Integration of Feature Group Embeddings into the TimeXer Model for Enhanced Time Series Forecasting

Aryan Dangi

September 22, 2025

Abstract

This report presents the comprehensive integration of feature group embeddings into the TimeXer model to enhance time series forecasting performance. The project encompasses data preprocessing to handle missing values, generation of unique feature tag embeddings capturing semantic relationships, and the adaptation of the TimeXer architecture to incorporate exogenous variables without merging them with raw data. The implementation ensures flexibility in forecasting across varying test data sizes and leverages a weighted loss function to optimize model performance based on sample weights. Detailed explanations of the theoretical foundations, code structure, and functional components are provided to facilitate understanding and reproducibility.

Contents

1 Introduction

Time series forecasting is a critical task in various domains, including finance, weather prediction, supply chain management, and healthcare. Accurate forecasts enable informed decision-making, resource allocation, and strategic planning. Traditional models like ARIMA and exponential smoothing have been widely used; however, the advent of deep learning has introduced sophisticated architectures that capture complex temporal dependencies and feature interactions.

The **TimeXer** model is one such advanced architecture designed to leverage both endogenous (primary time series data) and exogenous (external features) variables for enhanced forecasting accuracy. This project focuses on integrating feature group embeddings as exogenous variables into the TimeXer model, ensuring that the model effectively captures semantic relationships between features without introducing inconsistencies or biases, especially when handling missing data and varying test data sizes.

2 Theoretical Background

2.1 Time Series Forecasting

Time series forecasting involves predicting future values based on previously observed values. The primary challenge lies in capturing temporal dependencies, trends, and seasonality inherent in the data. Accurate forecasting models can adapt to various patterns and incorporate external information to improve predictions.

2.2 TimeXer Model

The **TimeXer** model is an encoder-based Transformer architecture tailored for time series forecasting. It processes endogenous data through a series of Transformer encoder layers, capturing intricate temporal patterns. The model also integrates exogenous variables, which provide additional context and improve forecasting performance by incorporating external influences.

Key components of the TimeXer model include:

- **Embedding Layers**: Project raw features into a higher-dimensional space to capture complex relationships.
- **Transformer Encoder**: Utilizes self-attention mechanisms to model dependencies within the data.
- **Cross-Attention**: Integrates exogenous embeddings, allowing the model to consider external context.
- **Forecasting Head**: Generates the final forecast based on the encoded representations.

2.3 Feature Group Embeddings

Feature Group Embeddings are dense vector representations that encapsulate semantic relationships and categorical information about features. By assigning unique embeddings to each feature, models can leverage the inherent relationships between features, enhancing their ability to capture complex dependencies and improve forecasting accuracy.

2.4 Handling Missing Values

Missing data is a common challenge in real-world datasets. Proper handling of missing values is crucial to prevent biases and ensure the model's robustness. Techniques include imputation, assigning default values, or incorporating missing value indicators as additional features to inform the model about the presence of missing data.

2.5 Weighted Loss Functions

Incorporating sample weights into the loss function allows the model to prioritize certain samples over others during training. This is particularly useful when dealing with imbalanced data or when certain samples are deemed more reliable or important. A weighted loss function scales the contribution of each sample's loss based on its weight, guiding the model's learning process accordingly.

3 Project Overview

This project involves the following key steps:

a) **Data Preprocessing**: Handling missing values by adding 'missing_feature_value' columns and preparing data for model consumption.

b) **Feature Tag Embeddings Generation**: Creating unique embeddings for each feature to capture semantic relationships.

c) **TimeXer Model Adaptation**: Modifying the TimeXer architecture to incorporate exogenous embeddings as separate inputs.

d) **Training Pipeline Setup**: Implementing data loaders, defining the training loop, and optimizing the model using a weighted Huber loss function.

e) **Model Saving**: Persisting the trained model for future inference.

4 Implementation Details

4.1 Data Loading and Preprocessing

The raw data is partitioned and stored in parquet files. Each partition contains features, target variables, and indicators for missing values. The 'sequential_data_loader' function iterates through these partitions, loading the data and yielding essential components for training.

4.2 Exogenous Embeddings Integration

Feature tag embeddings are loaded from a CSV file, ensuring alignment with the used features. Missing embeddings are handled by assigning zero vectors to maintain consistency. These embeddings are then projected and fed into the model as exogenous inputs, preserving their uniqueness without any averaging or modification.

4.3 Model Architecture

The TimeXer model is defined with separate projection layers for endogenous features and exogenous embeddings. The projected embeddings are concatenated and passed through Transformer encoder layers, which include self-attention and cross-attention mechanisms. The model concludes with a forecasting head that generates the final predictions.

4.4 Training Loop

The training loop encompasses data loading, model training, validation, and optimization. A custom weighted Huber loss function is employed to incorporate sample weights from the 'weight' column, enhancing the model's performance evaluation. The training process includes learning rate scheduling and gradient clipping to ensure stable convergence.

4.5 Model Saving

Upon completion of training, the model's state dictionary and feature names are saved for future inference tasks.

5 Code Description and Functionality

Below is a detailed description of each component in the implementation, including classes, functions, and their respective functionalities.

5.1 Custom Weighted Huber Loss

```python
class WeightedHuberLoss(nn.Module):
    def __init__(self, delta=1.0):
        """
        Initializes the Weighted Huber Loss.

        Args:
            delta (float): The point where the loss function changes from quadratic to linear.
        """
        super(WeightedHuberLoss, self).__init__()
        self.delta = delta

    def forward(self, y_pred, y_true, weights):
        """
        Computes the weighted Huber loss.

        Args:
            y_pred (Tensor): Predicted values (B,).
```

```
            y_true (Tensor): True target values (B,).
            weights (Tensor): Sample weights (B,).

        Returns:
            Tensor: The computed weighted Huber loss.
        """
        error = y_pred - y_true
        abs_error = torch.abs(error)
        quadratic = torch.min(abs_error, torch.tensor(self.delta).to(
    y_pred.device))
        linear = abs_error - quadratic
        loss = 0.5 * quadratic ** 2 + self.delta * linear
        return (loss * weights).mean()
```

Listing 1: WeightedHuberLoss Class

Description: This class implements a weighted version of the Huber loss function. The Huber loss is less sensitive to outliers in data than the squared error loss. By incorporating weights, the loss function can prioritize certain samples over others based on their importance or reliability.

5.2 Exogenous Embeddings Loader

```
def load_exogenous_embeddings(embedding_path, used_feature_names):
    """
    Loads and aligns feature tag embeddings with the used feature names
    .

    Args:
        embedding_path (str): Path to the feature tag embeddings CSV
    file.
        used_feature_names (list): List of feature names used in the
    model.

    Returns:
        Tensor: Exogenous embeddings tensor of shape (num_features,
    exog_dim).
    """
    embeddings_df = pd.read_csv(embedding_path)
    # Ensure all used features have embeddings
    missing_embeddings = set(used_feature_names) - set(embeddings_df['
    feature'].tolist())
    if missing_embeddings:
        print(f"Features missing embeddings: {missing_embeddings}")
        # Assign zero embeddings to missing features
        num_embed_dims = embeddings_df.shape[1] - 1  # Excluding '
    feature' column
        for feature in missing_embeddings:
            embeddings_df = embeddings_df.append({
                'feature': feature,
                **{f'embed_{i}': 0.0 for i in range(num_embed_dims)}
            }, ignore_index=True)
        print("Assigned zero embeddings to missing features.")
    else:
        print("All features have corresponding embeddings.")

    # Align embeddings with used features
```

```
    embeddings_df.set_index('feature', inplace=True)
    embeddings_aligned = embeddings_df.loc[used_feature_names].reset_index(drop=True)

    # Convert to tensor
    exog_embeddings = embeddings_aligned.values.astype(np.float32)  # (num_features, exog_dim)
    exog_embeddings_tensor = torch.tensor(exog_embeddings, dtype=torch.float32)  # (num_features, exog_dim)
    return exog_embeddings_tensor  # Shape: (num_features, exog_dim)
```

Listing 2: load_exogenous_embeddings Function

Description: This function loads feature tag embeddings from a CSV file and aligns them with the features used in the model. If any features lack corresponding embeddings, zero vectors are assigned to maintain alignment and prevent dimension mismatches.

5.3 Sequential Data Loader

```
def sequential_data_loader(data_dir):
    """
    Generator that yields data partitions for training.

    Args:
        data_dir (str): Directory containing partitioned data.

    Yields:
        tuple: (DataFrame, used_feature_names, target_name)
    """
    for p_id in range(3, 10):  # Loop through directories partition_id=3 to partition_id=9
        file_path = os.path.join(data_dir, f'partition_id={p_id}', 'part_0.parquet')
        if not os.path.isfile(file_path):
            print(f"File not found: {file_path}")
            continue

        print(f"Loading file: partition_id={p_id}")
        df = pl.read_parquet(file_path).to_pandas()

        # Define columns to exclude
        exclude_cols = ['date_id', 'time_id', 'responder_6', 'weight',
                        'date_id_missing', 'time_id_missing',
                        'responder_6_missing', 'weight_missing']

        used_feature_names = [col for col in df.columns if col not in exclude_cols]
        target_name = 'responder_6'

        # Yield necessary data
        yield df, used_feature_names, target_name

        # Clean up
        del df, used_feature_names, target_name
        gc.collect()
```

Listing 3: sequential_data_loader Function

Description: This generator function iterates through data partitions stored in parquet files, loading each partition and yielding the DataFrame, list of used feature names, and the target variable name. It ensures efficient memory usage by processing data partition by partition.

5.4 TimeSeriesDataset Class

```python
class TimeSeriesDataset(Dataset):
    def __init__(self, data, used_feature_names, target_name, exog_embeddings):
        """
        Initializes the TimeSeriesDataset.

        Args:
            data (DataFrame): The preprocessed data.
            used_feature_names (list): List of feature names used as inputs.
            target_name (str): Name of the target variable.
            exog_embeddings (Tensor): Exogenous embeddings tensor of shape (num_features, exog_dim).
        """
        self.features = data[used_feature_names].values  # (num_samples, num_features)
        self.targets = data[target_name].values         # (num_samples,)
        self.weights = data['weight'].values            # (num_samples,)
        self.exog_embeddings = exog_embeddings          # (num_features, exog_dim)

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        """
        Retrieves a single data point.

        Args:
            idx (int): Index of the data point.

        Returns:
            tuple: (features, exog_embeddings, target, weight)
        """
        x = self.features[idx]                              # (num_features,)
        y = self.targets[idx]                               # scalar
        w = self.weights[idx]                               # scalar
        exog = self.exog_embeddings                         # (num_features, exog_dim)
        return torch.tensor(x, dtype=torch.float32), exog, torch.tensor(y, dtype=torch.float32), torch.tensor(w, dtype=torch.float32)
```

Listing 4: TimeSeriesDataset Class

Description: This custom PyTorch ‘Dataset‘ class prepares data for training by providing features, exogenous embeddings, targets, and weights for each sample. It ensures that the model receives both endogenous and exogenous inputs separately.

5.5 TimeXer Model Definition

```python
class TimeXer(nn.Module):
    def __init__(self, input_dim, exog_dim, projected_dim, hidden_dim, num_layers, output_dim, dropout=0.1):
        """
        Initializes the TimeXer model.

        Args:
            input_dim (int): Number of endogenous features.
            exog_dim (int): Dimension of each feature's exogenous embedding.
            projected_dim (int): Dimension after projection.
            hidden_dim (int): Transformer hidden dimension.
            num_layers (int): Number of Transformer layers.
            output_dim (int): Forecast horizon (number of future time steps).
            dropout (float): Dropout rate.
        """
        super(TimeXer, self).__init__()
        # Projection for endogenous features
        self.feature_projection = nn.Linear(input_dim, projected_dim)
        # Projection for exogenous embeddings
        self.exog_projection = nn.Linear(exog_dim, projected_dim)
        # Dropout
        self.dropout = nn.Dropout(p=dropout)
        # Transformer Encoder
        self.encoder = Encoder(
            [
                EncoderLayer(
                    AttentionLayer(
                        FullAttention(False, 4, attention_dropout=dropout, output_attention=False),
                        projected_dim, 4
                    ),
                    AttentionLayer(
                        FullAttention(False, 4, attention_dropout=dropout, output_attention=False),
                        projected_dim, 4
                    ),
                    projected_dim,
                    hidden_dim,
                    dropout=dropout,
                    activation="relu",
                )
                for _ in range(num_layers)
            ],
            norm_layer=nn.LayerNorm(projected_dim)
        )
        # Head to produce forecast
        self.head_nf = projected_dim * (input_dim + 1)  # Adjust based on concatenation
        self.head = FlattenHead(n_vars=1, nf=self.head_nf, target_window=output_dim, head_dropout=dropout)

    def forward(self, x, exog):
        """
        Forward pass of the TimeXer model.
```

```

        Args:
            x (Tensor): Endogenous features tensor of shape (B, input_dim).
            exog (Tensor): Exogenous embeddings tensor of shape (num_features, exog_dim).

        Returns:
            Tensor: Forecasted values tensor of shape (B, output_dim).
        """
        # Project endogenous features
        x_proj = self.feature_projection(x)  # (B, projected_dim)
        x_proj = self.dropout(x_proj)        # (B, projected_dim)

        # Project exogenous embeddings
        exog_proj = self.exog_projection(exog)  # (num_features, projected_dim)
        exog_proj = self.dropout(exog_proj)      # (num_features, projected_dim)

        # Expand exog_proj to match batch size
        exog_proj = exog_proj.unsqueeze(0).repeat(x.size(0), 1, 1)  # (B, num_features, projected_dim)

        # Concatenate endogenous and exogenous projections as a sequence
        # Sequence length = 1 (endogenous) + num_features (exogenous)
        x_proj_seq = x_proj.unsqueeze(1)  # (B, 1, projected_dim)
        combined = torch.cat([x_proj_seq, exog_proj], dim=1)  # (B, 1 + num_features, projected_dim)

        # Pass through Transformer Encoder
        enc_out = self.encoder(combined, exog_proj)  # (B, 1 + num_features, projected_dim)

        # Pass through head to get forecast
        dec_out = self.head(enc_out)  # (B, n_vars, output_dim)
        dec_out = dec_out.permute(0, 2, 1)  # (B, output_dim, n_vars)
        dec_out = dec_out.squeeze(-1)  # (B, output_dim)

        return dec_out
```

Listing 5: TimeXer Class

Description: The ‘TimeXer‘ class defines the model architecture, incorporating projections for both endogenous features and exogenous embeddings. It utilizes a Transformer encoder to process the concatenated projections and employs a forecasting head to generate predictions.

5.6 FlattenHead Class

```
class FlattenHead(nn.Module):
    def __init__(self, n_vars, nf, target_window, head_dropout=0.0):
        """
        Initializes the FlattenHead.

        Args:
```

```
            n_vars (int): Number of variables (features).
            nf (int): Number of features after projection.
            target_window (int): Number of future time steps to
forecast.
            head_dropout (float): Dropout rate.
        """
        super(FlattenHead, self).__init__()
        self.n_vars = n_vars
        self.flatten = nn.Flatten(start_dim=-2)
        self.linear = nn.Linear(nf, target_window)
        self.dropout = nn.Dropout(head_dropout)

    def forward(self, x):  # x: [bs x nvars x d_model x patch_num]
        """
        Forward pass of FlattenHead.

        Args:
            x (Tensor): Encoder output tensor.

        Returns:
            Tensor: Forecasted values tensor.
        """
        x = self.flatten(x)  # [bs x nvars * d_model * patch_num]
        x = self.linear(x)   # [bs x target_window]
        x = self.dropout(x)
        return x
```

Listing 6: FlattenHead Class

Description: The 'FlattenHead' class processes the encoder's output by flattening the last two dimensions and applying a linear transformation followed by dropout. This generates the final forecasted values.

5.7 Encoder and EncoderLayer Classes

```
class Encoder(nn.Module):
    def __init__(self, layers, norm_layer=None, projection=None):
        """
        Initializes the Encoder.

        Args:
            layers (list): List of EncoderLayer instances.
            norm_layer (nn.Module): Normalization layer.
            projection (nn.Module): Projection layer.
        """
        super(Encoder, self).__init__()
        self.layers = nn.ModuleList(layers)
        self.norm = norm_layer
        self.projection = projection

    def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None,
    delta=None):
        """
        Forward pass of the Encoder.

        Args:
            x (Tensor): Input tensor.
```

```
            cross (Tensor): Cross-attention tensor.
            x_mask (Tensor, optional): Mask for x.
            cross_mask (Tensor, optional): Mask for cross.
            tau (Tensor, optional): Additional parameter.
            delta (Tensor, optional): Additional parameter.

        Returns:
            Tensor: Encoded output tensor.
        """
        for layer in self.layers:
            x = layer(x, cross, x_mask=x_mask, cross_mask=cross_mask, tau=tau, delta=delta)

        if self.norm is not None:
            x = self.norm(x)

        if self.projection is not None:
            x = self.projection(x)
        return x

class EncoderLayer(nn.Module):
    def __init__(self, self_attention, cross_attention, d_model, d_ff=None,
                 dropout=0.1, activation="relu"):
        """
        Initializes the EncoderLayer.

        Args:
            self_attention (AttentionLayer): Self-attention layer.
            cross_attention (AttentionLayer): Cross-attention layer.
            d_model (int): Model dimension.
            d_ff (int, optional): Feedforward dimension.
            dropout (float): Dropout rate.
            activation (str): Activation function.
        """
        super(EncoderLayer, self).__init__()
        d_ff = d_ff or 4 * d_model
        self.self_attention = self_attention
        self.cross_attention = cross_attention
        self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model, kernel_size=1)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = F.relu if activation == "relu" else F.gelu

    def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None, delta=None):
        """
        Forward pass of the EncoderLayer.

        Args:
            x (Tensor): Input tensor.
            cross (Tensor): Cross-attention tensor.
            x_mask (Tensor, optional): Mask for x.
```

```
            cross_mask (Tensor, optional): Mask for cross.
            tau (Tensor, optional): Additional parameter.
            delta (Tensor, optional): Additional parameter.

        Returns:
            Tensor: Output tensor after encoder layer.
        """
        B, L, D = cross.shape
        # Self-attention
        x = x + self.dropout(self.self_attention(
            x, x, x,
            attn_mask=x_mask,
            tau=tau, delta=None
        )[0])
        x = self.norm1(x)

        # Cross-attention
        x_glb_ori = x[:, -1, :].unsqueeze(1)  # (B, 1, D)
        x_glb = torch.reshape(x_glb_ori, (B, -1, D))  # (B, 1, D)
        x_glb_attn = self.dropout(self.cross_attention(
            x_glb, cross, cross,
            attn_mask=cross_mask,
            tau=tau, delta=delta
        )[0])
        x_glb_attn = torch.reshape(x_glb_attn,
                                   (x_glb_attn.shape[0] * x_glb_attn.shape[1], x_glb_attn.shape[2])).unsqueeze(1)
        x_glb = x_glb_ori + x_glb_attn
        x_glb = self.norm2(x_glb)

        # Combine
        y = x = torch.cat([x[:, :-1, :], x_glb], dim=1)  # (B, L, D)

        # Feedforward
        y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))))  # (B, D, L)
        y = self.dropout(self.conv2(y).transpose(-1, 1))  # (B, L, D)

        return self.norm3(x + y)
```

Listing 7: Encoder and EncoderLayer Classes

Description: The ‘Encoder‘ class stacks multiple ‘EncoderLayer‘ instances to form the Transformer encoder. Each ‘EncoderLayer‘ consists of self-attention, cross-attention, and feedforward convolutional layers, interleaved with normalization and dropout to ensure stable and efficient training.

5.8 Training Loop

```
def train_model(data_dir, embedding_path, model, optimizer, criterion, epochs=10, batch_size=64, validation_split=0.2):
    """
    Trains the TimeXer model.

    Args:
        data_dir (str): Directory containing partitioned training data.
```

```
        embedding_path (str): Path to the feature tag embeddings CSV file.
        model (nn.Module): The TimeXer model to train.
        optimizer (torch.optim.Optimizer): Optimizer for training.
        criterion (nn.Module): Loss function.
        epochs (int): Number of training epochs.
        batch_size (int): Training batch size.
        validation_split (float): Fraction of data to use for validation.
    """
    used_feature_names = []
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)

    # Learning rate scheduler
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

    for epoch in range(epochs):
        print(f"Epoch [{epoch+1}/{epochs}]")
        epoch_loss = 0.0
        val_loss = 0.0

        # Iterate through all data partitions
        for df, feature_names, target_name in sequential_data_loader(data_dir):
            if not used_feature_names:
                used_feature_names = feature_names
                # Load exogenous embeddings for used features
                exog_embeddings_tensor = load_exogenous_embeddings(embedding_path, used_feature_names)

            # Create Dataset
            dataset = TimeSeriesDataset(df, feature_names, target_name, exog_embeddings_tensor)
            val_size = int(len(dataset) * validation_split)
            train_size = len(dataset) - val_size

            # Split into training and validation datasets
            train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
            train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
            val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
            train_progress = tqdm(train_loader, desc="Training Progress", unit="batch", leave=True, dynamic_ncols=True, mininterval=0)

            # Training Phase
            model.train()
            for X_batch, exog_batch, y_batch, w_batch in train_progress:
                X_batch = X_batch.to(device)          # (B, input_dim)
                y_batch = y_batch.to(device)          # (B,)
                w_batch = w_batch.to(device)          # (B,)
                exog_batch = exog_batch.to(device)    # (num_features, exog_dim)
```

```python

            optimizer.zero_grad()
            outputs = model(X_batch, exog_batch)   # (B, output_dim)
            loss = criterion(outputs.view(-1), y_batch, w_batch)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            epoch_loss += loss.item()

            # Update tqdm progress bar
            train_progress.set_postfix({"loss": f"{loss.item():.4f}"})
        train_progress.close()

        # Validation Phase
        model.eval()
        val_progress = tqdm(val_loader, desc="Validation Progress", unit="batch")
        with torch.no_grad():
            for X_batch, exog_batch, y_batch, w_batch in val_progress:
                X_batch = X_batch.to(device)
                y_batch = y_batch.to(device)
                w_batch = w_batch.to(device)
                exog_batch = exog_batch.to(device)

                outputs = model(X_batch, exog_batch)
                loss = criterion(outputs.view(-1), y_batch, w_batch)
                val_loss += loss.item()

        # Print epoch losses
        print(f"Processed partition with training loss: {epoch_loss / len(train_loader):.4f}, validation loss: {val_loss / len(val_loader):.4f}")

    # Step the scheduler
    scheduler.step(val_loss)

    # Print average losses for the epoch
    print(f"Epoch {epoch+1} completed. Average training loss: {epoch_loss / len(train_loader):.4f}, validation loss: {val_loss / len(val_loader):.4f}")
    print()

# Save the trained model
print("Training completed.")
torch.save({'model_state_dict': model.state_dict(), 'feature_names': used_feature_names}, "/kaggle/working/timexer_model.pth")
print("Model saved to timexer_model.pth")
```

Listing 8: train_model Function

Description: The ‘train_model‘ function orchestrates the entire training process. It iterates through data partitions, loads and aligns exogenous embeddings, splits data into training and validation sets, and performs the training and validation phases for each epoch. The function also incorporates a learning rate scheduler and saves the trained model upon completion.

6 Complete Training Script

Below is the complete training script that integrates all components discussed above. This script is self-contained and ready for execution, assuming the necessary data files and custom layers are in place.

```python
# Import necessary libraries
import os
import gc
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, random_split
import polars as pl
import pandas as pd
import numpy as np
from tqdm.notebook import tqdm

# Import custom layers - ensure these modules are present in the '
    layers' directory
from layers.SelfAttention_Family import FullAttention, AttentionLayer
from layers.Embed import DataEmbedding_inverted, PositionalEmbedding

# ================================
# Custom Weighted Huber Loss
# ================================
class WeightedHuberLoss(nn.Module):
    def __init__(self, delta=1.0):
        """
        Initializes the Weighted Huber Loss.

        Args:
            delta (float): The point where the loss function changes
    from quadratic to linear.
        """
        super(WeightedHuberLoss, self).__init__()
        self.delta = delta

    def forward(self, y_pred, y_true, weights):
        """
        Computes the weighted Huber loss.

        Args:
            y_pred (Tensor): Predicted values (B,).
            y_true (Tensor): True target values (B,).
            weights (Tensor): Sample weights (B,).

        Returns:
            Tensor: The computed weighted Huber loss.
        """
        error = y_pred - y_true
        abs_error = torch.abs(error)
```

```
        quadratic = torch.min(abs_error, torch.tensor(self.delta).to(y_pred.device))
        linear = abs_error - quadratic
        loss = 0.5 * quadratic ** 2 + self.delta * linear
        return (loss * weights).mean()

# ================================
# Exogenous Embeddings Loader
# ================================
def load_exogenous_embeddings(embedding_path, used_feature_names):
    """
    Loads and aligns feature tag embeddings with the used feature names.

    Args:
        embedding_path (str): Path to the feature tag embeddings CSV file.
        used_feature_names (list): List of feature names used in the model.

    Returns:
        Tensor: Exogenous embeddings tensor of shape (num_features, exog_dim).
    """
    embeddings_df = pd.read_csv(embedding_path)
    # Ensure all used features have embeddings
    missing_embeddings = set(used_feature_names) - set(embeddings_df['feature'].tolist())
    if missing_embeddings:
        print(f"Features missing embeddings: {missing_embeddings}")
        # Assign zero embeddings to missing features
        num_embed_dims = embeddings_df.shape[1] - 1  # Excluding 'feature' column
        for feature in missing_embeddings:
            embeddings_df = embeddings_df.append({
                'feature': feature,
                **{f'embed_{i}': 0.0 for i in range(num_embed_dims)}
            }, ignore_index=True)
        print("Assigned zero embeddings to missing features.")
    else:
        print("All features have corresponding embeddings.")

    # Align embeddings with used features
    embeddings_df.set_index('feature', inplace=True)
    embeddings_aligned = embeddings_df.loc[used_feature_names].reset_index(drop=True)

    # Convert to tensor
    exog_embeddings = embeddings_aligned.values.astype(np.float32)  # (num_features, exog_dim)
    exog_embeddings_tensor = torch.tensor(exog_embeddings, dtype=torch.float32)  # (num_features, exog_dim)
    return exog_embeddings_tensor  # Shape: (num_features, exog_dim)

# ================================
# Sequential Data Loader
# ================================
def sequential_data_loader(data_dir):
```

```
    """
    Generator that yields data partitions for training.

    Args:
        data_dir (str): Directory containing partitioned data.

    Yields:
        tuple: (DataFrame, used_feature_names, target_name)
    """
    for p_id in range(3, 10):  # Loop through directories partition_id=3 to partition_id=9
        file_path = os.path.join(data_dir, f'partition_id={p_id}', 'part_0.parquet')
        if not os.path.isfile(file_path):
            print(f"File not found: {file_path}")
            continue

        print(f"Loading file: partition_id={p_id}")
        df = pl.read_parquet(file_path).to_pandas()

        # Define columns to exclude
        exclude_cols = ['date_id', 'time_id', 'responder_6', 'weight',
                        'date_id_missing', 'time_id_missing',
                        'responder_6_missing', 'weight_missing']

        used_feature_names = [col for col in df.columns if col not in exclude_cols]
        target_name = 'responder_6'

        # Yield necessary data
        yield df, used_feature_names, target_name

        # Clean up
        del df, used_feature_names, target_name
        gc.collect()

# ================================
# TimeSeries Dataset Class
# ================================
class TimeSeriesDataset(Dataset):
    def __init__(self, data, used_feature_names, target_name, exog_embeddings):
        """
        Initializes the TimeSeriesDataset.

        Args:
            data (DataFrame): The preprocessed data.
            used_feature_names (list): List of feature names used as inputs.
            target_name (str): Name of the target variable.
            exog_embeddings (Tensor): Exogenous embeddings tensor of shape (num_features, exog_dim).
        """
        self.features = data[used_feature_names].values  # (num_samples, num_features)
        self.targets = data[target_name].values         # (num_samples,)
        self.weights = data['weight'].values            # (num_samples,)
```

```python
        self.exog_embeddings = exog_embeddings        # (num_features, exog_dim)

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        """
        Retrieves a single data point.

        Args:
            idx (int): Index of the data point.

        Returns:
            tuple: (features, exog_embeddings, target, weight)
        """
        x = self.features[idx]                              # (num_features,)
        y = self.targets[idx]                               # scalar
        w = self.weights[idx]                               # scalar
        exog = self.exog_embeddings                         # (num_features, exog_dim)
        return torch.tensor(x, dtype=torch.float32), exog, torch.tensor(y, dtype=torch.float32), torch.tensor(w, dtype=torch.float32)

# ================================
# TimeXer Model Definition
# ================================
class TimeXer(nn.Module):
    def __init__(self, input_dim, exog_dim, projected_dim, hidden_dim, num_layers, output_dim, dropout=0.1):
        """
        Initializes the TimeXer model.

        Args:
            input_dim (int): Number of endogenous features.
            exog_dim (int): Dimension of each feature's exogenous embedding.
            projected_dim (int): Dimension after projection.
            hidden_dim (int): Transformer hidden dimension.
            num_layers (int): Number of Transformer layers.
            output_dim (int): Forecast horizon (number of future time steps).
            dropout (float): Dropout rate.
        """
        super(TimeXer, self).__init__()
        # Projection for endogenous features
        self.feature_projection = nn.Linear(input_dim, projected_dim)
        # Projection for exogenous embeddings
        self.exog_projection = nn.Linear(exog_dim, projected_dim)
        # Dropout
        self.dropout = nn.Dropout(p=dropout)
        # Transformer Encoder
        self.encoder = Encoder(
            [
                EncoderLayer(
                    AttentionLayer(
```

```
                        FullAttention(False, 4, attention_dropout=dropout, output_attention=False),
                        projected_dim, 4
                    ),
                    AttentionLayer(
                        FullAttention(False, 4, attention_dropout=dropout, output_attention=False),
                        projected_dim, 4
                    ),
                    projected_dim,
                    hidden_dim,
                    dropout=dropout,
                    activation="relu",
                )
                for _ in range(num_layers)
            ],
            norm_layer=nn.LayerNorm(projected_dim)
        )
        # Head to produce forecast
        self.head_nf = projected_dim * (input_dim + 1)  # Adjust based on concatenation
        self.head = FlattenHead(n_vars=1, nf=self.head_nf, target_window=output_dim, head_dropout=dropout)

    def forward(self, x, exog):
        """
        Forward pass of the TimeXer model.

        Args:
            x (Tensor): Endogenous features tensor of shape (B, input_dim).
            exog (Tensor): Exogenous embeddings tensor of shape (num_features, exog_dim).

        Returns:
            Tensor: Forecasted values tensor of shape (B, output_dim).
        """
        # Project endogenous features
        x_proj = self.feature_projection(x)  # (B, projected_dim)
        x_proj = self.dropout(x_proj)        # (B, projected_dim)

        # Project exogenous embeddings
        exog_proj = self.exog_projection(exog)  # (num_features, projected_dim)
        exog_proj = self.dropout(exog_proj)      # (num_features, projected_dim)

        # Expand exog_proj to match batch size
        exog_proj = exog_proj.unsqueeze(0).repeat(x.size(0), 1, 1)  # (B, num_features, projected_dim)

        # Concatenate endogenous and exogenous projections as a sequence
        # Sequence length = 1 (endogenous) + num_features (exogenous)
        x_proj_seq = x_proj.unsqueeze(1)  # (B, 1, projected_dim)
        combined = torch.cat([x_proj_seq, exog_proj], dim=1)  # (B, 1 + num_features, projected_dim)

```

```
        # Pass through Transformer Encoder
        enc_out = self.encoder(combined, exog_proj)  # (B, 1 + num_features, projected_dim)

        # Pass through head to get forecast
        dec_out = self.head(enc_out)  # (B, n_vars, output_dim)
        dec_out = dec_out.permute(0, 2, 1)  # (B, output_dim, n_vars)
        dec_out = dec_out.squeeze(-1)  # (B, output_dim)

        return dec_out

# ================================
# FlattenHead Class
# ================================
class FlattenHead(nn.Module):
    def __init__(self, n_vars, nf, target_window, head_dropout=0.0):
        """
        Initializes the FlattenHead.

        Args:
            n_vars (int): Number of variables (features).
            nf (int): Number of features after projection.
            target_window (int): Number of future time steps to forecast.
            head_dropout (float): Dropout rate.
        """
        super(FlattenHead, self).__init__()
        self.n_vars = n_vars
        self.flatten = nn.Flatten(start_dim=-2)
        self.linear = nn.Linear(nf, target_window)
        self.dropout = nn.Dropout(head_dropout)

    def forward(self, x):  # x: [bs x nvars x d_model x patch_num]
        """
        Forward pass of FlattenHead.

        Args:
            x (Tensor): Encoder output tensor.

        Returns:
            Tensor: Forecasted values tensor.
        """
        x = self.flatten(x)  # [bs x nvars * d_model * patch_num]
        x = self.linear(x)   # [bs x target_window]
        x = self.dropout(x)
        return x

# ================================
# Encoder and EncoderLayer Definitions
# ================================
class Encoder(nn.Module):
    def __init__(self, layers, norm_layer=None, projection=None):
        """
        Initializes the Encoder.

        Args:
            layers (list): List of EncoderLayer instances.
            norm_layer (nn.Module): Normalization layer.
```

```
            projection (nn.Module): Projection layer.
        """
        super(Encoder, self).__init__()
        self.layers = nn.ModuleList(layers)
        self.norm = norm_layer
        self.projection = projection

    def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None, delta=None):
        """
        Forward pass of the Encoder.

        Args:
            x (Tensor): Input tensor.
            cross (Tensor): Cross-attention tensor.
            x_mask (Tensor, optional): Mask for x.
            cross_mask (Tensor, optional): Mask for cross.
            tau (Tensor, optional): Additional parameter.
            delta (Tensor, optional): Additional parameter.

        Returns:
            Tensor: Encoded output tensor.
        """
        for layer in self.layers:
            x = layer(x, cross, x_mask=x_mask, cross_mask=cross_mask, tau=tau, delta=delta)

        if self.norm is not None:
            x = self.norm(x)

        if self.projection is not None:
            x = self.projection(x)
        return x

class EncoderLayer(nn.Module):
    def __init__(self, self_attention, cross_attention, d_model, d_ff=None,
                 dropout=0.1, activation="relu"):
        """
        Initializes the EncoderLayer.

        Args:
            self_attention (AttentionLayer): Self-attention layer.
            cross_attention (AttentionLayer): Cross-attention layer.
            d_model (int): Model dimension.
            d_ff (int, optional): Feedforward dimension.
            dropout (float): Dropout rate.
            activation (str): Activation function.
        """
        super(EncoderLayer, self).__init__()
        d_ff = d_ff or 4 * d_model
        self.self_attention = self_attention
        self.cross_attention = cross_attention
        self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model, kernel_size=1)
        self.norm1 = nn.LayerNorm(d_model)
```

```python
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = F.relu if activation == "relu" else F.gelu

    def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None, delta=None):
        """
        Forward pass of the EncoderLayer.

        Args:
            x (Tensor): Input tensor.
            cross (Tensor): Cross-attention tensor.
            x_mask (Tensor, optional): Mask for x.
            cross_mask (Tensor, optional): Mask for cross.
            tau (Tensor, optional): Additional parameter.
            delta (Tensor, optional): Additional parameter.

        Returns:
            Tensor: Output tensor after encoder layer.
        """
        B, L, D = cross.shape
        # Self-attention
        x = x + self.dropout(self.self_attention(
            x, x, x,
            attn_mask=x_mask,
            tau=tau, delta=None
        )[0])
        x = self.norm1(x)

        # Cross-attention
        x_glb_ori = x[:, -1, :].unsqueeze(1)  # (B, 1, D)
        x_glb = torch.reshape(x_glb_ori, (B, -1, D))  # (B, 1, D)
        x_glb_attn = self.dropout(self.cross_attention(
            x_glb, cross, cross,
            attn_mask=cross_mask,
            tau=tau, delta=delta
        )[0])
        x_glb_attn = torch.reshape(x_glb_attn,
                                   (x_glb_attn.shape[0] * x_glb_attn.shape[1], x_glb_attn.shape[2])).unsqueeze(1)
        x_glb = x_glb_ori + x_glb_attn
        x_glb = self.norm2(x_glb)

        # Combine
        y = x = torch.cat([x[:, :-1, :], x_glb], dim=1)  # (B, L, D)

        # Feedforward
        y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))))  # (B, D, L)
        y = self.dropout(self.conv2(y).transpose(-1, 1))  # (B, L, D)

        return self.norm3(x + y)
```

Listing 9: Complete Training Script

Description: The ‘Encoder‘ class stacks multiple ‘EncoderLayer‘ instances, each comprising self-attention and cross-attention mechanisms, followed by feedforward convolutional layers. These layers facilitate the modeling of complex dependencies within the data and the integration of exogenous embeddings.

6.1 Training Loop

```python
def train_model(data_dir, embedding_path, model, optimizer, criterion, epochs=10, batch_size=64, validation_split=0.2):
    """
    Trains the TimeXer model.

    Args:
        data_dir (str): Directory containing partitioned training data.
        embedding_path (str): Path to the feature tag embeddings CSV file.
        model (nn.Module): The TimeXer model to train.
        optimizer (torch.optim.Optimizer): Optimizer for training.
        criterion (nn.Module): Loss function.
        epochs (int): Number of training epochs.
        batch_size (int): Training batch size.
        validation_split (float): Fraction of data to use for validation.
    """
    used_feature_names = []
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)

    # Learning rate scheduler
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

    for epoch in range(epochs):
        print(f"Epoch [{epoch+1}/{epochs}]")
        epoch_loss = 0.0
        val_loss = 0.0

        # Iterate through all data partitions
        for df, feature_names, target_name in sequential_data_loader(data_dir):
            if not used_feature_names:
                used_feature_names = feature_names
                # Load exogenous embeddings for used features
                exog_embeddings_tensor = load_exogenous_embeddings(embedding_path, used_feature_names)

            # Create Dataset
            dataset = TimeSeriesDataset(df, feature_names, target_name, exog_embeddings_tensor)
            val_size = int(len(dataset) * validation_split)
            train_size = len(dataset) - val_size

            # Split into training and validation datasets
            train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
            train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

```python
        val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
        train_progress = tqdm(train_loader, desc="Training Progress", unit="batch", leave=True, dynamic_ncols=True, mininterval=0)

        # Training Phase
        model.train()
        for X_batch, exog_batch, y_batch, w_batch in train_progress:
            X_batch = X_batch.to(device)          # (B, input_dim)
            y_batch = y_batch.to(device)          # (B,)
            w_batch = w_batch.to(device)          # (B,)
            exog_batch = exog_batch.to(device)    # (num_features, exog_dim)

            optimizer.zero_grad()
            outputs = model(X_batch, exog_batch)   # (B, output_dim)
            loss = criterion(outputs.view(-1), y_batch, w_batch)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            epoch_loss += loss.item()

            # Update tqdm progress bar
            train_progress.set_postfix({"loss": f"{loss.item():.4f}"})
        train_progress.close()

        # Validation Phase
        model.eval()
        val_progress = tqdm(val_loader, desc="Validation Progress", unit="batch")
        with torch.no_grad():
            for X_batch, exog_batch, y_batch, w_batch in val_progress:
                X_batch = X_batch.to(device)
                y_batch = y_batch.to(device)
                w_batch = w_batch.to(device)
                exog_batch = exog_batch.to(device)

                outputs = model(X_batch, exog_batch)
                loss = criterion(outputs.view(-1), y_batch, w_batch)
                val_loss += loss.item()

        # Print epoch losses
        print(f"Processed partition with training loss: {epoch_loss / len(train_loader):.4f}, validation loss: {val_loss / len(val_loader):.4f}")

    # Step the scheduler
    scheduler.step(val_loss)

    # Print average losses for the epoch
    print(f"Epoch {epoch+1} completed. Average training loss: {epoch_loss / len(train_loader):.4f}, validation loss: {val_loss /
```

```
len(val_loader):.4f}")
        print()

    # Save the trained model
    print("Training completed.")
    torch.save({'model_state_dict': model.state_dict(), 'feature_names'
  : used_feature_names}, "/kaggle/working/timexer_model.pth")
    print("Model saved to timexer_model.pth")
```

Listing 10: train_model Function

Description: The 'train_model' function manages the training process over multiple epochs. It loads data partitions, aligns feature embeddings, splits data into training and validation sets, and iteratively trains the model while monitoring loss metrics. A learning rate scheduler adjusts the learning rate based on validation loss improvements, and the trained model is saved upon completion.

7 Execution and Usage

To execute the complete training script:

1. **Ensure Data Availability**: Verify that the data directory and feature tag embeddings CSV file are correctly placed and accessible.
2. **Custom Layers**: Ensure that the 'SelfAttention_Family.py' and 'Embed.py' modules are correctly implemented and located within the 'layers' directory.
3. **Adjust Paths**: Update the 'data_dir' and 'embedding_path' variables in the 'Main Execution Block' to match your file system.
4. **Run the Script**: Execute the script in an environment with the necessary dependencies installed, such as Kaggle Kernels or a local machine with PyTorch and related libraries.

8 Conclusion

This project successfully integrates unique feature group embeddings as exogenous variables into the TimeXer model, enhancing its capability to perform accurate time series forecasting. By maintaining separate inputs for endogenous and exogenous data, handling missing values appropriately, and incorporating sample weights into the loss function, the model achieves robustness and flexibility essential for real-world applications. The comprehensive implementation ensures that the model remains functional and error-free during both training and inference phases, even when faced with varying test data sizes and missing information.

9 Future Work

Future enhancements to this project could include:

- **Hyperparameter Optimization**: Experimenting with different model hyperparameters to further improve forecasting accuracy.

- **Advanced Embedding Techniques**: Exploring more sophisticated embedding strategies to capture deeper semantic relationships between features.
- **Model Evaluation Metrics**: Implementing additional evaluation metrics such as MAE, RMSE, and R^2 to provide a more comprehensive assessment of model performance.
- **Deployment**: Deploying the trained model in a production environment for real-time forecasting applications.
- **Handling Multivariate Forecasting**: Extending the model to handle multiple target variables simultaneously.