Parisa Suchdev
Bidirectional Associative Memory (BAM):

To code up BAM was pretty straightforward after looking at the lecture notes, the hardest part was to understand the patterns it's creating. BAM remembers the vectors that are incomplete or have some added noise into it. Kosko, in his paper, also gives an example " A given note or chord of the melody is often sufficient to recollect the rest of the melody, to "name that tune."". I looked through few other papers to see the applications of this ANN and found an interesting one. A paper by Meiyanti et al., Performance of BAM algorithm and Viterbi algorithm for lie detection system using voice use BAM to detect lie solely on the basis on words a person is using like lots of "hmm" in the sentences.

I couldn't test such examples on my algorithm because I will need to brush it more so it could adapt different datasets. The dataset given in lecture to code up required ANN to memorize one change in pattern at a time (from [1,0,0] to [0,1,0] to [0,0,1]) and it did memorize it by first changing into bipolar vectors and edging weights from both direction because BAM is bidirectional, and the propagation moves both ways. We keep running it until the values do not converge. The input vector given to BAM reached the stable state just in one cycle.

```
Pattern 1

forward:[1 0 0] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
forward:[1. 0. 0.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
forward:[1. 0. 0.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]

Pattern 2

forward:[0 1 0] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
forward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
forward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]

Pattern 3

forward:[0 0 1] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
forward:[0. 0. 1.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
forward:[0. 0. 1.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
```

I was hoping something interesting would happen when I change the values of A to see what B converges to. Something rather funny and obvious happened. The values of A were set to values in associative vector B. The weight matrix was opposite to original input weight matrix (obvious). Threshold stayed the same of forward and backward passes (obvious because both values are same) and converged quickly (obvious because network remembers) **(pro: memory association)**

```
A1 = np.array([[0,0,1], [0,1,0], [1,0,0]]) #testing associative vector as input
BAM.main(A1, B)
```

```
Pattern 1

forward:[0 0 1] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
forward:[0. 0. 1.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
forward:[0. 0. 1.] ---> [-1. -1.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [-1. -1.  3.] ---> [0. 0. 1.]

Pattern 2

forward:[0 1 0] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
forward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
forward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]
backward:[0. 1. 0.] ---> [-1.  3. -1.] ---> [0. 1. 0.]

Pattern 3

forward:[1 0 0] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
forward:[1. 0. 0.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
forward:[1. 0. 0.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]
backward:[1. 0. 0.] ---> [ 3. -1. -1.] ---> [1. 0. 0.]

array([[ 3., -1., -1.],
       [-1.,  3., -1.],
       [-1., -1.,  3.]])
```

The next testing input included changing two values of A and analyzing was B converges to: Here for pattern 1 and pattern 3, that are same. BAM gets the right outputs quickly, however for pattern 2, it didn't get it quite right mainly because of the threshold. Negative values in threshold are getting zeros in output but for backward pass all threshold values are positive getting [1,1,1] as result and it stops converging. The error is 25%. **(Con: incorrect convergence, pro: associative memory)**

```
1 BAM.main(A2, B)
```

```
Pattern 1

forward:[1 0 1] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]
forward:[1. 0. 1.] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]
forward:[1. 0. 1.] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]

Pattern 2

forward:[1 1 0] ---> [-2. -2.  2.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [1. 1. 1.] ---> [1. 1. 1.]
forward:[1. 1. 1.] ---> [-1. -5.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [1. 1. 1.] ---> [1. 1. 1.]
forward:[1. 1. 1.] ---> [-1. -5.  3.] ---> [0. 0. 1.]
backward:[0. 0. 1.] ---> [1. 1. 1.] ---> [1. 1. 1.]

Pattern 3

forward:[1 0 1] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]
forward:[1. 0. 1.] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]
forward:[1. 0. 1.] ---> [ 2. -6.  2.] ---> [1. 0. 1.]
backward:[1. 0. 1.] ---> [ 2. -2.  2.] ---> [1. 0. 1.]

array([[ 1., -3.,  1.],
       [-3.,  1.,  1.],
       [ 1., -3.,  1.]])
```

In last Hopfield assignment, I couldn't understand why rule 2 will give more error than rule 1, is it because Hopfield settles to the minimum energy required to reach the stable state? BAM also uses minimum point in energy land space but when (A, B) presented to it reaches bidirectionally stable state?

I looked at Anna Jane's output method and followed a similar one.