Feature Group Embeddings: Comprehensive Guide

Aryan Dangi

December 18, 2024

Contents

1 Introduction

In machine learning, particularly when dealing with complex datasets, it's common to encounter features categorized or grouped based on certain attributes or "tags." Feature Group Embeddings transform these categorical tags into continuous numerical representations, enabling models to process them effectively. This guide explores various embedding methods, their implementations, advantages, and considerations, providing a comprehensive understanding to aid in selecting the most suitable approach for your models.

2 Understanding Tags and Tag Dimensions

2.1 Tags

- **Definition**: Tags are categorical labels assigned to features to denote their characteristics, categories, or groups. For example, in a dataset containing various sensor readings, tags might represent the type of sensor (temperature, humidity, pressure), the location of the sensor (indoor, outdoor), or the unit of measurement (Celsius, Fahrenheit).
- **Purpose**: Tags help in organizing features, enabling the model to understand relationships and similarities between different features based on their categories.

2.2 Tag Dimensions

- **Definition**: The tag dimension refers to the number of unique tags (categories) assigned to features. If there are K unique tags, the tag dimension is K.
- **Example**:
  - **SensorType**: Temperature, Humidity, Pressure (3 categories)
  - **Location**: Indoor, Outdoor (2 categories)
  - **Unit**: Celsius, Fahrenheit, Kelvin (3 categories)
  - **Total Unique Tags (K)**: $3 + 2 + 3 = 8$ tags

3 Why Use Embeddings for Tags?

3.1 Challenge with Categorical Data

Machine learning models, especially neural networks, perform best with numerical input. However, categorical data (like tags) are non-numeric and can't be directly fed into these models. Traditional methods like one-hot encoding can lead to high-dimensional and sparse representations, which are inefficient and may not capture the inherent relationships between categories.

3.2 Solution with Embeddings

Embeddings transform categorical tags into dense, continuous vectors of lower dimensionality. These embeddings capture semantic relationships between tags, allowing the model to learn similarities and differences effectively.

3.3 Advantages

- **Dimensionality Reduction**: Converts high-dimensional categorical data into manageable lower dimensions (e.g., 8 tags $\rightarrow$ 16-dimensional embedding space).
- **Capturing Relationships**: Similar tags can have similar embeddings, enabling the model to generalize better.

4 Feature Group Embeddings Methods

There are several methods to embed feature tags, each with its unique advantages and considerations. Below, we explore each method in detail.

4.1 Sum of Embeddings

4.1.1 Conceptual Overview

Sum of Embeddings involves assigning a unique embedding vector to each tag and summing the embeddings of all active tags for a feature. This method treats each tag independently and aggregates their contributions linearly.

4.1.2 Detailed Working

- **Tag Embedding Assignment**: Each unique tag is associated with a learnable embedding vector of fixed dimension (`embedding_dim`). For K unique tags, this results in an embedding matrix of size (K, `embedding_dim`).
- **Feature Representation**: For a feature with multiple tags, the binary tag vector indicates which tags are active. The embeddings corresponding to these active tags are retrieved and summed element-wise to produce the feature's embedding.
- **Mathematical Formulation**:

$$\text{TagEmbed}(f_i) = \sum_{j=1}^{K} T_i[j] \cdot E_j$$

  Where:

  - $T_i[j]$ is 1 if tag j is active for feature f_i, else 0.
  - E_j is the embedding vector for tag j.

4.1.3 Code Explanation

```python
import torch
import torch.nn as nn
import pandas as pd
import numpy as np

# Parameters
num_tags = 17
embedding_dim = 8
output_dim = 1
```

```
input_csv = '/kaggle/input/jane-street-real-time-market-data-forecasting/features.csv'  # Path to your input dataset
output_csv = '/kaggle/working/tag_embeddings.csv'  # Path to save the processed dataset

# Define the SumEmbedding class
class SumEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim):
        """
        Initializes the SumEmbedding layer.

        Args:
            num_tags (int): Total number of unique tags.
            embedding_dim (int): Dimension of each embedding vector.
        """
        super(SumEmbedding, self).__init__()
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)
        # Initialize embeddings with Xavier Uniform for better convergence
        nn.init.xavier_uniform_(self.tag_embeddings.weight)

    def forward(self, tag_vectors):
        """
        Computes the sum of embeddings for active tags.

        Args:
            tag_vectors (torch.Tensor): Binary tensor of shape (num_features, num_tags)

        Returns:
            torch.Tensor: Summed embeddings of shape (num_features, embedding_dim)
        """
        embeddings = self.tag_embeddings.weight  # (num_tags, embedding_dim)
        embedded = torch.matmul(tag_vectors, embeddings)  # (num_features, embedding_dim)
        return embedded

def preprocess_data(input_csv, output_csv, num_tags, embedding_dim):
    """
    Preprocesses the dataset by applying sum of embeddings to tag vectors.

    Args:
        input_csv (str): Path to the input CSV file.
        output_csv (str): Path to save the processed CSV file.
        num_tags (int): Number of unique tags.
        embedding_dim (int): Dimension of the embeddings.
    """
    # Load the dataset
    df = pd.read_csv(input_csv)

    # Identify tag columns
    tag_columns = [f'tag_{i}' for i in range(num_tags)]  # ['tag_0', 'tag_1', ..., 'tag_16']

    # Verify tag columns exist
```

```
    missing_tags = [tag for tag in tag_columns if tag not in df.columns]
    if missing_tags:
        raise ValueError(f"Missing tag columns in the dataset: {missing_tags}")

    # Initialize the embedding layer
    sum_embedding = SumEmbedding(num_tags, embedding_dim)

    # Set the embedding layer to evaluation mode to prevent gradient computation
    sum_embedding.eval()

    # Extract tag vectors for all features
    # Shape: (num_features, num_tags)
    tag_vectors = df[tag_columns].values  # numpy array

    # Convert to torch tensor
    tag_vectors_tensor = torch.tensor(tag_vectors, dtype=torch.float32)
    # (num_features, num_tags)

    # Compute summed embeddings
    with torch.no_grad():
        embeddings = sum_embedding(tag_vectors_tensor)  # (num_features, embedding_dim)

    # Convert embeddings to numpy
    embeddings_np = embeddings.numpy()  # (num_features, embedding_dim)

    # Add embedding columns to the dataframe
    for dim in range(embedding_dim):
        embedding_col_name = f'embed_{dim}'
        df[embedding_col_name] = embeddings_np[:, dim]

    # Optionally, drop original tag columns to reduce data size
    # Uncomment the following line if you wish to remove tag columns
    # df.drop(columns=tag_columns, inplace=True)

    # Save the processed dataset
    df.to_csv(output_csv, index=False)
    print(f"Processed data saved to {output_csv}")


preprocess_data(input_csv,output_csv,num_tags,embedding_dim)
```

Listing 1: Sum of Embeddings Implementation

- **Initialization (`__init__`)**:
  - `num_tags`: Total number of unique tags (K).
  - `embedding_dim`: Dimension of each embedding vector (d_e).
  - `self.tag_embeddings`: An embedding layer that maps each tag index to its embedding vector.
- **Forward Pass (`forward`)**:
  - `tag_vectors`: A binary matrix of shape (batch_size, num_tags), where each row represents the active tags for a feature.

- `self.tag_embeddings.weight`: Retrieves the embedding matrix of shape (num_tags, embedd
- `embedded = tag_vectors @ embeddings`: Performs a matrix multiplication between the binary tag vectors and the embedding matrix, resulting in a summed embedding for each feature in the batch. The resulting shape is (batch_size, embedding_dim).

4.1.4 Advantages and Considerations

Advantages:

- **Simplicity**: Easy to implement and computationally efficient.
- **Parameter Efficiency**: Requires only $K \times d_e$ parameters.
- **Flexibility**: Can handle features with multiple active tags seamlessly.

Considerations:

- **Additive Nature**: Assumes each tag contributes independently and additively to the final embedding, potentially missing complex interactions between tags.
- **Independence Assumption**: Does not model dependencies or interactions between tags beyond simple addition.

4.2 Linear Mapping (Concatenation via Learnable Weight Matrix)

4.2.1 Conceptual Overview

Linear Mapping uses a learnable weight matrix to linearly map the binary tag vectors into a continuous embedding space, allowing the model to learn optimal combinations of tags.

4.2.2 Detailed Working

- **Weight Matrix (`W_tags`)**: A learnable matrix of shape (`embedding_dim`, `num_tags`). Each column corresponds to a tag's contribution to each dimension of the embedding.
- **Feature Embedding**: For a given feature's binary tag vector, the embedding is computed as the linear combination of the active tags' columns in `W_tags`.
- **Mathematical Formulation**:

$$\text{TagEmbed}(f_i) = W_{\text{tags}} \cdot T_i^T$$

  Where:

  - T_i is the binary tag vector for feature f_i.
  - W_{tags} is the weight matrix of shape (`embedding_dim`, `num_tags`).

4.2.3 Code Explanation

```python
class LinearEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim):
        super(LinearEmbedding, self).__init__()
        self.W_tags = nn.Parameter(torch.randn(embedding_dim, num_tags))

    def forward(self, tag_vectors):
        """
        Args:
            tag_vectors (torch.Tensor): Binary tensor of shape (batch_size, num_tags)

        Returns:
            torch.Tensor: Linearly mapped embeddings of shape (batch_size, embedding_dim)
        """
        embedded = tag_vectors.float() @ self.W_tags.t()  # (batch_size, embedding_dim)
        return embedded
```

Listing 2: Linear Mapping Implementation

- **Initialization (`__init__`)**:
  - `self.W_tags`: A learnable weight matrix initialized with random values. Shape: (`embedding_dim`, `num_tags`).
- **Forward Pass (`forward`)**:
  - `tag_vectors`: Binary matrix of shape (batch_size, `num_tags`).
  - `tag_vectors.float()`: Converts the binary tensor to floating-point for matrix multiplication.
  - `self.W_tags.t()`: Transposes `W_tags` to shape (`num_tags`, `embedding_dim`) to align for multiplication.
  - `embedded = tag_vectors.float() @ self.W_tags.t()`: Multiplies the binary tag vectors with the transposed weight matrix, resulting in (batch_size, `embedding_dim`).

4.2.4 Advantages and Considerations

Advantages:

- **Learned Combinations**: Enables the model to learn how different tags interact and contribute to the embedding.
- **Expressiveness**: More capable of capturing complex relationships between tags compared to simple summation.

Considerations:

- **Increased Parameters**: Requires `embedding_dim` $\times K$ parameters, which can be substantial for large K.

- **Risk of Overfitting**: More parameters increase the likelihood of overfitting, especially with limited training data.
- **Computational Cost**: Higher memory and computation requirements compared to simpler methods like summation.

4.3 Multi-Head Embeddings

4.3.1 Conceptual Overview

Multi-Head Embeddings take inspiration from the multi-head attention mechanism in Transformers. Instead of having a single embedding vector per tag, each tag has multiple embedding "heads." This allows capturing different aspects or dimensions of each tag's representation.

4.3.2 Detailed Working

- **Multiple Embedding Heads**: Each tag is associated with `num_heads` separate embedding vectors. The final embedding is a concatenation or combination of these heads.
- **Representation Splitting**: The embedding dimension is divided equally among the heads. For example, with `embedding_dim` $= 64$ and `num_heads` $= 4$, each head has a dimension of 16.
- **Mathematical Formulation**:

$$E_{j,h} \in \mathbb{R}^{\text{head\_dim}}$$

  Where $E_{j,h}$ is the embedding vector for tag j in head h.

- The final embedding for a feature is a combination (e.g., concatenation) of embeddings from all heads.

4.3.3 Code Explanation

```
class MultiHeadEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, num_heads=4):
        super(MultiHeadEmbedding, self).__init__()
        assert embedding_dim % num_heads == 0, "embedding_dim must be
   divisible by num_heads"
        self.num_heads = num_heads
        self.head_dim = embedding_dim // num_heads
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)

    def forward(self, tag_vectors):
        embeddings = self.tag_embeddings.weight  # (num_tags,
   embedding_dim)
        embedded = tag_vectors @ embeddings      # (batch_size,
   embedding_dim)
        # Split into multiple heads
        embedded = embedded.view(-1, self.num_heads, self.head_dim)  # (
   batch_size, num_heads, head_dim)
```

```
14         return embedded
```

Listing 3: Multi-Head Embeddings Implementation

- **Initialization (`__init__`)**:
  - `num_heads`: Number of embedding heads (default is 4).
  - `self.head_dim`: Dimension of each head, ensuring `embedding_dim` is divisible by `num_heads`.
  - `self.tag_embeddings`: Embedding layer with shape (`num_tags`, `embedding_dim`).
- **Forward Pass (`forward`)**:
  - `embeddings = self.tag_embeddings.weight`: Retrieves embedding matrix (`num_tags`, `embedding_dim`).
  - `embedded = tag_vectors @ embeddings`: Multiplies binary tag vectors with embedding matrix to get (batch_size, `embedding_dim`).
  - `embedded.view(-1, self.num_heads, self.head_dim)`: Reshapes the embeddings to separate the heads, resulting in (batch_size, `num_heads`, `head_dim`).

Note: This implementation returns multiple heads separately. Further processing (like attention mechanisms) would typically operate on these heads.

4.3.4 Advantages and Considerations

Advantages:

- **Enhanced Representation**: Captures multiple facets or perspectives of each tag's meaning.
- **Parallel Learning**: Similar to multi-head attention, allowing the model to learn diverse aspects simultaneously.

Considerations:

- **Increased Complexity**: More complex architecture requires careful integration with downstream components.
- **Higher Computational Overhead**: More parameters and operations due to multiple heads.
- **Potential Underutilization**: If not properly leveraged, multiple heads might not provide significant benefits over single-head embeddings.

4.4 Interaction Embeddings

4.4.1 Conceptual Overview

Interaction Embeddings aim to explicitly model interactions between pairs (or higher-order combinations) of tags. This is similar to techniques used in factorization machines, where interactions between features are crucial for capturing dependencies.

4.4.2 Detailed Working

- **Pairwise Interactions**: For each pair of active tags in a feature, an interaction term is computed and incorporated into the final embedding.
- **Feature Combination**: The interactions can be modeled by additional parameters or by augmenting the embedding vectors.
- **Mathematical Formulation**:

$$\text{Interaction}_{jk} = E_j \odot E_k$$

Where $\odot$ denotes element-wise multiplication.

The final embedding could then be:

$$\text{TagEmbed}(f_i) = \sum_{j=1}^{K} T_i[j] \cdot E_j + \sum_{j=1}^{K} \sum_{k=j+1}^{K} T_i[j] \cdot T_i[k] \cdot \text{Interaction}_{jk}$$

4.4.3 Code Explanation

```
class InteractionEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, interaction_order=2):
        super(InteractionEmbedding, self).__init__()
        self.num_tags = num_tags
        self.embedding_dim = embedding_dim
        self.interaction_order = interaction_order
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)

    def forward(self, tag_vectors):
        embeddings = self.tag_embeddings.weight  # (num_tags, embedding_dim)
        embedded = tag_vectors @ embeddings      # (batch_size, embedding_dim)

        if self.interaction_order >= 2:
            # Pairwise interactions
            interactions = torch.matmul(tag_vectors, tag_vectors.t())  # Simplistic example
            # More sophisticated interaction modeling can be implemented
            # For illustration, we concatenate
            embedded = torch.cat([embedded, interactions], dim=1)

        return embedded
```

Listing 4: Interaction Embeddings Implementation

- **Initialization (`__init__`)**:
  - `interaction_order`: Determines the order of interactions to model (default is 2 for pairwise).
  - `self.tag_embeddings`: Embedding layer (`num_tags`, `embedding_dim`).
- **Forward Pass (`forward`)**:

- `embeddings = self.tag_embeddings.weight`:
  Retrieves (`num_tags`, `embedding_dim`).
- `embedded = tag_vectors @ embeddings`:
  Computes summed embeddings (batch_size, `embedding_dim`).
- **Pairwise Interactions**:
  - `interactions = torch.matmul(tag_vectors, tag_vectors.t())`: **Note**: This line attempts to compute interactions but is incorrectly implemented for batch-wise pairwise interactions. It incorrectly computes a batch-wise matrix multiplication across batches, resulting in (batch_size, batch_size).
  - `embedded = torch.cat([embedded, interactions], dim=1)`: Concatenates the original embedding with the interaction terms, increasing the embedding dimension.

Note: The provided code has an issue in computing pairwise interactions correctly. Here's a corrected version for pairwise interactions within each sample:

```
class InteractionEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, interaction_order=2):
        super(InteractionEmbedding, self).__init__()
        self.num_tags = num_tags
        self.embedding_dim = embedding_dim
        self.interaction_order = interaction_order
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)

    def forward(self, tag_vectors):
        batch_size = tag_vectors.size(0)
        embedded = tag_vectors @ self.tag_embeddings.weight  # (batch_size, embedding_dim)

        if self.interaction_order >= 2:
            # Compute pairwise interactions for each sample
            interaction_embeddings = []
            for i in range(batch_size):
                active_tags = torch.nonzero(tag_vectors[i], as_tuple=False)[:,1]
                if len(active_tags) < 2:
                    interaction = torch.zeros(self.embedding_dim, device=tag_vectors.device)
                else:
                    tag_embed = self.tag_embeddings(active_tags)  # (num_active_tags, embedding_dim)
                    # Compute outer product (sum of pairwise interactions)
                    interaction = torch.matmul(tag_embed.t(), tag_embed)  # (embedding_dim, embedding_dim)
                    interaction = interaction.sum(dim=1)  # Sum over interactions
                interaction_embeddings.append(interaction)
            interaction_embeddings = torch.stack(interaction_embeddings)  # (batch_size, embedding_dim)
            # Concatenate
            embedded = torch.cat([embedded, interaction_embeddings], dim=1)  # (batch_size, 2*embedding_dim)

```

```
        return embedded
```

Listing 5: Corrected Interaction Embeddings Implementation

4.4.4 Advantages and Considerations

Advantages:

- **Captures Interdependencies**: Explicitly models how pairs of tags interact, potentially uncovering complex relationships.
- **Improved Performance**: Can enhance model performance by leveraging tag dependencies.

Considerations:

- **Scalability**: Pairwise (or higher-order) interactions can lead to a combinatorial explosion of terms, especially with many tags.
- **Increased Dimensionality**: Concatenating interaction terms significantly increases the embedding size.
- **Implementation Complexity**: Requires careful handling to compute interactions efficiently, especially for large batches or many tags.
- **Potential Overfitting**: More parameters and higher-dimensional embeddings can lead to overfitting, especially with limited data.

4.5 Neural Network-Based Embeddings (MLP Approach)

4.5.1 Conceptual Overview

The **MLP Embeddings** approach employs a small neural network (typically a two-layer MLP) to transform the binary tag vectors into continuous embeddings. This allows capturing non-linear relationships between tags, offering more flexibility than linear methods.

4.5.2 Detailed Working

- **MLP Structure**: A sequence of linear layers with activation functions (e.g., ReLU) that learn to map the input tag vectors to embeddings.
- **Non-Linear Transformation**: By incorporating activation functions, the MLP can model complex, non-linear interactions between tags.
- **Mathematical Formulation**:

$$\text{TagEmbed}(f_i) = \text{MLP}(T_i)$$

  Where MLP consists of multiple layers with non-linear activations.

4.5.3 Code Explanation

```
class MLPEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, hidden_dim=32):
        super(MLPEmbedding, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(num_tags, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, embedding_dim)
        )

    def forward(self, tag_vectors):
        embedded = self.mlp(tag_vectors.float())  # (batch_size, embedding_dim)
        return embedded
```

Listing 6: MLP Embeddings Implementation

- **Initialization (`__init__`)**:
  - `hidden_dim`: Number of neurons in the hidden layer (default is 32).
  - `self.mlp`: A sequential model comprising:
    1. `nn.Linear(num_tags, hidden_dim)`: First linear layer mapping input tags to hidden dimension.
    2. `nn.ReLU()`: Non-linear activation function.
    3. `nn.Linear(hidden_dim, embedding_dim)`: Second linear layer mapping hidden dimension to final embedding.
- **Forward Pass (`forward`)**:
  - `tag_vectors`: Binary matrix (batch_size, `num_tags`).
  - `tag_vectors.float()`: Converts binary tensor to floating-point.
  - `self.mlp(tag_vectors.float())`: Passes through the MLP to obtain embeddings (batch_size, `embedding_dim`).

4.5.4 Advantages and Considerations

Advantages:

- **Non-Linear Relationships**: Capable of modeling complex, non-linear interactions between tags.
- **Flexibility**: The architecture can be adjusted (e.g., number of layers, hidden dimensions) to suit specific needs.
- **Expressiveness**: Can capture intricate patterns in the tag data, potentially leading to better performance.

Considerations:

- **Increased Parameters**: More parameters than linear methods, which can be problematic with large tag sets or limited data.

- **Risk of Overfitting**: Higher model complexity necessitates regularization techniques to prevent overfitting.
- **Computational Cost**: More computationally intensive due to multiple layers and non-linear operations.

4.6 Bilinear Embeddings

4.6.1 Conceptual Overview

Bilinear Embeddings use bilinear transformations to capture multiplicative interactions between tags and embedding dimensions. This method enhances the expressiveness of embeddings by allowing each tag to influence each dimension of the embedding space uniquely.

4.6.2 Detailed Working

- **Bilinear Transformation**: Each tag has its own transformation matrix that interacts with the embedding dimensions. The final embedding is a combination of these transformations.
- **Mathematical Formulation**:

$$\text{TagEmbed}(f_i) = T_i \cdot W$$

  Where:

  - T_i is the binary tag vector for feature f_i.
  - W is a bilinear weight matrix of shape (`num_tags`, `embedding_dim`).

4.6.3 Code Explanation

```
class BilinearEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim):
        super(BilinearEmbedding, self).__init__()
        self.W = nn.Parameter(torch.randn(num_tags, embedding_dim))

    def forward(self, tag_vectors):
        # Tag vectors: (batch_size, num_tags)
        # W: (num_tags, embedding_dim)
        embedded = tag_vectors.float() @ self.W  # (batch_size,
   embedding_dim)
        return embedded
```

Listing 7: Bilinear Embeddings Implementation

- **Initialization (`__init__`)**:
  - `self.W`: A learnable bilinear weight matrix initialized randomly. Shape: (`num_tags`, `embedding_dim`).
- **Forward Pass (`forward`)**:
  - `tag_vectors`: Binary matrix (batch_size, `num_tags`).

- `tag_vectors.float()`: Converts to floating-point.
- `embedded = tag_vectors.float() @ self.W`: Performs matrix multiplication to obtain embeddings (batch_size, `embedding_dim`).

Note: This implementation is similar to the **Linear Mapping** method but framed as a bilinear transformation, emphasizing the multiplicative interaction between tags and embedding dimensions.

4.6.4 Advantages and Considerations

Advantages:

- **Interaction Modeling**: Captures multiplicative interactions between tags and embedding dimensions, enhancing expressiveness.
- **Expressive Power**: More capable of representing complex relationships compared to additive or simple linear methods.

Considerations:

- **Parameter Count**: Requires `num_tags` $\times$ `embedding_dim` parameters, similar to linear mapping.
- **Training Complexity**: Bilinear transformations can be harder to train and may require careful initialization and optimization.
- **Overfitting Risk**: More parameters can lead to overfitting, especially with limited data.

4.7 Attention-Based Embeddings

4.7.1 Conceptual Overview

Attention-Based Embeddings incorporate an attention mechanism to dynamically weigh the importance of each tag for a given feature. This allows the model to focus more on relevant tags while diminishing the influence of less important ones.

4.7.2 Detailed Working

- **Attention Mechanism**: Computes attention weights for each tag based on the feature's context, then uses these weights to create a weighted sum of tag embeddings.
- **Dynamic Weighting**: Unlike static methods, attention-based embeddings adjust the importance of each tag during training, allowing for more nuanced representations.
- **Mathematical Formulation**:

$$\text{AttentionWeights} = \text{Softmax}(\text{Linear}(T_i \cdot E))$$

$$\text{TagEmbed}(f_i) = \sum_{j=1}^{K} \text{AttentionWeights}_j \cdot E_j$$

Where:

– E_j is the embedding for tag j.

– $\text{AttentionWeights}_j$ is the attention weight for tag j.

4.7.3 Code Explanation

```python
class AttentionEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, attention_dim=32):
        super(AttentionEmbedding, self).__init__()
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)
        self.attention = nn.Sequential(
            nn.Linear(embedding_dim, attention_dim),
            nn.Tanh(),
            nn.Linear(attention_dim, 1),
            nn.Softmax(dim=1)
        )

    def forward(self, tag_vectors):
        # tag_vectors: (batch_size, num_tags)
        embeddings = self.tag_embeddings.weight  # (num_tags, embedding_dim)
        embedded = tag_vectors @ embeddings      # (batch_size, embedding_dim)

        # Calculate attention weights
        attention_weights = self.attention(embedded)  # (batch_size, 1)
        # Weighted sum (here it's scalar attention, can be extended)
        weighted = embedded * attention_weights       # (batch_size, embedding_dim)
        return weighted
```

Listing 8: Attention-Based Embeddings Implementation

- **Initialization (`__init__`)**:
  - `attention_dim`: Dimension of the intermediate attention layer (default is 32).
  - `self.tag_embeddings`: Embedding layer (`num_tags`, `embedding_dim`).
  - `self.attention`: Sequential model comprising:
    1. `nn.Linear(embedding_dim, attention_dim)`: Projects embeddings to attention dimension.
    2. `nn.Tanh()`: Activation function.
    3. `nn.Linear(attention_dim, 1)`: Projects to a single attention score per tag.
    4. `nn.Softmax(dim=1)`: Normalizes attention scores across tags.
- **Forward Pass (`forward`)**:
  - `embeddings = self.tag_embeddings.weight`:
    Retrieves embedding matrix (`num_tags`, `embedding_dim`).
  - `embedded = tag_vectors @ embeddings`:
    Computes summed embeddings (batch_size, `embedding_dim`).
  - **Attention Weights**:

- `self.attention(embedded)`: Passes the summed embeddings through the attention network to obtain attention weights (batch_size, 1).
- `weighted = embedded * attention_weights`: Applies the attention weights to the embeddings. Since `attention_weights` is (batch_size, 1), broadcasting occurs across the embedding dimension.

Note: The current implementation applies a single scalar attention weight to the entire embedding vector. To implement per-tag attention, the code needs to be adjusted to compute attention weights for each tag individually.

4.7.4 Correct Implementation for Per-Tag Attention

```
class AttentionEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, attention_dim=32):
        super(AttentionEmbedding, self).__init__()
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)
        self.attention = nn.Sequential(
            nn.Linear(embedding_dim, attention_dim),
            nn.Tanh(),
            nn.Linear(attention_dim, 1)
        )

    def forward(self, tag_vectors):
        batch_size = tag_vectors.size(0)
        num_tags = tag_vectors.size(1)
        embeddings = self.tag_embeddings.weight  # (num_tags, embedding_dim)
        embeddings = embeddings.unsqueeze(0).expand(batch_size, num_tags, -1)  # (batch_size, num_tags, embedding_dim)
        tag_vectors = tag_vectors.unsqueeze(2).float()  # (batch_size, num_tags, 1)
        # Mask inactive tags
        masked_embeddings = embeddings * tag_vectors  # (batch_size, num_tags, embedding_dim)
        # Compute attention scores for each tag
        attention_scores = self.attention(masked_embeddings.view(-1, embeddings.size(-1)))  # (batch_size*num_tags, 1)
        attention_scores = attention_scores.view(batch_size, num_tags)  # (batch_size, num_tags)
        # Apply mask to zero out inactive tags
        attention_scores = attention_scores * tag_vectors.squeeze(2)  # (batch_size, num_tags)
        # Re-normalize attention scores
        attention_weights = nn.functional.softmax(attention_scores, dim=1)  # (batch_size, num_tags)
        attention_weights = attention_weights.unsqueeze(2)  # (batch_size, num_tags, 1)
        # Weighted sum of embeddings
        weighted = (masked_embeddings * attention_weights).sum(dim=1)  # (batch_size, embedding_dim)
        return weighted
```

Listing 9: Corrected Attention-Based Embeddings Implementation

4.7.5 Advantages and Considerations

Advantages:

- **Dynamic Weighting**: Adjusts the influence of each tag based on the feature's context.
- **Focus on Relevant Tags**: Enhances the embedding by emphasizing more important tags.
- **Improved Performance**: Potentially leads to better representations by prioritizing significant tags.

Considerations:

- **Increased Complexity**: More layers and operations, making the model more complex.
- **Potential Training Instability**: Attention mechanisms can be sensitive to hyperparameters and may require careful tuning.
- **Computational Overhead**: Additional computations for calculating attention weights.

4.8 Hierarchical Embeddings

4.8.1 Conceptual Overview

Hierarchical Embeddings are designed for scenarios where tags have an inherent hierarchical structure (e.g., categories and subcategories). This method leverages the hierarchy to inform the embedding process, allowing the model to capture relationships at different levels of granularity.

4.8.2 Detailed Working

- **Hierarchy Structure**: Tags are organized into parent-child relationships, where parent tags encompass one or more child tags.
- **Embedding Incorporation**: The embeddings of child tags are combined with their parent tags' embeddings, allowing the model to inherit hierarchical information.
- **Mathematical Formulation**:

$$E'_c = E_c + E_p$$

  Where E_c is the original embedding of the child tag and E_p is the embedding of the parent tag.

4.8.3 Code Explanation

```python
class HierarchicalEmbedding(nn.Module):
    def __init__(self, num_tags, embedding_dim, hierarchy):
        """
        Args:
            num_tags (int): Total number of unique tags.
            embedding_dim (int): Dimension of the embedding vectors.
            hierarchy (dict): Dictionary defining parent-child relationships among tags.
                              e.g., {parent_tag_idx: [child_tag_idx1, child_tag_idx2, ...], ...}
        """
        super(HierarchicalEmbedding, self).__init__()
        self.tag_embeddings = nn.Embedding(num_tags, embedding_dim)
        self.hierarchy = hierarchy  # e.g., {parent_tag: [child_tags]}

    def forward(self, tag_vectors):
        batch_size = tag_vectors.size(0)
        embeddings = self.tag_embeddings.weight      # (num_tags, embedding_dim)
        combined = tag_vectors @ embeddings         # (batch_size, embedding_dim)

        # Incorporate hierarchical information
        for parent, children in self.hierarchy.items():
            parent_embedding = self.tag_embeddings(torch.tensor(parent, device=tag_vectors.device))  # (embedding_dim,)
            children_tensor = torch.tensor(children, device=tag_vectors.device)  # (num_children,)
            children_active = tag_vectors[:, children].float()  # (batch_size, num_children)
            children_sum = children_active.sum(dim=1)          # (batch_size,)
            combined += children_sum.unsqueeze(1) * parent_embedding  # (batch_size, embedding_dim)
        return combined
```

Listing 10: Hierarchical Embeddings Implementation

- **Initialization (`__init__`)**:
  - `hierarchy`: A dictionary mapping parent tag indices to lists of child tag indices.
  - `self.tag_embeddings`: Embedding layer (`num_tags`, `embedding_dim`).
- **Forward Pass (`forward`)**:
  - `tag_vectors`: Binary matrix (batch_size, `num_tags`).
  - `embeddings = self.tag_embeddings.weight`:
    Retrieves embedding matrix.
  - `combined = tag_vectors @ embeddings`:
    Computes summed embeddings (batch_size, `embedding_dim`).
  - **Hierarchical Information Incorporation**:

    - Iterates over each parent and its children.
    - `parent_embedding`: Retrieves the embedding for the parent tag.
    - `children_tensor`: Converts child tag indices to a tensor.
    - `children_active`: Extracts the active child tags for each sample (batch_size, `num_children`).
    - `children_sum`: Sums the active child tags for each sample (batch_size,).
    - `combined += children_sum.unsqueeze(1) * parent_embedding`: Adds the parent embedding scaled by the number of active child tags to the combined embeddings.

Note: This implementation assumes that for each active child tag, the parent tag's embedding should be added once. Depending on the specific hierarchy and desired behavior, alternative methods of combining parent and child embeddings might be more appropriate.

4.8.4 Advantages and Considerations

Advantages:

- **Captures Structured Relationships**: Leverages hierarchical structures to inform embeddings, enhancing representational power.
- **Improved Generalization**: Hierarchical information can help in better generalizing to unseen tag combinations.
- **Hierarchical Consistency**: Ensures that child tags inherit information from their parent categories.

Considerations:

- **Requires Hierarchical Structure**: Only applicable if tags have a meaningful hierarchy.
- **Implementation Complexity**: More involved to implement, especially with deep or complex hierarchies.
- **Potential Redundancy**: If not managed carefully, adding parent embeddings might lead to redundant information.

4.9 Pretrained Embeddings

4.9.1 9.1 Conceptual Overview

Pretrained Embeddings leverage embeddings learned from external sources or larger datasets. If such embeddings exist for your tags, they can be incorporated into your model to provide a head start, especially beneficial when training data is limited.

4.9.2 Detailed Working

- **Pretrained Embedding Matrix**: A precomputed embedding matrix, often obtained from models trained on similar tasks or large-scale data.
- **Integration**: The pretrained embeddings are loaded into the embedding layer, optionally allowing them to be fine-tuned during model training.
- **Mathematical Formulation**:

$$\text{TagEmbed}(f_i) = \text{PretrainedE}_j$$

Where PretrainedE_j is the pretrained embedding for tag j.

4.9.3 Code Explanation

```
class PretrainedEmbedding(nn.Module):
    def __init__(self, pretrained_embeddings):
        """
        Args:
            pretrained_embeddings (torch.Tensor): Pretrained embedding matrix of shape (num_tags, embedding_dim).
        """
        super(PretrainedEmbedding, self).__init__()
        self.tag_embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=False)

    def forward(self, tag_vectors):
        # tag_vectors: (batch_size, num_tags)
        # W_tags: (num_tags, embedding_dim)
        embedded = tag_vectors.float() @ self.tag_embeddings.weight  # (batch_size, embedding_dim)
        return embedded
```

Listing 11: Pretrained Embeddings Implementation

- **Initialization (`__init__`)**:
  - `pretrained_embeddings`: A tensor containing pretrained embeddings of shape (`num_tags`, `embedding_dim`).
  - `self.tag_embeddings = nn.Embedding.from_pretrained(...)`: Initializes the embedding layer with the pretrained embeddings. `freeze=False` allows these embeddings to be fine-tuned during training.
- **Forward Pass (`forward`)**:
  - `tag_vectors`: Binary matrix (batch_size, `num_tags`).
  - `self.tag_embeddings.weight`: Retrieves the pretrained embedding matrix.
  - `embedded = tag_vectors.float() @ self.tag_embeddings.weight`: Multiplies binary tag vectors with the embedding matrix to obtain summed embeddings (batch_size, `embedding_dim`).

4.9.4 Advantages and Considerations

Advantages:

- **Leverage External Knowledge**: Utilizes embeddings trained on larger or domain-specific datasets, potentially enhancing performance.
- **Faster Convergence**: Provides a good initialization, leading to faster training and better generalization.
- **Improved Representations**: Pretrained embeddings often capture semantic relationships that can benefit downstream tasks.

Considerations:

- **Availability**: Pretrained embeddings might not be available for all types of tags or specific domains.
- **Compatibility**: Ensure that the dimensions and semantics of pretrained embeddings align with your model's requirements.
- **Fine-Tuning**: Deciding whether to freeze the embeddings or allow them to be fine-tuned is crucial and depends on the size of your dataset and similarity to the pretrained data.

5 Choosing the Right Embedding Method

Selecting the appropriate embedding method depends on several factors, including the nature of your tags, the complexity of their relationships, computational resources, and the specific requirements of your model. Here's a guide to help you decide:

1. **Simplicity vs. Expressiveness**:
   - **Simple Methods**: Start with **Sum of Embeddings** or **Linear Mapping** for their simplicity and efficiency.
   - **Complex Relationships**: If tags have intricate interactions, consider **Interaction Embeddings** or **Attention-Based Embeddings**.
2. **Data Availability**:
   - **Limited Data**: Simpler methods like **Sum of Embeddings** are less prone to overfitting.
   - **Abundant Data**: More complex methods can be explored to capture deeper relationships.
3. **Hierarchy and Structure**:
   - If your tags are hierarchical, **Hierarchical Embeddings** can capture structured relationships effectively.
4. **Pretrained Resources**:

- Utilize **Pretrained Embeddings** if relevant embeddings are available to incorporate external knowledge.

5. **Computational Resources**:
   - Consider the computational cost of each method. Simpler methods are generally faster and require less memory.

6. **Hybrid Approaches**:
   - Combine multiple embedding techniques, such as using both **Sum of Embeddings** and **Attention-Based Embeddings**, to leverage their respective strengths.

6 Integrating Embeddings into the TimeXer Model

To effectively incorporate feature group embeddings into the `TimeXer` model, follow these steps:

1. **Embed Feature Tags**: Use the selected embedding method to convert binary tag vectors into continuous embeddings.

2. **Combine with Feature Data**: Integrate the tag embeddings with the original feature representations. This can be done via concatenation or addition.

3. **Modify the TimeXer Model**: Adjust the input dimensions of the `TimeXer` model to accommodate the additional embedding dimensions.

6.1 Example Integration Using Sum of Embeddings

```
import torch
import torch.nn as nn

class TimeXerWithSumEmbedding(nn.Module):
    def __init__(self, input_dim, projected_dim, hidden_dim, num_layers, output_dim, num_tags, embedding_dim):
        super(TimeXerWithSumEmbedding, self).__init__()
        self.tag_embedding = SumEmbedding(num_tags, embedding_dim)
        # Adjust input dimension to include embedding dimension
        self.projection = nn.Linear(input_dim + embedding_dim, projected_dim)
        self.dropout = nn.Dropout(p=0.1)
        self.transformer_layer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=projected_dim, nhead=4, dim_feedforward=hidden_dim, batch_first=True),
            num_layers=num_layers
        )
        self.fc = nn.Linear(projected_dim, output_dim)

    def forward(self, x, tag_vectors):
        # Embed tags
        embedded_tags = self.tag_embedding(tag_vectors)  # (batch_size, embedding_dim)
        # Expand embedded_tags to match sequence length
```

```
        embedded_tags = embedded_tags.unsqueeze(1).repeat(1, x.size(1),
    1)  # (batch_size, seq_len, embedding_dim)
        # Concatenate with input features
        x = torch.cat([x, embedded_tags], dim=-1)  # (batch_size,
    seq_len, input_dim + embedding_dim)
        # Proceed with TimeXer layers
        x = self.projection(x)  # (batch_size, seq_len, projected_dim)
        x = self.dropout(x)
        x = self.transformer_layer(x)
        x = self.fc(x)
        return x
```

Listing 12: TimeXer Integration with Sum of Embeddings

6.1.1 Explanation

1. **Embedding Layer Initialization**:
   - `self.tag_embedding = SumEmbedding(num_tags, embedding_dim)`: Initializes the tag embedding layer using the **Sum of Embeddings** method.
2. **Adjusting Input Dimensions**:
   - `self.projection = nn.Linear(input_dim + embedding_dim, projected_dim)`: Concatenates the original feature dimensions with the embedding dimensions to align with the transformer's expected input.
3. **Forward Pass Steps**:
   1. **Embedding Tags**: Converts binary tag vectors into continuous embeddings.
   2. **Broadcasting**: Expands the tag embeddings across the sequence length to match the temporal dimension.
   3. **Concatenation**: Merges the original features with the tag embeddings, enriching the feature representation.
   4. **Projection and Transformation**: Passes the combined features through the projection layer, dropout, transformer encoder, and finally the fully connected output layer.

6.1.2 Usage Example

```
# Define model parameters
input_dim = 10         # Number of original feature dimensions
projected_dim = 64     # Embedding size for transformer
hidden_dim = 128       # Hidden layer size in transformer
num_layers = 3         # Number of transformer layers
output_dim = 1         # Output dimension (e.g., regression)
num_tags = 17          # Number of unique tags
embedding_dim = 16     # Embedding dimension for tags

# Instantiate the model
model = TimeXerWithSumEmbedding(input_dim, projected_dim, hidden_dim,
    num_layers, output_dim, num_tags, embedding_dim)

```

```python
# Example input:
batch_size = 32
seq_length = 50
x = torch.randn(batch_size, seq_length, input_dim)            # Original features
tag_vectors = torch.randint(0, 2, (batch_size, num_tags)).float()  # Binary tag vectors

# Forward pass
output = model(x, tag_vectors)

print(output.shape)  # Expected: (32, 50, 1)
```

Listing 13: TimeXerWithSumEmbedding Usage Example

7 Final Recommendations and Best Practices

Choosing the right embedding method depends on your specific use case, dataset characteristics, and computational resources. Here are some guidelines to help you decide:

1. **Start Simple**:
   - **Sum of Embeddings**: Given its simplicity and effectiveness, especially with a manageable number of tags (17), this method is a good starting point.
   - **Linear Mapping**: If you require more expressiveness than summation offers, consider linear mapping.
2. **Evaluate Model Performance**:
   - Compare the performance of different embedding methods using validation metrics.
   - Monitor for overfitting, especially with more complex methods like **Linear Mapping** or **MLP Embeddings**.
3. **Consider Alternative Methods if Needed**:
   - If initial experiments with **Sum of Embeddings** do not yield satisfactory performance, consider exploring **Linear Mapping** or **MLP Embeddings** as next steps, keeping in mind the trade-offs in complexity and overfitting risks.
4. **Regularization Techniques**:
   - **Dropout**: Apply dropout to embedding layers and downstream layers to prevent overfitting.
   - **L2 Regularization**: Implement L2 regularization on embedding weights to encourage smaller weights and reduce overfitting.
5. **Hyperparameter Tuning**:
   - **Embedding Dimension (d_e)**: Experiment with different embedding dimensions (e.g., 16, 32) to find the optimal balance between expressiveness and computational efficiency.

- **Hidden Dimensions**: If using **MLP Embeddings**, tune the hidden dimensions to capture sufficient complexity without overfitting.

6. **Visualization**:
   - Use techniques like PCA or t-SNE to visualize learned embeddings, ensuring that they capture meaningful distinctions between tags despite their anonymity.
7. **Scalability Considerations**:
   - While 17 tags are manageable, anticipate potential increases in tag numbers. Ensure that your embedding method can scale accordingly without significant performance degradation.
8. **Integration with Downstream Layers**:
   - Ensure that the combined feature and embedding dimensions align with the expectations of downstream layers, such as transformers or other neural network components.
9. **Experimentation**:
   - Experiment with multiple embedding methods and evaluate their impact on model performance. Use cross-validation and ablation studies to guide your selection.

8 Conclusion

Feature Group Embeddings are pivotal in transforming categorical tag data into meaningful continuous representations that enhance machine learning models' performance. Among the various methods, **Sum of Embeddings** stands out for its simplicity, efficiency, and effectiveness, especially in scenarios with a manageable number of anonymized tags and features. By adhering to best practices and methodically evaluating each approach, you can optimize your models to leverage categorical information effectively, leading to more accurate and robust predictions.

Appendix: Summary of Embedding Methods

Method	Description	Pros & Cons
Sum of Embeddings	Sum the embeddings of active tags	**Pros**: Simple, efficient, parameter-light **Cons**: May miss complex tag interactions
Linear Mapping	Learn a weight matrix to map tag vectors	**Pros**: Captures interactions, more expressive **Cons**: More parameters, risk of overfitting
Multi-Head Embeddings	Multiple embeddings per tag to capture different aspects	**Pros**: Enhanced representation, parallel learning **Cons**: Increased complexity and computation
Interaction Embeddings	Explicitly model interactions between tags	**Pros**: Captures dependencies, improved performance **Cons**: Scalability issues with many tags, complex implementation
MLP Embeddings	Use a neural network to non-linearly transform tag vectors	**Pros**: Captures non-linear relationships, flexible **Cons**: More parameters, higher risk of overfitting
Bilinear Embeddings	Use bilinear transformations to model multiplicative interactions	**Pros**: Captures multiplicative interactions **Cons**: Increased parameters and complexity
Attention-Based Embeddings	Apply attention mechanisms to weigh tag importance	**Pros**: Dynamic weighting, focuses on relevant tags **Cons**: Adds computational overhead, training complexity
Hierarchical Embeddings	Leverage hierarchical relationships among tags	**Pros**: Captures structured relationships **Cons**: Requires hierarchical tag structure, complex implementation
Pretrained Embeddings	Utilize pretrained embeddings if available	**Pros**: Leverages external knowledge, potential performance boost **Cons**: Limited applicability, requires compatible pretrained embeddings

Table 1: Summary of Embedding Methods

References

1. PyTorch Documentation
2. Understanding Embeddings
3. Transformers and Attention Mechanisms