CS F433 Computational Neuroscience Assignment Report

Pratyush Bindal*, Kalash Bhattad*

***Abstract*- Hopfield Networks are associative memory models capable of storing and recalling information. In this study, we implement and evaluate a Hopfield Network trained on 16 x 16 binary images using the Hebbian Learning Rule. The network was tested under two distinct corruption methods - pixels randomly flipping with probability p and cropping images where only a central bounding box is preserved and in contrast, surrounding pixels are set to a uniform colour. We investigate the network's ability to recover original patterns through synchronous and asynchronous update rules, analysing convergence behaviour and quantifying the fraction of correctly recovered images, providing insights into the stability and recall capacity of the Hopfield Network.**

Keywords- Hopfield Networks, associative memory, Hebbian Learning Rule, synchronous updates, asynchronous updates

I. Dataset Preparation

We constructed the following dataset consisting of 25 16 X 16 pixels black and white images in the binary format. Furthermore, since PBM files consist of binary format (0/1), we converted them into bipolar representation (-1/1).

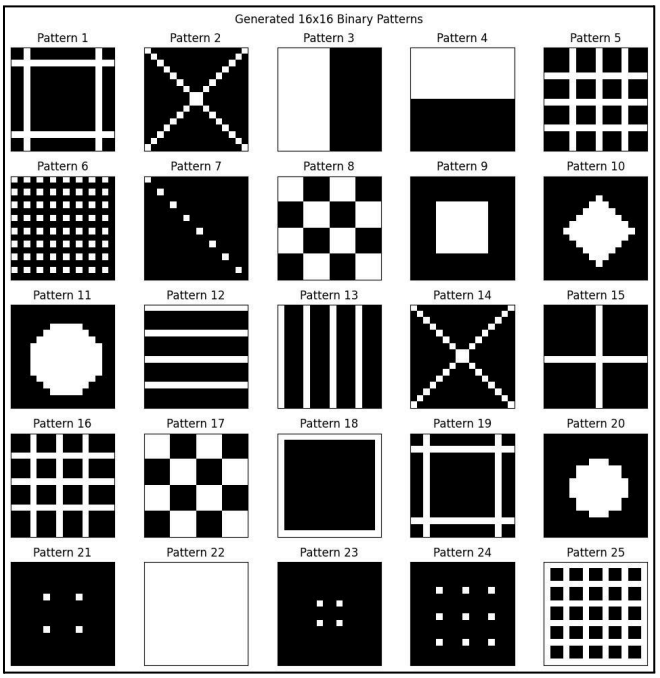

Fig 1. Generated 16 x 16 Binary Patterns

Stored patterns influence Hopfield Network's stability and recall performance. Highly correlated patterns introduce interference, which may lead to spurious attractors and reduce retrieval accuracy. Hence, we visualised the hamming distance and cosine similarity between the patterns to check for the level of correlation.

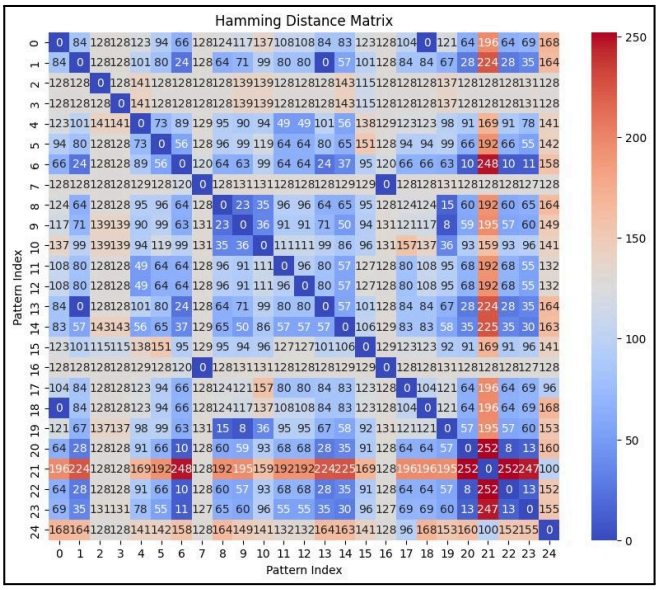

Fig 2. Hamming Distance Matrix

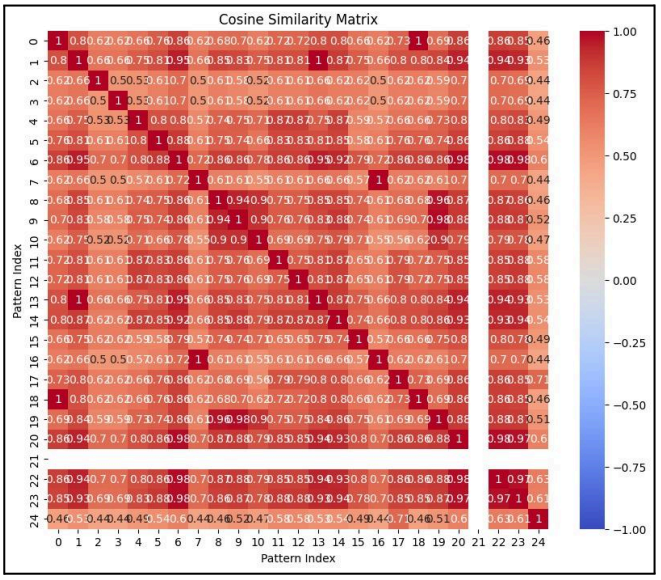

Fig 3. Cosine Similarity Matrix

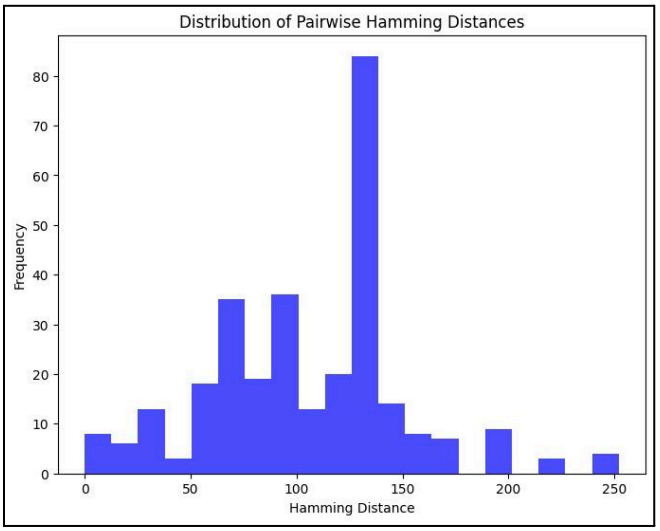

Fig 4. Distribution of pairwise Hamming Distances (Number of differing bits vs. Count of occurrences of each distance)

Furthermore, a fully connected Hopfield Network using the Hebbian Learning Rule with N neurons can theoretically store $N/(2\ln N)$ patterns before retrieval performance degrades due to spurious attractors and retrieval errors[1][2]. For a 16×16 binary image, the network consists of 256 neurons, which means the theoretical storage limit is around 16 patterns. For practical purposes, the limit further degrades due to the correlation between images, hence, we trained the built Hopfield network for 10 images out of 25 images in the dataset.

II. Implementation and Simulation Overview

A. Hopfield Network Training

We trained the Hopfield Network using the Hebbian Learning Rule, which is local and incremental and updates the weight matrix W as follows: $W = \frac{1}{N}\Sigma\varepsilon_i^u\varepsilon_j^u$ where ε_i^u represents bit i from pattern u, and N is the number of training images. The diagonal elements of W were set to zero to avoid self-feedback.

B. Pattern Corruption

We used two corruption methods to test the network:

1. **Pixel Flipping:** Each pixel was flipped with a probability p where p ∈ {0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}, achieved by randomly inverting pixel values in the images.
2. **Cropping:** A 10 X 10 bounding box was retained, while the outer pixels were set to black (-1) or white (+1).

C. Update Mechanisms and Convergence

We implemented two update methods on the network:

1. **Synchronous Update**: All neurons were updated simultaneously.
2. **Asynchronous Update**: We updated neurons one at a time in a random order, using the weighted sum of their connections to decide their new state.

The network was updated until convergence, that is, there are no further changes in state. Furthermore, intermediate states were saved as PBM files at the midpoint of the update process to analyse the modification of the Hopfield network.

D. Experimental Analysis

Experiments were conducted for p ∈ {0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}, with 20 trials per p value as stated in the problem statement. We evaluated the results obtained based on two performance metrics:

- **Average convergence steps**: The number of updates required to reach a stable state.
- **Fraction of correctly recovered patterns**: The fraction of runs where the final state matched the original stored pattern.

III. Results

We analysed the performance of the built network on both synchronous and asynchronous updates by visualising average convergence steps reached across varying image bit corruption probabilities.

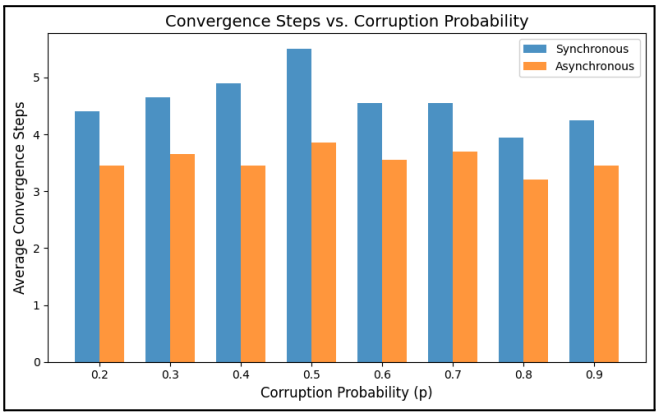

Fig 5. Convergence Steps vs. Corruption Probability

Furthermore, we analysed the recall performance by analysing the fraction of correctly retrieved patterns across varying image bit corruption probabilities for both synchronous and asynchronous updates.

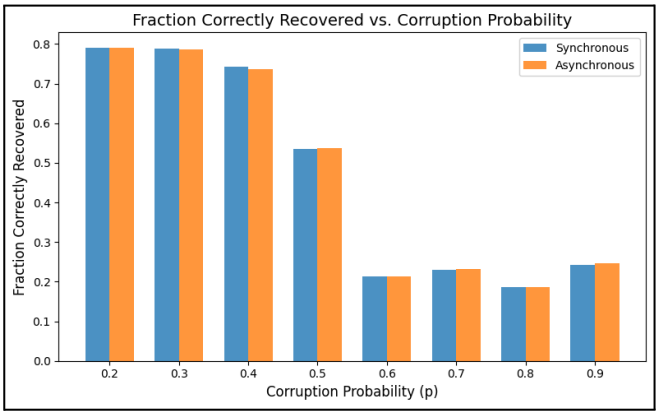

Fig 6. Fraction of correctly recovered images vs. Corruption probability

Also, we analysed the convergence behaviour of the Hopfield network by plotting histograms of update step distributions and recovery accuracy across varying corruption levels.

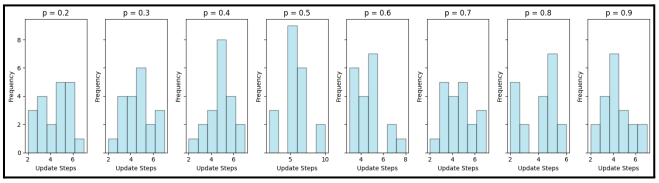

Fig 7. Required Update steps against frequency for each testing probability

Next, we demonstrate the sequential evolution of states via the Hopfield network through the following image on various scenarios possible for listed pattern corruption techniques and update mechanisms.

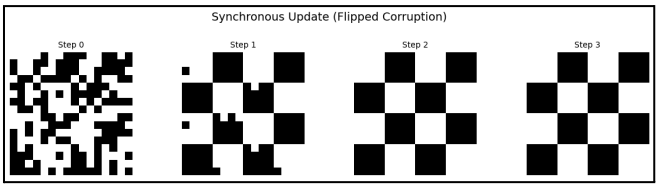

Fig 8. Synchronous Update on bit flipped corrupted image

The original image was corrupted by flipping the bits randomly with probability p = 0.3. The synchronous update required 3 steps to reach the convergence. We were able to fully recover the image in this case.

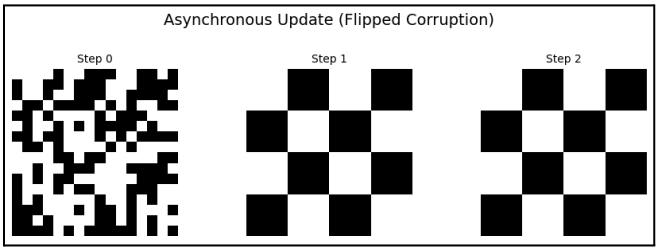

Fig 9. Asynchronous Update on bit flipped corrupted image

The original image was corrupted by flipping the bits randomly with probability p = 0.3. The asynchronous update also required 2 steps to reach the convergence for the same image. We were able to fully recover the image in this case.

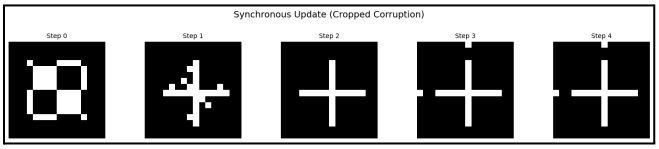

Fig 10. Synchronous Update on the cropped image with bounding box

The original image was cropped by a 10 X 10 bounding box where we kept part of the image inside the bounding box to be identical to the original image while changing every pixel outside to black. The synchronous update rule was applied which required 4 steps to reach the convergence, however, the image was not fully recovered in this case, which might be because of the rapid global changes in the pattern due to the synchronous update of pixels or significant information loss from the cropped pattern.

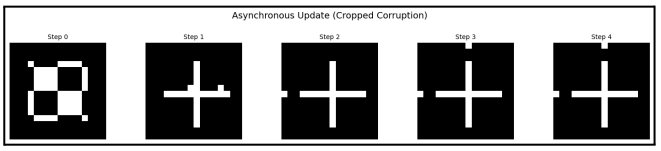

Fig 11. Asynchronous Update on the cropped image with bounding box

The original image was cropped by a 10 X 10 bounding box where we kept part of the image inside the bounding box to be identical to the original image while changing every pixel outside to black. The asynchronous update rule was applied which required 4 steps to reach the convergence for the same image, however, the image was not fully recovered in this case, which could be attributed to the loss of key distinguishable features present in the bounding box or the bias of the network towards a spurious state.

IV. References

1. McEliece, R., Posner, E., Rodemich, E., & Venkatesh, S. (1987). The capacity of the Hopfield associative memory. *IEEE Transactions on Information Theory*, *33*(4), 461–482. https://doi.org/10.1109/tit.1987.1057328

2. Storkey, A. (1997). Increasing the capacity of a hopfield network without sacrificing functionality. In *Lecture notes in computer science* (pp. 451–456). https://doi.org/10.1007/bfb0020196

APPENDIX

```
import numpy as np
import glob
import matplotlib.pyplot as plt
def readfile(filename):
    with open(filename,'r') as f:
        lines=f.readlines()
    lines=[l.strip() for l in lines if not l.startswith('#')]
    assert lines[0] =='P1'
    width, height=map(int, lines[1].split())
    data=np.array([int(b) for line in lines[2:] for b in line.split()])
    return data.reshape((height, width))
def conv(mem):
    return 2*mem-1
# PROBLEM STATEMENT 1 Hopfield Network Training using Hebbian Rule
def trainingnet(memories):
    n=memories[0].size
    W=np.zeros((n, n))
    for mem in memories:
        vec=mem.flatten()
        W+= np.outer(vec, vec)     #Hebbian Learning Rule
    W/=len(memories)
    np.fill_diagonal(W,0)
    return W
def bitflip(mem,p):     #Bit flip corruption
    corrupt =np.random.rand(*mem.shape) < p
    corrupted_mem=np.copy(mem)
    corrupted_mem[corrupt] *= -1
    return corrupted_mem
def cropping(mem, bbox_color):     #Bounding box cropping
    cropped = np.full(mem.shape, -1 if bbox_color == "black" else 1)
    cropped[3:13, 3:13] = mem[3:13, 3:13]
    return cropped
def syncupdate(state,W):    #Synchronous Update
    h, w=state.shape
    flat=state.flatten()
    updated= np.sign(W @ flat)
    updated[updated==0] = 1
    return updated.reshape(h, w)
def asyncupdate(state,W):    #Asynchronous Update
    h, w=state.shape
    flat=state.flatten()
    indices=np.arange(flat.size)
    np.random.shuffle(indices)       #Random Shuffle
    for i in indices:
        flat[i]=1 if np.dot(W[i],flat) >= 0 else -1
    return flat.reshape(h, w)
def runconv(initial, W,updating):    #Run until we reach convergence
    prev_state=None
    state=initial
    steps=0
    hist =[initial.copy()]
```

```
    while not np.array_equal(state,prev_state):
        prev_state=state.copy()
        state =updating(state, W)
        steps+= 1
        hist.append(state.copy())
    return state, steps, hist
# UTILITY FUNCTIONS FOR PLOTTING THE CURVES
def plotseq(hist,title):  #Sequence Plotting
    n=len(hist)
    fig, axes=plt.subplots(1,n,figsize=(n * 3, 3),squeeze=False)
    for i, state in enumerate(hist):
        ax = axes[0, i]
        ax.imshow(state,cmap='gray',interpolation='nearest')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f"Step {i}", fontsize=10)
        for spine in ax.spines.values():
            spine.set_visible(False)
    fig.suptitle(title,fontsize=14)
    plt.tight_layout(rect=[0, 0, 1, 0.93])
    plt.show()
def plothist(stepcnt,pval):  #Histogram Plotting
    fig,axes=plt.subplots(1,len(pval),figsize=(15, 4),sharey=True)
    for ax, p in zip(axes,pval):
        ax.hist(stepcnt[p],bins=6,alpha=0.5,edgecolor='black',color='skyblue')
        ax.set_title(f'p = {p}')
        ax.set_xlabel("Update Steps")
        ax.set_ylabel("Frequency")
    plt.tight_layout()
    plt.show()
def fillmemory(files):
    memories=[conv(readfile(f)) for f in files[:10]]
    return memories
def plotconvsteps(pval,syncsteps,asyncsteps):  #Convergence Steps Plotting
    plt.figure(figsize=(8,5))
    w=0.35
    indices = np.arange(len(pval))
    plt.bar(indices,syncsteps,w,label='Synchronous',alpha=0.8)
    plt.bar(indices + w,asyncsteps,w,label='Asynchronous',alpha=0.8)
    plt.xlabel("Corruption Probability (p)",fontsize=12)
    plt.ylabel("Average Convergence Steps",fontsize=12)
    plt.xticks(indices + w/2,[f"{p:.1f}" for p in pval],fontsize=10)
    plt.legend(fontsize=10)
    plt.title("Convergence Steps vs. Corruption Probability",fontsize=14)
    plt.tight_layout()
    plt.show()
def plotcorrect(pval,corrsync,corrasync):  #Correctly Recovered Plotting
    plt.figure(figsize=(8,5))
    w=0.35
    indices = np.arange(len(pval))
    plt.bar(indices, corrsync,w,label='Synchronous',alpha=0.8)
    plt.bar(indices + w, corrasync,w,label='Asynchronous',alpha=0.8)
    plt.xlabel("Corruption Probability (p)",fontsize=12)
    plt.ylabel("Fraction Correctly Recovered",fontsize=12)
```

```
    plt.xticks(indices + w/2,[f"{p:.1f}" for p in pval],fontsize=10)
    plt.legend(fontsize=10)
    plt.title("Fraction Correctly Recovered vs. Corruption Probability",fontsize=14)
    plt.tight_layout()
    plt.show()
# Saving the state of the Hopfield Network onto a PBM file midway during the updates as in PROBLEM STATEMENT 1
def savepbm(state,filename):
    # Convert from -1/1 to 0/1 for image reconstruction
    state_01=((state+1)/2).astype(int)
    h, w=state_01.shape
    with open(filename,'w') as f:
        f.write('P1\n')
        f.write(f'{w} {h}\n')
        for row in state_01:
            f.write(' '.join(str(val) for val in row)+ '\n')
# DRIVER CODE
def main():
    files=sorted(glob.glob('pattern_*.pbm'))
    memories=fillmemory(files)
    W=trainingnet(memories)

    # PROBLEM STATEMENT 2
    p=0.3
    memidx = np.random.randint(len(memories))
    orig=memories[memidx]
    bitflippy=bitflip(orig, p)
    croppy=cropping(orig, "black")

    finsyncflippy, syncstepflippy, histsyncflip = runconv(bitflippy, W, syncupdate)    #Bit flipped with synchronous update
    finasyncflippy, asyncstepflippy, histasyncflip = runconv(bitflippy, W, asyncupdate) #Bit flipped with asynchronous update
    finsynccroppy, syncstepcroppy, histsynccrop = runconv(croppy, W, syncupdate)   #Cropped with synchronous update
    finasynccroppy, asyncstepcroppy, histasynccrop = runconv(croppy, W, asyncupdate) #Cropped with asynchronous update

    print(f"Flipped corruption (p={p}): Synchronous steps = {syncstepflippy}, Asynchronous steps = {asyncstepflippy}")
    print(f"Cropped corruption: Synchronous steps = {syncstepcroppy}, Asynchronous steps = {asyncstepcroppy}")

    #Visualisations
    plotseq(histsyncflip, "Synchronous Update (Flipped Corruption)")
    plotseq(histasyncflip, "Asynchronous Update (Flipped Corruption)")
    plotseq(histsynccrop, "Synchronous Update (Cropped Corruption)")
    plotseq(histasynccrop, "Asynchronous Update (Cropped Corruption)")

    #Saving Midway updates to PBM files
    if histsyncflip:
        midpoint = len(histsyncflip)//2
        savepbm(histsyncflip[midpoint], 'midway_sync_flip.pbm')
        print("Saved midway synchronous flip state to 'midway_sync_flip.pbm'")
    if histasyncflip:
        midpoint = len(histasyncflip)//2
        savepbm(histasyncflip[midpoint], 'midway_async_flip.pbm')
        print("Saved midway asynchronous flip state to 'midway_async_flip.pbm'")
    if histsynccrop:
        midpoint = len(histsynccrop)//2
        savepbm(histsynccrop[midpoint], 'midway_sync_crop.pbm')
```

```
        print("Saved midway synchronous crop state to 'midway_sync_crop.pbm'")
    if histasynccrop:
        midpoint = len(histasynccrop)//2
        savepbm(histasynccrop[midpoint], 'midway_async_crop.pbm')
        print("Saved midway asynchronous crop state to 'midway_async_crop.pbm'")
    # PROBLEM STATEMENT 3
    pval=[0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
    paircntp=20
    syncstepcnt={p: [] for p in pval}
    asyncstepcnt={p: [] for p in pval}
    corrsync={p: [] for p in pval}
    corrasync={p: [] for p in pval}
    for p in pval:
      for _ in range(paircntp):
        memidx=np.random.randint(len(memories))
        orig=memories[memidx]
        corrupted=bitflip(orig, p)
        finsync, stepsync, _= runconv(corrupted, W, syncupdate)
        finasync, stepasync, _= runconv(corrupted, W, asyncupdate)
        syncstepcnt[p].append(stepsync)
        asyncstepcnt[p].append(stepasync)
        corrsync[p].append(np.mean(finsync==orig))
        corrasync[p].append(np.mean(finasync==orig))
    avgsyncsteps=[np.mean(syncstepcnt[p]) for p in pval]
    avgasyncsteps=[np.mean(asyncstepcnt[p]) for p in pval]
    avgcorrsync=[np.mean(corrsync[p]) for p in pval]
    avgcorrasync=[np.mean(corrasync[p]) for p in pval]
    plotconvsteps(pval,avgsyncsteps,avgasyncsteps)
    plotcorrect(pval,avgcorrsync,avgcorrasync)
    plothist(syncstepcnt,pval)
if __name__=="__main__":
    main()
```