Constructing an Oscillatory Associative Memory Model

Catherine Zeng

Spring 2019

1 Introduction

Biological memory functions very differently from computer memory because information is retrieved by content rather than by address; unlike computer memory, content-addressable memory is robust to noise and can be retrieved by partial content information. One of the most well-known model of associative memory is the Hopfield network [1], based on the Ising model of ferromagnetism. However, it is important to find alternative models capable of pattern recognition for studying memory and learning because the Hopfield network scales poorly in large simulations; it is fully-connected and has an asymptotic time complexity of $O(n^2)$ (every node is connected to every other node).

An alternative model of associative memory uses limit cycle attractors to store information rather than point attractors. This relatively unexplored model encodes patterns as constants of coupling in oscillators and is supported by experimental recordings that show the synchronization of neuronal firings plays an important role in information processing for the olfactory bulb, hippocampus, and thalamo-cortical system [2].

The goal of this project is to construct an oscillatory associative memory model and evaluate it relative to the Hopfield network for advantages such as scaling and interference levels. We will specifically explore a model that stores patterns using synchronized chaotic states and phase relations between oscillators called the Star Cellular Neural Network (Star CNN) [3].

2 Star Cellular Neural Network

2.1 Star Network Topology

For the Star CNN model, N cells of local dynamical systems are connected with a central system (also known as the master cell) in a unifying nonlinear dynamical system as shown in Figure 1 so that a fully-connected system, such as that of the Hopfield network, needs $N(N+1)/2$ connections whereas the Star CNN only needs $N+1$ connections. In this model, all cells communicate only through the central system.

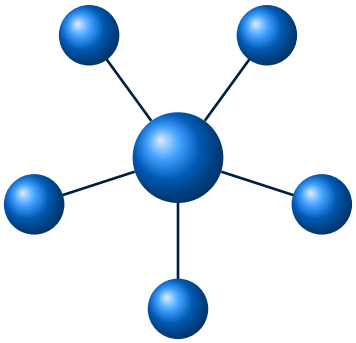


Figure 1: Star Network Topology

2.2 Storing memories and updating each cell

For storing memories, we represent memories as M binary patterns of length N containing -1s and 1s σ^1, σ^2, ..., σ^M:

$$\sigma^1 = \begin{bmatrix} \sigma_1^1 \\ \sigma_2^1 \\ \vdots \\ \sigma_N^1 \end{bmatrix}, \sigma^2 = \begin{bmatrix} \sigma_1^2 \\ \sigma_2^2 \\ \vdots \\ \sigma_N^2 \end{bmatrix}, \ldots, \sigma^M = \begin{bmatrix} \sigma_1^M \\ \sigma_2^M \\ \vdots \\ \sigma_N^M \end{bmatrix}$$

In our memory simulations later, these matrices would represent pixels values for flattened images with 1 and -1 representing black and white respectively.

For example, the following matrix represents a 5×5 pixel image with each of the cells being an element in a matrix of length $N = 25$.

$$\sigma^T = \big[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25\big]$$

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 2: 5×5 pixel image represented by a matrix of length 25

We represent each cell as a network of nodes and weights (coupling coefficients) so that the memory converges to a pattern for the given input when $M \ll N$. To find the coupling coefficients, we apply the following:

$$s_{ij} = \frac{1}{N} \sum_{m=1}^{M} \sigma_i^m \sigma_j^m \tag{1}$$

To update the nodes / states, we assign an initial state $v_i(0)$ and apply the following network update rule:

$$x_i(n+1) = sgn\left(\sum_{j=1}^{N} s_{ij} x_j(n)\right) \tag{2}$$

where:

$$\text{sgn(x)} = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases}$$

2.3 First-order Star CNNs

We model the change of a cell's state as a function of the current state x_i and input u_i as follows:

$$\frac{\partial x_i}{\partial t} = -x_i + u_i \tag{3}$$

For the Star CNN model, we can substitute output y_i and u_i into the above as follows:

$$y_i = h(x_i) = sgn(x_i) \tag{4}$$

$$u_i = sgn\left(\sum_{j=1}^{N} s_{ij} y_j\right) \longrightarrow \left(\sum_{j=1}^{N} s_{ij} sgn(x_j)\right) \tag{5}$$

$$\frac{\partial x_i}{\partial t} = -x_i + sgn\left(\sum_{j=1}^{N} s_{ij} sgn(x_j)\right) \tag{6}$$

Adding Δx to x gets equation (2), showing that the Star CNN model has the basic properties of associative memory.

2.4 Second-order Star CNNs

We first divide the dynamical system from the first-order Star CNN model into two subsystems:

$$\frac{\partial x_i^n}{\partial t} = f_i^1(x_i^1, x_i^2, \ldots, x_i^n) \tag{7}$$

$$\begin{cases} \frac{\partial x_i}{\partial t} = f(x_i, y_i), \\ \frac{\partial y_i}{\partial t} = g(x_i, y_i). \end{cases} \tag{8}$$

Applying the Star Network Topology, we supply an input signal u_i to each cell from the master cell (the central node), so that we modify the dynamics to be:

$$\left.\begin{array}{l} \frac{\partial x_i}{\partial t} = f(x_i, y_i) + u_i, \\ \frac{\partial y_i}{\partial t} = g(x_i, y_i). \end{array}\right\} (i = 1, 2, 3, \ldots, N) \tag{9}$$

The following system will synchronize when the coupling coefficient d is large enough and the signal x_0 is provided by the master cell:

$$\left.\begin{array}{l} \frac{\partial x_0}{\partial t} = f(x_0, y_0) \\ \frac{\partial y_0}{\partial t} = g(x_0, y_0). \end{array}\right\} \tag{10}$$

$$u_i = d\left(sgn\left(\sum_{j=1}^{N} s_{ij} * sgn(x_0 x_j)\right) x_0 - x_i\right) \tag{11}$$

where x_0 is the driving signal, u_i is the input signal, and d is the coupling coefficient.

2.5 Two-celled second-order CNN model

The example of a second-order Star CNN cell that we implemented in our memory simulations is the Two-celled CNNs model [4], which is described as follows:

$$\frac{\partial x_i}{\partial t} = -x_i + ph(x_i) + rh(y_i) \tag{12}$$

$$\frac{\partial y_i}{\partial t} = -y_i + sh(x_i) + qh(y_i) \tag{13}$$

where p, q, r and s are constants and:

$$h(x_i) = \frac{1}{2}(|x_i + 1| - |x_i - 1|) \tag{14}$$

The coefficients used for the variables respectively in our memory simulations were $p = 1.1, q = 1.1, r = -2$, and $s = 2$.

3 Synchronization

Proving that convergence is possible is important for any memory system model. In the Star CNN oscillatory associative memory model, patterns are retrieved by synchronizing the input state with a stored state in memory. This can be proved using the Lyapunov-Malkin theorem which which can be used to describe the nonlinear stability of feedback in a system of differential equations.

3.1 Lyapunov–Malkin theorem

Given the following first-order differential equation form:

$$\dot{x} = Ax + X(x, y), \quad \dot{y} = Y(x, y) \tag{15}$$

where A is a m by m matrix, $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, and $X(x, y)$ and $Y(x, y)$ represent higher order nonlinear terms, the Lyapunov–Malkin theorem states that the solution $x = 0$, $y = 0$ of this system is stable with respect to (x, y) and asymptotically stable with respect to x when $X(x, y)$, $Y(x, y)$ vanish when $x = 0$ and the eigenvalues of the matrix A have negative real parts. Further, if $(x(t), y(t))$ is close to the solution $x = 0$, $y = 0$, then:

$$\lim_{t\to\infty} x(t) = 0, \quad \lim_{t\to\infty} y(t) = c \tag{16}$$

3.2 Synchronization in our model

In order for patterns to be retrieved by synchronizing the input state with a stored state in memory, we want the trajectories of our system to converge to the same values as each other and remain in step so that the synchronization is structurally stable. We first rewrite the following second-order Star CNN model into a first-order differential equation system:

$$\left.\begin{array}{l} \frac{\partial x_i}{\partial t} = f(x_i, y_i) + d(x_0 - x_i), \\ \frac{\partial y_i}{\partial t} = g(x_i, y_i). \end{array}\right\} (i = 1, 2, 3, \ldots, N) \tag{17}$$

where $\Delta x = x_i - x_0$ and $\Delta y = y_i - y_0$:

$$\dot{\xi} = A\xi$$

$$\longrightarrow \frac{\partial}{\partial t}\begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix} = \begin{bmatrix} \frac{\partial f(x_0,y_0)}{\partial x_0} - d & \frac{\partial f(x_0,y_0)}{\partial y_0} \\ \frac{\partial g(x_0,y_0)}{\partial x_0} & \frac{\partial g(x_0,y_0)}{\partial y_0} \end{bmatrix} \times \begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix} \tag{18}$$

In equation (18), A is the Jacobian matrix. According to the Lyapunov-Malkin theorem, the system synchronizes when the eigenvalues of the Jacobian (we will call these the Lyapunov exponents) are negative. Although this is a necessary condition for synchronization, it is not the only condition; synchronization is also dependent on settings for the initial condition.

4 Memory Simulations

We run our simulations by first testing the algorithm on constrained 5×5 grid patterns, and then testing on digits from the MNIST database of handwritten digits (a much larger and more complex dataset consisting of digits that are 28×28 pixel grids. In both parts, we store the patterns in memory and feed it an input close to a stored pattern to retrieve, and image files are flattened into one-dimensional vectors.

For the purpose of reproducibility, our implementation is open-sourced on Github here. The repository contains, among others, the following files:

- mnist.pkl.gz - the MNIST database stored in pickle form
- MNIST digits.ipynb - implementation of a first-order Star CNN for the MNIST database
- Star CNN tests.ipynb - tests done on 5×5 grids of simpler patterns

4.1 Tests on 5×5 constrained grids

Some tests of smaller patterns were done first to test the convergence of the memory model. Code for this section can be found in MNIST digits.ipynb of the Github repository. Here, the following figures, composed of 5×5 grids, were stored in associative memory as seen in Figure 2 and Figure 3:

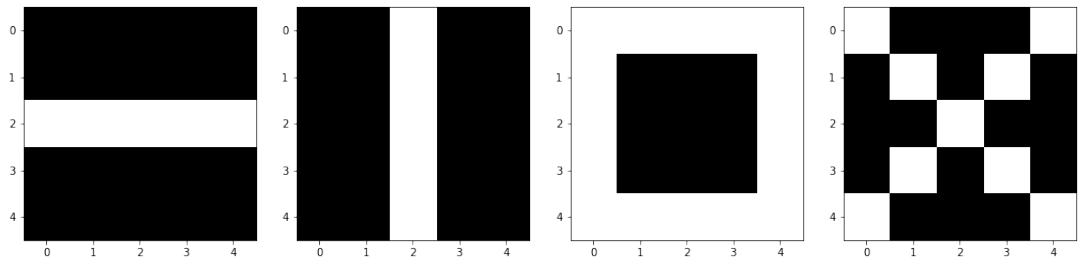


Figure 3: 5×5 grids of patterns stored in the memory model

In the following Figure 4, from left to right, the first pattern converged to the second, the third converged to the fourth, and so on.

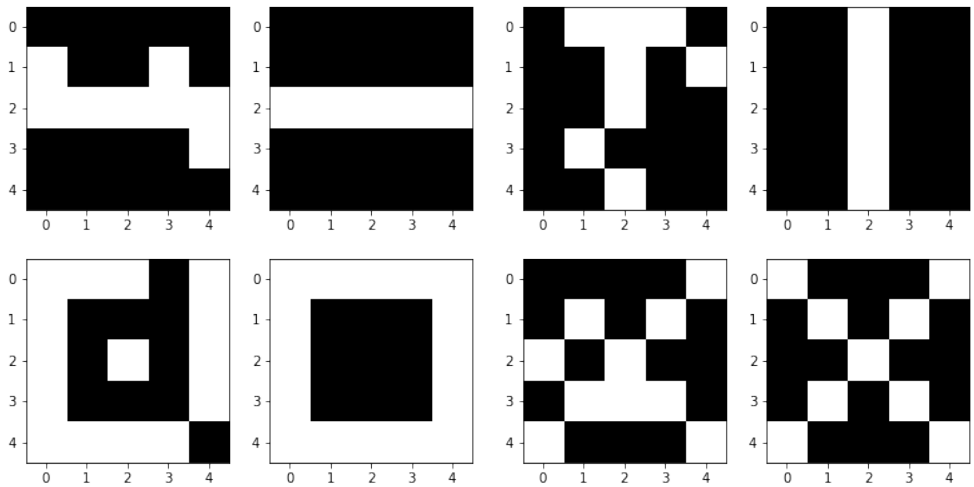


Figure 4: 5×5 grids patterns converging on patterns stored in the memory model

In this constrained case of 4 patterns of 5×5 grids, the memory system returned the correct grid extremely reliably without much interference.

4.2 MNIST database

We use the MNIST database, a large collection of handwritten digits that is commonly used in training image processing machine learning tasks. The MNIST database contains 60,000 training images and 10,000 testing images, however, since associative memory models does not need training to function, we will only be using a small subsection of this database.

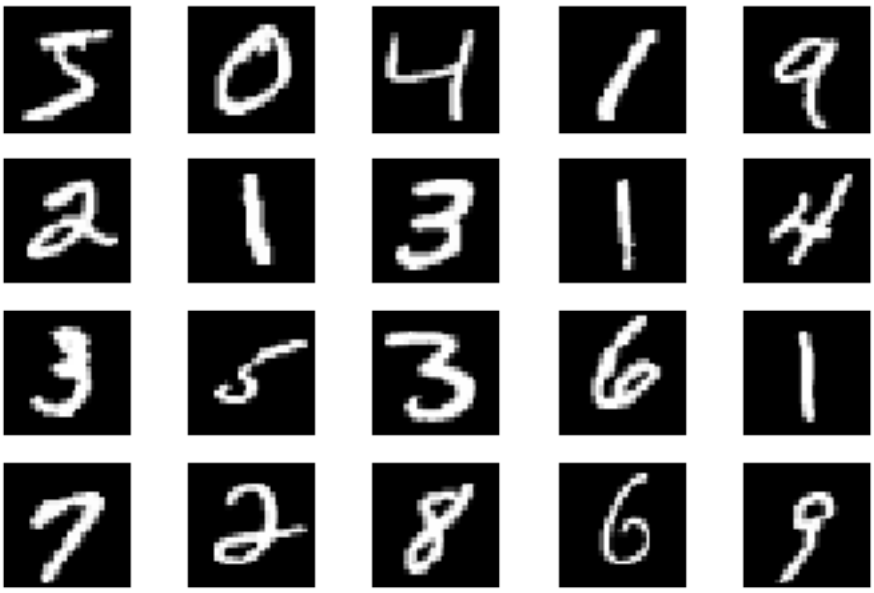


Figure 5: The first 20 digits from the training set of the MNIST database

We processed each image so that the pixels were either 1 or -1, so that Equation (2) could return a signed value for x.

Figure 6: Sample containing one of each of the pre-processed 10 digits

4.3 Retrieval tests

In contrast to the 5×5 grids, there was significantly more interference. In Figure 6 below, the digits retrieved by inputting digits 9 and 5 were very similar, with the line bleeding into the 5 that transitions into a 9. The digit 0 also bled in the middle due to interference from storing a different digit. However, most impressive was the 7, since the pattern returned did not fall in the same pixel locations as the pattern inputted.

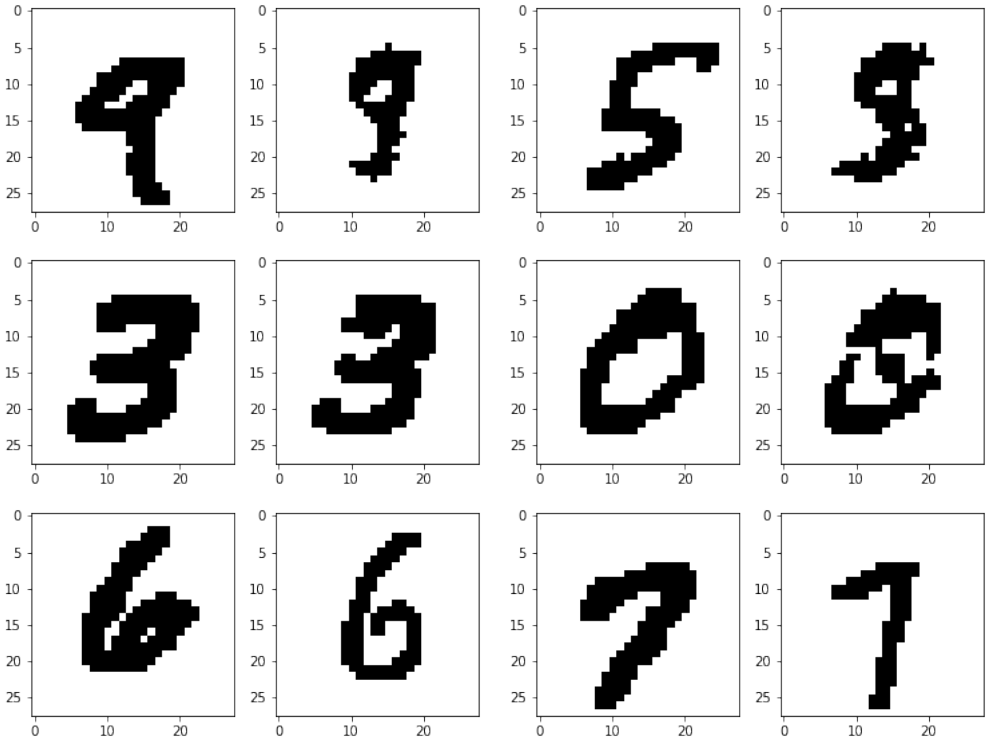


Figure 7: Retrieval of the digits where the left image is the input digit and the right image is the retrieved, output digit.

4.4 Interference

For the 5×5 grid example, a pattern of + was generated without ever being stored when prompted with the following input:

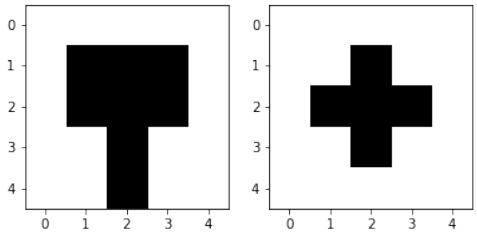


Figure 8: A pattern of + retrieved that was never stored.

This was produced as a result of interference, where the network returns a attractor, or pattern, following a lower energy state.

The memory network was further incapable of storing too many patterns without seeing interference in the MNIST database (In Figure 7, only 10 digits, one of each in the range $0 - 9$ from Figure 6, were stored). Storing more digits, even at just 20, led to significant interference. These are called spurious states or patterns, and can be seen in the figure below:

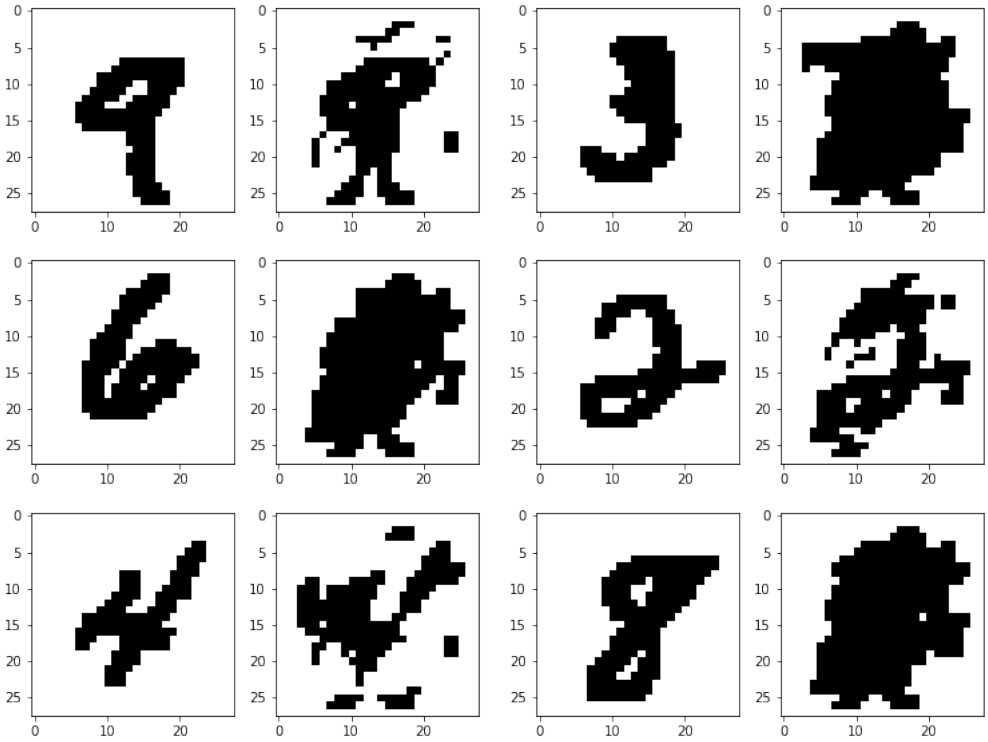


Figure 9: Input patterns would converge on a version of a pattern containing interference.

Although the patterns returned in response to inputs 6 and 3 were unrecognizable, the pattern returned to 2 and 9 retained some more form (the pattern returned by digit 9 even looks like an action figure or the Reddit logo). For Hopfield networks, the network can store up to around 14% before interference causes the network to start returning spurious states (unreliably retrieve inputs corresponding to local minima in the energy function).

It's unclear the role of spurious states in biological memory. Although spurious states allow patterns never seen before to be retrieved, they impact the reliability of memory retrieval.

5 Conclusion

We ran the first-order and second-order Star CNN associative memory models on 5×5 constrained grids and the MNIST database of handwritten numbers (containing 28×28 pixel digits) in simulations for testing memory retrieval. We stored patterns in the form of flattened images through their coupling coefficients and prompted the network with an input. Both the 5×5 constrained grids and the MNIST database containing one of each of the 10 digits returned the correct pattern reasonably reliably when few inputs were stored, however when more patterns were stored, we started seeing interference (this was especially noticeable in the MNIST database, see Figure 8). In the case of the 5×5 grids, because the samples were relatively constrained, the interference became a dynamic pattern in the shape of a + (see Figure 7), however the interference patterns seen in the MNIST database were less patterned.

Although the Star CNN associative memory network scales better than the Hopfield network, since it uses $N + 1$ connections as opposed to $N(N + 1)/2$ connections, it struggles with interference. For this reason, it is unclear whether the network capacity of oscillatory associative memory (that uses limit cycle attractors) is greater than an associative memory that uses point attractors for information retrieval such as the Hopfield network.

Future exploration would involve stress testing the oscillatory associative memory models to see exactly how and when it breaks, as well as creating benchmarks for performance. I would also like

to investigate how coupled oscillations work from a more theoretical basis through understanding the Kuramoto model and test other oscillatory models such as the Piecewise-linear Van der Pol oscillator, and third order Star CNN systems through implementing Chua's oscillator and Canonical Chua's oscillator [3].

6 Acknowledgements

I would first like to thank Jordan Wick and Helen Read for their camaraderie while taking this class, and Zachary Pitcher for providing emotional support.

I would also like to thank this class and Philip Pearce for inspiring me to become more observant of our world and equipping me with tools to mathematically model these observed phenomena. For example, I've noticed since taking this class when water would suddenly disperse in mid-air while spitting from a balcony, freckles that come from Turing instabilities, and the oscillatory patterns in the waves from the Charles river. These phenomena are beautiful, and I am excited to learn why they exist through modeling them in future personal projects.

References

[1] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, pp. 2554–8, 05 1982.

[2] F. Hoppensteadt and E. Izhikevich, "Pattern recognition via synchronization in phase-locked loop neural networks," *IEEE Transactions on Neural Networks*, vol. 11, no. 3, p. 734–738, 2000.

[3] M. Itoh and L. O. Chua, "Star cellular neural networks for associative and dynamic memories," *International Journal of Bifurcation and Chaos*, vol. 14, no. 05, p. 1725–1772, 2004.

[4] F. Zou and J. Nossek, "Stability of cellular neural networks with opposite-sign templates," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 6, p. 675–677, 1991.

MNIST digits

May 8, 2019

In [172]:
```
import pickle, gzip, numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import math
import random
```

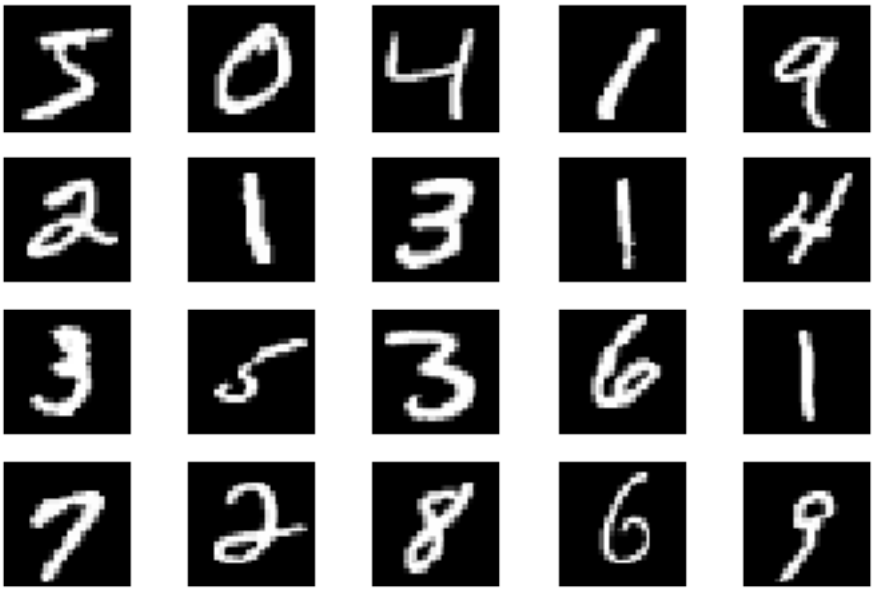
0.0.1 Importing MNIST database:

In [173]:
```
def plotImages(X):
    if X.ndim == 1:
        X = np.array([X])
    numImages = X.shape[0]
    numRows = math.floor(math.sqrt(numImages))
    numCols = math.ceil(numImages/numRows)
    for i in range(numImages):
        reshapedImage = X[i,:].reshape(28,28)
        plt.subplot(numRows, numCols, i+1)
        plt.imshow(reshapedImage, cmap = cm.Greys_r)
        plt.axis('off')
    plt.show()

def readPickleData(fileName):
    f = gzip.open(fileName, 'rb')
    data = pickle.load(f, encoding='latin1')
    f.close()
    return data

def getMNISTData():
    trainSet, validSet, testSet = readPickleData('mnist.pkl.gz')
    trainX, trainY = trainSet
    validX, validY = validSet
    trainX = np.vstack((trainX, validX))
    trainY = np.append(trainY, validY)
    testX, testY = testSet
    return (trainX, trainY, testX, testY)

def plotImagesHorizontal(X):
```

```
    if X.ndim == 1:
        X = np.array([X])
    numImages = X.shape[0]
    numRows = 1
    numCols = numImages
    for i in range(numImages):
        reshapedImage = X[i,:].reshape(28,28)
        plt.subplot(numRows, numCols, i+1)
        plt.imshow(reshapedImage, cmap = cm.Greys_r)
        plt.axis('off')
    plt.show()

(trainX, trainY, testX, testY) = getMNISTData()
```

We print out the first 20 digits of the MNIST training dataset to get a feel for the dataset.

```
In [174]: plotImages(trainX[0:20,:])
```



For our first proof of concept, we will store one of each of the 9 digits in associative memory.

```
In [175]: digits = [1, 6, 5, 7, 26, 0, 18, 42, 17, 43]

plotImagesHorizontal(np.array([trainX[i] for i in digits]))
```

Pre-process the data so that the digit arrays are composed ot 1s and -1s.

```
In [176]: N = 784 # the mnist digits are 28 by 28 pixels and 28^2 = 784
          s = np.zeros(shape=(784,784))

          # converts digits to 1s and -1s
          return_abs = lambda x: -1 if x > 0 else 1

          # grabs the absolute values for each item in the array for mnist digit
          get_abs = lambda x: np.array([return_abs(i) for i in trainX[x]])
```

Plot the images to see what they look like after processing.

```
In [177]: plotImagesHorizontal(data)
```

0.0.2 Storing Information:

We generate the coupling coefficients according to the following equation:

$$s_{ij} = \frac{1}{N} \sum_{m=1}^{M} \sigma_i^m \sigma_j^m \tag{1}$$

```
In [335]: # stores one of each digit
          data = np.array([get_abs(i) for i in digits])
          # data = trainX[0:20]

          # grabs the coupling coefficients
          for i in range(784):
              for j in range(784):
                  total = 0
                  for k in range(len(data)):
                      total += data[k][i] * data[k][j]
                  s[i][j] = 1/784 * total
```

0.0.3 Retrieving information:

$$y_i = h(x_i) = sgn(x_i) \tag{2}$$

$$u_i = sgn\left(\sum_{j=1}^{N} s_{ij} y_j\right) \longrightarrow \left(\sum_{j=1}^{N} s_{ij} sgn(x_j)\right) \tag{3}$$

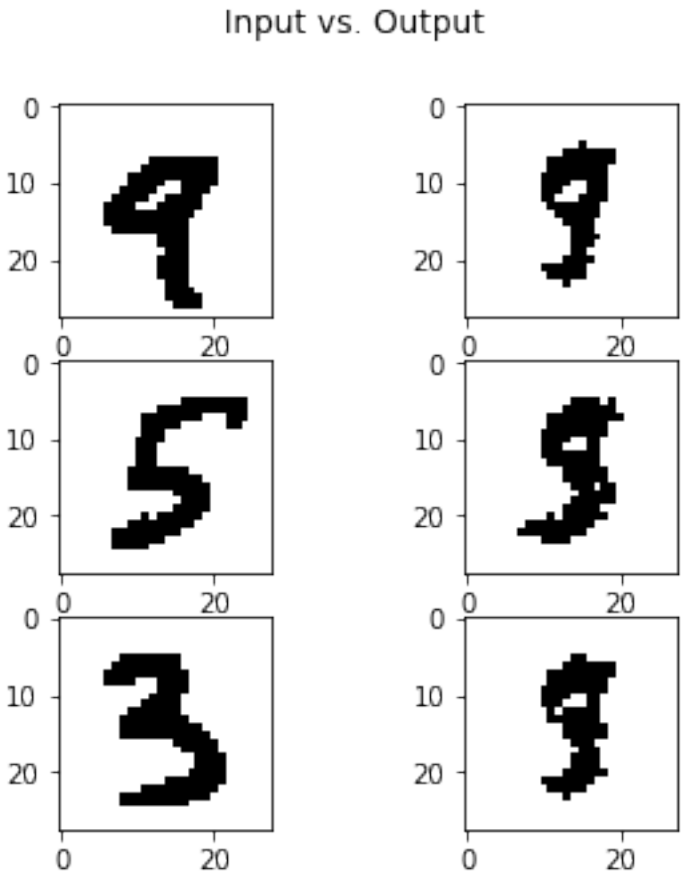
$$\frac{\partial x_i}{\partial t} = -x_i + sgn\left(\sum_{j=1}^{N} s_{ij} sgn(x_j)\right) \tag{4}$$

```
In [231]: def testDigit(input_digit):
              test = np.array([return_abs(i) for i in input_digit])
              V = test # this is the initial state (where the tested image goes)
              new_V = np.ones(784)

              for i in range(784):
                  total_sV = 0
                  for j in range(784):
                      total_sV += s[i][j] * V[j]

                  if total_sV >= 0:
                      new_V[i] = 1
                  else:
                      new_V[i] = -1

              V = new_V
              V_resized = np.array(V).reshape(28, 28)
              test_resized = np.array(test).reshape(28, 28)

              # returns array of input digit and array of retrieved digit
              return test_resized, V_resized
```

0.0.4 Test memory retrieval:

```
In [336]: # testing retrieval of digit 9 from memory
          input_9, output_9 = testDigit(trainX[4])

          # testing retrieval of digit 5 from memory
          input_5, output_5 = testDigit(testX[23])

          # testing retrieval of digit 5 from memory
          input_3, output_3 = testDigit(testX[32])

          f = plt.figure()
          f.set_size_inches(5, 5)
          f.suptitle("Input vs. Output", fontsize=12)

          f.add_subplot(321)
          plt.imshow(input_9, cmap = cm.Greys_r)

          f.add_subplot(322)
```

```
plt.imshow(output_9, cmap = cm.Greys_r)

f.add_subplot(323)
plt.imshow(input_5, cmap = cm.Greys_r)

f.add_subplot(324)
plt.imshow(output_5, cmap = cm.Greys_r)

f.add_subplot(325)
plt.imshow(input_3, cmap = cm.Greys_r)

f.add_subplot(326)
plt.imshow(output_3, cmap = cm.Greys_r)
plt.show()
```

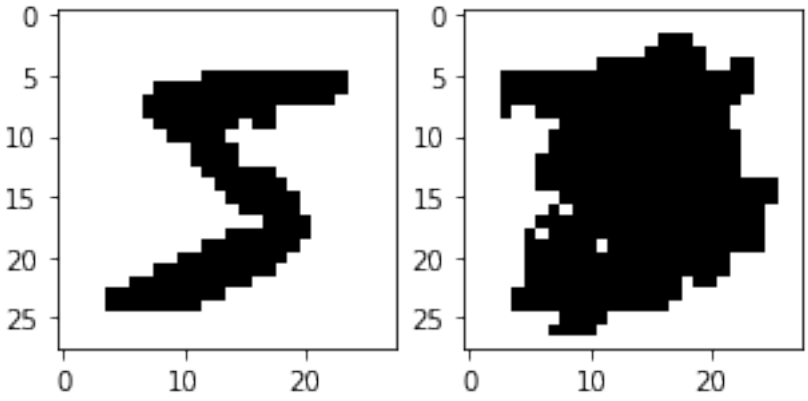


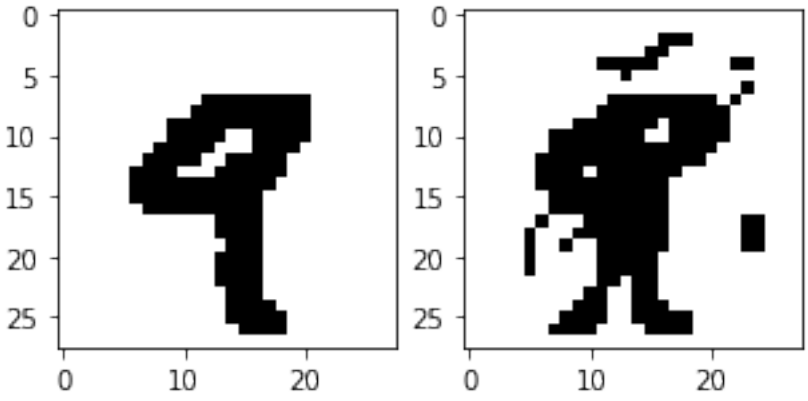
Here we can see how the digits bleed into each other due to interference from the network.

```
In [337]: # stores 20 digits
          data = trainX[0:20]
```

```
        # grabs the coupling coefficients
        for i in range(784):
            for j in range(784):
                total = 0
                for k in range(len(data)):
                    total += data[k][i] * data[k][j]
                s[i][j] = 1/784 * total
```

In [333]:
```
for i in [0, 4]:
    inputX, outputX = testDigit(trainX[i])

    f = plt.figure()
    f.set_size_inches(2, 2)
    f.add_subplot(121)
    plt.imshow(inputX, cmap = cm.Greys_r)
    f.add_subplot(122)
    plt.imshow(outputX, cmap = cm.Greys_r)
    plt.show()
```





Star CNN tests

May 8, 2019

0.1 1) Finding the Coupling Coefficients:

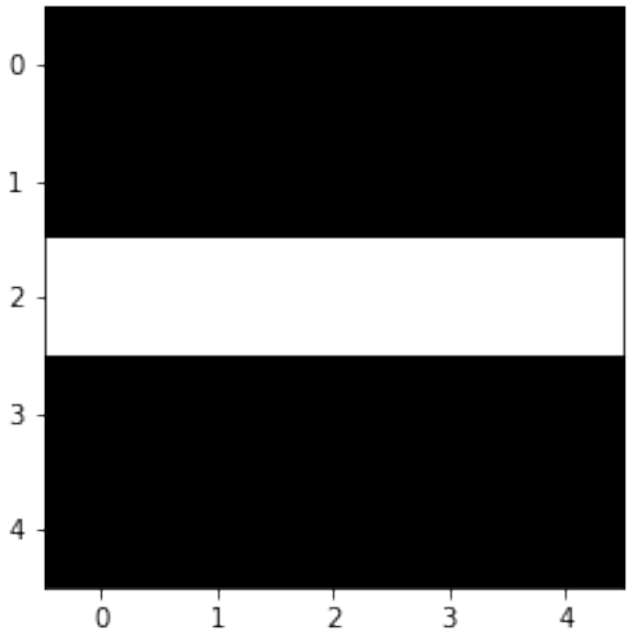
In [2]:
```
import numpy as np
import pickle, gzip, numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import math
import random
```

In [3]:
```
sigma_1 = np.array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, -1, -
sigma_2 = np.array([-1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1,
sigma_3 = np.array([1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1, 1,
sigma_4 = np.array([1, -1, -1, -1, 1, -1, 1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, 1,
```

0.1.1 Print the figures:

In [10]:
```
sigma_1_show = np.array(sigma_1).reshape(5, 5)
sigma_2_show = np.array(sigma_2).reshape(5, 5)
sigma_3_show = np.array(sigma_3).reshape(5, 5)
sigma_4_show = np.array(sigma_4).reshape(5, 5)
plt.imshow(sigma_1_show, cmap = cm.Greys_r)
# plt.imshow(sigma_2_show, cmap = cm.Greys_r)
# plt.imshow(sigma_3_show, cmap = cm.Greys_r)
# plt.imshow(sigma_4_show, cmap = cm.Greys_r)
```

Out[10]:
```
<matplotlib.image.AxesImage at 0x7f3ca52a9fd0>
```




```
In [13]: s = np.zeros(shape=(25,25))

         for i in range(25):
             for j in range(25):
                 s[i][j] = 1/25 * (sigma_1[i] * sigma_1[j] + sigma_2[i] * sigma_2[j] + sigma_3[i

         # print(25 * s)
```

This generates a 25 by 25 matrix of the coupling coefficients.

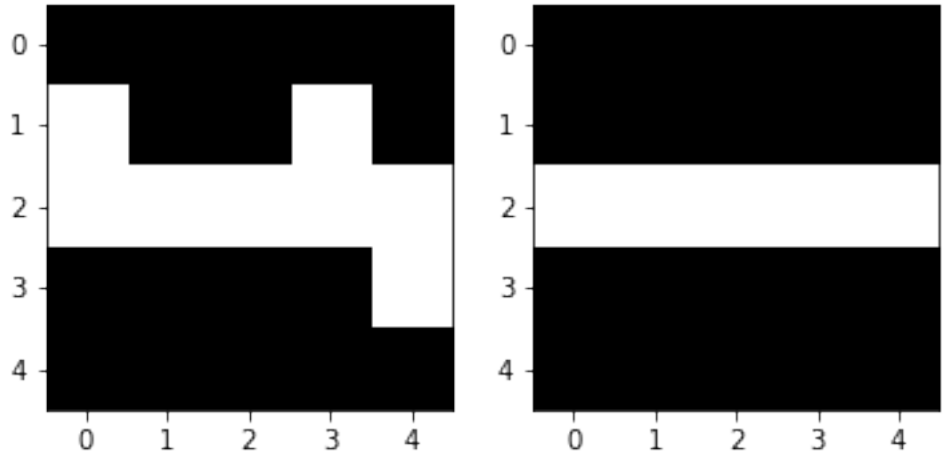
0.2 2) Assign the initial state $v_i(0)$ and update the network:

0.2.1 Test with a sample squares, see if converge.

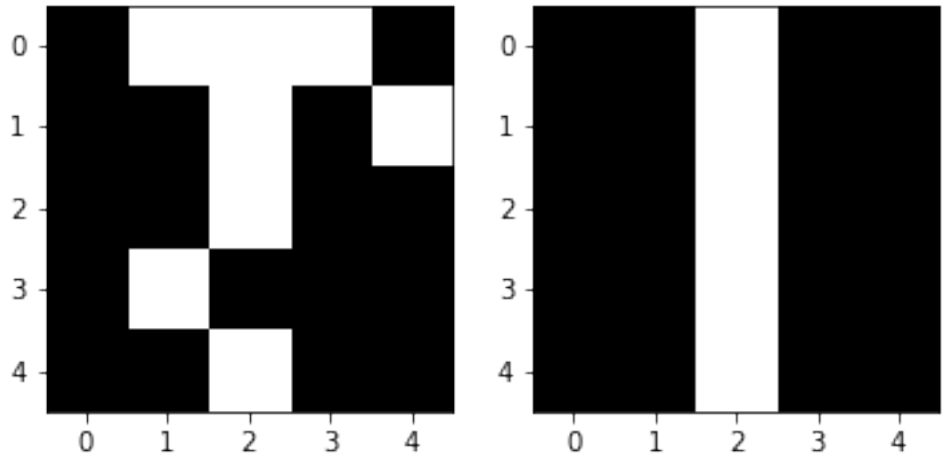
```
In [21]: def tryTest(sigma_test):
             sigma_test_resized = np.array(sigma_test).reshape(5, 5)

             V = sigma_test # this is the initial state (I guess it's the where the tested image
             new_V = np.ones(25)

             for i in range(25):
                 total = 0
                 for j in range(25):
                     total += s[i][j] * V[j]

                 # total signed total is just y
```

```
            if total >= 0:
                new_V[i] = 1
            else:
                new_V[i] = -1

        V = new_V
        # print(V)
        output = np.array(V).reshape(5, 5)

        f = plt.figure()
        f.add_subplot(121)
        plt.imshow(sigma_test_resized, cmap = cm.Greys_r)
        f.add_subplot(122)
        plt.imshow(output, cmap = cm.Greys_r)
        plt.show()
```

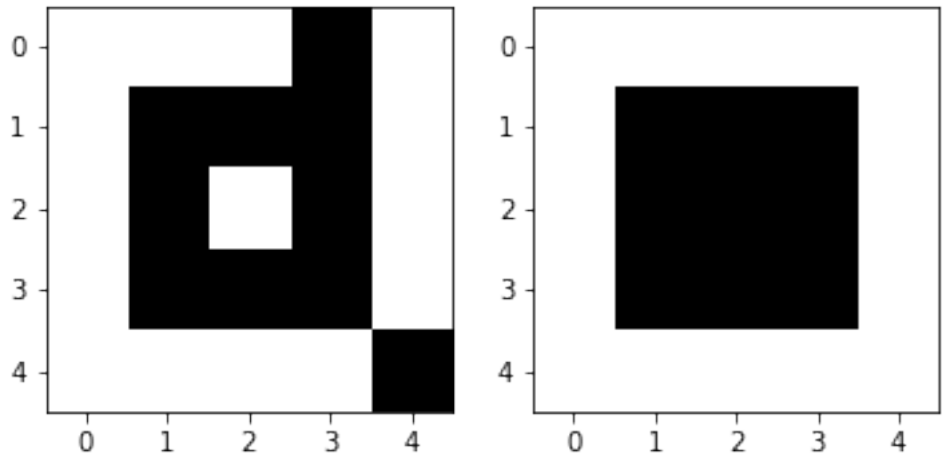
In [25]:
```
sigma_test = np.array([-1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, -1, -1, -1,
tryTest(sigma_test)
```



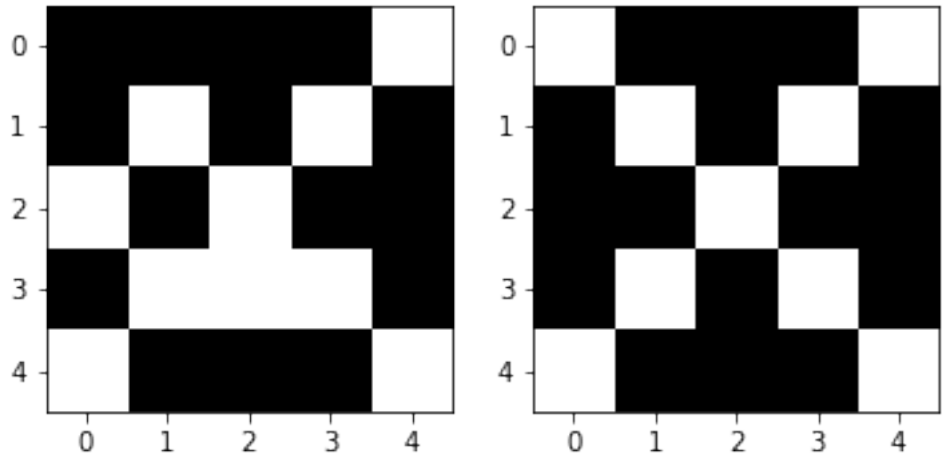
In [24]:
```
sigma_test2 = np.array([-1, 1, 1, 1, -1, -1, -1, 1, -1, 1, -1, -1, 1, -1, -1, -1, 1, -1
tryTest(sigma_test2)
```



In [27]:
```
sigma_test3 = np.array([1, 1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, 1, -1, 1, 1, -1, -1, -
tryTest(sigma_test3)
```

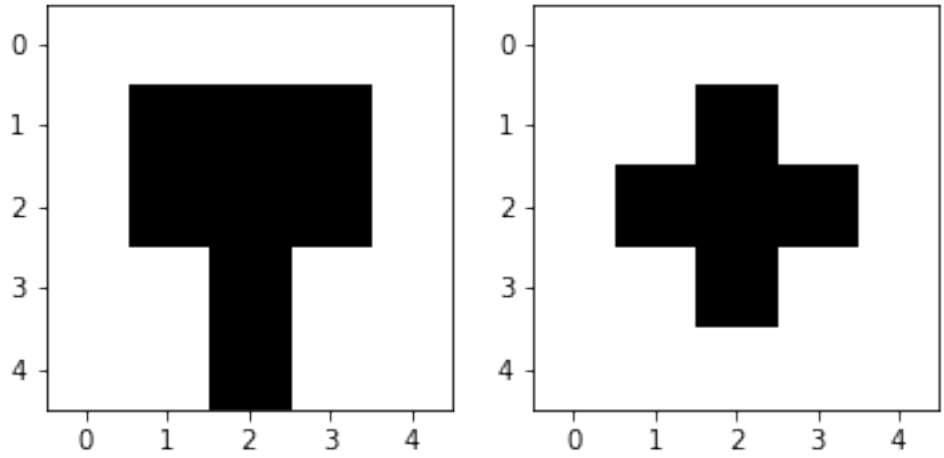


In [28]:
```
sigma_test4 = np.array([-1, -1, -1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, -1, -1, 1, 1
tryTest(sigma_test4)
```



The following is a spurious state:

```
In [68]: sigma_test5 = np.array([1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1, 1, 1, 1, -1, 1,
         tryTest(sigma_test5)
```



First-order Star CNNs (this is the same thing as the basic weight change rules above)

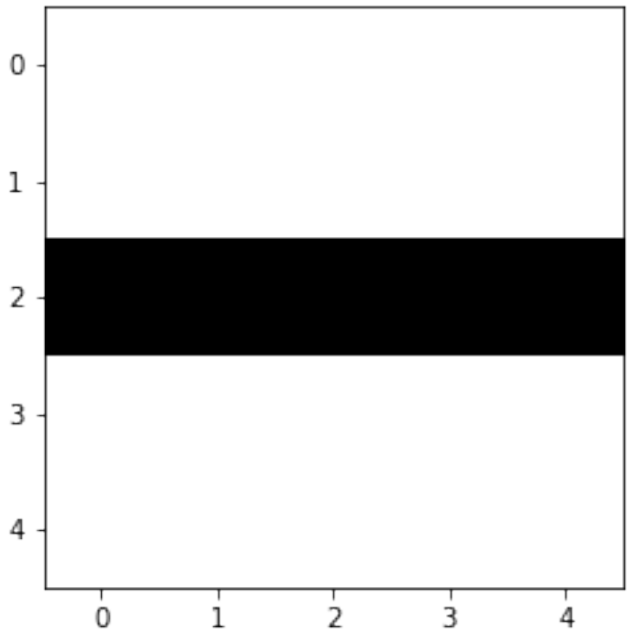
0.3 Second-order Star CNN

0.3.1 1) Two-cell CNNs [Zhou & Nosseck, 1991]

```
In [70]: # this converges on the first pattern
         V = sigma_test
         Y = np.ones(25)
```

```
new_V = np.ones(25)
new_Y = np.ones(25)

def h(x):
    return(1/2 * ((abs(x + 1)) - abs(x - 1)) )

def sgn(x):
    if x >= 0:
        return 1
    else:
        return -1

saved_x = []
saved_y = []
timestamp = 10
for t in range(timestamp):
#     print(V[0])
    saved_x.append(V[0])
    saved_y.append(Y[0])

    for i in range(25):
        p=1.1; q=1.1; r=-2; k=2

        dx = -V[i] + p*h(V[i]) + r*h(Y[i])
        dy = -Y[i] + k*h(V[i]) + q*h(Y[i])

        total = 0
        for j in range(25):
            total += s[i][j] * V[j]

        dx += sgn(total)

        new_V[i] = V[i] + dx
        new_Y[i] = Y[i] + dy

    V = new_V
    Y = new_Y
# print(V)
new_array=[sgn(i) for i in V]
new_array_resized = np.array(new_array).reshape(5, 5)
plt.imshow(new_array_resized, cmap = cm.Greys_r)
```

Out[70]: <matplotlib.image.AxesImage at 0x7f3ca3fd4940>




```
In [ ]: plt.plot([i + 1 for i in range(timestamp)], saved_x)
        plt.xlabel("t")
        plt.ylabel("x")
        plt.show()

        plt.plot([i + 1 for i in range(timestamp)], saved_y)
        plt.xlabel("t")
        plt.ylabel("y")
        plt.show()
```