# Homework1: One Step Error Propability

Hai Dinh (19960331-4494)

## 1. Code

```
import time
import random
import argparse
import numpy as np

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

def generate_random_patterns(p, N):
    """
    Generate `p` random patterns, each with `N` bits/neurons.
    """
    patterns = np.random.randint(2, size=(p, N))
    patterns[patterns == 0] = -1
    return patterns

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
            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 compute_one_step_error_probability(p, N=120, n_trials=int(1e5), zerodiag=False):
    """
    Compute one-step error probability for Hopfield network

    Args:
        p (int): Number of stored patterns.

        N (int): Number of bits/neurons in each pattern.

        n_trials (int): Number of trials for computing the one-step error probability.

        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:
        A float representing the one-step error probability.
    """

    print("Computing probability for {} stored patterns...".format(p))
    start_time = time.time()
    errors = 0

    for i in range(n_trials):
        selected_neuron = random.randrange(N)
        selected_pattern = random.randrange(p)

        stored_patterns = generate_random_patterns(p, N)
        W = compute_weight_matrix(stored_patterns, row=selected_neuron, zerodiag=zerodiag)

        updated_neuron = signum(np.dot(W[0], stored_patterns[selected_pattern]))
        if updated_neuron != stored_patterns[selected_pattern, selected_neuron]:
            errors = errors + 1

    probability = errors / n_trials
    print("  Prob: {:.5f}".format(probability))
    print("  Time: {} seconds".format(time.time() - start_time))
    return probability

def main(args):
    ps = [12, 24, 48, 70, 100, 120]
    probs = [compute_one_step_error_probability(p, zerodiag=args.zerodiag) for p in ps]

    print()
    print("p   | probability")
    print("----|------------")
```

```
    for p, prob in zip(ps, probs):
        print("{:<3d} | {:.5f}".format(p, prob))

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Compute one-step error probability for Hopfield network"
    )
    parser.add_argument(
        "--zerodiag",
        "-z",
        action="store_true",
        help="Zero out all the weights along the diagonal of the weight matrix"
    )
    main(parser.parse_args())
```

## 2. Results

With weight matrix having **zero** diagonals, I get the following results:

```
$ python3 OneStepErrorProbability.py --zerodiag

Computing probability for 12 stored patterns...
Prob: 0.00052
Time: 4.172587156295776 seconds
Computing probability for 24 stored patterns...
Prob: 0.01140
Time: 5.4861860275268555 seconds
Computing probability for 48 stored patterns...
Prob: 0.05458
Time: 7.858587741851807 seconds
Computing probability for 70 stored patterns...
Prob: 0.09360
Time: 9.817108154296875 seconds
Computing probability for 100 stored patterns...
Prob: 0.13691
Time: 12.687513828277588 seconds
Computing probability for 120 stored patterns...
Prob: 0.15650
Time: 14.801092863082886 seconds

p   | probability
----|------------
12  | 0.00052
24  | 0.01140
48  | 0.05458
70  | 0.09360
100 | 0.13691
120 | 0.15650
```

With weight matrix having **non-zero** diagonals, I get the following results:

```
$ python3 OneStepErrorPropability.py

Computing probability for 12 stored patterns...
```

```
Prob: 0.00012
Time: 4.316368103027344 seconds
Computing probability for 24 stored patterns...
Prob: 0.00284
Time: 5.483811378479004 seconds
Computing probability for 48 stored patterns...
Prob: 0.01238
Time: 7.717572927474976 seconds
Computing probability for 70 stored patterns...
Prob: 0.01924
Time: 9.802137613296509 seconds
Computing probability for 100 stored patterns...
Prob: 0.02217
Time: 12.803136348724365 seconds
Computing probability for 120 stored patterns...
Prob: 0.02202
Time: 14.67278242111206 seconds
```

| p | probability |
|---|---|
| 12 | 0.00012 |
| 24 | 0.00284 |
| 48 | 0.01238 |
| 70 | 0.01924 |
| 100 | 0.02217 |
| 120 | 0.0220 |

As you can see, the one-step error probability is a much lower for the **non-zero** diagonals, which means that the network has less errors if fed with the same pattern as one of the stored patterns.

This is no surprise, because these non-zero weights along the diagonals will act as strong attractor to pull the fed pattern into one of the stored pattern, thus making sure that the network will less likely to make a mistake if the fed pattern is one of the stored pattern. This can be quite good as $\alpha = p/N$ gets smaller, because the network is more likely to be "confused" when the number of patterns increases.