High Dimensional Linear Algebra

Matt Gormley
Lecture 6
September 17, 2018
Q&A
HEBIAN LEARNING & MATRIX MEMORIES
Matrix Memories

Chalkboard

– Matrix Multiplication
  • examples of vector-vector, matrix-vector, matrix-matrix
  • views of matrix-matrix

– Compactly representing the sum of several outer products

– Storing several pattern pairs in a single weight matrix
Example adapted from Paul Munro (Pitt, SIS)

**Ex: Matrix Memories in Numpy**

### matrix_memories_1.py

```python
import numpy as np

s = np.array([[1, 0, 1, -2, 2]]).T

# Transpose and multiply

t = np.array([[2, 1, -1]]).T

W = np.matmul(t, s.T)

r = np.matmul(W, s)

n = np.matmul(W, s) / np.matmul(s.T, s)

assert np.array_equal(n, t)

print('s =
', s)
print('t =
', t)
print('W =
', W)
print('r =
', r)
print('n =
', n)
```

### stdout

```
s =
 [[ 1]
 [ 0]
 [ 1]
 [-2]
 [ 2]]

t =
 [[ 2]
 [ 1]
 [-1]]

W =
 [[ 2 0 2 -4 4]
 [ 1 0 1 -2 2]
 [-1 0 -1 2 -2]]

r =
 [[ 20]
 [ 10]
 [-10]]

n =
 [[ 2.]
 [ 1.]
 [-1.]]
```

**Goal:** Construct an association matrix $W$ that associates the pattern pair $[1, 0, 1, -2, 2] \rightarrow [2, 1, -1]$  

(It works up to normalization!)
Ex: Matrix Memories in Numpy

**matrix_memories_2.py**

```python
import numpy as np

s = np.array([[.5, -.5, .5, -.5]]).T
s = np.array([[.5, -.5, .5, -.5]]).T
W = np.matmul(t, s.T)
W = np.matmul(t, s.T)
r = np.matmul(W, s)
r = np.matmul(W, s)
assert np.array_equal(r, t)
assert np.array_equal(r, t)
```

```python
print('s =\n', s)
print('t =\n', t)
print('W =\n', W)
print('r =\n', r)
print('s =\n', s)
print('t =\n', t)
print('W =\n', W)
print('r =\n', r)
```

**Goal:** Construct an association matrix W that associates the pattern pair \([.5, -.5, .5, -.5]\) \(\rightarrow\) \([2, 1, -1]\)

**stdout**

```
s =
[[ 1]]
[ 0]
[ 1]
[-2]
[ 2]]

t =
[[ 2]
[ 1]
[-1]]

W =
[[ 2  0  2 -4  4]
[ 1  0  1 -2  2]
[-1  0 -1  2 -2]]

r =
[[ 2.]
[ 1.]
[-1.]]
```
Ex: Matrix Memories in Numpy

Example adapted from Paul Munro (Pitt, SIS)

**matrix_memories_3.py**

```python
import numpy as np

s1 = np.array([[-.5, .5, -.5, .5]])
s2 = np.array([[-.5, .5, -.5, .5]])
t1 = np.array([[2, 1, -1]])
t2 = np.array([[0, -1, 0]])
W1 = np.matmul(t1, s1)
W2 = np.matmul(t2, s2)
W = W1 + W2
r1 = np.matmul(W, s1)
r2 = np.matmul(W, s2)
```

**stdout**

```python
W1 =
[[ 1. -1.  1. -1. ]
 [ 0.5 -0.5  0.5 -0.5]
 [-0.5  0.5 -0.5  0.5]]
W2 =
[[ 0.  0. -0. -0. ]
 [-0.5 -0.5  0.5  0.5]
 [ 0.  0. -0. -0. ]]
W =
[[ 1. -1.  1. -1. ]
 [ 0. -1.  1.  0. ]
 [-0.5  0.5 -0.5  0.5]]
r1 =
[[ 2.]
 [ 1.]
 [-1.]]
r2 =
[[ 0.]
 [-1.]
 [ 0.]]
```

**Goal:** Construct an association matrix W that associates **two** pattern pairs:

[.5, -.5, .5, -.5] \(\rightarrow\) [2, 1, -1] and [.5, .5, -.5, -.5] \(\rightarrow\) [0, -1, 0]
**Ex: Matrix Memories in Numpy**

```python
import numpy as np

# Each s-pattern is a column
S = np.array([[0.5, 0.5, 0.5],
              [0.5, -0.5, -0.5],
              [-0.5, 0.5, -0.5],
              [-0.5, -0.5, 0.5]])

# All s-patterns are othonormal
dot = np.matmul(S.T, S)

# Each t-pattern is a column
U = np.array([[1, 3, 5],
              [2, 4, 6]])

# Construct and sum the weight matrices
W = np.matmul(U, S.T)

# Compute responses
R = np.matmul(W, S)
```

**Goal:** Associate many three pattern pairs
Ex: Matrix Memories in Numpy

**Matrix Memories in Numpy**

Example adapted from Paul Munro (Pitt, SIS)

```python
import numpy as np

# Each s-pattern is a column
S = np.array([[0.5, 0.5, 0.5],
              [0.5, -0.5, -0.5],
              [-0.5, 0.5, -0.5],
              [-0.5, -0.5, 0.5]])

# All s-patterns are orthonormal
dot = np.matmul(S.T, S)

# Each t-pattern is a column
U = np.array([[1, 3, 5],
              [2, 4, 6]])

# Construct and sum the weight matrices
W = np.matmul(U, S.T)

# Compute responses
R = np.matmul(W, S)
```

**Goal:** Associate many three pattern pairs

**Claim:** This approach works for any stimulus matrix S and response matrix U where the columns of S are orthonormal!

**Proof:** Left as an exercise for Homework 1
MATRIX MEMORIES & APPROXIMATE RECOVERY
Extra Notes

• The length of a vector is called the “norm”
  – there are different types of norms
  – here, we use the **Euclidean norm**:
    \[ |v| = \sqrt{v \cdot v} = \sqrt{\sum_{i=1}^{n} v_i^2} \]
  – MATLAB: \( \text{L=norm(v)} \)

• The cosine of the angle between two vectors \( u \) and \( v \) is the inner product divided by the product of the two norms

\[
\cos(u, v) = \frac{u \cdot v}{|u||v|} = \frac{u_1 v_1 + u_2 v_2 + \cdots + u_N v_N}{\sqrt{u_1^2 + \cdots + u_N^2} \sqrt{v_1^2 + \cdots + v_N^2}}
\]

• By dividing a vector \( v \) by the value of the norm, the result is a “normalized vector”, \( n \).
  – The norm of \( n \) is 1
  – The **direction of** \( n \) is the same as the direction of \( v \)
  – The inner product of two normalized vectors is equal to the cosine of the angle between them
Cosines

- The cosine between two vectors $\cos(u,v)$ is sometimes used as a measure of the similarity between $u$ and $v$.
  - $\cos = 1$: $u$ and $v$ point in the same direction (angle is 0)
  - $\cos = 0$: $u$ and $v$ are orthogonal (angle = $\pi/2$)
  - $\cos = -1$: $u$ and $v$ point are opposite (angle is $\pi$)

- If the columns of $V$ are normalized vectors, then $V^TV$ is a matrix of cosines between pairs of columns

\[
\begin{bmatrix}
-1 & 2 & 1 \\
1 & 2 & 0
\end{bmatrix}
\begin{bmatrix}
2 & 0 & -1 \\
0 & 8 & 2 \\
-1 & 2 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1 & 1 & 1 \\
\sqrt{2} & \sqrt{2} & 0 \\
\sqrt{2} & \sqrt{2} & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -\frac{1}{\sqrt{2}} \\
0 & 1 & \frac{1}{\sqrt{2}} \\
-\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1
\end{bmatrix}
\]

Slide from Paul Munro (Pitt, SIS)
Cosines (cont)

\[ s = \begin{pmatrix} s^{(1)} & s^{(2)} & s^{(3)} \\ -1 & 2 & 1 \\ 1 & 2 & 0 \end{pmatrix} \quad s^T s = \begin{pmatrix} s^{(1)} & s^{(2)} & s^{(3)} \\ s^{(1)} & 2 & 0 & -1 \\ s^{(2)} & 0 & 8 & 2 \\ s^{(3)} & -1 & 2 & 1 \end{pmatrix} \]

Each element of \( s^T s \) is the inner product of two vectors from \( \{s^{(1)}, s^{(2)}, s^{(3)}\} \).

These are the cosines of \( \{v^{(1)}, v^{(2)}, v^{(3)}\} \) (they are normalized) and are the cosines of \( \{s^{(1)}, s^{(2)}, s^{(3)}\} \), since they have the same angles!

\[ v = \begin{pmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 \\ 1 & 1 & 0 \end{pmatrix} \quad v^T v = \begin{pmatrix} v^{(1)} & v^{(2)} & v^{(3)} \\ v^{(1)} & 1 & 0 & -\frac{1}{\sqrt{2}} \\ v^{(2)} & 0 & 1 & \frac{1}{\sqrt{2}} \\ v^{(3)} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 \end{pmatrix} \]
**High Dimensional Vectors**

*Observe:* Random vectors in high dimensions tend to be “almost orthogonal”

---

**high_dim.py**

```python
>>> S25d = np.random.rand(25, 5) - 0.5
>>> S25d = S25d / np.linalg.norm(S25d, axis=0)
>>> print(np.array2string(S25d, precision=2))
[[ 0.18 -0.1  0.23 -0.05  0.17]
 [ 0.2  0.19  0.24  0.2 -0.21]
 [ 0.17 -0.06 -0.07 -0.2 -0.17]
 [ 0.14  0.15  0.09  0.09  0.18]
 [ 0.23  0.31 -0.1  0.08 -0.17]
 [ 0.28  0.23  0.14 -0.07  0.08]
 [-0.1 -0.28 -0.09 -0.18  0.26]
 [ 0.04  0.26  0.14 -0.07 -0.19]
 [-0.27  0.24 -0.27  0.16 -0.14]
 [ 0.1  0.27  0.01 -0.37  0.24]
 [-0.3  0.24  0.15  0.1  0.33]
 [ 0.29  0.07 -0.24  0.36  0.3]
 [ 0.04 -0.14 -0.05  0.37  0.22]
 [ 0.09  0.33  0.3  0.15  0.24]
 [-0.07 -0.18 -0.13  0.21 -0.16]
 [ 0.32  0.13  0.19  0.05 -0.19]
 [ 0.18  0.03  0.22 -0.04 -0.25]
 [ 0.01 -0.24  0.3 -0.25 -0.08]
 [-0.33  0.25  0.18 -0.22 -0.01]
 [-0.29 -0.06 -0.27  0.27 -0.34]
 [-0.14  0.18 -0.25 -0.2 -0. ]
 [ 0.32  0.18  0.26 -0.24  0.04]
 [ 0.06 -0.02 -0.04  0.18  0.27]
 [-0.08  0.27  0.24 -0.21  0.18]
 [ 0.09  0.01  0.31  0.03 -0.03]]
```

---

**stdout**

```python
>>> dot = np.matmul(S25d.T, S25d)
>>> print(np.array2string(dot, precision=2))
[[ 1.  0.14  0.36  0.02  0.03]
 [ 0.14  1.  0.28 -0.09  0.08]
 [ 0.36  0.28  1.  -0.32  0.1]
 [ 0.02 -0.09 -0.32  1.  -0.01]
 [ 0.03  0.08  0.1 -0.01  1.  ]]
```

Each column of S is a random 25-dimensional normalized s-pattern

Because columns of S are normalized, each entry in dot is the cosine of a pair of s-patterns
High Dimensional Vectors

**Observe:** Distribution over cosine distances concentrates around zero as dimensionality increases

### 25 dimensional

\[
S_{25d} = \text{np.random.rand}(25, \text{numvecs}) - 0.5 \\
S_{25d} = \frac{S_{25d}}{\text{np.linalg.norm}(S_{25d}, \text{axis}=0)} \\
y_{25} = 1 - \text{distance.pdist}(S_{25d}.T, 'cosine') \\
\text{pd.DataFrame}(y_{25}).\text{hist}()[0,0]
\]

### 500 dimensional

\[
S_{500d} = \text{np.random.rand}(500, \text{numvecs}) - 0.5 \\
S_{500d} = \frac{S_{500d}}{\text{np.linalg.norm}(S_{500d}, \text{axis}=0)} \\
y_{500} = 1 - \text{distance.pdist}(S_{500d}.T, 'cosine') \\
\text{pd.DataFrame}(y_{500}).\text{hist}()[0,0]
\]

Note the difference in x-axis scale!
Using log(x)

boxplot(abs([h25',h500',h10k']))

boxplot(log(abs([h25',h500',h10k'])))

boxplot(log(abs([h25',h500',h10k']))/log(10))

Slide from Paul Munro (Pitt, SIS)
LINEAR ALGEBRA
Linear Algebra: Matrices

Chalkboard

– Types of Matrices
  • square matrix
  • diagonal matrix
  • identity matrix
  • symmetric matrix (and the set of symmetric matrices)
  • orthogonal matrix (different than orthogonal vectors!)

– Matrix Operations
  • The Transpose
  • The Trace
  • Inverse
  • Determinant
Linear Algebra: Linear Independence

Chalkboard

– Linear Independence
  • Linear combinations of vectors
  • Linear independence/dependence
  • Rank: column rank, row rank, full rank
  • Full rank \(\rightarrow\) invertible
Linear Algebra + Matrix Memories

Chalkboard

– Intuition: why Matrix Memories work
– Intuition: invertible Matrix Memories and geometry