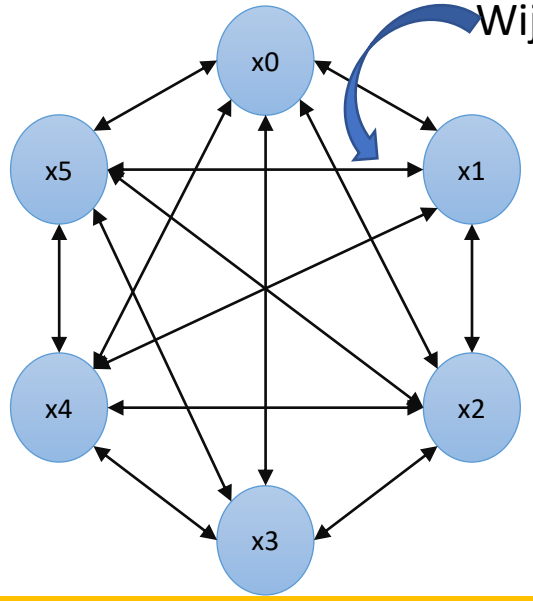
Hopfield Networks

-Abhilasha Dave

02/24/2023

Intro

- John Hopfield proposed a method for using neural network for Content addressable memory (CAM). The network learns the patterns and converges to the closest pattern when shown a partial pattern.
- CAM: It takes a part of a pattern and produce the most likely match from memory.



Wij is the connection strength from node i to node j

- Assumption: The weights on the both direction between two nodes are same

  Wij = Wji

$$x_j = \begin{cases} -1 & \text{if } \sum_{j \neq i} x_i W_{ij} < -b_j \\ 1 & \text{if } \sum_{j \neq i} x_i W_{ij} \geq -b_j \end{cases}$$

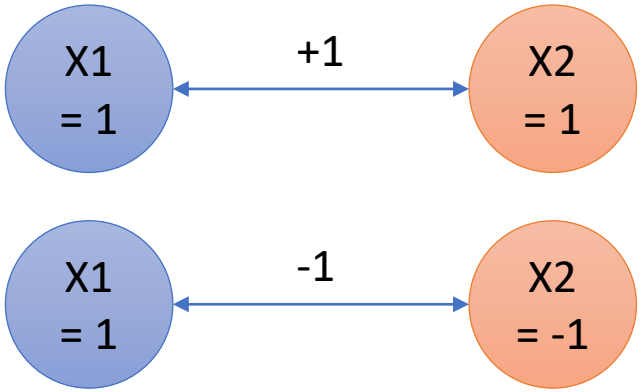
- Each node in the NW can be -1 or 1 (it can also be 1's and 0's)
- All the connections are in both direction. Like RNN
- Assumption: $b_j = 0$ (we can set different threshold here based on requirements)

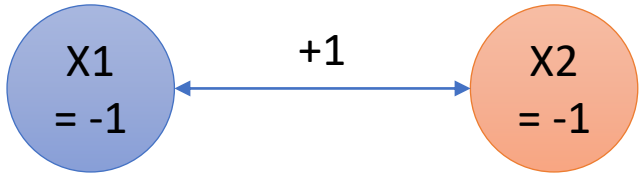
Update Rule

- How do we set the weights for the network to memorized a certain pattern?

$W_{ij} > 0$ We set the weights to +ve (+1) when the value of two nodes are same (i.e., stable)

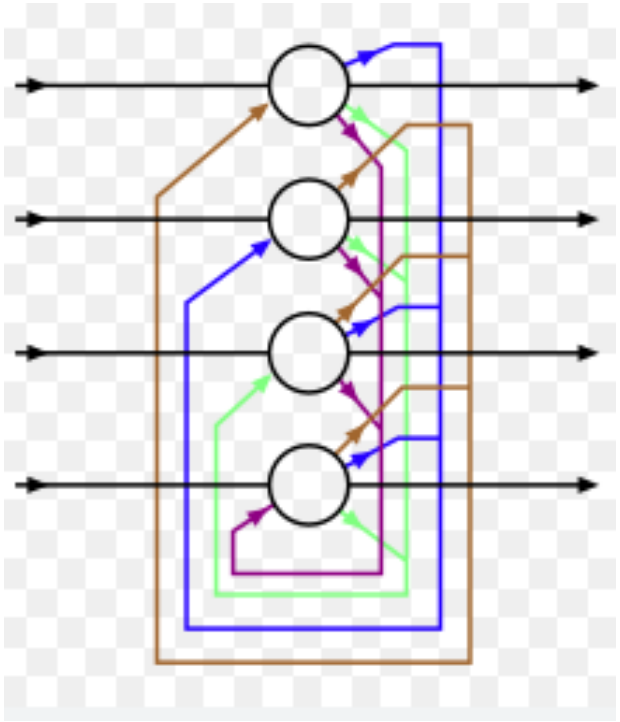
$W_{ij} < 0$ We set the weights to –ve (-1) when the value of two nodes are different (i.e., unstable)





Hopfield Neural Network Architecture

Inputs



Outputs

Example

- Let's say we have 3 patterns for Hopfield Network to learn
- X1 = $\begin{matrix}1\\-1\\-1\\-1\\-1\end{matrix}$ X2 = $\begin{matrix}-1\\1\\-1\\-1\\-1\end{matrix}$ X3 = $\begin{matrix}-1\\-1\\1\\-1\\-1\end{matrix}$
- Q: How many neurons Hopfield NW needs to detect above patterns?
- Ans: If we look at the individual pattern there are 5 activation values. Meaning 5 neurons are needed
- Q: How many synapsis/weights will be in our Hopfield NN model?
- Ans: (#of neurons * neurons itself) - #of neurons
  - (5*5) – 5 = 20

This subtraction is because neurons cannot get connected to itself

Simulate Learning in Hopfield Network

- We will start with initial weights (W0) matrix of 5x5 all 0's

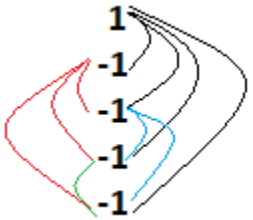
```
W0 = [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

- If we remember out pattern 1
- X1= [1, -1, -1, -1, -1]
- Based on Hebbian equation we will multiply the activation values and will get the new trained weights (W1):

$$T_{ij} = \sum_k v_i^k v_j^k$$

```
W1= [[ 0 -1 -1 -1 -1]
 [-1  0  1  1  1]
 [-1  1  0  1  1]
 [-1  1  1  0  1]
 [-1  1  1  1  0]]
```

Row	1	2	3	4	5
1	0	-1	-1	-1	-1
2	-1	0	1	1	1
3	-1	1	0	1	1
4	-1	1	1	0	1
5	-1	1	1	1	0



Simulate Learning in Hopfield Network

- Now we will get a second set of weights if we train the network with pattern X2
- X2= [-1, 1, -1, -1, -1]
- Based on Hebbian equation we will multiply the activation values and will get the new trained weights (W2):

```
W2= [[ 0 -1  1  1  1]
 [-1  0 -1  1  1]
 [ 1 -1  0  1  1]
 [ 1  1  1  0  1]
 [ 1  1  1  1  0]]
```

```
W1= [[ 0 -1 -1 -1 -1]
 [-1  0  1  1  1]
 [-1  1  0  1  1]
 [-1  1  1  0  1]
 [-1  1  1  1  0]]
```

For the addition reference W1

- If we add both the weights: (W1+W2) =

```
[[ 0 -2  0  0  0]
 [-2  0  0  2  2]
 [ 0  0  0  2  2]
 [ 0  2  2  0  2]
 [ 0  2  2  2  0]]
```

Simulate Learning in Hopfield Network

- Repeat the similar steps to get W3:

```
W3= [[ 0  1 -1  1  1]
 [ 1  0 -1  1  1]
 [-1 -1  0 -1 -1]
 [ 1  1 -1  0  1]
 [ 1  1 -1  1  0]]
```

```
W1= [[ 0 -1 -1 -1 -1]
 [-1  0  1  1  1]
 [-1  1  0  1  1]
 [-1  1  1  0  1]
 [-1  1  1  1  0]]
```

```
W2= [[ 0 -1  1  1  1]
 [-1  0 -1  1  1]
 [ 1 -1  0  1  1]
 [ 1  1  1  0  1]
 [ 1  1  1  1  0]]
```

For the addition reference W1 and W2

- After adding W1+W2+W3 we will get below updated weights for NN model:

```
[[ 0 -1 -1  1  1]
 [-1  0 -1  3  3]
 [-1 -1  0  1  1]
 [ 1  3  1  0  3]
 [ 1  3  1  3  0]]
```

Determine the stability of Network:

- To find the stability of network we will use the activation dynamics equitation.
- $\sin(\sum_{j=1}^{N} w_{ij}a_j - \theta)$

W1+W2+W3

- To find the stability of network we will use the activation dynamics equitation.
- $\sin(\sum_{j=1}^{N} w_{ij}a_j - \theta)$

- Here Total is 0 means the value of 1's activator will be +1
- Note: Same calculation can be done for all other node, and it shows the N/W is stable
- Comment: I think we can replace the attention mechanism of transformer

x1	1 *x	0	0
	-1 *w1,2	=-1 * -1	1
	-1 *w1,3	=-1 * -1	1
	-1 *w1,4	=-1 * 1	-1
	-1 *w1,5	=-1 * 1	-1

- If the total is >= 0 then assign the val +1
- Else -1

Matched Shows N/W is stable

Total= 0

$\theta = 0$

- Here Total is 0 means the value of 1's activator will be +1
- Note: Same calculation can be done for all other node, and it shows the N/W is stable
- Comment: I think we can replace the attention mechanism of transformer

Python Implementation

- Github Repo: https://github.com/slaclab/snl-Hopfield

Created small class to train and recall (test) the NW

```
class HopfieldNetwork:
    def __init__(self, size):
        self.size = size
        self.weights = np.zeros((size, size))
    def train(self, patterns):
        for pattern in patterns:
            pattern = np.reshape(pattern, (self.size, 1))
            self.weights += np.dot(pattern, pattern.T)
            np.fill_diagonal(self.weights, 1)
            #print('update weights:',self.weights)
    #print(self.weights)
def recall(self, pattern):
    pattern = np.reshape(pattern, (self.size, 1))
    while True:
        old_pattern = np.copy(pattern)
        pattern = np.sign(np.dot(self.weights, pattern))
        if np.array_equal(pattern, old_pattern):
            return pattern.flatten()
```

```
# Test the network by recalling each pattern and printing the result
for pattern in patterns:
    recalled_pattern = network.recall(pattern)
    print("Pattern: ", pattern, "Recalled pattern: ", recalled_pattern)
```

```
Pattern:  [ 1. -1.  1. -1. -1.] Recalled pattern:  [ 1. -1.  1. -1. -1.]
Pattern:  [ 1. -1. -1. -1.  1.] Recalled pattern:  [ 1. -1. -1. -1.  1.]
Pattern:  [-1. -1. -1. -1.  1.] Recalled pattern:  [-1. -1. -1. -1.  1.]
Pattern:  [-1.  1. -1.  1. -1.] Recalled pattern:  [-1.  1. -1.  1. -1.]
Pattern:  [-1. -1. -1.  1.  1.] Recalled pattern:  [-1. -1. -1.  1.  1.]
```

```
er_patterns = Perturb( patterns , p=0.2) #probability of each bit randomly fliped
print (er_patterns)
print(patterns)
```

```
[[ 1. -1.  1. -1. -1.]
 [ 1. -1. -1. -1.  1.]
 [-1. -1. -1. -1.  1.]
 [-1.  1. -1.  1. -1.]
 [-1. -1. -1.  1.  1.]]
[[ 1. -1.  1. -1. -1.]
 [ 1. -1. -1. -1.  1.]
 [-1. -1. -1. -1.  1.]
 [-1.  1. -1.  1. -1.]
 [-1. -1. -1.  1.  1.]]
```

- To test the network first I gave exactly same patterns to see if NW recalls it correctly
- After that added a random error with 0.2 probability.
- It recalled all the patterns correctly most of the time

```
# Random
N = 5 #number of neurons
n = 5 #number of patterns
patterns = Thresh(np.random.normal(size=(n,N)))
print(patterns)
```

Created a random 5x5 pattern

```
[[ 1. -1.  1. -1. -1.]
 [ 1. -1. -1. -1.  1.]
 [-1. -1. -1. -1.  1.]
 [-1.  1. -1.  1. -1.]
 [-1. -1. -1.  1.  1.]]
```

```
# Create a Hopfield network with 4 neurons
network = HopfieldNetwork(N)
```

Defined how may nodes Hopfield NW needs.

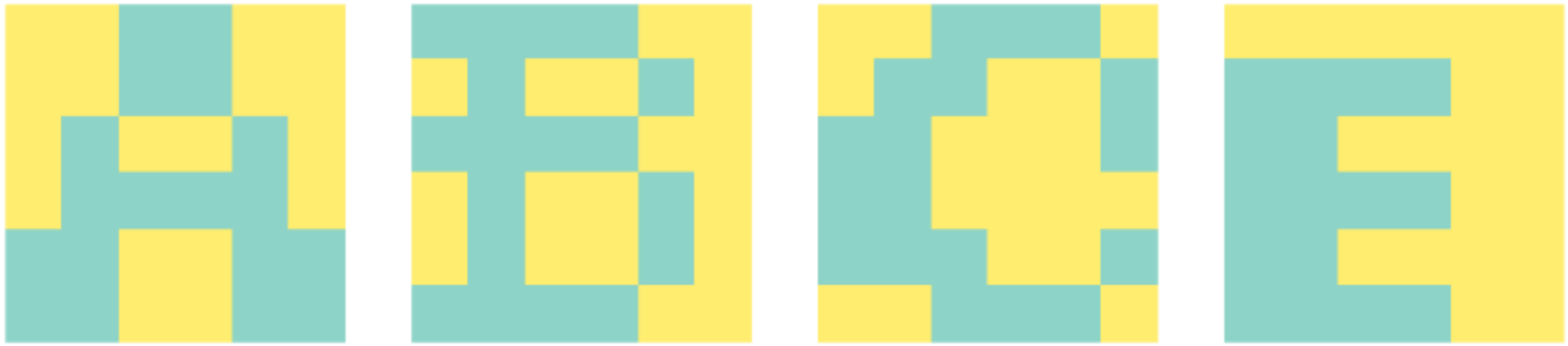
```
# Train the network with the patterns
network.train(patterns)
```

Trained the NW

```
update weights: [[ 1. -1.  1. -1. -1.]
 [-1.  1. -1.  1.  1.]
 [ 1. -1.  1. -1. -1.]
 [-1.  1. -1.  1.  1.]
 [-1.  1. -1.  1.  1.]]
update weights: [[ 1. -2.  0. -2.  0.]
 [-2.  1.  0.  2.  0.]
 [ 0.  0.  1.  0. -2.]
 [-2.  2.  0.  1.  0.]
 [ 0.  0. -2.  0.  1.]]
update weights: [[ 1. -1.  1. -1. -1.]
 [-1.  1.  1.  3. -1.]
 [ 1.  1.  1.  1. -3.]
 [-1.  3.  1.  1. -1.]
 [-1. -1. -3. -1.  1.]]
update weights: [[ 1. -2.  2. -2.  0.]
 [-2.  1.  0.  4. -2.]
 [ 2.  0.  1.  0. -2.]
 [-2.  4.  0.  1. -2.]
 [ 0. -2. -2. -2.  1.]]
update weights: [[ 1. -1.  3. -3. -1.]
 [-1.  1.  1.  3. -3.]
 [ 3.  1.  1. -1. -3.]
 [-3.  3. -1.  1. -1.]
 [-1. -3. -3. -1.  1.]]
```

Shows after training each pattern how the updated weighs looks like

Letters Example



```
Class  3
[[-1. -1.  1. -1. -1.  1. -1.  1.  1.  1. -1. -1.  1.  1. -1. -1. -1. -1.
   1.  1.  1. -1. -1.  1.  1.  1.  1. -1. -1. -1.  1.  1. -1. -1. -1. -1.]]
```


Distorted letter E

```
x[0,24:] = -1. #starting index 24 untill end off array fill with -1s
plt.imshow(np.reshape(x,[6,6]), cmap='tab20c_r'); plt.axis('off');
print(x)
```

```
[[-1. -1.  1. -1. -1.  1. -1.  1.  1.  1. -1. -1.  1.  1. -1. -1. -1. -1.
   1.  1.  1. -1. -1.  1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
```

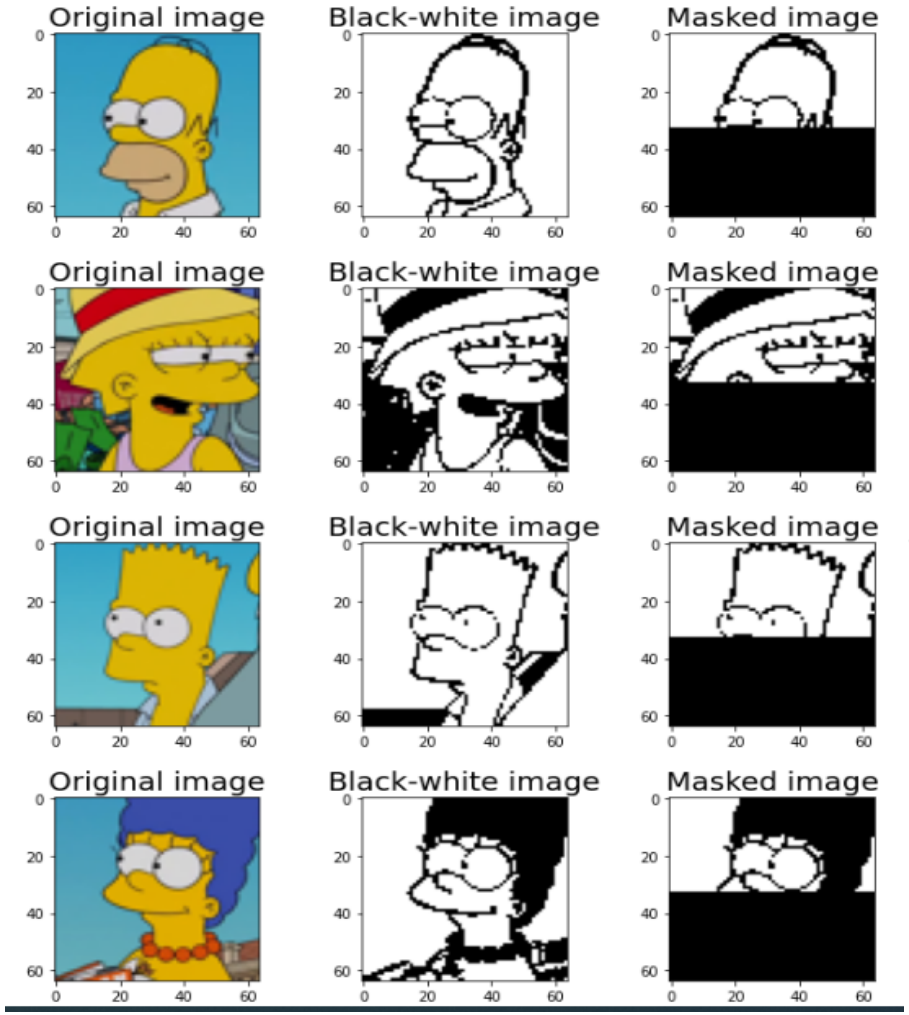
Further masked the lower bound of a distorted letter E

```
#Recall the test pattern
recalled_pattern = network.recall(x)
print(recalled_pattern)
plt.imshow(np.reshape(recalled_pattern,[6,6]), cmap='tab20c_r'); plt.axis('off');
```

```
[-1. -1. -1. -1. -1. -1.  1.  1.  1.  1. -1. -1.  1.  1. -1. -1. -1. -1.
  1.  1.  1.  1. -1. -1.  1.  1. -1. -1. -1. -1.  1.  1.  1.  1. -1. -1.]
```

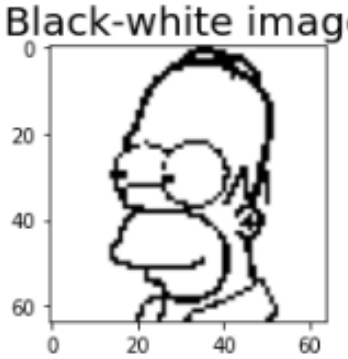
- Sent the Distorted Image to the Hopfield NW trained with Letters
- Recalled the letter E correctly

Hopfield NW with Simpsons Family!

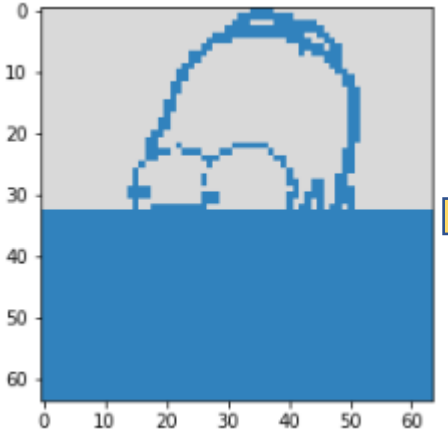


Got these Simpsons faces online

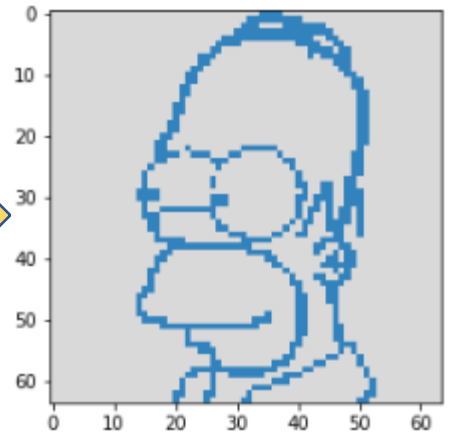
- So far tried only with Homer Jay Simpson's face
- Gave the black and white Homer image to Hopfield Network



- For recalling/testing gave the masked image to recall the face



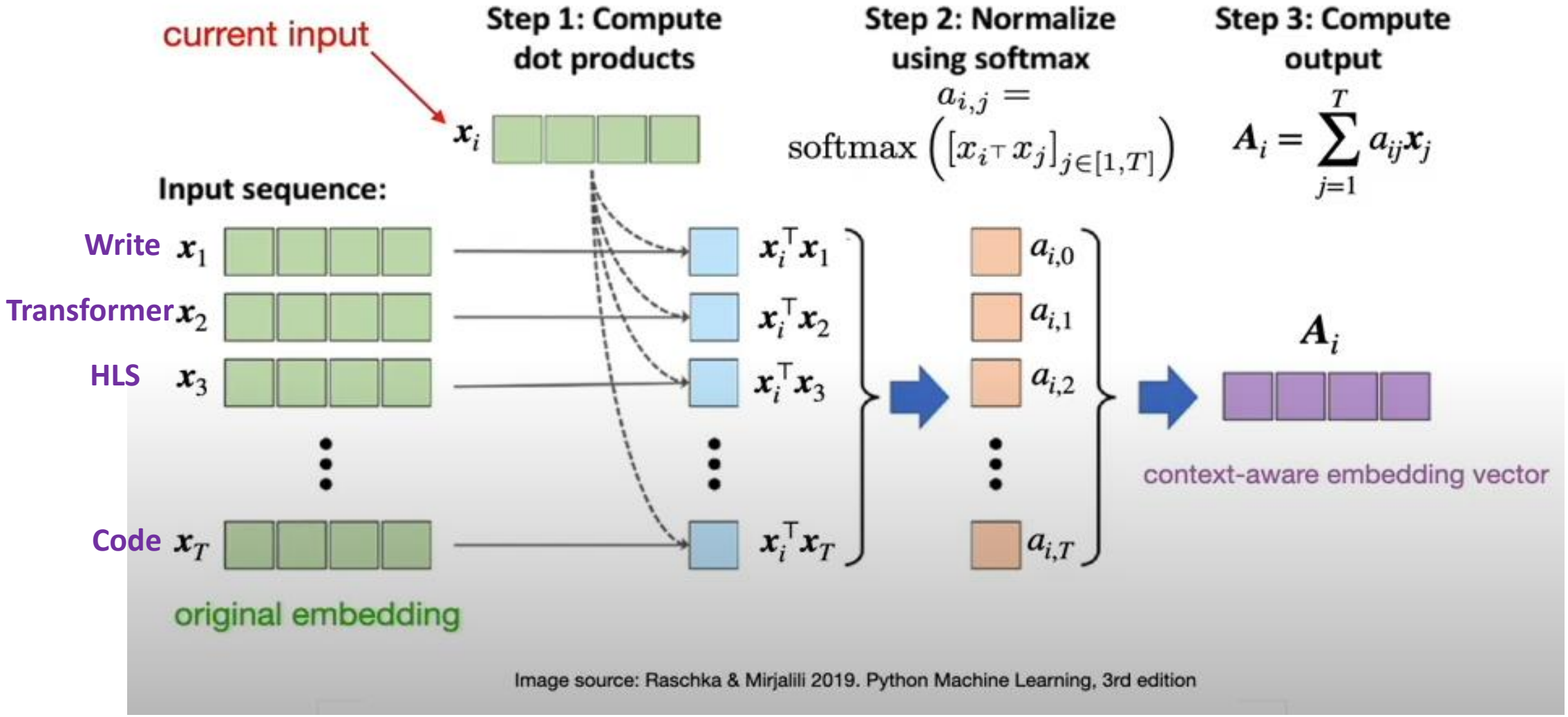
Masked Homer



Recalled Homer

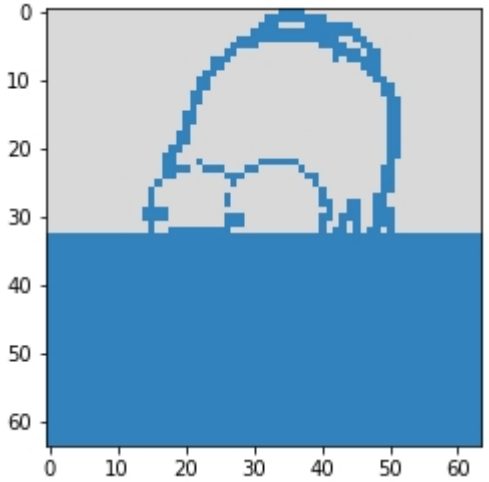
Q: If we give multiple family members faces to train will it be able to detect? Is Habbian learning enough?
If not, what are other flavors of Hopfield NW we can use?

Basic Self-Attention Mechanism in Transformer

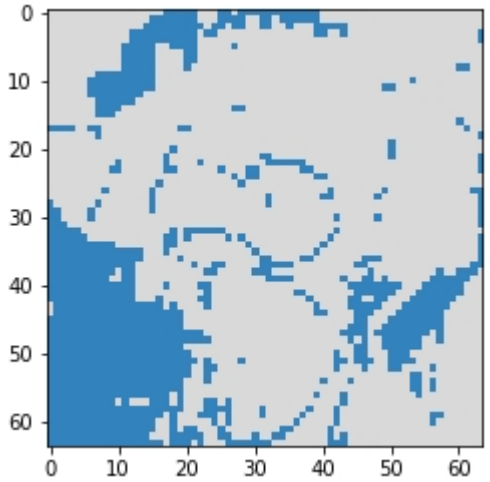


Hopfield NW with Simpsons Family!

Trained the network with above 5 images



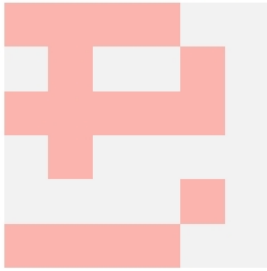
Masked Homer for Testing

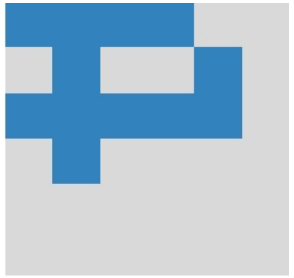


Model is not able to recall the Homers face! This is where traditional/old Hopfield fails when it got trained by multiple patterns

Modern Hopfield NW

- In this for the inference run we calculate the energy function
- When the energy is converged to a stable state (newly computed energy = old energy) we predict the output
- This energy function has a theta vals that we can play with to get the stable state of a network





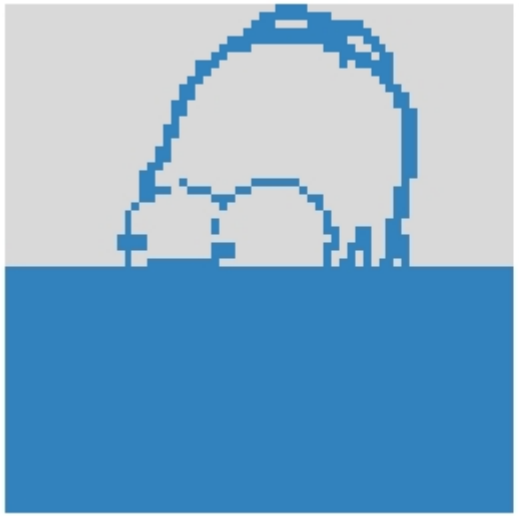
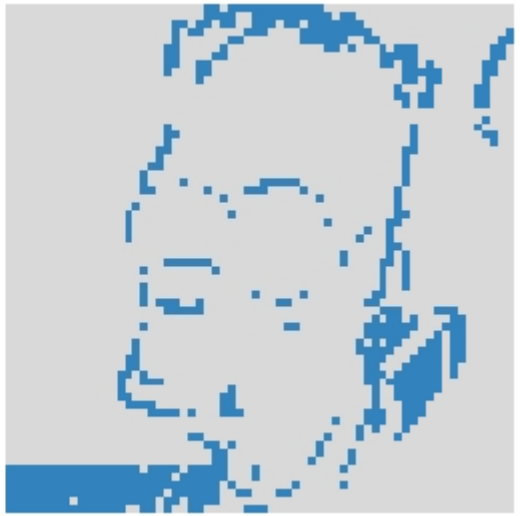
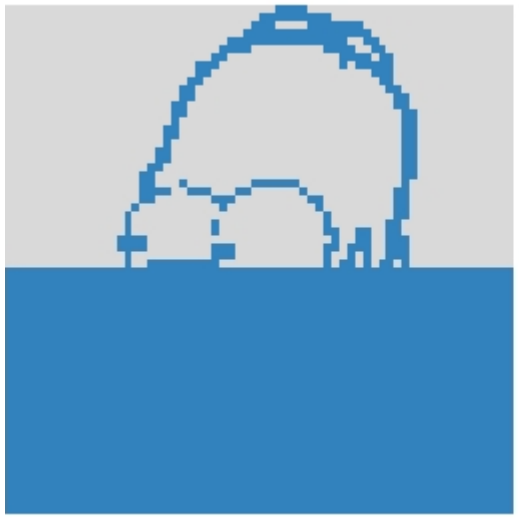
```
# Set the biases
theta = -np.sum(W, axis=1)

#Recall the test pattern
recalled_pattern =test(W, theta, x[0],100)
plt.imshow(np.reshape(recalled_pattern,[6,6]), cmap='tab20c_r'); plt.axis('off');
```

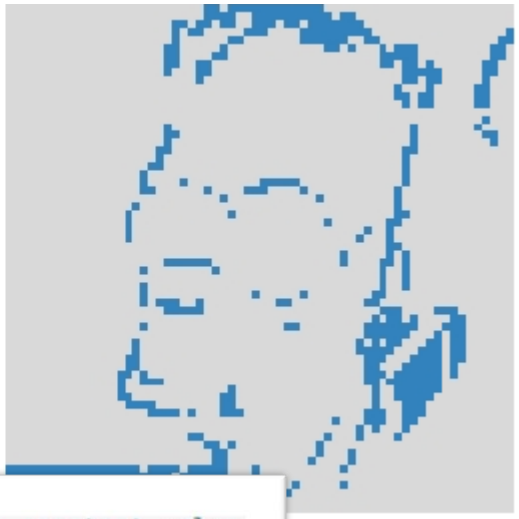
```
100
```

```
import numpy as np

# Define the energy function
def energy(W, s, theta):
    N = len(s)
    #e = -0.5 * np.sum(np.dot(s, W) * s) - np.dot(theta, s)
    e = -0.5 * np.sum(np.dot(s, W)) + np.dot(theta, s)
    return e

# Define the training function
def train(X):
    p, N = X.shape
    W = np.zeros((N, N))
    for mu in range(p):
        x = X[mu]
        W += np.outer(x, x)
    W /= N
    np.fill_diagonal(W, 0)
    return W

# Define the testing function
def test(W, theta, s0, max_iterations=500):
    s = np.copy(s0)
    for i in range(max_iterations):
        energy_old = energy(W, s, theta)
        for j in range(len(s)):
            s[j] = np.sign(np.dot(W[j], s) + theta[j])
        energy_new = energy(W, s, theta)
        if energy_new == energy_old:
            print(max_iterations)
            break
    return s
```

By setting theta to a constant value is working well for the alphabet recognition

Modern Hopfield NW with Simpsons Family!

```
# Set the biases to some -ve constant value
theta = -np.sum(W, axis=1)
```

```
# Compute the mean of the input patterns and set it as theta value
mean_pattern = np.mean(X, axis=0)
theta =  mean pattern
```

Ways to set Theta Vals

- **Set theta to a constant value**: This is the simplest way to set the bias term. For example, theta could be set to zero, or to a positive or negative value depending on the desired behavior of the network.
- **Set theta based on the mean of the input patterns**: In this approach, theta is set to the mean of the input patterns. This can help balance the overall activity of the network and prevent saturation of the neurons.
- **Set theta based on the mean and standard deviation of the input patterns**: This approach is similar to the previous one, but takes into account the variability of the input patterns. Theta is set to the mean plus a scaled value of the standard deviation.
- **Learn theta along with the weights**: In this approach, theta is treated as a trainable parameter along with the weights of the network. This can allow the network to adapt to different tasks and input distributions.

Fun Read: Quantum Hopfield

- https://journals.aps.org/pra/pdf/10.1103/PhysRevA.98.042308

PHYSICAL REVIEW A **98**, 042308 (2018)

Editors' Suggestion

Quantum Hopfield neural network

Patrick Rebentrost,[1,*] Thomas R. Bromley,[1,†] Christian Weedbrook,[1] and Seth Lloyd[2]

[1]*Xanadu, 372 Richmond Street West, Toronto, Ontario, Canada M5V 1X6*

[2]*Department of Mechanical Engineering, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, Massachusetts 02139, USA*

(Received 19 June 2018; published 5 October 2018)

Quantum computing allows for the potential of significant advancements in both the speed and the capacity of widely used machine learning techniques. Here we employ quantum algorithms for the Hopfield network, which can be used for pattern recognition, reconstruction, and optimization as a realization of a content-addressable memory system. We show that an exponentially large network can be stored in a polynomial number of quantum bits by encoding the network into the amplitudes of quantum states. By introducing a classical technique for operating the Hopfield network, we can leverage quantum algorithms to obtain a quantum computational complexity that is logarithmic in the dimension of the data. We also present an application of our method as a genetic sequence recognizer.

DOI: 10.1103/PhysRevA.98.042308