KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

SSL for EEG-based sleep staging

Tom Smeets

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
wiskundige ingenieurstechnieken

Promotor:
Prof. dr. ir. Maarten De Vos

Assessoren:
Prof. dr. ir. F. Maes
Prof. dr. ir. M. Ishteva

Begeleiders:
Ir. E. Heremans
Ir. K. Kontras

Academiejaar 2022 – 2023

Preface

First of all, I am grateful to my promoter for giving me the opportunity to investigate SSL methods in biomedical applications. I find these methods fascinating and strongly believe they will truly unlock the potential of AI. Second, I want to convey my sincere appreciation to my assistants, whose constant enthusiasm and willingness to help were integral to the completion of this thesis. Thirdly, I would like to thank my parents and friends for their support throughout the journey. Finally I want to thank the jury for taking the time to read the text.

Tom Smeets

Contents

Abstract

This thesis aims to reduce the reliance on labeled data for training a machine learning algorithm for EEG-based sleep staging. To achieve this, we propose leveraging unlabeled data through self-supervised learning (SSL) methods. These methods aim to create proxy tasks that learn information about the data without having any task-related labels. Obtaining labeled datasets for biomedical applications, such as sleep staging, can be challenging. However, unlabeled data is often more abundant and accessible. Thus, exploring SSL methods becomes valuable in this context.

The current state-of-the-art deep learning architectures for sleep staging are sequence-to-sequence models composed of an epoch encoder, a sequence encoder, and a classifier. First, an epoch encoder extracts features from the raw EEG data. Then, a sequence encoder augments these features with temporal dependencies and encodes information like stage transition rules. Finally, a classifier classifies a sequence of encoded input EEG epochs into the respective sleep stages.

In this thesis, a two-step SSL paradigm is proposed for sequence-to-sequence sleep staging architectures. Specifically, the proposal suggests pretraining the epoch encoder and the sequence encoder with different SSL objectives suitable for feature extraction and sequence modeling, respectively. Similar approaches have been successful in natural language programming (NLP) with models like GPT [8]. However, it has not been explored for EEG-based sleep staging, and this thesis demonstrates its viability.

The epoch encoder is pretrained using SimCLR [10], a contrastive framework that has shown success in a wide variety of applications. We show that the feature space obtained after pretraining with SimCLR effectively clusters the sleep stages. Additionally, a classifier trained on top of this pretrained epoch encoder outperforms the regular supervised model by 5 percentage points when only a small fraction (approximately 1%) of the labeled dataset is used for both.

In the second phase, a pretext task is designed to further pretrain the sequence encoder on top of the pretrained epoch encoder. This allows the sequence encoder to learn context from neighboring epochs. We demonstrate that the pretrained sequence encoder also outperforms its supervised counterpart, albeit by a smaller margin of 3 percentage points in the low data regime.

Overall, our two-stage SSL pretraining approach improves the sleep staging accuracy compared to random weight initialization. It achieves a 10% improvement in the low-data regime (using approximately 1% of the labeled dataset) and a 7% improvement when using 10% of the labeled dataset.

Samenvatting

Het doel van deze thesis is de hoeveelheid gelabelde (geannoteerde) data die nodig is voor het trainen van een machine learning algoritme voor slaapclassificatie op basis van EEG signalen te verminderen. Om dit te bereiken stellen we voor om ongelabelde data te benutten via self-supervised learning (SSL) methoden. Het verkrijgen van gelabelde datasets voor biomedische toepassingen, zoals slaapclassificatie, is niet evident, terwijl ongelabelde data vaak in overvoed verkrijgbaar zijn. Daarom is het verkennen van SSL-methoden in deze context waardevol.

De huidige state-of-the-art deep learning architecturen voor slaapclassificatie zijn sequence-to-sequence modellen. Zij bestaan uit een epoch-encoder, een sequence-encoder en een classifier. De epoch-encoder extrahereert features uit de ruwe EEG signalen, de sequence-encoder encodeert temporele informatie zoals regels voor overgangen tussen slaapfasen, en de classifier classificeert een reeks invoer EEG-epochs in de bijhorende slaapfasen op basis van de gecodeerde informatie.

In deze thesis wordt een SSL-paradigma voorgesteld voor sequence-to-sequence slaapclassificatie. Specifiek stellen we voor om de epoch-encoder en de sequence-encoder met verschillende SSL-objectieven 'voor te trainen'. Die SSL objectiven zijn geschikt voor respectievelijk feature extractie en sequentiële modellering. Dit paradigma was reeds succesvol in natural language programming (NLP) met modellen zoals GPT [8]. Echter, het is nog niet onderzocht voor slaapclassificatie op basis van EEG. Deze thesis toont aan dat dit paradigma ook hiervoor toepasbaar is.

De epoch-encoder wordt voor getrained met behulp van SimCLR [10], een contrasterend framework dat recent succesvol is gebleken in verschillende andere domeinen. We tonen aan dat de feature space, die wordt verkregen na voortraining met SimCLR, effectief de slaapfasen groepeert. Bovendien presteert een classifier op deze feature space beter dan het reguliere supervised model: 5 procent wanneer slechts een klein deel (ongeveer 1%) van de gelabelde dataset wordt gebruikt.

In de tweede fase wordt een pretext task ontworpen om de sequence-encoder verder voor te trainen bovenop de epoch-encoder. We tonen aan dat de voorgetrainde sequence-encoder ook beter presteert dan zijn supervised tegenhanger, zij het met een kleinere marge van 3 procent.

Globaal genomen verbetert onze twee-fasen SSL voortraining de nauwkeurigheid van de slaapclassificatie in vergelijking met willekeurige gewichtsinitialisatie. Een verbetering van 10% is bereikt wanneer slechts 1% van de gelabelde dataset ter beschikking is, en een verbetering van 5% wanneer 10% van de gelabelde dataset ter beschikking is.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

ANN	Artificial Neural Network
BPTT	BackPropagation Through Time
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
CV	Computer Vision
EEG	Electroencephalogram
GRU	Gated Recurrent Unit
KL	Kullback-Leibler divergence
LSTM	Long Short Term Memory
MLP	Multilayer Perceptron
NLP	Natural Language Processing
RNN	Recurrent Neural Network
SSL	Self-Supervised Learning

Chapter 1

Introduction

This thesis investigates self-supervised learning (SSL) methods for EEG-based sleep staging. SSL methods can be used to reduce the amount of *labeled* data necessary to train a machine learning (ML) algorithm, by exploiting massive amounts of *unlabeled* data.

Obtaining labeled datasets in biomedical applications is difficult due to several reasons. First of all, there are strict privacy and confidentiality regulations surrounding patient data, limiting access to large-scale datasets. Second, biomedical data is often limited in size, especially for rare diseases or specific conditions. Third, annotating biomedical data requires domain expertise, involving medical professionals who may be time-consuming and costly to engage. Furthermore, different experts may interpret data differently, leading to inter-annotator variability. Finally, ethical considerations and informed consent requirements for data usage can restrict the availability of labeled data. [20] [22]

However, there exist large datasets for sleep staging, making it an ideal task for evaluating SSL methods in biomedical applications. By comparing SSL methods to fully supervised approaches that utilize the entire dataset, meaningful comparisons and performance assessments can be conducted.

In addition to reducing the amount of labeled data necessary to train an ML algorithm, there is also evidence that SSL models can learn representations that are more robust to adversarial examples, label corruption, and input perturbations. [16]

1.1 What are EEG signals?

Electroencephalography (EEG) is a non-invasive technique in which the electrical activity of the brain is recorded at the level of the scalp. To this end, a set of electrodes are placed on the head using the International 10-20 System of electrode placement (see Figure 1.1) and the potential differences between the electrodes are measured. The result is then a multi-channel EEG recording, one of which can be seen in Figure 1.2. In general EEG signals are noisy, hard to interpret, and challenging to use in automated scenarios. Furthermore, the measurement process introduces various types of artifacts affecting the signal.

The electrical activity in the EEG is mostly the result of synchronized synaptic currents coming from thousands of neurons. When a neuron fires, it releases neurotransmitters that bind to receptors on the postsynaptic neuron, generating an electrical current. EEG signals are highly non-Gaussian, non-stationary, and have a nonlinear nature.[31]

One of the primary use cases of EEG is as a bio-marker for brain function and malfunction. For example, epilepsy is characterized by a sudden increase in electrical activity that can be clearly detected in the EEG (Figure 1.2). The EEG can also be used to diagnose and characterize various sleep disorders, including sleep apnea, narcolespy and even depression.[29] Beyond simple diagnostics, brain-computer interfaces (BCIs) can use EEG to allow individuals to communicate using brain activity. A good example of this is given by KUL students at NeuroTech (https://www.ntxl.org) who used EEG signals to allow a subject to remotely browse the web using their brain.

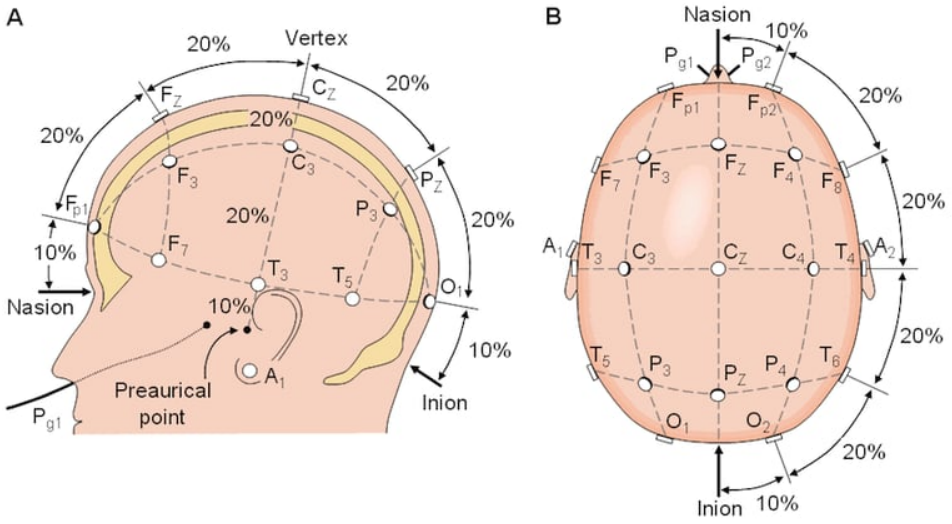


FIGURE 1.1: The International 10-20 System of electrode placement, adopted from [33]. Each point in the Figure consists of a letter and a number to indicate a possible electrode position. Each site has a letter (to identify the lobe) and a number or another letter to identify the hemisphere location. The letters F, T, C, P, and O stand for Frontal, Temporal, Central, Parietal and Occipital. Smaller numbers correspond to areas closer to midline and points on the midline have the letter 'z' instead of a number. Even numbers (2,4,6,8) refer to the right hemisphere and odd numbers (1,3,5,7) to the left hemisphere.

1.2 What is sleep staging?

EEG activity changes during sleep. The American academy of sleep medicine scoring standard [7] identifies 5 different qualitative EEG behavior during sleep and categorizes them into 5 sleep stages: Wake, Rem, N1, N2, and N3. N1 and N2 are

FIGURE 1.2: Example of the onset of an epileptic seizure in a multi-channel EEG, adopted from [42]. The different signals correspond to different EEG channels, i.e. measurements coming from different electrodes. The onset of a seizure can be clearly detected in this figure by the sudden increase in both the amplitude and frequency of the EEG signals.

considered light sleep and N3 deep sleep. Rapid eye movement (REM) sleep is where dreaming happens. An example of EEG signals corresponding to the different sleep stages can be seen in Figure 1.3.

Stage 1, also known as N1, serves as a transitional phase between wakefulness and sleep, marking the initial stages of falling asleep. During this period, there is a noticeable deceleration in both respiration and heartbeat rates. Additionally, stage 1 sleep is characterized by a significant reduction in muscle tension and core body temperature. In the early part of stage 1, the brain exhibits α waves, which are low-frequency (8-13 Hz) waves with relatively high amplitude. However, as one progresses deeper into stage 1, there is an increase in theta wave activity, characterized by even lower frequencies (4-7 Hz). [2], [29]

During the transition to stage 2, the body enters a state of profound relaxation. Theta waves continue to dominate the brain's activity; however, they are periodically interrupted by short bursts of sleep spindles, as illustrated in Figure 1.4. Sleep spindles consist of higher-frequency brain waves. These spindles are often accompanied by K-complexes, which are long delta waves that last for approximately one second and

are known to be the longest and most distinct of all brain waves. An example of a K-complex is also given in Figure 1.4. [29]

Stage 3, also known as deep sleep or slow-wave sleep, is a crucial phase to achieve restorative rest. This stage is distinguished by the presence of very low-frequency δ waves, ranging from 0 to 4 Hz, indicating a significant decrease in both heart and respiration rates. The individual's physiology experiences a profound slowdown during this stage, making it considerably more challenging to awaken them compared to earlier stages of sleep. Deep sleep plays a vital role in physical recovery, immune system functioning, and overall well-being. [29]

Moving on to REM (Rapid Eye Movement) sleep, it is a distinct stage characterized by the rapid movement of the eyes. During this stage, the brain waves bear remarkable similarity to those exhibited when an individual is awake. It is during REM sleep that dreaming primarily occurs. The brain activity becomes more active and resembles wakefulness, while the body undergoes temporary muscle paralysis, likely to prevent the physical enactment of dreams. REM sleep is thought to play a significant role in cognitive processes, memory consolidation, emotional regulation, and learning. It is an essential part of the sleep cycle and contributes to overall sleep quality and mental rejuvenation. [2], [29]

Manually annotating is a hard and time-consuming task that requires a trained medical professional to annotate hours of sleep into the different stages. Furthermore, the interrater agreement for overall manual sleep staging is only 85%, as studied by the authors of a big meta-analysis on interrater reliability for sleep staging: [22].

The result of sleep staging is a *hypnogram* from which sleep quality can be derived, see Figure 1.5 for an example.

1.3 Sleep staging with ML

A popular definition of ML, due to Tom Mitchell [23], is as follows:

> A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning algorithms rely on training data as input to learn patterns and relationships within the data. The desired outcome is to obtain a function that can *generalize* well to unseen test data, meaning it can make accurate predictions or classifications beyond the training set. By leveraging the power of machine learning, particularly deep learning methods, the process of sleep staging can be automated, allowing medical professionals to allocate their attention to other critical tasks.

1.3.1 The engineering approach: manual feature design

Classical ML algorithms, including logistic regression, naive Bayes, linear discriminant analysis (LDA), k-nearest neighbors (k-NN), random forests, etc., typically require a set of "features" as input. These features act as lower dimensional representations

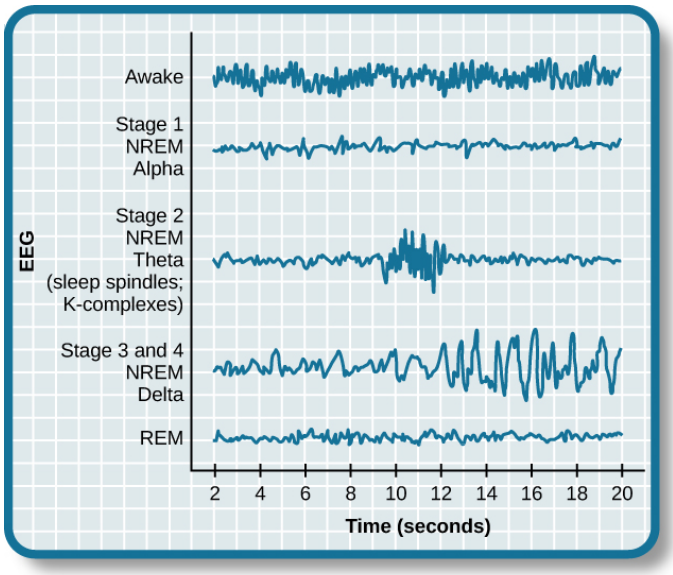


FIGURE 1.3: Typical EEG-signals and their corresponding sleep stages, adopted from [2]. Wake segments correspond to high amplitude, high-frequency waves. Stage 1 sleep, the lightest form of sleep, is characterized by lower frequency waves of still relatively large amplitude. Stage 2 sleep is characterized by low-frequency waves and brief bursts of sleep spindles, which are often followed by a K-complex (Figure 1.4). Deep sleep (N3) is characterized by very low-frequency waves, and REM sleep resembles Wake segments in terms of increased brain activity during dreaming.

of the EEG signal, making it easier for these algorithms to learn mappings that generalize well to unseen test data. To achieve this, manual feature engineering is often performed on the preprocessed EEG signal, and these engineered features are then fed into classical ML classification algorithms.

A comprehensive study on manual features can be found in [45], where a random forest was trained on 59 complex features. These features encompass time-domain, frequency-domain, and nonlinear characteristics. Time-domain features include statistical measures such as mean, variance, skewness, kurtosis, minimum and maximum values, zero-crossing rate, as well as absolute and mean values of first-order and second-order differences. Frequency-domain features involve the mean band powers in significant frequency bands, including delta, theta, alpha, beta, and gamma. Nonlinear features encompass measures like fractal dimension (FD), the non-stationary index that quantifies the temporal variation of the local mean, and sample entropy, among others.

However, in recent years, there has been a shift towards leveraging deep learning techniques to *learn* the most useful features from data. This approach, known as *representation learning*, eliminates the need for manual feature engineering and allows the learning of optimal features. The current state-of-the-art methods employ deep learning approaches, which will be discussed in the upcoming section.

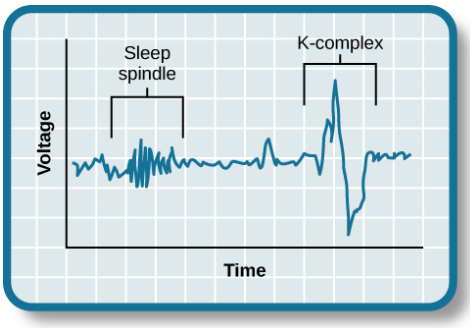


FIGURE 1.4: An example of a sleep spindle and K-complex, adopted from [2]. A sleep spindle is a burst of high-frequency waves, typical for N2 sleep and they are often followed by K-complexes. K-complexes are large, slow waveforms that appear as sudden, sharp spikes followed by a slower negative component.

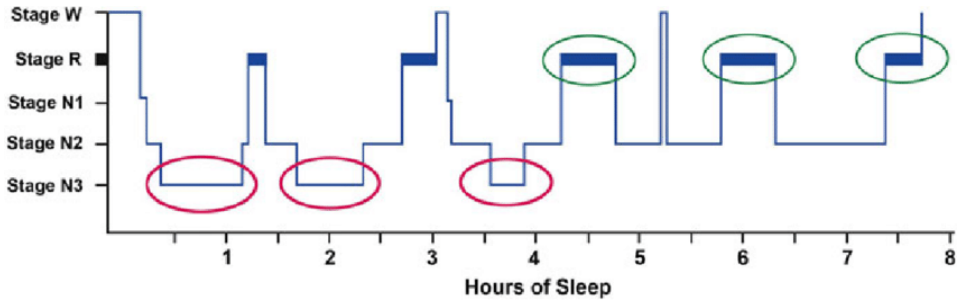


FIGURE 1.5: An example of a hypnogram, adopted from [28]. It shows the sleep stage of a subject as a function of time and can be used by medical professionals for diagnosis.

1.3.2 Representation learning with deep learning

Deep learning, as described by Goodfellow et al. (2016) [13], refers to the utilization of artificial neural networks (ANNs) with multiple layers of 'hidden' neurons. These networks, commonly known as deep neural networks, have witnessed significant advancements in recent years, primarily due to the availability of increased computational power and several key innovations. Historically, neural networks were relatively shallow, consisting of only a few layers, and were primarily employed for basic classification and regression tasks. Support vector machines (SVMs) were the preferred choice within the machine learning community, owing to their comprehensive understanding and solid mathematical grounding in Statistical Learning Theory (Vapnik, 1998) [39].

The surge in computational resources, coupled with notable advancements in activation functions, regularization techniques (such as dropout), residual connections, and architectural design (to be discussed in Section 2.2 in the subsequent chapter), has facilitated the development of deeper neural networks. The term "deep" in deep learning specifically refers to the presence of multiple layers between the input and

output layers. Each layer performs computations on the received data and transmits the results to the subsequent layer. The earlier layers primarily learn low-level features, such as edge detectors, while the deeper layers progressively learn more complex and abstract representations. For instance, in digit recognition tasks, earlier layers may identify fine-grained features like edges, whereas later layers may learn intricate nonlinear features like circles or parts of digits.

An alternative perspective on deep learning is viewing it as a mechanism for learning multistep computer programs. Each layer in the network's representation can be likened to the state of a computer's memory after executing a set of instructions in parallel. Networks with greater depth can execute a more extensive sequence of instructions, thereby increasing their computational power (Goodfellow et al., 2016) [13].

Despite the notable advancements enabled by deep learning, one critical drawback emerges: the reliance on large labeled datasets to prevent overfitting[1]. A larger network — a network with more parameters — necessitates a larger dataset for training. However, this requirement poses a significant challenge in scenarios where limited labeled data is available, as is often the case in biomedical applications. This has motivated the exploration of self-supervised learning (SSL) techniques, aiming to leverage the abundant unlabeled data to reduce the dependence on labeled data for training neural networks.

1.4 Self-supervised learning: the dark matter of AI

How is it possible that a child can recognize a cow after seeing only a few, whereas a deep learning model may require thousands of examples and still struggle to identify a cow in an unconventional setting, such as lying on the beach? Yann Lecun, a prominent figure in modern machine learning, discusses this issue in a renowned blog post[3]. He argues that machines lack what is known as "common sense": fundamental knowledge about the world, including concepts like gravity and abstract reasoning, which children naturally acquire through observation from an early age. This absence of common sense is why humans can learn to drive in just twenty hours, while machines, despite being trained on thousands of hours of driving footage, still struggle to do so.

To address this limitation, self-supervised learning has emerged as a method to impose prior knowledge on the neural networks by exposing them to vast amounts of unlabeled data. The neural networks are initially "pretrained" with an SSL objective using unlabeled data. When this objective is closely aligned with the downstream task[2], the pretraining process yields well-initialized parameters for the neural networks. The well-initialized neural network can then be transferred to the downstream task, which won't need as much data to generalize well as a consequence

[1] Overfitting in ML happens when a model performs well on training data, but fails to generalize well to unseen test data.

[2] The downstream task is the task we really want to solve. In our case, this task is sleep staging.

of the pretraining. This approach is known as "transfer" learning, and is illustrated in Figure 1.6.

In the field of natural language processing (NLP), SSL commonly involves predicting a word in a sentence based on the surrounding words, as exemplified in BERT [12]. This objective encourages the model to capture word relationships without the need for explicit labels. The pretrained network can then be fine-tuned for various downstream tasks, such as machine translation, summarization, or text generation.

In computer vision, similar SSL objectives exist, such as predicting masked patches of an image using a masked autoencoder (MAE). More contemporary SSL methods employ contrastive learning, which involves comparing and encouraging similar representations of augmented views of the same image, achieved through techniques like rotation or cropping. These contrastive methods will be thoroughly explored in Chapter 2.

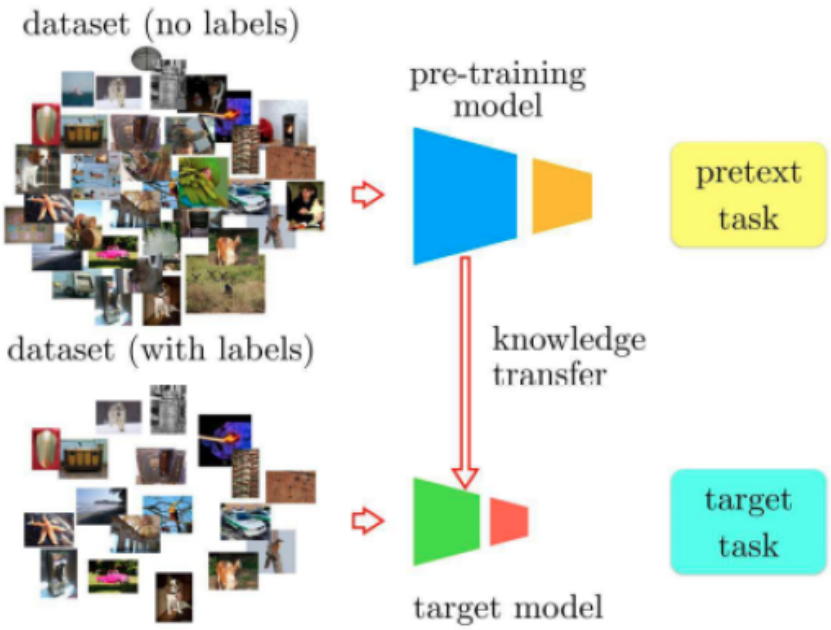


FIGURE 1.6: General SSL pipeline, adopted from [27]. The general SSL setting consists of pretraining a neural network with a large dataset of unlabeled samples on a pretext task, which is a task derived from the unlabeled data itself. After pretraining on a pretext task, the model is fine-tuned on a downstream task with a small labeled dataset.

1.5 Ultimate goal

Our evaluation will focus on assessing SSL methods in the context of sleep staging as a downstream task. The process involves pretraining a neural network using an SSL

objective on a large unlabeled dataset. Subsequently, we will transfer this pretrained neural network to perform sleep staging using a smaller labeled dataset. To establish a baseline, we will compare the performance of the pretrained model with that of the same neural network but with randomly initialized parameters, i.e., without any pretraining.

To comprehensively evaluate the SSL pretraining, we will repeat this comparison using a series of labeled datasets, each increasing in size. By progressively introducing more labeled data, we can observe the impact of SSL pretraining on model performance as the dataset's supervisory signal strengthens.

Our criterion for determining the success of SSL pretraining will be if the sleep staging accuracy of the pretrained model surpasses that of the regular fully supervised model. If the pretrained model consistently achieves higher accuracy across multiple dataset sizes, we will conclude that the SSL pretraining process has been successful in improving sleep staging performance.

By conducting this evaluation, our goal is to showcase the effectiveness of SSL in leveraging unlabeled data to enhance sleep staging performance.

1.6 Modus operandi

The upcoming chapter will present a comprehensive literature study that highlights the key deep-learning architectures employed in sleep staging. Firstly, a summary of these architectures will be provided to establish a foundational understanding. Subsequently, the focus will shift towards discussing state-of-the-art self-supervised learning (SSL) methods, with particular emphasis on contrastive learning which has shown promising results in various domains.

In Chapter 3, a specific deep learning architecture will be selected for sleep staging. An SSL method to pretrain the network with will be proposed, and some experiments to test the method will be outlined. The subsequent chapter, Chapter 4, will present and analyze the outcomes of these experiments, delving into the performance and effectiveness of the chosen architecture and SSL technique.

Finally, the thesis will culminate in Chapter 5, where a comprehensive conclusion will be drawn. This section will provide a summary of the most significant results and insights. Additionally, it will outline potential avenues for future research.

1.7 Conclusion

In conclusion, this chapter provided an introduction to the topic of Self-Supervised Learning (SSL) methods for EEG-based sleep staging. It highlighted the challenges of obtaining labeled datasets in biomedical applications and the potential of SSL to leverage unlabeled data to reduce the reliance on labeled data. The chapter also introduced the concept of EEG signals, their significance in studying brain function and sleep disorders, and the process of sleep staging using EEG data. It discussed the manual feature engineering approach and the transition towards end-to-end learning approaches with deep learning methods.

Overall, this chapter sets the stage for further exploration of SSL methods for EEG-based sleep staging, with the aim of improving the automation and accuracy of this important task. The upcoming chapter will delve into the state-of-the-art literature on SSL and examine the key deep learning architectures employed in sleep staging.

Chapter 2

Literature study

2.1 Introduction

In the field of sleep staging, cutting-edge deep-learning techniques rely heavily on specialized neural network architectures that offer significant advantages over standard multilayer perceptron (MLP) models. These architectures are exceptionally parameter efficient, enabling them to achieve greater depth and enhanced representation capacity. The most important of these architectures and the literature in which they are applied to EEG-based sleep staging will first be discussed in Section (2.2).

Following that, Section 2.3 will provide a comprehensive overview of the existing literature on Self-Supervised Learning (SSL) methods, with a focus on contrastive methods.

2.2 Sleep-staging using deep learning

The multilayer perceptron (MLP) serves as a fundamental example of an artificial neural network (ANN) and is composed of multiple interconnected layers of neurons, arranged in a feedforward manner (see Figure 2.1). Each neuron in this architecture takes an input vector $\boldsymbol{x} \in \mathbb{R}^n$, performs a learnable affine transformation, and applies a continuous elementwise nonlinearity denoted as σ. The output of a neuron can be represented as $\boldsymbol{h} = \sigma(W\boldsymbol{x} + \boldsymbol{b}) \in \mathbb{R}^m$, where $W \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$ represent the weights and biases, respectively, and m corresponds to the size of the next layer.

In essence, an MLP can be understood as a continuous mathematical function denoted by f, which is parameterized by the weights and biases $\boldsymbol{w} = \{W, \boldsymbol{b}\}$ of its constituent neurons. This function maps an input vector $\boldsymbol{x} \in \mathbb{R}^n$ to an output vector $\hat{\boldsymbol{y}} \in \mathbb{R}^m$, such that $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{w})$.

When provided with a dataset $\mathcal{D} = (\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^N$, a continuous loss function can be defined to measure the discrepancy between the model's output and the actual labels. This loss function, denoted as $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{w})$, is also dependent on the weights since it involves the predicted output $\hat{\boldsymbol{y}}$.

The backpropagation algorithm is a clever technique used to efficiently calculate the gradient $\nabla_{\boldsymbol{w}} \mathcal{L}$. By leveraging this algorithm, various continuous optimization

techniques can be employed to minimize $\mathcal{L}$ with respect to the network's parameters, thereby enabling the model to fit f to $\mathcal{D}$. It has been proven that even a neural network with just a single hidden layer serves as a *universal function approximator*, meaning it can approximate any continuous function to arbitrary precision by employing a sufficient number of neurons [17].

However, MLPs suffer from a scalability issue as the number of parameters grows quadratically with the size of the hidden layer. For example, in a fully connected feedforward network with n layers, each containing h hidden neurons, each weight matrix W consists of h^2 weights, and each bias $\boldsymbol{b}$ comprises h weights. Consequently, for n layers, the total number of parameters scales as $\mathcal{O}(n * h^2)$. This is a problem, particularly when dealing with high-dimensional inputs such as images that can contain millions of pixels. To enable deep networks, more parameter-efficient neural network architectures have been developed, including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and the more recent Transformer architecture. These architectures are widely employed in state-of-the-art sleep staging and will be introduced in the following sections.

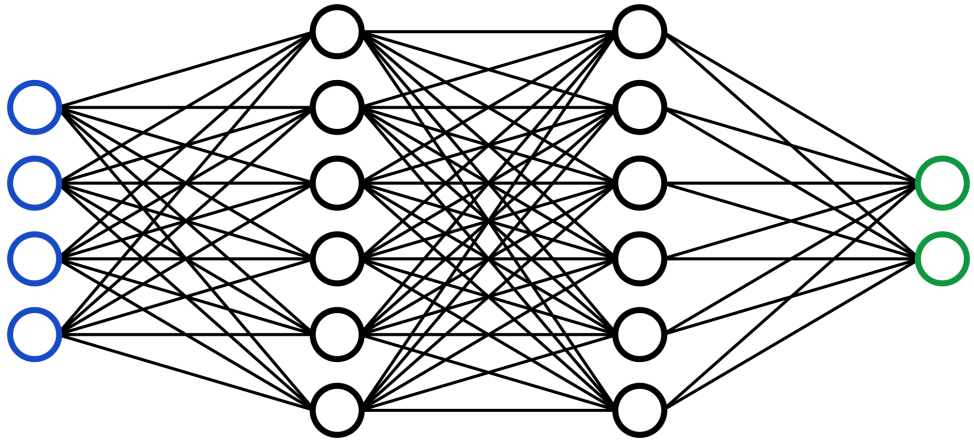


FIGURE 2.1: An example of a Multi-layer perceptron (MLP) with two hidden layers and an input layer of dimension 4 and output layer of dimension 2, adopted from [1]. These are also called fully-connected layers because every node in a layer is connected to every node of the previous layer. Such networks have been proven to be *universal function approximators* [17].

2.2.1 Convolutional neural network

MLPs cannot be made very deep for very high-dimensional data such as images because their amount of parameters becomes huge. A huge amount of parameters introduces two problems: first, it is computationally difficult and the second more profound problem is that a bigger network needs more data to avoid overfitting

[13]. A fundamental insight of high-dimensional data such as images and large time series is that they should be processed in a hierarchical way from a *local* to a *global* level of abstraction. For instance, in digit recognition, initial layers can specialize in detecting local features like edges, while subsequent layers can learn to assemble these features into more complex abstractions like circles. Finally, the last layers can identify combinations of circles, such as two circles forming the shape of the digit 8. This hierarchical processing is precisely what convolutional neural networks (CNNs) aim to achieve.

What is a CNN?

A CNN layer extracts local information from a signal by *convolving* it with a set of learned filters (also called the *kernels*), which act as feature detectors. A discrete convolution (symbolized by $*$) between an input vector $\boldsymbol{x} \in \mathbb{R}^N$ and a filter $\boldsymbol{h} \in \mathbb{R}^k$ produces an output (feature map) as follows:

$$\boldsymbol{y}_i = (\boldsymbol{x} * \boldsymbol{h})(i) = \sum_{j=0}^{N-1} x[i+j]h[k-j] \tag{2.1}$$

However, a convolution is often implemented as a cross-correlation which is the same as a convolution, but with horizontally flipped kernel weights:

$$\boldsymbol{y}_i = cross\_corr(\boldsymbol{x}, \boldsymbol{h})(i) = \sum_{j=0}^{N-1} x[i+j]h[j] \tag{2.2}$$

Since the kernel weights are learned from data, both lead to the same result but cross-correlation is slightly easier to implement. An example of this simple operation for a 1D input can be seen in Figure 2.2. An important property of convolution is

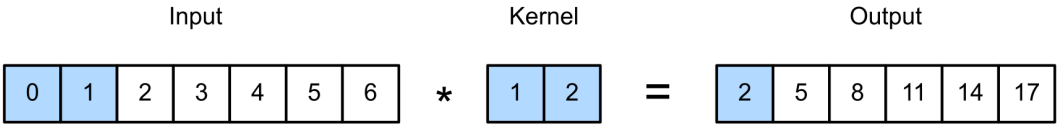


FIGURE 2.2: Example of 1d cross-correlation, adopted from [25]. Cross-correlation is very closely related to convolution: it is a convolution with horizontally flipped filter weights. The kernel 'passes' over the input and outputs a weighted sum of the input at each position.

that it is *translation invariant*. This is exactly what we want: an edge detector is not dependent on its location in the image.

The input to a convolutional layer comprises C_{in} input channels, representing various components such as color channels in images. Several parameters define the convolutional layer's behavior, including the kernel size (k), stride, padding, and the number of output channels (C_{out}). Each output channel is generated by convolving each input channel with a learned kernel, and the resulting values are summed.

Padding involves adding zeros at the input's boundary, while stride determines the displacement of the convolutional filter as it moves across the input. A stride of one indicates that the filter moves one sample at a time, whereas a stride of two causes it to move two samples at a time, resulting in downsampling of the original signal.

Each layer introduces $C_{in} \times k^d \times C_{out}$ weights, where d represents the dimension of the kernel (e.g., 1 for 1D convolution, 2 for 2D convolution), and k denotes the kernel size. Remarkably, these are relatively few parameters in comparison to a regular fully connected layer, particularly when dealing with high-dimensional inputs. Consequently, one can stack a significant number of convolutional layers on top of each other, enabling the creation of very deep networks.

However, the depth of neural networks can lead to the vanishing gradient problem. This issue arises when the gradients used to update the weights in deep neural networks diminish significantly as they propagate backward through the layers.

To address the vanishing gradient problem, various techniques have been developed and proven effective. For instance, the use of activation functions with larger gradients, such as Rectified Linear Units (ReLU), helps alleviate the issue. Additionally, normalization techniques like batch normalization can be employed to ensure stable gradients during training [18]. Another approach involves incorporating skip connections, such as residual connections [15], which enable gradients to bypass multiple layers. These techniques play a crucial role in mitigating the vanishing gradient problem and facilitating the successful training of deep neural networks.

Application to EEG

Convolutional neural networks have been extensively used for the task of sleep staging. In [34] for example, a 12-layer convolutional network was applied to a sequence of 4 raw single-channel EEG epochs[1]: the epoch to classify, two previous and the next epoch such that temporal information is incorporated as well (if an epoch is surrounded by e.g. 'wake' epochs, it is more likely to be 'wake' as well). Their architecture can be seen in Figure 2.3 and they claim to reach state-of-the-art performance on the SHHS dataset, which is the dataset used in this work as well: Section 3.2.

2.2.2 Recurrent Neural Network

A recurrent neural network is another type of artificial neural network, which maps from an input space of sequences $\{\boldsymbol{x}_t\}_{t=1}^T$ to an output space of sequences $\{\boldsymbol{y}_t\}_{t=1}^T$ in a stateful way [25]. That is, the prediction of output $\boldsymbol{y}_t \in \mathbb{R}^M$ depends not only on the input $\boldsymbol{x}_t \in \mathbb{R}^N$, but also on the hidden state of the system $\boldsymbol{h}_t \in \mathbb{R}^H$, which gets updated over time, as the sequence is processed. The fact that an RNN has a state means that it has *unbounded memory*, i.e. it has the potential to remember information from $t = 0$ until time t, which makes it a powerful autoregressive model. This is in contrast with standard Markov models, which are models that assume that the future is independent of the past.

[1] See 3.2 for what is meant by a sleep epoch

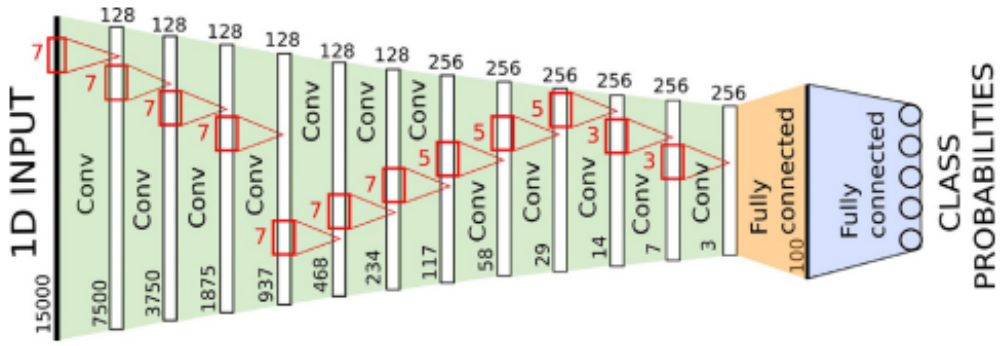


FIGURE 2.3: Architecture of a 1D convolutional neural network for EEG, adopted from [34]. The input consists of a sequence of 4 sequential epochs: the epoch to classify, one before and two after. This high dimensional input is downsampled through a set of 12 convolutional layers before being input to a logistic classifier (a linear layer with a softmax appended at the end).

Such RNN models can be used for sequence generation, sequence classification, and sequence translation.

Classical RNN architecture

The classical Recurrent Neural Network (RNN) employs a Multilayer Perceptron (MLP) denoted as f to update the state. In the case of a one-layer MLP, the state update can be represented as $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) = \sigma(U\boldsymbol{h}_{t-1} + V\boldsymbol{x}_t + \boldsymbol{b}) \in \mathbb{R}^H$, where σ is a nonlinear function like the sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$, applied elementwise. The output is often modeled as a simple affine transformation of the hidden state: $\boldsymbol{y}_t = W\boldsymbol{h}_t + \boldsymbol{d}$.

In this formulation, the RNN's hidden state $\boldsymbol{h}_t$ at time step t is updated based on the previous hidden state $\boldsymbol{h}_{t-1}$ and the input $\boldsymbol{x}_t$ at the current time step. The weights $U \in \mathbb{R}^{H\times H}$, $V \in \mathbb{R}^{H\times N}$, and $W \in \mathbb{R}^{M\times H}$ along with biases $\boldsymbol{b} \in \mathbb{R}^H$ and $\boldsymbol{d} \in \mathbb{R}^M$ are learnable parameters. The nonlinear activation function σ introduces nonlinearity into the state update, enabling the RNN to capture complex temporal dependencies in the data. The output $\boldsymbol{y}_t$ is computed by applying a linear transformation to the hidden state $\boldsymbol{h}_t$.

The fact that an RNN has unbouded memory through a state, means that the current output $\boldsymbol{y}_t$ is indirectly a function of all the previous hidden states $\boldsymbol{h}_{<t}$. The gradient of the loss function w.r.t. the parameters must then go back through time as well. Backpropagation becomes BackPropagation Through Time (BPTT), which takes $\mathcal{O}(T)$ to compute per timestep [25]. This is obviously infeasible and one therefore chooses an appropriate truncation K.

One significant challenge faced by classical Recurrent Neural Networks (RNNs) is the issue of vanishing and exploding gradients. These problems arise when the gradients used to update the weights in deep neural networks become either extremely small or excessively large as they propagate backward through the layers. In the

case of classical RNNs, this occurs due to the repeated multiplication by the same weight matrix as the input is processed over time, and the loss is backpropagated.

The vanishing gradient problem emerges when the gradients gradually diminish in magnitude as they propagate through the layers, eventually becoming extremely small. As a result, the earlier layers of the network receive negligible gradient updates, hindering their ability to learn and capture long-term dependencies in the data.

Conversely, the exploding gradient problem occurs when the gradients grow exponentially as they backpropagate, leading to unstable and divergent weight updates. This can cause the network to become unstable during training and adversely affect its ability to converge to an optimal solution.

Both of these problems pose significant challenges in training classical RNNs, limiting their effectiveness in capturing long-term dependencies in sequential data. Various techniques have been developed to mitigate these issues, such as using alternative RNN architectures like Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU). They employ specialized gating mechanisms to update the hidden state in an additive instead of multiplicative way, therefore alleviating the vanishing gradient problem and enabling better gradient flow during training. They will be discussed below.

A last fundamental limitation of both classic RNNs, LSTMs and GRUs is that they cannot be parallelized due to their sequential nature and dependencies between time steps. This means that training them can be a lot slower compared to other neural network architectures such as feedforward layers or convolutional networks.

GRU

The key idea of the Gated Recurrent Unit (GRU) is to learn when to update the hidden state and when to reset it, by using an update gate $\boldsymbol{z}_t$ and a reset gate $\boldsymbol{r}_t$. Figure 2.4 provides an example of a GRU, and the equations governing its operation are as follows:

$$
\begin{aligned}
\boldsymbol{z}_t &= \sigma(W_z \boldsymbol{x}_t + U_z \boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \\
\boldsymbol{r}_t &= \sigma(W_r \boldsymbol{x}_t + U_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \\
\hat{\boldsymbol{h}}_t &= \phi(W_h \boldsymbol{x}_t + U_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \\
\boldsymbol{h}_t &= (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \hat{\boldsymbol{h}}_t
\end{aligned}
$$

In these equations, σ represents the sigmoid activation function $\sigma(x) = \frac{1}{1+\exp(-x)} \in [0, 1]$, and ϕ denotes the hyperbolic tangent (tanh) function. The weights (W_z, U_z, W_r, U_r, W_h, U_h) and biases ($\boldsymbol{b}_z$, $\boldsymbol{b}_r$, $\boldsymbol{b}_h$) are learned from the data. Both $\boldsymbol{z}_t$ and $\boldsymbol{r}_t$ are constrained to values between zero and one, serving as indicators of the proportion by which the hidden state should be updated or reset, respectively. The Hadamard product ($\odot$) denotes element-wise multiplication. [25]

The GRU architecture incorporates these gates and their corresponding equations to provide enhanced control over the flow of information in recurrent neural networks. By selectively updating and resetting the hidden state, GRUs can capture and retain

relevant information over long sequences, effectively addressing the vanishing gradient problem and enabling improved modeling of sequential data.

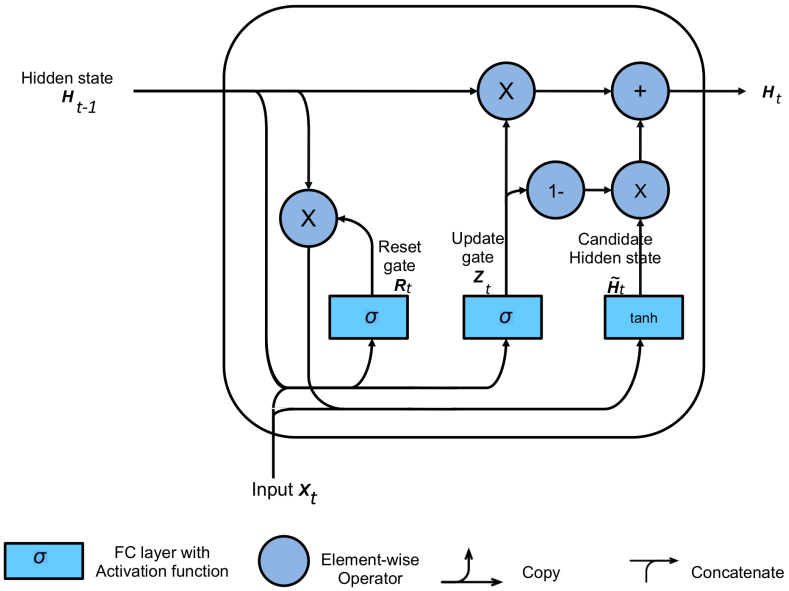


FIGURE 2.4: An example of a Gated Recurrent Unit (GRU), adopted from [25]. A reset and update gate are learned from data and allow the unit to update or reset parts of the hidden state in an additive way.

LSTMs

A Long Short Term Memory (LSTM) is a more sophisticated version of the GRU that also updates the hidden state in an additive way but, in addition, also implements a long-term memory cell $\boldsymbol{c}_t$. In order to update the long-term memory, three gates are implemented: an output gate $\boldsymbol{o}_t$ which controls what part of the long-term memory gets read out to predict the next hidden state; an input gate $\boldsymbol{i}_t$ which controls what gets read in from the hidden state and the current input; and a forget gate $\boldsymbol{f}_t$ which controls what is forgotten based on the current state and the hidden state. An example of an LSTM is given in Figure 2.5 and the equations are very similar to those of the GRU: [25]

$$
\begin{aligned}
\boldsymbol{o}_t &= \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{i}_t &= \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \\
\bar{\boldsymbol{c}}_t &= tanh(W_{xc}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \bar{\boldsymbol{c}}_t \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned}
$$

The weights and biases are again learned with data and σ is again the sigmoid and $\odot$ the element-wise multiplication (Hadamard product).

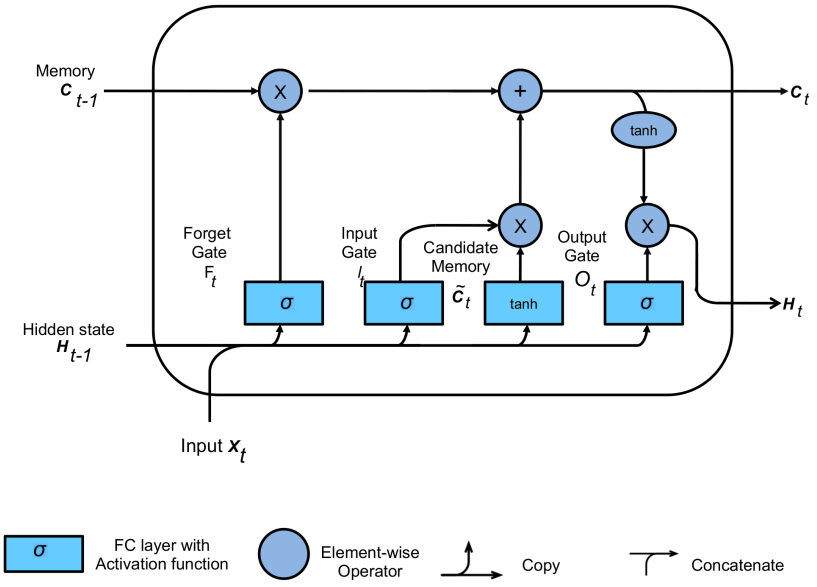


FIGURE 2.5: A typical LSTM cell, adopted from [25]. The hidden state $\boldsymbol{H}$ is augmented with a 'long-term' memory cell $\boldsymbol{c}$. This memory can be selectively accessed, updated, and reset, through an output gate $\boldsymbol{O}$, input gate $\boldsymbol{I}$, and a forget gate $\boldsymbol{F}$. The weights and biases that control these gates are learned from data.

Bi-directional RNNs

In non-causal models, it is often beneficial to let the current hidden state $\boldsymbol{h}_t$ of an RNN depend both on the past **and** the future. This leads to a bidirectional RNN, which is updated as follows [25]:

$$\begin{aligned}\boldsymbol{h}_t^{\rightarrow} &= \varphi(W_{xh}^{\rightarrow}\boldsymbol{x}_t + W_{hh}^{\rightarrow}\boldsymbol{h}_{t-1}^{\rightarrow} + \boldsymbol{b}_h^{\rightarrow}) \\ \boldsymbol{h}_t^{\leftarrow} &= \varphi(W_{xh}^{\leftarrow}\boldsymbol{x}_t + W_{hh}^{\leftarrow}\boldsymbol{h}_{t-1}^{\leftarrow} + \boldsymbol{b}_h^{\leftarrow})\end{aligned}$$

The hidden state that is used to produce the output is then simply defined to be the concatenation of both: $\boldsymbol{h}_t = [\boldsymbol{h}_t^{\rightarrow}, \boldsymbol{h}_t^{\leftarrow}]$.

This idea can be extended to come up with bidirectional versions of GRUs and LSTMs as well.

Application to EEG

Sleep staging can be viewed as a sequence-to-sequence task, where the input consists of a sequence of EEG epochs, and the goal is to predict the corresponding sequence of sleep stages. To achieve this, a common approach is to employ a combination

of Convolutional Neural Networks (CNNs) for initial feature extraction from the raw EEG data, followed by a Bidirectional Recurrent Neural Network (RNN) for sequence-level modeling.

In the DeepSleepNet architecture developed by authors in [36], a CNN was trained specifically for feature extraction from the single-channel EEG input. The CNN effectively captures informative patterns and representations from the raw EEG data. Subsequently, a bidirectional Long Short-Term Memory (LSTM) network was trained on top of the CNN's extracted features. The bidirectional LSTM takes into account both past and future information during the prediction process, allowing for improved modeling of the temporal dependencies in the sleep staging task.

The overall architecture of DeepSleepNet, combining the CNN and bidirectional LSTM, is depicted in Figure 2.6. This architecture enables the model to capture both local and global dependencies in the EEG epochs, leveraging the hierarchical nature of sleep patterns for more accurate sleep stage predictions.

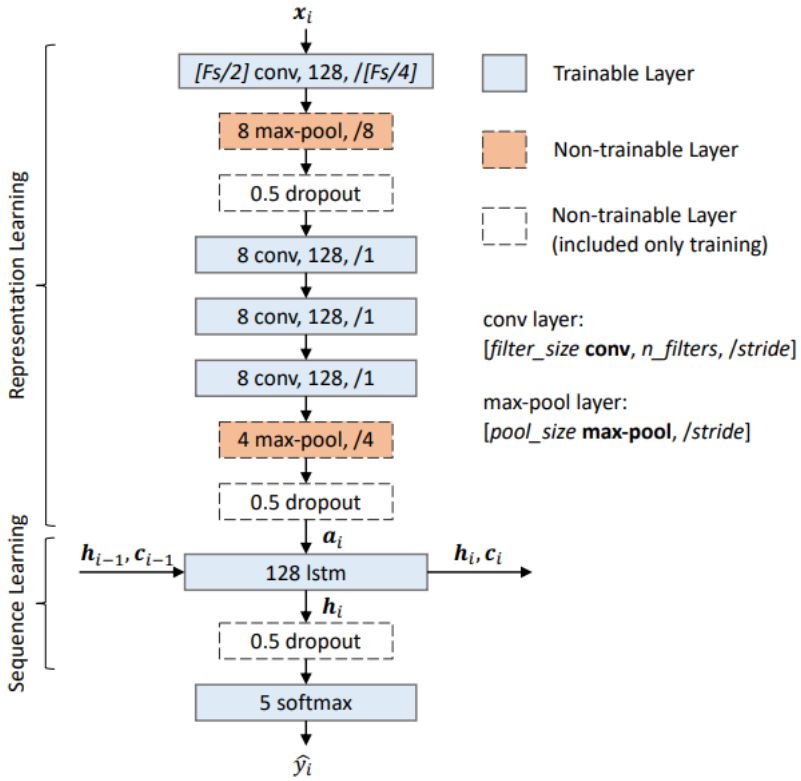


FIGURE 2.6: The architecture used by TinySleepNet [36], uses a convolutional encoder at the epoch level to extract meaningful features directly from the raw single-channel EEG, and an LSTM to augment it with sequence level information at the sequence level. This Figure is adapted from [36].

2.2.3 Transformer neural network

The transformer architecture, initially introduced in the realm of sequence processing, particularly for natural language processing (NLP) [40], addresses two key issues encountered with recurrent neural networks (RNNs). Firstly, transformers can process the entire input sequence simultaneously, thanks to parallelization, resulting in faster training and inference times. On the other hand, RNNs handle data sequentially, limiting their parallel processing capabilities. Secondly, transformers employ self-attention mechanisms to compute the relevance of each input token to all others, enabling them to capture relationships more effectively. In contrast, RNNs rely on hidden states, which may lead to information loss or distortion over time.

The fundamental building block of transformers is **self-attention**, which is just attention applied to the input itself, which allows the network to *attend* to certain parts of the input and connect them to each other. In NLP for example, it allows the network to link the personal pronoun to the correct tense of the verb by attending to both.

Attention as a dictionary lookup

Attention can be conceptualized as a 'soft' dictionary lookup operation. Given a set of m feature vectors consisting of **key-value** pairs $(\boldsymbol{k} \in \mathbb{R}^k, \boldsymbol{v} \in \mathbb{R}^v)_{i=1}^m$, we organize them into matrices $\boldsymbol{K} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{m \times v}$. When provided with a **query** $\boldsymbol{q} \in \mathbb{R}^q$, a differentiable (i.e., soft) dictionary lookup can be performed by computing a convex combination of the values as follows:

$$Attn(\boldsymbol{q}, (\boldsymbol{k}_1, \boldsymbol{v}_1), \ldots, (\boldsymbol{k}_m, \boldsymbol{v}_m) = \sum_{i=1}^{m} \alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:m}) \boldsymbol{v}_i \in \mathbb{R}^v$$

Here, $\alpha_i(\boldsymbol{q}, \boldsymbol{k}1 : m)$ denotes the i-th attention weight, which satisfies $0 \leq \alpha_i(\boldsymbol{q}, \boldsymbol{k}1 : m) < 1$ for each i, and $\sum_i \alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:m}) = 1$.

To calculate the attention weights, **scaled dot-product attention** is often used. This only works when the dimension of the keys and query equal each other: $k = q = d$ because their dot-product can then be calculated:

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^T \boldsymbol{k} / \sqrt{d} \in \mathbb{R}$$

where $a(\boldsymbol{q}, \boldsymbol{k})$ is the attention function from which the attention weights can be calculated with a softmax: $\alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:m}) = \frac{\exp(a(\boldsymbol{q}, \boldsymbol{k}_i))}{\sum_{j=1}^{m} \exp(a(\boldsymbol{q}, \boldsymbol{k}_j))}$. This dictionary lookup is nicely visualized in Figure 2.7.

To do this for a minibatch of n vectors at a time, there need to be n queries of dimension d: $Q = \begin{bmatrix} \boldsymbol{q}_1 & \ldots & \boldsymbol{q}_n \end{bmatrix}^T \in \mathbb{R}^{n \times d}$. If the keys and value matrices are denoted by $\boldsymbol{K} \in \mathbb{R}^{m \times d}$ and $\boldsymbol{V} \in \mathbb{R}^{m \times v}$, the attention-weighted outputs can be calculated as follows:

$$Attn(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathcal{S}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}})\boldsymbol{V} \in \mathbb{R}^{n \times v}$$

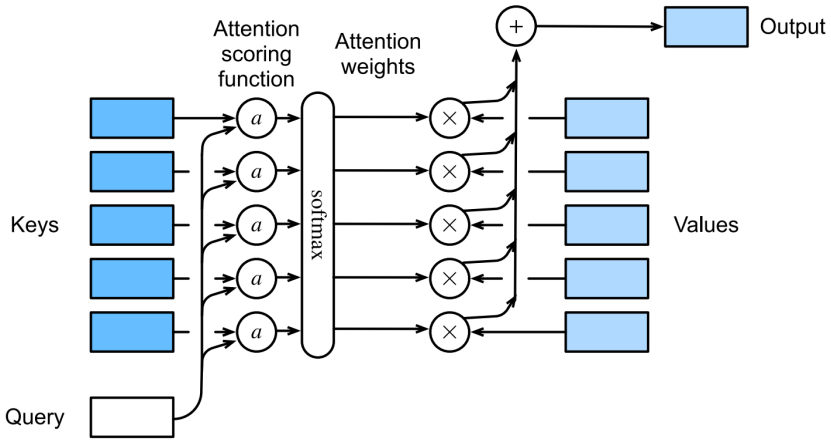


FIGURE 2.7: Attention as as a soft dictionary lookup, adopted from [25]. The output is a convex combination of the *values* with weights calculated as a function of how similar the *query* is to the *keys*.

Self-Attention and the Transformer Architecture

Self-attention is a mechanism that allows the input sequence to attend to itself. In the context of a sequence of input tokens $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, self-attention generates an output sequence of the same length using the following formulation:

$$\boldsymbol{y}_i = \text{Attn}(\boldsymbol{x}_i, (\boldsymbol{x}_1, \boldsymbol{x}_1), \ldots, (\boldsymbol{x}_n, \boldsymbol{x}_n))$$

In the transformer architecture, self-attention layers are used. They perform self-attention in a *learned* embedding space. Specifically, both the query and the key-value pairs are obtained through linear transformations of the input sequence. The output $\boldsymbol{h}_i$ of an individual **attention head** can be computed as:

$$\boldsymbol{h}_i = \text{Attn}(\boldsymbol{W}_i^{(q)}\boldsymbol{q}, \{\boldsymbol{W}_i^{(k)}\boldsymbol{k}_j, \boldsymbol{W}_i^{(v)}\boldsymbol{v}_j\})$$

Here, $\boldsymbol{W}_i^{(q)}$, $\boldsymbol{W}_i^{(k)}$, and $\boldsymbol{W}_i^{(v)}$ are learnable linear transformation matrices. Both the query $\boldsymbol{q}$ and the key-value pairs $(\boldsymbol{k}_j, \boldsymbol{v}_j)$ are the inputs to the self-attention layer.

Multiple attention heads can be utilized to form a **multi-headed attention** $\boldsymbol{h}$, where the outputs of each attention head are concatenated and linearly transformed:

$$\boldsymbol{h} = \text{MHA}(\boldsymbol{q}, \{\boldsymbol{k}_j, \boldsymbol{v}_j\}) = \boldsymbol{W}_o \begin{bmatrix} \boldsymbol{h}_1 \\ \vdots \\ \boldsymbol{h}_h \end{bmatrix}$$

In this equation, h represents the number of attention heads, and o denotes the output dimension. $\boldsymbol{W}_o$ is another learnable linear transformation applied to the concatenated outputs.

By employing self-attention with multiple attention heads, the transformer architecture can capture different types of relationships and dependencies in the input sequence, leading to more expressive and effective representations.

The transformer architecture consists of an encoder and a decoder network as in Figure 2.8. The encoder is used to encode the sequence into a meaningful representation that can be used by the decoder to generate sequences.

The encoder in the Transformer architecture plays a crucial role in processing the input sequence and creating a comprehensive representation of it. It consists of a stack of identical layers, each containing two sub-layers. The first sub-layer is a self-attention layer that enables each position in the sequence to attend to all other positions, capturing both local and global dependencies. This self-attention mechanism allows the encoder to incorporate contextual information from the entire input sequence for each position. The second sub-layer is a feed-forward neural network that applies a non-linear transformation to each position independently. This network enhances the representation by introducing additional complexity and modeling higher-level interactions between the elements in the sequence. Through these stacked layers of self-attention and feed-forward neural networks, the encoder generates a sequence of context-aware representations that encode rich information about the input sequence. [40]

The decoder is not used in this thesis.

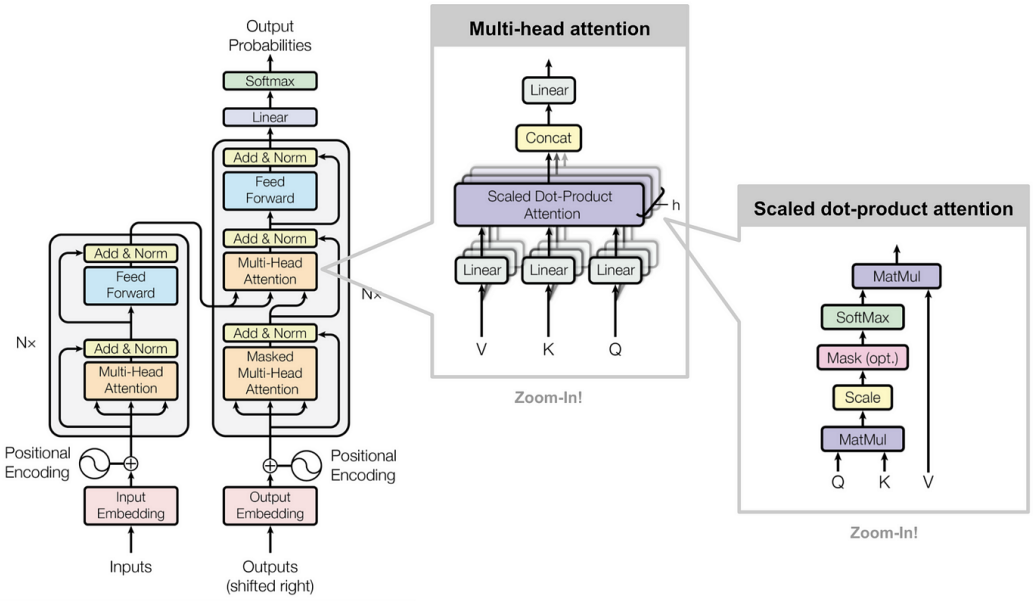


FIGURE 2.8: The transformer-model architecture. Adopted from [40].

Application to EEG

The study presented in [30] focuses on the exclusive use of the encoder component of the transformer model for sequence-to-sequence classification in sleep staging. The proposed architecture, illustrated in Figure 2.9, incorporates a hierarchical structure comprising two transformers.

In this architecture, the "inner" transformer functions as a feature extractor, transforming a sleep epoch's time-frequency image $\boldsymbol{S} \in \mathbb{R}^{l \times d}$ into a feature vector $\boldsymbol{x} \in \mathbb{R}^d$. This transformation is achieved through a series of transformer-encoder blocks, where each block consists of a multi-headed attention mechanism $\boldsymbol{H}_i$ and a feedforward neural network. The attention mechanism calculates queries $\boldsymbol{Q}_i$, keys $\boldsymbol{K}_i$, and values $\boldsymbol{V}_i$, which are then used to compute the attention output $\boldsymbol{H}_i$. By concatenating the individual attention heads $\boldsymbol{H}_1$ to $\boldsymbol{H}_H$, the final attentive output $\tilde{\boldsymbol{Z}}$ is obtained.

$$\boldsymbol{Q}_i = \boldsymbol{S}\boldsymbol{W}_i^Q,\ \boldsymbol{K}_i = \boldsymbol{S}\boldsymbol{W}_i^K,\ \boldsymbol{V}_i = \boldsymbol{S}\boldsymbol{W}_i^V, \tag{2.3}$$

$$\boldsymbol{H}_i = \text{Attention}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i) \tag{2.4}$$

$$= \text{softmax}(\frac{\boldsymbol{Q}_i \boldsymbol{K}_i^T}{\sqrt{d}})\boldsymbol{V}_i \tag{2.5}$$

$$\tilde{\boldsymbol{Z}} = \text{Concat}(\boldsymbol{H}_1, \ldots, \boldsymbol{H}_H)\boldsymbol{W}^Z \tag{2.6}$$

Here $\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i \in \mathbb{R}^{l \times \frac{d}{H}}$ are the mapped queries, keys, and values. $\boldsymbol{H}_i$ is the i-th attention head. $\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K, \boldsymbol{W}_i^V \in \mathbb{R}^{d \times \frac{d}{H}}$ and $\boldsymbol{W}^Z \in \mathbb{R}^{d \times d}$ are the learnable weight matrices. Finally $\tilde{\boldsymbol{Z}} \in \mathbb{R}^{l \times d}$ is the attentive output.

The feed-forward network is a fully connected MLP with one hidden layer and ReLU activation. Layer normalization and a residual connection are applied to enhance training convergence in each transformer-encoder block.

$$\boldsymbol{Z}_{mid} = \text{Layernorm}(\tilde{\boldsymbol{Z}} + \boldsymbol{Z}) \tag{2.7}$$

$$\boldsymbol{Z}_{FF} = \text{ReLU}(\boldsymbol{Z}_{mid}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{2.8}$$

$$\boldsymbol{O} = \text{LayerNorm}(\boldsymbol{Z}_{mid} + \boldsymbol{Z}_{FF}) \tag{2.9}$$

Each input epoch $\boldsymbol{S}$ goes through $N_E = 6$ of these transformer-encoder blocks and at the end, a final attention is performed to reduce the dimension of the output from $\mathbb{R}^{l \times d}$ to a single feature vector $\boldsymbol{x} \in \mathbb{R}^d$ for each input epoch.

Then at the sequence level, the 'outer' transformer takes as input a set of L sequential feature vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_L\}$, and sends them through $N_S = 6$ transformer-encoder blocks, to produce L output vectors $\{\boldsymbol{O}_1, \ldots, \boldsymbol{O}_L\}$. Those are then finally fed through a classification head, an MLP with one hidden layer, to produce logits over the sleep stages. The whole network can then be trained with a cross-entropy loss:

$$\mathcal{L} = -\frac{1}{L}\sum_{i=1}^{L} \boldsymbol{y}_i \log(\hat{\boldsymbol{y}}_i)$$

This architecture is considered to be state-of-the-art at the moment.

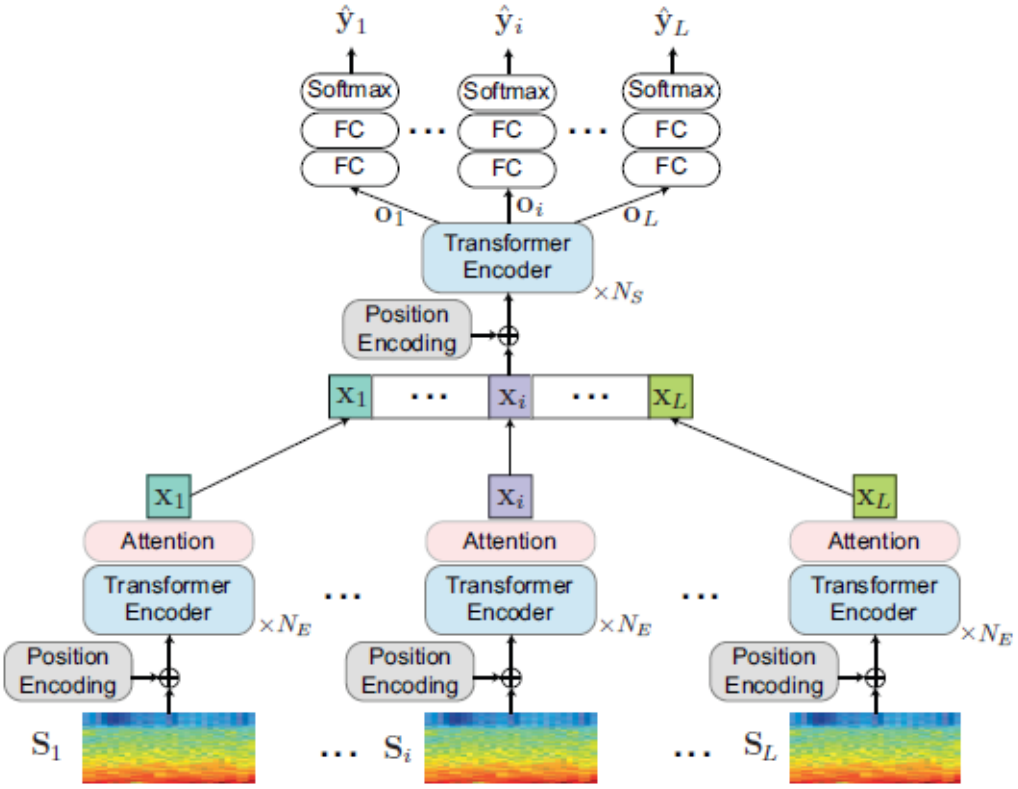


FIGURE 2.9: The SleepTransformer architecture, adopted from [30].

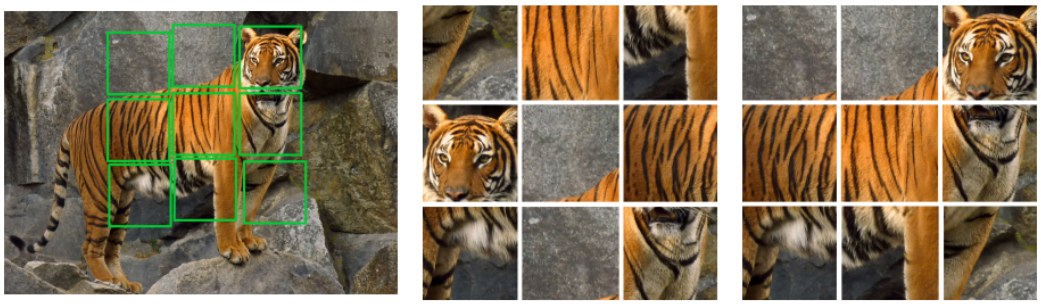
2.3 Existing literature on self-supervised learning

To harness the potential of unlabeled data, a neural network is initially *pretrained* using an SSL objective. When this objective is closely aligned with the downstream task, the pretraining process yields well-initialized parameters for the neural network. This network is then transferred to the downstream task for fine-tuning, which will require much fewer data due to the good parameter initialization.

In a Bayesian framework, one can think of SSL pretraining as imposing a "prior" distribution over the parameters of the neural network. This prior knowledge aids in the subsequent fine-tuning on labeled data.

SSL objectives can be generically classified into three categories: pretext tasks, generative methods, and contrastive methods. Pretext tasks are tasks constructed from the unlabeled data itself and they should require the neural network to learn fundamental information about the data in order to solve them. For example, a popular task in computer vision is to solve a jigsaw puzzle [26] as in Figure 2.10.

The authors of [6] propose two pretext tasks for EEG: "relative positioning" and "temporal shuffling". These tasks involve training a model to predict whether two sleep epochs are close in time. The authors argue that the model learns the underlying structure by solving these tasks and show that the learned representations are physiologically meaningful.



(a) (b) (c)

FIGURE 2.10: A jigsaw puzzle pretext task, adopted from [26]. A neural network is trained to solve the puzzle and therefore has to learn fundamental information about the image.

Generative methods, on the other hand, require a neural network to *generate* the input. Examples of generative methods include masked, denoising, and variational autoencoders. The key concept behind this approach is that the latent space should contain all the relevant information necessary for generating valid examples, which can then be utilized in downstream tasks.

The focus of this thesis is on *contrastive methods* however, which have gained much attention recently due to their success in computer vision with SimCLR [10] and in natural language processing (NLP) with BERT [12]. In SimCLR, a neural network is trained to recognize an augmented version of an input image, within a batch comprising the original image and other images. Augmentations may involve

slight rotations, flips, or other modifications. To solve this task, it has to *contrast* the augmented input image with all the other images in the batch and find the original. SimCLR will be explained in detail in the next section.

In BERT, a large language model, a neural network is trained to predict a missing word in a sentence by picking one from a corpus of possible words. In order to solve this task, BERT has to *contrast* the context given by the sentence to every word in the corpus, until it finds the most likely one [3]. Figure 2.11 shows what this architecture looks like at a high level.

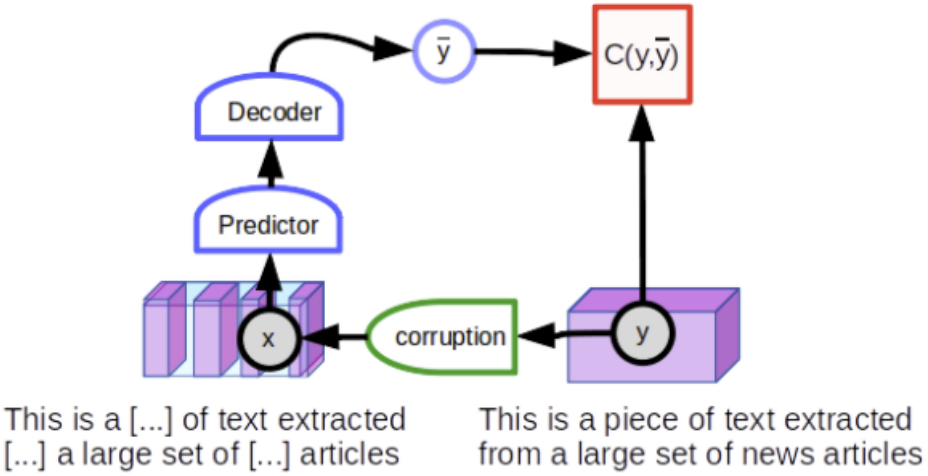


FIGURE 2.11: This Figure, adopted from [3], shows an example of how a large language model like BERT [12] can be pretrained with a contrastive learning task. The model is trained to predict which word is masked by comparing the context to a corpus of words.

The most promising of the state-of-the-art contrastive SSL methods will now be discussed in detail. We will then choose one for EEG-based sleep staging in Chapter 3.

2.3.1 Contrastive methods

SimCLR

A simple way of teaching a neural network a representation of objects in images is by teaching it to recognize the object when viewed slightly differently: a rotated dog is still a dog. This is exactly what the SimCLR framework does: it learns a representation *invariant to augmentations*.

Figure 2.12 displays what SimCLR does in computer vision (CV): two different augmented views of every image in a minibatch of N images are fed through a neural network encoder — for CV, a CNN is the natural choice — until $2N$ lower dimensional representation vectors $\{\boldsymbol{h}\}_{i=1}^{2N}$ are obtained.

A contrastive loss is not calculated in this space but in $\boldsymbol{z}$-space by sending the features through an MLP: the projection head. One can intuitively explain the need for $\boldsymbol{z}$-space by looking through the lens of *transfer learning*. One can think of the features being a rich representation of the images themselves and the $\boldsymbol{z}$ features only being relevant to solve the contrastive learning task. Thus when transferring the model to a downstream task, the projection head is discarded [5].

The contrastive loss, used in SimCLR is the *NT-Xent* (Normalized temperature-scaled cross-entropy loss):

$$l_{i,j} = -\log \frac{\exp\left(sim(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau\right)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(sim(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)} \tag{2.10}$$

where $sim(\boldsymbol{z}_i, \boldsymbol{z}_j) = \frac{\boldsymbol{z}_i^T \boldsymbol{z}_j}{\boldsymbol{z}_i \boldsymbol{z}_j}$ is the cosine similarity.

This loss function can be understood as classifying one augmentation of the input with the other augmentation coming from the same image. This classification happens on the hyperdimensional ball in $\boldsymbol{z}$-space with radius τ defined by the $2N-1$ 'negatives' in the batch and the 'positive', i.e. the other augmentation. *The positives are pulled closer and the negatives are pushed away from the positives.*

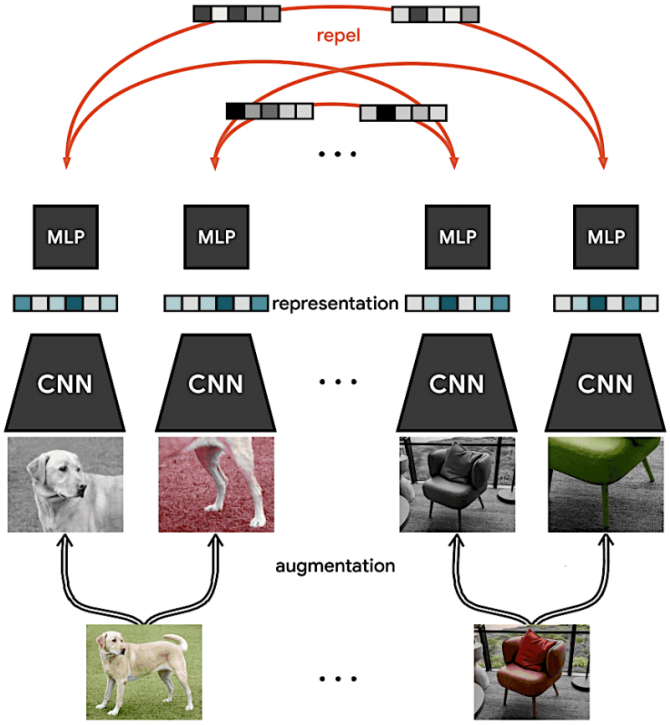


FIGURE 2.12: SimCLR for images, adopted from [10]. In a batch containing N input images, each image undergoes two augmentations. As a result, there are $2N-1$ 'negatives' for every 'positive'. The contrastive loss operates by attracting the positive pairs closer together while pushing the negative pairs further apart.

SwAV

One limitation of SimCLR is the requirement for a large batch size to ensure an adequate number of negative samples for the contrastive learning task. However, recent discussions have raised some disputes regarding this argument [5]. To address this issue, Swapped Assignments between multiple Views (SwAV) [9] offers a novel approach that eliminates the need for negatives and pairwise comparisons.

Instead of relying on negative samples, SwAV employs clustering techniques to group similar data points together. It enforces the augmented versions of the input to share the same cluster assignment by predicting the code of one view using the representation of another view. This approach allows for the comparison of multiple views of the input based on their cluster assignments rather than their features.

To be more specific, SwAV computes a code for an augmented version of the input and predicts this code using another augmented version of the same input. Given two features, $\boldsymbol{z}_s$ and $\boldsymbol{z}_t$, obtained from different augmentations of the same image, their corresponding codes, $\boldsymbol{q}_s$ and $\boldsymbol{q}_t$, are computed by matching these features to a set of K prototypes, $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$. Then, a "swapped" prediction problem is formulated with the following loss function:

$$\mathcal{L}(\boldsymbol{z}_s, \boldsymbol{z}_t) = \ell(\boldsymbol{z}_s, \boldsymbol{q}_t) + \ell(\boldsymbol{z}_t, \boldsymbol{q}_s) \tag{2.11}$$

where $\ell(\boldsymbol{z}, \boldsymbol{q})$ measures the similarity between features $\boldsymbol{z}$ and code $\boldsymbol{q}$.

To illustrate the distinction between classic contrastive instance learning and SwAV, refer to Figure 2.13, which provides a visual representation of the two approaches.

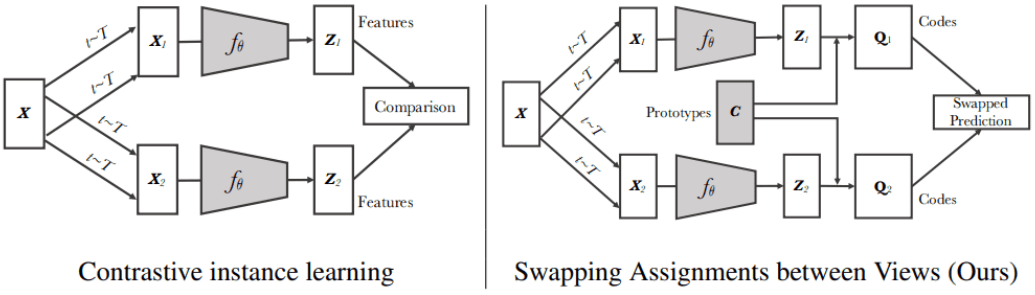


FIGURE 2.13: Contrastive instance learning (left) vs. SWaV (right), adopted from [9]. Instead of directly comparing features as in SimCLR, SwAV first predicts a code for each feature by assigning each feature to a prototype vector. A 'swapped' prediction problem is then solved where the code of one view is predicted from the code of another view of the same input.

The loss function $\ell(\boldsymbol{z}, \boldsymbol{q})$ is the cross entropy loss between the code and the probability obtained by taking a softmax of the dot products of $\boldsymbol{z}_i$ and all prototypes

$\{\boldsymbol{c}_j\}_{j=0}^{C}$:

$$\ell(\boldsymbol{z}_k, \boldsymbol{q}_s) = -\sum_k \boldsymbol{q}_s^{(k)} \log \boldsymbol{p}_t^{(k)}, \quad \text{where } \boldsymbol{p}_t^{(k)} = \frac{\exp(\boldsymbol{z}_t^T \boldsymbol{c}_k/\tau)}{\sum_j \exp(\boldsymbol{z}_t^T \boldsymbol{c}_j/\tau)} \tag{2.12}$$

Minimizing this loss function can be interpreted as a classification problem where the code of the first augmentation of an input is predicted from the code of the second augmentation of the input.

BYOL

Bootstrap Your Own Latent (BYOL)[14] is another contrastive method that doesn't require a big batch size and claims to outperform SimCLR.

SimCLR learns meaningful representations by teaching a neural network to discriminate a positive pair (two augmentations from the same image) from many negative pairs (augmentations of different images). BYOL takes a different approach by eliminating the negatives and directly predicting the representation of an augmented view of an image using another augmented view of the same image. However, the main challenge in this task is to prevent *representation collapse*, where the network predicts a constant value for all inputs as the easiest solution.

To prevent representation collapse, BYOL employs two networks: an online network parameterized by $\boldsymbol{\theta}$ and a target network with an identical architecture but parameterized by $\boldsymbol{\xi}$, as depicted in Figure 2.14. Two augmentations, $\boldsymbol{v}$ and $\boldsymbol{v}'$, are generated from the input $\boldsymbol{x}$. One augmentation is passed through the online network to produce a representation $\boldsymbol{y}_{\boldsymbol{\theta}}$, while the other augmentation is fed into the target network to produce $\boldsymbol{y}'_{\boldsymbol{\xi}}$. Similar to SimCLR, both representations are projected into another space, yielding $\boldsymbol{z}_{\boldsymbol{\theta}}$ and $\boldsymbol{z}'_{\boldsymbol{\xi}}$, where a contrastive loss is calculated. The loss function $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\xi}}$ measures the reconstruction error between the target representation $\boldsymbol{z}'_{\boldsymbol{\xi}}$ and the online projection, which is first passed through a prediction head $q_{\boldsymbol{\theta}}(\boldsymbol{z}_{\boldsymbol{\theta}})$. Crucially, this reconstruction loss is only backpropagated through the online network, while the parameters of the target network are updated as an exponentially moving average of the online network's parameters. This procedure is iterated.

The authors argue representation collapse is avoided because although the collapse is a minimum of the loss function $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\xi}}$, the network is updated only in the direction of $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\xi}}$ and the target network updates are **not** in the direction of $\nabla_{\boldsymbol{\xi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\xi}}$ such that a joint minimum is avoided.

The reconstruction loss itself is a ℓ_2-normalized MSE:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\xi}} = \|\overline{q_{\boldsymbol{\theta}}}(\boldsymbol{z}_{\boldsymbol{\theta}}) - \overline{\boldsymbol{z}_{\boldsymbol{\xi}}}'\|_2^2 \tag{2.13}$$

where $\overline{q_{\boldsymbol{\theta}}}(\boldsymbol{z}_{\boldsymbol{\theta}}) = \frac{q_{\boldsymbol{\theta}}(\boldsymbol{z}_{\boldsymbol{\theta}})}{\|q_{\boldsymbol{\theta}}(\boldsymbol{z}_{\boldsymbol{\theta}})\|}$ and $\overline{\boldsymbol{z}_{\boldsymbol{\xi}}}' = \frac{\boldsymbol{z}'_{\boldsymbol{\xi}}}{\|\boldsymbol{z}'_{\boldsymbol{\xi}}\|}$.

SiamSiam

In the paper "SiamSiam" [11], researchers from Facebook argue against the necessity of a momentum encoder in the BYOL method to prevent collapse. They propose

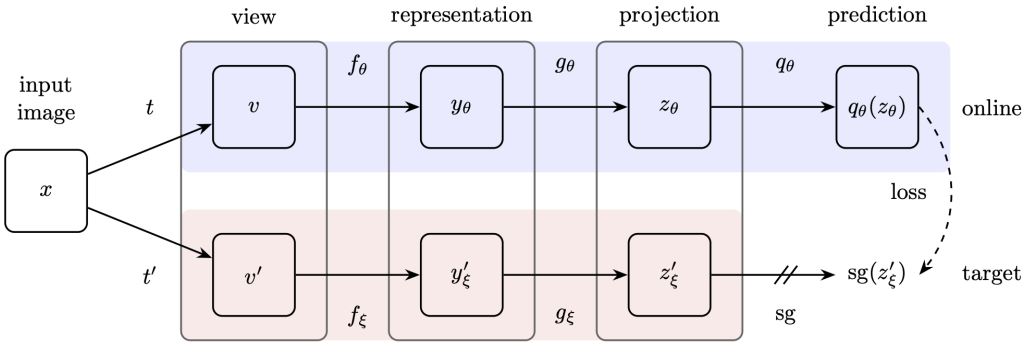


FIGURE 2.14: BYOL methodology, adopted from [14]. Two different augmented views of an input $\boldsymbol{x}$ are put through the same encoder-projection architecture, but with different parameters to produce two projections: $\boldsymbol{z_\theta}$ and $\boldsymbol{z'_\xi}$. The second augmentation is then predicted from the first through a projection head: $q_{\boldsymbol{\theta}}(\boldsymbol{z_\theta})$. Importantly, only the parameters of the network parameterized by $\boldsymbol{\theta}$ are updated to prevent collapse.

an alternative approach where they utilize a single encoder, denoted as f, for both augmentations, as illustrated in Figure 2.15.

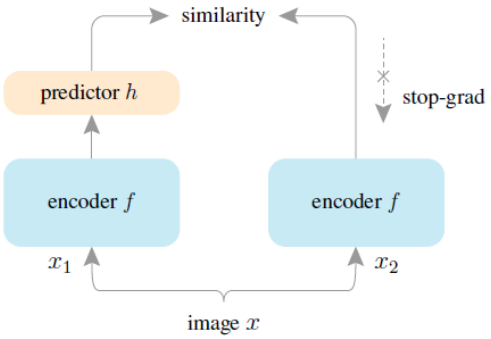


FIGURE 2.15: Siamsiam methodology, adopted from [11]. Every input $\boldsymbol{x}$ is augmented twice to produce $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. Both are then sent through the same encoder f to produce their feature representations $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$. The network is then trained by predicting $\boldsymbol{z}_2$ from $\boldsymbol{z}_1$ through a prediction head h.

The inputs to the architecture are two differently augmented views of the same input: $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. They are both fed through the same encoder f to produce two representation vectors $\boldsymbol{z}_1$, $\boldsymbol{z}_2$. A prediction head further predicts $\boldsymbol{z}_2$ from $\boldsymbol{z}_1$ with an MLP: $\boldsymbol{p}_1 = h(\boldsymbol{z}_1) = h(f(\boldsymbol{x}_1))$, with a negative cosine similarity loss function:

$$\mathcal{D}(\boldsymbol{p}_1, \boldsymbol{z}_2) = -\frac{\boldsymbol{p}_1}{\|\boldsymbol{p}_1\|_2} \cdot \frac{\boldsymbol{z}_2}{\|\boldsymbol{z}_2\|_2} \tag{2.14}$$

To achieve symmetry in the loss function, the representation $\boldsymbol{z}_1$ is also predicted

from $\boldsymbol{z}_2$ using $\boldsymbol{p}_2 = h(\boldsymbol{z}_2)$:

$$\mathcal{L} = \frac{1}{2}\mathcal{D}(\boldsymbol{p}_1, \boldsymbol{z}_2) + \frac{1}{2}\mathcal{D}(\boldsymbol{p}_2, \boldsymbol{z}_1) \tag{2.15}$$

However, it is important that the loss function is only backpropagated through the prediction network to prevent collapse, which happens when the network predicts a constant for all inputs.

Thus SiamSiam doesn't need (i) negative sample pairs, (ii) large batches, (iii) momentum encoders; which makes it a very straightforward method.

CPC

Contrastive Predictive Coding (CPC) is not an improvement upon SimCLR, but exploits the sequential nature of time series $\{\boldsymbol{x}_t\}_{t=0}^{T}$.

It works by first extracting a representation $\boldsymbol{z}_t = g_{enc}(\boldsymbol{x}_t)$ from the high dimensional data $\boldsymbol{x}_t$. This representation is then used by an autoregressive model $\boldsymbol{c}_t = g_{ar}(\boldsymbol{z}_{\leq t})$ that maximizes the mutual information between $\boldsymbol{c}_t$ and the prediction $\boldsymbol{z}_{t+k}$. The main intuition behind this idea is for the model to learn representations $\boldsymbol{z}_t$ that encode the underlying shared information between different parts of the high dimensional signal while discarding low-level information and noise that is more local. This methodology is nicely displayed in Figure 2.16.

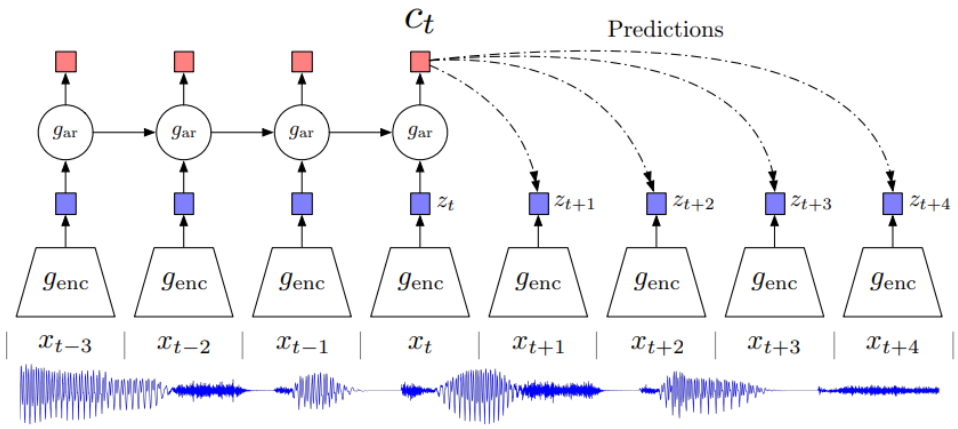


FIGURE 2.16: CPC Methodology, adopted from [37].

More precisely, one doesn't model the generative distribution $p_k(\boldsymbol{x}_{t+k}|\boldsymbol{c}_t)$ directly, but instead a density ratio which preserves the mutual information between $\boldsymbol{x}_{t+k}$ and $\boldsymbol{c}_t$:

$$f_k(\boldsymbol{x}_{t+k}, \boldsymbol{c}_t) \propto \frac{p_k(\boldsymbol{x}_{t+k}|\boldsymbol{c}_t)}{p_k(\boldsymbol{x}_{t+k})} \tag{2.16}$$

The authors propose a simple log-bilinear model for f_k:

$$f_k(\boldsymbol{x}_{t+k}, \boldsymbol{c}_t) = \exp\left(\boldsymbol{z}_{t+k}^T W_k \boldsymbol{c}_t\right) \tag{2.17}$$

where W_k is different for every step k. By using a density ratio, the model is relieved from modeling the high dimensional distribution $\boldsymbol{x}_{t+k}$.

After successful training, either $\boldsymbol{c}_t$ or $\boldsymbol{z}_t$ could be used as a representation for downstream tasks, depending on whether the past is important or not for the downstream task.

Both the encoder and autoregressive model are trained to jointly optimize a loss based on NCE (Noise Contrastive Estimation) which the authors call InfoNCE. Given a set $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ of N random samples containing one positive sample from $p(\boldsymbol{x}_{t+k}|\boldsymbol{c}_t)$ (the sample itself) and $N-1$ negative samples from the 'proposal' distribution $p(\boldsymbol{x}_{t+k})$, the following objective is optimized:

$$\mathcal{L}_N = -\mathbb{E}_X \left[\log \frac{f_k(\boldsymbol{x}_{t+k}, \boldsymbol{c}_t)}{\sum_{\boldsymbol{x}_j \in X} f_k(\boldsymbol{x}_j, \boldsymbol{c}_t)} \right] \tag{2.18}$$

This loss function can be interpreted as the categorical cross-entropy of classifying the positive samples correctly. It can be shown that optimizing this loss results in $f_k(\boldsymbol{x}_{t+k})$ estimating the density of equation (2.16).

2.4 Conclusion

In this chapter, we explored the application of deep learning techniques for sleep staging, with a specific focus on specialized neural network architectures. We began by introducing deep learning and highlighting the limitations of traditional multilayer perceptron (MLP) models when dealing with high-dimensional data. To address these limitations, we examined three popular neural network architectures: Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformers.

CNNs were presented as a solution for processing high-dimensional data, such as images, in a hierarchical manner. We discussed the fundamental principles of CNNs, including the use of convolutional layers to extract local information and the benefits of parameter sharing and translation invariance. In the context of sleep staging using EEG data, CNNs are frequently employed for extracting features from raw EEG signals.

Next, we delved into RNNs, which are designed to process sequential data by incorporating recurrent connections. We described the architecture of a classical RNN, where the hidden state is updated based on the previous hidden state and the current input. We emphasized the unbounded memory property of RNNs, enabling them to capture temporal dependencies in the data. However, we also acknowledged the challenge of the vanishing gradient problem associated with classical RNNs.

To mitigate the vanishing gradient problem, we introduced two specialized types of RNNs: GRU and LSTM. These variants update the hidden state additively, rather than multiplicatively, through a gating mechanism.

Finally, we introduced the transformer architecture, which relies on self-attention. This allows the transformer to selectively focus on relevant inputs. Transformers are highly powerful architectures that can be trained in parallel, in contrast to RNNs. They can serve as both feature extractors and sequence models, as exemplified by

SleepTransformer [30].
In the second part of this chapter, we conducted a comprehensive literature study on state-of-the-art self-supervised learning (SSL) methods. We explored pretext tasks, which are tasks derived from the data itself, but our primary focus was on contrastive methods. These approaches enable learning of features that are invariant to augmentations, thereby capturing fundamental information about the input.

In the subsequent chapter, we will select an SSL method and a deep learning architecture to evaluate the chosen method. We will outline the experimental procedure, and ultimately present the results in Chapter 4.

Chapter 3

Methods

3.1 Introduction

To test SSL methods in EEG-based sleep staging, a deep learning architecture needs to be chosen first.

Almost all state-of-the-art deep learning models for sleep staging use a sequence-to-sequence architecture, that consist of three parts: an epoch encoder, a sequence encoder, and a classifier. The encoder extracts time-invariant features directly from the raw EEG epochs, and a non-causal, nonlinear regression model at the sequence level encodes temporal information such as stage transition rules. For example, DeepSleepNet [35] uses a CNN (Section 2.2.1) for the epoch encoder and an LSTM (Section 2.2.2) for the sequence encoder. SleepTransformer [30] (Section 2.2.3) on the other hand uses two transformers: an 'inner' transformer for representation learning (feature extraction) and an 'outer' transformer on top for sequence modelling. We will therefore also use a sequence-to-sequence deep learning architecture for sleep staging in this thesis.

The existing SSL methods for sleep staging either only pretrain the epoch-encoder, or pretrain both the epoch encoder and the sequence encoder simultaneously with the same pretext task. However, SSL has also been very successful where the different networks in the hierarchy are pretrained with different SSL objectives. For example, GPT [8], a very powerful large language model, consists of a hierarchy of transformers and they are pretrained with a different SSL objective at each level of abstraction. The innermost transformer is trained to predict a masked word in a sentence, while the outer transformer is trained to do next-sentence prediction.

This thesis will investigate whether the same paradigm can be successfully applied to EEG-based sleep staging. The epoch encoder will be pretrained with an SSL objective appropriate for feature extraction, and the sequence encoder with a different SSL objective, appropriate for sequences.

The SimCLR framework (section 2.3.1) will be used to pretrain the epoch encoder for feature extraction. Although better methods have arisen in the last few years such as ByOL [14] (Section 2.3.1), SwAV [9] (Section 2.3.1) and data2vec [4], none of these have been applied to EEG yet and SimCLR will be a good baseline for future

work. Researchers at MIT have done this already in [24], but this work will try to validate and improve their results by testing other architectures and another dataset.

To pretrain the sequence encoder, on the other hand, we come up with a pretext task that encourages the model to learn sequence-level information. Future work could investigate a contrastive method for time series, such as CPC [37] (Section 2.3.1), at this level as well.

The chapter will proceed as follows: Section 3.2 introduces the available dataset, followed by Section 3.3, where the adaptation of the SimCLR framework to EEG is discussed. Section 3.4 elaborates on the pretext task employed to pretrain the sequence encoder. Subsequently, sections 3.5.1 and 3.5.2 will discuss a qualitative and quantitative way to assess the quality of the SSL pretraining. Lastly, Section 3.6 addresses the available hardware and its associated limitations in relation to the learning task.

3.2 SHHS dataset

Throughout this thesis, the models are tested on only one dataset: the sleep heart health study (SHHS). This is a large-scale database collected from multiple centers to study the effect of sleep-disordered breathing on cardiovascular diseases [32]. The dataset consists of two rounds of polysomnography (PSG) records, but we only use the first: SHHS-1, which consists of 5791 subjects aged 39-90. Manual scoring was completed using the R&K guideline [43]. Similar to other databases annotated with this rule, N3 and N4 stages were merged into N3 and MOVEMENT and UNKNOWN epochs were discarded. Only the single channel EEG (C4-A1) is used, sampled at 125Hz. [30]

The EEG is then resampled at 100Hz and filtered with a bandpass FIR filter on [0.3,40] Hz as in [19]. Finally, the patients are split into a train-test set of 70-30% with a random split. Thus the total training set, including validation, consists of 4054 patients and the test set consists of 1737 patients.

Throughout the thesis there won't be any distinction between patients, i.e. the dataset is regarded as if it was collected from a single person that goes through a full night's sleep multiple times (the amount of actual patients).[1] Every sleep epoch $\boldsymbol{x}$ consists of 30 seconds of single-channel raw EEG sampled at 100Hz, such that $\boldsymbol{x} \in \mathbb{R}^{3000}$. Every epoch $\boldsymbol{x}$ has a corresponding sleep stage (label) $y \in \{\text{WAKE, N1, N2, N3, REM}\}$. When a consecutive sequence of L sleep-epochs is needed for temporal information, the following notation is used: $\boldsymbol{S} = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_L}, y_L)\}$.

3.3 Pretraining the epoch encoder with SimCLR

The SimCLR framework was originally developed for CV, and explained in detail in Section 2.3.1. In order to use the same framework to pretrain the epoch encoder

[1] Recently there has been some work where people exploit the fact that there *is* difference between patients to construct contrastive methods.[44]

for feature extraction, it needs to be adapted to EEG. Figure 4.3 shows what the framework looks like in this setting.

The inputs to the model are now 30-second segments of raw single-channel EEG time series, sampled at a rate of 100Hz. Augmentations specific for this input are needed and discussed below.

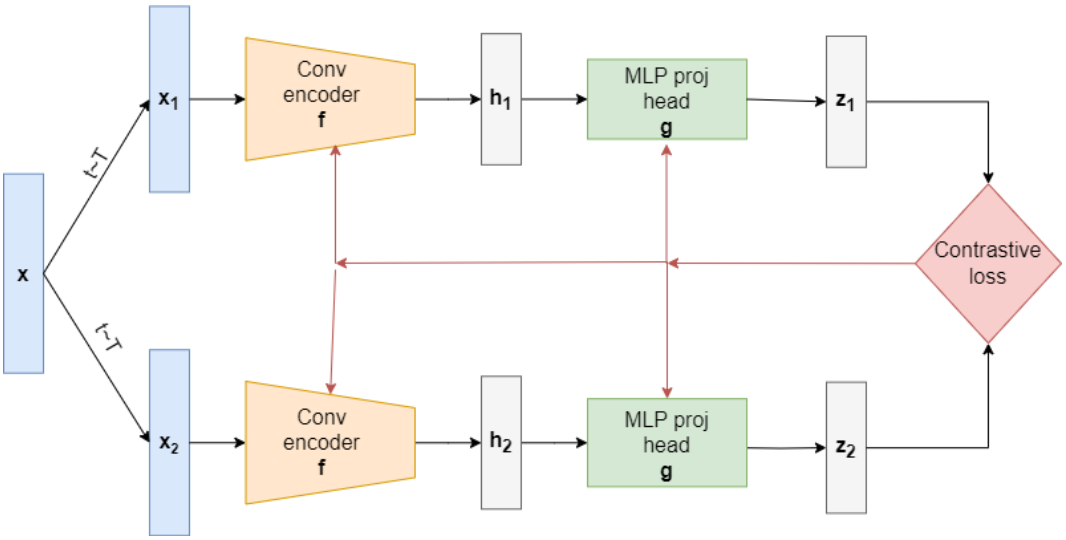


FIGURE 3.1: SimCLR for EEG: two different augmentations of every input EEG epoch $\boldsymbol{x} \in \mathbb{R}^{3000}$ in the batch are first produced by an augmentation module T: $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. Those are then fed through the encoder to obtain the $\boldsymbol{h}$-features and a projection head to obtain the $\boldsymbol{z}$-features: $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$. A contrastive loss pulls the $\boldsymbol{z}$-features closer and pushes those from other inputs further. Finally, when the pretraining is finished, the projection head is discarded and the $\boldsymbol{h}$-features are rich representations that can be used for the downstream task of sleep staging.

3.3.1 Augmentations

Before devising augmentations for EEG data, it is worthwhile to examine the augmentations employed for CV in the original SimCLR paper [10]. These are given in Figure 3.2. Clearly, certain augmentations like color distortion are not directly applicable to the 1-dimensional EEG time series. However, augmentations that can be adapted include horizontal flipping, adding Gaussian noise, cropping, and zero-masking. Notably, the authors conducted an ablation study, which concluded that both cropping and color distortion are the most important augmentations. Additionally, their findings indicate that a more challenging contrastive learning task, achieved through stronger augmentations, enhances the quality of the learned representations.

It is decided in this thesis to only retain zero-masking and the addition of Gaussian noise. Any augmentation involving cropping and resizing is discarded because this would lead to frequency distortion. However, a small time shift and a band-pass filter,

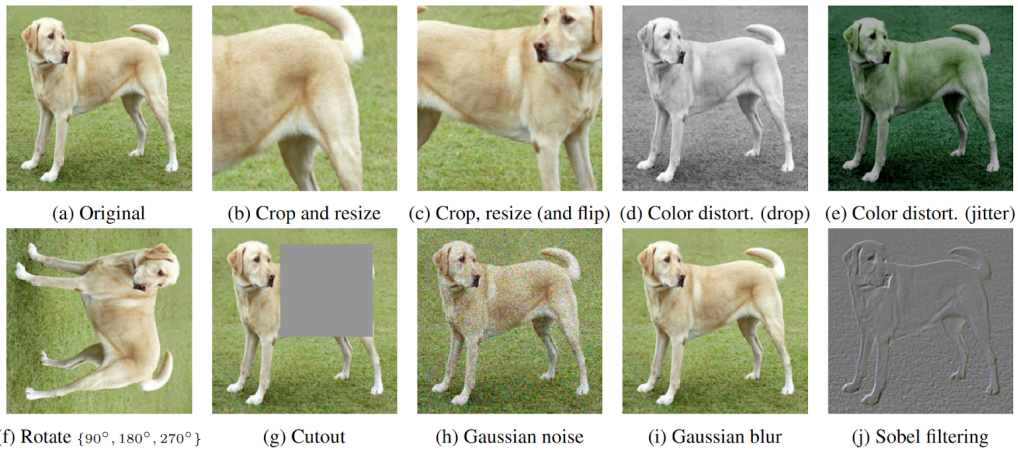


FIGURE 3.2: Augmentations used in original SimCLR paper, adopted from [10].

which can be understood as a masking operation in the frequency domain, are added. It is suspected that the band-pass filter will be especially important for sleep staging since some sleep stages are very active in specific frequency bands and others not as was explained in Section 1.2. Figure 3.3 displays what these augmentations look like for a typical EEG signal. It is further noted that these are the same augmentations that were successfully used in the paper [24].

Finally the augmentations are implemented in a way such that they can be performed on a whole mini-batch of training examples at once, on the GPU. This is important for the efficient use of the available resources (Section 3.6).

FIGURE 3.3: Augmentation used for EEG. The specific hyperparameters used to make this plot will be discussed in Chapter 4.

3.4 Pretraining the sequence encoder with a pretext task

We come up with a pretext task for pretraining the sequence encoder, which works as follows:

Given a sequential set of sleep epochs $\boldsymbol{S} = \{\boldsymbol{x}_1 \dots, \boldsymbol{x}_L\}$, the epoch encoder first transforms this set into a set of feature vectors $\boldsymbol{\mathcal{H}} = \{\boldsymbol{h}_1, \dots, \boldsymbol{h}_L\}$. The task of the sequence encoder is then to extract slow-moving, shared information between the features, while discarding low-level information and noise that is more local. Contrastive predictive coding (CPC), explicitly maximizes this objective, although only looking at the past, as was explained in detail in Section 2.3.1. We don't use CPC in this thesis, but come up with a pretext task that approximates the same objective. Future work can investigate whether CPC can improve upon our results.

Given a batch of N sequences of feature vectors $\{\boldsymbol{\mathcal{H}}^{(i)}\}_{i=1}^{N}$ (encoded sleep epochs): $\begin{bmatrix} \{\boldsymbol{h}_1^{(1)}, \dots, \boldsymbol{h}_L^{(1)}\} \\ \vdots \\ \{\boldsymbol{h}_1^{(N)}, \dots, \boldsymbol{h}_L^{(N)}\} \end{bmatrix}$, a random index in the range $[1, L]$ is chosen for each sequence in the batch. These N indices correspond to N random feature vectors across the batch. Those are shuffled through the batch such that each sequence contains one feature vector that was not in the sequence originally. The pretext task for the network to solve is then simply to predict which index in each sequence is out of place. However, it may happen by chance that a feature ends up in the same place after shuffling: with a probability of $1/N$, thereby imposing an upper bound of $1 - \frac{1}{N}$ on the attainable accuracy of the pretext task.

In order to solve this task, the sequence encoder must extract slow-moving, shared information between the epochs and decide which of the epochs is out of place, thereby learning crucial information about the sequence.

The task of the network is thus to 'classify' the out-of-place feature vector for each example in the batch. To this end, the features $\{\boldsymbol{h}_1, \dots, \boldsymbol{h}_C, \dots, \boldsymbol{h}_L\}$, where $\boldsymbol{h}_C$ is the corrupted feature vector, are first sent through the sequence encoder to produced new feature vectors that are augmented with temporal information: $\{\boldsymbol{t}_1, \dots, \boldsymbol{t}_L\}$. Those features are then 'aggregated' into a single context vector $\boldsymbol{c}$ with a single self-attention (Section 2.2.3). This context is then the input to an MLP prediction head which predicts the index of the corrupted feature vector (C). The whole pretraining architecture is depicted in Figure 3.4.

Finally the loss function is the categorical cross-entropy between the 'pseudo-labels' $\boldsymbol{y}$, i.e. the indices of the feature vectors that really are out of place, and the network's prediction $\hat{\boldsymbol{y}}$:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{L} y_i \log \hat{y}_i \tag{3.1}$$

Just like in SimCLR, the prediction head is discarded after pretraining.

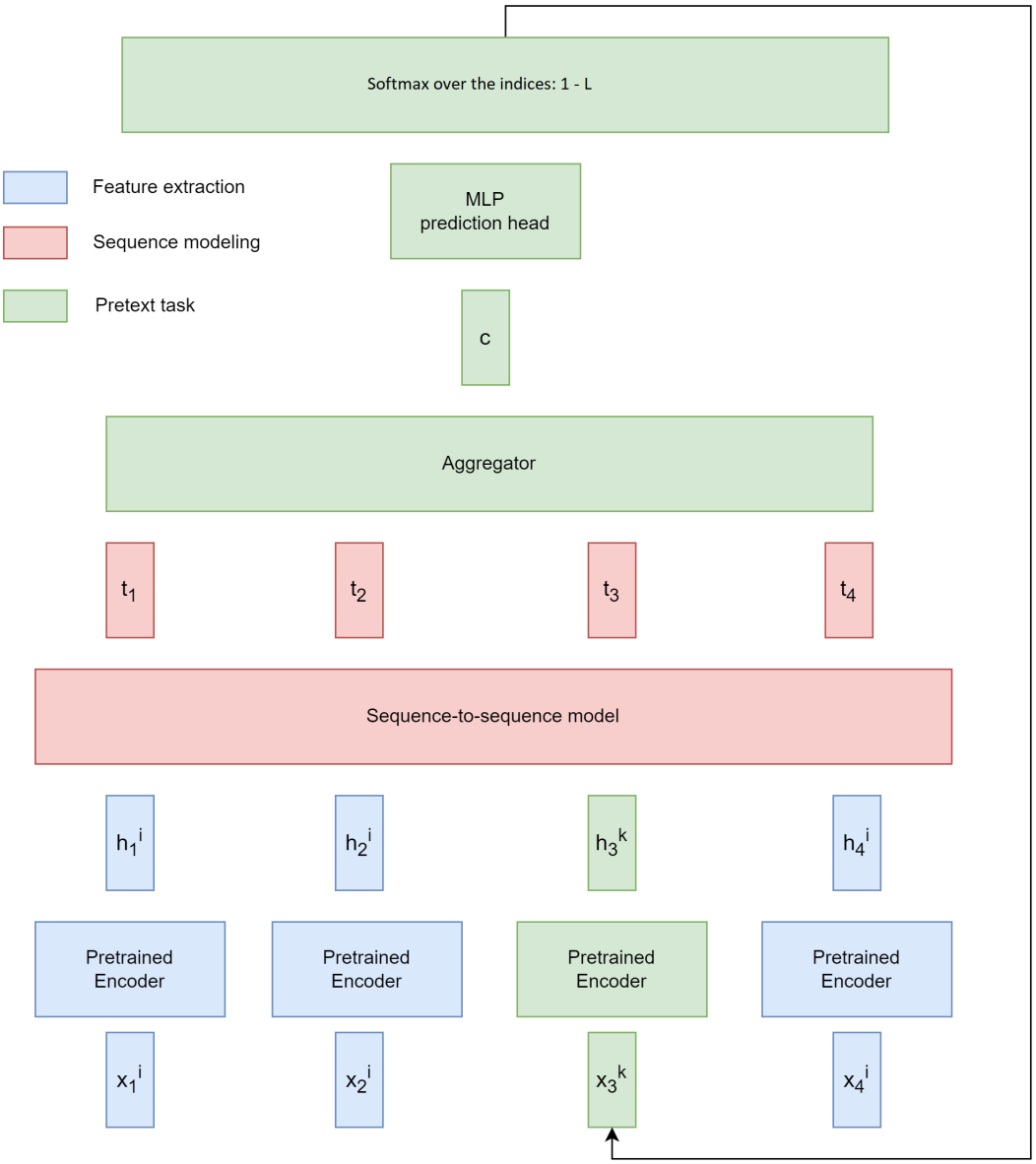


FIGURE 3.4: Example of pretext task for $L = 4$, where the third feature vector does not belong in the sequence. The 4 input EEG epochs $\{\boldsymbol{x}_1 \ldots, \boldsymbol{x}_4\}$ are first encoded into 4 feature vectors by the pretrained epoch encoder: $\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_L\}$. Those are encoded by the sequence encoder into 4 feature vectors $\{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_L\}$, that have incorporated sequential information. An aggregator module uses self-attention to combine those 4 into 1 context vector $\boldsymbol{c}$, which is the input to a prediction head. The prediction head predicts the index of the out-of-place vector (3 in this example) by outputting probabilities over the indices.

3.5 Assessment of the feature space

To assess the effectiveness of pretraining, we can look at the feature representations of the input. For SimCLR pretraining, these are the encoded input vectors $\{\boldsymbol{h}\}$ in Figure 4.3. For pretraining the sequence encoder with a pretext task, on the other hand, these are the output features $\{\boldsymbol{t}\}$ of the sequence encoder in Figure 3.4.

3.5.1 Qualitative assessment of the feature space

One way to get a qualitative understanding of the representation quality of the feature space is through a t-SNE plot (Stochastic Neighbour Embedding with a t-distribution). It works by mapping high-dimensional data into a lower-dimensional space, allowing the identification of clusters within the data [38]. A good feature space for sleep staging will have clustered the sleep stages well.

The underlying concept of t-SNE is based on the assumption that points that are close to each other in the original feature space, as measured by Euclidean distance, should also be close to each other in the lower-dimensional space. To achieve this, probability distributions need to be defined for both the feature space and the lower-dimensional space, specifying the likelihood of two points having a certain distance.

In the high dimensional feature space, a Gaussian distribution is selected. For a given point i, the probability that another point j would select point i as its neighbor is calculated as:

$$p_{j|i} = \frac{\exp\left(-\frac{1}{2\sigma_i^2}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\right)}{\sum_{k\neq i}\exp\left(-\frac{1}{2\sigma_i^2}\|\boldsymbol{x}_i - \boldsymbol{x}_k\|^2\right)}$$

Here, each σ_i is implicitly determined by choosing a perplexity value. The perplexity influences the spread of the Gaussian distribution.

In the lower-dimensional space, instead of a Gaussian distribution, a t-distribution is used. This choice is motivated by the fact that a t-distribution has heavier tails, preventing points that are far away from each other from being excessively pulled together. This addresses the crowding problem, as far away points are more likely to be considered far under a t-distribution, thus promoting clustering.

In the high-dimensional space, a t-distribution with one degree of freedom ($\nu = 1$) is chosen, which is essentially a Cauchy distribution:

$$q_{j|i} = \frac{(1 + \|\boldsymbol{z}_i - \boldsymbol{z}_j\|^2)^{-1}}{\sum_{k\neq l}(1 + \|\boldsymbol{z}_k - \boldsymbol{z}_l\|^2)^{-1}}$$

The loss function used in t-SNE is the Kullback-Leibler (KL) divergence between the two distributions:

$$\mathcal{L} = \sum_i \mathbb{KL}(P_i|Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

Although this loss function is non-convex, it can be minimized using gradient descent to find a local minimum, providing an effective way to visualize and analyze the data in the lower-dimensional space.[25]

3.5.2 Quantitative assessment of the feature space: sleep staging

To quantitatively evaluate the quality of the features, we will test the features for the downstream task of sleep staging. The sleep staging accuracy will be determined for a sequence of (small) labeled datasets of increasing size.

As a first experiment, a logistic classifier can be trained on top of the features. This involves augmenting the pretrained encoder with a linear layer and a softmax layer. The pretrained encoder, also known as the backbone, remains unchanged while only the parameters of the linear layer are trained on a small labeled dataset. This experimental setup, which we call the 'frozen backbone' experiment, is depicted in Figure 3.5.

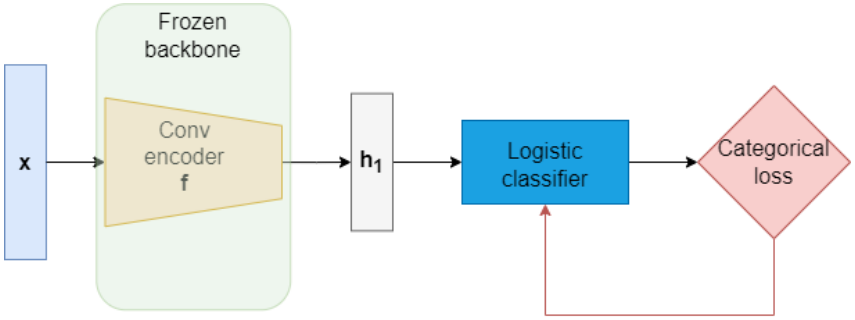


FIGURE 3.5: The frozen backbone experiment: the EEG inputs epoch $\{\boldsymbol{x}\}$ of a small labeled dataset are encoded with the pretrained encoder to their corresponding feature vectors $\{\boldsymbol{h}\}$. A classifier is then trained directly on these feature vectors.

Additionally, after the 'frozen backbone' experiment, it is possible to further refine the entire structure through a process called fine-tuning. This involves training the entire model with a small learning rate. Figure 3.6 illustrates the fine-tuning experiment. Finally, the same neural network, now consisting of a randomly initialized

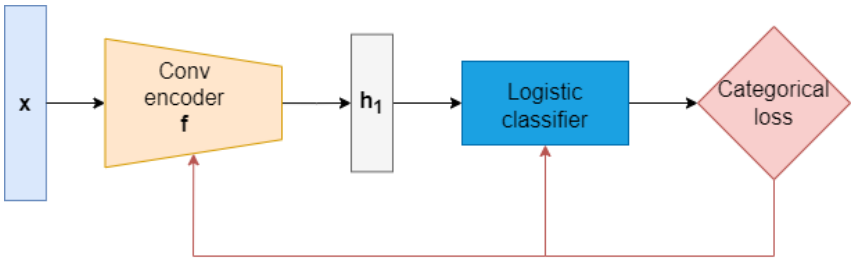


FIGURE 3.6: The finetune experiment: the pretrained encoder is now combined with the classifier from the backbone experiment. Both the encoder and the classifier are then finetuned with a small labeled dataset.

encoder and classifier, is trained on the same labeled dataset, and this is referred to as the 'fully-supervised base model' experiment. One can then assess whether the pretraining improved the accuracy or not.

The models are trained with the categorical cross-entropy loss function, which is defined as follows:

$$\mathcal{L} = -\sum_{i=1}^{5} p_i \log q_i \tag{3.2}$$

In this equation, p_i represents the true probability of obtaining label i in the dataset. On the other hand, q_i corresponds to the i'th output of the softmax layer.

The categorical cross-entropy loss function measures the dissimilarity between the predicted probabilities (q_i) and the true probabilities (p_i). By minimizing this loss, the model is encouraged to accurately classify the input samples based on their corresponding labels.

The performance can then be assessed quantitatively by calculating the sleep staging accuracy. However, there exists a better metric: Cohen's Kappa (κ) which accounts for the probability of agreement by chance.

Cohen's kappa coefficient

The Cohen's kappa coefficient (κ) is a better metric for inter-rater agreement than accuracy for categorical items because it takes into account the possibility of the agreement occurring by chance [41].

The definition of κ is:

$$\kappa \equiv \frac{p_0 - p_e}{1 - p_e} = 1 - \frac{1 - p_0}{1 - p_e} \tag{3.3}$$

where p_0 is the relative observed agreement among raters, and p_e is the hypothetical probability of chance agreement. If the raters are completely in agreement then $\kappa = 1$ and if there is no agreement among the raters other than would be expected by chance (as given by p_e), $\kappa = 0$.

Let the probability of rater i classifying an item as k be p_{ki}, and the joint probability of both rater i and j classifying an item as k be p_{k12}. The different raters are independent, such that their joint probability is the product of their marginal probabilities: $p_{kij} = p_{ki} * p_{kj}$.

Of course the real p's are not available, but they can be estimated by $p_{ki} \approx \hat{p_{ki}} = \frac{n_{ki}}{N}$, given N samples and n_{ki} samples of class k. For two raters, the total estimated probability of chance agreement is then:

$$p_e = \sum_k p_{k12} \approx \sum_k \hat{p_{k1}} \hat{p_{k2}} = \frac{1}{N^2} \sum_k n_{k1} n_{k2}$$

Landis and Koch [21] gave some guidelines for the meaning of the magnitude of κ values: $\kappa < 0$ indicates no agreement $0 < \kappa < 0.20$ as slight, $0.21 < \kappa < 0.40$ as fair, $0.41 < \kappa < 0.60$ as moderate, $0.61 < \kappa < 0.80$ as substantial, and $0.81 < \kappa < 1$ as almost perfect agreement.

3.6 Training infrastructure and limitations

This section briefly discusses the available hardware to train the neural networks, and what implementation decisions are taken to maximize the available resources. In order to optimize the resources for deep learning, it first crucial to understand the training loop.

3.6.1 The training loop

To train a neural network model f with parameters $\boldsymbol{\theta}$ that can predict outputs $\boldsymbol{y}$ given inputs $\boldsymbol{x}$, the training loop follows a specific process. The goal is to minimize a loss function $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ with respect to the parameters $\boldsymbol{\theta}$. The training process involves two key steps: the forward pass and the backward pass.

During the forward pass, a mini-batch of inputs is passed through the neural network. The network processes these inputs to generate predicted outputs $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The predicted outputs are then compared to the true outputs $\boldsymbol{y}$ using the loss function $\mathcal{L}$, which quantifies the discrepancy between the predicted and true outputs.

After the forward pass, the backward pass is performed to calculate the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ of the loss function with respect to the parameters. This gradient indicates the direction and magnitude of the change needed in the parameters to reduce the loss. Backpropagation, a technique based on the chain rule of calculus, efficiently computes these gradients by propagating the errors from the output layer to the input layer.

Modern optimizers, such as Adam and stochastic gradient descent (SGD), utilize this gradient to update the parameters iteratively. By taking small steps in the direction of the negative gradient, the optimizer aims to minimize the loss function gradually. The learning rate, a hyperparameter, determines the step size taken during parameter updates.

To leverage the capabilities of modern hardware setups, neural network training heavily relies on GPUs equipped with CUDA (Compute Unified Device Architecture). GPUs consist of thousands of cores that can perform tasks concurrently, making them highly parallel processors. Developed by NVIDIA, CUDA provides a platform that includes a suite of optimized libraries such as cuBLAS for linear algebra, cuDNN for deep neural networks, and cuFFT for Fast Fourier Transforms.

These libraries enable rapid parallel execution of mathematical operations required in neural network training. The neural network model is stored in the GPU's dedicated memory known as VRAM, allowing parallel processing of mini-batches of training examples. Both the forward pass, where inputs are processed to generate predictions, and the backward pass, where gradients are computed, are performed on the GPU. Additionally, the parameters of the neural network are updated directly on the GPU.

In order to be able to use the GPU, it is crucial to ensure that the size of the neural network and the mini-batch of training examples are compatible with the available VRAM memory. This compatibility ensures efficient utilization of the GPU's parallel processing capabilities and prevents memory overflow.

3.6.2 Available hardware

The neural network training was conducted on Google Cloud using one of their compute engines, which includes a single NVIDIA T4 GPU with a 16GB VRAM. Additionally, the compute engine offers 4 virtual CPUs and 26GB of RAM.

3.7 Conclusion

In this chapter we first decided to use a sequence-to-sequence deep learning architecture for EEG-based sleep staging, comprising an epoch encoder and a sequence encoder. The epoch encoder extracts features from raw EEG data, while the sequence encoder captures temporal information such as stage transitions. This is the architecture that most state-of-the art deep learning methods for EEG-based sleep staging use.

We further proposed a two-step pretraining strategy for this model, drawing inspiration from the remarkable achievements of GPT in NLP [8]. Specifically, we proposed to pretrain the epoch encoder with an SSL objective suitable for feature extraction, and the sequence encoder with a different SSL objective, suitable for sequences. This strategy allows the model to effectively learn both feature-level and sequence-level representations, leveraging the benefits of SSL in each aspect.

To pretrain the epoch encoder, we chose the SimCLR framework, known for its success in computer vision. It will be a good baseline to compare more exotic methods with, such as ByOL (Section 2.3.1) and SwAV (Section 2.3.1). Furthermore this framework was already successfully applied to the sleep staging task in [24].

We then proposed a new pretext task to pretrain the sequence model. It was argued that this task forces the sequence encoder to learn crucial information about the sequence. This approach enables us to leverage the inherent sequential nature of EEG data to improve the overall performance of our sleep staging model.

To evaluate the quality of our pretraining, we suggested two assessment methods. First, we recommended a qualitative assessment using t-SNE, a technique for lower-dimensional projection of the feature space, facilitating the identification of clusters. This allows us to visually analyze the effectiveness of our pretraining in capturing meaningful patterns in the EEG data. Additionally, we proposed a series of quantitative experiments to quantitatively evaluate the feature space's quality.

We are now ready to test the proposed two-stage SSL paradigm in the next chapter.

Chapter 4

Results and Discussion

4.1 Introduction

In this chapter, we will evaluate the methods outlined in Chapter 3 using the SHHS dataset (Section 3.2). This involves pretraining a sequence-to-sequence deep learning model for sleep staging in two stages. First the epoch encoder is pretrained with SimCLR to extract features at the epoch level. Then the sequence encoder is trained with the pretext task that was outlined in Section 3.4.

We will explore two neural network epoch encoder architectures: a convolutional neural network (CNN) and a transformer neural network. Both architectures have been previously employed in sleep staging for this task [35], [30]. A comparison between these models will be conducted, leading to the selection of the final epoch encoder.

Due to its parallelization ability and its demonstrated generalization performance in previous works we will use a transformer architecture for the sequence encoder. Furthermore, the transformer has already been used for this task in SleepTransformer [30], where it achieved state-of-the art results. *Future work could investigate whether other architectures could improve the results.*

In all the experiments of this chapter, the pretraining is done using the whole training dataset SHHS (Section 3.2) without using its labels. Fine-tuning the model for the downstream task of sleep staging will be done for a sequence of labeled datasets of increasing size. The result will be a graph showcasing the obtained accuracy on sleep staging as a function of the size of the labeled dataset. This is compared with regular supervised training, i.e. the same neural network, but initialized with random weights.

4.2 Pretraining the epoch encoder with SimCLR

The first experiment consists of pretraining a convolutional encoder with the SimCLR framework. The devised CNN encoder, depicted in Figure 4.1, transforms each input EEG epoch of dimension 3000 into a feature vector of dimension 184. It comprises four convolutional layers with GELU activation and batch normalization in between.

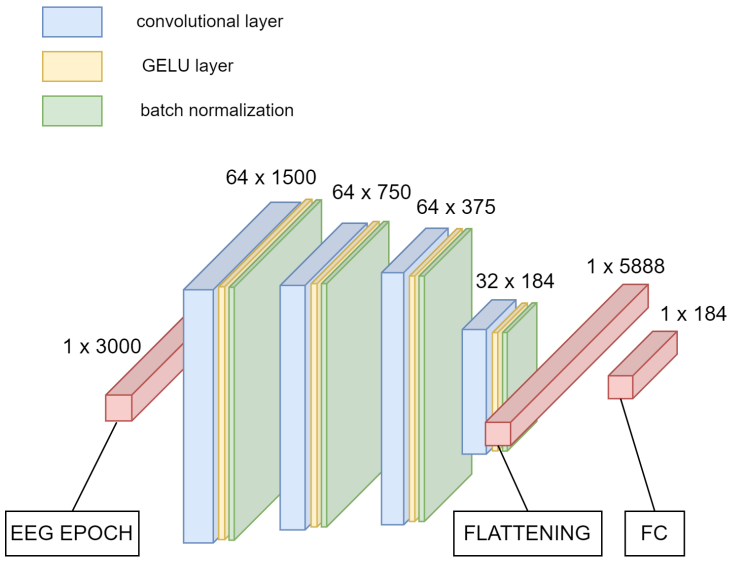


FIGURE 4.1: The proposed CNN: 4 sequential convolutional layers with stride 2, downsample the EEG input epoch $\boldsymbol{x} \in \mathbb{R}^{3000}$ to 32 feature vectors of dimension 184. Those are then flattened and reduced to one feature vector of dimension 184 through a fully connected layer.

Subsequently, a flattening operation is applied, followed by a fully connected layer, leading to a total of 1.2 million parameters (weights and biases).

In a second experiment, we will compare the CNN encoder to a transformer encoder. Specifically, we will pretrain the 'inner' transformer of SleepTransformer (refer to Section 2.2.3) with SimCLR. The input to the inner transformer of SleepTransformer was a time-frequency image, but we use the output of the CNN encoder before flattening as the input instead. The input to the transformer is thus a set of 32 feature vectors of dimension 184. The output is of the same size and is aggregated into a single feature vector of dimension 184, through a self-attention mechanism, exactly as in SleepTransformer. The final architecture consists of 2.1 million parameters and is depicted in Figure 4.2.

4.2.1 Pretraining with SimCLR

As described in Section 3.3, an augmentation module is needed for SimCLR. The parameters listed in Table 4.1 were determined through a small gridsearch. It is worth noting that more optimal hyperparameters might exist. However, conducting a hyperparameter search within the training setup would be excessively resource-intensive. Figure 3.3 illustrates some of the resulting augmentations.

As explained in Section 3.3, the contrastive loss is not directly computed in the feature space but rather in a different space known as the z-space, which is reached through a projection head. This projection head is a one-layer MLP with a hidden layer dimension of 256, and the z-space itself has a dimension of 128. Although

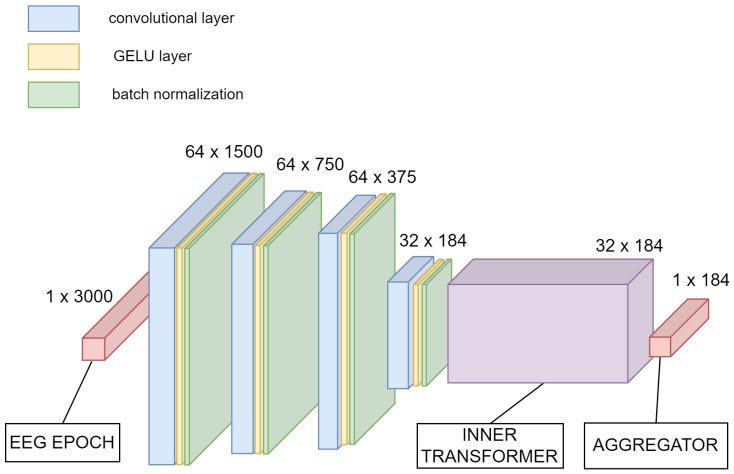


FIGURE 4.2: The proposed transformer encoder: 4 sequential convolutional layers with stride 2, downsample the EEG input epoch $\boldsymbol{x} \in \mathbb{R}^{3000}$ to 32 feature vectors of dimension 184. Those are then fed through a transformer to produce again 32 feature vectors of dimension 184. Those are then 'aggregated', using a self-attention, to one feature vector of dimension 184.

these dimension sizes were chosen somewhat arbitrarily, the authors of the original SimCLR paper demonstrate that the results are largely unaffected by the specific dimensions chosen. [10]

The batch size is a critical hyperparameter in the SimCLR framework. A larger batch size poses a more challenging task since the contrastive loss essentially transforms the pretraining task in a classification problem of identifying the one positive example among the $2N - 1$ negatives within a batch of N examples. In our case, a batch size of 512 is selected as it is the largest size that can be accommodated within our GPU's RAM (VRAM)[1]. *While it is conceivable that an even larger batch size might enhance the results, this thesis does not investigate that possibility.*

Another parameter to be chosen is the temperature parameter of the contrastive loss. Decreasing this parameter makes the contrastive learning task harder. We choose it to be $\tau = 1e - 3$ by trial and error.

For optimization, the Adam optimizer with weight decay and a cosine annealing learning rate scheduler are chosen. The specific hyperparameters for both encoders are summarized in Table 4.2.

To monitor the pretraining of the encoders using SimCLR, we employ the 'top-1 accuracy' and 'top-5 accuracy' metrics on a validation set. These metrics correspond to the percentage of correctly identified positives within a batch (top-1) and the percentage of instances that the network correctly places within the top-5 most probable results.

The training curves of both encoders can be seen in Figure 4.3. Clearly, the network has learned to solve the contrastive learning task in both cases.

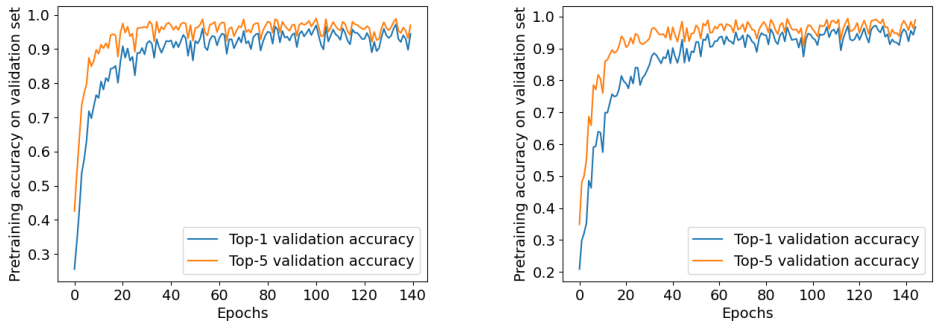
[1] Refer to Section 3.6 for details on our hardware limitations.

Augmentation	min	max
Amplitude scale	0.75	1.5
Zero masking (samples)	400	800
Time shift (samples)	-100	100
Std Gaussian noise (additive)	0	0.3
Bandpass filter width (Hz)	0	10

TABLE 4.1: The augmentation module is defined with these hyperparameters.

Hyperparameter	CNN	Transformer
τ	1e-3	1e-3
dim h-space	256	256
dim z-space	128	128
batch-size	512	512
optimizer	Adam	Adam
learning rate	3e-4	1e-4
weight decay	1e-4	1e-5

TABLE 4.2: Hyperparameters for pretraining SimCLR.



(A) CNN encoder (B) Transformer encoder

FIGURE 4.3: Top-1 and top-5 validation accuracy of pretraining with SimCLR. These metrics correspond to the percentage of correctly identified positives within a batch (top-1) and the percentage of instances that the network correctly places within the top-5 most probable results.

4.2.2 Qualitative assessment of the feature space

As mentioned in Section 3.5.1, a qualitative evaluation of the representation ability of the feature space can be achieved through the t-SNE plot. In Figure 4.4a, we present two t-SNE plots generated by the two pretrained epoch encoders.

The t-SNE plots reveal that the pretraining process has been highly successful in distinguishing between the WAKE state and deep sleep (N3) in comparison to

the other stages. However, the clustering of N1, N2, and REM stages appears to be less distinct. *To further improve the clustering of these stages, future research could explore alternative augmentation techniques for generating a more tightly-clustered feature space.*

We finally note that there appears to be minimal qualitative difference observed in the representation space between the two encoders.

(A) A t-SNE plot of the feature space, generated by the CNN encoder.

(B) A t-SNE plot of the feature space, generated by the transformer encoder.

FIGURE 4.4: These t-SNE plots are generated by encoding a random subset of the test set with the pretrained encoder and performing a t-SNE on the resulting feature vectors. The perplexity, which is a hyperparameter of the t-SNE, is set at 100.

4.2.3 Quantitative assessment of the feature space: sleep staging

The pretrained epoch encoder, which was trained on the whole unlabeled dataset, can now be transferred to the downstream task of sleep staging. The sleep staging accuracy will be determined for a sequence of (small) labeled datasets of increasing size.

The procedure is as outlined in Section 3.5.2. It involves training a logistic classifier, which is essentially a linear layer, on top of the pretrained epoch encoder, using a labeled dataset. In the first 'frozen backbone' experiment, the parameters of the pretrained encoder are kept fixed (frozen). Only the classifier is trained. In a second 'fine-tune' experiment, the logistic classifier is combined with the pretrained encoder, and the entire network is fine-tuned with a small learning rate. Finally, the same neural network, now consisting of a randomly initialized encoder and classifier, is trained on the labeled dataset, and this is referred to as the 'fully-supervised base model' experiment. All models can be compared with the accuracy or kappa score.

The hyperparameters for these three training experiments, applicable to both encoders, are provided in Table 4.3a. The experiments were repeated five times, with a different labeled dataset employed in each iteration. The resulting sleep staging performance metrics were averaged, and both the mean and standard deviation of the results are presented in Figure 4.5 and in Figure 4.7. These figures depict the performance metrics obtained from the respective experiments, enabling a comprehensive comparison between the encoders.

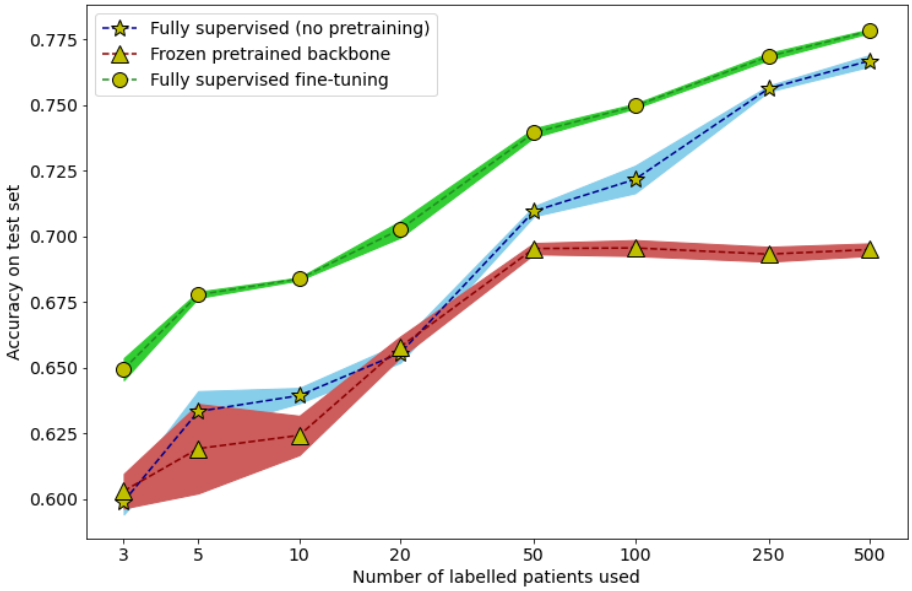
Method	learning rate	weight decay	max #training epochs
Fixed backbone	1e-4	1e-6	40
Fine-tuning	1e-5	1e-6	40
Supervised base	1e-4	1e-5	40

(A) Training hyperparameters convolutional encoder

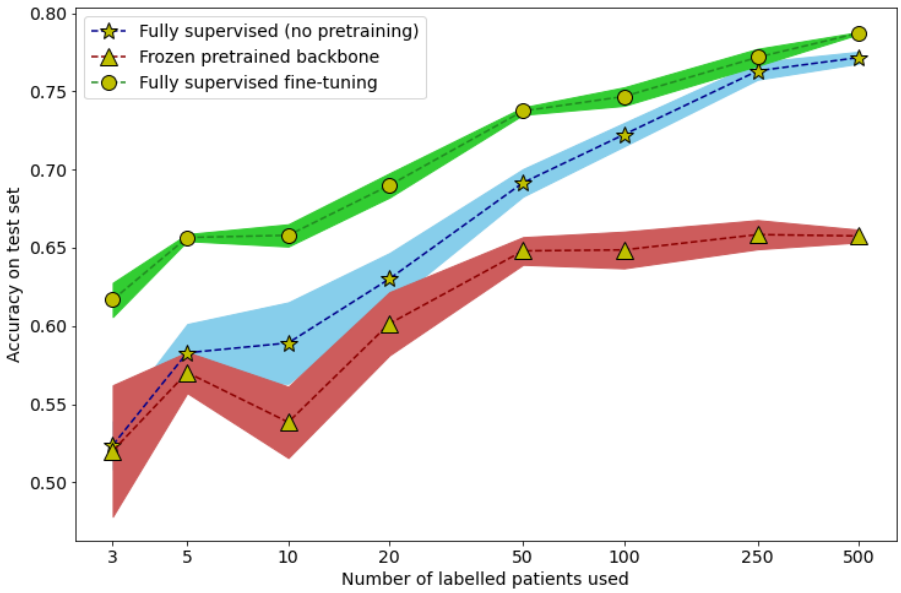
Method	learning rate	weight decay	max #training epochs
Fixed backbone	1e-4	0	80
Fine-tuning	5e-6	0	40
Supervised base	3e-5	3e-5	40

(B) Training hyperparameters transformer encoder

TABLE 4.3: The hyperparameters used in the quantitative assessment of the representation quality.

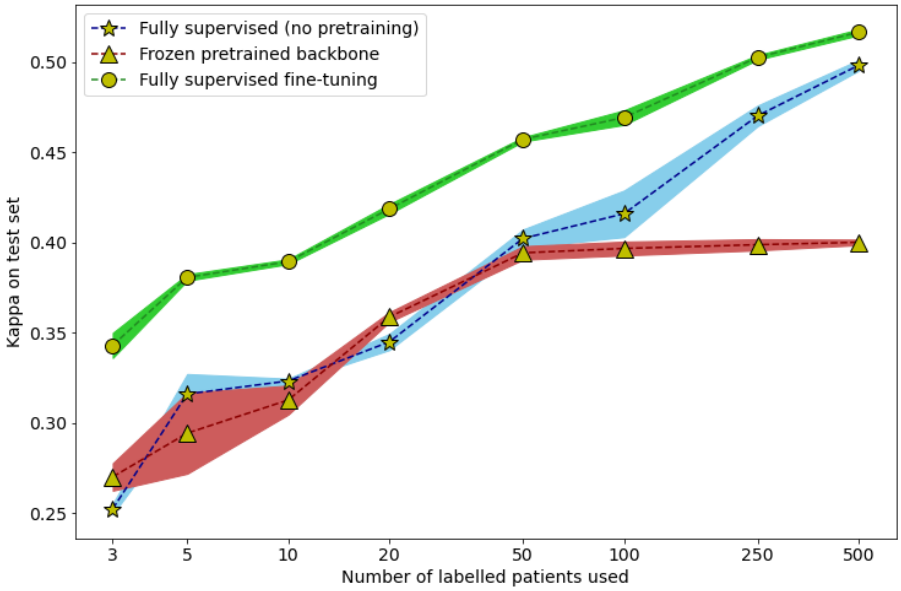


(A) CNN encoder

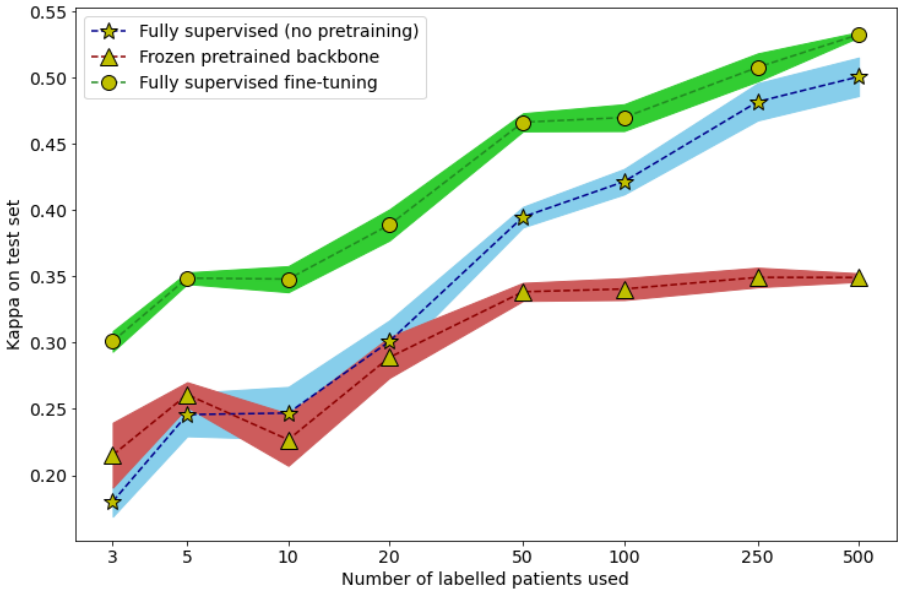


(B) Transformer encoder

FIGURE 4.5: This figure shows the sleep staging accuracy of both encoders as a function of the size of the labeled dataset. We use a fully supervised baseline model, in which the weights were randomly initialized. This is first compared with the frozen backbone experiment, in which a classifier is trained on top of the feature space. Finally we unfreeze the backbone and fine-tune both the encoder, and the classifier with the labeled data.



(A) CNN encoder



(B) Transformer encoder

FIGURE 4.6: This figure shows the Cohen's kappa score of sleep staging for both encoders as a function of the size of the labeled dataset. We use a fully supervised baseline model, in which the weights were randomly initialized. This is first compared with the frozen backbone experiment, in which a classifier is trained on top of the feature space. Finally we unfreeze the backbone and fine-tune both the encoder, and the classifier with the labeled data.

4.2.4 Discussion of the results

An initial significant finding is that the "fully supervised fine-tuning" experiment, which utilizes the pretrained epoch encoder, consistently outperforms regular fully supervised training. This is a crucial preliminary result when considering any SSL method.

Furthermore, "fully supervised fine-tuning" demonstrates superior performance over regular supervised training, particularly in scenarios with limited data. For instance, when utilizing only 5 patients (approximately 0.1% of the dataset), the fully supervised fine-tuning approach surpasses the fully supervised model by approximately 5 percentage points for both encoders. Even when using 100 patients (approximately 2% of the dataset), the performance gap reduces to slightly less than 3%, which still represents a noteworthy improvement.

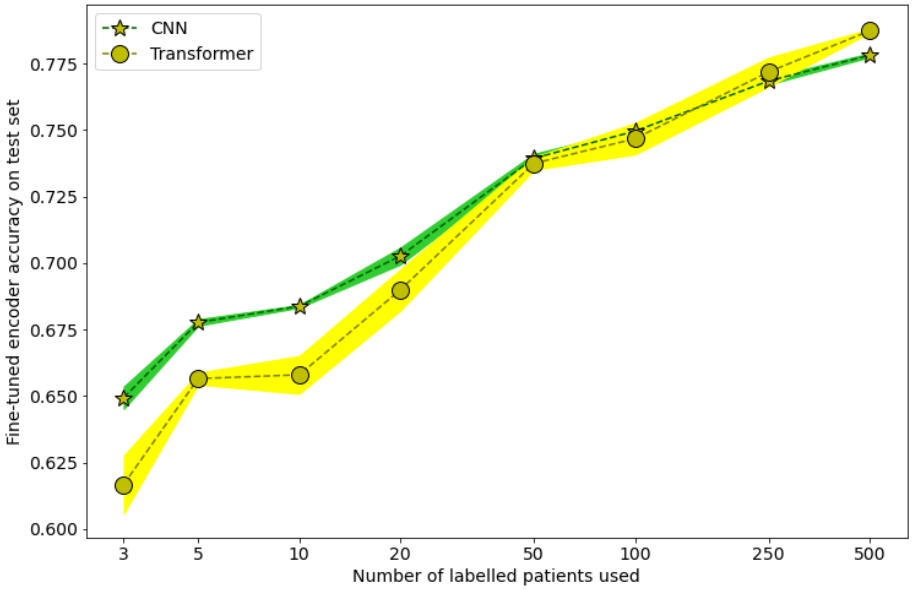
The frozen backbone experiment provides evidence that the feature space after pretraining is not yet optimally suited for sleep staging. Training a logistic classifier directly on the feature space does not outperform supervised training. However, after fine-tuning, it surpasses regular supervised training by a significant margin. This leads us to hypothesize that the feature space contains valuable information about EEG signals due to contrastive learning, but further training specifically tailored towards the sleep staging task is necessary to outperform regular supervised training.

These results were somewhat anticipated, as a similar experiment conducted in [24] yielded comparable outcomes but on a different dataset. Consequently, this study validates their findings and suggests that the method is applicable to other datasets as well.

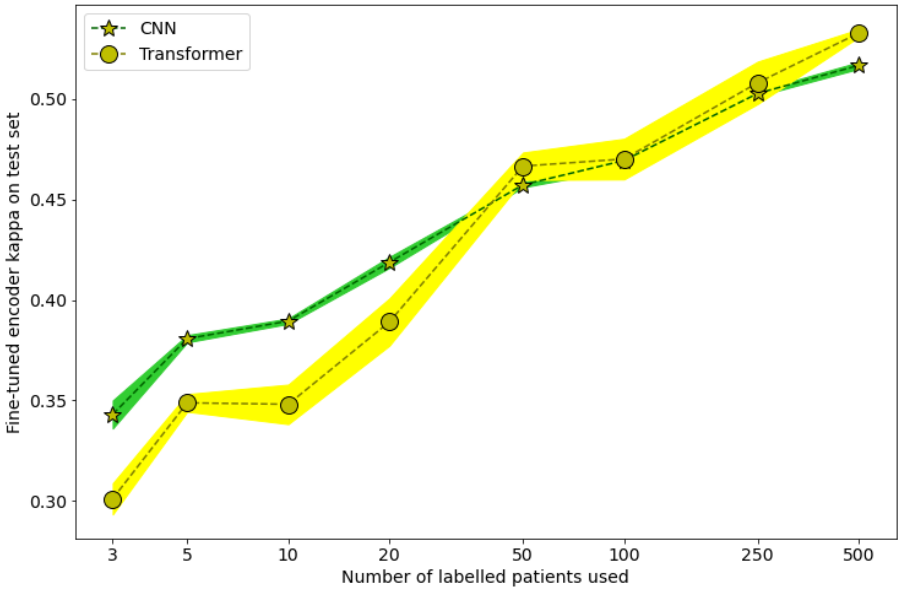
Comparison between epoch encoders

The fine-tuned accuracy of both encoders is again visualized in Figure **??**. In the initial stages with very limited data (up to around 100 patients), the CNN encoder exhibits higher accuracy. However, beyond this point, the transformer encoder surpasses the CNN encoder in terms of accuracy. This shift can likely be attributed to the fact that the transformer encoder has double the amount of parameters, which results in a higher expressiveness. While this enhanced expressiveness allows the transformer encoder to capture more intricate patterns, it also necessitates a larger amount of data to prevent overfitting and achieve optimal performance.

Since we want to use the encoder in the low-data regime, it is decided to use the CNN encoder as the feature extractor in the sequence model of the next section.



(A) Test accuracy



(B) Cohen's kappa

FIGURE 4.7: Comparison between the two encoders. We conclude that the CNN encoder performs better in the low data regime.

4.3 Experiment 2: pretraining the sequence encoder with the pretext task

We utilize the 'outer' transformer architecture from SleepTransformer (Section 2.2.3) for the sequence encoder, where it was successfully used for this task. In our setup, we made a deliberate decision to set the sequence length to $L = 6$. While longer sequences have the potential to yield better results by incorporating more temporal information, they also need a bigger sequence encoder, which needs more data to avoid overfitting. *Exploring the impact of longer sequences could be an avenue for future research.* Consequently, our sequence encoder transforms input sequences composed of six feature vectors (representing encoded sleep epochs), denoted as $\{\boldsymbol{h}\}_{i=1}^{6}$, into a sequence of six new feature vectors, denoted as $\{\boldsymbol{t}\}_{i=1}^{6}$, which are enriched with additional temporal information.

4.3.1 Pretraining the sequence encoder

To pretrain te sequence encoder, we use the pretext task that was outlined in Section 3.4. To this end, we first have to choose an epoch encoder, since the input to the pretext task are the encoded sleep epochs $\{\boldsymbol{h}\}_{i=1}^{6}$.

We decided to use the fine-tuned epoch encoder from the previous section. This is the pretrained epoch encoder that is fine-tuned on a small labeled dataset. This means that there is a different fine-tuned epoch encoder corresponding to each labeled dataset size and therefore also a different pretrained sequence encoder for each labeled dataset size. By leveraging the fine-tuned epoch encoder, we aim to maximize the utilization of all available information.

Before proceeding with the pretraining of the sequence encoders, we must select additional hyperparameters, which are outlined in Table 4.4.

Hyperparameter	Value
Hidden layer MLP	256
sequence length L	6
batch-size	64
optimizer	Adam
learning rate	1e-4
weight decay	1e-5

TABLE 4.4: Hyperparameters for training the sequence encoder with the pretext task of Section 3.4.

An example of the training and validation curves of pretraining are given in Figure 4.8. These are the curves corresponding to pretraining the sequence encoder on top of the pretrained epoch encoder, which was fine-tuned on a labeled dataset of 100 patients. The training and validation curves for the other dataset sizes are very similar and are therefore omitted.

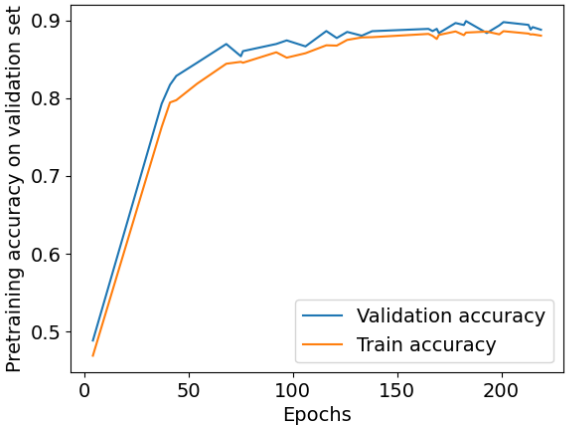


FIGURE 4.8: Training and validation accuracy of training the model to solve the pretext task.

4.3.2 Quantitative assessment of the feature space: sleep staging

To evaluate the quality of the feature space obtained through the combined pretraining of the epoch and sequence encoders, we conducted a quantitative analysis by examining the sleep staging performance across varying dataset sizes.

To ensure a fair assessment of the efficacy of pretraining the sequence encoder with a pretext task, we conducted three separate experiments for each labeled dataset size.

In the first experiment, we trained a classifier on top of both the pretrained epoch and sequence encoder. Additionally, we fine-tuned the entire structure to further optimize its performance.

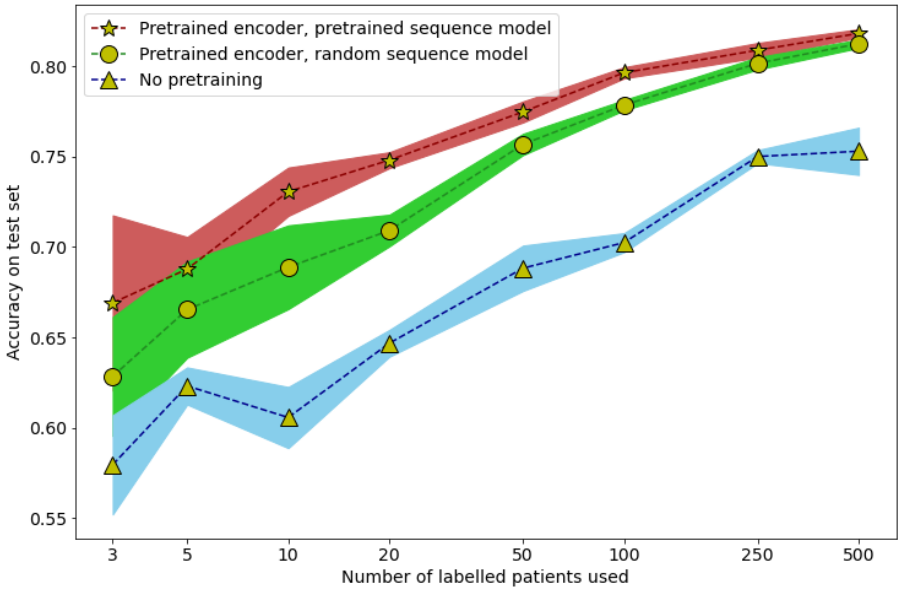
To accurately compare the impact of the pretext task, a second experiment was conducted where only the epoch encoder was pretrained, while the sequence encoder remained unpretrained, i.e. random weights. Subsequently, we fine-tuned the entire structure.

Finally, we established a fully supervised baseline model, wherein the epoch encoder, sequence sequence, and classifier were initialized with random weights. This baseline serves as a point of reference against which we can assess the effectiveness of our entire SSL paradigm.

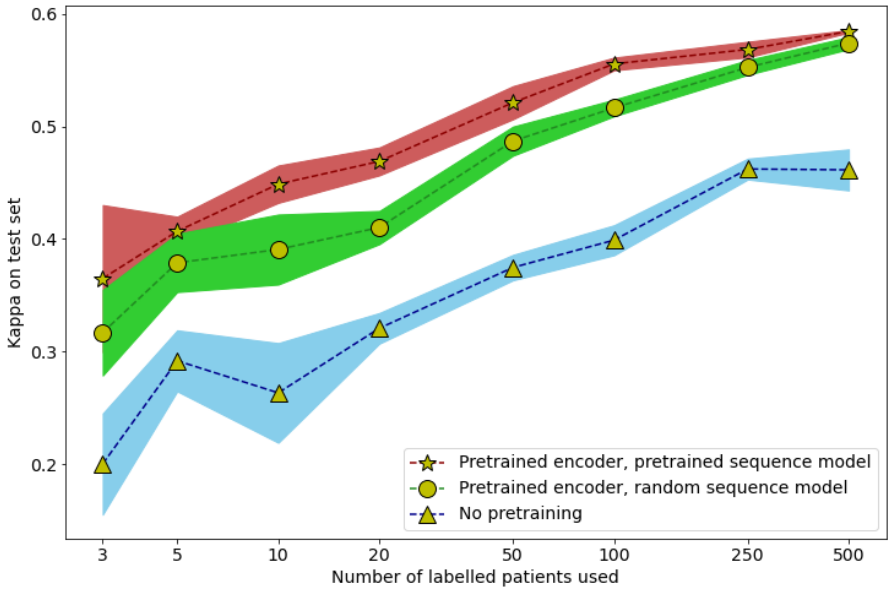
In order to conduct these experiments, we employed the hyperparameters as outlined in Table 4.5. It is important to note that each experiment was repeated three times, with a different labeled dataset employed in each iteration. The resulting sleep staging performance metrics were averaged, and both the mean and standard deviation of the results can be observed in Figure 4.9.

Method	learning rate	weight decay
Pretrained encoder and pretrained sequence	1e-5	1e-6
Pretrained encoder and random sequence	1e-4	1e-3
Supervised base: random encoder and sequence	1e-4	1e-3

TABLE 4.5: Training hyperparameters for quantitative experiments. These are chosen by trial and error. We further used early stopping.



(A) Test accuracy



(B) Cohen's kappa

FIGURE 4.9: This figure shows the sleep staging accuracy and kappa score of the full sequence-to-sequence model as a function of the size of the labeled dataset. We first show the fully pretrained model in which both the epoch and sequence encoder are pretrained. In order to assess the effectiveness of the pretext task, this is compared to a model in which only the encoder was pretrained. Finally we compare the whole SSL paradigm to a regular supervised base model, in which both the epoch and sequence encoder are initialized with random weights.

4.3.3 Discussion of the results

Again, it is first noted that the fully fine-tuned model, i.e. pretrained epoch and sequence encoder, always outperforms the regular supervised model with random weights. This is a preliminary result when considering any SSL method.

Second, it can be concluded that the pretext task does improve the overall sleep staging accuracy, especially in the low-data regime. The difference with the model consisting of a pretrained epoch encoder and sequence encoder with random weights, is 3 percentage points in the very low-data regime. However, this performance gap diminishes significantly as more patients are being added.

We can finally assess the performance of the fully fine-tuned model, where both the epoch and sequence encoder were pretrained, by comparing it with the same model initialized with random weights. This comparison unequivocally demonstrates the effectiveness of our proposed SSL paradigm.

Even in the extremely limited data scenario, where only 20 labeled patients (0.2% of the labeled dataset) are utilized, our SSL model achieves an impressive accuracy of 75%. In contrast, the fully supervised baseline falls short at 65%. As we increase the number of labeled patients to 500 (approximately 10% of the labeled dataset), our SSL paradigm continues to enhance the sleep staging accuracy by more than 7 percentage points. *Finally it is noted that it would be interesting to extend the graph to the right. Future work could investigate this.*

4.4 Conclusion

In this chapter, we evaluated the methods outlined in the previous chapter using the SHHS dataset. We pretrained a sequence-to-sequence deep learning model for sleep staging in two stages. First, we pretrained an epoch encoder with SimCLR to extract features at the epoch level, and then we pretrained a sequence encoder with the pretext task outlined in Section 3.4. We explored two neural network epoch encoder architectures: a convolutional neural network (CNN) and a transformer neural network. We further decided to only explore a transformer architecture for the sequence encoder.

After pretraining the epoch encoders with SimCLR, we performed a qualitative and quantitative assessment of the learned feature space. The t-SNE plots showed that the pretraining process successfully distinguished between the WAKE state and deep sleep (N3) but had less distinct clustering for N1, N2, and REM stages.

For the quantitative assessment, we transferred the pretrained epoch encoder to the sleep staging task using labeled datasets of increasing size. We compared the performance of the frozen backbone, fine-tuning, and fully-supervised base models. Pretraining with SimCLR led to a performance gain of 5 percentage points over regular supervised training in the low data regime (<0.1% of labeled the dataset). Furthermore, this trend continues, although less significantly, as a larger dataset is being used. Finally, the results showed that both encoders achieved comparable performance, with the CNN encoder slightly outperforming the transformer encoder in the low data regime. It was therefore decided to continue with the CNN encoder.

The second phase of our SSL paradigm involved pretraining the sequence encoder with a pretext task. To evaluate the effectiveness of this pretraining, we again analyzed the sleep staging accuracy across labeled datasets of increasing sizes. Firstly, we compared the sleep staging accuracy between a pretrained sequence encoder on top of a pretrained epoch encoder, with a randomly initialized sequence encoder on top of the same epoch pretrained encoder. This comparison allowed us to assess the impact of the pretext task, and our findings indicated that the pretext task indeed improved the sleep staging accuracy. Notably, the most significant improvement was observed in the low data regime, with an increase of over 3 percentage points. However, the performance gap between the models diminished rapidly as the size of the labeled dataset increased.

Finally we assessed the performance of the fully fine-tuned model, where both the epoch and sequence encoder were pretrained, by comparing it with the same model initialized with random weights. This comparison unequivocally demonstrated the effectiveness of our proposed SSL paradigm. Even in the extremely limited data scenario, where 0.2% of the labeled dataset was utilized, our SSL model achieved an impressive accuracy of 75%. In contrast, the fully supervised baseline fell short at 65%. As the size of the labeled dataset grew to 10 %, our SSL paradigm continued to enhance the sleep staging accuracy by more than 7 percentage points.

In conclusion, our study confirms the validity of the proposed SSL paradigm for sleep staging, showcasing its effectiveness in enhancing sleep staging accuracy across different labeled dataset sizes.

Chapter 5

Conclusion

The goal of this thesis was to reduce the amount of *labeled* data necessary to train a machine learning algorithm for EEG-based sleep staging. To achieve this, we leveraged unlabeled data using self-supervised learning (SSL) methods.

The current state-of-the-art deep learning architectures for sleep staging are sequence-to-sequence models, consisting of an epoch encoder, a sequence encoder, and a classifier. We proposed a two-stage SSL paradigm for pretraining such an architecture. This paradigm involves first pretraining the encoder with an SSL objective appropriate for feature extraction, and then pretraining the sequence model with a different SSL objective, appropriate for sequences.

To pretrain the epoch encoder, we adapted the SimCLR framework [10], originally designed for CV, to the domain of EEG. Specifically, we evaluated the efficacy of two epoch encoder architectures: a Convolutional Neural Network (CNN) and a Transformer model.

We assessed the feature space of both epoch encoders after pretraining and both exhibited significant improvements over the conventional supervised training approach, particularly in scenarios with limited available data. Notably, when utilizing less than 1% of the labeled dataset, the pretraining phase yielded a sleep staging accuracy enhancement of more than 5 percentage points.

The second phase of our SSL paradigm involved pretraining the sequence model with a pretext task. Our findings indicated that the pretext task itself indeed improved the sleep staging accuracy. Notably, the most significant improvement was observed in the low data regime, with an increase of over 3 percentage points in sleep staging accuracy. However, the performance gap between the models diminished rapidly as the size of the labeled dataset increased. Overall, the improvement due to SSL pretraining was smaller in this phase than in the first.

Finally we conducted a comparison between the fully pretrained model, in which both the encoder and the sequence model were pretrained, with the unpretrained model, in which the parameters of both encoders are randomly initialized. This comparison clearly showed the validity of our proposed two-stage SSL paradigm. It achieves a 10% improvement in the low-data regime (using approximately 1% of the labeled dataset) and a 7% improvement when using 10% of the labeled dataset.

To summarize, our research provides compelling evidence for the efficacy of the proposed SSL paradigm in sleep staging. Our study demonstrates the effectiveness of this approach in improving sleep staging accuracy, particularly when working with limited datasets.

5.0.1 Future work and limitations

There are several avenues for future work that researchers could explore..

Firstly, it is worth investigating alternative architectures for both the epoch and sequence encoder, as they may potentially yield improved results. Specifically, emphasizing larger models could be beneficial, as there is a hypothesis suggesting that SSL methods particularly benefit from models with a larger parameter count [5]. Unfortunately, due to limited computational resources, our study was unable to explore larger models. Thus, future investigations can delve into the impact of model size on SSL performance.

Moreover, our ability to conduct extensive hyperparameter optimization was constraint by the available computational resources. SSL approaches often involve multiple hyperparameters that significantly influence the model's performance. For example, in the SimCLR framework, important hyperparameters include the choice and strength of augmentations, the temperature parameter, and the batch size [10]. Additionally, our study did not explore the effect of sequence length, which is a crucial parameter for the pretext task.

Furthermore, future research could delve into the potential of other SSL methods, such as BYOL [14], SWAV [9], CPC [**?**], and more, to enhance results.

Finally, future work could validate our findings on other datasets.

Bibliography

[1] Neural networks from scratch. https://victorzhou.com/series/neural-networks-from-scratch/. Accessed on 20-05-2023.

[2] Stages of sleep. https://www.coursehero.com/study-guides/wsu-sandbox/stages-of-sleep/. Accessed: 2023-05-5.

[3] Yann lecun: Ssl, the dark matter of ai. https://ai.facebook.com/blog/self-supervised-learning-the-dark-matter-of-intelligence/. Accessed: 2023-05-19.

[4] A. Baevski, W.-N. Hsu, Q. Xu, A. Babu, J. Gu, and M. Auli. Data2vec: A general framework for self-supervised learning in speech, vision and language. In *International Conference on Machine Learning*, pages 1298–1312. PMLR, 2022.

[5] R. Balestriero, M. Ibrahim, V. Sobal, A. Morcos, S. Shekhar, T. Goldstein, F. Bordes, A. Bardes, G. Mialon, Y. Tian, et al. A cookbook of self-supervised learning. *arXiv preprint arXiv:2304.12210*, 2023.

[6] H. J. Banville, I. Albuquerque, A. Hyvärinen, G. Moffat, D. Engemann, and A. Gramfort. Self-supervised representation learning from electroencephalography signals. *CoRR*, abs/1911.05419, 2019.

[7] R. B. Berry, R. Brooks, C. E. Gamaldo, S. M. Harding, C. Marcus, B. V. Vaughn, et al. The aasm manual for the scoring of sleep and associated events. *Rules, Terminology and Technical Specifications, Darien, Illinois, American Academy of Sleep Medicine*, 176:2012, 2012.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[9] M. Caron, I. Misra, J. Mairal, P. Goyal, P. Bojanowski, and A. Joulin. Unsupervised learning of visual features by contrasting cluster assignments. *CoRR*, abs/2006.09882, 2020.

[10] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton. A simple framework for contrastive learning of visual representations. *CoRR*, abs/2002.05709, 2020.

[11] X. Chen and K. He. Exploring simple siamese representation learning. *CoRR*, abs/2011.10566, 2020.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] J. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. Á. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko. Bootstrap your own latent: A new approach to self-supervised learning. *CoRR*, abs/2006.07733, 2020.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[16] D. Hendrycks, M. Mazeika, S. Kadavath, and D. Song. Using self-supervised learning can improve model robustness and uncertainty. *Advances in neural information processing systems*, 32, 2019.

[17] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[18] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[19] K. Kontras, C. Chatzichristos, H. Phan, J. Suykens, and M. D. Vos. Core-sleep: A multimodal fusion framework for time series robust to imperfect modalities, 2023.

[20] R. Krishnan, P. Rajpurkar, and E. J. Topol. Self-supervised learning in medicine and healthcare. *Nature Biomedical Engineering*, pages 1–7, 2022.

[21] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[22] Y. J. Lee, J. Y. Lee, J. H. Cho, and J. H. Choi. Interrater reliability of sleep stage scoring: a meta-analysis. *Journal of Clinical Sleep Medicine*, 18(1):193–202, 2022.

[23] T. Mitchell and M. L. McGraw-Hill. Edition, 1997.

[24] M. N. Mohsenvand, M. R. Izadi, and P. Maes. Contrastive representation learning for electroencephalogram classification. In *Machine Learning for Health*, pages 238–253. PMLR, 2020.

[25] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[26] M. Noroozi and P. Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. *CoRR*, abs/1603.09246, 2016.

[27] M. Noroozi, A. Vinjimoor, P. Favaro, and H. Pirsiavash. Boosting self-supervised learning via knowledge transfer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9359–9367, 2018.

[28] S. R. Pandi-Perumal, D. Spence, and A. Bahammam. *Polysomnography: An Overview*, pages 29–42. 07 2014.

[29] S. K. e. a. Patel AK, Reddy V. Physiology, sleep stages. In: StatPearls [Internet]. Treasure Island (FL): StatPearls Publishing; 2023 Jan-. Available from: https://www.ncbi.nlm.nih.gov/books/NBK526132/. Accessed on 25-02-2023.

[30] H. Phan, K. B. Mikkelsen, O. Y. Chén, P. Koch, A. Mertins, and M. D. Vos. Sleeptransformer: Automatic sleep staging with interpretability and uncertainty quantification. *CoRR*, abs/2105.11043, 2021.

[31] S. D. Puthankattil, P. Joseph, U. R. Acharya, and C. Lim. Eeg signal analysis: a survey. *Journal of medical systems*, 34:195–212, 04 2010.

[32] S. F. Quan, B. V. Howard, C. Iber, J. P. Kiley, F. J. Nieto, G. T. O'Connor, D. M. Rapoport, S. Redline, J. Robbins, J. M. Samet, et al. The sleep heart health study: design, rationale, and methods. *Sleep*, 20(12):1077–1085, 1997.

[33] R. Shriram, M. Sundhararajan, and N. Daimiwal. Eeg based cognitive workload assessment for maximum efficiency. *IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)*, 08 2012.

[34] A. Sors, S. Bonnet, S. Mirek, L. Vercueil, and J.-F. Payen. A convolutional neural network for sleep stage scoring from raw single-channel eeg. *Biomedical Signal Processing and Control*, 42:107–114, 2018.

[35] A. Supratak, H. Dong, C. Wu, and Y. Guo. Deepsleepnet: a model for automatic sleep stage scoring based on raw single-channel eeg. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, PP, 03 2017.

[36] A. Supratak and Y. Guo. Tinysleepnet: An efficient deep learning model for sleep stage scoring based on raw single-channel eeg. In *2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*, pages 641–644, 2020.

[37] A. van den Oord, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018.

[38] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[39] V. N. Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[41] Wikipedia contributors. Cohen's kappa — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Cohen%27s_kappa&oldid=1130024730, 2022. [Online; accessed 18-May-2023].

[42] Wikipedia contributors. Epilepsy — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Epilepsy&oldid=1155123133, 2023. [Online; accessed 25-May-2023].

[43] E. A. Wolpert. A manual of standardized terminology, techniques and scoring system for sleep stages of human subjects. *Archives of General Psychiatry*, 20(2):246–247, 1969.

[44] T. Zhang and B. Wang. Contrastive learning for sleep staging based on inter subject correlation. *arXiv preprint arXiv:2305.03178*, 2023.

[45] S. Zhao, F. Long, X. Wei, X. Ni, H. Wang, and B. Wei. Evaluation of a single-channel eeg-based sleep staging algorithm. *International Journal of Environmental Research and Public Health*, 19(5), 2022.