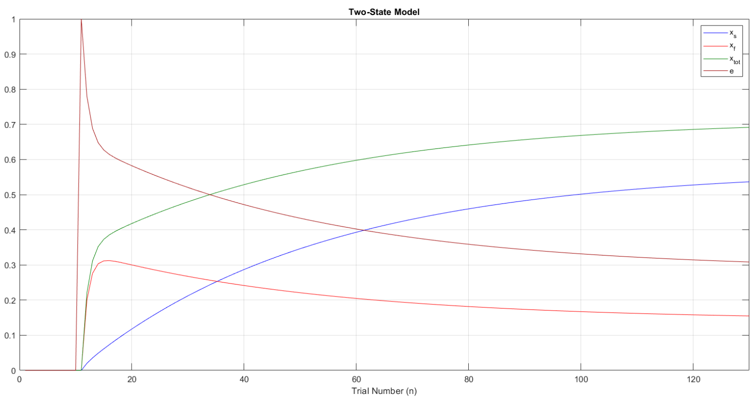


Figure 11: Fast, Slow, and Total Learning States in Parallel Two-State Learning Model.

We simulate the above model using $A_f = 0.6$, $A_s = 0.99$, $B_f = 0.2$, and $B_s = 0.02$, and plot all four state variables (x_f, x_s, x, and e) for 120 trials, beginning after 10 trials with no learning. We can see that our error progressively decreases as the overall learning state increases, as expected since our ideal learning value is constant so these two quantities must sum to 1. We also observe a rapid increase in the overall state variable, reflecting the rapid increase in the fast state variable x_f. However, this variable quickly begins decreasing as the error decreases, and so the slower state variable begins to dominate learning after less than 20 trials. Thus, we can see that the overall learning results from interplay between an initial fast phase and prolonged slow phase. We can also note that the asymptotic value for the overall learning matches the prediction from our derived formula.

5.3 c)

The derivation in part **a)** of this question can be easily extended for an arbitrary number of state variables. For each state variable $x_i(n)$, the update rule is given as $x_i(n+1) = A_i x_i(n) + B_i e(n)$ and the error (assuming constant ideal value of C as before) is $e(n) = C - x(n) = C - \Sigma_i x_i(n)$, where the sum is over all the state variables x_i from 1 to N. An analogous expression can be derived for each asymptotic state variable in terms of the asymptotic error, $x_i(\infty) = \frac{B_i}{1-A_i} e(\infty)$, which can be inserted into the error equation and rearranged as before, yielding $e(\infty) = \frac{C}{1+\Sigma_i \frac{B_i}{1-A_i}}$. Thus, the asymptotic value for each state variable is given by $x_i(\infty) = \frac{B_i}{1-A_i} \frac{C}{1+\Sigma_i \frac{B_i}{1-A_i}}$, and so the asymptotic learning level, which is the sum of all the state variables, is $x(\infty) = \frac{C}{1+\Sigma_i \frac{B_i}{1-A_i}} \Sigma_i \frac{B_i}{1-A_i}$.

6 Appendix

The first code is adapted from the provided script for simulating a 3-layer neural network with 8 hidden units. This version initially simulates a two-layer network with two inputs, a bias, and an output, and then iterates through several hidden unit sizes to observe trends in learning. Up to a specified number of trials, the inputs are randomly selected amongst the four choices for an XOR function, are propagated through the network, and the errors are backpropagated to update the weights via gradient descent. The results in terms of the learning trajectory and the accuracy of the function between 0 and 1 and between -1 and 2 are plotted.

```
        clear all
N_Hidden_list = [1, 2, 4, 8];
figure;

NH0 = 0;
W = randn(3,1);
x1_list = [0 0; 0 1; 1 0; 1 1]';
ideal_output = [0 1 1 0];
nonlinfunc = @(x) exp(x)./(1+exp(x));
dnonlinfunc = @(x) nonlinfunc(x).*(1-nonlinfunc(x));
alpha = 0.1; N=1e6; % alpha = learning rate, N = # of iterations
clear e_;
e_ = zeros(1,N);
for k=1:N
 i = randperm(4,1);
 x1 = x1_list(:,i); x1a = [x1;1]; % x1a = x1-augmented (augmented with a
    constant offest of 1)
 x2 = nonlinfunc(W'*x1a);
 e = x2 - ideal_output(i);
 e_(k) = e;

 gradient = e_(k)*dnonlinfunc(W'*x1a)*x1a';
 W = W - alpha*gradient';
end
subplot(length(N_Hidden_list)+1,1,1);
plot(e_,'.'); title(['Error e(n) for N\_Hidden = 0']);


for NH = 1:length(N_Hidden_list)
N_Hidden = N_Hidden_list(NH);
W1=randn(3,N_Hidden);
W2=randn(N_Hidden+1,1);
x1_list = [0 0; 0 1; 1 0; 1 1]';
ideal_output = [0 1 1 0];
nonlinfunc = @(x) exp(x)./(1+exp(x));
dnonlinfunc = @(x) nonlinfunc(x).*(1-nonlinfunc(x));
alpha = 0.1; N=1e6; % alpha = learning rate, N = # of iterations
clear e_;
e_ = zeros(1,N);
for k=1:N
 i = randperm(4,1);
 x1 = x1_list(:,i); x1a = [x1;1]; % x1a = x1-augmented (augmented with a
    constant offest of 1)
 x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (augmented with a
    constant offest of 1)
 x3 = nonlinfunc(W2'*x2a);
 e = x3 - ideal_output(i);
 e_(k) = e;
 gradient2 = e_(k)*dnonlinfunc(W2'*x2a)*x2a';
 gradient1 = e_(k)*dnonlinfunc(W2'*x2a)*W2(1:N_Hidden).*dnonlinfunc(W1'*x1a).*
    x1a';
 W1 = W1 - alpha*gradient1';
 W2 = W2 - alpha*gradient2';
end
subplot(length(N_Hidden_list)+1,1,NH+1);
plot(e_,'.'); title(['Error e(n) for N\_Hidden = ',num2str(N_Hidden)]);
end
 xlabel('Iteration Number during training (n)')
x = 0:0.01:1;
y = x;
for m = 1:length(x)
    for n = 1:length(y)
        xin = x(m);
        yin = y(n);
        x1 = [xin;yin]; x1a = [x1;1]; % x1a = x1-augmented (augmented with a
            constant offest of 1)
        x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (augmented
            with a constant offest of 1)
        x3 = nonlinfunc(W2'*x2a);
        z(m,n) = x3;
    end
end
figure; imagesc(x,y,z); colorbar;
title('Output of Network For Values Between 0 and 1');
xlabel('Input 1');
ylabel('Input 2');
set(gca,'Ydir','normal');
set(gca,'Xdir','normal');

x = -1:0.01:2;
y = x;
for m = 1:length(x)
    for n = 1:length(y)
        xin = x(m);
        yin = y(n);
        x1 = [xin;yin]; x1a = [x1;1]; % x1a = x1-augmented (augmented with a
            constant offest of 1)
        x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (augmented
            with a constant offest of 1)
        x3 = nonlinfunc(W2'*x2a);
        z(m,n) = x3;
    end
end
figure; imagesc(x,y,z); colorbar;
title('Output of Network For Values Between -1 and 2');
xlabel('Input 1');
ylabel('Input 2');
set(gca,'Ydir','normal');
set(gca,'Xdir','normal');
```

The following script was used for the 3-layer neural network trained to approximate the product-of-sines function. It follows a similar workflow to the algorithm presented above, but uses a tanh nonlinearity to allow for negative output values.

```
clear all
N_Hidden_list = [5:5:30];
```

```
%N_Hidden_list = [1000,2000];
figure;
figure;

x = -2*pi:0.1:2*pi;
y = x;

for NH = 1:length(N_Hidden_list)
N_Hidden = N_Hidden_list(NH);
W1=randn(3,N_Hidden);
W2=randn(N_Hidden+1,1);
xx = -2*pi:0.1:2*pi;
for k1 = 1:length(xx)
    for k2 = 1:length(xx)
        ideal_output(k1,k2) = sin(xx(k1))*sin(xx(k2));
    end
end
nonlinfunc = @(x) tanh(x);
dnonlinfunc = @(x) (1-nonlinfunc(x).^2);
alpha = 0.1; N=1e6; % alpha = learning rate, N = # of iterations
clear e_;
e_ = zeros(1,N);
for k=1:N
 i = randperm(length(xx),1);
 j = randperm(length(xx),1);
 x1 = [xx(i);xx(j)]; x1a = [x1;1]; % x1a = x1-augmented (augmented with a
    constant offest of 1)
 x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (augmented with a
    constant offest of 1)
 x3 = nonlinfunc(W2'*x2a);
 e = x3 - ideal_output(i,j);
 e_(k) = e;

 gradient2 = e_(k)*dnonlinfunc(W2'*x2a)*x2a';
 gradient1 = e_(k)*dnonlinfunc(W2'*x2a)*W2(1:N_Hidden).*dnonlinfunc(W1'*x1a).*
    x1a';
 W1 = W1 - alpha*gradient1';
 W2 = W2 - alpha*gradient2';
end
figure(1)
subplot(length(N_Hidden_list),1,NH);
plot(e_,'.'); title(['Error e(n) for N\_Hidden = ',num2str(N_Hidden)]);
hold on
figure(2)
for m = 1:length(x)
    for n = 1:length(y)
        xin = x(m);
        yin = y(n);
        x1 = [xin;yin]; x1a = [x1;1]; % x1a = x1-augmented (augmented with a
           constant offest of 1)
        x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (augmented
           with a constant offest of 1)
        x3 = nonlinfunc(W2'*x2a);
        z(m,n) = x3;
    end
end
sq = ceil(sqrt(length(N_Hidden_list)));
```

```
    sq2 = floor(sqrt(length(N_Hidden_list)));
    subplot(sq,sq2,NH);
    imagesc(x,y,z); colorbar;
title(['Output of Network For Values Between -2\pi and 2\pi for N\_Hidden = ',
   num2str(N_Hidden)]);
end
figure(1)
 xlabel('Iteration Number during training (n)')
 hold off
 figure(2)


% for m = 1:length(x)
%     for n = 1:length(y)
%         xin = x(m);
%         yin = y(n);
%         x1 = [xin;yin]; x1a = [x1;1]; % x1a = x1-augmented (augmented with a
   constant offest of 1)
%         x2 = nonlinfunc(W1'*x1a); x2a = [x2;1]; % x2a = x2-augmented (
   augmented with a constant offest of 1)
%         x3 = nonlinfunc(W2'*x2a);
%         z(m,n) = x3;
%     end
% end

for k1 = 1:length(x)
    for k2 = 1:length(y)
        id(k1,k2) = sin(x(k1))*sin(y(k2));
    end
end
figure; imagesc(x,y,id);colorbar;
```

In order to convert the TrainABasicConvolutionalNeuralNetworkForClassificationExample.xml tutorial to one which could be used to examine network performance as a function of the amount of training data, the initial example was saved as a .m file and a small number of changes were made to vary the size of the training data. The first three lines below were used to replace line 54 in order to loop through a number of different training sizes. The accuracy output originally in on line 194 and now on line 196 was made a vector to store the accuracies for each training set size, and the results were plotted on a semilog plot after the loop ends.

```
        numTrainFiles_list = [10,20,50,100,200,500,750];
        NList = length(numTrainFiles_list);
        for nlisti = 1:NList

        accuracy(nlisti) = sum(YPred == YValidation)/numel(YValidation);
        end
        figure;
        semilogy(numTrainFiles_list,1-accuracy); title('Error rate as a function
            of training set size');
```

The following script is adapted from the provided code for simulating the state space model with one state variable over a range of different retention and learning coefficients. For each coefficient combination, learning begins after 10 trials without any dynamics (all values are set to zero), and learning is then simulated for 50 total trials according to the two-equation model discussed above. The state variable is plotted as a function of trial number for all combinations.

```
        A_list = [0.9,0.9,0.9,0.8,0.95,0.99];
B_list = [0.1,0.2,0.3,0.1,0.1,0.1];
figure;
for k = 1:length(A_list)
```

```
    A = A_list(k);
    B = B_list(k);
    lab(k) = convertCharsToStrings(['A = ', num2str(A), ', B = ', num2str(B)]);
    x_ideal = [zeros(10,1);ones(50,1)];
    N = length(x_ideal);
    [x, e] = deal(zeros(N,1));
    for n = 1:N
        e(n) = x_ideal(n) - x(n);
        x(n+1) = A*x(n) + B*e(n);
    end
    x = x(1:N);
    plot(x);
    hold on;
    grid on;
end

xlabel('Trial Number (n)');
ylabel('State x(n)');
title('Single State Space Model for Different Parameters');
legend(lab);
```

Finally, the following script was used to simulate the parallel two-state model for a single combination of learning and retention coefficients for both the fast and slow state variables for 120 trials after a 10-trial waiting period. After simulation, the trends of the two state variables, the total state variable equal to the sum of the two state variables, and the error are plotted over the course of learning.

```
        Af = 0.6;
As = 0.99;
Bf = 0.2;
Bs = 0.02;

ntri = 120;
xideal = [zeros(10,1);ones(ntri,1)];
N = length(xideal);
[x, xf, xs, e] = deal(zeros(N,1));

for n = 1:N
    e(n) = xideal(n) - x(n);
    xf(n+1) = Af*xf(n) + Bf*e(n);
    xs(n+1) = As*xs(n) + Bs*e(n);
    x(n+1) = xf(n+1) + xs(n+1);
end

figure;
plot(xs);
hold on
plot(xf);
plot(x);
plot(e);
xlabel('Trial Number (n)');
xlim([0,N]);
title('Two-State Model');
legend('x_s','x_f','x_{tot}','e');
hold off;
```