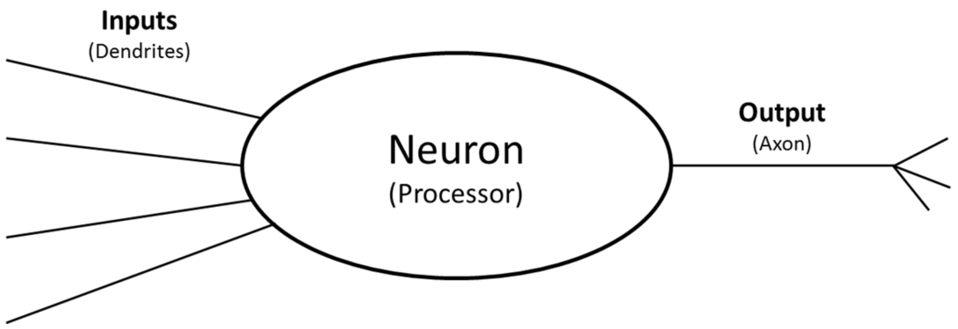
Lab 6: Neural Network for Pattern Recognition

A. Background Information[1]

The field of artificial intelligence is an exciting area in computer science that has seen enormous growth in the last ten years. Computers small enough to fit in your pocket regularly use face recognition for security, pattern recognition to learn your schedule, and network organization to almost instantaneously communicate with millions of people around the world. Some of the algorithms used for these purposes cluster data into groups. Some – like the one in this lab – attempt to mimic the human brain.

The human brain functions via connections between millions of cells called *neurons*. A schematic of a neuron is shown below. Each neuron has numerous inputs from other cells (*dendrites*), and one main output, called an *axon*. This single axon may eventually split into many different channels, and send its input to other neurons. Other neurons use this as an input, and the cycle repeats.



The interesting thing about neurons is that they function as mini computers. There are inputs, processing, and outputs. The processing done by neurons is simple – if enough of the inputs are ‘on’, then the neuron turns its output ‘on’. Some inputs may be more important than other inputs, and this allows a neuron to perform specialized functions. For example, a neuron that helps someone recognize an apple would pay much more attention to a *red* signal than a *blue* signal.

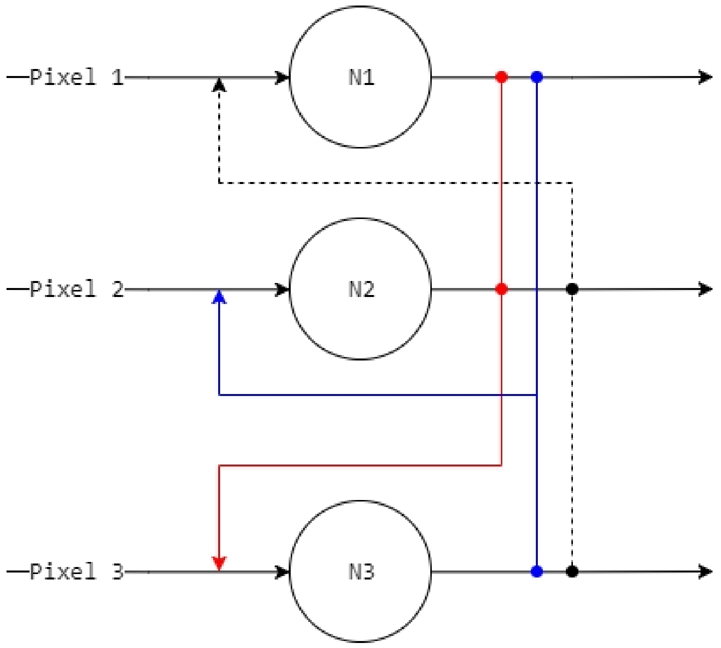
B. Understanding the Algorithm

In principle, a human brain may be simulated by a computer if it is programmed to simulate every neuron, and to calculate how all the neurons interact. The computer would then be able to perform any task that the human brain can. One of these tasks is image recognition, and is the subject of this lab. Luckily, we do not need to simulate each of the billions of neurons in the brain. Many algorithms have been developed that apply the fundamental principle of how neurons work to smaller groups of cells.

The algorithm presented here is a variation of the Hopfield Auto-Associative Model, which was made famous by John Hopfield in the 1980s. Hopfield’s network functions based on a group of

[1] Information from this section is adapted from *Artificial Intelligence: A Systems Approach* by M. Tim Jones

neurons working together to recognize a pattern. The input to the network is an image, and there is one neuron for each pixel in the image. To process a 20-pixel by 20-pixel image, the network would need 20*20 = 400 neurons[2]. A schematic of a network that recognizes patterns in a 3-pixel image is shown below:



To understand how this works, we'll focus on neuron 1, labeled "`N1`". This neuron has two inputs. One input says whether this cell's pixel is on (white) or off (black). It also has an input that connects it to both neurons 2 and neurons 3, shown above as the dotted black line. This allows N1 to 'see' what N2 and N3 are thinking, and to adjust its thinking accordingly. A pattern recognition sequence might go something like this[3]:

> **N1:** Based on pixel 1, this could be either pattern A or pattern B.
>
> **N2**: I am absolutely sure that, based on pixel 2, this must be pattern B.
>
> **N3**: I have no idea what is going on, because this pixel could be for pattern A, B, or C.
>
> **N1:** Well, N2 thinks that it's B, and my first guess was either A or B, so I'll go with B
>
> **N3:** Sounds good guys, I'll go with pattern B as well.

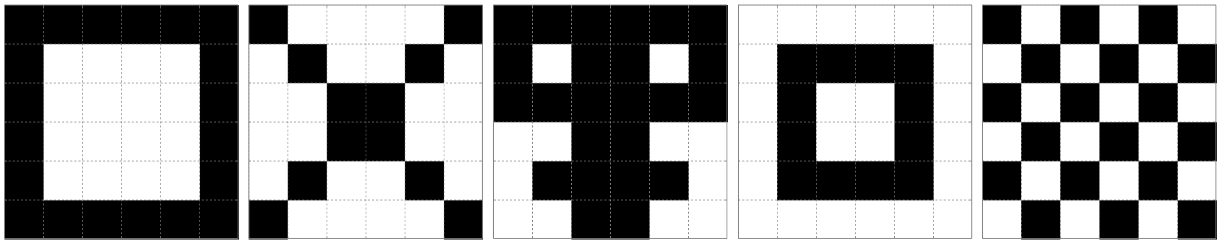
Because each neuron can see the output of the other neurons, the network is able to work as a group to come up with what it thinks the underlying pattern is.

[2] This becomes a problem with an HD image at 1920x1080, which requires 2.1 million neurons. There are ways to get around this requirement, and there are also much more efficient algorithms.
[3] This is a dramatization – the computer doesn't actually talk to itself

C. Training the Network

Just like a human, before a computer can recognize a pattern, it must be taught the pattern first. This process, called training, is what allows a network to have a *memory* of each possible pattern, and to then *recognize* the appropriate pattern when it is fed into the network.

The neural network in this lab will be trained to recognize five patterns, each of which are black and white images. These patterns are shown below, and are stored by the computer as ***1x36 vectors***. A special function, described later, reformats the 1x36 vector into a 6x6 matrix for display. ***You will be using the 1x36 vectors for calculations during this lab.***



The Hopfield network is trained to recognize patterns via a *weights matrix*. Each pattern has its own weight matrix, and when matrices for multiple patterns are added together, the system can recognize more than one pattern. The weight matrix for a single pattern can be calculated by finding the outer product of a pattern vector with itself, and then setting all the diagonal entries to zero. An example for a 3-entry pattern vector is shown below:

$$Outer\ Product: \qquad v \otimes v = v^T * v = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} * \begin{bmatrix} p_1 \, p_2 \, p_3 \end{bmatrix} = \begin{bmatrix} p_1p_1 & p_1p_2 & p_1p_3 \\ p_2p_1 & p_2p_2 & p_2p_3 \\ p_3p_1 & p_3p_2 & p_3p_3 \end{bmatrix}$$

$$Weight\ Matrix: \qquad W = \begin{bmatrix} 0 & p_1p_2 & p_1p_3 \\ p_2p_1 & 0 & p_2p_3 \\ p_3p_1 & p_3p_2 & 0 \end{bmatrix}$$

To practice this computation, type the following MATLAB commands at the command prompt to calculate the outer product for v1 and v2. Put your results in the tables below.

```
>> v1 = [1 2 3]

>> (v1')*v1
```

$v_1 \otimes v_1$		
1	2	3
2	4	6
3	6	9

```
>> v2 = [-1 1 2]

>> (v2')*v2
```

$v_2 \otimes v_2$		
1	-1	-2
-1	1	2
-2	2	0

The weight matrix for a network that will remember both v1 and v2 from above is calculated by adding the two outer products, and then setting the diagonal entries to zero. Perform this calculation, and place your results in the table below.

W for v1 and v2		
0	**1**	**1**
1	**0**	**8**
1	**8**	**0**

Now that you have explored how to calculate the weight matrix, you can train the network.

1. Download `patterns.mat`, `DisplayPattern.m`, and `AddNoise.m` from Canopy. Make sure that all three files are in MATLAB's current folder.
2. Create a new script file, and add `clear; close all; clc; commandwindow;` to the top of the file.
3. Use the `load` command to load `patterns.mat`.

Run the script, and open the `patterns` matrix that has loaded into MATLAB. Each **row** of the matrix is a pattern, and each **column** of the matrix is a pixel. Values of 1 represent white pixels, and values of −1 represent black pixels.

4. Use the `size` command to determine the number of patterns in the `patterns` variable, and the number of pixels in each pattern.
5. Display pattern 1 by adding the command `DisplayPattern(patterns(1,:))`. Run the script.
    a. The `DisplayPattern()` command is a custom function built for this lab. Its input is a row vector of 1's and −1's, and it outputs a figure like the ones shown at the beginning of Part C.
6. Initialize a *square* matrix called `weights` that is `NxN` and filled with zeros, where `N` is the number of pixels in each pattern vector.

The overall weight matrix for the five training patterns will be calculated using the same method that was used for v1 and v2 above. A for-loop will be used to pull out each *row* of the `patterns` variable. This is *not* a nested loop!

7. **For each row** in the `patterns` variable:
    a. Pull the entire row out of the patterns vector, and copy it into a variable `v`. (Hint: look at the code inside of the `DisplayPattern()` function. This code pulls out the first pattern).
    b. Calculate the outer product of `v` with itself
    c. Add the result of the above calculation into the current `weight` variable
8. After the loop has processed all the patterns, set the diagonal entries of the weight matrix to zero (hint: diagonal entries are where the `row == column`)
9. Once you have finished creating the weights matrix, run the following command at the command prompt, and paste the results below. **Check these results with your T.A.**

```
>> weights(3:6,3:6)
```

Results:

```
ans =

     0     3     5     1
     3     0     3     3
     5     3     0     1
     1     3     1     0
```

D. Recognizing a Pattern

The strength of a neural network is its ability to recognize a pattern even when the input signal contains noise. For example, face recognition should function even if a photo is taken at night, or if half of the face cropped out. In the second part of this lab, the weights matrix will be used to classify a noisy image as one of the original patterns. This process involves three steps:

1. Feed a noisy pattern to the network
2. Manipulate the raw output to make it easier to process
3. Compare the cleaned output to each of the original patterns, to determine which pattern matches best

Before we can feed noisy data to the computer, we must first get some additional input from the user. Implement the steps below inside your script file, **before the weight matrix calculation**.

1. The user will select which pattern to add noise to. Add an input statement to ask the user for a number between 1 and 5, and store this in the variable `patternNum`.
2. The user will select how much noise to add to their chosen pattern. Add an input statement to ask the user for noise level between 0 and 1, and store this in the variable `noiseLevel`.
3. Modify the `DisplayPattern()` statement already in the code to display the user's selected pattern, `patternNum`. Use a `subplot` command before the `DisplayPattern()` statement to create a 2x2 window, and put the original pattern in the first subplot. Don't forget a title!

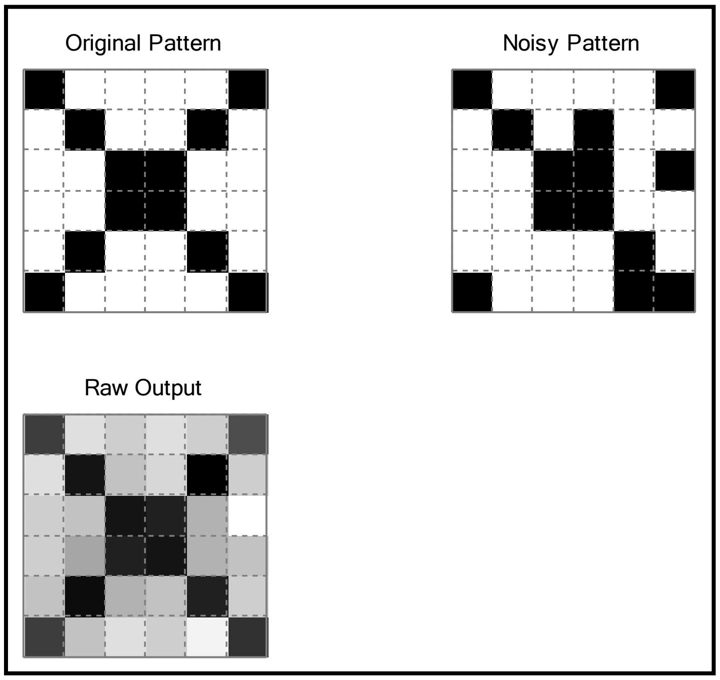
Now that we have the necessary user input, we are ready to use the weights matrix that we already created. Implement the steps below **after the weight matrix calculation** to create a noisy pattern, and feed it into the computer.

4. Add code to pull out the user's chosen pattern from the `patterns` matrix, and store this in a vector called `noisePattern`.
5. Use the `AddNoise()` function to add nose. This function was developed specifically for this lab. Its inputs are a vector and a noise level. Its output is a vector where some of the values have been randomly flipped from black to white (and vice-versa). The number of flipped pixels is based on noise level.

```
noisePattern = AddNoise(noisePattern, noiseLevel);
```

6. Use the `subplot` command to select the second subplot, and display the noisy pattern.
7. Feed the pattern into the network by multiplying the `noisePattern` variable and the `weights` variable. Store the output in a variable called `outputVector`.
    a. *Pay attention to the dimensions of `noisePattern` and `weights` to determine the proper order for multiplication! The outputVector variable should be 1x36.*
8. Display the `outputVector` pattern in the third window of the subplot.

At this point, run your code with test pattern number `2` and a noise level of `0.1`. If your code has run correctly, you should see something similar (but not identical) to the figure below:



As shown above, the user selected pattern 2, and the program randomly added noise to this pattern. The noisy pattern was fed into the neural network, and the neurons worked together to get the raw output shown above. The closer a pixel is to black in the raw output, the more certain the computer is that that pixel is black in the original pattern. Look in the bottom two rows. Notice how the network has filled in a missing black pixel, and removed an extra pixel! All of this was accomplished using a single matrix multiplication!

E. Cleaning Up the Network's Output

As humans, it is easy for us to see that the network's output matches the overall layout of pattern two. However, the computer has no way of 'seeing' like we do. The raw data must be processed to help the computer determine which of the original patterns it recognized.

To do this, the `outputVector` variable will be compared to each of the original vectors in the `patterns` matrix. For this to be a valid comparison, all the values in `outputVector` must be either 1 or −1, to match the original patterns.

1. Apply a *sigmoid squashing function* to the output vector using the line of code below. Look at values in `outputVector` before and after applying the function, and answer the question below.

```
outputVector = exp(outputVector)./(exp(outputVector) + 1);
```

The sigmoid function is often used in neural networks to keep the output of the network within reasonable bounds. What happens to values in the vector after the function has been applied?

All the values are bounded within 0 and 1.

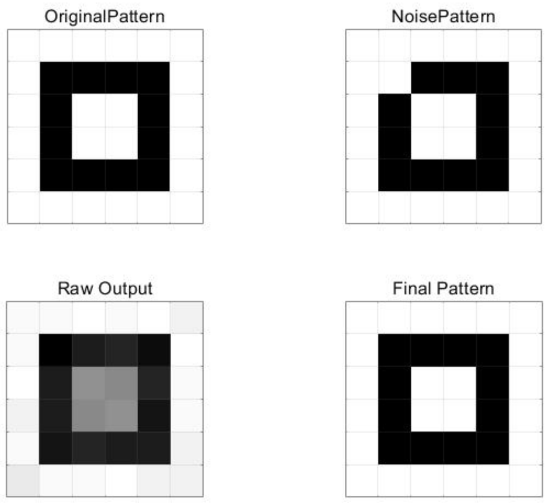
2. After the sigmoid function, develop code to look at every entry of `outputVector`. If the entry is > 0.9, reset the value to 1. Otherwise, reset the value to −1.
   Remember that `outputVector` is a 1x36 vector, and not a matrix. You should not need a nested for loop!
3. *After the code for step 2*, develop code that calculates a score for each of the original patterns. The score for each pattern is the count of how many entries in `outputVector` are equal to the corresponding entries in the original pattern. The scores should be stored in a vector that has as many entries as there are original patterns. In other words, the first entry in the score vector would be a count of how many entries in `outputVector` exactly match the first pattern (first row) in the `patterns` matrix.
4. *After the code for step 3,* develop code to determine which index in the scores vector contains the largest value. The location of the largest value is the 'recognized' pattern.

   Note: If the same maximum value occurs at multiple locations, pick the first location where the maximum occurs as the final index.

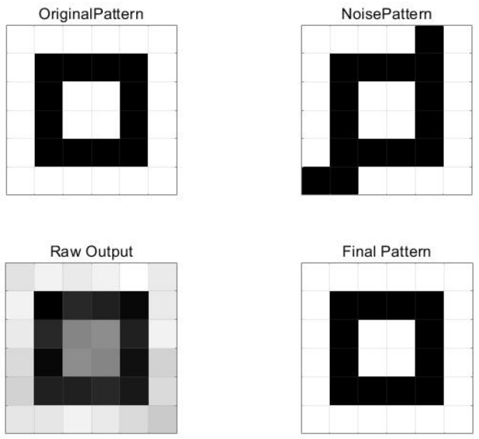
5. Based on the index where the largest score was found, display the final recognized pattern in the fourth subplot. Don't forget to add a title!

After you are satisfied that your script is functioning properly, run the program with the inputs shown below, and paste the resulting figures in the spaces provided. Then, answer the thought questions. To answer these questions, it may be useful to run your code multiple times for inputs other than those required below.

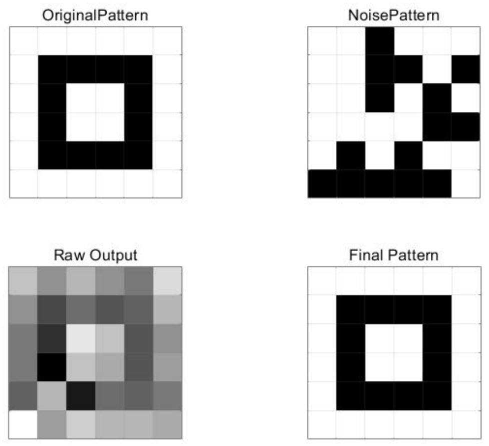
Pattern 4, Noise 0.05 (Paste figure below)



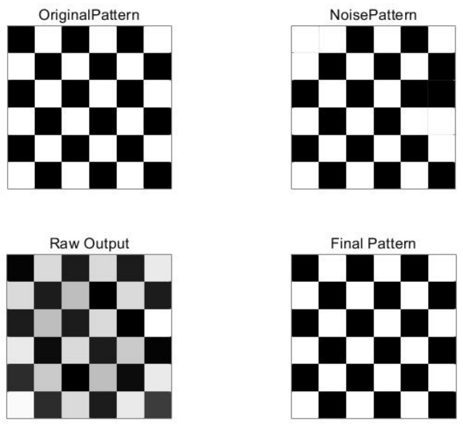
Pattern 4, Noise 0.15 (Paste figure below)



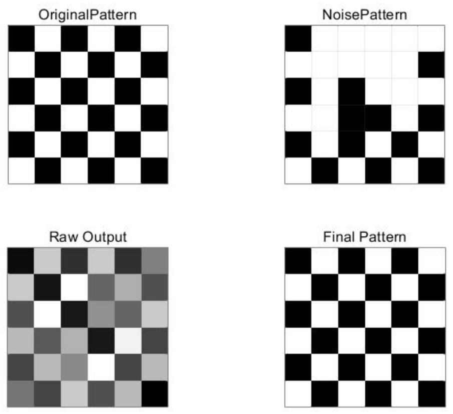
Pattern 4, Noise 0.45 (Paste figure below)



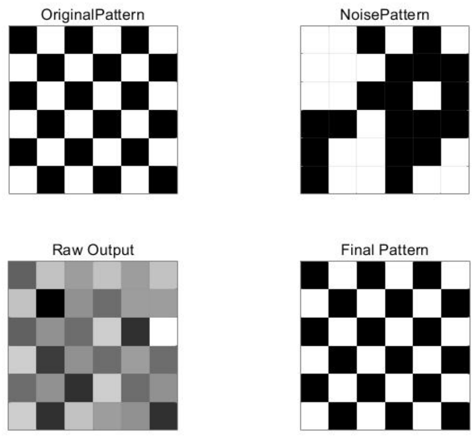
Pattern 5, Noise 0.05 (Paste figure below)



Pattern 5, Noise 0.15 (Paste figure below)



Pattern 5, Noise 0.45 (Paste figure below)



Thought Questions:

In general, what is the maximum amount of noise that the network can withstand and still reliably recognize the pattern?

1

Is there a difference between the amount of acceptable noise for a simple pattern, like pattern 4, and a complex pattern such as pattern 5?

The amount of acceptable noise for a simple pattern is more than a complex one.

When the network was trained, the original, noise-free patterns were used. However, there are times when a network may be trained with noisy data. What do you think is the best method? Provide justification for your answer!

I think the best method is to train a network with noisy data.
Because compared with the former one, this method can improve the network better.

PASTE MATLAB CODE HERE:

```
%% LAB6
% Name: Horace
% Date: 5 Mar 2019

%% Code
%clear processor
```

```
clear; close all;clc;commandwindow
%% Preparation
load patterns
patterns_size=size(patterns);
n=length(patterns);
weights=zeros(n,n);
%% Original pattern
patternNum=input('What number do you want to input between
1 and 5?');
noiseLevel=input('What noise level do you want to input
between 0 and 1?');
subplot(2,2,1)
DisplayPattern((patterns(patternNum,:)))
title('OriginalPattern');
%% Weight matrix calculation
for i=1:patterns_size(1)
v=patterns(i,:);
weights=weights+(v')*v;
end

for row=1:n
    for column=1:n
        if row==column
            weights(row,column)=0;
        end
    end
end
%% NoisePattern & RawOutput
noisePattern = patterns(patternNum,:);
noisePattern = AddNoise(noisePattern, noiseLevel);
subplot(2,2,2)
DisplayPattern(noisePattern);
title('NoisePattern');
outputVector=noisePattern*weights;
subplot(2,2,3)
DisplayPattern(outputVector);
title('Raw Output');
%%  Cleaning Up the Network¡¯s Output
outputVector = exp(outputVector)./(exp(outputVector) + 1);
for i=1:36
    if outputVector(i)>0.9
        outputVector(i)=1;
    else
        outputVector(i)=-1;
    end
end
count=zeros(1,5);
for i=1:5
    for j=1:36
    if outputVector(j)==patterns(i,j)
        count(1,i)=count(1,i)+1;
    end
    end
end
[m,p]=max(count);
subplot(2,2,4)
DisplayPattern(patterns(p,:));
title('Final Pattern');
```