Author: Riccardo Cecchini (rcecchini.ds@gmail.com) 21 Feb 2025

Angular Embeddings: A Novel Hierarchical Representation for Natural Language Semantics

Abstract:

Sentence embedding techniques are fundamental to numerous natural language processing tasks. Traditional methods, while powerful, often struggle with high dimensionality, lack inherent structure, and can be difficult to interpret. This paper introduces *angular embeddings*, a novel approach that represents semantic information using angles within a hierarchical framework. We depart from conventional Cartesian coordinate systems, embracing a circular topology where each dimension is characterized by an angle. This naturally encodes similarity and offers a bounded representation. We further enhance this concept with a *three-pole* angular representation, significantly enriching the expressiveness within each dimension. We present the mathematical foundations of both two-pole and three-pole angular embeddings, detail a hierarchical model architecture leveraging pre-trained sentence encoders, and discuss training procedures with a combined similarity and concept classification loss. We conclude by outlining potential avenues for future research and applications.

Introduction:

The quest for meaningful numerical representations of text, particularly at the sentence level, is a cornerstone of modern natural language processing. Sentence embeddings, which map sentences to fixed-size vectors, are essential for tasks ranging from semantic search and question answering to machine translation and text summarization. While techniques like averaging word embeddings (Word2Vec, GloVe) provide a simple baseline, they often fail to capture the nuances of sentence-level meaning. More sophisticated approaches, such as Sentence Transformers based on architectures like BERT and RoBERTa, have achieved remarkable success. However, these models often operate in high-dimensional spaces, making them computationally expensive and difficult to interpret. Furthermore, the lack of inherent structure in these vector spaces can hinder generalization and make the models susceptible to adversarial attacks.

This paper proposes a departure from the conventional Cartesian representation of sentence embeddings. We introduce *angular embeddings*, a novel framework that leverages the properties of angles to encode semantic information. Instead of representing each dimension as a coordinate along an axis, we represent it as an angle on a circle. This seemingly simple change has profound implications. First, it introduces a natural notion of similarity based on angular distance. Second, it provides a bounded representation, mitigating some of the issues associated with high dimensionality. Third, it opens the door to hierarchical representations that capture semantic information at multiple levels of granularity.

We begin by formalizing the concept of two-pole angular embeddings, where each dimension is characterized by a single angle, and similarity is determined by the squared cosine of half the angular difference. We then introduce a significant extension: *three-pole angular embeddings*. This innovation addresses a fundamental limitation of the two-pole representation, namely its inability to capture more than two opposing concepts within a single dimension. By introducing three poles, analogous to the primary colors red, green, and blue, we significantly increase the representational capacity of each angular dimension.

The angular embeddings are organized into a hierarchical structure, inspired by the multi-faceted nature of meaning. At the highest level, we posit a set of fundamental semantic categories – for instance, "what," "when," "how," and "why" – that provide a coarse-grained representation of the sentence. Subsequent levels refine this representation, with each level branching into subcategories, creating a tree-like structure. This hierarchical organization allows the model to capture both general and specific aspects of meaning.

Related Work:

The field of sentence embeddings has a rich history. Early methods, such as Bag-of-Words and TF-IDF, while simple and interpretable, lack the ability to capture semantic relationships. Distributed representations of words, like Word2Vec and GloVe, marked a significant advance, but their application to sentence-level meaning often requires averaging, which can lead to information loss. The advent of recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) allowed for modeling sequential dependencies, but these models can struggle with long-range dependencies. The transformer architecture, with its self-attention mechanism, has revolutionized the field, leading to state-of-the-art results in various NLP tasks. Sentence Transformers, which fine-tune pre-trained transformer models for sentence-level tasks, represent the current state of the art.

Our work builds upon this foundation but introduces a fundamentally different representational paradigm. While most existing methods operate in Euclidean space, our angular embeddings reside on a circular manifold. This connects our work to research on directional statistics and the von Mises-Fisher distribution, which is commonly used to model data on spheres. Spherical embeddings have also found applications in areas like recommendation systems, where they can capture cyclical patterns.

Hierarchical models have a long history in NLP, particularly in areas like parsing and discourse analysis. Recursive neural networks have been used to model tree-structured data, and hierarchical attention networks have been applied to document classification and summarization. Our hierarchical angular embeddings share some similarities with these approaches, but they differ in their fundamental representation and their focus on capturing a broad spectrum of semantic information.

Theta Vector Representation: A Mathematical Framework

We now delve into the mathematical formalization of theta vectors, beginning with the two-pole variant and then extending it to the more expressive three-pole representation.

Two-Pole Theta Vectors:

Let θ represent an angle in the range $[0, \pi]$. A two-pole theta vector of dimension d is simply a collection of d such angles: $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_d)$. The core of the two-pole representation lies in its similarity function. Given two angles, θ_1 and θ_2, their similarity is defined as:

$$\mathrm{sim}(\theta_1, \theta_2) = \cos^2((\theta_1 - \theta_2) / 2)$$

This function possesses several desirable properties. First, it is bounded between 0 and 1, providing a normalized measure of similarity. Second, it is symmetric: $\mathrm{sim}(\theta_1, \theta_2) = \mathrm{sim}(\theta_2, \theta_1)$. Third, it reaches its maximum value of 1 when $\theta_1 = \theta_2$ and its minimum value of 0 when θ_1 and θ_2 are diametrically opposite (i.e., $|\theta_1 - \theta_2| = \pi$). This aligns with the intuitive notion of similarity on a circle. The similarity between two d-dimensional theta vectors, $\boldsymbol{\theta}^{(1)}$ and $\boldsymbol{\theta}^{(2)}$, can be computed as the average similarity of their corresponding angles:

$$\mathrm{sim}(\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}) = (1/d)\ \Sigma_{i=1}^{d}\ \mathrm{sim}(\theta_1^{(1)}, \theta_i^{(2)})$$

Three-Pole Theta Vectors:

While the two-pole representation offers advantages over Cartesian coordinates, it is inherently limited in its ability to capture complex relationships within a single dimension. Consider the analogy of color representation. A single dimension with two poles can represent a spectrum from red to cyan, but it cannot capture the full richness of a color wheel with red, green, and blue.

To address this limitation, we introduce the three-pole theta vector. Crucially, each dimension is *still* represented by a single angle, $\theta \in$ 0, 2π. The three-pole concept manifests in the *similarity function*. Given two angles, θ_1 and θ_2, their three-pole similarity is defined as:

$$\mathrm{sim}_{3pole}(\theta_1, \theta_2) = \max(\cos^2((\theta_1 - \theta_2) / 2), \cos^2((\theta_1 - \theta_2 + 2\pi/3) / 2), \cos^2((\theta_1 - \theta_2 - 2\pi/3) / 2))$$

This function takes the maximum similarity across three possible "alignments" of the angles: a direct comparison $(\theta_1 - \theta_2)$, a comparison after shifting θ_2 by $2\pi/3$ (120 degrees), and a comparison after shifting θ_2 by $-2\pi/3$ (or equivalently, $4\pi/3$ or 240 degrees). This effectively introduces three "poles" of attraction on the circle, analogous to the primary colors. The similarity between two d-dimensional three-pole theta vectors is computed, as before, by averaging the similarities of their corresponding angles.

An alternative, and mathematically equivalent, formulation of the three-pole similarity can be achieved using complex numbers. Represent each angle θ as a complex number on the unit circle: $z = \exp(i\theta)$. Then, the three-pole similarity can be expressed as:

$$\mathrm{sim}_{3pole}(\theta_1, \theta_2) = |(z_1 * \mathrm{conj}(z_2))^3 + (\mathrm{conj}(z_1) * z_2)^3| / 2$$

Where `conj(z)` is the complex conjugate of z. This formulation highlights the rotational invariance of the three-pole similarity.

Hierarchical Theta Vectors:

To capture semantic information at multiple levels of granularity, we construct a hierarchical representation. At the top level (level 0), we have a set of *base_categories* theta vectors, representing fundamental semantic aspects (e.g., "what," "when," "how," "why"). Each subsequent level refines these categories. At level l, each of the *base_categories* vectors from level l-1 is expanded into 2^l sub-vectors. This creates a tree-like structure where each node represents a theta vector, and the children of a node represent more specific aspects of the concept represented by the parent.

The number of angular dimensions in each theta vector can vary across levels. A common strategy is to increase the dimensionality with each level, allowing for finer-grained distinctions at lower levels of the hierarchy. Alternatively, the dimensionality could be kept constant, or even decreased, depending on the specific application.

Model Architecture and Training

The hierarchical theta vector representation is incorporated into a neural network model for sentence embedding. The model consists of the following key components:

1. **Sentence Encoder:** A pre-trained Sentence Transformer (e.g., all-mpnet-base-v2) is used to obtain a dense vector representation of each input sentence. This provides a strong starting point, leveraging the knowledge encoded in the pre-trained model. The sentence encoder's weights are typically frozen during training to preserve its general-purpose knowledge.
2. **Angle Transformation Modules:** For each level of the hierarchy, a separate *angle transformation module* is employed. This module maps the output of the sentence encoder to a set of angles representing the theta vector at that level. Several options exist for the architecture of this module:
   - **Single Linear Layer:** The simplest approach is to use a single linear layer, mapping the sentence encoder's output directly to the desired number of angles. This is computationally efficient but may lack the capacity to capture complex non-linear relationships.
   - **Multi-Layer Perceptron (MLP):** A more powerful approach is to use an MLP with one or more hidden layers. This allows the model to learn more complex mappings from the sentence encoder's output to the angular space. The choice of activation function (ReLU, sigmoid, tanh) can influence the model's behavior.
   - **Custom Architectures:** More specialized architectures, such as convolutional or recurrent networks, could also be explored, particularly if the input text has specific structural properties.
3. **Hierarchical Grouping:** The outputs of the angle transformation modules are organized into a hierarchical structure, as described earlier. This is achieved through a recursive grouping process, resulting in nested lists of `AngularVector` (or `ThreePoleAngularVector`) instances.
4. **Similarity Computation:** To compare two sentence embeddings, a recursive similarity function is used. This function traverses the hierarchical structures of the two embeddings, computing the similarity between corresponding theta vectors at each level and averaging the results. The specific similarity function used depends on whether two-pole or three-pole theta vectors are employed.

Training Procedure:

The model is trained using a dataset of sentence pairs with associated similarity labels (e.g., 1 for similar pairs, -1 for dissimilar pairs). The training objective is to minimize a loss function that encourages similar sentences to have similar theta vector representations and dissimilar sentences to have dissimilar representations.

Several loss function options can be considered:

1. **Cosine Embedding Loss (Adapted):** The standard cosine embedding loss can be adapted to the angular setting by replacing the cosine similarity with the appropriate angular similarity function (either two-pole or three-pole).
2. **Margin Ranking Loss:** This loss function encourages the similarity score for a positive pair to be higher than the similarity score for a negative pair by a certain margin.
3. **Contrastive Loss:** This loss function pushes positive pairs closer together in the embedding space and negative pairs farther apart.
4. **Combined Similarity and Classification Loss:** To further guide the learning process, a *concept classification loss* can be incorporated. This involves training an auxiliary classifier to predict the base category (e.g., "what," "when," "how," "why") of each input sentence. The cross-entropy loss from this classifier is then combined with the similarity loss, providing a regularization signal that encourages the theta vectors to align with these fundamental semantic categories. This additional loss term can improve the interpretability and semantic coherence of the embeddings. The relative weighting of similarity loss and classification loss can be adjusted using a hyperparameter.

The model is trained using a standard optimization algorithm, such as Adam, with appropriate hyperparameter tuning (learning rate, batch size, etc.).

Alternative Similarity Measures and Distance Metrics While cosine similarity (adapted for angles) is a natural choice, other distance or similarity metrics could be used. For example, using a von-Mises distribution, to calculate the probability density of an angle. Or, defining a distance as the geodesic distance on the n-sphere.

Future Research Directions:

Several promising avenues for future research exist:

- **Exploring Different Similarity Functions:** Investigating alternative similarity functions for theta vectors, beyond the cosine-based approaches. This could involve exploring metrics from directional statistics or developing novel functions tailored to the three-pole representation.
- **Adaptive Hierarchy Depth:** Developing methods for dynamically determining the optimal depth of the hierarchy for a given input, rather than using a fixed maximum level.
- **Applications to Other NLP Tasks:** Applying theta vectors to a wider range of NLP tasks, such as text classification, question answering, and machine translation.
- **Theoretical Analysis:** Conducting a more in-depth theoretical analysis of the properties of theta vectors, including their representational capacity and generalization capabilities.
- **Connections to Cognitive Science:** Exploring potential connections between theta vectors and cognitive models of semantic representation.
- **Alternative Base Categories**: Instead of "what, when, where, why", exploring other options, such as using WordNet, or other ontologies.

Conclusion:

Angular embeddings offer a novel and promising approach to sentence representation. By embracing a circular topology and a hierarchical structure, they provide a bounded, interpretable, and potentially more robust alternative to traditional Cartesian embeddings. The introduction of three-pole theta vectors further enhances the representational capacity, allowing for the capture of more nuanced semantic relationships. While this paper has laid out the foundational concepts and a basic model architecture, numerous avenues for future research remain, promising to further unlock the potential of this approach. The inherent structure and interpretability of angular embeddings, combined with their potential for capturing complex semantic relationships, make them a compelling area for continued investigation.

Pseudocode examples

Code autogenerated with prompts.

```
# --- Fundamental Angle Handling Functions ---

FUNCTION ensure_angle_range(angles: TENSOR, two_pole: BOOLEAN = True) -> TENSOR:
    # Ensures angles are within the valid range.
    IF two_pole:
        RETURN angles % math.pi  # Modulo pi for [0, pi]
    ELSE:
        RETURN angles % (2 * math.pi)  # Modulo 2pi for [0, 2pi]

FUNCTION two_pole_similarity(theta1: TENSOR, theta2: TENSOR) -> TENSOR:
    # Computes the two-pole similarity between two angles (or tensors of angles).
    diff = (theta1 - theta2) / 2
    RETURN torch.cos(diff) ** 2

FUNCTION three_pole_similarity(theta1: TENSOR, theta2: TENSOR) -> TENSOR:
    # Computes the three-pole similarity.
    diff = (theta1 - theta2) / 2
    similarity1 = torch.cos(diff) ** 2
    similarity2 = torch.cos(diff + (math.pi / 3)) ** 2  # +120 degrees
    similarity3 = torch.cos(diff - (math.pi / 3)) ** 2  # -120 degrees
    RETURN torch.max(torch.stack([similarity1, similarity2, similarity3]), dim=0).values

FUNCTION two_pole_distance(theta1: TENSOR, theta2: TENSOR) -> TENSOR:
    # Angular distance (two-pole).  Takes the smaller angle between the two.
    diff = torch.abs(theta1 - theta2)
    RETURN torch.min(diff, math.pi - diff)

FUNCTION three_pole_distance(theta1: TENSOR, theta2: TENSOR) -> TENSOR:
    # Angular distance (three-pole).
    diff = torch.abs(theta1 - theta2)
    distance1 = torch.min(diff, 2 * math.pi - diff)
    distance2 = torch.min(torch.abs(theta1 - theta2 + 2*math.pi/3), 2*math.pi - torch.abs(theta1-theta2 + 2*math.pi/3) )
    distance3 = torch.min(torch.abs(theta1 - theta2 - 2*math.pi/3), 2*math.pi - torch.abs(theta1-theta2 - 2*math.pi/3))
    RETURN torch.min(torch.stack([distance1, distance2, distance3]), dim=0).values

FUNCTION circular_mean(angles: TENSOR) -> TENSOR:
    # Computes the circular mean of a tensor of angles.
    # Using the trigonometric method (more stable than complex number method for backpropagation).
    sum_sin = torch.sum(torch.sin(angles))
    sum_cos = torch.sum(torch.cos(angles))
    RETURN torch.atan2(sum_sin, sum_cos)

FUNCTION angular_addition(angle1: TENSOR, angle2: TENSOR, two_pole: BOOLEAN = True) -> TENSOR:
    # Adds two angles, ensuring the result is within the valid range.
    sum_angles = angle1 + angle2
    RETURN ensure_angle_range(sum_angles, two_pole)

FUNCTION angular_scalar_multiplication(angle: TENSOR, scalar: SCALAR, two_pole: BOOLEAN = True) -> TENSOR:
    # Multiplies an angle by a scalar, ensuring the result is within the valid range.
    product_angle = angle * scalar
    RETURN ensure_angle_range(product_angle, two_pole)

# --- Classes ---

CLASS TwoPoleAngularVector: #Added TwoPole Class
    # Represents a single vector using two-pole angles.
    ATTRIBUTES:
        angles: TENSOR  # Angles in radians

    METHODS:
        INIT(angles: TENSOR):
            # Constructor.  Ensures angles are in the correct range.
            SELF.angles = ensure_angle_range(angles, two_pole=True)

        SIMILARITY(other: TwoPoleAngularVector) -> TENSOR:
            # Calculates similarity between two-pole vectors.
            RETURN two_pole_similarity(SELF.angles, other.angles)

        DISTANCE(other: TwoPoleAngularVector) -> TENSOR:
            RETURN two_pole_distance(SELF.angles, other.angles)

        ADD(other: TwoPoleAngularVector) -> TwoPoleAngularVector:
            # Adds two angular vectors (wraps around).
            new_angles = angular_addition(SELF.angles, other.angles, two_pole=True)
            RETURN TwoPoleAngularVector(new_angles)

        SCALE(scalar: SCALAR) -> TwoPoleAngularVector:
          #Scales the vector
          new_angles = angular_scalar_multiplication(SELF.angles, scalar, two_pole=True)
          RETURN TwoPoleAngularVector(new_angles)

        __repr__(): #String representation

CLASS ThreePoleAngularVector: #Remains the same as before

    # Represents a single vector using three-pole angles.
    ATTRIBUTES:
        angles: TENSOR  # Angles in radians

    METHODS:
        INIT(angles: TENSOR):
            # Constructor.  Ensures angles are in the correct range.
            SELF.angles = ensure_angle_range(angles, two_pole=False)

        SIMILARITY(other: ThreePoleAngularVector) -> TENSOR:
            # Calculates similarity between two three-pole vectors.
            RETURN three_pole_similarity(SELF.angles, other.angles)

        DISTANCE(other: ThreePoleAngularVector) -> TENSOR:
            RETURN three_pole_distance(SELF.angles, other.angles)

        ADD(other: ThreePoleAngularVector) -> ThreePoleAngularVector:
            # Adds two angular vectors (wraps around).
            new_angles = angular_addition(SELF.angles, other.angles, two_pole=False)
            RETURN ThreePoleAngularVector(new_angles)

        SCALE(scalar: SCALAR) -> ThreePoleAngularVector:
          #Scales the vector
          new_angles = angular_scalar_multiplication(SELF.angles, scalar, two_pole=False)
          RETURN ThreePoleAngularVector(new_angles)

        __repr__(): #String representation


CLASS HierarchicalThetaEmbeddingModel:  # Renamed class - more generic
    ATTRIBUTES:
        tokenizer: PRETRAINED_TOKENIZER
        sentence_encoder: PRETRAINED_MODEL
        base_categories: INTEGER
        max_level: INTEGER
        sentence_limit: INTEGER
        angle_transformations: LIST of LAYERS
        pole_type: STRING  # "two_pole" or "three_pole"

    METHODS:
        INIT(pretrained_model_name, base_categories, max_level, sentence_limit, pole_type="three_pole", ...): #Added pole_type
            SELF.tokenizer = LOAD_TOKENIZER(pretrained_model_name)
            SELF.sentence_encoder = LOAD_MODEL(pretrained_model_name)
            SELF.base_categories = base_categories
            SELF.max_level = max_level
            SELF.sentence_limit = sentence_limit
            SELF.pole_type = pole_type #Store pole type

            SELF.angle_transformations = nn.ModuleList() #Use ModuleList for layers
            FOR level FROM 0 TO max_level:
                num_angles = base_categories * (2 ** level)
                SELF.angle_transformations.APPEND(nn.Linear(sentence_encoder.config.hidden_size, num_angles)) #Linear layer for

        FORWARD(input_text: STRING, max_level_override: INTEGER = None) -> LIST of LISTS of (TwoPoleAngularVector or ThreePoleA
            sentences = SPLIT_TEXT(input_text)
            all_level_embeddings = []
            angular_vector_class = ThreePoleAngularVector if SELF.pole_type == "three_pole" else TwoPoleAngularVector #Dynamica

            FOR level FROM 0 TO (max_level_override OR SELF.max_level):
                level_embeddings = []
                FOR sentence IN sentences:
                    sentence_embedding = ENCODE_SENTENCE(sentence)  # Using pretrained model
                    num_angles = SELF.base_categories * (2 ** level)
                    angles = TRANSFORM_TO_ANGLES(sentence_embedding, level, num_angles) #Output of the layer
                    angles = torch.sigmoid(angles) * (2 * math.pi) if SELF.pole_type == "three_pole" else torch.sigmoid(angles)
                    level_embeddings.APPEND(angular_vector_class(angles.squeeze())) #Use dynamic class

                all_level_embeddings.APPEND(level_embeddings)

            RETURN GROUP_EMBEDDINGS(all_level_embeddings)

        SPLIT_TEXT(text):
            #Split intelligently
            ...

        GROUP_EMBEDDINGS(all_level_embeddings):
            #Recursive grouping into arrays
            ...

        COMPUTE_SIMILARITY(embeddings1: LIST of LISTS of (TwoPoleAngularVector or ThreePoleAngularVector), embeddings2: LIST of
            # Recursively computes similarity between two hierarchies, now pole-type aware.
            FUNCTION recursive_similarity(group1, group2):
                IF isinstance(group1, (TwoPoleAngularVector, ThreePoleAngularVector)): #Check for both classes
                    RETURN group1.SIMILARITY(group2).mean()  # Use the class method - polymorphic dispatch

                similarities = []
                FOR subgroup1, subgroup2 IN zip(group1, group2):
                    similarities.APPEND(recursive_similarity(subgroup1, subgroup2))
                RETURN (SUM(similarities) / LENGTH(similarities)) IF LENGTH(similarities) > 0 ELSE 0.0

            total_similarity = 0.0
            total_weight = 0.0
            FOR level FROM 0 TO LENGTH(embeddings1) - 1:
                level_similarity = recursive_similarity(embeddings1[level][0], embeddings2[level][0])
                weight = 2 ** level
                total_similarity += level_similarity * weight
                total_weight += weight

            RETURN (total_similarity / total_weight) IF total_weight > 0 ELSE 0.0

        SAVE(path):
            #Saves parameters

        LOAD(path):
           #Load parameters


# --- Training Example (Conceptual) ---

# ... (Dataset loading, DataLoader setup) ...

# 1. Initialize Model and Optimizer
model = HierarchicalThetaEmbeddingModel(pretrained_model_name="all-mpnet-base-v2", base_categories=4, max_level=2, pole_type="t
optimizer = AdamW(model.parameters(), lr=1e-5) # Or other optimizer

# 2. Custom Loss Function (Conceptual - needs to be implemented based on your choice)
FUNCTION custom_loss(embeddings1, embeddings2, targets, inputs1=None, inputs2=None): #Added inputs as example for concept loss
    similarity_score = model.COMPUTE_SIMILARITY(embeddings1, embeddings2) #Use model's similarity function
    similarity_loss = 1.0 - similarity_score if targets == 1 else torch.relu(similarity_score - 0.5) #Example Contrastive-like

    # --- Optional: Concept Classification Loss (Example) ---
    IF inputs1 IS NOT None: #Example: if input text is available, could add concept classification
        # --- (Assume you have a way to get concept labels for inputs - e.g., from dataset) ---
        concept_labels1 = GET_CONCEPT_LABELS(inputs1) #Function to get labels - needs to be defined
        concept_labels2 = GET_CONCEPT_LABELS(inputs2)

        # --- (Assume you have a classifier to predict concepts from embeddings - needs to be implemented in model or separatel
        predicted_concepts1 = PREDICT_CONCEPTS(embeddings1) #Function to predict concepts
        predicted_concepts2 = PREDICT_CONCEPTS(embeddings2)

        concept_loss1 = CROSS_ENTROPY_LOSS(predicted_concepts1, concept_labels1)
        concept_loss2 = CROSS_ENTROPY_LOSS(predicted_concepts2, concept_labels2)
        total_concept_loss = concept_loss1 + concept_loss2
        combined_loss = similarity_loss + 0.1 * total_concept_loss #Weighted combination

        RETURN combined_loss
    ELSE:
        RETURN similarity_loss


# 3. Training Loop
num_epochs = 10
FOR epoch FROM 1 TO num_epochs:
    model.TRAIN() #Set to training mode
    FOR batch IN dataloader: #Iterate through your dataset
        inputs1_batch, inputs2_batch, targets_batch = batch #Assuming dataset returns pairs and similarity targets
        optimizer.ZERO_GRAD()

        embeddings1_batch = model.FORWARD(inputs1_batch)
        embeddings2_batch = model.FORWARD(inputs2_batch)

        loss = custom_loss(embeddings1_batch, embeddings2_batch, targets_batch, inputs1_batch, inputs2_batch) #Pass inputs for

        loss.BACKWARD()
        optimizer.STEP()

        PRINT(f"Epoch: {epoch}, Loss: {loss.item()}")

    # ... (Optional: Validation loop, saving checkpoints) ...


PRINT("Training Finished!")


# --- Inference Example ---

# 1. Load Trained Model
model = HierarchicalThetaEmbeddingModel.LOAD("path/to/saved/model")
model.EVAL()  # Set to evaluation mode

# 2. Input Sentence
input_sentence = "This is an example sentence for inference."

# 3. Get Hierarchical Theta Embeddings
embeddings = model.FORWARD(input_sentence)

# 4. Embeddings is a LIST of LISTS of (TwoPoleAngularVector or ThreePoleAngularVector)
#    Access and inspect the embeddings at different levels.
FOR level_idx, level_embedding_list IN enumerate(embeddings):
    print(f"Level {level_idx} Embeddings:")
    FOR sentence_embedding IN level_embedding_list:
        print(f"  Sentence Embedding Angles: {sentence_embedding.angles}") #Access angles via .angles attribute
```

Notes: the pseudo code is heavily incomplete. TwoPoleAngularVector and ThreePoleAngularVector would be a unique class, and the dynamic detail level management is absent.