Module 3

Simulating Attention in Memory

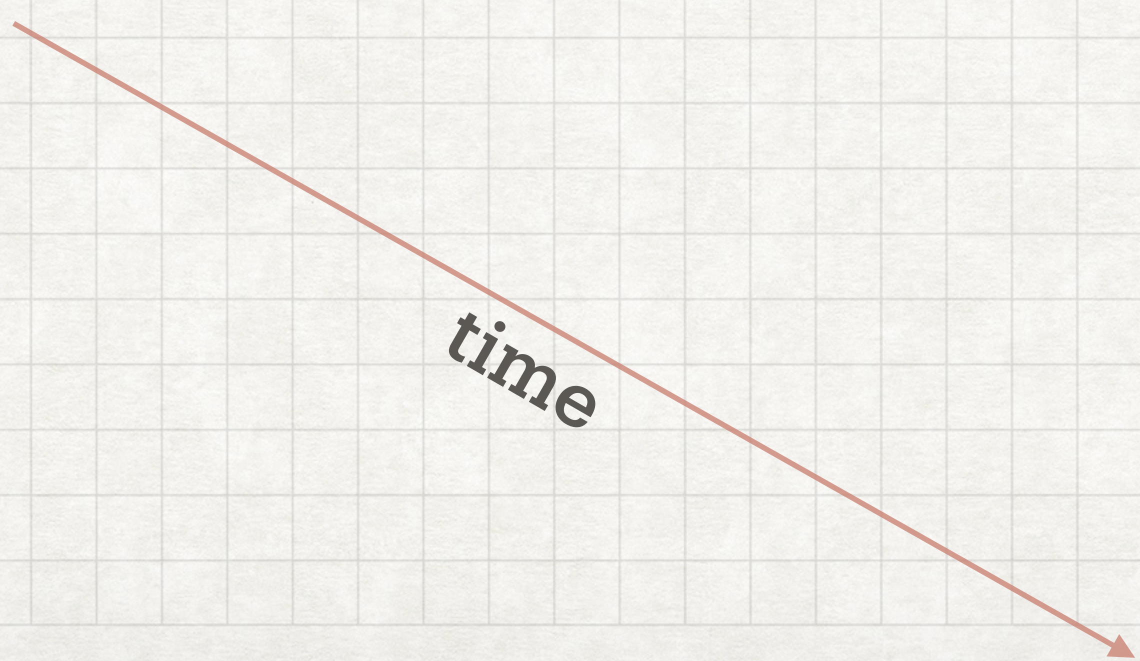
&

Learning Dot-product attention

Mini-Project 3: Attention-based retrieval from memory

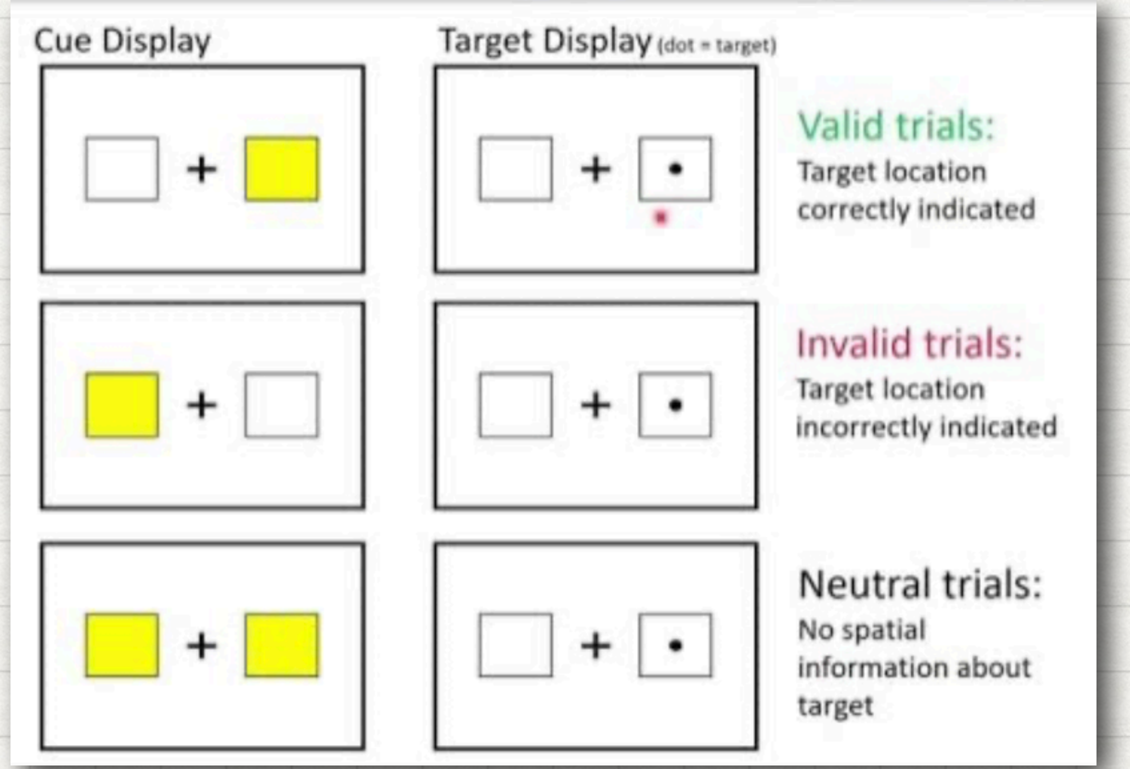
Recall a word from memory based on 'cue'

cue: valid / invalid



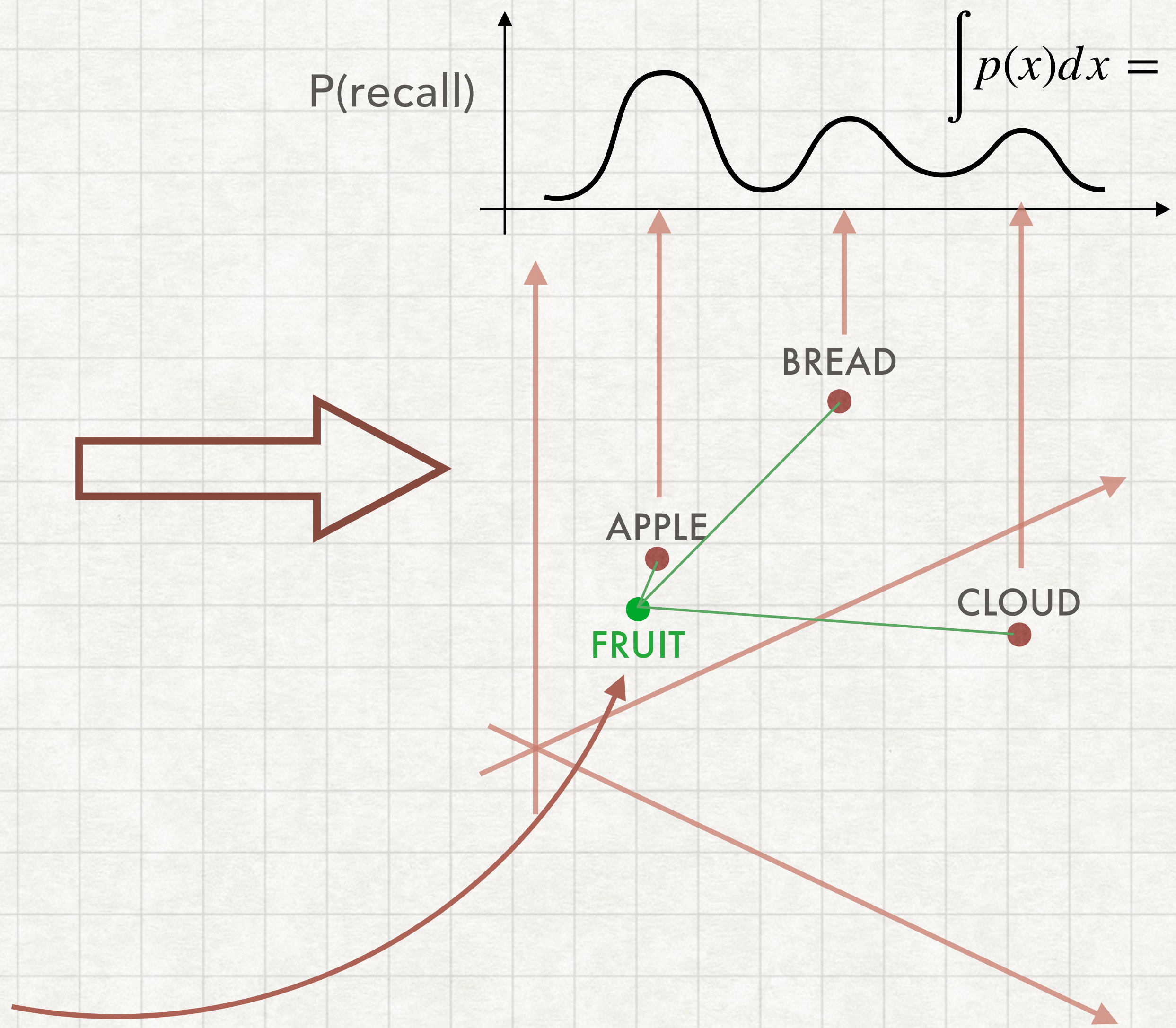
Mini-Project 3: Attention-based retrieval from memory

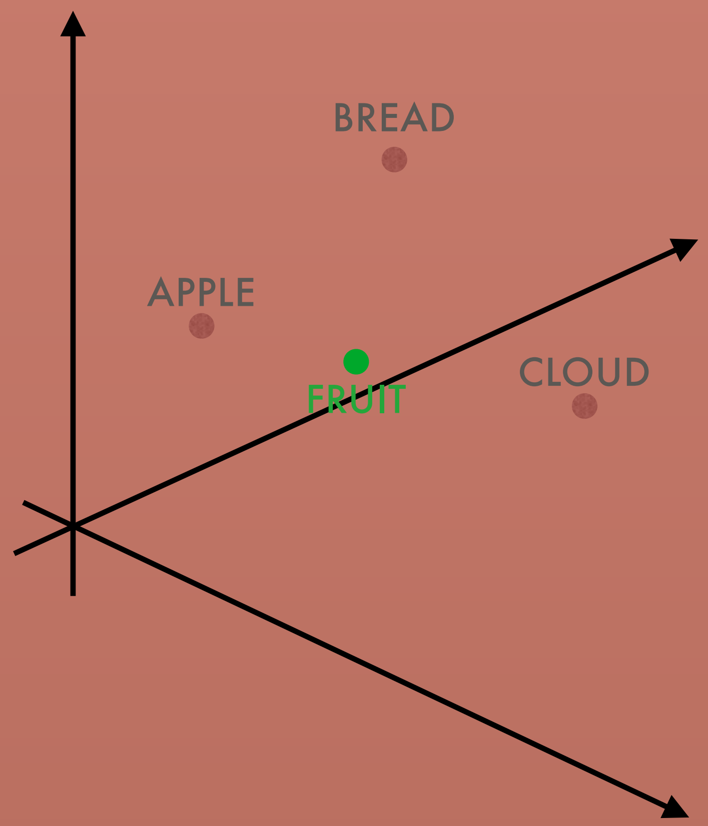
* Recall depends on *match* between cue & memory (Tulving & Thomson, 1973)
* Words are encoded in a semantic context in memory
* Attention can be diverted to target (valid) or away from target (invalid)… Posner cueing paradigm



Mini-Project 3: Attention-based retrieval from memory

How can we simulate this memory process?

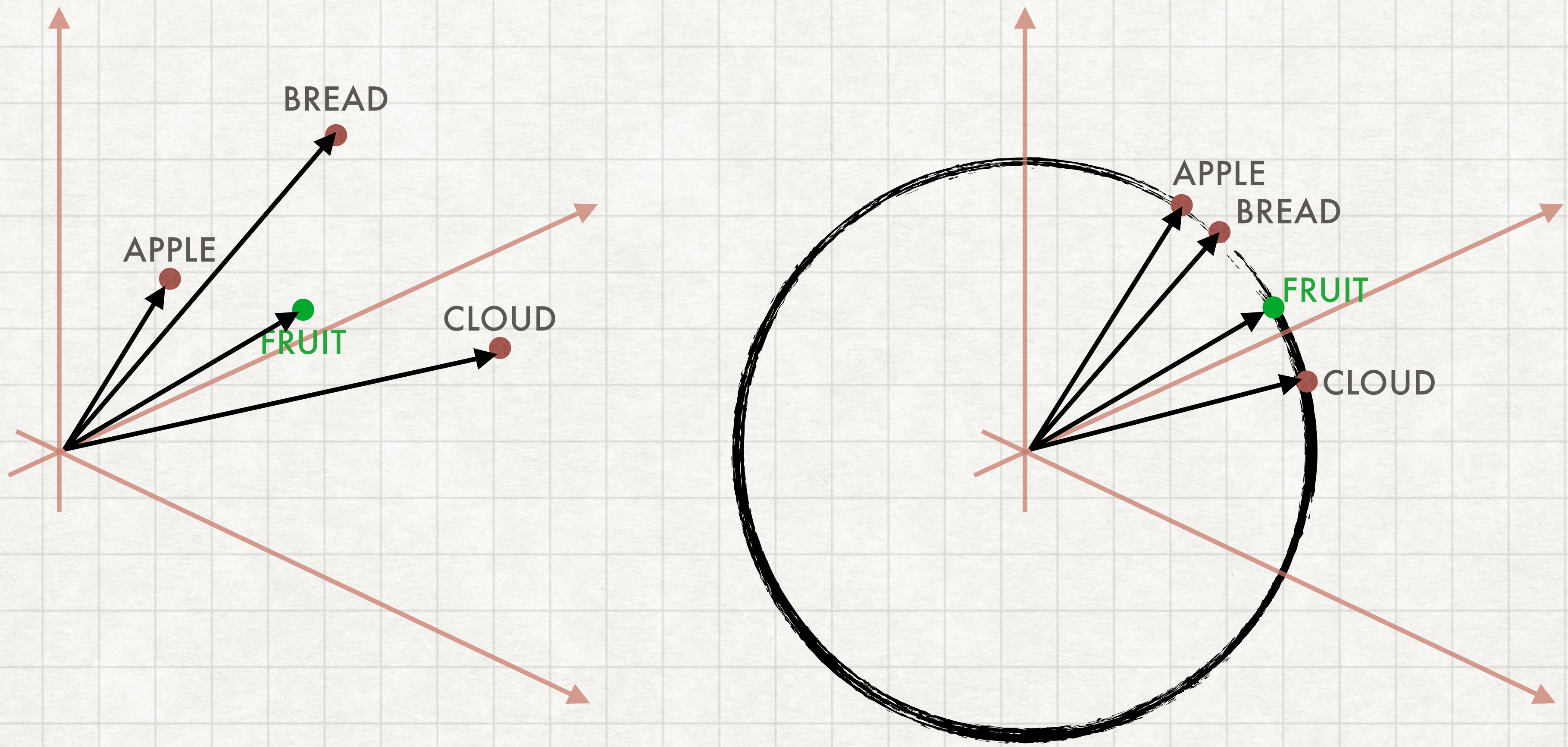




Step 1: Encode words in a high-dimensional space

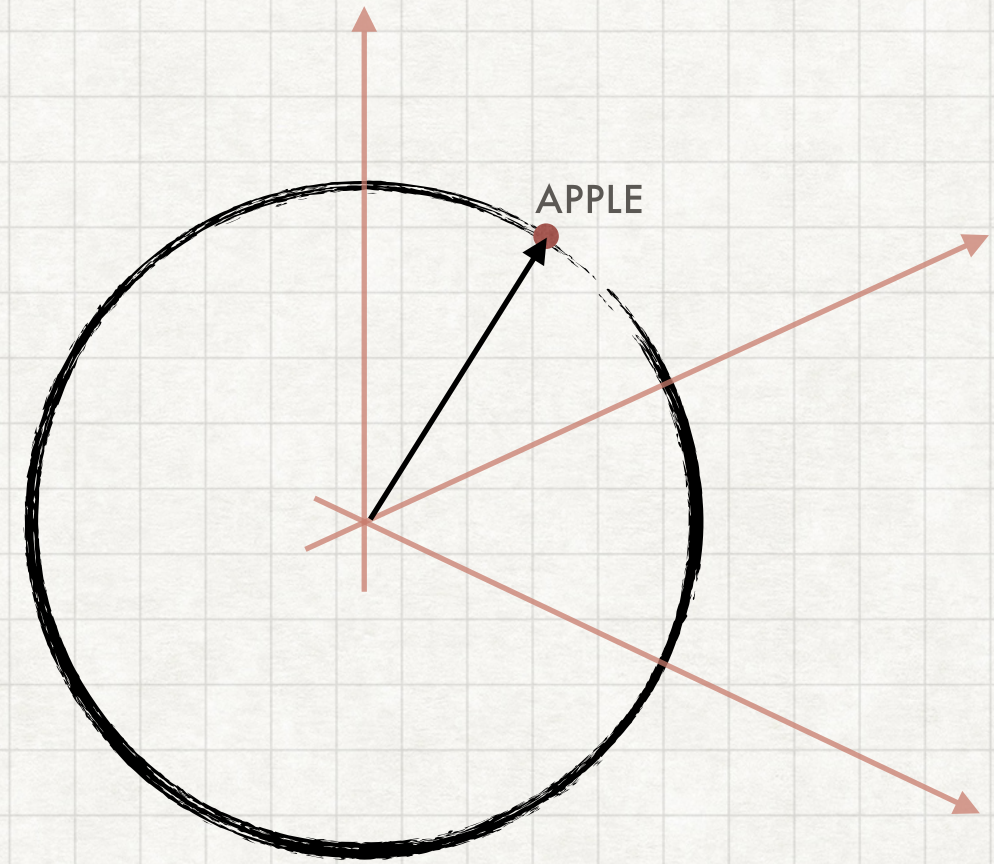
Note: Similarity using cosine similarity measure

Key part of “dot-product attention”



Demo 1: Encode words in high-dimensional space

APPLE



We are going to write Python code to convert:

APPLE

$n\text{-}dim$

Demo 1: Encode words in high-dimensional space

Write Python code to:

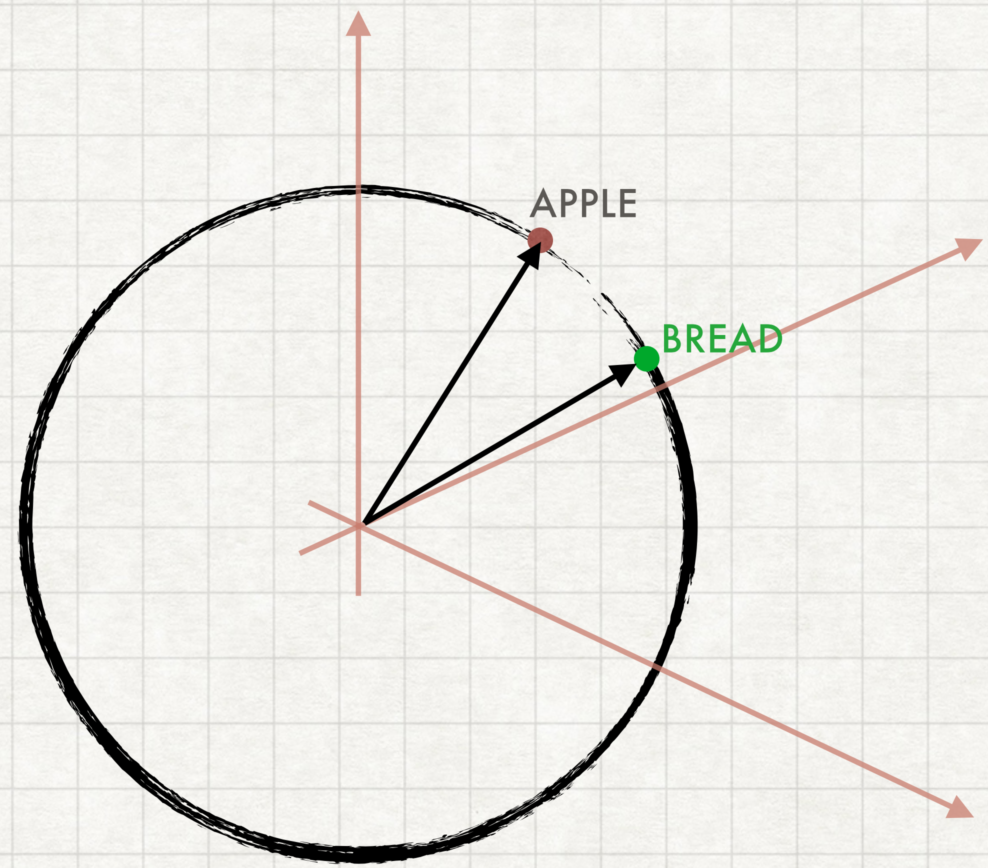
* Create a ‘WordEncoder’ class. This class should:
    * Contain an attribute called ‘dim’ (the dimension of vector space)
    * Contain a method call encode()
    * This method should take a string as input and convert it into a random vector in an n-dimensional space
    * This random vector should lie on the unit circle

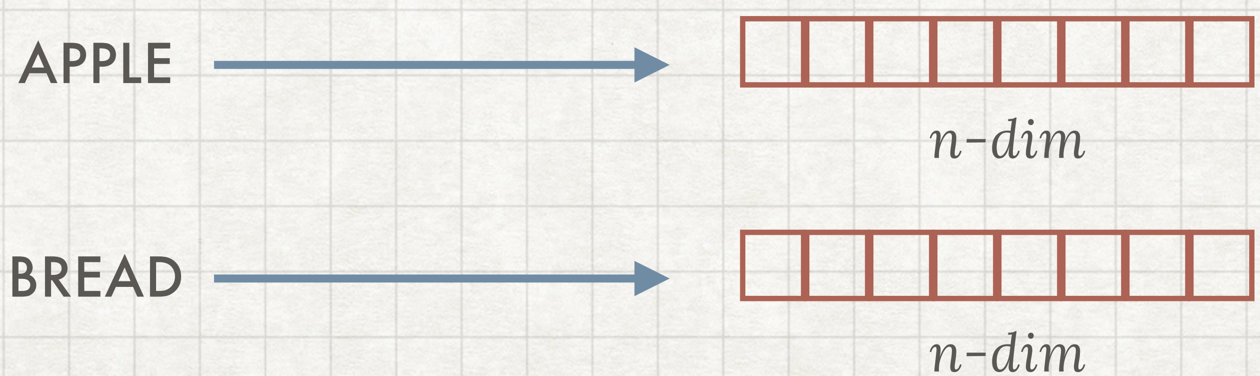
Demo 2: Find similarity between representations

Write Python code to:

* Define a function class called 'Similarity', with functions cosine() & euclidean() that calculate the cosine & euclidean distances

* Use one of these functions based on the initialisation of Similarity instance.



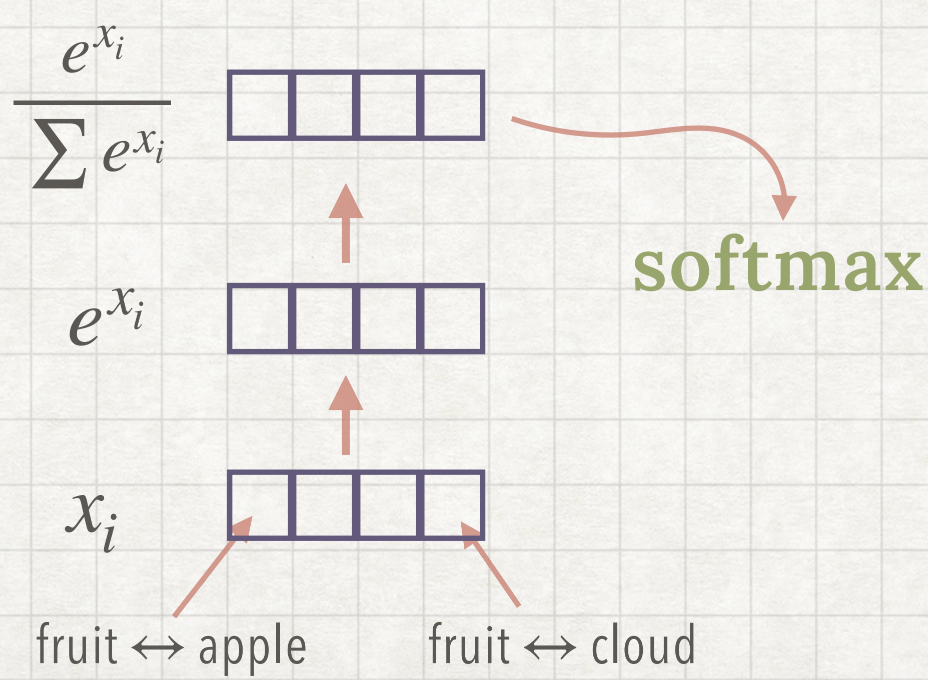


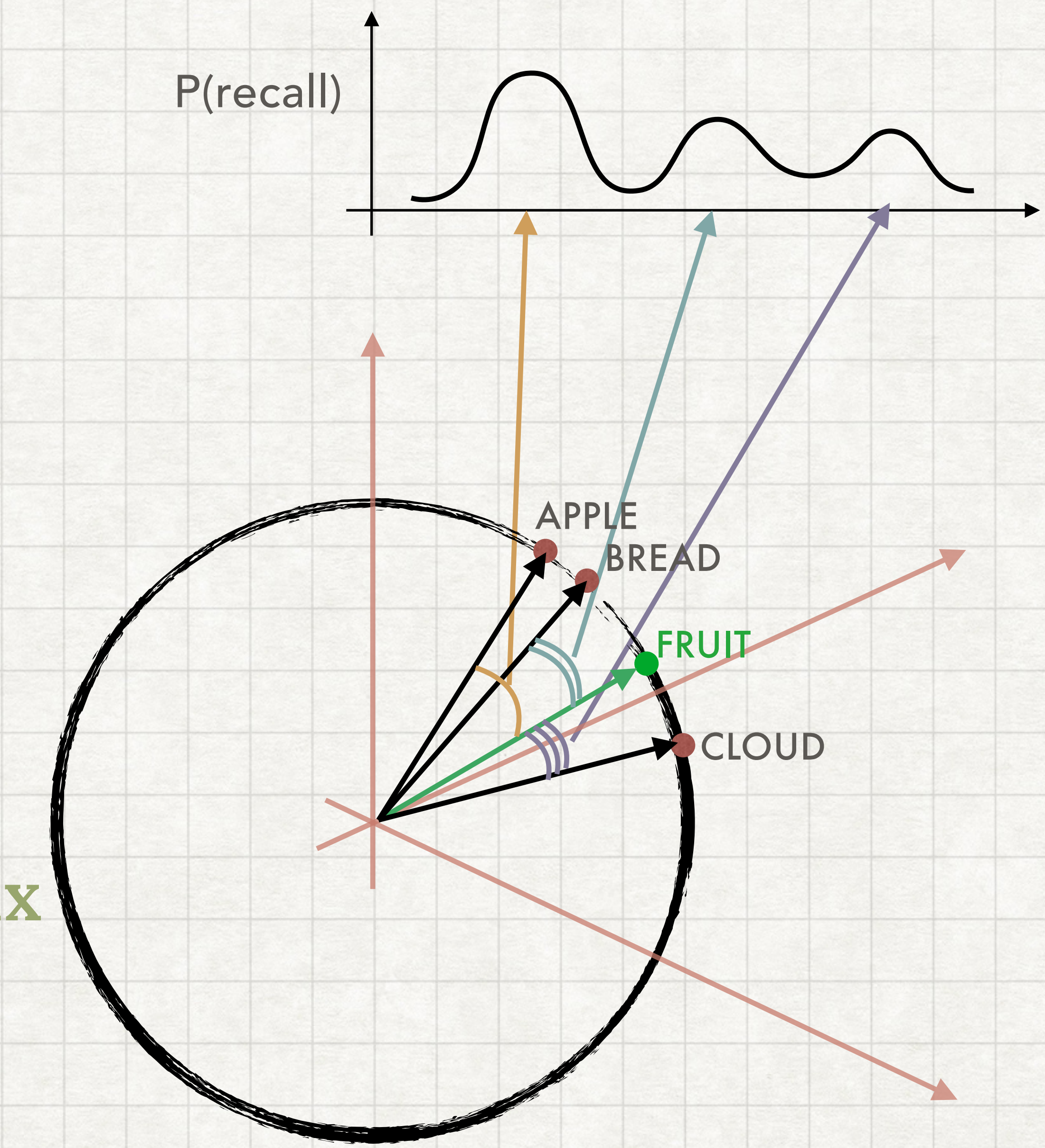
$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}$$

Demo 3: Scores to probabilities

Write Python code to:

* Create a numpy array called scores, with distances to each stored memory
* Convert these scores to a probability distribution over memory traces



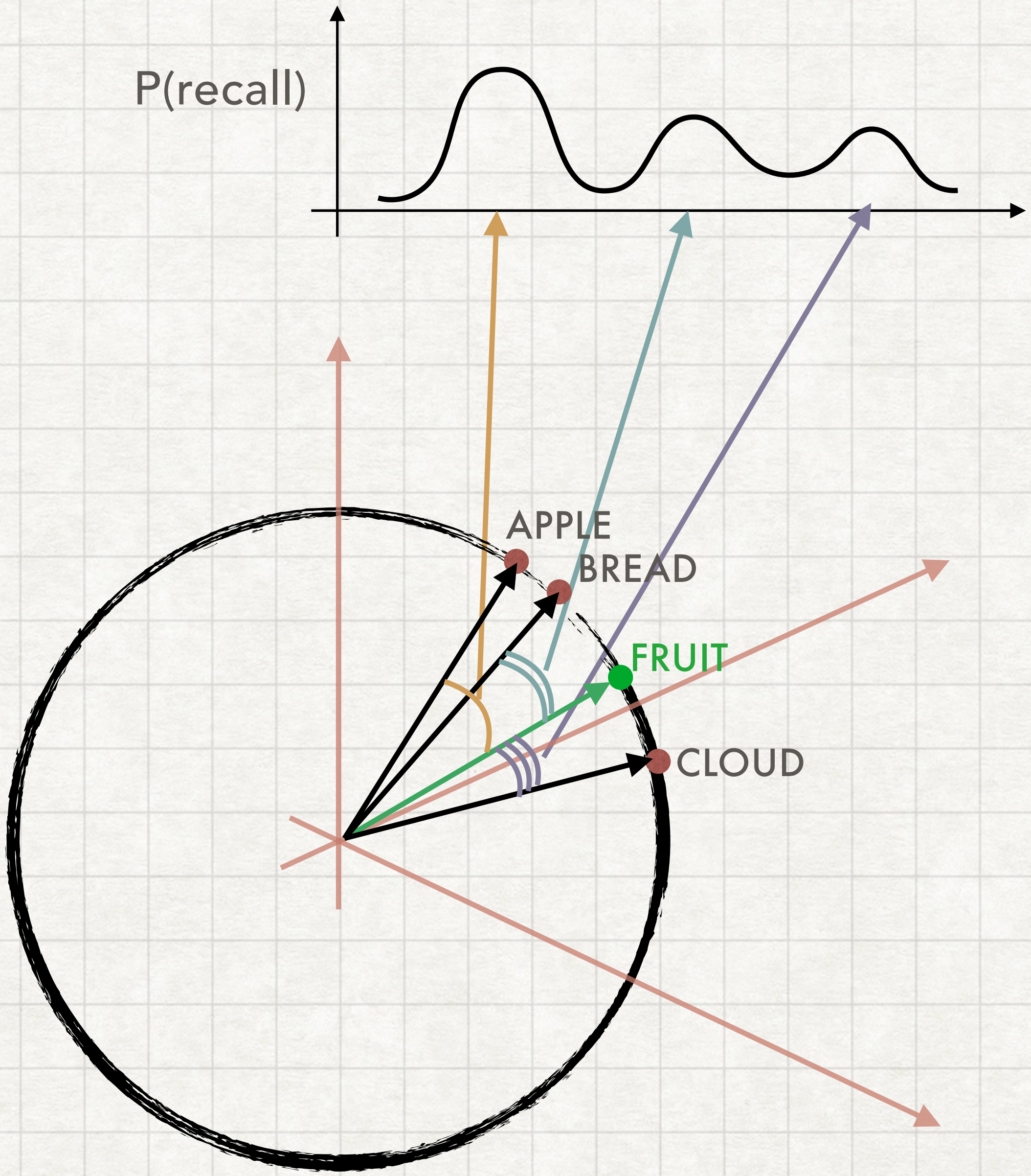


Demo 3: Pseudocode

```
FUNCTION softmax(scores):
    # 1) exponentiate to amplify differences
    e_i ← exp(z_i) for each z_i

    # 2) normalize to make probabilities
    total ← sum(e_i)
    w_i ← e_i / total for each i

    RETURN [w_1, ..., w_n]
```



Demo 4: Scale the attention based on temperature

Softmax function allows temperature scaling - that is how sharply focused the attention is. When temperature is high, attention is "diffused", while when temperature is low, attention is highly focused (winner-take-all).

$$\frac{e^{x_i}}{\sum e^{x_i}} \longrightarrow \frac{e^{\frac{x_i}{\tau}}}{\sum e^{\frac{x_i}{\tau}}}$$

* Write Python code to change the softmax function so that the probabilities are computed based on temperature.

* Test the function for different values of temperature parameter, but same scores

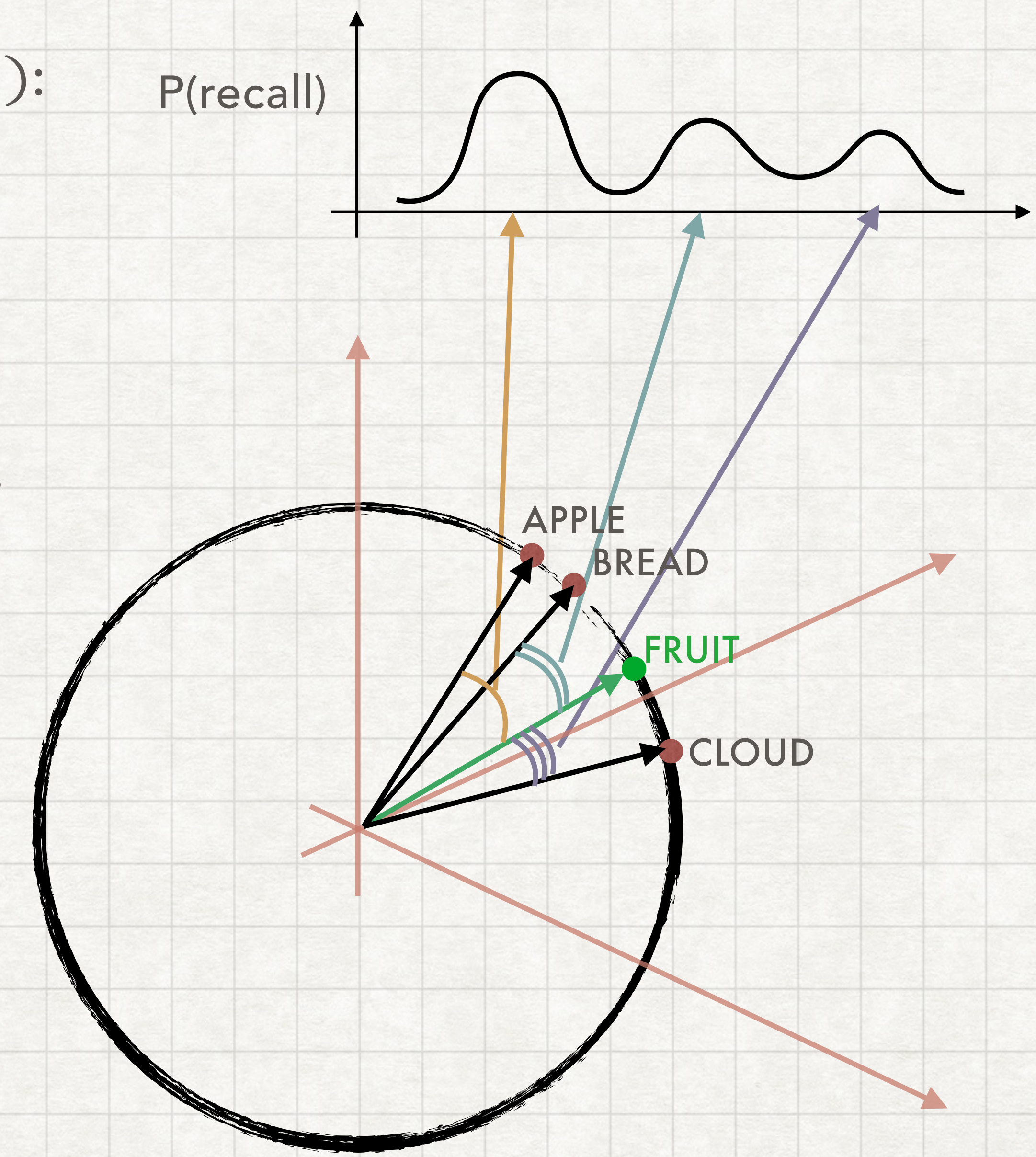
Demo 3: Pseudocode

```
FUNCTION softmax(scores, temperature τ):
  # 1) temperature scaling
  for each s in scores:
    z_i ← s / τ

  # 2) exponentiate to amplify differences
  e_i ← exp(z_i) for each z_i

  # 3) normalize to make probabilities
  total ← sum(e_i)
  w_i ← e_i / total for each i

  RETURN [w_1, ..., w_n]
```



Demo 5: Compute Context vector via Dot-product Attention

Cue → Query

Memory index → Key

Memory rep → Value

Let us put it all together and parallelise the attention mechanism.

Write a class called DotProductAttention with a function attend() that

1. computes the scores for all memories in parallel
2. Computes the attention weights for each memory
3. Computes a "context" based on aggregated memory recalled
4. Returns this context and attention weights

Test this function for memories: ['apple', 'bread', 'cloud', 'drum', 'eagle'] & query 'apple'

Demo 5: Compute Context vector via Dot-product Attention

Cue → Query

Memory index → Key

Memory rep → Value

```
CLASS DotProductAttention(temperature τ):
    METHOD attend(query q, keys K, values V):
        d ← dimension of q

        # 1) compute similarity scores (dot products)
        for each key k_i in K:
            score_i ← q · k_i

        # 2) convert scores to attention weights via softmax
        w ← softmax([score_1, ..., score_n], τ)

        # 3) compute context vector as weighted sum of values
        c ← Σ_i (w_i * v_i)

        RETURN w, c
```