Minor in AI

Vectors, Matrices, and Markov Models

1 Introduction: The Matrix in Real Life

Imagine shopping online. Every product you see has multiple attributes: price, size, color, rating. At the backend, this data is organized as a **matrix** where rows represent products and columns represent features. Similarly, digital images are matrices of pixel values, with three matrices (red, green, blue) combining to form colors. These real-world examples show how matrices structure complex data.

In AI, we use matrices for:

- **Linear regression**: Predicting house prices from features
- **Image recognition**: Processing pixel matrices in CNNs
- **Recommendation systems**: User-item interaction matrices
- **Markov models**: Predicting next states (e.g., focused/distracted students)

2 Core Concepts and Implementation

2.1 The Great List vs Array Debate

Key Difference

Lists = Mixed data types (heterogeneous)
Arrays = Same data type (homogeneous)

Why does this matter? Arrays are **100x faster** for mathematical operations. Here's why:

1. *Memory efficiency*: Arrays use contiguous memory blocks
2. *Implicit addressing*: Calculate positions via base address + (index $\times$ data size)
3. *Optimized operations*: Single instruction for all elements

```
import numpy as np

# List (heterogeneous - mixed types)
list_ex = [5.2, "hello", 1.4, 'a']
print(list_ex) # Output: [5.2, 'hello', 1.4, 'a']

# Array with mixed types $->$ converts to string
arr_mixed = np.array([5.2, 3.5, "hello", 1.7])
print(arr_mixed) # Output: ['5.2' '3.5' 'hello' '1.7']

# Pure numeric array $->$ retains float type
arr_num = np.array([5.2, 3.5, 4.5, 1.7])
print(arr_num) # Output: [5.2 3.5 4.5 1.7]
```

Listing 1: Lists vs Arrays in Python

2.2 Matrix Operations in Machine Learning

2.2.1 Dot Product: The AI Workhorse

Used in linear regression and neural networks. Calculates weighted sums: $\text{dot}(w, x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n$

```
w = [3, 4, 5, 2]  # Model weights
x = [1, 2, 5, 3]   # Input features

# Method 1: Using np.dot()
prediction = np.dot(w, x)  # 3*1 + 4*2 + 5*5 + 2*3 = 42

# Method 2: Convert to arrays and use @ operator
w_arr = np.array(w)
x_arr = np.array(x)
pred = w_arr @ x_arr  # Also 42
```

Listing 2: Dot Product Implementation

2.2.2 Matrix Multiplication: Neural Network Brains

Each layer in a neural network uses matrix multiplication. Shape matters: (m×n) matrix × (n×p) matrix $->$ (m×p) result.

```
# Neural network layer simulation
inputs = np.array([[1, 2, 3]])     # Shape (1, 3)
weights = np.array([[0.1, 0.2],     # Shape (3, 2)
                    [0.3, 0.4],
                    [0.5, 0.6]])

output = inputs @ weights  # Result shape: (1, 2)
# Output: [[1*0.1 + 2*0.3 + 3*0.5, 1*0.2 + 2*0.4 + 3*0.6]]
#        = [[2.2, 2.8]]
```

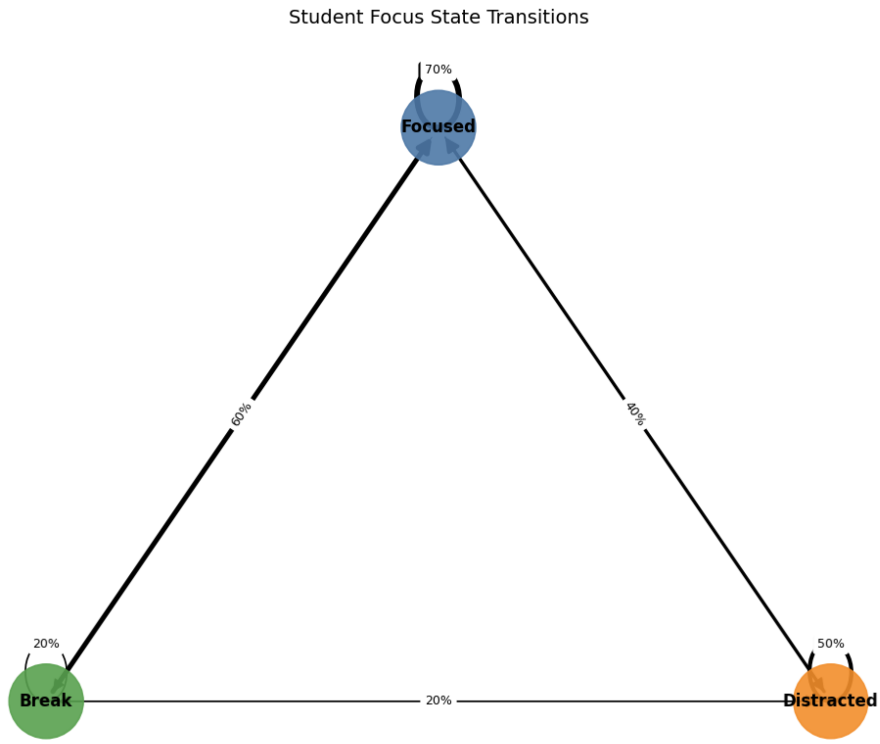
2.2.3 Transpose and Inverse

- **Transpose**: Swap rows/columns ($A_{ij} \to A_{ji}$)
- **Inverse**: Matrix "division" (A^{-1} where $A \times A^{-1} = I$)

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

A_transposed = A.T  # [[1, 4],
                    #  [2, 5],
                    #  [3, 6]]

B = np.array([[4, 7],
              [2, 6]])
B_inverse = np.linalg.inv(B)  # [[ 0.6, -0.7],
                              #  [-0.2,  0.4]]
```



2.3 Markov Models: Predicting the Present

Core Principle

"The future depends only on the present, not the past."

Real-world applications:

- Student focus states (focused $->$ distracted $->$ break)
- Weather prediction (sunny $->$ rainy)
- PageRank algorithm

Implementation components:

1. **States**: Possible conditions (e.g., focused, distracted, break)
2. **Transition Matrix**: Probabilities of moving between states

```
states = ["Focused", "Distracted", "Break"]

# Transition matrix [from, to]:
# Rows: Current state, Columns: Next state
tm = np.array([
    [0.7, 0.2, 0.1],  # From Focused
    [0.4, 0.5, 0.1],  # From Distracted
    [0.6, 0.2, 0.2]   # From Break
])

# Simulation
current_state = 0  # Start focused
```

```
for step in range(5):
    next_state = np.random.choice([0,1,2], p=tm[current_state])
    print(f"Step {step}: {states[current_state]} $->$ {states[next_state
   ]}")
    current_state = next_state

# Example output:
# Step 0: Focused $->$ Distracted
# Step 1: Distracted $->$ Focused
# Step 2: Focused $->$ Focused
# Step 3: Focused $->$ Break
```

Listing 3: Student Focus Markov Model

3 Why This Matters

- **Efficiency**: Arrays process ML data 100x faster than lists
- **AI Foundations**: Matrix operations power neural networks and deep learning
- **Real-time prediction**: Markov models enable quick decisions based on current state
- **Exam focus**: These concepts are crucial for Module B and offline exams

Pro Tip

Always convert data to NumPy arrays before ML processing.
The speed difference becomes crucial with real-world datasets!