Homework1: Recognizing Digits

Hai Dinh (19960331-4494)

1. Code

```
import argparse
import numpy as np
import matplotlib.pyplot as plt

digits = [
    np.array([[-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], [-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],
              [-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], [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]]),

    np.array([[-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], [-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],
              [-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], [-1,-1,-1,+1,+1,+1,+1,-1,-1,-1]]),

    np.array([[+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], [-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],
              [+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], [+1,+1,+1,+1,+1,+1,+1,+1,-1,-1]]),

    np.array([[-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], [-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],
              [-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], [-1,-1,+1,+1,+1,+1,+1,+1,-1,-1]]),
```

```
    np.array([[-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], [-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],
              [-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], [-1,-1,-1,-1,-1,-1,-1,+1,+1,-1]]),
]

inputs = [
    np.array([[+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], [+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],
              [-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], [-1,-1,-1,-1,-1,-1,-1,+1,+1,-1]]),

    np.array([[+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], [-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],
              [-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], [+1,+1,+1,-1,-1,-1,-1,+1,+1,+1]]),

    np.array([[+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], [+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],
              [-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], [-1,-1,-1,+1,+1,+1,+1,-1,-1,-1]]),
]

def signum(value):
    """
    Compute signum for a float value.
    """
    return -1 if value < 0 else 1

def compute_weight_matrix(stored_patterns, row=None, zerodiag=False):
    """
    Create weight matrix for Hopfield network using Hebb's rule.

    Args:
        stored_patterns (ndarray): Tensor of shape `(p, N)` containing `p` stored patterns, each
            represented by `N` neurons/bits.

        row (int): Create weight matrix only for this row, such that the return value will have the
```

```python
            shape of `(1, N)` (where N is the number of bits in each stored pattern). If no row is
            given, then the entire weight matrix of shape `(N, N)` will be computed.

        zerodiag (bool): If true, the weights along the diagonal of the weight matrix will be zeroed
            out. Otherwise, the diagonal weights will be computed using the normal Hebb's rule.

    Returns:
        The weight matrix of shape `(1, N)` if row is given. Otherwise, the entire matrix of shape
        `(N, N)` will be returned.
    """
    N = stored_patterns.shape[1]
    W = stored_patterns if row is None else stored_patterns[:, row, None]
    W = (W.T @ stored_patterns) / N

    if zerodiag:
        if row is None:
            np.fill_diagonal(W, 0)
        else:
            W[:,row] = 0
    return W

def run_hopfield_network(input_pattern, weight_matrix):
    """
    Run the Hopfield network by asynchronously updating all neurons until convergence.
    The asynchronous updates are done using the typewriter scheme.

    Args:
        input_pattern (ndarray): Tensor of shape `(w, h)` representing the input pattern.

        weight_matrix (ndarray): Tensor of shape `(N, N)` representing weight matrix of the network,
            where `N = w * h` is the number of neurons/bits of the network.

    Returns:
        Tensor of shape `(w, h)` representing the final pattern produced by the network.
    """
    pattern = input_pattern.flatten()
    N = len(pattern)
    total_updates = 0

    while True:
        converged = True
        total_updates += N

        for i in range(N):
            updated_neuron = signum(np.dot(weight_matrix[i], pattern))
            converged &= (updated_neuron == pattern[i])
            pattern[i] = updated_neuron

        if converged:
            break

    pattern = pattern.reshape(input_pattern.shape)
    recognized_digit = "Unknown"
```

```
    for (i, digit) in enumerate(digits):
        if np.array_equal(pattern, digit):
            recognized_digit = str(i)
        elif np.array_equal(pattern, -digit):
            recognized_digit = "Inverted " + str(i)

    print("Number of asynchronous updates:", total_updates)
    print("Recognized digit:", recognized_digit)
    print(pattern, "\n")
    return pattern

def main(args):
    stored_patterns = np.array([digit.flatten() for digit in digits])
    W = compute_weight_matrix(stored_patterns, zerodiag=True)
    outputs = [run_hopfield_network(P, W) for P in inputs]

    n_rows, n_cols = len(inputs), 2
    fig = plt.figure(figsize=(3 * n_cols, 4 * n_rows))
    plt.title("INPUTS (left) ==> OUTPUT (right)")
    plt.axis("off")

    for i in range(n_rows):
        fig.add_subplot(n_rows, n_cols, i * n_cols + 1)
        plt.imshow(inputs[i], cmap="gray_r")
        fig.add_subplot(n_rows, n_cols, i * n_cols + 2)
        plt.imshow(outputs[i], cmap="gray_r")

    plt.savefig(args.outdir + "/RecognizingDigits.png")
    plt.show()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Recognizing distorted digits using Hopfield network"
    )
    parser.add_argument(
        "--outdir",
        "-o",
        type=str,
        default=".",
        help="Out directory to output the image",
    )
    main(parser.parse_args())
```

2. Results

Since there're $16 \times 10 = 160$ neurons/bits, the results below show that convergence occurs after:

- 480 asynchronous updates $\equiv$ 3 synchronous updates for 1st pattern, converging to digit 3.
- 320 asynchronous updates $\equiv$ 2 synchronous updates for 2nd pattern, converging to digit 1.
- 320 asynchronous updates $\equiv$ 2 synchronous updates for 3rd pattern, converging to inverted digit 2.

Printouts:

```
$ python3 RecognizingDigits.py
```

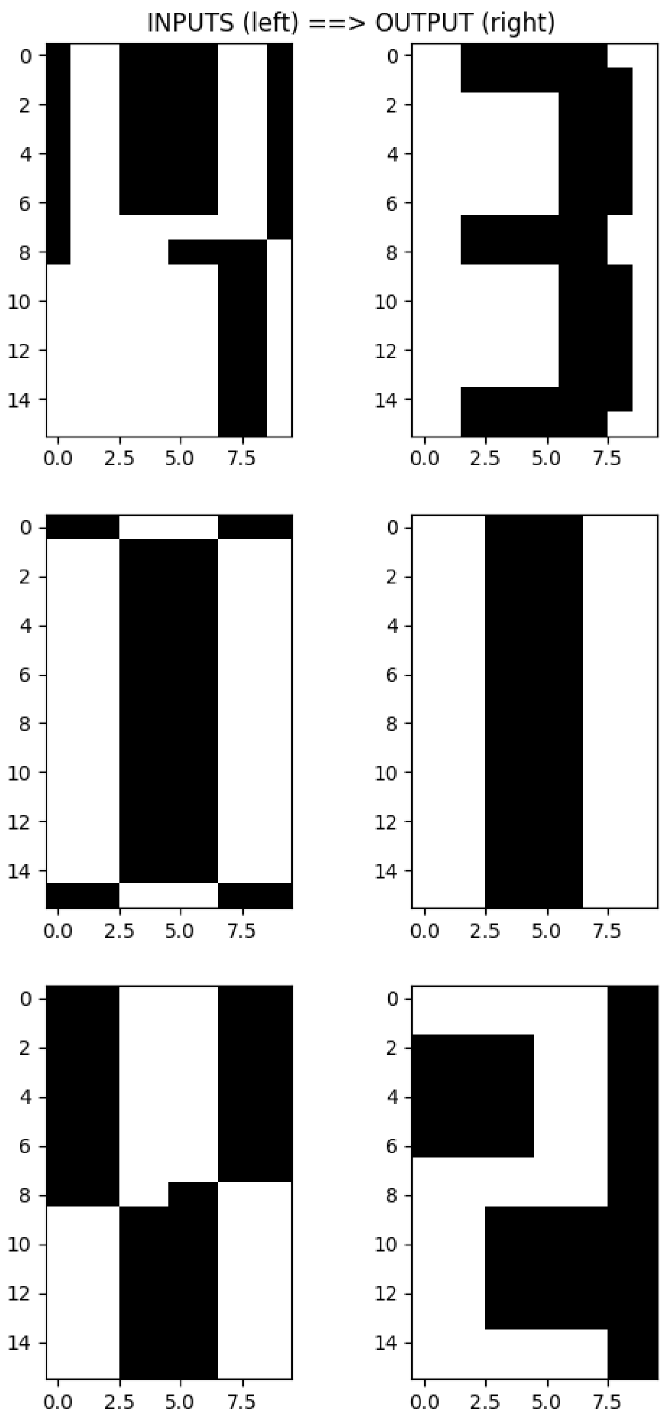
```
Number of asynchronous updates: 480
Recognized digit: 3
[[-1 -1  1  1  1  1  1  1 -1 -1]
 [-1 -1  1  1  1  1  1  1  1 -1]
 [-1 -1 -1 -1 -1 -1  1  1  1 -1]
 [-1 -1 -1 -1 -1 -1  1  1  1 -1]
 [-1 -1 -1 -1 -1 -1  1  1  1 -1]
 [-1 -1 -1 -1 -1 -1  1  1  1 -1]
 [-1 -1 -1 -1 -1 -1  1  1  1 -1]
 [-1 -1  1  1  1  1  1  1 -1 -1]
 [-1 -1  1  1  1  1  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  1  1  1 -1]
 [-1 -1  1  1  1  1  1  1  1 -1]
 [-1 -1  1  1  1  1  1  1 -1 -1]]

Number of asynchronous updates: 320
Recognized digit: 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 -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 -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 -1]
 [-1 -1 -1  1  1  1  1 -1 -1 -1]]

Number of asynchronous updates: 320
Recognized digit: Inverted 2
[[-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]
 [ 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]
 [-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]
[-1 -1 -1 -1 -1 -1 -1 -1  1  1]]
```

Image of the results is shown below. In the image, the left column is the input patterns, and the right column is the final patterns after convergence.


INPUTS (left) ==> OUTPUT (right)