Automatic Sleep Stage Classification with CNN and Seq2Seq Models

https://www.youtube.com/watch?v=L5BjQNSeVqA

https://github.com/atbasu/DL4H_1433_Experiments_With_Sleep

Alan Lekah
alekah2@illinois.edu

David John
davidj4@illinois.edu

Trevor Overfelt
trevoro2@illinois.edu

Atri Basu
atrib3@illinois.edu

1. ABSTRACT

Manual sleep stage scoring using electroencephalogram (EEG) data is a time consuming task, requiring a large amount of medical resources for labeling and classifying signals to diagnose sleep disorders. In this paper we describe an attempt to recreate, experimentally, the SleepEEGNet model proposed in [4]. Our goal was to build a customised version of each component of the original architecture and analyse the performance. We hoped to arrive at a deeper understanding of the reasons for why the architecture described in the original paper learns as well as it does. Our model was built using the data preprocessing and deep learning approach found in the SleepEEG study, which utilized a coupled pair of CNN models with multiple layers using different kernel sizes to extract and concatenate time series and frequency level information from EEG data processed into 30 second epochs. What was implemented was a translated architecture written in PyTorch (rather than TensorFlow) that attempts to utilize the CNN architecture found in the base model with a simpler sequence-to-sequence architecture to replicate the results of the original study.

2. INTRODUCTION

2.1 Problem Statement and Background

Quality sleep plays a vital role in proper mental and physical health, quality of life, and safety of an individual. Sleep deficiency is a very common public health problem, and an estimated 50-70 million Americans have chronic sleep disorders [5]. Reduced or abnormal sleep can lead to several physical and mental risk factors including injury/accidents, depression, and heart disease, making it a very important topic for medical research. Sleep stage scoring is the process of monitoring Electroencephalogram (EEG) signals to determine brain activity and diagnose sleep disorders.

Manual sleep quality classification using experts and domain knowledge is a very labor-intensive and time-consuming task, which requires a large amount of medical resources for recording, labeling, and classifying each epoch of data. In addition, the scoring procedure used by human experts is based on generalized guidelines and is no doubt prone to error. For these reasons, automated sleep stage classification has become a large area of study for deep learning applications.

Deep learning models are able to handle large amounts of data and form a final complete classifier using patterns learned from batches and sequences of information presented during training. Deep learning models are also able to learn and automatically determine the important underlying features from sleep signal and patient demographic information, reducing the need for costly and complex feature engineering and prior knowledge of EEG analysis. Convolution Neural Networks in particular have been widely used and highly successful in characterising the latent features in image data (and more recently, biosignals including electrocardiography and electroencephalography) without utilizing prior domain knowledge. The integration of automated feature extraction with other classification algorithms can be used to more accurately recognize objects and predict classifications of raw data input.

Sleep is a cyclical process encompassing several phases in either REM or Non-REM sleep, and multiple cycles are performed within a night. Because of this sequential nature, we are choosing to follow the SleepEEGNet method of sequence-to-sequence classification. By taking this sequential structure into consideration, we can correlate different stages of sleep to capture nonlinear dependencies present in the time series to enhance the accuracy of the scoring process [6]. In this paper, we will utilize publicly available sleep sensor datasets and apply a sequence-to-sequence deep learning model in addition to a convolution neural network to achieve automatic sleep stage classification. Our custom model will be evaluated against several existing classifiers that either utilize only CNN architectures, or those that employ more complex sequence-to-sequence architectures to capture time-series information between sleep stage epochs.

Our initial results show that even by utilizing a combination of these models to automatically learn latent features from our dataset and classify these sequences, there are many important components necessary due to issues within the dataset. Without proper class balancing, certain sleep stages can become overrepresented in our input data (the number of W and N2 sleep stages are much higher than any of the other stages), causing bias

within the model. The base SleepEEG model utilized both data oversampling and a novel loss function to treat the error of each misclassified sample equally regardless of being a member of the majority or minority class. The translation efforts from TensorFlow to Pytorch also presented challenges when attempting to recreate the layer architectures, resulting in the extraction of features that did not enable proper learning. Because of these limitations, several modifications will be required to improve the performance of this custom classifier.

2.2 Related Works

The model architecture created for this project is based on the research found in the SleepEEGNet implementation [4]. The highly customized model in this research combined convolutional neural networks to extract feature data from EEG signals and sequence-to-sequence encoder/decoders with attention to automatically produce learnable features and classify the data into separate sleep stages. This model also utilized several mechanisms to handle class imbalance within the reference dataset such as novel loss calculation and oversampling/undersampling of the data for training and evaluation. Compared to other state of the art models for classifying sleep stage and sleep scoring, this model shows some of the best class-based performance and metric scoring, making it a useful model for reference and possible optimization.

Other works utilize the same sequence-to-sequence architecture with multi-channel EEG signal input to capture and encode feature information from EEG, EOG, and EMG sleep data. The architecture found in [8] uses parallel filterbank layers in preprocessing to learn channel-specific frequency-domain data, as well as perform dimensionality reduction and frequency smoothing. The learned filterbank is expected to emphasize the subbands that are more important for the task at hand and attenuate those that are less important, emulating an attention mechanism before the sequence-to-sequence modeling. Epoch-wise and sequence-wise RNNs are then tied together using another epoch-wise attention layer to create a single feature vector at different time steps to emphasize those parts of the sequence that are most important to sleep stage classification.

Some implementations extend the sequence-to-sequence modeling and instead utilize a transformer model to bypass the limitations on classifier complexity and data dimensionality when training using large datasets. For the model proposed in [9], two stages are used - one performs downsampling and processes the raw input data to create a maximally consistent representation of EEG sequences across all datasets used, and the other utilizes a transformer encoder to map the input to a new sequence that embodies the correct latent features. The proposed model has the benefit of handling large amounts of multi-signal data, and can perform accurate classification with unlabeled input data from multiple sleep datasets; however, the use of transformers and self-attention when applied to EEG data is not well studied, and may require significant pre-training for the model to learn appropriate features.

Different algorithms, like the simpler convolutional neural network found in [12] were also considered and referenced for their layer structure and filter kernel sizing. The architecture in this literature uses non-preprocessed raw signal input data. Signal data is normalized and passed through a pair of convolutional neural networks before being passed through a set of fully connected layers to produce the final output. The unique mechanism used in this configuration is a “stacking” layer that combines the output of the first CNN layer into a 2D stack of filtered and sub-sampled signals that is then fed to the second convolutional layer, with the purpose of capturing the relationships across the filtered signals for a specific time window.

3. METHOD

3.1 Dataset and Preprocessing

For this project, we have utilized the Sleep-EDF database (expanded). This dataset contains 197 whole night PolySomnoGraphic (PSG) sleep recordings containing EEG, EOG, chin EMG, event markers, respiration, body temperature and their corresponding hypnograms (sleep patterns). The PSG data contains signal information sampled at a rate of 100Hz with hypnograms manually labeled by sleep scoring experts. There are five types of sleep files contained in the dataset; PSG.edf files contain the EEG, EOG, EMG, and event markers. The SC*PSG.edf file contains the oro nasal respiration and rectal body temperature. The Hypnogram.edf files contains the annotations and the SC* (sleep cassette) files. There are also ST files (sleep telemetry) which contain patients who either slept without or without temazepam intake. The annotations within the *Hyponogram.edf files contain annotations of sleep stages W, R, 1, 2, 3, 4, M (movement time), and ? (not scored).

We began our implementation by replicating the data preprocessing steps found in the SleepEEG study for automated sleep stage classification. We utilized a publicly available Github repository published by Sajad Mousavi[2] which enabled us to pre-process the 2018 data found in the Sleep-EDF expanded dataset. After running the appropriate preprocessing methods, a dataset package was uploaded to a Google Drive shared folder for use by the team. This shared folder was then added to each member of the team’s “local drive”; there, the zipfile with each of the EEG and corresponding labels in the form of npz (numpy) files were then mounted in the Google Colab notebook. After completing this step, we were able to extract the zip file and use the built in memory of the notebook to store the data. To decrease the time necessary for both training and evaluation of our data, we enabled native GPU support within the notebook to support the PyTorch GPU enabled model training.

3.2 Model Training and Evaluation Setup

We created a list of classes associated with each sleep stage (wake, rapid eye movement, N1-N4, M), as well as those which were not scored. Per the pre-processing found in the base model, N3/N4 were combined, and unlabeled/unscored classes were removed, leaving a total of 5 different classes. The input for our pre-processing methods are sequences of 30s EEG epoch signal data, which were segmented into sequences of certain time step length based on hyperparameter selection, and normalized such that each segment had zero mean and unit variance. Each sequence was then scored based on an annotation file with expert

labels. Finally, this pre-processed data was loaded into a custom dataset implementation and data loader that utilized a custom collate function and set batch size to separate the data. Data loaders were separated for training and validation sets to provide capability for K-fold cross validation.

The implementation provided in the base model allowed us to easily split the dataset between training and test sets with subjects and samples being chosen randomly from the available data. The model was then evaluated using K-fold cross validation where the data was split (our model utilized 20 folds which was recommended by the original implementation) with one unique fold taken as the test set and the remaining folds taken as the training set. During evaluation, each iteration sampled the dataset to get a new list of subjects used to compare model output and compute the confusion matrix. After the K-fold evaluation, all metrics were combined to give the average scores of all iterations. The input for our deep learning models were vectors with size [*batch size, sequence length, num_examples*], with the number of examples for test and validation being extracted using the aforementioned pre-processing steps, and the sequence number and batch size being selected as hyperparameters for our model architecture (default 20 batch size and sequence length of 10).

3.3 Model Architecture

3.3.1 Convolutional Neural Network

The first part of our model was a CNN architecture similar to that described in the SleepEEG sequence to sequence network[2] . The model consists of two separate CNNs, which are run together using the same input data before outputs for both models are combined for later processing. The first CNN uses smaller filter kernels in an attempt to extract temporal information from the data, and the second uses large filter kernels to extract frequency-level information. Each CNN in this model consists of four separate conv1d functions followed by ReLu nonlinear activation. The original model was written using TensorFlow, and had access to some advanced functionality that is not available in PyTorch. For each convolution and max pooling operation, the image is padded evenly on all dimensions to force the layer's outputs to have the same spatial dimensions as the input using TensorFlow's "SAME" parameter. To achieve similar functionality, we needed to implement custom padding functions that calculated the padding length of each dimension based on the input/output size, kernel size, and stride. This padding was then added whenever performing convolution and max pooling operations on the sequence of signal data. After all convolutions, max poolings, and activation operations were complete for both the large and small kernel CNNs, the data was concatenated over the last dimension and sent to a final dropout layer to produce the complete CNN output - a new vector of size [*batch size, sequence length, num_features*].

3.3.2 Sequence-to-Sequence Encoder

The results of the CNN layers, which attempted to automatically identify the features found in our EEG signals, were then passed to a simplified sequence-to-sequence auto-encoder architecture, which utilized the sequence structure of the data to

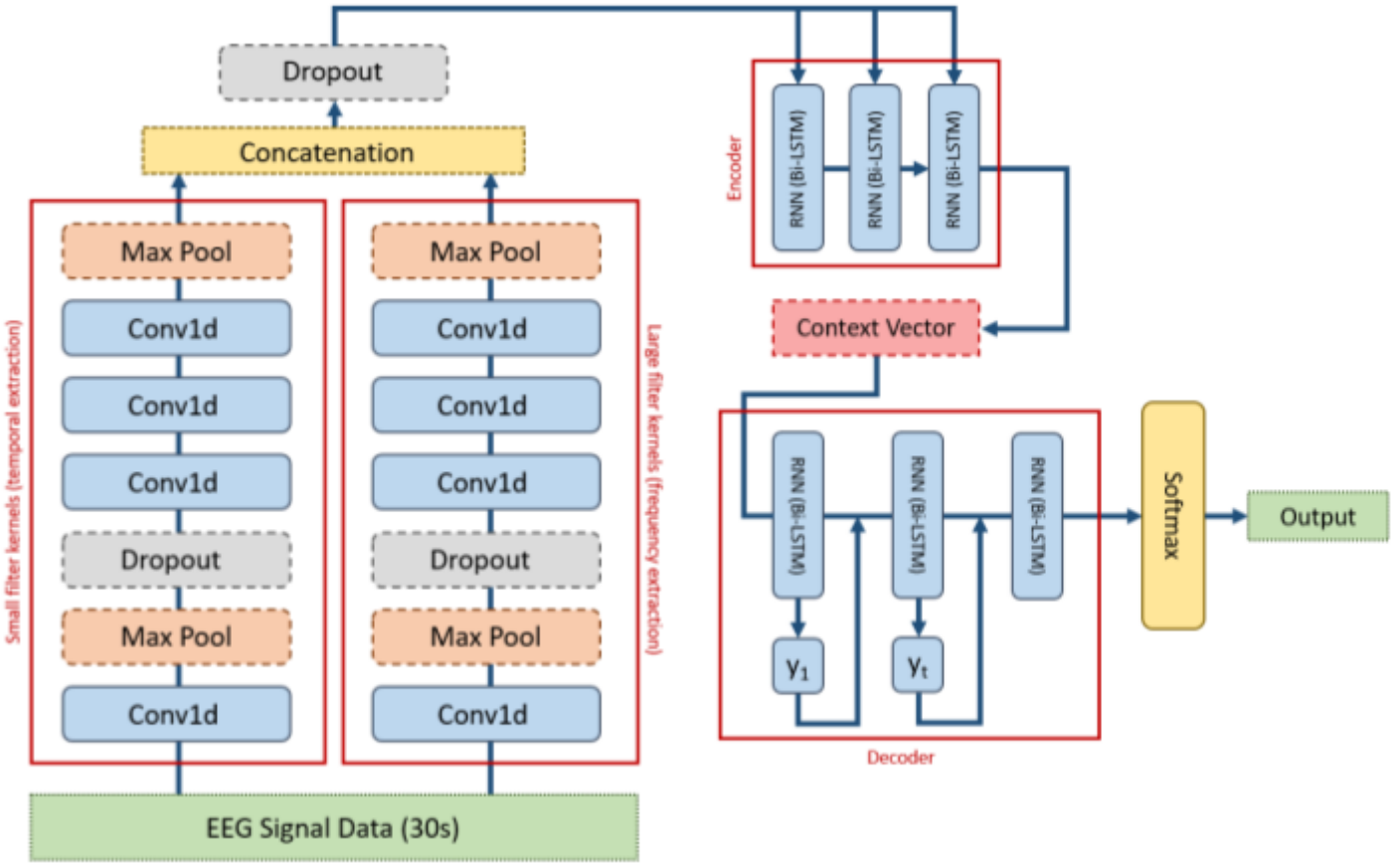

Figure 1: Complete Architecture with Convolutional Neural Network and Sequence-to-Sequence encoder/decoder

create learnable vectors before final label classification. For this architecture we build a complete parent model structure containing the CNN, encoder, and decoder. The output of the CNN layers were passed to the encoder, which created a vector containing the laten representation of the input sequence that could be passed to our RNN layer. For this we implemented a 10-layer bi-directional RNN using PyTorch Bi-LSTM. The final dimensions of this RNN output were then flattened to produce a two-dimensional vector to represent our embedding of the entire sequence for the batch.

3.3.3 Sequence-to-Sequence Decoder

To construct the decoder, we first needed to create the first state input, which was a sequence of all zeros built in the shape of our desired decoded output [*batch size, num_classes*]. Our first state and encoder output were then passed to our decoder model, which reshaped our inputs to the appropriate size, and passed them through another Bi-LSTM to reconstruct the vector with the correct values. The output of our RNN layer was then passed through multiple linear fully-connected layers and ReLu non-linear activations. The resulting vector was returned to our parent model, and concatenated to a list. This decoding process was completed for each encoded representation in the sequence, using the previous hidden state and reconstructed vector output as parameters for the newly decoded token. Once the sequence was decoded, the resulting vectors were stacked together, and passed through a final Softmax layer to produce the probabilities for each available class. During evaluation, we utilize the PyTorch argmax() function to identify the class with the highest probability, and compare these to the ground truth class labels to calculate our performance metrics.

3.4 Model Evaluation

The evaluation metrics followed the same calculations performed in the base SleepEEG model. First, a confusion matrix was generated to provide the true positive, true negative, false positive, and false negative instances and indicate the number of sleep stages that were correctly or incorrectly labeled. These calculations were then used to generate a set of standard metrics based on the average value of the predictions calculated among all folds. Accuracy, precision, recall, specificity, as well as F1 score were used during experimentation to show the efficacy of our model for automatically classifying sleep stage scores from single-channel EEG data. These evaluations were completed over a certain number of folds to provide support for cross-validation (the number of folds was determined as a hyperparameter for our model), and the final scores were averaged to produce our model performance. For final model evaluation, we chose to only utilize the precision, recall, specificity, and F1 scores to match the class-based performance metrics shown in the reference SleepEEGNet architecture [4]. The results published in this research also compare the performance of the proposed architecture to other state-of-the-art implementations, allowing us to compare our results with a variety of model architectures.

4. EXPERIMENTAL RESULTS

After following the preprocessing steps outlined in the above sections, our dataset consists of 153 separate signal data files. To simplify the generation of our initial metrics and to reduce the time needed to train and evaluate our model while adjusting layers and hyperparameters, all metrics discussed below were generated using a subset of this data. The first 10 signal data files were loaded into our dataset and used for training and evaluation of our model.

To begin our experiment in creating a sleep stage scoring neural network and generate initial performance metrics, we implemented our solution in parts. As described above, the first section of our model consisted of a CNN architecture that utilized two different CNNs to extract time series and frequency level information. After creating our beginning model with only a CNN, followed by a series of fully-connected linear layers and softmax probability generation, and training our model for 10 epochs, the performance we were able to achieve was as follows:

	W	N1	N2	N3	REM
Acc	0.50	0.70	0.70	0.80	0.80
F1	0.40	0.20	0.20	0.10	0.10
Sens	0.30	0.30	0.20	0.20	0.20
Spec	0.70	0.80	0.80	0.80	0.90
PPV	0.50	0.10	0.30	0.10	0.10

Table 1: CNN-only architecture class-based performance metrics

Average Validation Accuracy: 0.6947916
F1 Macro: 0.20117024

However, looking at our loss distribution across the epochs, it is evident that our current model isn't learning. The poor training performance of our CNN could be due to an incorrect implementation in the layer architecture resulting in an inability to discover a set of appropriate and learnable features from our singal data, or the lack of the novel loss function found in the original SleepEEG implementation, which utilized a modified version of Mean Squared False Error (MSFE) to handle the issues with class imbalance within the dataset:

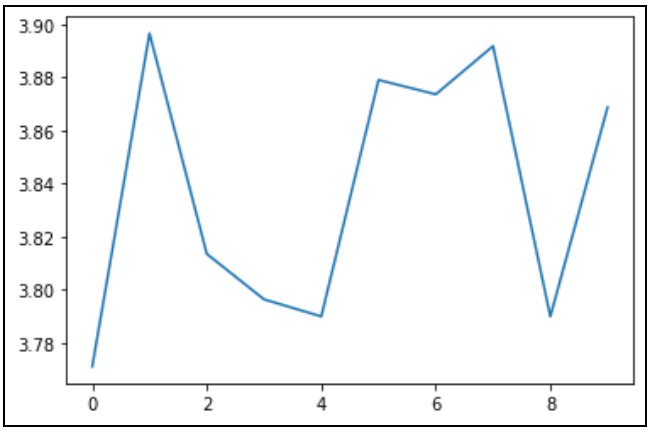

Figure 2: CNN-only architecture training loss across epochs

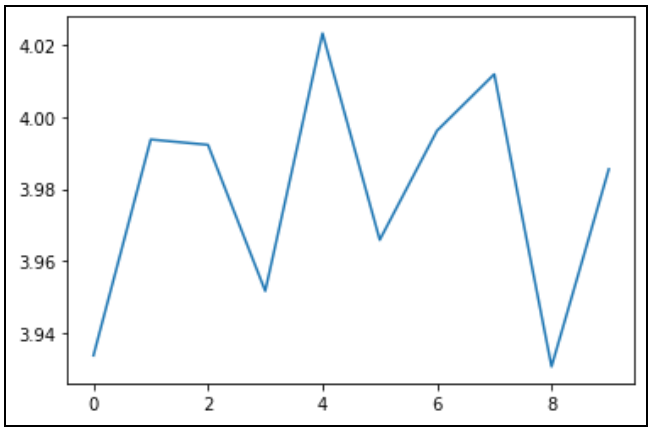

Figure 3: CNN + Auto-encoder architecture training loss across epochs

Next, we attempted the implementation of a simple sequence-to-sequence auto-encoder architecture, and attempted to combine it with the previously built CNN into a holistic model (the architectural design of which is referenced in the sections above). The output of our new stacked architecture, which included feature extraction using a multi-layer CNN and sequence encoding/decoding using a sequence-to-sequence auto-encoder and softmax classification was as follows:

	W	N1	N2	N3	REM
Acc	0.50	0.70	0.70	0.80	0.70
F1	0.30	0.20	0.20	0.10	0.20
Sens	0.30	0.30	0.20	0.10	0.20
Spec	0.80	0.80	0.80	0.90	0.80
PPV	0.50	0.10	0.30	0.10	0.10

Table 2: CNN + Auto-encoder architecture class-based performance metrics

Average Validation Accuracy: 0.693125

F1 Macro: 0.20242676

While the architecture was more closely aligned with the original SleepEEG framework, this did not seem to benefit overall performance. Our accuracy seemed to decrease slightly and the distribution of our loss functions across the epochs became even more erratic:

A major problem encountered with the attempted implementation of our simple auto-encoder was the failure to tokenize and decode each sample within the sequence. Instead, the decoder would reconstruct the entire sequence at once, failing to gain the benefit of the sequential nature of our sleep epochs or utilize previous decoded hidden states when reconstructing later sleep vectors. A separate attempt was made to build a sequence-to-sequence model using a transformer and self attention (the attempt and the code can be found here). Due to time constraints we were unable to get a fully working model using self-attention.

A final attempt was made to resolve the architectural errors of our sequence-to-sequence model in an attempt to improve classification performance. First, we utilized a trivial mechanism to embed the output of our CNN to allow for more effective feature discovery and enable proper learning within our encoder. The latent features discovered in our encoder were then sent to our decoder in a sequential fashion, passing a single token from the sequence of samples at a time, and using the new hidden state and predictions of our current decoding pass as input to the next sequence. These predictions were then stacked to create the final vector used for softmax classification:

	W	N1	N2	N3	REM
Acc	0.50	0.70	0.70	0.70	0.70
F1	0.30	0.10	0.00	0.10	0.20
Sens	0.20	0.20	0.00	0.20	0.30
Spec	0.80	0.80	1.00	0.70	0.70
PPV	0.50	0.10	0.10	0.10	0.10

Table 3: CNN + Auto-encoder architecture with trivial embedding class-based performance metrics

Average Validation Accuracy: 0.6714584

F1 Macro: 0.1459755

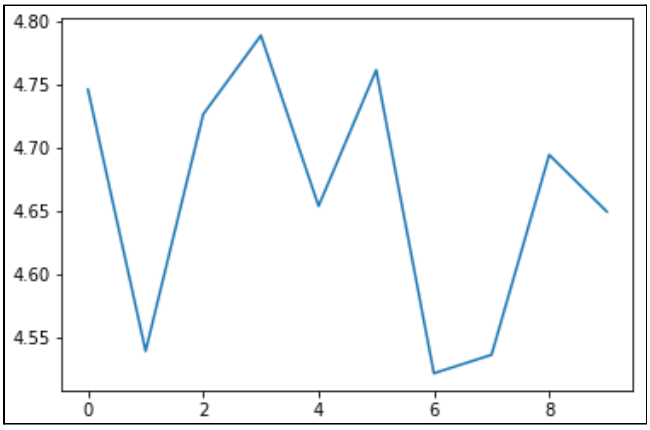

Figure 4: CNN + Auto-encoder architecture with trivial embedding training loss across epochs

While it seemed that the learning performance of our latest iteration might be improving (based on a small number of epochs), it was clear that our performance did not improve overall. The trivial embedding mechanism that was used resulted in highly biased classification, with some classes being excluded entirely from the prediction results. The preliminary results of our final model did not show improved performance when compared against competing classifiers, and it did not reach the classification accuracy of the highly customized SleepEEG reference model (class-based performance referenced below) [4].

	W	N1	N2	N3	REM
Prec	0.88	.50	.91	.82	.82
Recall	0.91	0.55	0.83	0.89	0.89
Spec	0.97	0.96	0.94	0.97	0.96
F1	0.89	0.52	0.87	0.85	0.85

Table 4: SleepEEG reference model class-based performance metrics

For our final iteration we decided to rework the structure of our encoding/decoding layers to more closely resemble the reference model from SleepEEGNet. The simple GRU RNN used in the previous version was replaced with a functional bi-directional LSTM unit which utilized the previous hidden state in addition to the output of the previously decoded sequence token. We also adjusted our loss calculation to use Negative Log Likelihood (NLL) instead of mean squared error (MSE) loss. NLL is traditionally applied to models with softmax function output and leads the model to being more certain about correct predictions by punishing the model when smaller probabilities are used in the final classification.

We found that utilizing the trivial embedding mechanism discussed in the previous section led to only a few output classes being predicted. While this resulted in decent accuracy when the sample data was biased toward the predicted classes, overall performance was poor. The trivial embedding was removed and raw data output from the CNN was used as input to the encoder/decoder. The changes made in this iteration showed some improvement to training performance for a small number of epochs, but overall performance did not match that of the reference model. While the average validation accuracy had improved, we had failed to predict an entire class (REM stage), causing issues with our prediction metrics and overall average F1 scoring for the model. For this iteration we also adjusted the metrics used to get a better comparison of class-based performance to the above reference architecture.

	W	N1	N2	N3	REM
Prec	0.20	0.21	.37	.08	N/A
Recall	0.04	0.55	0.43	0.03	0.00
Spec	0.98	0.53	0.52	0.96	100
F1	0.06	0.30	0.40	100	N/A

Table 5: Final Custom CNN plus Sequence-to-Sequence model training class-based performance metrics

Average Validation Accuracy: 0.7125231

F1 Macro: N/A - unable to compute due to failure of predicting REM class

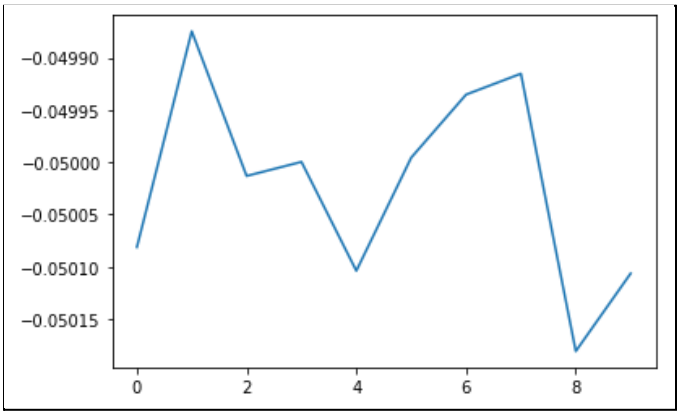

Figure 5: Final Custom CNN plus Sequence-to-Sequence model training training loss across epochs

5. DISCUSSION

A major difficulty we had with getting good performance from our model was the translation required from the highly customized sequence-to-sequence model used in the SleepEEG project. The data pre-processing produced specific dimensions for our batched input which required a specific CNN architecture to produce the appropriate features. The EEG signal data was first processed into 30 second epochs, which were randomly selected from the dataset and loaded into sequences of data defined by a maximum time step, up to a certain number of samples. These multi-dimensional sequences then required embedding, encoding, and decoding before being passed to the final softmax classification. Because the reference architecture was so specific to the input dimensions, we found it difficult to customize the network layers or change hyperparameters to improve the performance. In order to properly handle the shape of our preprocessed data input, we needed to constantly reshape and restructure our layer outputs such that they would pass through the various model layers to produce the expected dimensions of our output label sequence. In addition, the functional differences between TensorFlow and PyTorch made it difficult to accurately translate the model layers, ultimately resulting in our model creating non-learnable features, shown by the lack of improvement in model performance over several training and evaluation epochs.

In the absence of a fully functioning and feature-rich sequence-to-sequence encoder/decoder with appropriate attention mechanisms, we were forced to build a much more trivial implementation that was quite different from the SleepEEG reference model. Initially, we built our CNN model with the layers described in the SleepEEG dataset, which produced an output of concatenated frequency-level and time-series level features. In the reference model, these features are to be encoded, and run through a bi-directional LTSM which extracts the important features contributing to sleep stages across epochs within the sequence. A final embedding is created using the decoder, which is then passed to a softmax function to provide the probabilistic distribution of each available class. The resulting concatenated features extracted from the CNN in our custom model were fed through the simple implementation of our auto-encoder, much like the original model; however, we did not enhance this sequence-to-sequence classifier with attention mechanisms, and we used a simple implementation of the PyTorch Bi-LSTM rather than a more complete memory network. We found that the lack of attention mechanism and inability to create a customized sequence-to-sequence implementation significantly impacted the accuracy of our model. Our CNN was unable to extract useful features from the EEG signals, and our auto-encoder network was unable to learn the time-series information from the model input to appropriately classify the data.

We also encountered several issues due to class imbalance within our dataset. Because some sleep stages are highly overrepresented within the initial dataset, the SleepEEG implementation uses oversampling to get an equal distribution of samples to use for training and evaluation of the model. In addition, the SleepEEG model also employs the use of a customized loss function based on mean squared false error with L2 regularization to reduce overfitting. Our model did not include steps to create an even distribution of data samples, and utilizes simple mean squared error found within the PyTorch library to calculate loss during each epoch, which did not provide an increase in the classification performance, and produced a high amount of loss that did not directly decrease throughout training.

Though we managed to improve our validation accuracy and performance metrics throughout our model iterations, the overall performance compared with the reference architecture was not comparable. Our final model failed to predict one of the designated output classes (REM stage) causing inconsistency in the metric calculations. We also noticed that performance would vary greatly based on the subjects that were sampled from the database, possibly due to the imbalance between classes that was not addressed or some problematic features that were collected from the CNN or sequence-to-sequence layers.

6. CONCLUSION / OPTIMIZATION

6.1 Conclusion / Closing remarks

In this paper we presented a simplified model based on the SleepEEG sequence-to-sequence classifier for automatic sleep stage scoring of single channel EEG data. Based on our results, it seems evident that simply passing the input data through convolutional and sequence-to-sequence layers doesn't properly encode the relationship between the EEG signals and the sleep cycle stages, and simply stacking any auto-encoder onto the initial CNN implementation does not have any positive impact on the model's ability to learn and train end-to-end. It seems that the key to a successful neural network implementation is figuring out exactly which kind of transformations are likely to discern the most relevant patterns given a structured set of input data. A deeper investigation into understanding the properties of the data preprocessed by the base model and how the structure of the input data can be utilized to extract relevant features and determine the latent representation for sequence learning would be a valuable step forward in gaining a deeper understanding of how to effectively apply neural networks to our problem statement.

A large improvement could be also made to the model if steps were taken to handle the uneven class distribution in our sleep dataset, either through oversampling or class-balanced random sampling for each batch of data. Furthermore, as mentioned previously, while building our model we had opted to use the standard Mean Squared Error as our loss function. Upon further investigation[6] we found that MSE is great when the "the distribution of the target variable is Gaussian". Looking at the confusion matrix, it's clear that the sample distribution across our classes is not gaussian. This is also exemplified by the lack of accuracy in the largest class(W). One of the optimizations worth exploring are variations of the MSE loss function including the one mentioned in the original paper[4].

6.2 Learning Take-aways

While attempting to recreate this high-performing and highly-customized reference model, we learned that the architecture of the model is highly important in determining learning performance. The devil's in the details. Incorrect layer

structure, loss function decisions, or dataset splitting may lead to inappropriately learned features, which impacts overall classification accuracy. Proper consideration and understanding of the required preprocessing steps should also be incorporated. Incorrect structure of the input data leads to difficulty with size and shape as the data is passed and transformed through the model layers - this only becomes more apparent as multiple models are concatenated together to form an end-to-end deep learning network. This end-to-end network is difficult to create, but extremely powerful in terms of classification accuracy. While building, training, and evaluating this large and complex model structure, it is important to validate that the gradient is being fully back-propagated to ensure proper learning of the network throughout the layers. Given the challenges faced, we believe this model will require several additional iterations to resolve the issues outlined above and create a complete end-to-end network that can accurately classify sleep stages from raw signal data input

7. REFERENCES

[1] *Google Colaboratory*. (n.d.). https://colab.research.google.com/github/tensorflow/examples/blob/master/courses/udacity_intro_to_tensorflow_for_deep_learning/l01c01_introduction_to_colab_and_python.ipynb.

[2] MousaviSajad. (n.d.). *MousaviSajad/SleepEEGNet*. GitHub. https://github.com/MousaviSajad/SleepEEGNet.

[3] Kemp, B. (2013, October 24). *Sleep-EDF Database Expanded*. Sleep-EDF Database Expanded v1.0.0. https://physionet.org/content/sleep-edfx/1.0.0/.

[4] Mousavi, S. (n.d.). SleepEEGNet: Automated Sleep Stage Scoring with Sequence to Sequence Deep Learning Approach. https://arxiv.org/pdf/1903.02108.pdf.

[5] Reviewers, A. (2021). Sleep Statistics - Data About Sleep and Sleep Disorders. American Sleep Association. Retrieved 30 March 2021, from https://www.sleepassociation.org/about-sleep/sleep-statistics/

[6] Brownlee, J. (2019, January 30). How to Choose Loss Functions When Training Deep Learning Neural Networks https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

[7] Ahnaf Rashik Hassan, Abdulhamit Subasi. (2017). A decision support system for automated identification of sleep stages from single-channel EEG signals. Retrieved 30 March 2021, from https://doi.org/10.1016/j.knosys.2017.05.005

[8] Phan, H., Andreotti, F., Cooray, N., Chén, O., & De Vos, M. (2019). *SeqSleepNet: End-to-End Hierarchical Recurrent Neural Network for Sequence-to-Sequence Automatic Sleep Staging*. arxiv.org. Retrieved 30 March 2021, from https://arxiv.org/abs/1809.10932.

[9] Kostas, D., Aroca-Ouellette, S., & Rudzicz, F. (2021). *BENDR: using transformers and a contrastive self-supervised learning task to learn from massive amounts of EEG data*. arXiv.org. Retrieved 30 March 2021, from https://arxiv.org/abs/2101.12037.

[10] Olesen, A., Jennum, P., Mignot, E., & Sorensen, H. (2020). *Automatic sleep stage classification with deep residual networks in a mixed-cohort setting*. arvix.org. Retrieved 30 March 2021, from https://arxiv.org/abs/2008.09416.

[11] Zhang, J., & Wu, Y. (2017). *A New Method for Automatic Sleep Stage Classification*. Ieeexplore.ieee.org. Retrieved 30 March 2021, from https://ieeexplore.ieee.org/document/8010320.

[12] Tsinalis, O., Matthews, P., Guo, Y., & Zafeiriou, S. (2016, October 05). *Automatic sleep STAGE scoring with SINGLE-CHANNEL EEG using convolutional neural networks*. Retrieved May 04, 2021, from https://arxiv.org/abs/1610.01683

[13] Mansar, Y. (2018). *Sleep Stage Classification from Single Channel EEG using Convolutional Neural Networks*. towardsdatascience. Retrieved 30 March 2021, from https://towardsdatascience.com/sleep-stage-classification-from-single-channel-eeg-using-convolutional-neural-networks-5 c710d92d38e.