UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA
DEPARTAMENTO DE MATEMÁTICA APLICADA

ELDIANE BORGES DOS SANTOS DURÃES

Aprendizado profundo auto-supervisionado em séries temporais: uma aplicação na análise de sinais de eletroencefalogramas

Campinas
24/06/2024

ELDIANE BORGES DOS SANTOS DURÃES

Aprendizado profundo auto-supervisionado em séries temporais: uma aplicação na análise de sinais de eletroencefalogramas*

Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos para obtenção de créditos na disciplina Projeto Supervisionado, sob a orientação do(a) Prof. Dr. João Batista Florindo.

*Este trabalho foi financiado pelo CNPq.

Resumo

O presente projeto trata da classificação automatizada dos estágios do sono no corpo humano através de sinais de eletroencelafograma (EEG) coletados durante o sono com uso de aprendizado profundo, tarefa esta importante para a identificação de doenças relacionadas ao sono, mentais, cardíacas ou neurológicas. A referência principal deste trabalho, Kumar et al. [2022], apresenta o mulEEG, um método de aprendizado auto-supervisionado que usa múltiplas visualizações dos dados para aprendizado não-supervisionado de representações efetivas dos sinais de EEG para posterior classificação, de modo a aproveitar a complementariedade das informações. A princípio, buscou-se entender a estrutura do mulEEG, bem como sua implementação, permitindo o pré-processamento dos dados e treinamento do modelo conforme os códigos disponibilizados em Kumar et al. [2022], processo este que foi longo devido à complexidade do modelo, mas que trouxe grandes aprendizados. Devido ao alto custo computacional requerido para treinar o mulEEG, considerou-se simplificar o algoritmo, substituindo uma ResNet-50 com convoluções unidimensionais por uma ResNet-18 em determinada etapa do modelo, de modo a estudar esta importante estrutura de aprendizado profundo. Contudo, a ResNet-50 apresentou desempenho ainda superior, compensando o longo treinamento. Ademais, uma vez que as representações dos sinais de EEG foram aprendidas, o foco voltou-se à tarefa de classificação, que não havia sido explorada na referência principal. Para tanto, utilizou-se como classificador o algoritmo SVM (Support Vector Machine), que proporcionou melhores métricas que aquelas obtidas na avaliação linear do mulEEG no artigo supracitado. Dessa forma, as representações obtidas pelo mulEEG mostraram-se efetivas para a etapa de classificação, sendo que o maior desafio do projeto foi lidar com a limitação de recursos ao trabalhar com um modelo complexo e com grande volume de dados.

Abstract

This project encloses automated sleep-stage classification in the human body by means of electroencephalogram (EEG) signals collected during sleep using deep learning, an important task to identify sleep-related diseases, whether they are mental, cardiac or neurological. This project main reference, Kumar et al. [2022], presents the mulEEG, a multi-view self-supervised method for unsupervised EEG representation learning, in order to effectively utilize the complementary information of the EEG multi-view signals. Firstly, the mulEEG structure had been analysed, such as its implementation, allowing the data pre-processing and model training to be done according to the available code on Kumar et al. [2022], which had been a long process due to the complexity of the model, but it brought vast knowledge. Due to the high computacional cost required to train the mulEEG, it was considered simplifying the algorithm, replacing a ResNet-50 with 1D-convolutions by a ResNet-18 on a specific step of the model, in order to study this important deep learning structure. However, the ResNet-50 presented superior performance, compensating the long training. Furthermore, once the EEG representations were learned, the focus became the classification task, which had not been explored on the main reference. For that, it was used the SVM (Support Vector Machine) algorithm as the classifier, which provided better metrics than the ones obtained by the mulEEG linear evaluation on the aforementioned article. Hence, the representations obtained by mulEEG were effective to the classification step, whereas the biggest challenge of this project was dealing with the resources limitation when working with a complex model and huge data volume.

Conteúdo

1 Introdução

Ter a quantidade e qualidade suficientes de sono no momento correto é essencial para o bem estar, conforme Kumar et al. [2022]. De acordo com Patel et al. [2024], no corpo humano o sono é dividido em ciclos que possuem duas fases. A primeira é o movimento ocular rápido (REM), enquanto a segunda é o movimento ocular não rápido (NREM), que ainda é dividida nos estágios N1, N2 e N3. Cada fase é classificada conforme variações no tônus muscular, nos padrões de ondas cerebrais e nos movimentos oculares. Contudo, o corpo humano passa por cada ciclo de 4 a 6 vezes por noite, cada um com duração de aproximadamente 90 minutos.

No entanto, a qualidade do sono e o tempo gasto em cada estágio podem ser alterados por transtornos relacionados ao sono ou mentais, como apneia obstrutiva do sono, depressão, esquizofrenia e demência, além de lesões cerebrais traumáticas, medicamentos e distúrbios do ritmo circadiano. Em razão disso, identificar as fases do sono é uma tarefa importante na identificação de tais doenças, de modo que automatizar essa classificação pode ser muito proveitoso para a avaliação clínica do sono.

Nesse sentido, o mulEEG, modelo apresentado na referência principal do presente projeto, Kumar et al. [2022], tem como objetivo a automatização da classificação das fases do sono a partir de sinais de eletroencefalograma (EEG) coletados neste período. Para isso, realiza uma tarefa de pretexto, que trata de aprender representações efetivas desses sinais de EEG a partir dos dados sem o rótulo. Isso se justifica na hipótese de que estas gravações de sinais de EEG são custosas de serem feitas, além de inconfiáveis, já que não há concordância sobre como essa coleta deve ser feita, podendo levar a um treinamento tendencioso do modelo. Em resumo, a tarefa de pretexto é feita com um método de aprendizado auto-supervisionado, que será o de aprendizagem contrastiva, a partir de múltiplas visualizações dos sinais de EEG, as quais são obtidas via data augmentation, de modo a aproveitar a complementaridade das informações. Uma vez aprendidas as representações efetivas dos dados não rotulados, a tarefa de classificação pode ser executada sobre elas.

Dessa forma, a princípio o objetivo do presente projeto é entender a construção, na prática, do mulEEG, debugando e executando os códigos disponibilizados em Ku-

mar et al. [2022]. Assim, espera-se adquirir experiência com modelos complexos de aprendizado profundo e manipulação de dados, lidando ainda com métodos e conceitos de inteligência artificial estudados anteriormente, como a ResNet. Feito isso, o próximo passo natural é implementar modificações ao modelo com o intuito de acelerar seu treinamento, bem como explorar a tarefa de classificação após o aprendizado da representação dos dados.

2 Entendendo o mulEEG

Na presente sessão será apresentado o algoritmo mulEEG desenvolvido no artigo Kumar et al. [2022], referência principal deste trabalho. Trata-se de um método auto-supervisionado de múltiplas visões para aprendizado não supervisionado de representações de sinais de eletroencefalograma (EEG) coletados durante o sono, já que os modelos existentes performam mal. O foco do método é aproveitar a complementaridade das informações de cada visão para melhorar a aprendizagem, introduzindo a perda diversa para este fim, de modo a superar o treinamento supervisionado. O objetivo aqui é entender o que significam estes termos.

2.1 O que significam esses termos?

A aprendizagem auto-supervisionada (self-supervised learning, SSL) é um método de aprendizagem não supervisionada utilizado dentro de um problema de aprendizagem supervisionada, no qual o modelo prevê parte da entrada a partir de outras partes dela, o que permite aprender representações da mesma. Nesse sentido, a SSL de multi-visões treina em conjunto todas as diferentes visões dos dados para que se influenciem positivamente na aprendizagem dessas representações. Note que, para extrair multi-visões dos sinais de EEG é nessário aplicar estratégias de ampliação de dados (data augmentation), modificando os dados originais.

Portanto, o objetivo do mulEEG é aprender representações efetivas dos sinais de EEG a partir de múltiplas visões treinando-as em conjunto em um método auto-supervisionado, de modo que as informações se complementem e se influenciem positivamente durante o treinamento, e esta é a chamada tarefa de pretexto. A técnica

a ser utilizada para este fim é a aprendizagem contrastiva, que consiste em aprender características gerais do conjunto de dados sem olhar para os rótulos, identificando similaridades e diferenças entre eles antes mesmo de ter uma tarefa como classificação ou segmentação, de acordo com Tiu [2021].

Uma vez aprendidas essas representações é possível trabalhar no problema de aprendizagem supervisionada, classificando-as de acordo com os estágios do sono (Wake, REM, N1, N2 e N3, conforme orientações do American Academy of Sleep Medicine). Esta é a chamada tarefa principal.

2.2 Descrição do modelo mulEEG e avaliação

A princípio, vamos descrever os métodos utilizados para data augmentation, parte essencial do modelo, já que os métodos de aprendizagem contrastivos são fortemente influenciados pela ampliação de dados utilizada. Na primeira família de ampliações, T_1, usou-se jittering, no qual um ruído uniforme aleatório é adicionado aos sinais de EEG, juntamente com máscaras, onde sinais são mascarados aleatoriamente. Na segunda família de ampliações, T_2, aplicou-se flipping, em que sinais de EEG foram invertidos horizontalmente aleatoriamente, e escalonamento, onde os sinais de EEG são escalados com ruído Gaussiano.

Para cada segmento de sinais de EEG, obtivemos conforme descrito anteriormente duas séries temporais por ampliação, denotadas como t_1, da família de ampliações T_1, e t_2, da T_2. Essas saídas, denominadas nos códigos como forte e fraca, respectivamente, serão convertidas ainda nos seus respectivos espectrogramas, s_1 e s_2. Daí, todas essas ampliações são passadas para os seus respectivos encoders, sendo E_t para as séries temporais e E_s para os espectrogramas, de modo a extrair suas representações latentes, obtendo as features de tempo e de espectrograma.

Temos ainda a estrutura nomeada projection head, que nada mais é do que uma rede neural totalmente conectada, que irá mapear as representações para o espaço em que a função de perda contrastiva será aplicada, obtendo $\mathbf{z_i}$ e $\mathbf{z_j}$. Daí, no método mulEEG, as features de tempo, espectrograma e a concatenação de ambas são passadas para suas respectivas projection heads, que totalizam seis, três para cada família de ampliações (f, g e h).

Para ter flexibilidade de otimizar cada feature durante o treinamento, usa-se três perdas contrastivas (uma para cada espaço): L_{TT} referente às features de séries temporais, L_{SS} referente às features do espectrograma e L_{FF} referente às features concatenadas. Foi usada uma variante da perda contrastiva chamada NT-Xent, que maximiza a similaridade entre duas visões do mesmo exemplo enquanto minimiza a similaridade com visões dos outros exemplos, apresentada a seguir:

$$l(i,j) = -log\left(\frac{exp(cosine(\mathbf{z_i}, \mathbf{z_j})/\tau)}{\sum_{k=1}^{2N} \mathbb{I}_{[k\neq i]} exp(cosine(\mathbf{z_i}, \mathbf{z_k})/\tau)}\right), \tag{1}$$

$$L(\mathbf{z_i}, \mathbf{z_j}) = \frac{1}{2N}\sum_{k=1}^{N} l(2k-1, 2k) + l(2k, 2k-1), \tag{2}$$

onde N é o tamanho do batch, lembrando que cada exemplo tem duas ampliações, τ é o parâmetro de temperatura e a similaridade é medida pelo cosseno.

Contudo, o método ainda introduz a função de perda diversa L_D, que força as informações de ambas as visões a se complementarem ao deixar a feature concatenada de lado, uma vez que a L_{FF} tende a maximizar a informação mútua entre as visões e ignorar a complementaridade. Dessa forma, a perda diversa é aplicada apenas nas features de tempo e do espectrograma, deixando a concatenada de lado, isto é, em $z_k = [\mathbf{z_i^t}, \mathbf{z_j^t}, \mathbf{z_i^s}, \mathbf{z_j^s}]$, de modo que z_k representa a k-ésima amostra. Por fim, há ainda a função de perda total L_{tot}, estando as equações e representação visual do mulEEG dispostos a seguir. Dessa forma, temos a construção do modelo para a tarefa de pretexto, isto é, aprender representações efetivas dos sinais de EEG.

$$l_d(z_k, a, b) = -log\left(\frac{exp(cosine(z_k[a], z_k[b])/\tau_d)}{\sum_{i=1}^{4} \mathbb{I}_{[i\neq a]} exp(cosine(z_k[a], z_k[i])/\tau_d)}\right), \tag{3}$$

$$L_D = \frac{1}{4N}\sum_{k=1}^{N} l_d(z_k, 1, 2) + l_d(z_k, 2, 1) + l_d(z_k, 3, 4) + l_d(z_k, 4, 3), \tag{4}$$

$$L_{tot} = \lambda_1(L_{TT} + L_{FF} + L_{SS}) + \lambda_2 L_D \tag{5}$$

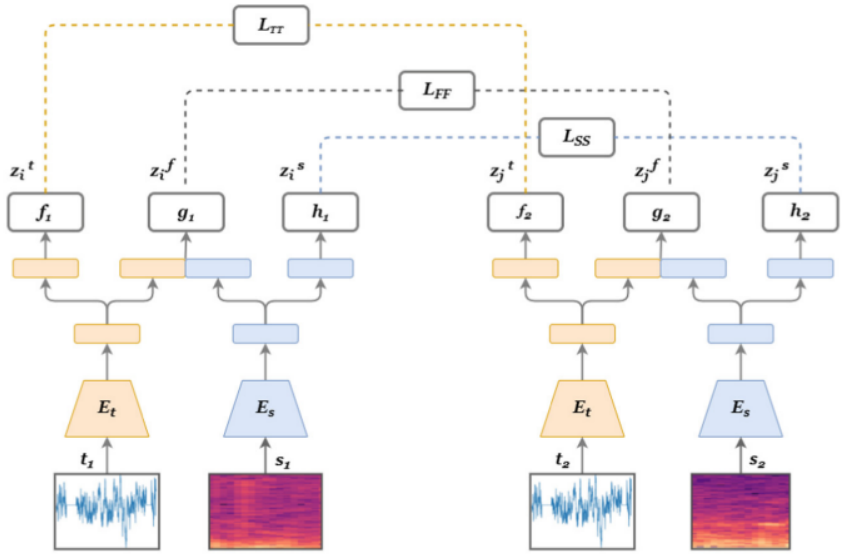


Figura 1: Visão geral da arquitetura do mulEEG. Fonte: Kumar et al. [2022].

Quanto à avaliação do método, os encoders, que foram pré-treinados usando o grupo de pretexto não rotulado, passam por uma avaliação linear. Nela, um classificador linear é anexado no topo do encoder pré-treinado e apenas ele é treinado, de modo que as métricas relativas à essa classificação dos estágios do sono são avaliadas no grupo de teste. Como será usada apenas com a base de dados SleepEDF, que será detalhada adiante, realizou-se uma avaliação de 5-fold nos grupos de treino e teste. As métricas utilizadas foram kappa de Cohen, que está entre 0, quando não há concordância além do acaso, e 1, se os avaliadores estiverem de acordo, acurácia e macro F1-score (MF1), que considera as medidas de precision e recall. Assim, temos exposto a tarefa principal de classificação.

3 Executando o mulEEG

3.1 Pré-processamento dos dados

O primeiro passo para reproduzir os resultados do artigo de Kumar et al. [2022] foi trabalhar na base de dados de sinais de eletroencefalograma coletados durante o sono a ser utilizada no treinamento. Investigando as referências do artigo, notou-se que a base de dados SHHS é um pouco mais burocrática para ser obtida, além de ter um volume maior de dados. Já a Sleep-EDF, em Kemp [2013], pode ser

facilmente acessada, sendo que todas as informações referentes à ela estão detalhadas. Assim, decidiu-se trabalhar apenas com a base de dados Sleep-EDF, sem transferência de aprendizado da outra base.

Em Sleep-EDF temos dados sobre uma coleção de 78 pessoas, sendo que para cada uma há gravação de uma ou duas noites completas, em um total de 153 gravações de noites completas que serão utilizadas neste trabalho, dentre outros tipos de dados. Os arquivos referentes à estas gravações estão na pasta *sleep-cassette*, advindos de um estudo de 1987 à 1991 sobre os efeitos da idade no sono em caucasianos saudáveis com idades entre 25 e 101 anos, sem qualquer medicação relacionada ao sono, conforme Kemp [2013].

Os 153 arquivos do tipo *PSG.edf* são os registros polissonográficos do sono de uma noite inteira a partir de um único canal de EEG (da localização dos eletrodos Fpz-Cz), com sinais amostrados em 100 Hz. Já os arquivos *Hypnogram.edf* contém anotações dos padrões do sono (hipnogramas) correspondentes, sendo eles W (Wake), R (REM), 1 (N1), 2 (N2), 3 (N3), 4 (N3), M (tempo de movimento) e ? (não pontuado), todos registrados manualmente por técnicos bem treinados. Os arquivos são nomeados no formato *SC4ssNEO-PSG.edf*, onde *ss* é o número do sujeito e *N* é a noite.

Uma vez baixada a base de dados da pasta *sleep-cassette* de SleepEDF, os esforços concentraram-se em entender os códigos disponibilizados por Kumar et al. [2022] referentes ao pré-processamento desses dados, isto é, os arquivos da pasta "./preprocessing/sleepedf/", em especial o *preprocess_sleepedf.py*. Ali, foi ficando mais claro como organizar os diretórios, alocando a base de dados na pasta "./SLEEP_data/physionet-sleep-data/", por exemplo. Ademais, outros diretórios são criados ao longo do código:

- "./SLEEP_data/numpy_saves/": arquivos em formato numpy;
- "./SLEEP_data/numpy_subjects/": arquivos numpy separados por pessoa, já que cada pessoa pode ter mais de uma gravação;
- "./SLEEP_data/data/": arquivos da pasta anterior separados nos conjuntos de pretexto, treino e teste;

O pré-processamento dos dados foi feito pelo Google Colab por razões de memória e capacidade computacional, com todas as pastas e arquivos carregados no

Google Drive. Após instalar os pacotes necessários e mais algumas alterações de sintaxe, em especial o uso do pacote *sys* para localizar códigos e importar funções, finalmente foi possível pré-processar os dados ao rodar o arquivo *preprocess_sleepedf.py*, o que levou cerca de 34 minutos na CPU.

É válido mencionar que os arquivos das pessoas foram embaralhados aleatoriamente e selecionou-se 58 delas como pretexto e as 20 restantes para validação cruzada (5-fold) na avaliação linear. Assim, as entradas do modelo são segmentos de 30 segundos da gravação contínua de um sinais de EEG de um sujeito, sendo estes segmentos não sobrepostos chamados de época. Como a amostragem do sinal é de 100 Hz, segue que cada época corresponde a um array com 3000 elementos. As épocas são divididas entre os conjuntos de pretexto, treino e teste, tendo o conjunto de pretexto um número grande de exemplos não rotulados se comparado ao número de exemplos rotulados nos demais conjuntos. Assim, a label de cada época corresponde a uma das 6 classes conforme o estágio do sono: 0 para Wake, 1 para N1, 2 para N2, 3 para N3, 4 para REM e 5 para desconhecido. A imagem abaixo ilustra o resultado final dos dados.

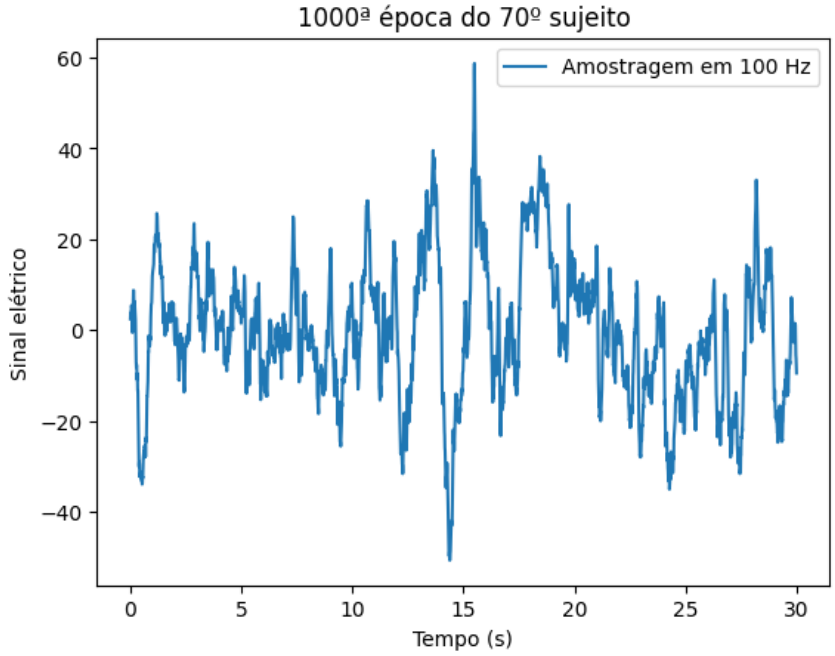


Figura 2: Representação dos sinais elétricos da 1000ª época do 70º sujeito na amostragem original de 100 Hz, cujo estágio do sono é o N2. Fonte: Autora.

3.2 Organização dos códigos para treinamento do modelo

Nesta subseção será detalhado a implementação do mulEEG e estruturas utilizadas conforme códigos disponibilizados em Kumar et al. [2022]. Por tratar-se de um modelo complexo, espera-se que as informações descritas a seguir auxiliem os interessados em manipular tais códigos.

3.2.1 Data augmentation e dataloaders

O primeiro passo do modelo é o de data augmentation e na pasta *utils* temos as implementações relacionadas em *augmentations.py*.

Entre as funções, temos

- *add_noise*: adiciona ruído de baixa e alta frequência ao sinal EEG;
- *jitter*: aplica ruído uniforme aleatório ao sinal EEG;
- *scaling*: aplica ruído gaussiano ao sinal EEG;
- *masking*: mascara um único segmento do sinal EEG aleatoriamente;
- *multi_masking*: mascara vários segmentos do sinal EEG aleatoriamente;
- *flip*: aplica uma inversão horizontal ao sinal EEG;
- *augment*: constrói o pipeline de data augmentation;

Veja que a função *augment* combina as modificações para obter as famílias de ampliações desejadas. Assim , para a família T_1 usou-se *jitter* e *masking* ou *multi_masking*, dependendo da escolha, obtendo como saída ampliações chamadas fracas (do inglês, weak). Já na segunda família de ampliações, T_2, aplicou-se *flip* e *scaling*, tendo as saídas fortes (do inglês, strong).

Ainda na pasta *utils*, temos o arquivo *dataloader.py* e nele as classes *Load_Dataset* e *data_generator*. A primeira é a responsável por aplicar as ampliações aos dados para o caso em que o treinamento é o auto-supervisionado. Daí, em *data_generator*, o conjunto de pretexto é carregado, aplica-se as ampliações com a classe *Load_Dataset* e cria-se o dataloader para o modelo com uso da biblioteca *torch.utils.data*.

No mesmo arquivo, está ainda a classe *cross_data_generator*, que gera um dataloader do tipo k-fold para a avaliação linear a partir dos conjuntos de treino e teste obtidos no pré-processamento dos dados. Aqui, uma observação importante é que as numerações dos sujeitos dos conjuntos de treino e teste, bem como o número de épocas referentes a cada um, devem ser inseridos manualmente, afim de garantir que não haverá vazamento de dados. Vale ressaltar que os conjuntos de teste estão sendo referidos como conjunto de validação ao longo dos códigos.

3.2.2 Encoders, projection head e funções de perda

Na pasta *models* em *spectrogram_model.py*, a classe *CNNEncoder2D_SLEEP* possui o método *torch_stft* para aplicar a Short Time Fourier Transform (STFT) à série temporal e obter o espectrograma, com número de pontos FFT definido para 256 e comprimento do salto para 64. Daí, o método *foward* executa essa transformação e a passa por uma rede convolucional, que é o encoder do espectrograma, E_s. Já o encoder para a série temporal, E_t, trata-se da conhecida arquitetura ResNet-50, mas com convoluções 1D, a qual se encontra na mesma pasta em *resnet1d.py*, na classe *BaseNet*. Após passar pela ResNet, a série temporal ainda é passada por um bloco de atenção, obtendo a feature de tempo. Assim, ainda na pasta *models*, em *model.py*, a classe *encoder* retorna as features do tempo e do espectrograma após passarem pelos seus respectivos encoders, estando ainda nesse mesmo arquivo a classe *attention* que contém o bloco de atenção de E_t

Em *model.py* está ainda a classe *projection_head*, rede neural totalmente conectada não linear de duas camadas, incluindo Batch Norms (normalização na camada escondida), ficando com a seguinte estrutura: Linear - BatchNorm - ReLU - Linear. Finalmente, a classe *sleep_model* é responsável por obter as features de tempo, espectrograma e concatenação de ambas para as entradas fraca e forte ao passá-las pelo *encoder* e *projection_head*.

Ainda em *model.py*, a classe *contrast_loss* é a responsável por aplicar as funções de perda contrastiva após a aplicar o *sleep_model* às entradas, conforme descrito na seção 2.2. Assim, o método *foward* retorna a perda total dada pela equação 5, com parâmetros $\lambda_1 = \lambda_2 = 1$, bem como as outras perdas que a compõem (perda con-

trastiva das features do tempo, do espectrograma e concatenação de ambas e a perda diversa).

Quanto à avaliação linear do encoder E_t, temos ainda a classe *ft_loss* em *model.py*. Ela é responsável por carregar os parâmetros desse encoder, aplicá-los à entrada e em seguida passar por uma transformação linear, retornando saída com cinco elementos, um para cada fase do sono.

3.2.3 Treinamento do modelo

Em *helper_train.py*, a classe *sleep_pretrain* é destinada ao treinamento da tarefa de pretexto, que a princípio deve contemplar 200 épocas. Segue abaixo breve descrição dos seus métodos:

- *training_step*: obtém perdas para um batch, tanto para as entradas fortes quanto fracas, ao chamar o modelo *contrast_loss* definido na seção 3.2.2;
- *training_epoch_end*: ao fim do treino da época, obtém média das perdas entre todos batchs conforme método anterior, além de reduzir a taxa de aprendizado se o modelo não estiver melhorando;
- *on_epoch_end*: ao fim da época, salva pesos dos encoders como *mulEEG.pt* e de todo o modelo, *contrast_loss*, como *mulEEG_full.pt* na pasta de referência, no caso *saved_weights*.
- *ft_fun*: gera o dataloader para os conjuntos de treino e teste a partir da classe *cross_data_generator* apresentada em 3.2.1, passando-os por *sleep_ft.fit* (detalhes adiante), que treinam o encoder E_t conforme avaliação linear descrita na seção anterior, retornando as métricas obtidas;
- *do_kfold*: executa *ft_fun* sobre os k subconjuntos de treino e teste (k-fold), retornando as médias das métricas de avaliação;
- *fit*: para cada época, para cada batch obtém as perdas pelo método *training_step* e otimiza os parâmetros do modelo *contrast_loss*. Daí, com o método *training_epoch_end*

obtém as médias das perdas e salva os pesos com o método *on_epoch_end*. Em seguida, realiza avaliação linear do modelo se a época for múltipla de 4 e maior ou igual a 80 ao chamar o método *do_kfold*, de modo que se as métricas F1-score e/ou sua média forem maiores que o máximo delas até então, atualiza esses máximos e salva os pesos dos encoders como *mulEEG_best.pt* e/ou *mulEEG_mean_best.pt*, respectivamente, na pasta de referência *saved_weights*.

Já a classe *sleep_ft* é destinada à avaliação linear do mulEEG descrita na seção 2.2, que tem como modelo a *ft_loss* definida em 3.2.2 e a Cross Entropy Loss como critério, além de usar o otimizador Adam, de modo a treinar E_t e a camada linear por 100 épocas. Seus métodos são

- *train_dataloader*: retorna o dataloader de treino, passado como parâmetro da classe;
- *val_dataloader*: retorna o dataloader de teste, passado como parâmetro da classe;
- *training_step*: passa um batch do treino pelo modelo e retorna a loss function;
- *validation_step*: passa um batch do teste pelo modelo e retorna a loss function, acurária, previsão de saída e saída esperada;
- *validation_epoch_end*: ao fim da época, obtém métricas sobre o todo o conjunto de teste, isto é, todos os batchs (acurácia, macro F1-score e kappa de Cohen) e, se a MF1 obtida for a máxima até então, atualiza os valores máximos de cada métrica para aqueles obtidos nessa época;
- *on_train_end*: ao final do treinamento, isto é, após todas as épocas, obtém média de MF1 sobre todas as épocas, retornando esta e o máximo das demais métricas;
- *fit*: para cada época da avaliação linear, sendo 100 ao todo, passa pelos loopings de treino e teste. No primeiro, passa cada batch por *traininga_step* e otimiza os parâmetros do modelo. No de teste, passa cada batch por *validation_step* e ao fim usa o método *validation_epoch_end*. A função retorna o método *on_train_end* com todas as métricas.

Finalmente, em *train.py*, treina-se o modelo ao unir tudo o que foi definido anteriormente. Nesse arquivo, as pastas são organizadas e o wandb configurado, ferramenta esta que permite acompanhar o treinamento, o dataloader de pretexto é carregado pela classe *data_generator* da seção 3.2.1 e o modelo *sleep_pretrain* carregado, chamando o seu módulo *fit* para treiná-lo.

3.3 Rodando os códigos: dificuldades e soluções

A princípio, foi necessário instalar uma série de pacotes Python, bem como realizar correções de sintaxe e de referência à arquivos ou funções nos códigos descritos na seção 3.2. Além disso, criou-se uma conta no Weights & Biases para acompanhar o treinamento.

Uma vez feito o pré-processamento dos dados conforme a seção 3.1, iniciaram-se as tentativas de treinar o modelo a partir de *train.py*, novamente via Google Collab com uso da CPU. Uma vez solucionados os erros ao longo do código, o problema passou a ser o tempo de execução: mesmo rodando por horas, só era possível treinar uma das 200 épocas da tarefa de pretexto programadas. Trocando para a GPU, foi possível realizar o treinamento de pretexto por 24 épocas até esgotar os recursos do Google Collab.

Contudo, o resultado alcançado ainda não era suficiente, momento este em que considerou-se reduzir a resolução dos dados - de 100 Hz para 50 Hz, por exemplo - para acelerar o treinamento. Aqui, a maneira mais simples de fazê-lo era, dados os arquivos já em formato numpy e divididos por pessoa, tomar os sinais de dois em dois em cada época, de modo que a amostragem passou a ser 50 Hz. O resultado pareceu satisfatório e essa resolução já seria suficiente para tentar treinar o modelo, sem ter que diminuir o número de exemplos, como podemos inferir da figura a seguir.

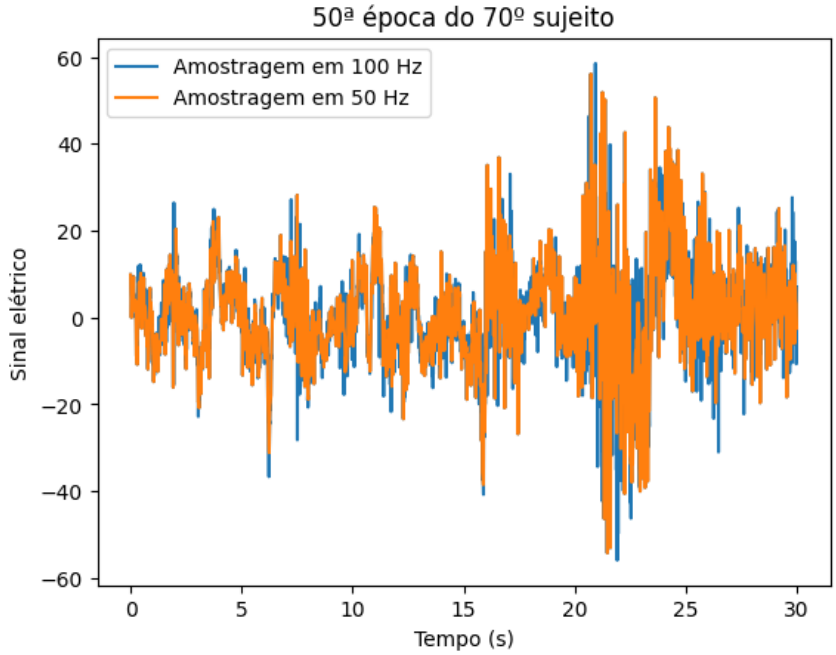


Figura 3: Representação dos sinais elétricos da 50ª época do 70º sujeito na amostragem original, de 100 Hz, e a de 50 Hz, cujo estágio do sono é Wake. Fonte: Autora.

No entanto, ao tentar treinar o mulEEG com estes dados redimensionados, o código apontou problemas de incompatibilidade de dimensões de arrays que não foram possíveis de serem solucionados.

Como ainda era importante diminuir o tempo de treinamento do modelo, não restou outra alternativa que não diminuir o conjunto de treino, o que foi feito tomando as épocas de cada indivíduo de n em n, para n inteiro. A princípio, foi escolhido $n = 2$ e, por meio desse novo conjunto de dados, foi possível treinar 80 épocas da tarefa de pretexto com a GPU do Google Collab. Dessa vez, a interrupção do treinamento se deu por mais problemas no código, dessa vez na etapa de avaliação linear.

Nesse momento o interesse era apenas completar o treinamento, sem tanta importância para a sua qualidade, então foi escolhido $n = 5$ para acelerá-lo, iniciando a avaliação linear logo na primeira época para debugar o código. Resolvidos os problemas de sintaxe, mais uma tentativa de treinar o modelo foi feita para esse conjunto de dados, mas agora com a avaliação linear no padrão original. Com todos os recursos do Google Collab, usando GPU, foi possível treinar a tarefa de pretexto por 104 épocas. Vale lembrar que a cada etapa de avaliação linear o modelo escolhido é treinado por 100 épocas, o que implica em um treinamento do mulEEG ainda mais lento a partir da

sua 80ª época.

Contudo, os objetivos dessa primeira parte do projeto foram atingidos: entender a construção dos códigos do modelo, corrigir erros e conseguir treiná-lo, ainda que não tenham sido todas as épocas programadas por falta de recursos. O próximo passo concentra-se em reduzir a complexidade do modelo e realizar uma série de testes afim de comparar as métricas obtidas.

4 Complexidade do modelo

Uma vez que o treinamento do modelo ainda era muito longo e, observando nas tentativas de treinamento descritas na seção 3.3 que a *loss function* decai lentamente, considerou-se a ideia de que o modelo pode estar muito complexo. Para averiguar essa hipótese, conforme sugestão do professor orientador, o próximo passo foi substituir a ResNet-50 do encoder para a série temporal, E_t, por uma ResNet-18, também com convoluções unidimensionais. Independente de ser mais vantajoso ou não, foi interessante trabalhar nessa ideia para entender por completo como funciona a construção de E_t, já que ele pode desempenhar papel importante na tarefa principal de classificação.

4.1 ResNet adaptada para séries temporais

O primeiro passo para construir uma ResNet com convoluções 1D menor para E_t foi entender como foi feita a adaptação da ResNet-50 original, disponível em He et al. [2015], para dados unidimensionais. A seguir, está apresentada uma descrição detalhada dessa importante estrutura de aprendizado profundo.

De acordo com Florindo [2023], redes neurais muito profundas são difíceis de treinar, pois há maior chance de o gradiente desaparecer ou explodir. Além disso, ao aumentar o número de camadas em uma rede neural típica (ou *"plain networks"*) seria esperado que isso ao menos não piorasse o treinamento. Porém, na prática, o erro de treino pode aumentar. Empiricamente, a razão atribuída à isso é que quando a rede já obtém o resultado esperado em determinada camada, com as não linearidades é difícil mantê-lo ao longo das camadas seguintes, isto é, que estas camadas aprendam a função identidade.

A solução para isso foi acrescentar atalhos (ou *"skip connections"*) à rede, isto é, quando a ativação de uma camada alimenta diretamente outra camada mais profunda na rede. Intuitivamente, fica mais fácil manter uma mesma ativação ao longo de várias camadas, resolvendo o problema descrito anteriormente e permitindo o treino de redes muito profundas sem que o erro de treino aumente.

Com isso, o elemento básico dessa nova arquitetura é o bloco residual. Para melhor elucidar, seja B uma sequência de N camadas convolucionais de uma rede neural, onde $B_1, B_2, ..., B_N$ são as camadas que o compõem, $z^{[B_1]}, z^{[B_2]}, ..., z^{[B_N]}$ e $a^{[B_1]}, a^{[B_2]}, ..., a^{[B_N]}$ são as combinações lineares e ativações delas, respectivamente, e $a^{[K]}$ a entrada deste bloco, isto é, a entrada da camada B_1. Se B fosse um bloco de uma rede típica, teríamos que $a^{[i]} = g(z^{[i]})\ \forall i = B_1, B_2, ..., B_N$, onde g é a função de ativação da rede. Sendo B um bloco de uma rede residual, teremos que $a^{[i]} = g(z^{[i]})\ \forall i = B_1, B_2, ..., B_{N-1}$ e $a^{[N]} = g(z^{[N]} + a^{[K]})$, adicionando uma *skip connection* entre a primeira e a última camada do bloco. Vale destacar que é necessário que $z^{[N]}$ e $a^{[K]}$ tenham a mesma dimensão; do contrário basta redimensionar $a^{[K]}$ ao multiplicá-la por uma matriz de pesos (chamada no artigo de He et al. [2015] de projeção linear), os quais podem ser aprendidos ou fixos.

Em outras palavras, em um bloco residual as entradas são passadas normalmente ao longo das camadas até a penúltima, sendo que na última somamos a entrada do bloco à combinação linear desta camada e aplicamos a função da ativação sobre este novo valor, obtendo assim a saída do bloco residual. Vale notar que, em geral, os blocos residuais possuem mais de uma camada, já que com uma única camada a estrutura é similar à do bloco típico. Assim, uma ResNet é formada empilhando vários desses blocos residuais.

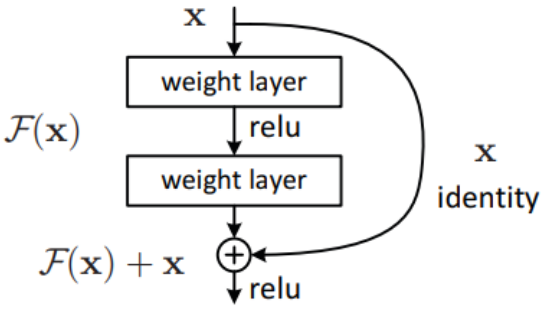


Figura 4: Exemplo de bloco residual de duas camadas com função de ativação ReLU. Fonte: He et al. [2015].

Na ResNet-18 e na ResNet-34, onde o número se refere à quantidade de camadas, observamos conforme He et al. [2015] blocos residuais com 2 camadas. Em uma ResNet mais profunda, como a ResNet-50, temos um bloco residual especial em razão da preocupação com o tempo de treinamento: o *bottleneck block*. Ele possui três camadas, com convoluções 1×1, 3×3 e 1×1, nesta ordem, onde as convoluções 1×1 são responsáveis por reduzir e depois aumentar a dimensão do dado, de modo que a camada maior 3×3 tenha entrada e saída menores.

Contudo, analisando em detalhes a implementação da ResNet-50 adaptada com convoluções 1D do mulEEG, tem-se a arquitetura na tabela abaixo. Considere que a entrada da rede é um array de dimensão $[256 \times 1 \times 3000]$, onde 256 é o tamanho do batch, isto é, o número de épocas (aqui, época assume o sentido descrito na seção 3.1), 1 se refere ao único canal (dimensão) que a série temporal possui e 3000 é o comprimento de cada época (novamente, conforme especificado na seção 3.1).

ResNet-50 com convoluções 1D usada no mulEEG

Nome da camada	Dimensão da saída	Arquitetura
Conv0	$[256 \times 16 \times 1500]$ $[256 \times 16 \times 750]$	$k = 71, f = 16, s = 2, p = 35$ $k = 71, s = 2, p = 35$ Max-Pooling
Conv1_x	$[256 \times 32 \times 750]$	$\begin{bmatrix} k=1 & f=8 & s=1 & p=0 \\ k=25 & f=8 & s=1 & p=12 \\ k=1 & f=32 & s=1 & p=0 \end{bmatrix} \times 3$
Conv2_x	$[256 \times 64 \times 375]$	$\begin{bmatrix} k=1 & f=16 & s=1 & p=0 \\ k=25 & f=16 & s=2 & p=12 \\ k=1 & f=64 & s=1 & p=0 \end{bmatrix} \times 4$
Conv3_x	$[256 \times 128 \times 188]$	$\begin{bmatrix} k=1 & f=32 & s=1 & p=0 \\ k=25 & f=32 & s=2 & p=12 \\ k=1 & f=128 & s=1 & p=0 \end{bmatrix} \times 6$
Conv4_x	$[256 \times 256 \times 94]$	$\begin{bmatrix} k=1 & f=64 & s=1 & p=0 \\ k=25 & f=64 & s=2 & p=12 \\ k=1 & f=256 & s=1 & p=0 \end{bmatrix} \times 3$

Tabela 1: Arquitetura da ResNet-50 com convoluções 1D, isto é, adaptada para dados do tipo série temporal, utilizada no mulEEG. Os *bottleneck blocks* estão representados entre parênteses (veja também a Figura 5) e a quantidade de vezes que eles são repetidos indicada ao lado, sendo k o *kernel* (tamanho do filtro), f o número de filtros, s o *stride* e p o *padding* da convolução 1D. Onde há $s = 2$, considera-se de fato *stride* igual a 2 apenas para a primeira repetição do *bottleneck block* em questão, momento este em que é feito o *downsampling*, nas demais o *stride* é 1. Fonte: Autora.

Por fins de completude, veja que a dimensão da saída de uma convolução 1D, n_{out}, é dada por

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - (k-1) - 1}{s} + 1 \right\rfloor$$

onde n_{in} é a dimensão da entrada, p o *padding*, k o *kernel* e s o *stride*. Daí o *downsampling* para convoluções com *stride* igual a 2.

Nesse sentido, comparando como os autores adaptaram as convoluções da ResNet-50 em He et al. [2015] para convoluções 1D, construiu-se a arquitetura da ResNet-18 com convoluções 1D de maneira análoga, como mostra a tabela abaixo. Basicamente, os *bottleneck blocks* foram substituídos por blocos residuais de duas camadas, sem incluir o processo de reduzir e depois aumentar a dimensão do dado.

ResNet-18 com convoluções 1D

Nome da camada	Dimensão da saída	Arquitetura
Conv0	$[256 \times 16 \times 1500]$ $[256 \times 16 \times 750]$	$k = 71, f = 16, s = 2, p = 35$ $k = 71, s = 2, p = 35$ Max-Pooling
Conv1_x	$[256 \times 8 \times 750]$	$\begin{bmatrix} k = 25 & f = 8 & s = 1 & p = 12 \\ k = 25 & f = 8 & s = 1 & p = 12 \end{bmatrix} \times 2$
Conv2_x	$[256 \times 16 \times 375]$	$\begin{bmatrix} k = 25 & f = 16 & s = 1 & p = 12 \\ k = 25 & f = 16 & s = 2 & p = 12 \end{bmatrix} \times 2$
Conv3_x	$[256 \times 32 \times 188]$	$\begin{bmatrix} k = 25 & f = 32 & s = 1 & p = 12 \\ k = 25 & f = 32 & s = 2 & p = 12 \end{bmatrix} \times 2$
Conv4_x	$[256 \times 64 \times 94]$	$\begin{bmatrix} k = 25 & f = 64 & s = 1 & p = 12 \\ k = 25 & f = 64 & s = 2 & p = 12 \end{bmatrix} \times 2$

Tabela 2: Arquitetura da ResNet-18 com convoluções 1D, isto é, adaptada em comparação com He et al. [2015] e Kumar et al. [2022] para dados do tipo série temporal pela autora. Os blocos residuais estão representados entre parênteses (veja também a Figura 5) e a quantidade de vezes que eles são repetidos indicada ao lado, sendo k o *kernel* (tamanho do filtro), f o número de filtros, s o *stride* e p o *padding* da convolução 1D. Onde há $s = 2$, considera-se de fato *stride* igual a 2 apenas para a primeira repetição do bloco residual em questão, momento este em que é feito o *downsampling*, nas demais o *stride* é 1. Fonte: Autora.

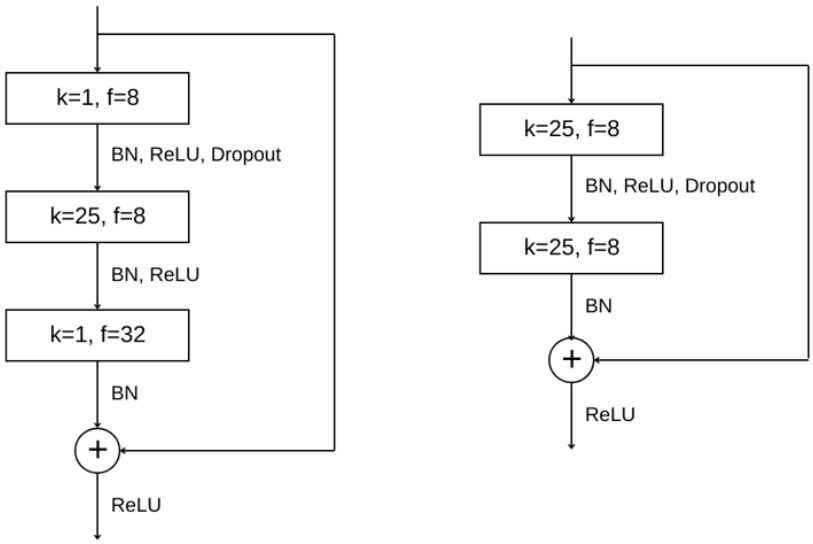


Figura 5: À esquerda, um *bottleneck block* da ResNet-50 com convoluções 1D conforme a Tabela 1. À direita, um bloco residual da ResNet-18 com convoluções 1D conforme a Tabela 2. Fonte: Autora.

4.2 Implementação da ResNet-18 com convoluções 1D

Conforme descrito na seção 3.2.2, a construção da ResNet-50 com convoluções 1D utilizada no encoder para série temporal está contida no arquivo *resnet1d.py* da pasta *models*. A implementação da ResNet-18 com convoluções 1D descrita na tabela 2 foi feita de maneira análoga e no mesmo arquivo, incluindo as classes *BasicBlock_Bottle_18* e *BaseNet_18*, sendo esta última o modelo em si.

Com esta implementação, outras partes do código necessitaram correção, sendo aquelas que pressupunham que após o batch passar pelo encoder E_t a saída teria dimensão $[256 \times 256 \times 94]$, como acontecia com a ResNet-50. Agora, com a ResNet-18, a saída passou a ter dimensão $[256 \times 64 \times 94]$, reduzindo em 4 vezes o número de canais. As correções ocorreram nas classes *attention* e *projection_head* do arquivo *model.py* da pasta *models*. Agora, uma variável no começo do arquivo indica o número de canais da feature de tempo, *time_channels*, bastando alterar a mesma conforme o modelo da ResNet indicada na classe *encoder*.

4.3 Procedimentos para treinamento do modelo

Uma vez que os códigos de Kumar et al. [2022] foram compreendidos, explorados e debugados, além da implementação da ResNet-18 com convoluções 1D descrita na seção anterior, o passo seguinte foi realizar o treinamento completo do modelo, de modo a variar a ResNet e a quantidade de dados utilizada para comparações de desempenho. Aqui, um dos objetivos era averiguar se havia necessidade de um treino tão longo e um modelo tão complexo quanto o proposto originalmente em Kumar et al. [2022].

Para tanto, utilizou-se uma máquina com placa de vídeo Nvidia Titan V para que não houvesse dependência dos recursos disponibilizados gratuitamente pelo Google Collab. O acesso remoto à essa máquina foi feito pelo programa TeamViewer e um ambiente virtual foi criado em Python para executar os programas.

A princípio, foi realizado o treinamento do modelo utilizando a ResNet-18 e apenas 20% dos dados, já que este deveria ser o mais rápido de ser treinado. Infelizmente, em cerca de 136 minutos e após 180 épocas treinadas, a execução do

programa foi interrompida com a mensagem de que o kernel morreu. Isso geralmente decorre do estouro da memória da GPU e a recomendação clássica para este problema é a redução do tamanho do batch com o qual é feito o treinamento, que neste caso era 256. Nesse momento, o objetivo tornou-se simplesmente completar o treinamento do modelo com os recursos disponíveis.

Nesse sentido, um novo treinamento com a mesma configuração foi realizado, com a única alteração sendo o tamanho do batch, que foi igual a 128. Foi possível treinar 190 épocas em cerca de 244 minutos antes de o kernel morrer; nota-se que apesar de ter sido possível treinar 10 épocas a mais, para tanto foi necessário quase duas horas adicionais. Diminuindo o tamanho do batch para 64, treinou-se 195 épocas em cerca de 254 minutos. O fato é que a redução do tamanho do batch não trouxe ganhos expressivos em termos de processamento, pois além de aumentar expressivamente a duração do treinamento, o número de épocas rodadas foi comparável. Vale ressaltar que o treino ter sido mais longo provavelmente decorre do fato de que a diminuição do tamanho do batch implica na diminuição da paralelização do algoritmo, característica fundamental no treinamento de redes profundas.

Contudo, a ideia de reduzir o tamanho do batch foi descartada. Para completar o treinamento, optou-se então por uma alternativa simples: salvar o parâmetros do modelo e do estágio do treinamento a cada época para retomar o treino quando o kernell morrer. O interessante dessa solução é que a plataforma Wandb (Weights and Biases), que registra o histórico de todas as métricas relevantes do treinamento e está sendo utilizada no presente projeto, já possui como opção retomar um treinamento, de modo que as métricas continuam a serem registradas no mesmo arquivo. Assim, com estes pequenos ajustes no código, foi possível treinar os modelos por completo.

4.4 Resultados

Para treinar o mulEEG e realizar comparações, foram escolhidas quatro configurações, variando a quantidade de dados entre 20 e 100% e o enconder temporal, E_t, entre a ResNet-50 e a ResNet-18. Contudo, dado que o treinamento do mulEEG é muito longo, o objetivo principal é analisar se o modelo original com a ResNet-50 é muito complexo para o problema e se de fato são necessários tantos dados para obter

bons resultados.

Nesse sentido, as métricas a serem comparadas são a função de custo total da tarefa de pretexto, cuja função de perda associada é dada por 5, além daquelas relativas à avaliação linear do modelo, isto é, kappa de Cohen (κ), acurácia (Acc) e MF1. Ademais, também serão apresentados a duração de cada treinamento, afim de avaliar o custo-benefício. Abaixo, estão dispostos dados e gráficos relativos aos treinamentos supramencionados.

Método	**Acc**	κ	**MF1**	**Duração do treino**
20% dos dados + ResNet-18	0.6979	0.5705	0.5252	3h 2m 55s
20% dos dados + ResNet-50	0.7483	0.6469	0.6056	4h 27m 26s
100% dos dados + ResNet-18	0.7549	0.6528	0.6189	13h 21m 22s
100% dos dados + ResNet-50	0.7653	0.6704	0.6546	21h 2m 56s
Resultados do artigo	**0.7806**	**0.6850**	**0.6782**	**-**

Tabela 3: Métricas da avaliação linear para diferentes configurações de treinamento do mulEEG, além daquelas observadas na referência principal.

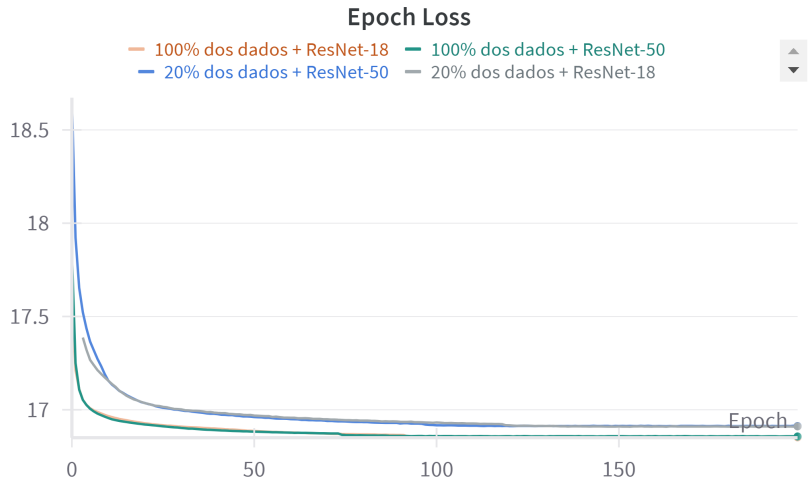


Figura 6: Evolução da função de custo total do mulEEG ao longo das épocas. Fonte: Autora (via WandB).

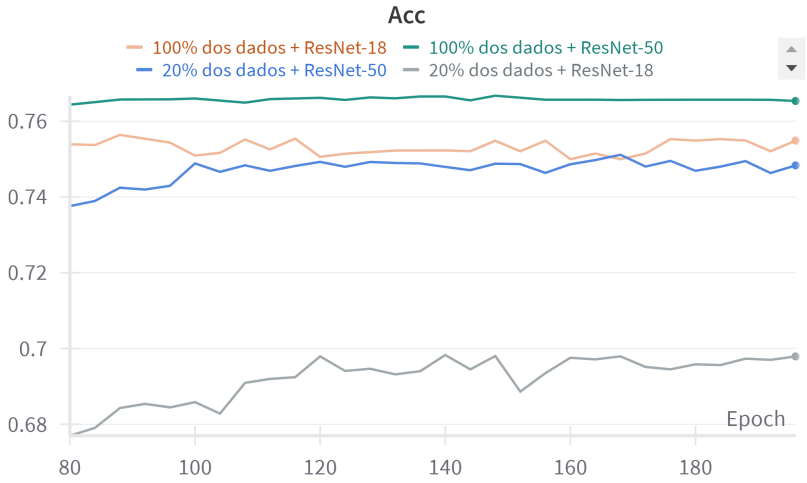


Figura 7: Evolução da medida de acurácia na avaliação linear do mulEEG ao longo das épocas. Fonte: Autora (via WandB).

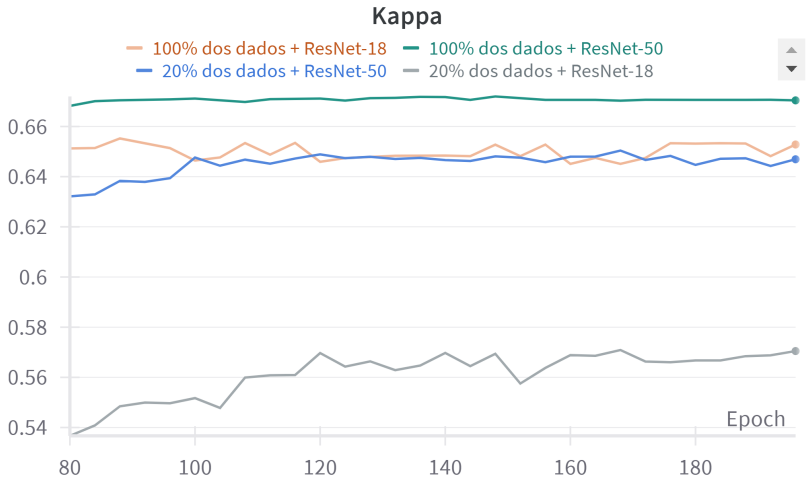


Figura 8: Evolução da medida de kappa de Cohen na avaliação linear do mulEEG ao longo das épocas. Fonte: Autora (via WandB).

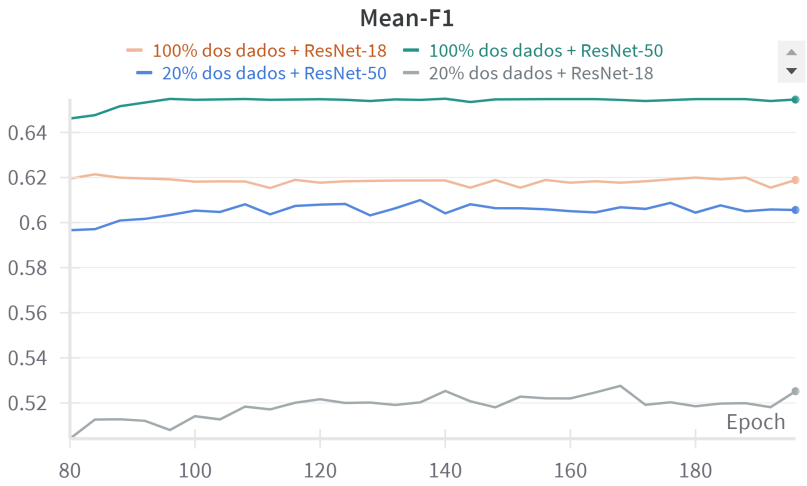


Figura 9: Evolução da medida de MF1 na avaliação linear do mulEEG ao longo das épocas. Fonte: Autora (via WandB).

A princípio, note da tabela 3 que, de fato, o uso da ResNet-50 torna o treinamento muito mais demorado em relação ao uso da ResNet-18, em especial com 100% dos dados, além de apresentar melhores resultados. Nesse sentido, as métricas da avaliação linear obtidas com a ResNet-50 e 100% dos dados foram próximas àquelas de Kumar et al. [2022], assim o objetivo de reproduzir os resultados do artigo foi alcançado.

Da evolução das métricas ao longo das épocas, perceba da figura 6 que a função de custo da tarefa de pretexto se estabiliza em poucas épocas em todos os casos, sendo em torno de um valor menor quando usados 100% dos dados. Contudo, as métricas da avaliação linear mostram-se bastante instáveis, isto é, com bastante variação ao longo dos treinamentos, com exceção do caso com $ResNet - 50$ e todos os dados. Assim, pode não ser necessário que esta última configuração seja treinada por tantas épocas, reduzindo sua duração.

Dessa forma, o treinamento do mulEEG com ResNet-50 e 100% dos dados apresentou as melhores métricas dentre as configurações experimentadas, mas foi também o de maior custo computacional. Por outro lado, o treino usando a ResNet-18 e 100% dos dados forneceu métricas muito próximas daquelas usando ResNet-50 e 20% dos dados, sendo que o treinamento deste último teve cerca de um terço da duração do primeiro. Assim, pode-se concluir que o treino com ResNet-50 e 20% dos

dados apresenta o melhor custo-benefício, ou seja, é preferível diminuir o volume de dados que simplificar o modelo para acelerar o treinamento.

5 Classificador

Conforme descrito na seção 2.1, o mulEEG tem como objetivo aprender representações efetivas dos sinais de EEG. Assim, o classificador simples utilizado na avaliação linear do método tem apenas este fim, de modo que a tarefa de classificação a partir da representação aprendida não é explorada em Kumar et al. [2022]. Assim, esta seção tem como objetivo explorar um classificador a partir das representações aprendidas conforme a seção 4.4.

5.1 Support Vector Machine (SVM)

O Support Vector Machine, ou simplesmente SVM, é um algoritmo de aprendizado de máquina supervisionado que pode ser utilizado para problemas de classificação. Seu objetivo é obter um hiperplano que separe as diferentes classes com a maior distância possível entre eles, motivo pelo qual é considerado um classificador de margem de separação larga. Trata-se de um método muito utilizado para diagnóstico médico e, por essa razão, foi escolhido para ser explorado no presente projeto. A seguir, veremos a intuição matemática por trás da sua formulação baseado em Florindo [2022].

Na regressão logística, classificador binário, dado um exemplo $(\mathbf{x}, y)$, em que $\mathbf{x} \in \mathbb{R}^{n\times 1}$ é a entrada e $y \in \{0, 1\}$ a saída, temos como função de hipótese a sigmoide logística dada por

$$h(\mathbf{x}; \mathbf{w}, b) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x}+b)}}, \tag{6}$$

onde $\mathbf{w} \in \mathbb{R}^{n\times 1}$ é o vetor de parâmetros e $b \in \mathbb{R}$ é o termo *bias*. Definindo ainda a variável $z = \mathbf{w}^T\mathbf{x} + b$, podemos reescrever

$$h(z) = \frac{1}{1 + e^{-z}}. \tag{7}$$

com fronteira de decisão sendo tal que a previsão do modelo $\hat{y}$ é dada por

$$\hat{y} = \begin{cases} 1, \text{ se } h(z) > 0.5 \Leftrightarrow z > 0 \\ 0, \text{ se } h(z) \leq 0.5 \Leftrightarrow z \leq 0 \end{cases} \tag{8}$$

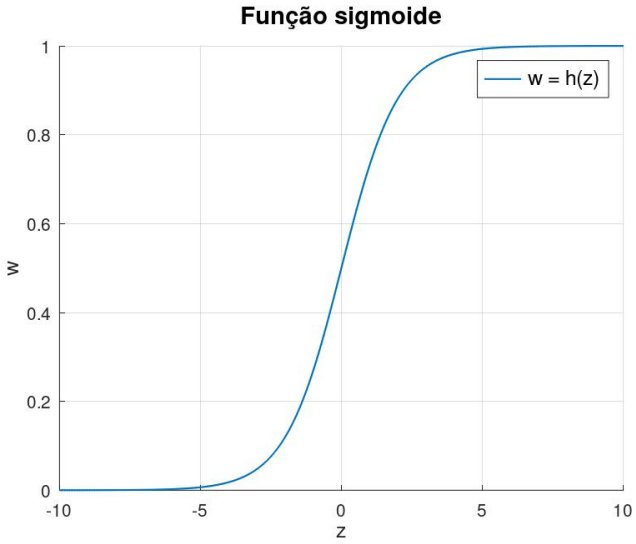


Figura 10: Gráfico da função sigmoide dada por 7. Fonte: Autora.

A respectiva função de perda logística (ou *logistic loss*) para esse exemplo é dada por

$$L(z) = -[y \log(h(z)) + (1-y) \log(1-h(z))]. \tag{9}$$

Nesse sentido, sendo m o número de exemplos, $(x^{(i)}, y^{(i)})$ o i-ésimo exemplo, com $x^{(i)} \in \mathbb{R}^{n \times 1}$ sendo a entrada e $y^{(i)} \in \{0,1\}$ a saída, $z^{(i)} = \mathbf{w}^T x^{(i)} + b$ e λ o parâmetro de regularização, temos a função de custo para a regressão logística

$$J(\mathbf{w}, b) = \sum_{i=1}^{m} L(z^{(i)}) + \frac{\lambda}{2} \sum_{j=1}^{n} w_j^2. \tag{10}$$

Observe que se $y = 1$, então desejamos que $z \gg 0$ para que $h(z) \approx 1$, obtendo a função de perda para esse exemplo

$$L_1(z) = -log(h(z)) = -\log\left(\frac{1}{1+e^{-z}}\right). \tag{11}$$

Já se $y = 0$, então exigimos que $z \ll 0$ para que $h(z) \approx 0$, obtendo a seguinte função de perda

$$L_0(z) = -\log(1 - h(z)) = -\log\left(1 - \frac{1}{1 + e^{-z}}\right) \tag{12}$$

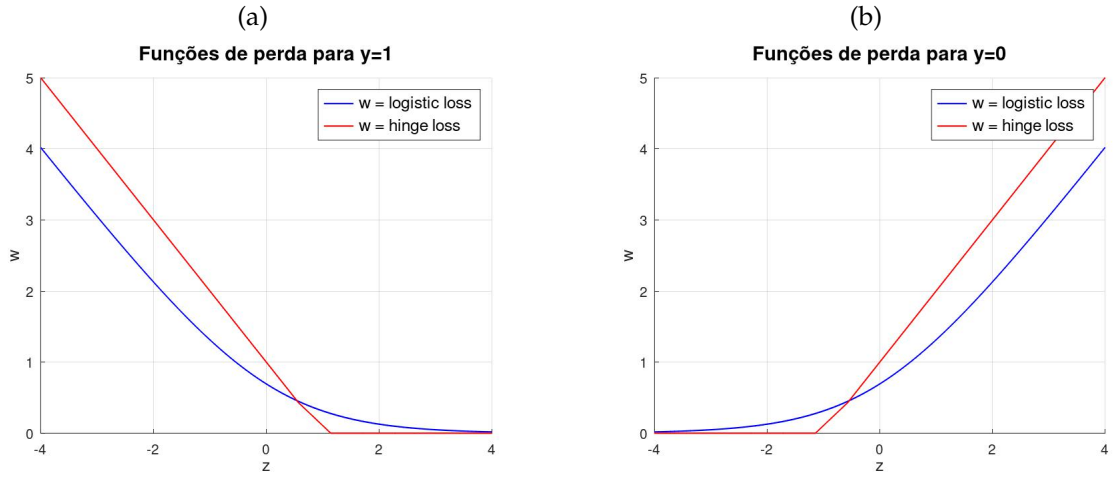
No SVM, a ideia é aproximar as funções 11 e 12 por funções lineares por partes, de modo que $L_1(z) \approx \max(0, 1 - z)$ e $L_0(z) \approx \max(0, 1 + z)$. Se convencionarmos que ao invés de usar a classe $y = 0$ usaremos $y = -1$, podemos ainda escrever a forma geral dessa aproximação,

$$\bar{L}(z) = max(0, 1 - yz), \tag{13}$$

a qual chamamos de perda de articulação (ou *hinge loss*). Note que $L(z) \approx \bar{L}(z)$, assim, pela mesma convenção anterior, temos também uma aproximação para a função de custo da regressão logística, dada por

$$\bar{J}(\mathbf{w}, b) = \sum_{i=1}^{m} \bar{L}(z^{(i)}) + \frac{\lambda}{2} \sum_{j=1}^{n} w_j^2. \tag{14}$$

Figura 11: Funções de perda logística e de acumulação para as classes 0 e 1.



Sendo $C = 1/\lambda$, podemos ainda dividir a equação 14 por λ, definindo

$$\hat{J}(\mathbf{w}, b) = C \sum_{i=1}^{m} \bar{L}(z^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} w_j^2, \tag{15}$$

de modo que minimizar a função 14 é equivalente ao problema de minimizar a função 15. Assim, a função de custo do SVM é dada pela função 15.

Contudo, para entender a intuição da margem larga, observamos que, tomando o exemplo $(\mathbf{x}, y)$, para zerar a função de perda $\bar{L}(z)$, queremos $z \geq 1$ para $y = 1$ e $z \leq -1$ para $y = -1$. Exigindo que isso aconteça para todos os m exemplos, temos que o primeiro somatório da função 15 vai a zero, obtendo o seguinte problema de minimização para o modelo

$$\begin{aligned} &\arg\min_{\mathbf{w}} \frac{1}{2} \sum_{j=1}^{n} w_j^2 \\ &\text{sujeito a } \begin{cases} z^{(i)} \geq 1 \text{ se } y^{(i)} = 1 \\ z^{(i)} \leq -1 \text{ se } y^{(i)} = -1 \\ \forall i \in \{1, ..., m\} \end{cases} \end{aligned} \tag{16}$$

O algoritmo SVM tem como objetivo resolver esse problema ao minimizar a função de custo $\hat{J}(z)$, de modo a proporcionar um hiperplano separador com a maior margem de separação entre as classes. Veremos a seguir porquê isso acontece.

Veja que da equação 8, temos que a fronteira de decisão do problema, isto é, o hiperplano separador é dado pela curva $z = 0$. Além disso, definindo $\theta = (\mathbf{w}^T, b)^T$ e $\mathbf{x}' = (\mathbf{x}^T, 1)^T$, temos $z = \theta^T \mathbf{x}'$. Assim, $z = 0$ implica que θ é perpendicular à fronteira de decisão.

Observe ainda que $z = \theta^T \mathbf{x}' = p \, \|\theta\|$, onde p é o comprimento "orientado"da projeção de $\mathbf{x}'$ sobre θ. Daí, como θ é perpendicular ao hiperplano separador, segue que p representa a distância "orientada"de $\mathbf{x}'$ ao mesmo.

Do problema 16, temos que $\|\theta\|$ é minimizado e exigimos $p \, \|\theta\| \geq 1$ para $y = 1$ e $p \, \|\theta\| \leq -1$ para $y = -1$, o que implica em p de magnitude grande, sendo positivo para $y = 1$ e negativo para $y = -1$. Lembrando que p é a distância "orientada"de $\mathbf{x}'$ ao hiperplano separador, temos assim a implicação de margem de separação larga entre as classes. Ademais, os dados que estão na margem de separação são chamados vetores de suporte.

Note que até aqui apenas tratamos do problema de classificação binária. O algoritmo pode ser extendido à classificação multiclasses com a abordagem "um-contra-todos", isto é, para cada classe tomar ela como sendo 1 e as demais -1. Além

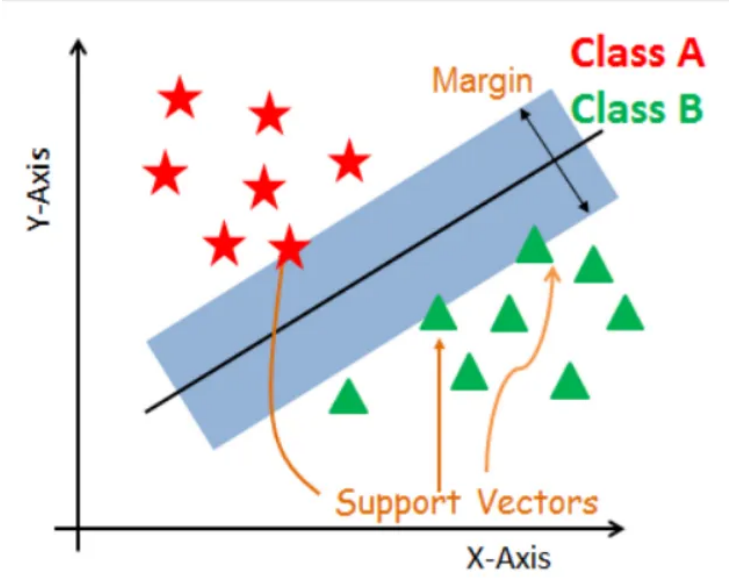


Figura 12: Exemplo de classificação binária com SVM. Fonte: Segatto [2023].

disso, é possível aumentar a complexidade do SVM ao associá-lo à uma função de kernel, que transforma os dados de entrada de modo que a separação das classes por hiperplanos seja mais assertiva. No entanto, kernels complexos aumentam consideravelmente o custo computacional e, considerando que o presente trabalho lida com um grande volume de dados, utilizaremos SVM sem kernel.

5.2 Implementação

Para aplicar o algoritmo SVM ao presente problema de classificação, optou-se por utilizar apenas os dados de teste e validação, desconsiderando os de pretexto, assim como é feito na avaliação linear do mulEEG para fins de comparação e também pensando no alto custo computacional do SVM. Analogamente os dados também são passados pelo encoder temporal, E_t, para obter suas representações efetivas a partir dos parâmetros aprendidos na tarefa de pretexto. Porém, não foi observada a necessidade de K-fold devido ao grande volume de dados, sendo que ao invés disso escolheu-se tomar 25% dos dados para teste, ficando os demais para treinamento do modelo;

Nesse sentido, para carregar e tratar os dados criou-se, em *utils/dataloader.py*, a função *data_generator_classification*, a qual é responsável por concatenar os dados de teste e validação, passar as entradas ao encoder temporal, E_t, e, finalmente, separar

este novo conjunto em treino e teste.

A implementação do algoritmo SVM em si está em um novo arquivo *svm.py*, a partir da função *LinearSVC* do pacote *sklearn.svm*, com parâmetros como o quadrado da *hinge loss* sendo a função de perda, sua tolerância igual a 1^{-5}, $C = 1$ e número máximo de interações igual a 2000. Vale lembrar que as entradas foram normalizadas antes de passarem pelo modelo em si, além de que todas as métricas pertinentes também foram calculadas sobre o conjunto de teste.

5.3 Resultados

O SVM foi treinado usando dois tipos de entradas: a primeira sendo os dados originais, para efeitos de comparação, e a segunda passando-os pelo encoder temporal e tomando as representações aprendidas dos mesmos. Para este último, usaremos os parâmetros de melhor MF1 aprendidos na tarefa de pretexto a partir do treinamento com 100% dos dados e usando a ResNet-50, que chamaremos de representações A, e ResNet-18, representações B. A seguir, estão dispostas as métricas observadas.

Método	Acurácia	Kappa de Cohen	F1
SVM com dados originais	0.3268	0.038	0.2131
SVM com representações A	0.8109	0.7363	0.7348
SVM com representações B	0.7730	0.6810	0.6652
Resultados do artigo	**0.7806**	**0.6850**	**0.6782 (MF1)**

Tabela 4: Métricas observadas na tarefa de classificação utilizando SVM com dados originais, dados com representações A e B, além dos resultados da referência principal para comparação. Fonte: Autora.

Primeiro, observe que as métricas obtidas pelo SVM usando os dados originais foram bastante inferiores quando comparadas às demais. Isso indica que as representações aprendidas pelo mulEEG são de fato efetivas para a tarefa de classificação, atuando como um bom extrator de features desses dados.

Além disso, as métricas da avaliação linear do artigo foram superadas por aquelas geradas ao usar o SVM a partir das representações A dos sinais de EEG. Assim, o objetivo de explorar a tarefa de classificação mostra-se também bem sucedido. Contudo, ao usar representações do tipo B, ainda que as métricas do artigo não tenham

diso superadas, elas foram bastante próximas, demonstrando que mesmo o modelo simplificado pode ser útil para a classificação dos sinais de EEG.

6 Conclusão

Os estudos em aprendizado profundo trazem aplicações cada vez mais interessantes na área da saúde, como apresentado na referência principal deste projeto, Kumar et al. [2022]. Este trata da automatização da classificação das fases do sono e o modelo utilizado para tal é nomeado mulEEG, que se propõe a aprender representações efetivas dos sinais de EEG antes de classificá-los, melhorando a qualidade desta última tarefa.

Nesse sentido, o mulEEG é bastante complexo e entender sua implementação foi um grande desafio. Até então, o conhecimento da beneficiária em aprendizado profundo era limitado à teoria e problemas simples, de modo que o desenvolvimento do presente projeto fora crucial para aprimorá-lo. Assim, manipular os dados e debugar os códigos foi um processo longo e que felizmente trouxe aprendizado e resultados satisfatórios.

Contudo, a limitação de recursos computacionais fora o maior desafio no desenvolvimento do trabalho. Por tratar-se de modelos complexos com um grande volume de dados, foi-se necessário recorrer à truques como salvar os parâmetros do treinamento e retomá-lo a cada certa quantidade de épocas. Naturalmente, isso compromete o andamento do projeto, mas felizmente ele pôde ser concluído.

Ademais, a tentativa de reduzir a complexidade do modelo levou à um interessante estudo da ResNet adaptada com convoluções 1D. Ainda que o uso da ResNet-18 no encoder temporal tenha demonstrado um desempenho inferior à ResNet-50, implementá-la foi uma tarefa bastante proveitosa ao aprendizado.

Dessa forma, ao trabalhar na tarefa de classificação, uma vez aprendida a representação dos dados, também houve uma maior familiarização com o SVM, algoritmo muito usado em aprendizado de máquinas. Contudo, as métricas da avaliação linear do artigo foram superadas por meio do classificador supramencionado, demonstrando um avanço interessante ao problema, além da efetividade das representações

aprendidas pelo mulEEG.

Referências

João Batista Florindo. Máquinas de vetores de suporte (apresentação de slides), 2022. URL florindo@unicamp.br.

João Batista Florindo. Redes convolucionais clássicas (apresentação de slides), 2023. URL florindo@unicamp.br.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Bob Kemp. Sleep-edf database expanded, Out. 2013. URL https://www.physionet.org/content/sleep-edfx/1.0.0/.

Vamsi Kumar, Likith Reddy, Shivam Kumar Sharma, Kamalaker Dadi, Chiranjeevi Yarra, Raju S. Bapi, and Srijithesh Rajendran. muleeg: A multi-view representation learning on eeg signals. *Lecture Notes in Computer Science*, 13433, Set. 2022. URL https://doi.org/10.1007/978-3-031-16437-8_38.

Aakash K. Patel, Vamsi Reddy, Karlie R. Shumway, and John F. Araujo. Physiology, sleep stages, Jan. 2024. URL https://www.ncbi.nlm.nih.gov/books/NBK526132/#:~:text=Sleep%20occurs%20in%20five%20stages,leading%20to%20progressively%20deeper%20sleep.

Vinicius Segatto. Svm, ou support vector machine, Out. 2023. URL https://medium.com/liga-mackenzie-de-ia-ci%C3%AAncia-de-dados/svm-ou-support-vector-machine-7efcabdcc7be.

Ekin Tiu. Understanding contrastive learning. Jan. 2021. URL https://towardsdatascience.com/understanding-contrastive-learning-d5b19fd96607.