Delft University of Technology

Team Epoch IV

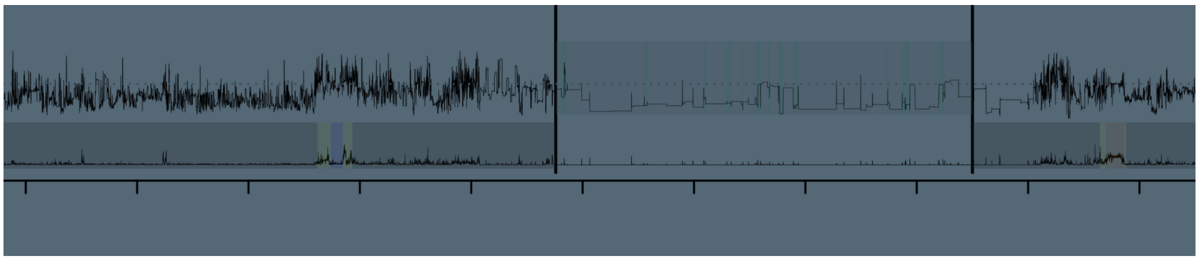
Subepoch 1

Child Mind Institute - Detect Sleep States

Detect sleep onset and wake from wrist-worn accelerometer data.

Engineering team:

Cahit Tolga Kopar
Emiel Witting
Hugo de Heer
Jasper van Selm
Jeffrey Lim



December 5, 2023

Preface

We've kicked off the year with tremendous momentum, securing the 23^{rd} position on the leaderboard with an impressive score of 0.816. We have learned a great deal from this competition about everything from data processing to hyperparameter optimization. This comprehensive report aims to dissect both our missteps and successes, providing a roadmap for future teams to excel in upcoming competitions by learning from our experiences and steering clear of avoidable mistakes.

This paper presents the outcomes of our team's participation in the recent competition focused on the detection of sleep onset and wake periods using wrist-worn accelerometer data. The objective was to develop a model that could reliably determine an individual's sleep state based on nuanced patterns in the accelerometer recordings.

Sleep monitoring is a crucial aspect of healthcare and well-being. Our research addresses the challenge of creating a robust model capable of accurately discerning sleep onset and wake periods. Leveraging wrist-worn accelerometer data, we aimed to contribute innovative solutions to improve the precision of sleep state detection. The work enables researchers to conduct more reliable, larger-scale sleep studies across a range of populations and contexts. The results of such studies could provide even more information about sleep.

The successful outcome of this competition can also have significant implications for children and youth, especially those with mood and behaviour difficulties. Sleep is crucial in regulating mood, emotions, and behaviour in individuals of all ages, particularly children. By accurately detecting periods of sleep and wakefulness from wrist-worn accelerometer data, researchers can gain a deeper understanding of sleep patterns and better understand disturbances in children.

Contents

1

Embarking on a Journey

It is the start of the year and we need to select our first competition. How do we actually do this? We start scanning all popular AI competitions hosting platforms for competitions. Then we create a long list where we evaluate the competitions according to the following criteria: Our sustainable Epoch Goals, Timeline, Prize/Popularity, Educational Value, Originality and Feasibility. After creating a method of selecting a competition, the process of selecting the current competition was started.

Five competitions made it into the shortlist. The first two were City Learn Challenges. These competitions were about energy and temperature control and forecasting. These were not selected due to being too similar to previous competitions and the code not being able to run within 2 days. A third option was the Google – Fast or Slow? Predict AI Model Runtime competition on Kaggle. After some days of research, the conclusion was that this competition would be too hard to do with limited experience and understanding of the topic. For the fourth competition, the Melting Pot Challenge was selected. This competition was about creating a reinforcement learning agent that could cooperate in multiple environments and scenarios. This was not selected due to reinforcement learning agents being very hard to train and taking a lot of time although this was a good second option.

During the research phase a new competition opened, the Child Mind Institute - Detect Sleep States competition. In this competition, participants had to detect sleep onset and wake-up based on accelerometer data. It was chosen as the knowledge required was less than other competitions so it was deemed the best option for the first competition.

This report aims to share our findings during the competition, highlighting major problems, what went well and how teams can improve in the future. The first two chapters will go over the planning and research phase, sharing the teamwork processes used and how the research was conducted. Research consisted of several parts: exploring the data and performing analysis; doing research on models; and how the submissions and scoring work on Kaggle.

Following the planning and research phase, most other aspects were done throughout the project. Data processing was a very important part of this competition and we started by setting up a data pipeline to reduce memory, add features and downsample the data. Four models were created which included: UNet, a transformer encoder, GRU, and a spectrogram. Out of these four, three were eventually used in the ensemble. During the competition, multiple approaches were thought out on how to make predictions such as regression or segmenting the series on asleep state.

These models using the approaches had to be trained and there were a lot of improvements made to speed up and stabilize training. One of these was finding a good loss function and making sure the models do not overfit. After training the models can make predictions which are then postprocessed using some methods to improve the score with sinc interpolation, offsetting the prediction and others.

2 A Blueprint for Achievement

2.1. Phases

At the start of the competition, it was decided to split it into 4 phases: Research, Baseline, Improvement, and Optimisation. The research, baseline and optimisation phases were two weeks and the improvement phase was six weeks. The research phase mainly focused on getting familiar with the state-of-the-art models in the field of the competition. Furthermore, an exploratory data analysis was done to inspect the data we are working with. This setup worked well until halfway through the baseline phase. Almost a day into the baseline phase a simple model was already created and already started being improved. The amount of time required to get to a baseline was therefore overestimated and for most competitions, especially on Kaggle can be done a lot faster. After this, following the phases seemed unnecessary so it was decided to throw it away and work more iteratively.

2.2. Workflow

Working in sprints seemed the best way to approach the workflow and to use an issue board. At the start, a sprint was one week with a start on Tuesday and a reflection on Monday. Having the reflection and a sprint startup separate was not very efficient so they were combined into a one-and-a-half-hour meeting on Tuesday morning called the sprint-storm. We found that having a meeting one time a week was very little as a lot of new information was discovered throughout the week. To work with this an emergency meeting idea was introduced where we could have a meeting after lunch if we needed to share important information. In actuality, we hardly ever used this concept and mainly had discussions when needed. This is something that could be improved in the future as often not everyone knows the newest findings but sometimes it is also not necessary. Finding a balance between sharing information with another member or the entire team is something that should be worked on in the future.

Next to working in sprints, we created issues to manage and assign tasks, using the GitLab platform. Issues worked well during the programming phase but not a lot during the final stages of the competition. This is mainly because throughout most of the competition, we had code or research deliverables but at the end, we were only trying out new combinations and hyperparameters using our existing code base. Each issue had labels and a weight/time estimate assigned to get an overview of how long each one would take. A burndown chart was not very useful as often we created issues in the middle of sprints due to issues popping up or good new ideas that had to be tried out instantly.

2.3. Code Quality

To ensure a high standard of code quality, we adhered to multiple principles. Firstly, at the start of our baseline phase, we set up a GitLab continuous integration (CI) pipeline that ran style checks and unit tests. This pipeline was run for every merge request, so we made sure that our main branch was kept clean. For style checks, we used the Flake8 package which prevents things like syntax errors, typos, bad formatting, and incorrect styling. Everyone had to make sure their code was clean since this was enforced in the pipeline. Everyone also had SonarLint as a plugin installed locally, which also caught some code smells. There is also a cloud option of SonarLint so that we could run it on the GitLab CI, but that was quite an expensive plan so the team decided not to invest in that.

2.4. Testing

When developing our code, we mainly wrote unit tests for complex functions that also helped us debug. In the initial stages of the competition, the team expressed the desire to implement automated integration and system tests. However, this aspiration presented a significant challenge: hosting our complete dataset online would demand considerable time and computational power. Therefore, the team decided not to make any integration and system tests, but to only make unit tests. To write the tests, the "unittest" package in Python was used which made it very straightforward to start creating unit tests. During development, we could have made our code more modular for increased testability because we had some parts of the code that were dependent on each other which made it hard to test individually.

3
Breaking New Ground

3.1. Exploratory Data Analysis

At the beginning of the research phase, an exploratory data analysis was made using Jupyter Notebooks. Here are some interesting statistics that were observed in the data:

- In total, sleep was recorded for 277 different people with an average of 26 night per person.
- 34% of the data are NaNs. Only 11% of the series did contain no NaNs at all.
- Multiple series that had data that looked like normal sleeping behaviour were unlabeled.
- There were 3 occurrences where there were multiple onset events after each other in the data, which is illegal. Every onset should be matched by a wakeup. This was later fixed in a new dataset released by the competition host.

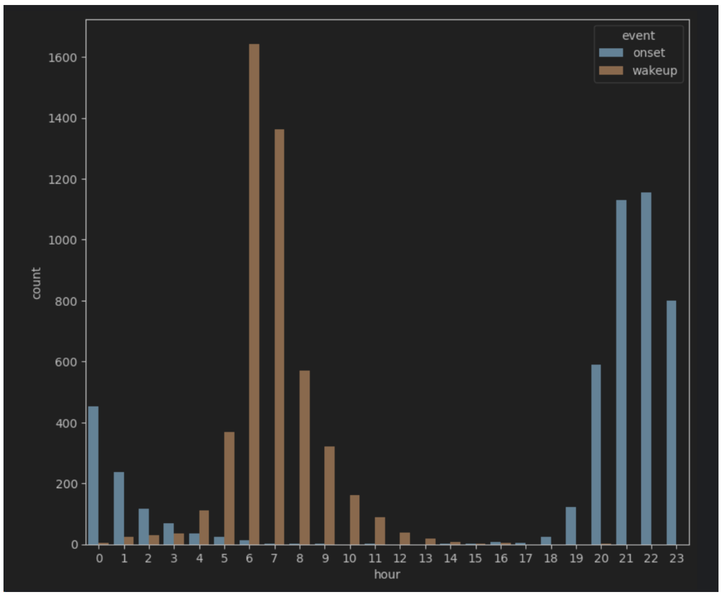


Figure 3.1: Average onset and wakeup times of all persons in the data

After the EDA, the team knew that they were up for a challenge mainly due to the labelling inconsistency and many NaN values that had to be handled by the models. NaNs were intentionally left in by the competition organizers since they could represent people not wearing the watch for example. Also, a lot of series were unlabelled and looked like they contained normal behaviour. An example can be seen below, where parts of the series has been annotated; Labelled means this part of the series was annotated with sleep onset or awake labels. NaN was not labelled and unlabelled in green shows parts of the sequence at the end that were also totally unlabelled. The orange segments actually look like normal sleeping data, but here we can observe that there were not labelled at all for these periods in the series.

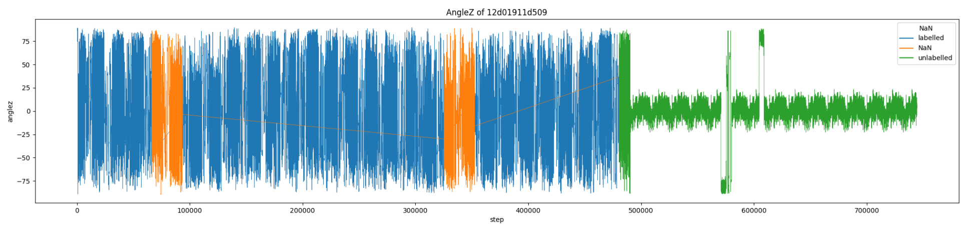


Figure 3.2: AngleZ feature of improperly labelled series.

3.2. Existing solutions

Before we start thinking of solutions, it is very important that we understand why this is a competition in the first place. It means that there are major limitations to existing solutions, which is where the challenge lies.

The gold standard for detecting sleep stages is polysomnography (PSG). This requires a patient to be connected to various devices on multiple location on the body. Not only will this disturb the sleep in and of itself, but it is especially troublesome with children, especially those who already may have trouble sleeping.

A different solution to detect sleep is to use a smartwatch. Research has shown that modern smartwatches can be almost as accurate as PSG in detecting sleep onset and wakeup times. One problem, however, is that these smartwatches are equipped with more advanced sensors. Modern smartwatches can not only detect motion with both an accelerometer and gyroscope, but they can also detect heart rate, blood pressure, snoring, respiratory rate, and even skin temperature. Because of these sensors, they are a lot more expensive than simple actimeters, which only have an accelerometer. Another problem is that these smartwatches use proprietary software for detecting sleep, which is not freely available to scientists.

Scientists have made efforts in using actimeters with an accelerometer only. The state-of-the-art algorithm for processing wrist-worn accelerometer data is the HDCZA algortihm. Not only has this only been tested on adults, but it is also not accurate enough and does not deal well with periods of non-wear.

Based on this, we know that the challenge lies in only having accelerometer to work with, and dealing with messy data. Especially the messy data is what we should put our focus on during the competition.

3.3. Domain Research

Obtaining more domain knowledge can potentially give you the winning edge in an competition. Most participants are so focused on finding the best models, which is important too, but domian knowledge can be just as important.

3.3.1. About Sleep

We performed some research on sleep in general. We found information about recommended and average amount of sleep, about circadian rhythms, and sleep in children. We did not find that much useful information in this regard.

3.3.2. About the Demographic

Researching how the data was collected and with what demographic proved to be more useful. We started with doing more research on Child Mind Institute and the Healthy Brain Network (HBN), the initiative for creating a huge open biobank of medical data from children. We found that CMI only has locations in New York, which makes it very likely that the patients the data was collected from also live in New York. This proved to be quite useful; more on that later.

We also found the specific study for which HBN collected the data. It's called the Rythms & Blues study and more information about that can be found on the HBN webiste. The most important result for us was that we found out that the participants of this study are teenagers from ages 11-14 with various sleep disorders. We also found out that children are supposed to log their sleep themselves using the MindLogger app, which would explain the low quality of the labels.

3.4. Similar competitions

Looking at similar competitions was a good way of starting the competition by looking at the models and approaches used there and then continuing to research those. One competition was found that was very similar to this competition and was researched intensively, the Parkinson's Freezing of Gait Prediction. The goal of that competition was to detect the freezing of gait from data collected by a 3D lower back sensor. Since the data was also accelerometer data and participants had to find changes in acceleration to segment time series acceleration data, models used for this competition would probably be very applicable to the current one. The first place solution used a combination of a transformer encoder with two bidirectional LSTM layers, patching and reducing the resolution of targets. For the second place solution, the participants used an ensemble of GRUs. Lastly, the rest of the solutions were similar to the first two except the sixth which used a combination of spectrograms, wavelets, UNet, ResNet and transformers.

3.5. Transformers

After finding that transformers were used in similar competitions previously, more research was done into them, specifically transformers applied to time series data. From the research, it was found that transformers were often used for time series forecasting such as in stock prediction. This was because the encoder-decoder structure is good at predicting values based on input. To use the transformer for the current application, the decoder part had to be cut off and replaced with a linear layer or something else such as a bidirectional LSTM layer.

3.6. Semi-supervised training

Another technique that was researched was semi-supervised training. Looking at the data, a lot of unlabeled segments were discovered that did not look like it was unlabeled due to non-wear. A possibility to get more data was therefore to use semi-supervised training. A couple of techniques and methods found were: Mean Teacher, Virtual Adversarial Training, MixMatch and LadderNet. These techniques were eventually not used due to time constraints but may have been successful.

3.7. Sequential Neural Networks

In the research phase, we already figured out that sequential neural networks were very promising for this competition due to the data being sequential and having a time dependency. During this research, we looked at 3 types of architectures: Recurrent Neural Network (RNN), Gated Recurrent Unit (GRU), and Long Short-Term Memory (LSTM).

3.7.1. Recurrent Neural Network

RNN is a type of artificial neural network designed for processing sequential data. Unlike traditional feedforward neural networks, that multiply the current input layer by the weight matrix, RNNs use hidden states that captures information about previous inputs in a sequence. This hidden state enables RNNs to model dependencies and patterns in sequential data, making them well-suited for tasks involving time series data, natural language processing, and other sequential data types.

Limitations

In essence, RNNs excel in tasks where the order and context of the data matter, making them a fundamental tool in various fields of artificial intelligence and machine learning. However, they do have limitations, such as difficulties in capturing long-term dependencies, which has led to the development of more advanced recurrent architectures like LSTM and GRU networks to address these issues.

Recommendation that arose from research

RNNs are not performing that well on long sequences, because of the vanishing/exploding gradient as seen below in the figure. Therefore, we would recommend to use patching (step segmentation) and aim to create smaller sequences, possibly splitting up series of multiple days into smaller ones of 1-5 days. In this way, we keep the sequence length manageable, and the network is able to learn better and suit this task well.

3.7.2. Gated Recurrent Unit

GRUs are a type of recurrent neural network (RNN) architecture designed to model and analyze sequential data efficiently. They were introduced as a simpler alternative to Long Short-Term Memory (LSTM) networks, another type of RNN. GRUs are known for their effectiveness in capturing short- to medium-term dependencies in sequential data while having a more streamlined architecture. Gating Mechanisms: GRUs use gating mechanisms similar to LSTMs, but with a simplified structure. They have two gates: an update gate and a reset gate. These gates control the flow of information through the network, allowing GRUs to selectively update the hidden state and handle different time steps in a sequence. Hidden State: Like LSTMs, GRUs have a hidden state that can capture information from previous time steps in the sequence. This hidden state can be thought of as a memory that retains relevant information over time.

Limitations

- **Limited Capacity for Capturing Long-Term Dependencies GRUs**. Like other recurrent neural networks, may struggle to capture very long-term dependencies in sequential data. While they are designed to mitigate the vanishing gradient problem, their capacity to store and retrieve information from distant time steps is still limited compared to more complex architectures like Long Short-Term Memory networks.

- **Sensitivity to Hyperparameters**. The performance of GRUs can be highly sensitive to hyperparameters like the number of units, layers, and learning rates. Finding the right hyperparameter configuration can be a time-consuming and sometimes challenging process.

- **Difficulty Handling Variable-Length Sequences**. GRUs, like most RNNs, are designed to work with fixed-length input sequences. Dealing with variable-length sequences often involves preprocessing steps such as padding or truncation, which can introduce inefficiency and additional complexity into the modeling pipeline.

Recommendation that arose from research

GRUs are definitely very suitable for this task, mainly because they excel in capturing short- to medium-term dependencies in sequential data. This fits well with the data that we have, which I would classify as short - medium length as well. A lot of submissions used GRUs in similar competitions, such as the Parkinson Freeze of gait one.

3.7.3. Long Short-Term Memory

An LSTM is a type of RNN architecture designed to overcome some of the limitations of traditional RNNs. LSTMs are specifically engineered to better capture and manage long-term dependencies in sequential data. They achieve this by introducing a more complex internal structure called a "memory cell" that can store and retrieve information over extended sequences, making them highly suitable for tasks involving sequential data.

Limitations

LSTM cells have some drawbacks when compared to simple RNN cells. They are more computationally expensive and require more memory and time to train and run due to their additional parameters and operations. Additionally, they are more prone to overfitting, necessitating regularization techniques such as dropout, weight decay, or early stopping. Finally, they are harder to interpret and explain than simple RNN cells since they have more hidden layers and states.

Recommendation that arose from research

Recommendation LSTMs are very suited for this problem since they could support longer time series due to not having big problems due the vanishing gradient for long series. We have to watch out for the fact that they might be harder to configure correctly and are more computationally expensive.

3.8. Convolutional Neural Networks (CNN)

We thought that CNNs could be a viable approach due to CNNs having convolutional layers that can handle spatial information available in images and sensor data. Therefore we looked at two types of CNNs, 1D-CNNs that work on sensor data and 2D-CNNs that work on image data such as spetrograms.

3.8.1. 1D-CNN

We researched that 1D-CNNs are useful for time series since they can take sequential data as input and use convolutions to learn sleep schedules and predict which state the series is in. Here we thought of multiple approaches, where the first one is state segmentation where the model outputs the probability of being in states 0, 1 and 2. Where 0 means asleep, 1 means awake and 2 is unlabelled / NaN (we don't want to make a prediction here). Then we could further post-process this to find the transitions from 0 -> 1 being a wakeup event and 1 -> 0 being a sleep onset event. We found a 1D-Unet architecture online here that we started to use as one of our baselines which already performed quite decently.

3.8.2. 2D-CNN

The main application of 2D CNNs on sensor data is to convert the sensor data to 2d spectrograms, similar to audio classification and speech recognition tasks. The difference between that case and our case is that for audio applications either 44.1kHz or 48kHz is used for sampling frequency. So 48000 samples is equivalent to 1 second. In our case the sampling frequency is 0.2 Hz so for us 48000 samples equates to 240000 seconds (66h40m). So the classification/segmentation methods used for audio applications are not directly applicable to our data. Still we could use large overlap between windows at the cost of computation time to provide better time resolution in the spectrogram and help preserve more temporal information.

3.9. Kaggle submission and scoring

Probably the most important thing to understand about the competition is the scoring metric. If we understand the scoring metric we can use it to our advantage. To this end we spent several days looking into the scoring metric code. The scoring is based on a complicated implementation of the area under the precision recall curve (see 8.1 and plot-precision-recall).

The precision-recall curve shows the trade-off between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. (plot-precision-recall).

Simply put this metric rewards or penalizes your predictions depending on how confident you are with the use of different thresholds. If you make a prediction with high confidence and it is wrong, you will be penalized more compared to making a wrong prediction with a lower confidence. This means that making submissions with proper confidences instead of using a global confidence will have a huge positive impact on the score since the predictions too far away from an event get penalized less.

4

The Engine Room: Data Processing

4.1. Preprocessing

4.1.1. Memory reduction

Memory reduction was an important part of this competition as the original data set was very large and since a notebook had to be submitted with limited memory, to create a successful submission, memory usage had to be reduced. Some columns were using more memory than what was required such as step, enmo, anglez and timestamp. Reducing these to the minimum value needed decreased our memory by 90%. Additionally, each series was saved separately in parquet files so the series_id did not have to be stored in a column, resulting in an additional decrease in memory usage.

4.1.2. Labeling

Since the targets and data were split into different files, a processing step was to combine them into one file. This could be done in different manners which are examined below.

Event labels

The event labels are two columns where for each row 0 represents no event and 1 represents an event. To help with the training and to predict multiple confidences, Gaussian smoothing is applied to the column. This allows for easier training for the model as the label distribution is more balanced. There are two parameters for this function, smoothing and steepness. Steepness represents how fast the labels transition from 1 to 0 while smoothing represents how smooth the transitions between rows are.

State labels

For the state labels a state was added to each row/step of the data. The state had four options: 0 for asleep, 1 for awake, 2 for NaN and 3 for unlabeled data. Adding these labels had three parameters. For the first parameter, 'use_similarity_nan' it used the similarity_nan column to fill in the correct state. The next parameter, 'fill_limit' was used for the maximum number of steps to fill in. Lastly, a NaN tolerance window was used to calculate the number of steps to tolerate NaNs before filling in the state column. An additional implementation of this was added that added the states in a one-hot encoding manner.

4.1.3. Splitting windows

Feeding all the data into a model is usually too much for a model to handle. Additionally, each series is a different length, sometimes differing by multiple days or even weeks. To account for this we decided to split each series into a certain window length. It was decided to split the series into windows of length 17280 as this represented the number of timesteps in a day. The window start was set at 3 pm as this was where the least amount of onsets and wakeups occurred as seen from the exploratory data analysis.

4.2. Feature Engineering

Extracting features from the time series data resulted in significant improvements. We found features partly by reasoning about the problem, and with a brute force search using a heuristic for feature importance.

4.2.1. Rotation

When interacting with the data, we found that anglez changes very infrequently when the subject was asleep. Thus, taking the slope (diff) of anglez, then absolute value, then clipping it gave a very useful feature. Smoothing the peaks of the result with a rolling median resulted in a mostly accurate indication of a person being awake or asleep at a given point in time. As it relies on the change in angle, we named this “rotation”.

4.2.2. Time

We extracted the features hour and minute from the timestamp. This proved useful according to our feature importance analysis. Hour is an indication of sleep, as people generally go to bed and wake up at similar times. This makes it easier for the model to infer when wakeups and onsets occur.

4.2.3. Non-wear detection (Similarity NaN)

We noticed that there is a repeating pattern in the non-wear periods, for which predictions should not be made. Strangely enough, the pattern repeated perfectly for every 24 hours, although it was different per child. To detect this, we computed the vector difference between each combination of 24 hour windows, then created a feature from the smallest difference at each time step. This distance would be zero for points in the repeating pattern. As this value uses the similarity between every 24 hours, and indicated NaN/missing events, we named it Similarity NaN.

4.2.4. Sunlight

Knowing the data was recorded in New York, we can calculate the position of the sun and use the azimuth and altitude as features. Since sunlight affects the circadian rhythm, it strongly correlates to when people fall asleep and wake up. Because of this strong correlation, our scores increased significantly with this feature.

4.2.5. Holidays

Since we can assume that the data was collected from 11-14 year old children in New York, we can add all the middle and high school holidays as well. Unfortunately, it seemed to only decrease our score.

4.2.6. School Hours

Like with the holidays, since we can assume that the data was collected from 11-14 year old children in New York, we can add the school times as a feature too. School hours are from 8:00-15:45 for middle schools and high schools on Monday to Wednesday. For Thursdays and Fridays, high schools end at 16:20. Unfortunately, this feature did not increase our score.

4.2.7. Weather

With the timestamp as well as a longitude and latitude in New York, we used Meteostat to add various weather features to the data. However, this gave us lower scores on average. It also required a workaround for saving historical weather data, as our notebook had no internet access to the Meteostat API.

4.2.8. Feature importance and selection

Besides these manually created features, we automatically generated more features and estimated their importance. Generation was done by computing all combinations of rollings statistics (rolling mean, median, var etc.) applied to each feature, with each of a range of window sizes. For some features, the diff was also applied to create another. Selecting went as follows:

First, a feature’s usefulness was judged based on its ability to predict the awake/asleep state. A first pass did this by fitting a logistic regression model on each feature separately, and scored the feature by the model’s AUROC classification score. This gives a good indication of how the features perform on their own.

Then, features were filtered. A loop iterated over features from highest to lowest score, and added them to the selection, unless the correlation to any previously selected feature was too high. This roughly halved the amount of features and improved the score in the next step.

To evaluate the usefulness when combining features in a model, Catboost was trained with the selection to perform sleep/awake classification. Catboost, like most GBDT models, gives direct access to the feature importance.

This process was repeated for two other problem settings. The first was a dataset consisting only of small windows around ground truth transitions. The latter had event labels. Combined, this now gives feature importance for: long-term state classification, short-term state classification, and event detection. The same correlation pruning procedure was used on each. The final features were a manual selection of the best of each category.

4.3. Pretraining

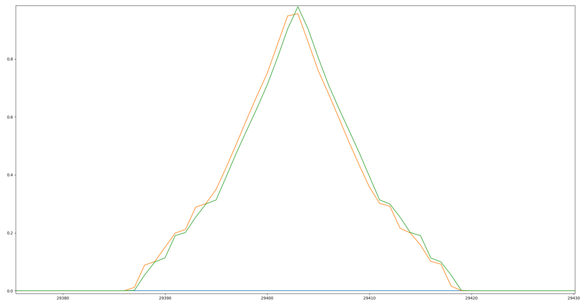
4.3.1. Downsampling

In the critical point notebook that used the GRU model, they downsampled the series to create features, calculating statistical values of every 12 items like mean, median, and variance and using these statistical values as the features the model is trained on. We have added some more methods to this such as range min and max and used these for all our models. The downsampling factor of 12 gave the best overall performance and was not changed.

All our final models ended up with a GRU model at the last layer, which meant we also had to downsample our labels because a GRU model returns 1 result per time step in the input sequence. (The GRU could not be made bigger due to memory limitations, and we did not use upsampling networks at the output). For downsampling our labels, we tried two approaches:

- Subsampling: Taking the first label every 12 labels
- Median downsampling: Taking the median of every 12 item in the labels

Figure 4.1: Comparison of different label downsampling methods. The orange curve is the median downsampled and the green one is subsampled labels for one event



From figure 4.1 we can clearly see that the median subsampling is skewed to the left and would be expected to perform worse. However with some post processing the median downsampling ended up giving us higher scores. (see 8.2.3)

4.4. Caching

The preprocessing, feature engineering steps, and pretraining can take a long time to run. To reduce waiting times, we cache intermediate results files. While this does increase storage usage, it drastically reduced waiting times. We use a hash of the configuration so we can detect when the configuration has changed so that we know when it is appropriate to reuse the cached data or not.

5 Precision in Practice

5.1. 1D-UNet

A 1D Unet is a type of CNN that is effective for segmentation tasks where the goal is to classify each step into a category. The downward arrows in the figure represent the contraction path of the encoder part of the U-Net, where convolutional layers are used to capture the context of the input image. The network learns features at various resolutions. The vertical bars could represent the feature maps generated by each convolutional operation. Pooling layers (not explicitly shown) are typically applied after each convolution to reduce the spatial dimensions of the feature maps.

The horizontal arrows represent skip connections where information flow from the encoder is concatenated with later flow to the decoder. In this way, the network can use features learned at different scales to improve precision.

The upward arrows increase the resolution of the feature maps, this means increasing the window size while decreasing the number of channels by applying deconvolution. The output of the UNet consists of the same window size where the model predicts the state of each step in a one-hot encoded way where the number of channels is the number of classes to predict.

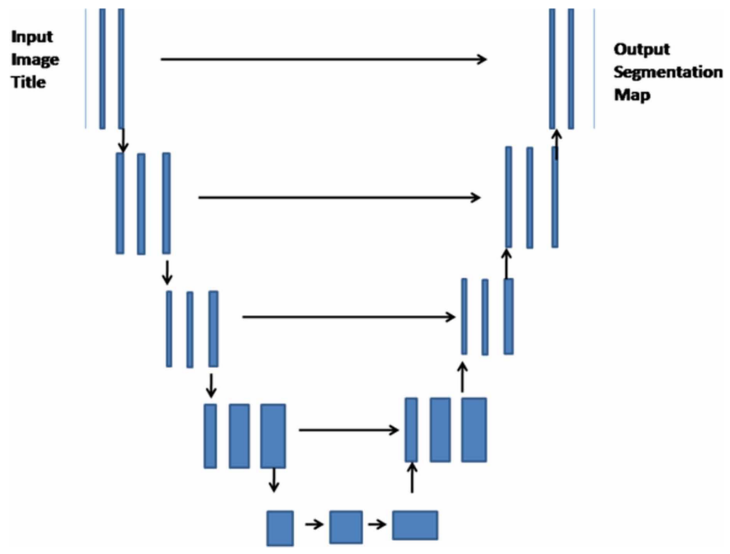


Figure 5.1: UNet Architecture diagram

5.2. Transformer Encoder

Research showed transformers had the potential for time series analysis and were therefore implemented in our code and used during ensembling. Getting the transformer to work was quite difficult due to memory issues and starting with the wrong approach: regression. In this section the final architecture created will be covered and some things to watch out for while using and training a transformer.

5.2.1. Patching and memory usage

The transformer uses a lot of memory as the attention matrix uses $O(n^2)$ space where n is the sequence length. To circumvent this, the input data was patched. This meant that the sequence length was divided by the patch size and the number of features per step multiplied by the patch size. After the data was patched, it was fed through a linear layer to get the last dimension equal to the transformer embedding dimension.

5.2.2. Encoder

A transformer usually consists of two parts, an encoder and a decoder. The decoder is special as it can take previously generated output and combine it with the encoder input to generate new values. This is very useful for LLMs and forecasting problems. Since we are doing segmentation in the current competition, the decoder part is not as useful and was therefore cut out. The encoder architecture can be seen in figure 5.2.

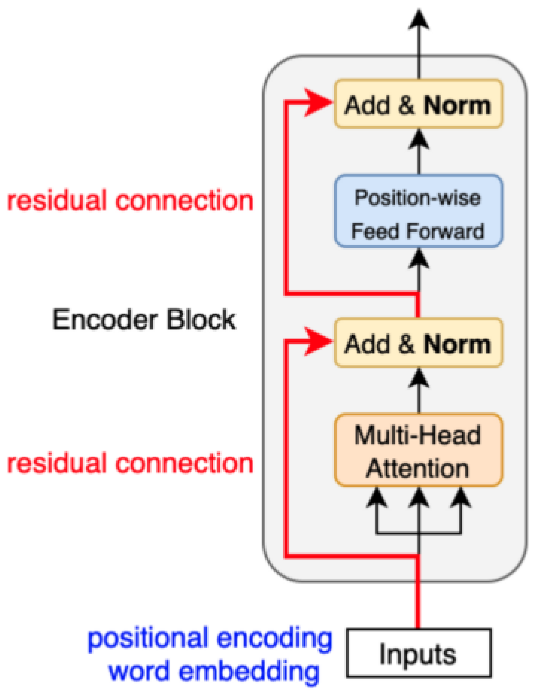


Figure 5.2: Transformer encoder.

Positional Encoding

To allow the transformer to get a better insight in the positions of features in the data, a positional encoding is added. There were multiple options that were experimented with during the competition: Fixed, Learnable and Other. Fixed positional encoding ended up being the best and uses sine and cosine values.

Attention

Another part of the transformer encoder is the attention calculation. This process is very memory intensive so a large focus during the competition was to decrease the memory used. Research into multiple attention types was done and the following were experimented with: Normal, Blocksparse, and Bahdanau. These mostly performed the same but the memory usage of blocksparse was a lot less and was therefore used in the final model.

5.2.3. Residual model

After passing the data through the transformer it was summed with the original data. Once it was summed it was passed through a bidirectional GRU which is fully explained in section 5.3. The full architecture can be seen in figure 5.3

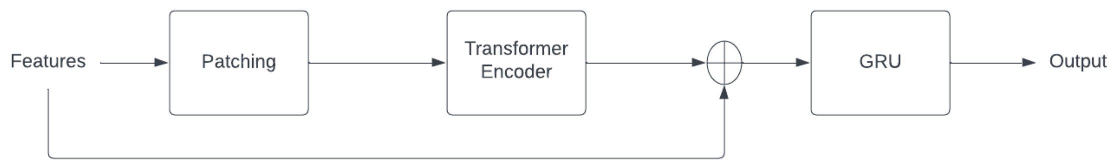


Figure 5.3: Final transformer architecture.

5.3. GRU

Our GRU architecture is based on a public notebook:
https://www.kaggle.com/code/werus23/sleep-critical-point-infer
This consists of multiple repetitions of a so-called ResidualBiGRU block, ending with a dense layer and an activation. This was named after the residual connection through addition at the end of each block, and the bidirectional GRU. We adapted this with three changes, the network can be seen below:

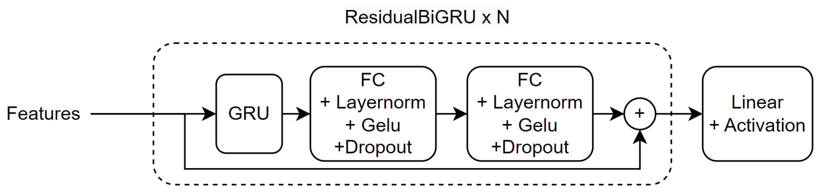

ResidualBiGRU x N
Features
GRU
FC + Layernorm + Gelu +Dropout
FC + Layernorm + Gelu +Dropout
+
Linear + Activation

Figure 5.4: MultiResidualBiGRU, with added dropout layers

The first change was the addition of dropout layers, dropout of 30% to 60% was used. This initially lowered the performance, but when paired with an increased architecture size gave a smoother validation loss and converged to a better score.

Secondly, we noticed that models would often converge to a local minimum early on, which is where it would predict all zeroes across the timeline. This is due to the events being very sparse. A model can easily predict all zeroes by giving a negative result, then passing it through an activation that maps negative values to zero. In this flat output range of the activation, the derivative is zero and the model cannot learn anymore, as gradient descent has no direction to go. These activations, like Relu and Gelu were useful later on in the training process, but made it easy to get the model stuck at this hurdle. So, an activation delay was added for the last layer, which forced the model to use a linear activation at the start, ensuring that it learns to not predict only negative values. Later, after e.g. 15 epochs, the activation is turned on, allowing better fine-tuning.

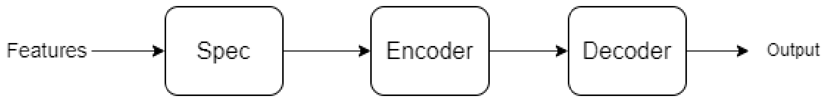
Lastly, we used shrinkage loss, which gives more weight to samples with a higher label value (the events), this performed better than MSE or MAE.

5.4. Spectrogram

This model is based on:
https://github.com/tubo213/kaggle-child-mind-institute-detect-sleep-states/tree/main
https://www.kaggle.com/code/werus23/sleep-critical-point-train

The GRU model is the same model used in the critical point notebook with tuned hyper-parameters. The original spectrogram model, as the name suggests, creates a spectrogram from the time series data. Then uses a pre-trained resnet34 model to encode the images. Lastly there is a CNN based decoder with 3 output channels that outputs event predictions and a state prediction (awake or not awake) per timestep (see fig 5.5). The model made in this codebase was not optimized in the least and did not even train on the full dataset for submmission. However it did get a better score than our best submissions so the architecture was copied over to work with our pipeline.

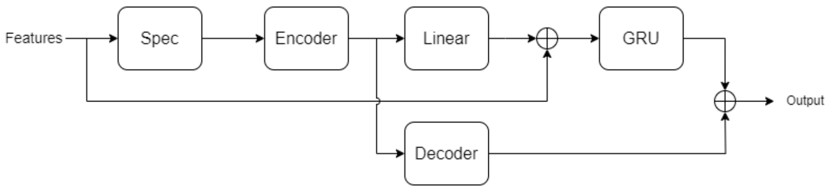
Figure 5.5: Initial Spectrogram Architecture



After copying the model we were still not getting better scores than the public model. However after we started to make multiple predictions per window our local scores jumped to 0.75-0.76 with the spectrogram model. At one point this was our best scoring model but work done on other models (mainly GRU) and feature engineering, the GRU models outperformed the spectrogram model.

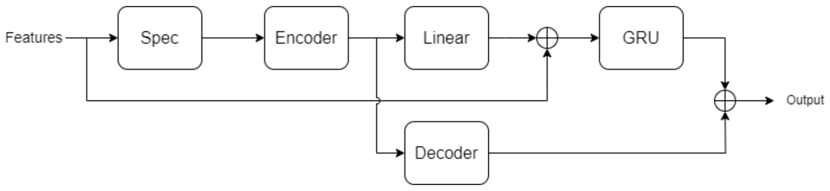
At this point the spectrogram model hit a plateau. Since some of the features that were made as a result of feature engineering were state information (altitude, azimuth) and not necessarily time series information they did not bring any meaningful information to the model. Because of this a new architecture was made to use the GRU model as a decoder with residual features and where only select features would be passed to the spectrogram and encoder layers (see fig 5.6).

Figure 5.6: Final Spectrogram Architecture



However this model could not outperform the original GRU model with the new features and it looked like the GRU model was not well suited for decoding the encoded spectrogram features. At the end a final architecture was made that utilized both the original decoder and residual features with the GRU model which ended up giving the best results (see figure 5.7).

Figure 5.7: Final Spectrogram Architecture



It was quite tricky to get this model to preform very well. Even though we had a model with 0.78 score locally after retraining with the same parameters the results could not be reproduced and it seemed like the model was too dependent on the random initialization. However less than 2 days before the deadline we created a sweep to try out different seeds and with this we were able to find a seed that again resulted in a 0.78 local score. In the last day this model was retrained (now with a fixed seed) with 5 fold cross validation and ended up giving the highest CV score we had (This was also the first time this model was cross validated since every fold took over an hour of training). This model was retrained on the complete dataset (now with fixed seed) and was included in our last submission. This submission ended up giving us our best public score.

6 Strategic Approaches

6.1. Regression

For regression, the model predicts two numbers per window which are the steps at which onset and wakeup events happen. At first, it seemed like a good approach but in practice yielded very poor results. Additionally, with the way scoring worked using confidence, it penalised a wrong prediction even more by having a confidence of 1. Lastly, it limited the models to make 1 prediction for each event per window which we later found makes the score worse.

6.2. State Segmentation

Our first (successful) approach was to treat the problem like a state segmentation task where we classified each step with the current state of the person, awake or asleep. State segmentation model outputs were then converted to event predictions using standard algorithms and signal processing (no ML) to figure out at what step an event occurs. However when the critical point notebook was released the event segmentation approach outperformed this approach and this approach was no longer pursued.

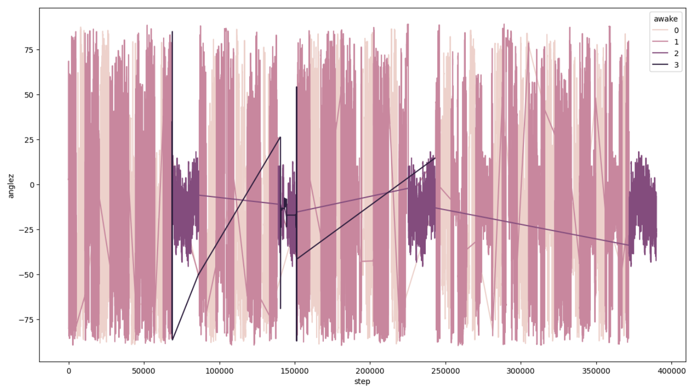


Figure 6.1: One series shown with hues of its state labels. 0 means asleep, 1 means awake, 2 means NaN and 3 means unlabelled (4.2.3). Note the jumping lines are caused by seaborn plot artifacts

6.3. Event Segmentation

This approach, inspired by the critical point notebook, instead of predicting the current state aimed to predict at which timestep an event occurred by fitting a gaussian curve to the events, effectively making this a regression problem where the model would learn to fit this smooth curve with the peak at the timestep of the events. In our approach instead of a gaussian curve, our labels were made to reflect the scoring metric by creating a blocky curve where the blocks represented the scoring metric. For +-12 steps away from the event the block has a height of 1 and from 12-36 height of 0.9 and so on (In the end making the windows half as big as the original scoring metric windows improved our score). However it was still tricky for the model to learn the sharp changes in the labels which is why the labels were also smoothed out with a gaussian filter.

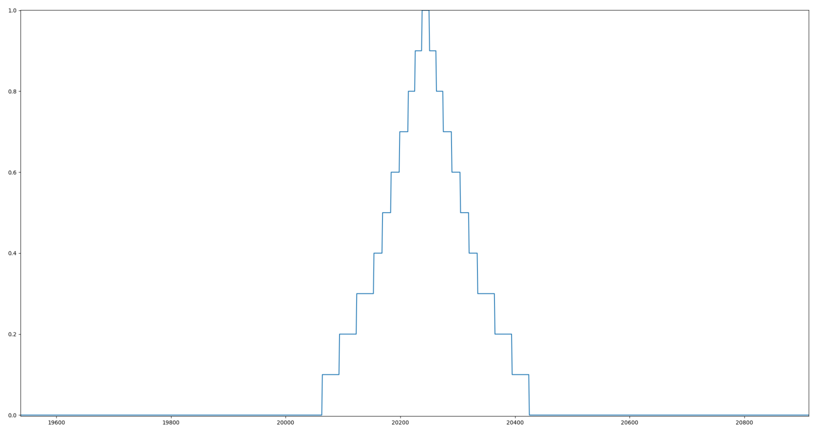


Figure 6.2: Blocky event segmentation labels

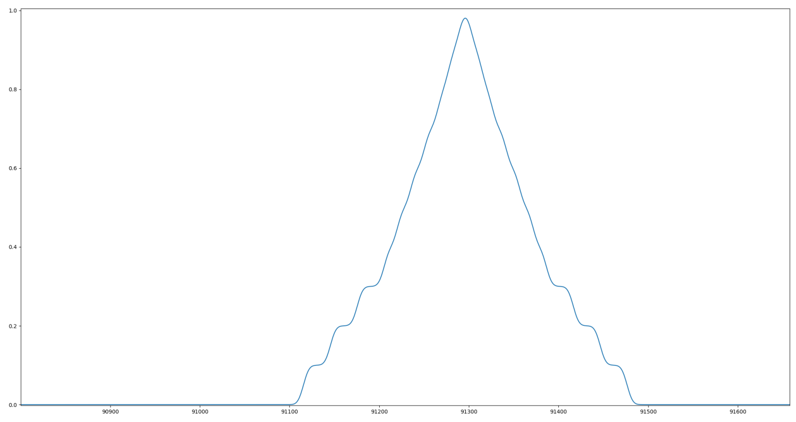


Figure 6.3: Smooth event segmentation labels

Since we had already tried and approach of regression calling this method regression would make it confusing. For that reason this method is called event segmentation.

6.4. Event and State Segmentation

Using both event segmentation and state segmentation approaches by making the models predict both events and current states and using a weighted loss between one hot encoded awake state predictions (awake, asleep or NaN) and the event labels. This approach did not give improvements to most models but the spectrogram model did improve with it. In the end only the spectrogram model used this approach.

7

On the Slopes of Success

7.1. Loss

There were many loss functions tested during the competition. The loss functions used and their performances for the event segmentation approach are explained in this section.

7.1.1. Mean Squared Error

Mean squared error is the most commonly used loss function for regression tasks. Since the event segmentation approach turned the problem in to a regression problem this loss worked quite well. However the models that trained with shrinkage loss outperformed ones trained on MSE loss and MSE was not used in the later stages of development.

7.1.2. Mean Absolute Error

Mean absolute error, like mean squared error, is commonly used in regression tasks. However in the early stages of development MAE was always underperforming in comparison to MSE and was not used further.

7.1.3. Kullback-Leibler Divergence Loss

The Kullback-Leibler divergence is a metric used to compare two data distributions. It contrasts the information contained in two probability distributions. Since our best approach was event segmentation where our model aims to predict a Gaussian around the onset and wakeup events, we thought of using this loss function for our models. During our hyperparameter tuning of our models, KL-Div loss was seen multiple times in our best-winning CV scores, making it a suitable loss function for our models.

7.1.4. Shrinkage loss

Shrinkage loss aims to address challenges related to the vast search area around target objects. The inclusion of a substantial background, surrounding target objects is crucial for providing valuable context information that enhances the discriminative power of these objects. However, this approach also introduces a significant number of easy samples from the background, leading to a large loss during the learning process. The absolute difference, denoted as l, between the estimated probability p and its corresponding soft label y is utilized to quantify the loss. To mitigate the impact of easy samples, the conventional square loss is modified using a modulating factor, resulting in the proposed shrinkage loss for regression learning. The modulating factor is shaped like a Sigmoid-like function and is applied to re-weight the square loss, penalizing easy samples while keeping the loss from hard samples unchanged. This strategy is designed to improve the tracking accuracy and accelerate the training speed. We found this loss to work the best while training our models, with the fastest training speed and the best results.

7.1.5. Masking

Since there was a lot of unlabeled data in the train set which did not look the same as the NaN data, we thought masking the loss for these steps would allow the model to train better as it does not get confused. From sweeping it turned out that it did not really matter whether the loss was masked or not, although some models did have a preference for masking or not masking.

7.2. Early stopping

Early stopping is a method to ensure the final trained model is not overfitting. The model is saved if it reaches a new minimum validation loss and training continues. If the validation loss starts to increase and does not decrease after a specified number of epochs, the stored optimal model will be used instead. In the last two weeks we have also used the validation score (score calculated on the validation set using the scoring metric of the competition) for determining if we should early stop. This approach gave better models for some architectures but worse models for others.

7.3. Learning Rate Scheduler

A learning rate scheduler is something that can change the learning rate for a model over epochs. We first saw it being used with the critical point notebook and implemented in our pipeline. It is generally accepted that learning rate should start high and slow down when converging. With the Adam optimizer, this is already adaptively implemented, which is usually sufficient.

What can also be done however with schedulers however is resetting the learning rate to a high value. This can be interpreted as giving the model a push out of the local minimum which it might have converged to. Alternatively, it can be seen as retraining the model starting with a good initialization: a warm start. We used a scheduler that's implemented in Pytorch as CosineAnnealingWarmRestarts. For certain models, this gave more consistent improvements than others. For the future, we could consider also looking into schedulers that reset the learning rate on Plateau's.

7.4. Adaptive Labels

One approach we tried to use was changing the labels for the model during training, making the event labels more narrow over epochs. However since we only managed to implement this idea 2 days before the deadline we could not find a proper set of parameters that lead to improvements on our models.

The methodology was the following: With the help of a lookup table we identify the original value of the label at a given width (the distance to the peak before downsampling). Then we figure out a coefficient k such that at a specified number of epochs the original value of the label at the given width will become equal to an epsilon value when taken to the power of k*current epoch. Even though this approach worked on paper with downsampling the labels ended up being too narrow for the models to learn anything meaningful. When setting the width to a larger value to accommodate for this issue the coefficient k became smaller than 1 which made the labels less narrow (large values around the peak) for some epochs which from previous experimenting is known to make the results worse.

$$k = \frac{log(eps)}{log(y_{\mathrm{k}}) * \text{epochs to min}} \tag{7.1}$$

Where $\mathrm{y_k}$ is the original value of the label at the given width before downsampling.

7.5. Cross Validation

To determine the quality of our models, we cross-validate each model. This means that we train a model on one slice of our dataset and test it with the rest. We do this for k folds and each fold uses different slices of the dataset. The average score of each model gives us a good indication of how well the model generalizes to new data and helps us prevent overfitting.

We initially used 80% of our training data (we already had a 80-20 split for train and test at this point) for the train splits of the folds. In the last week we have used all of our train data for cross validation and we ended up with better results.

7.6. Hyperparameter optimization

We used Weights and Biases to make finding the best settings for our model easier. We did this by trying out different combinations of settings, called sweeps. This made our process more organized and helped us quickly figure out the best setup. After just a day of trying different combinations, we consistently found the best settings for our model. We tested our model on all of our data using cross-validation, which helped us make sure it worked well across the board. Despite these positive outcomes, we realized that for future projects, we need to set things up even better when it comes to tweaking model settings.

8

Polishing Perfection

8.1. Scoring metric

The scoring metric uses the are under the precision-recall curve. However what makes it complicated is that a different curve is generated for 10 different tolerences (The allowed error to match a prediction to an event) and for each different event type (onset and wake-up). The method ends up making 20 different precision recall curves and averages the area from all these curves are averaged.

To create the precision recall curve all the unique confidences from the predictions is iterated over and in each iteration only predictions with confidences (probabilities) larger than that confidence are considered for the matching to real events. For each confidence in this loop a precision recall point is created, then at the end all these points are used to calculate the area under the precision recall curve with rectangular blocks to approximate the area.

8.2. Tricks

8.2.1. Sinc interpolation

Since we use downsampling for our features to get back the original dimensionality of our outputs we need to upsample our predictions. Instead of making convolution layers for upsampling the data we ended up using standard interpolation methods in postprocessing instead. We initially tried simply repeating the datapoints to get an upsampled signal but this was not optimal. After trying ideas like linear interpolation we ended up using sinc interpolation. This method, under certain assumptions, results in a theoretically perfect reconstruction of a sampled signal. We tried to use the interpolation for both the median downsampled (taking the median value of every 12 steps) and subsampled (1st value in every 12 steps). Even though the interpolation resulted in more accurate reconstruction with subsampling the median downsampling method seemed to give better results with an offset (see 8.2.3) and it was used in the end.

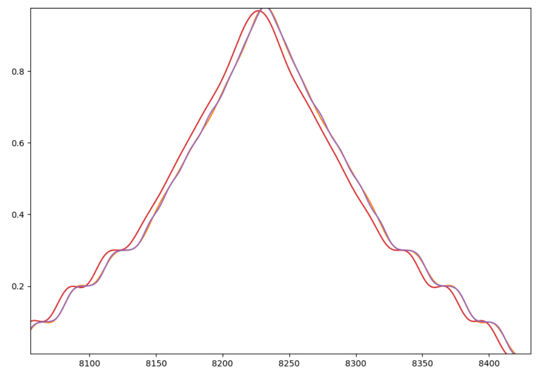
8.2.2. Threshold

We have experimented with only including predictions with confidences larger than a set threshold. It did not make an improvement on our score and we ended up not using a threshold larger than 0.

8.2.3. Offset

In our submissions, we noticed that when we give a positive offset to our predicted steps it drastically improved our score. As discussed in 4.3.1 our downsampled labels seemed to be skewed to the left when compared to subsampling the data by taking the 1st item from every 12 points. (see 4.1). By adding a positive offset to the steps of our predictions we were able to correct for the skew in our model predictions (since the model was trained on skewed labels the predictions were also skewed) and get a better score.

Figure 8.1: Comparison of different label downsampling methods when upsampled again using sinc interpolation. The red curve is the medain downsampled labels interpolated. The purple curve is the sinc interpolated subsampled labels. The orange (under the purple curve, very hard to see) is the original labels that were downsampled.



8.2.4. Find peaks

The find peaks postprocessing step was one of the largest improvements that we made causing a massive jump on the leaderboard. It increased our scores by around +0.05-0.06 just by adding this postprocessing step. We figured out that we could make multiple predictions in a day without getting our score penalized. Our best event segmentation models output 2 channels, one with confidences of onset and one with confidences with wakeups. From these channels, we can take multiple peaks which means the steps where our model thinks it is most likely that there is an event happening. Before applying this, we would take the step with the highest confidence. But by adding this find peaks method we could take multiple peaks of our channel. For this, we used the scipy $find_{peaks}$ method, and after some hyperparameter tuning we found out that taking 10 events per day with peaks of a $distance$ of 100 with a $width$ of 24 gave us the highest scores.

8.3. Ensembling

For our final submission we made an ensemble of multiple models by combining their outputs using confidence averaging. For our final ensemble we used a combination of all models mentioned in this report except for Unet (our scores were much lower with Unet compared to the rest of our models) with various features. To see exactly which models we used in an ensemble and what hyper-parameters and features each of these models used please refer to our public notebook `https://www.kaggle.com/code/schobbejak/cmi-submission-team-epoch` and refer to our Github repo where the models are explained in detail `https://github.com/TeamEpochGithub/cmi-detect-sleep-states`

9 Parting Thoughts

When the competition was over we were in 12th place in the public leaderboard. However we went down to 23rd place. Looking at the private leaderboard scores of everyone we believe that the private leaderboard has a higher percentage of clean data (data with no NaN/unlabelled steps) compared to the public leaderboard (Our private score is also much more similar to our local clean scores, ie. score on data without NaNs). This also explains why we were placed so high in the public leaderboard. Our model was better at handling not clean series compared to the people who were below us in the public leaderboard which is why we were on a higher rank, but other peoples models were better at handling clean series compared to us and that reflected on our private rank.

9.1. The winning playbook

The winning solution that have been made public at the time of writing are different combinations of GRUs, Transformers, CNNs and LGBM models and post-processing techniques. See:
https://www.kaggle.com/competitions/child-mind-institute-detect-sleep-states/discussion/459715 https://www.kaggle.com/competitions/child-mind-institute-detect-sleep-states/discussion/459627 https://www.kaggle.com/competitions/child-mind-institute-detect-sleep-states/discussion/459599 https://www.kaggle.com/competitions/child-mind-institute-detect-sleep-states/discussion/459597 https://www.kaggle.com/competitions/child-mind-institute-detect-sleep-states/discussion/459604

Our deep learning approaches are quite similar to what the people at the top of the leaderboard is using. We also used some of the post-processing steps that were used like similarity-NaN (the people that used it called it something else) and including multiple predictions per window.

9.2. In hindsight

We spent the first 4 to 6 weeks of the competition mainly focusing on creating our own pipeline and not making many submissions. This pipeline did end up being very useful and we could develop individual models quite quickly once it was done. But once we started making more models we rewrote a lot of code to make models more scalable using a model base class. With this change it was a lot easier to work with different models since only the base class needed to be changed if you wanted to change something with the models.

However once we wanted to work on ensembling models and cross validation a lot of the code had to be rewritten multiple times. Cross validation took over a week to get working and once we wanted to use ensembling it had problems with the way cross validation worked which made our code very messy at the end. This made us lose quite a bit of time on coding instead of making new models.

Even after getting cross validation to work we only took the cross validation results of individual models outside of sweeps in to account towards the end of the competition. Seeing the results at the time of writing we should have prioritized cross validation much earlier.

For the spectrogram models making submissions was especially difficult because the code used packages that were not on Kaggle. We got around it by including the packages in our source code folder and changing all imports to read files from inside the source code. This did work at the end but it took some time and made our codebase even messier by the end of it.

Probably the most important thing we should avoid in future competitions is to make non-deterministic experiments. We tried different approaches but we did not set the seed for the comparison of with and without said change which gave us totally different models. This meant that the changes we got could easily be a result of random initialization and not our addition or removal of new features (We could only notice very large improvements and smaller improvements could not be distinguished from the influence of the random initialization). We have now figured out how to get deterministic results using PyTorch (`torch.manual_seed()` is not enough) and will make proper experiments in the future.

9.3. The road not taken

One of the things that we found out only after we had made our own pipeline that there already is an scikit-learn pipeline. We had already started developing models by this point but it is definitely a consideration to use it for our future competitions.
After reading Rob Mulla's comment to this post, it is clear that we should not always trust the public leaderboard score. He concluded that the public scores dataset had very noisy labels because their CV scores were much higher than their public score (even with their worst fold score) when their validation split was using a larger dataset than the public score. With this conclusion they completely ignored the public leaderboard score and focused on their CV score. This way they ended up in 7^{th} place in the private leaderboard. With our submissions we had also noticed that our submissions scored lower than our CV scores but we were not able to come to that conclusion and kept trying to improve our public leaderboard score (We did not necessarily overfit on the public score but our private score increase was not as large as the public one).

Here is a list of roads we missed out on:

- Use data augmentation techniques such as selecting random windows from our data or reversing the time series to effectively have a dataset that is twice as big.
- Use multiple ensemble techniques such as WBF and NMS instead of only trying averaging confidences.
- Really look more at postprocessing with such a scoring metric, there was more to win there. We could have looked at reranking scores.
- We have not tried using neural networks to upsample our downsampled labels. We trained our models to predict downsampled labels and used sinc interpolation to upsample the outputs back. We could have tried using upConv layers instead.