

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

Aryan Dangi

December 28, 2024

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	3
2	Theoretical Background	3
2.1	Time Series Forecasting	3
2.2	TimeXer Model	3
2.3	Feature Group Embeddings	3
2.4	Handling Missing Values	4
2.5	Weighted Loss Functions	4
3	Project Overview	4
4	Implementation Details	4
4.1	Data Loading and Preprocessing	4
4.2	Exogenous Embeddings Integration	5
4.3	Model Architecture	5
4.4	Training Loop	5
4.5	Model Saving	5
5	Code Description and Functionality	5
5.1	Custom Weighted Huber Loss	5
5.2	Exogenous Embeddings Loader	6
5.3	Sequential Data Loader	7

5.4	TimeSeriesDataset Class	8
5.5	TimeXer Model Definition	9
5.6	FlattenHead Class	10
5.7	Encoder and EncoderLayer Classes	11
5.8	Training Loop	13
6	Complete Training Script	16
6.1	Training Loop	24
7	Execution and Usage	26
8	Conclusion	26
9	Future Work	26

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

```
1 class WeightedHuberLoss(nn.Module):
2     def __init__(self, delta=1.0):
3         """
4         Initializes the Weighted Huber Loss.
5
6         Args:
7             delta (float): The point where the loss function changes
8             from quadratic to linear.
9         """
10        super(WeightedHuberLoss, self).__init__()
11        self.delta = delta
12
13    def forward(self, y_pred, y_true, weights):
14        """
15        Computes the weighted Huber loss.
16
17        Args:
18            y_pred (Tensor): Predicted values (B,).
```

```

18         y_true (Tensor): True target values (B,).
19         weights (Tensor): Sample weights (B,).
20
21     Returns:
22         Tensor: The computed weighted Huber loss.
23     """
24     error = y_pred - y_true
25     abs_error = torch.abs(error)
26     quadratic = torch.min(abs_error, torch.tensor(self.delta).to(
y_pred.device))
27     linear = abs_error - quadratic
28     loss = 0.5 * quadratic ** 2 + self.delta * linear
29     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

```

1 def load_exogenous_embeddings(embedding_path, used_feature_names):
2     """
3     Loads and aligns feature tag embeddings with the used feature names
4     .
5     Args:
6         embedding_path (str): Path to the feature tag embeddings CSV
7         file.
8         used_feature_names (list): List of feature names used in the
9         model.
10
11     Returns:
12         Tensor: Exogenous embeddings tensor of shape (num_features,
13         exog_dim).
14     """
15     embeddings_df = pd.read_csv(embedding_path)
16     # Ensure all used features have embeddings
17     missing_embeddings = set(used_feature_names) - set(embeddings_df['
feature']).tolist())
18     if missing_embeddings:
19         print(f"Features missing embeddings: {missing_embeddings}")
20         # Assign zero embeddings to missing features
21         num_embed_dims = embeddings_df.shape[1] - 1 # Excluding '
feature' column
22         for feature in missing_embeddings:
23             embeddings_df = embeddings_df.append({
24                 'feature': feature,
25                 **{f'embed_{i}': 0.0 for i in range(num_embed_dims)}
26             }, ignore_index=True)
27         print("Assigned zero embeddings to missing features.")
28     else:
29         print("All features have corresponding embeddings.")
30
31     # Align embeddings with used features

```

```

29     embeddings_df.set_index('feature', inplace=True)
30     embeddings_aligned = embeddings_df.loc[used_feature_names].
reset_index(drop=True)
31
32     # Convert to tensor
33     exog_embeddings = embeddings_aligned.values.astype(np.float32) # (
num_features, exog_dim)
34     exog_embeddings_tensor = torch.tensor(exog_embeddings, dtype=torch.
float32) # (num_features, exog_dim)
35     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

```

1 def sequential_data_loader(data_dir):
2     """
3     Generator that yields data partitions for training.
4
5     Args:
6         data_dir (str): Directory containing partitioned data.
7
8     Yields:
9         tuple: (DataFrame, used_feature_names, target_name)
10    """
11    for p_id in range(3, 10): # Loop through directories partition_id
=3 to partition_id=9
12        file_path = os.path.join(data_dir, f'partition_id={p_id}', '
part_0.parquet')
13        if not os.path.isfile(file_path):
14            print(f"File not found: {file_path}")
15            continue
16
17        print(f"Loading file: partition_id={p_id}")
18        df = pl.read_parquet(file_path).to_pandas()
19
20        # Define columns to exclude
21        exclude_cols = ['date_id', 'time_id', 'responder_6', 'weight',
22                        'date_id_missing', 'time_id_missing',
23                        'responder_6_missing', 'weight_missing']
24
25        used_feature_names = [col for col in df.columns if col not in
exclude_cols]
26        target_name = 'responder_6'
27
28        # Yield necessary data
29        yield df, used_feature_names, target_name
30
31        # Clean up
32        del df, used_feature_names, target_name
33        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

```

1 class TimeSeriesDataset(Dataset):
2     def __init__(self, data, used_feature_names, target_name,
3         exog_embeddings):
4         """
5             Initializes the TimeSeriesDataset.
6
7         Args:
8             data (DataFrame): The preprocessed data.
9             used_feature_names (list): List of feature names used as
10            inputs.
11             target_name (str): Name of the target variable.
12             exog_embeddings (Tensor): Exogenous embeddings tensor of
13            shape (num_features, exog_dim).
14         """
15         self.features = data[used_feature_names].values # (num_samples
16            , num_features)
17         self.targets = data[target_name].values # (num_samples,)
18         self.weights = data['weight'].values # (num_samples,)
19         self.exog_embeddings = exog_embeddings # (num_features,
20            exog_dim)
21
22     def __len__(self):
23         return len(self.features)
24
25     def __getitem__(self, idx):
26         """
27             Retrieves a single data point.
28
29         Args:
30             idx (int): Index of the data point.
31
32         Returns:
33             tuple: (features, exog_embeddings, target, weight)
34         """
35         x = self.features[idx] # (
36            num_features,)
37         y = self.targets[idx] # scalar
38         w = self.weights[idx] # scalar
39         exog = self.exog_embeddings # (
40            num_features, exog_dim)
41         return torch.tensor(x, dtype=torch.float32), exog, torch.tensor
42            (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

```
1 class TimeXer(nn.Module):
2     def __init__(self, input_dim, exog_dim, projected_dim, hidden_dim,
3         num_layers, output_dim, dropout=0.1):
4         """
5         Initializes the TimeXer model.
6
7         Args:
8             input_dim (int): Number of endogenous features.
9             exog_dim (int): Dimension of each feature's exogenous
10            embedding.
11            projected_dim (int): Dimension after projection.
12            hidden_dim (int): Transformer hidden dimension.
13            num_layers (int): Number of Transformer layers.
14            output_dim (int): Forecast horizon (number of future time
15            steps).
16            dropout (float): Dropout rate.
17        """
18        super(TimeXer, self).__init__()
19        # Projection for endogenous features
20        self.feature_projection = nn.Linear(input_dim, projected_dim)
21        # Projection for exogenous embeddings
22        self.exog_projection = nn.Linear(exog_dim, projected_dim)
23        # Dropout
24        self.dropout = nn.Dropout(p=dropout)
25        # Transformer Encoder
26        self.encoder = Encoder(
27            [
28                EncoderLayer(
29                    AttentionLayer(
30                        FullAttention(False, 4, attention_dropout=
31                        dropout, output_attention=False),
32                        projected_dim, 4
33                    ),
34                    AttentionLayer(
35                        FullAttention(False, 4, attention_dropout=
36                        dropout, output_attention=False),
37                        projected_dim, 4
38                    ),
39                    projected_dim,
40                    hidden_dim,
41                    dropout=dropout,
42                    activation="relu",
43                )
44                for _ in range(num_layers)
45            ],
46            norm_layer=nn.LayerNorm(projected_dim)
47        )
48        # Head to produce forecast
49        self.head_nf = projected_dim * (input_dim + 1) # Adjust based
50        on concatenation
51        self.head = FlattenHead(n_vars=1, nf=self.head_nf,
52            target_window=output_dim, head_dropout=dropout)
53
54        def forward(self, x, exog):
55            """
56            Forward pass of the TimeXer model.
57        """
```

```

50
51     Args:
52         x (Tensor): Endogenous features tensor of shape (B,
input_dim).
53         exog (Tensor): Exogenous embeddings tensor of shape (
num_features, exog_dim).
54
55     Returns:
56         Tensor: Forecasted values tensor of shape (B, output_dim).
57     """
58     # Project endogenous features
59     x_proj = self.feature_projection(x) # (B, projected_dim)
60     x_proj = self.dropout(x_proj)      # (B, projected_dim)
61
62     # Project exogenous embeddings
63     exog_proj = self.exog_projection(exog) # (num_features,
projected_dim)
64     exog_proj = self.dropout(exog_proj)   # (num_features,
projected_dim)
65
66     # Expand exog_proj to match batch size
67     exog_proj = exog_proj.unsqueeze(0).repeat(x.size(0), 1, 1) # (
B, num_features, projected_dim)
68
69     # Concatenate endogenous and exogenous projections as a
sequence
70     # Sequence length = 1 (endogenous) + num_features (exogenous)
71     x_proj_seq = x_proj.unsqueeze(1) # (B, 1, projected_dim)
72     combined = torch.cat([x_proj_seq, exog_proj], dim=1) # (B, 1 +
num_features, projected_dim)
73
74     # Pass through Transformer Encoder
75     enc_out = self.encoder(combined, exog_proj) # (B, 1 +
num_features, projected_dim)
76
77     # Pass through head to get forecast
78     dec_out = self.head(enc_out) # (B, n_vars, output_dim)
79     dec_out = dec_out.permute(0, 2, 1) # (B, output_dim, n_vars)
80     dec_out = dec_out.squeeze(-1) # (B, output_dim)
81
82     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

```

1 class FlattenHead(nn.Module):
2     def __init__(self, n_vars, nf, target_window, head_dropout=0.0):
3         """
4         Initializes the FlattenHead.
5
6         Args:

```

```

7         n_vars (int): Number of variables (features).
8         nf (int): Number of features after projection.
9         target_window (int): Number of future time steps to
forecast.
10        head_dropout (float): Dropout rate.
11        """
12        super(FlattenHead, self).__init__()
13        self.n_vars = n_vars
14        self.flatten = nn.Flatten(start_dim=-2)
15        self.linear = nn.Linear(nf, target_window)
16        self.dropout = nn.Dropout(head_dropout)
17
18    def forward(self, x): # x: [bs x nvars x d_model x patch_num]
19        """
20        Forward pass of FlattenHead.
21
22        Args:
23            x (Tensor): Encoder output tensor.
24
25        Returns:
26            Tensor: Forecasted values tensor.
27        """
28        x = self.flatten(x) # [bs x nvars * d_model * patch_num]
29        x = self.linear(x) # [bs x target_window]
30        x = self.dropout(x)
31        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

```

1 class Encoder(nn.Module):
2     def __init__(self, layers, norm_layer=None, projection=None):
3         """
4         Initializes the Encoder.
5
6         Args:
7             layers (list): List of EncoderLayer instances.
8             norm_layer (nn.Module): Normalization layer.
9             projection (nn.Module): Projection layer.
10        """
11        super(Encoder, self).__init__()
12        self.layers = nn.ModuleList(layers)
13        self.norm = norm_layer
14        self.projection = projection
15
16    def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None,
delta=None):
17        """
18        Forward pass of the Encoder.
19
20        Args:
21            x (Tensor): Input tensor.

```

```

22         cross (Tensor): Cross-attention tensor.
23         x_mask (Tensor, optional): Mask for x.
24         cross_mask (Tensor, optional): Mask for cross.
25         tau (Tensor, optional): Additional parameter.
26         delta (Tensor, optional): Additional parameter.
27
28     Returns:
29         Tensor: Encoded output tensor.
30     """
31     for layer in self.layers:
32         x = layer(x, cross, x_mask=x_mask, cross_mask=cross_mask,
33                 tau=tau, delta=delta)
34
35     if self.norm is not None:
36         x = self.norm(x)
37
38     if self.projection is not None:
39         x = self.projection(x)
40     return x
41
42 class EncoderLayer(nn.Module):
43     def __init__(self, self_attention, cross_attention, d_model, d_ff=
44         None,
45         dropout=0.1, activation="relu"):
46         """
47         Initializes the EncoderLayer.
48
49         Args:
50             self_attention (AttentionLayer): Self-attention layer.
51             cross_attention (AttentionLayer): Cross-attention layer.
52             d_model (int): Model dimension.
53             d_ff (int, optional): Feedforward dimension.
54             dropout (float): Dropout rate.
55             activation (str): Activation function.
56         """
57         super(EncoderLayer, self).__init__()
58         d_ff = d_ff or 4 * d_model
59         self.self_attention = self_attention
60         self.cross_attention = cross_attention
61         self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff,
62                                kernel_size=1)
63         self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model,
64                                kernel_size=1)
65         self.norm1 = nn.LayerNorm(d_model)
66         self.norm2 = nn.LayerNorm(d_model)
67         self.norm3 = nn.LayerNorm(d_model)
68         self.dropout = nn.Dropout(dropout)
69         self.activation = F.relu if activation == "relu" else F.gelu
70
71     def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None,
72                 delta=None):
73         """
74         Forward pass of the EncoderLayer.
75
76         Args:
77             x (Tensor): Input tensor.
78             cross (Tensor): Cross-attention tensor.
79             x_mask (Tensor, optional): Mask for x.

```

```

75         cross_mask (Tensor, optional): Mask for cross.
76         tau (Tensor, optional): Additional parameter.
77         delta (Tensor, optional): Additional parameter.
78
79     Returns:
80         Tensor: Output tensor after encoder layer.
81     """
82     B, L, D = cross.shape
83     # Self-attention
84     x = x + self.dropout(self.self_attention(
85         x, x, x,
86         attn_mask=x_mask,
87         tau=tau, delta=None
88     )[0])
89     x = self.norm1(x)
90
91     # Cross-attention
92     x_glb_ori = x[:, -1, :].unsqueeze(1) # (B, 1, D)
93     x_glb = torch.reshape(x_glb_ori, (B, -1, D)) # (B, 1, D)
94     x_glb_attn = self.dropout(self.cross_attention(
95         x_glb, cross, cross,
96         attn_mask=cross_mask,
97         tau=tau, delta=delta
98     )[0])
99     x_glb_attn = torch.reshape(x_glb_attn,
100                               (x_glb_attn.shape[0] * x_glb_attn.
shape[1], x_glb_attn.shape[2])).unsqueeze(1)
101     x_glb = x_glb_ori + x_glb_attn
102     x_glb = self.norm2(x_glb)
103
104     # Combine
105     y = x = torch.cat([x[:, :-1, :], x_glb], dim=1) # (B, L, D)
106
107     # Feedforward
108     y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))
109 )) # (B, D, L)
110     y = self.dropout(self.conv2(y).transpose(-1, 1))
111     # (B, L, D)
112
113     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

```

1 def train_model(data_dir, embedding_path, model, optimizer, criterion,
2 epochs=10, batch_size=64, validation_split=0.2):
3     """
4     Trains the TimeXer model.
5
6     Args:
7         data_dir (str): Directory containing partitioned training data.

```

```

7         embedding_path (str): Path to the feature tag embeddings CSV
file.
8         model (nn.Module): The TimeXer model to train.
9         optimizer (torch.optim.Optimizer): Optimizer for training.
10        criterion (nn.Module): Loss function.
11        epochs (int): Number of training epochs.
12        batch_size (int): Training batch size.
13        validation_split (float): Fraction of data to use for
validation.
14        """
15        used_feature_names = []
16        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
')
17        model.to(device)
18
19        # Learning rate scheduler
20        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.5, patience=2)
21
22        for epoch in range(epochs):
23            print(f"Epoch [{epoch+1}/{epochs}]")
24            epoch_loss = 0.0
25            val_loss = 0.0
26
27            # Iterate through all data partitions
28            for df, feature_names, target_name in sequential_data_loader(
data_dir):
29                if not used_feature_names:
30                    used_feature_names = feature_names
31                    # Load exogenous embeddings for used features
32                    exog_embeddings_tensor = load_exogenous_embeddings(
embedding_path, used_feature_names)
33
34                # Create Dataset
35                dataset = TimeSeriesDataset(df, feature_names, target_name,
exog_embeddings_tensor)
36                val_size = int(len(dataset) * validation_split)
37                train_size = len(dataset) - val_size
38
39                # Split into training and validation datasets
40                train_dataset, val_dataset = random_split(dataset, [
train_size, val_size])
41                train_loader = DataLoader(train_dataset, batch_size=
batch_size, shuffle=True)
42                val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
43                train_progress = tqdm(train_loader, desc="Training Progress
", unit="batch", leave=True, dynamic_ncols=True, mininterval=0)
44
45                # Training Phase
46                model.train()
47                for X_batch, exog_batch, y_batch, w_batch in train_progress
:
48                    X_batch = X_batch.to(device)           # (B, input_dim)
49                    y_batch = y_batch.to(device)           # (B,)
50                    w_batch = w_batch.to(device)           # (B,)
51                    exog_batch = exog_batch.to(device)     # (num_features,
exog_dim)

```

```

52         optimizer.zero_grad()
53         outputs = model(X_batch, exog_batch)    # (B, output_dim
54     )
55         loss = criterion(outputs.view(-1), y_batch, w_batch)
56         loss.backward()
57         torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)
58         optimizer.step()
59         epoch_loss += loss.item()
60
61         # Update tqdm progress bar
62         train_progress.set_postfix({"loss": f"{loss.item():.4f}"
})
63     train_progress.close()
64
65     # Validation Phase
66     model.eval()
67     val_progress = tqdm(val_loader, desc="Validation Progress",
unit="batch")
68     with torch.no_grad():
69         for X_batch, exog_batch, y_batch, w_batch in
val_progress:
70             X_batch = X_batch.to(device)
71             y_batch = y_batch.to(device)
72             w_batch = w_batch.to(device)
73             exog_batch = exog_batch.to(device)
74
75             outputs = model(X_batch, exog_batch)
76             loss = criterion(outputs.view(-1), y_batch, w_batch
)
77             val_loss += loss.item()
78
79     # Print epoch losses
80     print(f"Processed partition with training loss: {epoch_loss
/ len(train_loader):.4f}, validation loss: {val_loss / len(
val_loader):.4f}")
81
82     # Step the scheduler
83     scheduler.step(val_loss)
84
85     # Print average losses for the epoch
86     print(f"Epoch {epoch+1} completed. Average training loss: {
epoch_loss / len(train_loader):.4f}, validation loss: {val_loss /
len(val_loader):.4f}")
87     print()
88
89     # Save the trained model
90     print("Training completed.")
91     torch.save({'model_state_dict': model.state_dict(), 'feature_names'
: used_feature_names}, "/kaggle/working/timexer_model.pth")
92     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.

```
1 # Import necessary libraries
2 import os
3 import gc
4 import torch
5 import torch.nn as nn
6 import torch.optim as optim
7 from torch.utils.data import DataLoader, Dataset, random_split
8 import polars as pl
9 import pandas as pd
10 import numpy as np
11 from tqdm.notebook import tqdm
12
13 # Import custom layers - ensure these modules are present in the '
14   layers' directory
15 from layers.SelfAttention_Family import FullAttention, AttentionLayer
16 from layers.Embed import DataEmbedding_inverted, PositionalEmbedding
17
18 # =====
19 # Custom Weighted Huber Loss
20 # =====
21 class WeightedHuberLoss(nn.Module):
22     def __init__(self, delta=1.0):
23         """
24         Initializes the Weighted Huber Loss.
25
26         Args:
27             delta (float): The point where the loss function changes
28             from quadratic to linear.
29         """
30         super(WeightedHuberLoss, self).__init__()
31         self.delta = delta
32
33     def forward(self, y_pred, y_true, weights):
34         """
35         Computes the weighted Huber loss.
36
37         Args:
38             y_pred (Tensor): Predicted values (B,).
39             y_true (Tensor): True target values (B,).
40             weights (Tensor): Sample weights (B,).
41
42         Returns:
43             Tensor: The computed weighted Huber loss.
44         """
45         error = y_pred - y_true
46         abs_error = torch.abs(error)
```



```

45         quadratic = torch.min(abs_error, torch.tensor(self.delta).to(
y_pred.device))
46         linear = abs_error - quadratic
47         loss = 0.5 * quadratic ** 2 + self.delta * linear
48         return (loss * weights).mean()
49
50 # =====
51 # Exogenous Embeddings Loader
52 # =====
53 def load_exogenous_embeddings(embedding_path, used_feature_names):
54     """
55     Loads and aligns feature tag embeddings with the used feature names
56     .
57
58     Args:
59         embedding_path (str): Path to the feature tag embeddings CSV
60         file.
61         used_feature_names (list): List of feature names used in the
62         model.
63
64     Returns:
65         Tensor: Exogenous embeddings tensor of shape (num_features,
66         exog_dim).
67     """
68     embeddings_df = pd.read_csv(embedding_path)
69     # Ensure all used features have embeddings
70     missing_embeddings = set(used_feature_names) - set(embeddings_df['
feature']).tolist()
71     if missing_embeddings:
72         print(f"Features missing embeddings: {missing_embeddings}")
73         # Assign zero embeddings to missing features
74         num_embed_dims = embeddings_df.shape[1] - 1 # Excluding '
feature' column
75         for feature in missing_embeddings:
76             embeddings_df = embeddings_df.append({
77                 'feature': feature,
78                 **{f'embed_{i}': 0.0 for i in range(num_embed_dims)}
79             }, ignore_index=True)
80         print("Assigned zero embeddings to missing features.")
81     else:
82         print("All features have corresponding embeddings.")
83
84     # Align embeddings with used features
85     embeddings_df.set_index('feature', inplace=True)
86     embeddings_aligned = embeddings_df.loc[used_feature_names].
reset_index(drop=True)
87
88     # Convert to tensor
89     exog_embeddings = embeddings_aligned.values.astype(np.float32) # (
num_features, exog_dim)
90     exog_embeddings_tensor = torch.tensor(exog_embeddings, dtype=torch.
float32) # (num_features, exog_dim)
91     return exog_embeddings_tensor # Shape: (num_features, exog_dim)
92
93 # =====
94 # Sequential Data Loader
95 # =====
96 def sequential_data_loader(data_dir):

```

```

93     """
94     Generator that yields data partitions for training.
95
96     Args:
97         data_dir (str): Directory containing partitioned data.
98
99     Yields:
100         tuple: (DataFrame, used_feature_names, target_name)
101     """
102     for p_id in range(3, 10): # Loop through directories partition_id
103         #3 to partition_id=9
104         file_path = os.path.join(data_dir, f'partition_id={p_id}', '
105         part_0.parquet')
106         if not os.path.isfile(file_path):
107             print(f"File not found: {file_path}")
108             continue
109
110         print(f"Loading file: partition_id={p_id}")
111         df = pl.read_parquet(file_path).to_pandas()
112
113         # Define columns to exclude
114         exclude_cols = ['date_id', 'time_id', 'responder_6', 'weight',
115                         'date_id_missing', 'time_id_missing',
116                         'responder_6_missing', 'weight_missing']
117
118         used_feature_names = [col for col in df.columns if col not in
119         exclude_cols]
120         target_name = 'responder_6'
121
122         # Yield necessary data
123         yield df, used_feature_names, target_name
124
125         # Clean up
126         del df, used_feature_names, target_name
127         gc.collect()
128
129 # =====
130 # TimeSeries Dataset Class
131 # =====
132 class TimeSeriesDataset(Dataset):
133     def __init__(self, data, used_feature_names, target_name,
134     exog_embeddings):
135         """
136         Initializes the TimeSeriesDataset.
137
138         Args:
139             data (DataFrame): The preprocessed data.
140             used_feature_names (list): List of feature names used as
141             inputs.
142             target_name (str): Name of the target variable.
143             exog_embeddings (Tensor): Exogenous embeddings tensor of
144             shape (num_features, exog_dim).
145         """
146         self.features = data[used_feature_names].values # (num_samples
147         , num_features)
148         self.targets = data[target_name].values # (num_samples,)
149         self.weights = data['weight'].values # (num_samples,)

```

```

143         self.exog_embeddings = exog_embeddings          # (num_features,
exog_dim)
144
145     def __len__(self):
146         return len(self.features)
147
148     def __getitem__(self, idx):
149         """
150         Retrieves a single data point.
151
152         Args:
153             idx (int): Index of the data point.
154
155         Returns:
156             tuple: (features, exog_embeddings, target, weight)
157         """
158         x = self.features[idx]                          # (
num_features,)
159         y = self.targets[idx]                            # scalar
160         w = self.weights[idx]                            # scalar
161         exog = self.exog_embeddings                      # (
num_features, exog_dim)
162         return torch.tensor(x, dtype=torch.float32), exog, torch.tensor
(y, dtype=torch.float32), torch.tensor(w, dtype=torch.float32)
163
164 # =====
165 # TimeXer Model Definition
166 # =====
167 class TimeXer(nn.Module):
168     def __init__(self, input_dim, exog_dim, projected_dim, hidden_dim,
num_layers, output_dim, dropout=0.1):
169         """
170         Initializes the TimeXer model.
171
172         Args:
173             input_dim (int): Number of endogenous features.
174             exog_dim (int): Dimension of each feature's exogenous
embedding.
175             projected_dim (int): Dimension after projection.
176             hidden_dim (int): Transformer hidden dimension.
177             num_layers (int): Number of Transformer layers.
178             output_dim (int): Forecast horizon (number of future time
steps).
179             dropout (float): Dropout rate.
180         """
181         super(TimeXer, self).__init__()
182         # Projection for endogenous features
183         self.feature_projection = nn.Linear(input_dim, projected_dim)
184         # Projection for exogenous embeddings
185         self.exog_projection = nn.Linear(exog_dim, projected_dim)
186         # Dropout
187         self.dropout = nn.Dropout(p=dropout)
188         # Transformer Encoder
189         self.encoder = Encoder(
190             [
191                 EncoderLayer(
192                     AttentionLayer(

```

```

193         FullAttention(False, 4, attention_dropout=
dropout, output_attention=False),
194         projected_dim, 4
195     ),
196     AttentionLayer(
197         FullAttention(False, 4, attention_dropout=
dropout, output_attention=False),
198         projected_dim, 4
199     ),
200     projected_dim,
201     hidden_dim,
202     dropout=dropout,
203     activation="relu",
204 )
205     for _ in range(num_layers)
206 ],
207     norm_layer=nn.LayerNorm(projected_dim)
208 )
209     # Head to produce forecast
210     self.head_nf = projected_dim * (input_dim + 1) # Adjust based
on concatenation
211     self.head = FlattenHead(n_vars=1, nf=self.head_nf,
target_window=output_dim, head_dropout=dropout)
212
213     def forward(self, x, exog):
214         """
215         Forward pass of the TimeXer model.
216
217         Args:
218             x (Tensor): Endogenous features tensor of shape (B,
input_dim).
219             exog (Tensor): Exogenous embeddings tensor of shape (
num_features, exog_dim).
220
221         Returns:
222             Tensor: Forecasted values tensor of shape (B, output_dim).
223         """
224         # Project endogenous features
225         x_proj = self.feature_projection(x) # (B, projected_dim)
226         x_proj = self.dropout(x_proj) # (B, projected_dim)
227
228         # Project exogenous embeddings
229         exog_proj = self.exog_projection(exog) # (num_features,
projected_dim)
230         exog_proj = self.dropout(exog_proj) # (num_features,
projected_dim)
231
232         # Expand exog_proj to match batch size
233         exog_proj = exog_proj.unsqueeze(0).repeat(x.size(0), 1, 1) # (
B, num_features, projected_dim)
234
235         # Concatenate endogenous and exogenous projections as a
sequence
236         # Sequence length = 1 (endogenous) + num_features (exogenous)
237         x_proj_seq = x_proj.unsqueeze(1) # (B, 1, projected_dim)
238         combined = torch.cat([x_proj_seq, exog_proj], dim=1) # (B, 1 +
num_features, projected_dim)
239

```

```

240     # Pass through Transformer Encoder
241     enc_out = self.encoder(combined, exog_proj)  # (B, 1 +
num_features, projected_dim)
242
243     # Pass through head to get forecast
244     dec_out = self.head(enc_out)  # (B, n_vars, output_dim)
245     dec_out = dec_out.permute(0, 2, 1)  # (B, output_dim, n_vars)
246     dec_out = dec_out.squeeze(-1)  # (B, output_dim)
247
248     return dec_out
249
250 # =====
251 # FlattenHead Class
252 # =====
253 class FlattenHead(nn.Module):
254     def __init__(self, n_vars, nf, target_window, head_dropout=0.0):
255         """
256         Initializes the FlattenHead.
257
258         Args:
259             n_vars (int): Number of variables (features).
260             nf (int): Number of features after projection.
261             target_window (int): Number of future time steps to
forecast.
262             head_dropout (float): Dropout rate.
263         """
264         super(FlattenHead, self).__init__()
265         self.n_vars = n_vars
266         self.flatten = nn.Flatten(start_dim=-2)
267         self.linear = nn.Linear(nf, target_window)
268         self.dropout = nn.Dropout(head_dropout)
269
270     def forward(self, x):  # x: [bs x nvars x d_model x patch_num]
271         """
272         Forward pass of FlattenHead.
273
274         Args:
275             x (Tensor): Encoder output tensor.
276
277         Returns:
278             Tensor: Forecasted values tensor.
279         """
280         x = self.flatten(x)  # [bs x nvars * d_model * patch_num]
281         x = self.linear(x)  # [bs x target_window]
282         x = self.dropout(x)
283         return x
284
285 # =====
286 # Encoder and EncoderLayer Definitions
287 # =====
288 class Encoder(nn.Module):
289     def __init__(self, layers, norm_layer=None, projection=None):
290         """
291         Initializes the Encoder.
292
293         Args:
294             layers (list): List of EncoderLayer instances.
295             norm_layer (nn.Module): Normalization layer.

```

```

296         projection (nn.Module): Projection layer.
297     """
298     super(Encoder, self).__init__()
299     self.layers = nn.ModuleList(layers)
300     self.norm = norm_layer
301     self.projection = projection
302
303     def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None,
304                 delta=None):
305         """
306         Forward pass of the Encoder.
307
308         Args:
309             x (Tensor): Input tensor.
310             cross (Tensor): Cross-attention tensor.
311             x_mask (Tensor, optional): Mask for x.
312             cross_mask (Tensor, optional): Mask for cross.
313             tau (Tensor, optional): Additional parameter.
314             delta (Tensor, optional): Additional parameter.
315
316         Returns:
317             Tensor: Encoded output tensor.
318         """
319         for layer in self.layers:
320             x = layer(x, cross, x_mask=x_mask, cross_mask=cross_mask,
321                     tau=tau, delta=delta)
322
323         if self.norm is not None:
324             x = self.norm(x)
325
326         if self.projection is not None:
327             x = self.projection(x)
328         return x
329
330 class EncoderLayer(nn.Module):
331     def __init__(self, self_attention, cross_attention, d_model, d_ff=
332         None,
333         dropout=0.1, activation="relu"):
334         """
335         Initializes the EncoderLayer.
336
337         Args:
338             self_attention (AttentionLayer): Self-attention layer.
339             cross_attention (AttentionLayer): Cross-attention layer.
340             d_model (int): Model dimension.
341             d_ff (int, optional): Feedforward dimension.
342             dropout (float): Dropout rate.
343             activation (str): Activation function.
344         """
345         super(EncoderLayer, self).__init__()
346         d_ff = d_ff or 4 * d_model
347         self.self_attention = self_attention
348         self.cross_attention = cross_attention
349         self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff,
350                                kernel_size=1)
351         self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model,
352                                kernel_size=1)
353         self.norm1 = nn.LayerNorm(d_model)

```

```

349     self.norm2 = nn.LayerNorm(d_model)
350     self.norm3 = nn.LayerNorm(d_model)
351     self.dropout = nn.Dropout(dropout)
352     self.activation = F.relu if activation == "relu" else F.gelu
353
354 def forward(self, x, cross, x_mask=None, cross_mask=None, tau=None,
355             delta=None):
356     """
357     Forward pass of the EncoderLayer.
358
359     Args:
360         x (Tensor): Input tensor.
361         cross (Tensor): Cross-attention tensor.
362         x_mask (Tensor, optional): Mask for x.
363         cross_mask (Tensor, optional): Mask for cross.
364         tau (Tensor, optional): Additional parameter.
365         delta (Tensor, optional): Additional parameter.
366
367     Returns:
368         Tensor: Output tensor after encoder layer.
369     """
370     B, L, D = cross.shape
371     # Self-attention
372     x = x + self.dropout(self.self_attention(
373         x, x, x,
374         attn_mask=x_mask,
375         tau=tau, delta=None
376     )[0])
377     x = self.norm1(x)
378
379     # Cross-attention
380     x_glb_ori = x[:, -1, :].unsqueeze(1) # (B, 1, D)
381     x_glb = torch.reshape(x_glb_ori, (B, -1, D)) # (B, 1, D)
382     x_glb_attn = self.dropout(self.cross_attention(
383         x_glb, cross, cross,
384         attn_mask=cross_mask,
385         tau=tau, delta=delta
386     )[0])
387     x_glb_attn = torch.reshape(x_glb_attn,
388                               (x_glb_attn.shape[0] * x_glb_attn.
389                                shape[1], x_glb_attn.shape[2])).unsqueeze(1)
390     x_glb = x_glb_ori + x_glb_attn
391     x_glb = self.norm2(x_glb)
392
393     # Combine
394     y = x = torch.cat([x[:, :-1, :], x_glb], dim=1) # (B, L, D)
395
396     # Feedforward
397     y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))
398     )) # (B, D, L)
399     y = self.dropout(self.conv2(y).transpose(-1, 1))
400     # (B, L, D)
401
402     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 con-

volutional layers. These layers facilitate the modeling of complex dependencies within the data and the integration of exogenous embeddings.

6.1 Training Loop

```

1 def train_model(data_dir, embedding_path, model, optimizer, criterion,
2 epochs=10, batch_size=64, validation_split=0.2):
3     """
4     Trains the TimeXer model.
5
6     Args:
7         data_dir (str): Directory containing partitioned training data.
8         embedding_path (str): Path to the feature tag embeddings CSV
9         file.
10        model (nn.Module): The TimeXer model to train.
11        optimizer (torch.optim.Optimizer): Optimizer for training.
12        criterion (nn.Module): Loss function.
13        epochs (int): Number of training epochs.
14        batch_size (int): Training batch size.
15        validation_split (float): Fraction of data to use for
16        validation.
17    """
18    used_feature_names = []
19    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
20    model.to(device)
21
22    # Learning rate scheduler
23    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
24 mode='min', factor=0.5, patience=2)
25
26    for epoch in range(epochs):
27        print(f"Epoch [{epoch+1}/{epochs}]")
28        epoch_loss = 0.0
29        val_loss = 0.0
30
31        # Iterate through all data partitions
32        for df, feature_names, target_name in sequential_data_loader(
33 data_dir):
34            if not used_feature_names:
35                used_feature_names = feature_names
36                # Load exogenous embeddings for used features
37                exog_embeddings_tensor = load_exogenous_embeddings(
38 embedding_path, used_feature_names)
39
40            # Create Dataset
41            dataset = TimeSeriesDataset(df, feature_names, target_name,
42 exog_embeddings_tensor)
43            val_size = int(len(dataset) * validation_split)
44            train_size = len(dataset) - val_size
45
46            # Split into training and validation datasets
47            train_dataset, val_dataset = random_split(dataset, [
48 train_size, val_size])
49            train_loader = DataLoader(train_dataset, batch_size=
50 batch_size, shuffle=True)

```



```

42         val_loader = DataLoader(val_dataset, batch_size=batch_size,
    shuffle=False)
43         train_progress = tqdm(train_loader, desc="Training Progress
    ", unit="batch", leave=True, dynamic_ncols=True, mininterval=0)
44
45         # Training Phase
46         model.train()
47         for X_batch, exog_batch, y_batch, w_batch in train_progress
    :
48             X_batch = X_batch.to(device)           # (B, input_dim)
49             y_batch = y_batch.to(device)           # (B,)
50             w_batch = w_batch.to(device)           # (B,)
51             exog_batch = exog_batch.to(device)      # (num_features,
    exog_dim)
52
53             optimizer.zero_grad()
54             outputs = model(X_batch, exog_batch)    # (B, output_dim
    )
55             loss = criterion(outputs.view(-1), y_batch, w_batch)
56             loss.backward()
57             torch.nn.utils.clip_grad_norm_(model.parameters(),
    max_norm=1.0)
58             optimizer.step()
59             epoch_loss += loss.item()
60
61             # Update tqdm progress bar
62             train_progress.set_postfix({"loss": f"{loss.item():.4f}
    "})
63         train_progress.close()
64
65         # Validation Phase
66         model.eval()
67         val_progress = tqdm(val_loader, desc="Validation Progress",
    unit="batch")
68         with torch.no_grad():
69             for X_batch, exog_batch, y_batch, w_batch in
    val_progress:
70                 X_batch = X_batch.to(device)
71                 y_batch = y_batch.to(device)
72                 w_batch = w_batch.to(device)
73                 exog_batch = exog_batch.to(device)
74
75                 outputs = model(X_batch, exog_batch)
76                 loss = criterion(outputs.view(-1), y_batch, w_batch
    )
77                 val_loss += loss.item()
78
79             # Print epoch losses
80             print(f"Processed partition with training loss: {epoch_loss
    / len(train_loader):.4f}, validation loss: {val_loss / len(
    val_loader):.4f}")
81
82             # Step the scheduler
83             scheduler.step(val_loss)
84
85             # Print average losses for the epoch
86             print(f"Epoch {epoch+1} completed. Average training loss: {
    epoch_loss / len(train_loader):.4f}, validation loss: {val_loss /

```

```

len(val_loader):.4f}")
87     print()
88
89     # Save the trained model
90     print("Training completed.")
91     torch.save({'model_state_dict': model.state_dict(), 'feature_names'
: used_feature_names}, "/kaggle/working/timexer_model.pth")
92     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.