Child Mind Institute - Detect Sleep States

Our Journey Through Kaggle Competition

Yarden Rotem and Rotem Abend
Workshop in Machine Learning and Data Analysis

April 27, 2025

Abstract

This file documents our journey through the Child Mind Institute - Detect Sleep States Kaggle competition. We developed machine learning solutions to detect sleep onset and wakeup events from wrist-worn accelerometer data. Starting with simple models achieving a score of 0.162, we iteratively improved our approach through sophisticated preprocessing, deep learning architectures, and post-processing techniques to achieve a final score of 0.756, earning us a medal. This paper outlines our methodology, challenges, and solutions, offering insights into time-series analysis for sleep event detection.

1 Introduction to the Problem

The Child Mind Institute - Detect Sleep States competition required participants to develop algorithms capable of identifying two critical sleep-related events in accelerometer data: `onset` (falling asleep) and `wakeup` (awakening). The challenge involved analyzing time-series data from wrist-worn accelerometers to accurately detect these relatively rare events.

1.1 Dataset and Telemetry

The dataset consisted of approximately 500 multi-day recordings from wrist-worn accelerometers, with each recording representing data from a unique participant. The data included two key measurements:

- `anglez`: The angle of the arm relative to the body's vertical axis
- `enmo`: Euclidean Norm Minus One, a common acceleration measurement

The data was provided in multiple files:

- `train_series.parquet` and `test_series.parquet`: Continuous multi-day accelerometer recordings
- `train_events.csv`: Sleep event annotations (`onset`, `wakeup`) with timestamps and timesteps

1.2 Evaluation Metric

The competition used a specialized evaluation metric based on Mean Average Precision (MAP) with a unique twist. Rather than using traditional thresholds, the evaluation calculated precision at different step thresholds for each event type (onset and wakeup). A prediction was considered correct if it fell within 30 minutes of the ground truth event. The final score represented the area under the precision curve, with a maximum possible value of 1.0.

Additionally, sleep periods had specific constraints:

- Sleep periods must last at least 30 minutes
- Sleep is considered interrupted if activity lasts more than 30 minutes
- Only the longest sleep window of the night is recorded
- Predictions during periods when the device is not worn are counted as false positives

2 Preprocessing

Our preprocessing pipeline was critical to the success of our models. We developed several key techniques to prepare the raw accelerometer data for modeling.

2.1 Bucketing Strategy

We implemented a custom bucketing approach to organize the continuous time-series data into more manageable chunks. Rather than using calendar days, we defined each bucket as a period from 18:00 to 18:00 the following day, which better aligned with typical sleep patterns (users typically sleep during night hours, which would otherwise split a sleep event across two calendar days).

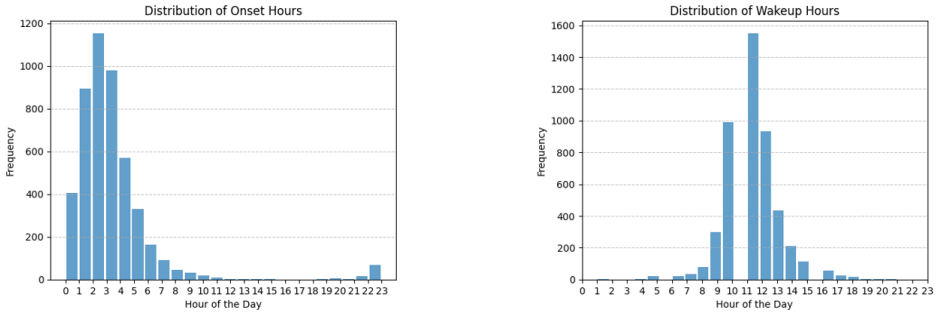


Figure 1: Distribution of sleep onset and wakeup hours, illustrating the rationale for the 18:00-18:00 bucketing strategy.

```python
def assign_bucket(timestamp):
    if timestamp.hour >= 18:
        return timestamp.date()
    else:
        return (timestamp - pd.Timedelta(days=1)).date()
```

Listing 1: Custom Bucket Definition

2.2 Minute-Level Aggregation and Feature Engineering

After defining the daily buckets, we further processed the raw 5-second interval data within each bucket:

- **Minute Aggregation**: The time-series data was grouped into one-minute intervals. Within each minute, we calculated statistical features such as the mean, standard deviation, and median for both `anglez` and `enmo`. We also retained the minimum timestep for each minute and the hour of the day.
- **Target Assignment**: For each minute, the target score was assigned based on the maximum absolute score value present within that minute's 5-second intervals, using the exponentially decayed target signal described later (Section 4.1).
- **Clipping and Log Transformation**: The mean `anglez` values were first clipped to the range [-90, 90] and then log-transformed after shifting to ensure positive values (`log1p(anglez_mean + 91)`). This helped normalize the distribution of angle values.

- **Exponentially Weighted Moving Average (EWMA)**: Applied EWMA with a span of 30 minutes to the minute-aggregated mean `anglez_log` and mean `enmo` values. This smoothed the time series and captured longer-term trends while reducing noise.
- **Value Counts Feature**: We calculated the frequency of the most common `anglez` value within each minute as an additional feature.
- **Per-Bucket Standardization**: Key numerical features (`enmo_mean`, `enmo_std`, `anglez_log`, `anglez_std`, and their EWMA counterparts) were standardized within each daily bucket using Z-score normalization. This ensured features had a mean of 0 and a standard deviation of 1 within each day's context, making them comparable across different series and days.
- **Padding**: Since the duration of data within each daily bucket could vary, all feature sequences (like `enmo_mean`, `anglez_log`, target score, etc.) were padded to a fixed length of 1440 (representing the number of minutes in a 24-hour period) using edge padding ('mode='edge''). This created uniform input sequences for the deep learning models.
- **Sinusoidal Hour Encoding**: To capture the cyclical nature of time without introducing artificial breaks (like the jump between hour 23 and 0), the hour of the day was transformed into two sinusoidal features: `hour_sin` and `hour_cos`.

These steps transformed the high-frequency raw data into fixed-length sequences of rich features for each day, suitable for input into sequence models. The sinusoidal hour encoding proved particularly valuable:

```
buckets['hour_sin'] = buckets['hour'].apply(lambda x: np.sin(2 * np.pi
    * x / 24))
buckets['hour_cos'] = buckets['hour'].apply(lambda x: np.cos(2 * np.pi
    * x / 24))
```

Listing 2: Sinusoidal Hour Encoding

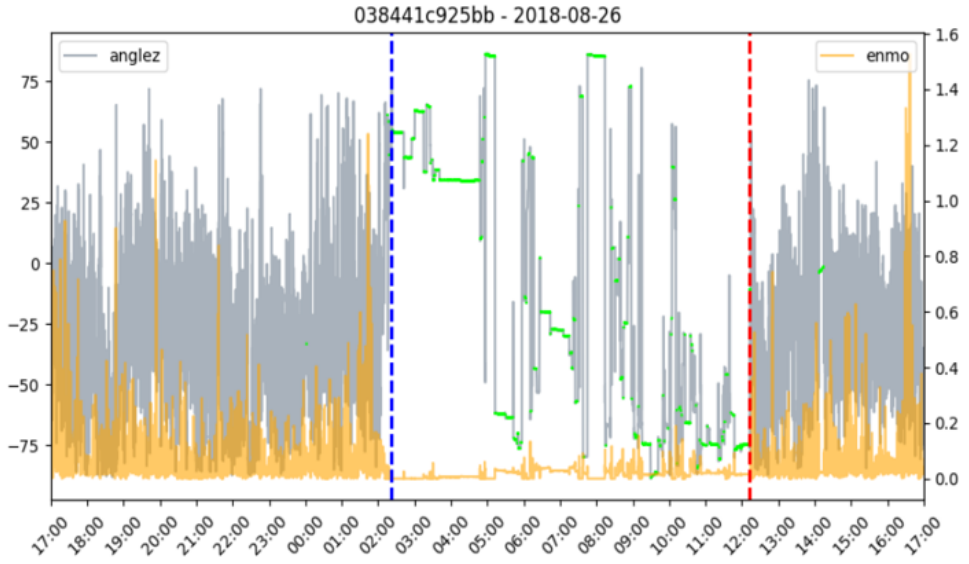


Figure 2: Value counts for the `anglez` feature during sleep periods, highlighting static arm positions during between onset (blue) and wakeup (red).

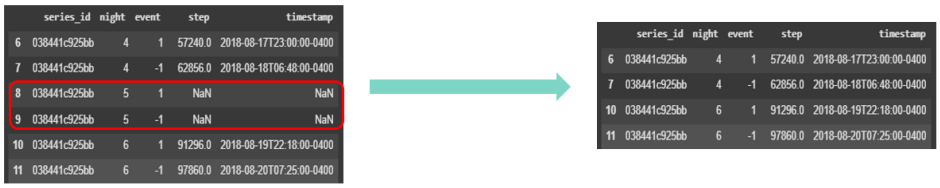
3 Worn/Unworn Classification

A critical aspect of the competition was handling periods when the device was not worn. Predictions during unworn periods were scored as false positives, so accurately identifying these periods was essential.

3.1 Model Development

We experimented with several approaches for worn/unworn classification:

- **XGBoost**: A gradient boosting model that achieved high accuracy (>95%)
- **Random Forest**: Provided good performance but slightly lower than XGBoost
- **CNN**: A 1D convolutional neural network for sequence classification
- **Ensemble**: Majority voting between models for the final prediction



	series_id	night	event	step	timestamp
6	038441c925bb	4	1	57240.0	2018-08-17T23:00:00-0400
7	038441c925bb	4	-1	62856.0	2018-08-18T06:48:00-0400
8	038441c925bb	5	1	NaN	NaN
9	038441c925bb	5	-1	NaN	NaN
10	038441c925bb	6	1	91296.0	2018-08-19T22:18:00-0400
11	038441c925bb	6	-1	97860.0	2018-08-20T07:25:00-0400

	series_id	night	event	step	timestamp
6	038441c925bb	4	1	57240.0	2018-08-17T23:00:00-0400
7	038441c925bb	4	-1	62856.0	2018-08-18T06:48:00-0400
10	038441c925bb	6	1	91296.0	2018-08-19T22:18:00-0400
11	038441c925bb	6	-1	97860.0	2018-08-20T07:25:00-0400

Figure 3: Illustration of the process for filtering out periods when the device was likely not worn.

After extensive testing, we found that XGBoost provided the best individual performance for worn/unworn classification, with accuracy consistently above 95%. It was surprising, because we used different methods such as a majority voting approach that combined all three models:

```
# Predict using each model
worn_predictions_1 = xgboost_worn_unworn_model.predict(X_test_flattened
    )
worn_predictions_2 = rf_worn_unworn_model.predict(X_test_flattened)
cnn_probs = cnn_worn_unworn_model.predict(X_test).ravel()
worn_predictions_3 = (cnn_probs > 0.4).astype(int)

# Majority vote (element-wise)
final_preds = (worn_predictions_1 + worn_predictions_2 +
    worn_predictions_3) >= 2
worn_predictions = final_preds.astype(int)
```

Listing 3: Majority Voting Ensemble

3.2 Feature Importance

Analysis revealed that the most important feature for worn/unworn was the `value_counts` of the anglez, we assume that the device was put on the table or in a drawer, and the anglez value was static.

4 Modeling for Sleep Event Detection

Our approach to sleep event detection evolved significantly throughout the competition, starting with simple regression models and culminating in sophisticated deep learning architectures with custom loss functions.

4.1 Initial Approach: Logistic Regression for Sleep/Awake Detection

Our first approach involved using a simple logistic regression model to classify sleep and awake states. This method worked well for detecting general sleep/awake patterns, achieving reasonable accuracy in distinguishing between the two states. However, it fell short for the competition's requirements due to the following reasons:

- **Lack of Event-Specific Predictions**: The model did not directly predict the critical sleep events (`onset` and `wakeup`), which were essential for the competition.
- **Complex Post-Processing**: Translating the continuous sleep/awake predictions into discrete event detections required extensive post-processing, which introduced additional complexity and reduced overall performance.

Recognizing these limitations, we shifted our focus to directly detecting sleep events (`onset` and `wakeup`) using deeper models tailored for sequence-to-sequence regression tasks.

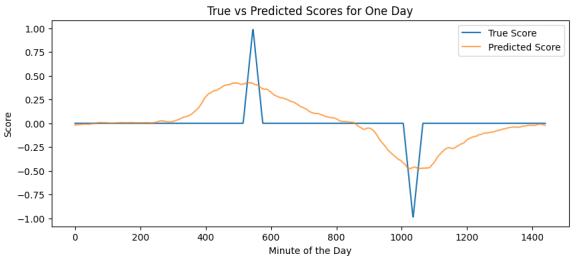
4.2 Current Approach: Target Engineering

We experimented with several approaches to transform the discrete onset/wakeup events into continuous target signals for regression-based learning:

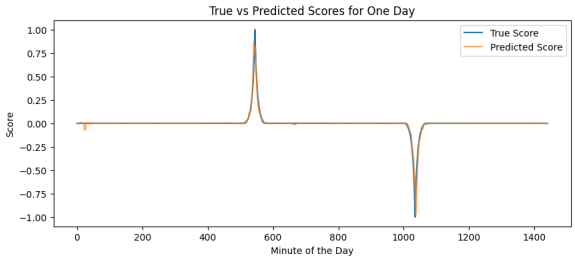
1. **Linear Target**: Simple linear ramps up/down around events
2. **Composite Target**: Combined multiple signals with different decay rates
3. **Exponential Target**: Exponential decay from event points, inspired by a previous Kaggle solution

The exponential target proved most effective, as it created a smoother signal that preserved the prominence of event points while providing a continuous gradient for the model to learn from. This approach was particularly valuable for optimizing the MAP score because:

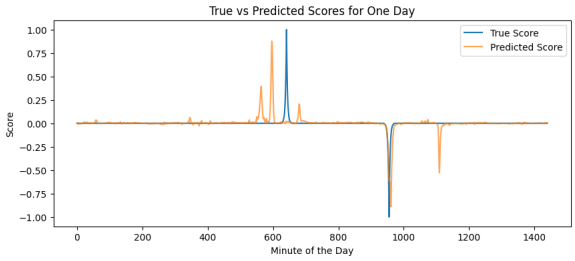
- It generated more distinguishable peaks in the prediction output
- The exponential decay created a natural confidence scoring mechanism
- It allowed the model to detect multiple potential events with varying confidence levels
- The smoother gradient helped the model learn the transition between sleep states



(a) Linear Target



(b) Composite Target



(c) Exponential Target

Figure 4: Comparison of different target engineering approaches. Note how the exponential target (bottom) creates more distinct peaks, allowing for better detection of multiple sleep events - a critical advantage for maximizing MAP score.

As shown in Figure 4c, the exponential target produced clearer, more distinctive peaks compared to linear and composite approaches. This resulted in more potential event predictions with varying confidence scores, which is ideal for optimizing the area under the precision curve used in the MAP evaluation metric. We encoded onset events with positive values (peaking at +1) and wakeup events with negative values (peaking at -1), allowing the model to simultaneously predict both event types with a single output:

```
def create_continuous_labeling(train_series_df, train_events_df,
    decay_tau=2.8, smoothing_steps=180):

    # Initialize score column
    merged_df = train_series_df.copy()
    merged_df['score'] = 0.0

    # Map event type to value: onset = +1, wakeup = -1
    event_sign = {'onset': 1.0, 'wakeup': -1.0}
    train_events_df = train_events_df.copy()
    train_events_df['event_val'] = train_events_df['event'].map(
    event_sign)

    # Group by series_id
    grouped_series = merged_df.groupby('series_id', group_keys=False)
    grouped_events = train_events_df.groupby('series_id')

    def apply_event_smoothing(group):
        sid = group['series_id'].iloc[0]
        group = group.sort_values('step').reset_index(drop=True)
        n = len(group)
        event_signal = np.zeros(n)

```

```
        if sid in grouped_events.groups:
            events = grouped_events.get_group(sid)
            step_to_index = {step: idx for idx, step in enumerate(group
    ['step'])}

            for _, evt in events.iterrows():
                evt_step = evt['step']
                sign = evt['event_val']
                if evt_step not in step_to_index:
                    continue
                center_idx = step_to_index[evt_step]

                for offset in range(-smoothing_steps, smoothing_steps +
     1):
                    idx = center_idx + offset
                    if 0 <= idx < n:
                        weight = np.exp(-abs(offset//12) / decay_tau)
                        event_signal[idx] += sign * weight

        group['score'] = np.clip(event_signal, -1, 1)
        return group

    merged_df = grouped_series.apply(apply_event_smoothing)
    return merged_df
```

Listing 4: Exponential Target Generation

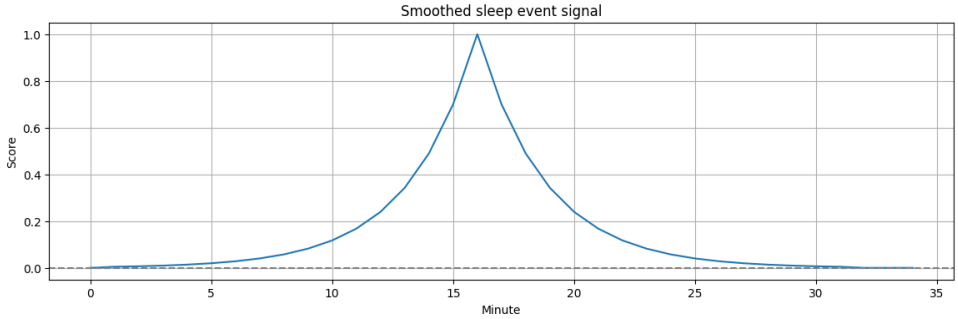


Figure 5: Example of a smoothed target signal generated using the exponential decay method.

4.3 Custom Loss Functions

One of our major innovations was the development of sophisticated custom loss functions tailored to the specific requirements of the sleep detection task. We experimented with several variants and ultimately settled on a composite loss function combining two components:

1. **Weighted Focal MSE**: Emphasizes harder examples by dynamically scaling the squared error, while also increasing the weight of important events based on the magnitude of the true signal.

2. **Ranking Loss**: Encourages event points (where $|y_{\text{true}}| > 0.6$) to have higher predicted magnitudes than non-event points, using a margin-based approach.

The final custom loss function implementation is shown below:

```python
def custom_loss_improved(y_true, y_pred):
    # Weighted focal MSE
    alpha = 6.0
    gamma = 2.0
    weight = 1 + alpha * K.abs(y_true)
    error = y_true - y_pred
    focal_weight = K.pow(K.abs(error), gamma)
    mse_loss = K.mean(weight * focal_weight * K.square(error))

    # Ranking loss: Encourage event time steps (|y_true| > 0.6) to have higher |y_pred|
    pos_mask = K.cast(K.greater(K.abs(y_true), 0.6), 'float32')
    neg_mask = 1.0 - pos_mask
    pos_count = K.sum(pos_mask) + K.epsilon()
    neg_count = K.sum(neg_mask) + K.epsilon()
    pos_mean = K.sum(K.abs(y_pred) * pos_mask) / pos_count
    neg_mean = K.sum(K.abs(y_pred) * neg_mask) / neg_count
    margin = 0.6
    ranking_loss = K.relu(margin - (pos_mean - neg_mean))

    return mse_loss + 0.1 * ranking_loss
```

Listing 5: Custom Loss Function

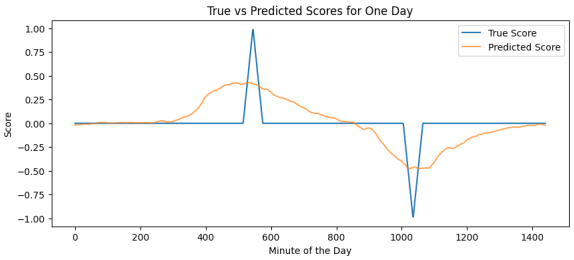
4.4 Model Architectures

We experimented with various deep learning architectures to find the optimal approach for sequence-to-sequence regression:

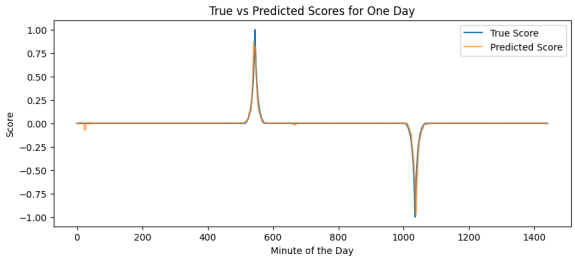
- **CNN**: Simple convolutional networks with 1D convolutions
- **LSTM**: Long short-term memory networks with attention mechanisms
- **TCN**: Temporal convolutional networks with dilated convolutions
- **UNet**: 1D U-Net architecture with skip connections and attention
- **Transformer**: Self-attention based architecture with positional encoding

The 1D U-Net with attention mechanisms emerged as our strongest individual model. Key architectural innovations included:

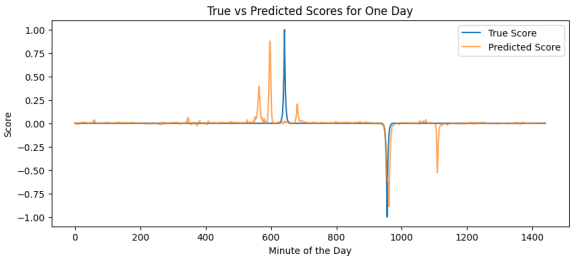
- **Attention Mechanisms**: Enhanced the model's ability to focus on relevant time steps
- **GeLU Activation**: Used instead of ReLU for better gradient flow and performance
- **Positional Encoding**: Added sinusoidal positional encoding for sequence awareness
- **Manual Epochs Limits**: Prevented overfitting by monitoring validation loss



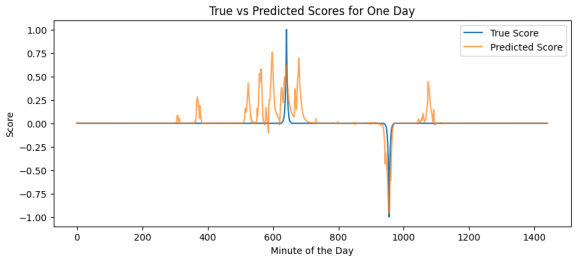
(a) CNN with Linear Target



(b) U-Net with Composite Target



(c) U-Net with Exponential Target



(d) LSTM with Exponential Target

Figure 6: Examples of model predictions using different architectures and target engineering approaches.

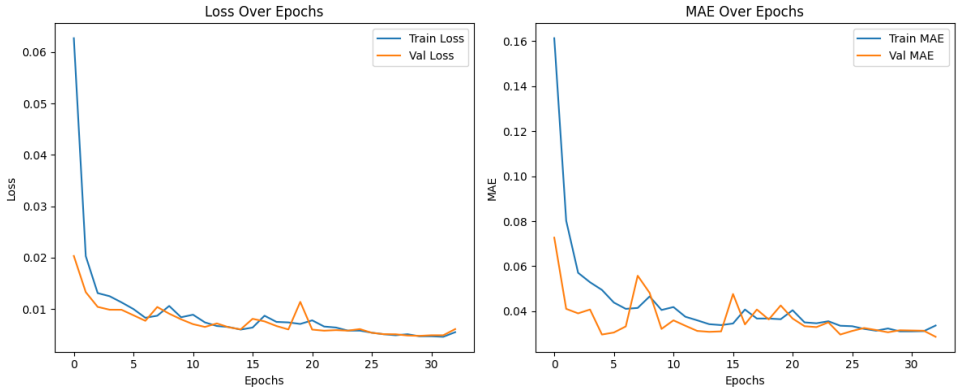


Figure 7: Training progress for the LSTM model using the exponential target.

```python
def build_unet_model(input_shape, filters=64):
    inputs = Input(shape=input_shape)
    x = PositionalEncoding()(inputs)

    # Encoder path
    conv1 = Conv1D(filters, 3, activation='gelu', padding='same')(x)
    conv1 = BatchNormalization()(conv1)
    conv1 = Conv1D(filters, 3, activation='gelu', padding='same')(conv1)
    conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling1D(pool_size=2)(conv1)

    conv2 = Conv1D(filters*2, 3, activation='gelu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Conv1D(filters*2, 3, activation='gelu', padding='same')(conv2)
    conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling1D(pool_size=2)(conv2)

    # Bridge with attention
    bridge = Conv1D(filters*4, 3, activation='gelu', padding='same')(pool2)
    bridge = BatchNormalization()(bridge)
    bridge = attention_block(bridge)
    bridge = Conv1D(filters*4, 3, activation='gelu', padding='same')(bridge)
    bridge = BatchNormalization()(bridge)

    # Decoder path with skip connections
    up1 = UpSampling1D(size=2)(bridge)
    concat1 = Concatenate(axis=-1)([up1, conv2])
    conv3 = Conv1D(filters*2, 3, activation='gelu', padding='same')(concat1)
    conv3 = BatchNormalization()(conv3)
    conv3 = Conv1D(filters*2, 3, activation='gelu', padding='same')(conv3)
    conv3 = BatchNormalization()(conv3)

    up2 = UpSampling1D(size=2)(conv3)
    concat2 = Concatenate(axis=-1)([up2, conv1])
    conv4 = Conv1D(filters, 3, activation='gelu', padding='same')(concat2)
    conv4 = BatchNormalization()(conv4)
    conv4 = Conv1D(filters, 3, activation='gelu', padding='same')(conv4)
    conv4 = BatchNormalization()(conv4)

    # Output layer
    outputs = Conv1D(1, 1, activation='tanh')(conv4)

    model = Model(inputs=inputs, outputs=outputs)
    return model
```

Listing 6: U-Net Model with Attention

4.5 Model Ensemble

Our highest performing solution combined multiple models with different strengths. After extensive experimentation, we found that a weighted ensemble of U-Net and LSTM models yielded the best results:

```
# Predict using both models
unet_predictions = score_predict_unet_model.predict(X)
lstm_predictions = score_predict_lstm_model.predict(X)

# Weighted ensemble (0.6 UNet + 0.4 LSTM)
filtered_buckets["predicted_target"] = [(0.6*unet_predictions[i].flatten() +
                                         0.4*lstm_predictions[i].flatten())
                                        for i in range(len(filtered_buckets))]
```

Listing 7: Model Ensemble Strategy

The ensemble achieved a superior score compared to any individual model, with the optimal weights being 0.6 for U-Net and 0.4 for LSTM.

5 Post-Processing

Post-processing was crucial for translating continuous model predictions into discrete event detections that maximized the MAP metric. Our post-processing pipeline evolved significantly throughout the competition.

5.1 Initial Approach

Our initial approach used a simple logistic regression model to classify sleep vs. awake states, then identified the longest sleep period and marked its beginning as onset and end as wakeup. While functional, this approach achieved a modest score of around 0.247.

5.2 Peak Detection and Filtering

We evolved to a more sophisticated approach using peak detection to identify potential events from the continuous predictions:

```
# Find peaks for onset (positive peaks) and wakeup (negative peaks)
onset_peaks, _ = find_peaks(predicted_scores, height=threshold_onset)
wakeup_peaks, _ = find_peaks(-predicted_scores, height=threshold_wakeup
    )
```

Listing 8: Peak Detection

5.3 MOD15 Step Adjustment

A critical insight came from analyzing the competition discussion forums, where we discovered the "MOD15 trick". We found that the distribution of ground truth events had specific patterns related to the minute value.

We implemented a step adjustment function that shifted predicted steps based on empirical patterns observed in the training data:

```
def adjust_step(step, event_type):
    step = int(step)  # ensure it's a standard Python integer
    if step % 12 in [0, 6]:  # Specific minute patterns
        step -= 1
    # Pattern based on minute % 15 and event type
    e15 = event_type[0] + str((step // 12) % 15)
    if e15 in ['o1', 'o8', 'o9', 'w0', 'w4', 'w8', 'w12']:
        step -= 12
    elif e15 in ['o4', 'o12', 'w1', 'w5', 'w9', 'w13']:
        step += 12
    return step
```

Listing 9: MOD15 Step Adjustment

5.4 Duration-Based Filtering

To comply with the competition's sleep duration requirements, we implemented filtering based on the duration between onset and wakeup events:

```
min_sleep_duration, max_sleep_duration = 60 * 12, 1000 * 12  # in steps
max_sleep_break = 30 * 12  # 30 minutes in steps

```

```python
# Apply duration filtering to onset-wakeup pairs
for onset_step, onset_score in zip(onset_steps_adjusted, onset_scores):
    for wakeup_step, wakeup_score in zip(wakeup_steps_adjusted,
    wakeup_scores):
        duration = wakeup_step - onset_step
        if duration < 0:
            continue  # Wakeup must be after onset

        # Accept if proper sleep duration or short break
        if (min_sleep_duration <= duration <= max_sleep_duration or
           (0 < duration <= max_sleep_break)):
            # Add to submission with confidence scores
            if abs(onset_score) > threshold_onset:
                submission_rows.append({
                    'series_id': str(row['series_id']),
                    'step': int(onset_step),
                    'event': 'onset',
                    'score': float(abs(onset_score))
                })

            if abs(wakeup_score) > threshold_wakeup:
                submission_rows.append({
                    'series_id': str(row['series_id']),
                    'step': int(wakeup_step),
                    'event': 'wakeup',
                    'score': float(abs(wakeup_score))
                })
```

Listing 10: Duration-Based Filtering

5.5 Optimal Threshold

Finding better threshold values for peak detection was essential for maximizing the MAP score. Through experiments, we found that threshold values around 0.05 for both onset and wakeup events was the best.

6 Chronological Improvements- Our Log

In our learning process, we have submitted over 290+ notebooks with different ideas and models. We have summarized some of the 290 ideas / tunings into a log of the ideas and their impact on the MAP score:

No.	Idea	Contribution	Score	Comment
1	Linear regression sleep/awake	-	0.162	*Starting point*
2	Worn/unworn model (baseline)	Improved	0.247	
3	Target wakeups/onsets with CNN	Bad	<0.1	Linear smooth target, bug?
4	Best smoothed target wakeup/onset	Improved	-	Target tuning
5	UNet with different params	Improved	0.554	Loss function tuning
6	Transformer with early stop	Bad	0.34	Overfitting?
7	UNet with best params + attention	Improved	0.597	
8	LSTM	Didn't improve	0.592	
9	Post-process using score of 1 on argmax	Not good	0.529	
10	LSTM + UNet combined	Improved	0.629	Ensemble
11	Double prediction (Union)	Bad	0.556	
12	UNet with max groupby buckets instead of mean	Improved	0.606	
13	Score with max min buckets on combined	Improved	0.636	
14	Using MOD15 trick (adjust step)	Improved	0.649	Post-processing
15	Best combination of UNet+TCN+Transformer	Improved	0.658	Ensemble
16	Multiple worn unworn models combined	No difference	0.658	XGBoost alone best
17	Fill forward short buckets in pre-process	Improved	0.687	Pre-processing
18	Combined with different losses	Improved	0.689	Peak finding
19	Exponent target smoothness	Neutral	0.677	Target tuning
20	Best post-processing (longest sleep, break, MAP thresholds)	Improved	0.736	Post-processing
21	Improving peak thresholds + 0.6 UNet + 0.4 LSTM	Improved	**0.756**	**Final version**
22	Training with mod15 feature	No difference	0.753	Feature engineering
23	Fourier transform for feature extraction	Not successful	-	
24	Multiple worn/unworn models ensemble	Not successful	-	XGBoost alone performed better
25	Training with minute %15 feature	Not successful	-	
26	Different SGD optimizers	Not successful	-	
27	Early stopping in U-Net	Not successful	-	Replaced with manual epoch limit of 300

Table 1: Chronological improvements and their impact on the MAP score

7 Summary and Results

Our final solution achieved a MAP score of 0.756, earning us a medal in the Kaggle competition. This represents a substantial improvement over our initial score of 0.162 (0.247 in the midpoint review) and demonstrates the power of iterative refinement in machine learning projects.

The key components that contributed to our success were:

1. **Sophisticated Preprocessing**: Custom bucketing, feature engineering, and normalization, target functions (exponential decay)
2. **Effective Worn/Unworn Classification**: Accurate filtering of unworn periods
3. **Advanced Model Architectures**: U-Net with attention and LSTM models
4. **Custom Loss Functions**: Tailored to the specific requirements of event detection
5. **Ensemble Approach**: Combining the strengths of multiple models
6. **Sophisticated Post-processing**: Optimized step adjustment and threshold tuning

We achieved all our initial goals outlined at the beginning of the course:

- We successfully implemented advanced techniques including feature engineering and model optimization
- We developed effective methods for handling rare events such as onset and wakeup
- We gained valuable hands-on experience in time-series analysis with accelerometer data
- We learned to optimize for a specialized evaluation metric (MAP)

This project contributed to our final success and deepened our understanding of time-series analysis for event detection.

A Appendix: Code Notebooks

The full code for our solution is available in the following notebooks:

- `WorkshopML_preprocessing.ipynb`: Data preprocessing pipeline
- `WorkshopML_training_worn_unworn_model.ipynb`: Worn/unworn classification models
- `WorkshopML_training_score_predict_models.ipynb`: Sleep event detection models
- `Kaggle submission - score 0.756.ipynb`: Best-performing submission (0.756 score)