Investigating the effectiveness of AI deep learning approaches toward the analysis and categorization of biosignal time-series data

Research Question: To what extent are deep learning algorithms such as Convolutional Neural Networks and Recurrent Neural Networks effective in the accurate interpretation and classification of biosignal time-series data?

Topic: Deep Learning Algorithms and Biosignal Classification

Subject: IB DP Computer Science

Session: May 2025

Word Count: 3986

Table of Contents

1 Introduction:

In recent years, deep learning has seen an explosive rise in successful applications, which span areas including computer vision, natural language processing, predictive analytics, and fraud detection (Holdsworth; Goodfellow). However, the ideas that underpin the conceptual foundations of deep learning are less novel than one may presume, with surges of academic interest in neural networks appearing during the 1960s and the 1980s to 1990s, though at the time these algorithms were considered to be arduous to train and implement in digital systems (Goodfellow). Due to a novel series of breakthroughs following the mid-2000s such as greedy layer-wise pretraining proposed by Geoffrey Hinton, alongside developments in hardware, neural networks could be scaled immensely, enabling what is now referred to as ‘deep’ learning (Goodfellow).

The method used to train deep neural networks are not unique, like classical neural networks they iterate through a cycle of forward propagation, where data is passed forward through the model by a series of weights and connections, and backpropagation, an algorithm which calculates the gradient of the loss function in respect to the model’s weights (Holdsworth). Gradient-descent-based algorithms use the values found by backpropagation to adjust the weights and other parameters of the network, decreasing the value of the loss function and thus obtaining a superior model by the next iteration (Holdsworth). This iterative cycle of improvement substitutes the need for explicitly programmed instructions, especially when these are difficult to define. The tradeoff is that the quantity of training data available strongly influences the network’s capacity to learn a task.

A rising frontier for deep learning over the last five years has been the processing of biosignal data for mental and physical health (Sajno). A biosignal is any data stream supplied by a sensor monitoring a human, the most common including ECG (electrocardiogram), PPG (photoplethysmography), RESP (respiration) and EDA (electrodermal activity), which each convey information about a subject’s autonomic and central nervous system alongside other biological activity (Cowley 157; 166).

Biologists and radiologists observe these signals to assess organs, tissues, and protein sequences, which can then be used by doctors to diagnose conditions and track patients throughout different treatment phases (Swapna). They also serve as useful indicators of a subject's general health profile and fitness (Swapna).

The field of machine learning provides powerful tools to aid the analysis of biosignal data, but this data must often be heavily pre-processed, or have certain features of the raw data selected and extracted (Sajno). A major advantage of deep learning is that it can perform effectively given raw data directly or with minimal pre-processing, enabling simpler end-to-end frameworks (Sajno). The two most prevalent deep learning architectures specialized for signal processing include convolutional neural networks (CNNs), and recurrent neural networks (RNNs) (Holdsworth). CNNs primarily use two types of layers, convolutional layers and pooling layers, which process information while improving efficiency, removing model complexity and mitigating the risk of overfitting to the training data, but result in a high amount of hyperparameters that must be tuned; on the other hand, RNNs can easily adapt to signals of any length, but without specialized options such as long-short term memory (LSTM) suffer from an issue during training known as the exploding and vanishing gradient problem (Holdsworth).

This study aims to explore these two deep learning architectures within the application of biosignal processing by examining their performance results on the WESAD dataset, which uses biosignals as inputs to predict the affect state of a subject, such as stress and amusement (Schmidt). By investigating the effectiveness of CNNs and RNNs through metrics such as predictive accuracy, training stability, and computational efficiency, this study evaluates the accuracy and viability of current deep learning approaches in the context of psychophysiological and medical applications.

The research question is as follows: To what extent are deep learning algorithms such as Convolutional Neural Networks and Recurrent Neural Networks effective in the accurate interpretation and classification of biosignal time-series data?

2 Background Research:

2.1 Types of Biosignals

The task to be approached by CNNs and RNNs involves using multiple modes of biosignal data to predict a subject's affect state, essentially, one's basic emotional state (Giannakakis). A general understanding of the main types of biosignals used can clarify the theoretical relationship between the input and expected output while hinting towards the patterns and information that we are expecting the models to find within the raw data.

For instance, electrocardiogram (ECG) indicates the electrical activity and rhythm of the heart, from which other signals such as heart rate and heart rate variability can be extracted. In broader medical applications they can detect cardiovascular issues and structural abnormalities in the heart, but increased activity in the ECG may also indicate temporary states of stress or excitement (Dahal; Cowley 169).

Electrodermal activity (EDA) records changes in the electrical conductivity and resistance of the skin, usually caused by the activation of sweat glands (Cowley 176-177). This allows EDA to predict emotional and mental activity, but would also be influenced heavily by physical activity such as exercise (Cowley 176-177).

Electromyography (EMG) measures the electrical activity relating to the contractions of muscles attached to the skeleton, which can then detect the contraction of facial muscles for affect assessment (Cowley 191). Blood volume pulse (BVP) is the data contained within the photoplethysmography (PPG) signal, but the terms are used interchangeably to indirectly predict a subject's blood pressure through unobtrusive means (Cowley 170).

Other signals, such as respiration (RESP) and body temperature (TEMP) may be less powerful individually in the deduction of a subject's affected state, but can provide useful supplementary information to be used in conjunction with other biosignals (Cowley 161). A commonality

between the signals discussed is that they are recorded in time-series: a sequence of many data points recorded over time (Cowley 170).

The most effective deep learning models will efficiently leverage and combine useful information from a variety of time-series signals to create the most accurate predictions.

2.2 Convolutional Neural Networks

Convolutional networks are specialized neural networks that were developed to process data with a grid-like topology, including time-series based signals, which are essentially 1D grids, and more prevalently images, which are arranged in a 2D grid of pixel values (Goodfellow 326). Forward propagation of data in a CNN is done via two main operations: convolution layers and pooling layers (Goodfellow 326).

The convolutional operation applies the weights of a small kernel repeatedly across a larger input to produce an output feature map (Goodfellow 328). The kernel slides over the input grid, such that at each position, the kernel's values are multiplied to the corresponding values in the input, and summed to produce the value at that position in the output (What Are Convolutional Neural Networks?). The convolution operation that produces feature map S is defined for 2D layers as below:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i + m, j + n)K(m,n),$$

where *(K * I)* denotes that kernel K has been applied to the input I (Goodfellow 329). In the processing of time-series biosignal data, we primarily consider the 1-dimensional convolutional operation, which follows the same principle but would be defined using the modified formula:

$$S(i) = (K * I)(i) = \sum_m I(i + m)K(m)$$

There are several benefits to using convolutional operations over standard fully connected layers, including parameter sharing, where one parameter can serve multiple computations in the model, as well as sparse connections, where every node in a layer no longer needs a corresponding weight or connection to every node in the next layer (Goodfellow 331). This allows training to

be more efficient, with less necessary storage and memory required to keep track of weights and parameters, and it also allows CNNs to apply learned patterns in the data irrespective of its exact position (Goodfellow 331).

Pooling layers scale down the data being passed forward in the model by summarizing activation values over a region (Goodfellow 337). Different types of pooling layers exist, for instance, average pooling computes the average value for patches of a feature map to create a new downsampled version. Max pooling similarly downsamples the feature map, but takes the maximum value of each patch instead (CNN: Introduction to Pooling Layer). The result of using these layers is that the model is more invariant to small translations to the input data.

2.3 Recursive Neural Networks and Long Short-Term Memory Networks

Recurrent neural networks act as an alternative to CNNs, specialized to process a sequence of values such as time-series data, and hold a distinct advantage of being able to process sequences of variable length, unlike CNNs which require a fixed input size (Goodfellow 367).

RNNs use shared parameters to process the input value at each time step, while taking advantage of information in the hidden state in the previous timestep, to allow it to carry over information from the past (DiPietro). The representation of the hidden state for a simple RNN is:

$$h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h),$$

in which h_t represents the value of the hidden state at timestep t, x_t represents the value of the input sequence at timestep t, W_{hh} and W_{xh} represent the weight parameters shared and learnt over time, and b_h represents the shared bias parameters (DiPietro).

While in theory, the design of RNNs should allow them to utilize information from the past to process data in the current time step, they fail in practice due to what is known as the exploding and vanishing gradient problem (DigitalSreeni). When repeatedly multiplying the same weights to an input signal, it is common for the effect of a past weight to slowly reduce to zero if the weight value is less than 1, or explode to extremely high values if the weight value is more than

one, leading to more volatile learning, and a model that is not able to effectively adjust and leverage information across its time series input (DiPietro).

This led to the development of long short-term memory or LSTM, which has become one of the most widely used RNN architectures due to being able to significantly reduce the effects of the vanishing gradient problem in standard RNNs. LSTM introduces a memory cell that can be acted upon by the forget gate, the input gate and the output gate which adjust over time, and the additional complexity creates more paths for information and weights from a past time step to propagate to future time steps (DiPietro).

2.4 Other Deep Learning Concepts

As mentioned in the introduction, deep learning models learn through an iterative training process that takes in data and uses a loss function to then calculate how to optimally adjust the parameters of the model (Holdsworth). However, in practical applications of machine learning, an algorithm will be tested on new, unseen data that it has not trained on (Goodfellow 107). Therefore, the goal of a practical model is to learn from its training data in such a way that its parameters are broadly applicable even on data that it has never encountered, an ability known as generalization (Goodfellow 107).

A model that struggles to learn its training set is considered underfitting and a model that performs exceptionally well on its training set but severely underperforms on unforeseen data is considered overfitting; adjustments made to the model or learning algorithm to prevent the latter are known as regularization techniques (Goodfellow 117). Regularization will be a key component of achieving high test set performance in many practical deep-learning tasks.

2.5 Machine Learning in Biosignal Processing

Machine learning methods that are not based on neural networks are also prevalent techniques in processing biosignals. Decision tree algorithms were applied with success to the task of EMG signal classification by Ercan Gokgoz and Abdulhamit Subasi. Following basic pre-processing, these authors performed feature extraction on the data before applying the decision tree

algorithm through discrete wavelet transform (DWT), which breaks down the information of the signal into smaller, isolated components (Gokgoz). Among the decision tree-based machine learning algorithms applied, random forest was the most successful when paired with DWT feature extraction achieving a total classification accuracy of 96.67% (Gokgoz).

Even on WESAD, the biosignal dataset used for this investigation, the authors demonstrated the performance of machine learning algorithms such as Decision Tree, Random Forest, AdaBoost Decision Tree, and Linear discriminant analysis, and even in the absence of deep learning were able to perform with relative accuracy on the three-class classification task (Schmidt). However, part of this investigation will be to use deep learning models: CNNs and RNNs, to surpass the benchmark set by the dataset's original paper.

3 Dataset and Method:

3.1 Dataset

The WESAD dataset contains biosignals recorded at different frequencies on two recording devices: the chest-worn RespiBAN Professional, and the wrist-worn Empatica E4 to collect multimodal data for stress and affect recognition (Schmidt). On the RespiBAN: ACC, RESP, ECG, EDA, EMG, and TEMP were all sampled at 700 Hz, whereas on the E4, BVP, EDA, TEMP and ACC were sampled at 64 Hz, 4 Hz, 4 Hz and 32 Hz, respectively (Schmidt). The dataset contains recordings from 15 subjects with a mean age of 27.5 years, who were placed through guided activities to provoke conditions of amusement, stress and calm state, as well as a baseline state to serve as a control variable (Schmidt). The paper focuses on the 3-class classification task, with the ground truth for labels obtained from the current phase of the lab procedure (Schmidt).

To meaningfully compare the performance of deep learning models on the WESAD dataset to the machine learning approaches in the paper, the same amount of input data must be given for the prediction of a subject's affect states. The data was divided using 1-minute sliding windows with a step size of 30 seconds to segment the dataset, with the label of each window determined

by the most prevalent label within that time frame (Schmidt). The result is a dataset containing a total of 1105 windows, with 186 (17%) labelled with amusement, 332 (30%) labelled as stress, and 587 (53%) labelled as calm/baseline (Dzieżyc).

As part of data pre-processing, 3 to 97% winsorization was applied to remove extreme values from the signals, the signals from the Respibann were downsampled to reduce unnecessary computational and memory costs, and min-max normalization was applied to the data, which can help to yield less volatile training. These choices are inspired by the work done by Maciej Dziezyc et al., and their code implementation in this investigation can be seen in Appendix B.

3.2 Model Architectures

To create a deep learning model, it is necessary to design a specific architecture, which outlines the parameters and operations needed for forward-propagation. To process information from multiple signals in parallel, the approach used in this investigation is to process each signal separately via branches, after which the output of each branch will be concatenated and passed through a series of fully connected layers, allowing the model to use the information gained from each signal in conjunction.

The convolutional branch was modified from the TimeCNN architecture outlined by Hassan Ismail Fawaz et al., which specifies kernel size 7 for convolutions and average pooling layers of kernel size 3. ReLU activation functions were placed after convolution operations instead of sigmoid functions, and the max pool replaced the average pool, due to being considered more efficient in training (Thakur).

The recursive branch utilizes a bidirectional LSTM layer that maintains a hidden state vector of size 64, meaning that the recursive network passes both forward and backward across the input signal (Bidirectional LSTM in NLP). The last hidden state is passed forward from the branch to be concatenated with the output of other branches.

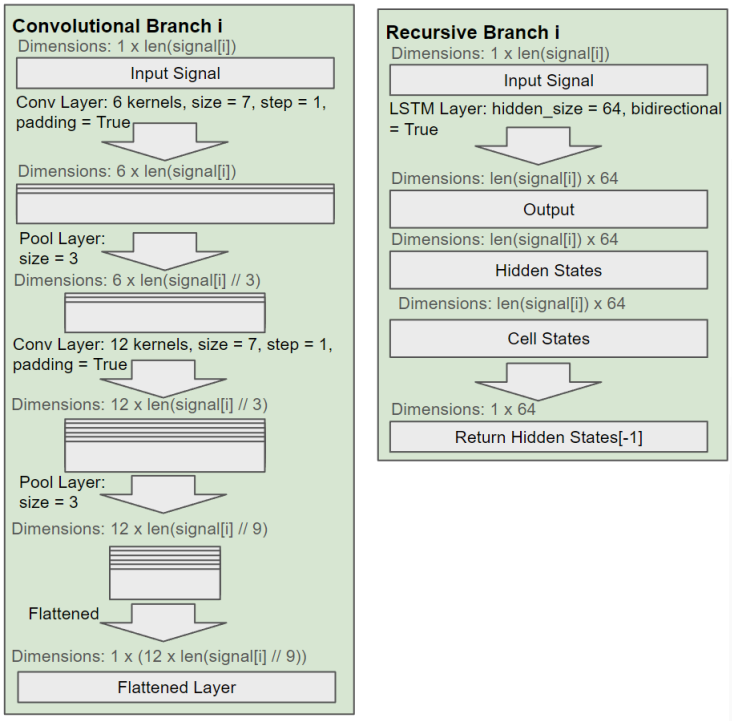


Figure 1. Diagram displaying the architecture of an individual branch in the CNN and RNN model

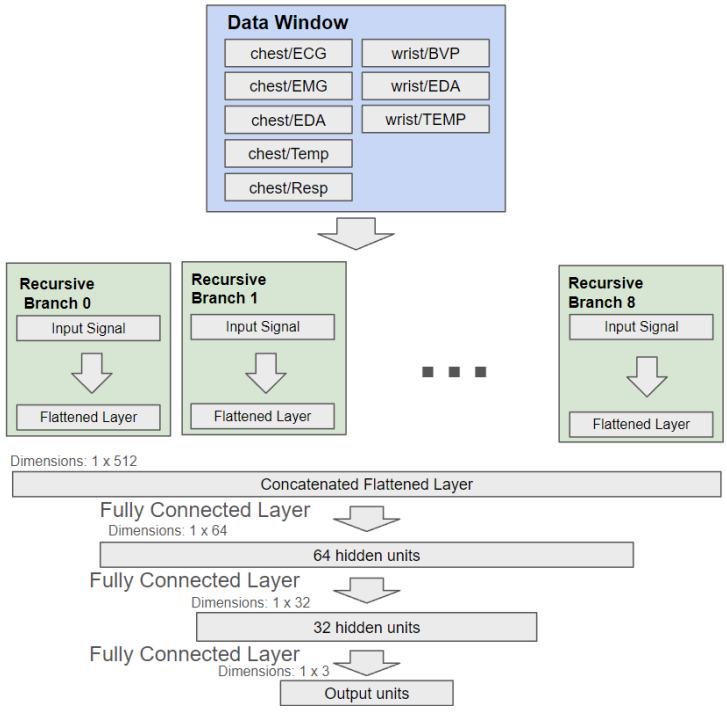


Figure 2. Diagram of multimodal deep RNN architecture

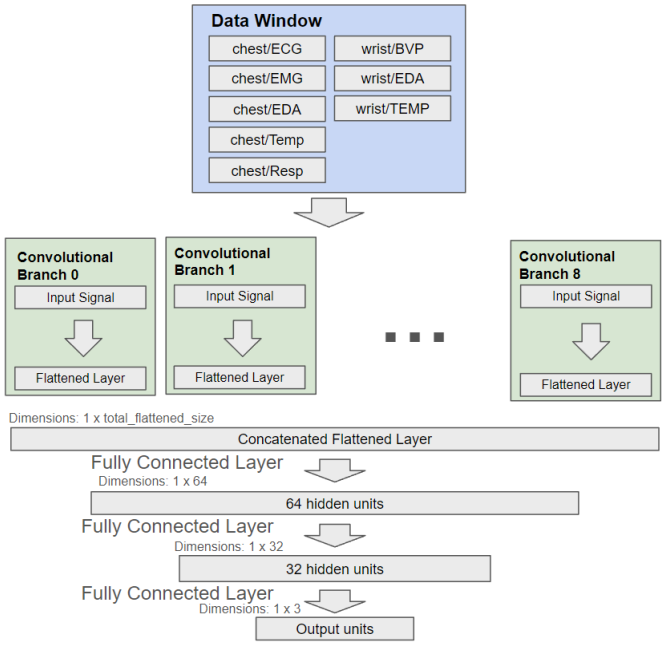


Figure 3. Diagram of multimodal deep CNN architecture

The 1-minute sliced data windows are passed to the model as a dictionary, and the 8 chosen signals are each associated with one of the model's branches. ACC (acceleration) was excluded from the forward pass due to being composed of 3 signals and demonstrating poor predictive power in all machine learning algorithms in the results of the WESAD paper (Schmidt).

Due to the task involving multi-class classification, cross-entropy loss will be used as the loss function between the output activations and the target label to evaluate the error in the model's prediction (Crossentropyloss). Parameter adjustment via gradient descent will be handled by the ADAM optimization algorithm, which is more powerful than stochastic gradient descent as it maintains a separate learning rate for each parameter during training (Brownlee).

The specific details of the Pytorch implementation of these architectures are in Appendix C.

3.3 Evaluation Methods

The main accuracy score used to compare the models will be obtained using Leave One Subject Out cross-validation (LOSO CV), which involves reserving one subject as the test set, and all other subjects for the training set (Schmidt). The final accuracy score is the average accuracy

obtained when running the procedure for all subjects to be used as the test set once (Schmidt). Graphs of training and testing accuracy against the number of epochs spent learning, as well as graphs of training and testing loss against the number of epochs, provide visual qualitative data regarding the stability of each model's training, and data is also collected on the average time that it takes a model to train over its entire training set (i.e. one epoch), as an additional point of comparison.

The raw data and graphs are in Appendix D.

4 Experimental Results and Analysis:

There were some curious trends seen when utilizing LOSO CV to evaluate the CNN and RNN models' performance on the WESAD biosignal dataset. Overall, both the CNN and RNN achieved a similar average accuracy metric, with 88.76% accuracy and 89.22% accuracy respectively. Below is a summary of the raw data found in the appendix.

Table 1. Summary of CNN and RNN performance on WESAD

Deep Learning Architecture Type	Multimodal CNN (8 Convolutional branches) + 3 FC layers	Multimodal RNN (8 LSTM branches) + 3 FC layers
Average LOSO Accuracy (%)	88.76	89.22
Average time to train per epoch (s)	16.26	29.84
Average best epoch in terms of loss	12.67	10.67
Best performed subject(s)	S5, S9, S11, S15, S16	S5, S9, S11, S13, S16
Worst	S2	S14

performed subject		

One clear discrepancy between the two models while training was the time efficiency of their iterations, as, despite an overall similar architecture design, the RNN model took nearly double the time to train over an epoch of data. One reason that CNNs may be more computationally efficient than RNNs comes from the advantage of pooling, while CNNs can apply max pooling between layers to subsample the data being propagated forward, LSTMs must pass through all values of the input in sequence with the added complexity of input, output and forget gates that add to the time needed to process the dataset (Goodfellow; DiPietro).

The two models tended to perform well on similar subjects, as both models were able to predict the affect state of the subject to perfect accuracy at least once over a 20-epoch period on subjects 5, 9, 11 and 16. Given the LOSO CV procedure uses all other subjects to predict the labels on the excluded subject, these 4 subjects may simply have possessed a signal to affect state mapping that was similar to other signals in the WESAD dataset (Schmidt). On the opposite end, subjects such as subject 2 may have had noisier signal data, or been a relative outlier among the rest of the biosignal data.

Below are examples of training and test loss graphs plotted for subject 13:

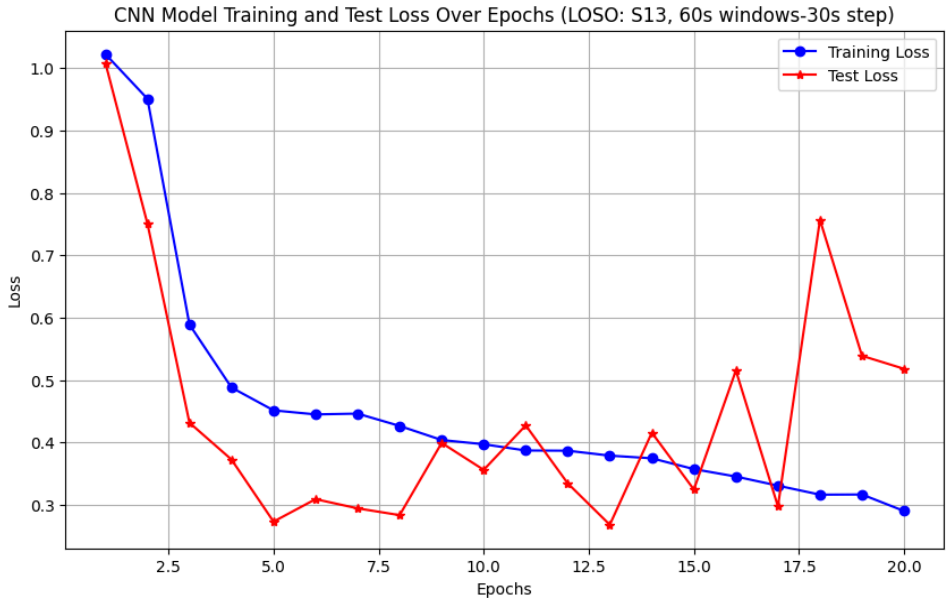


Figure 4. CNN Model Training and Test Loss over Epochs (LOSO: S13)

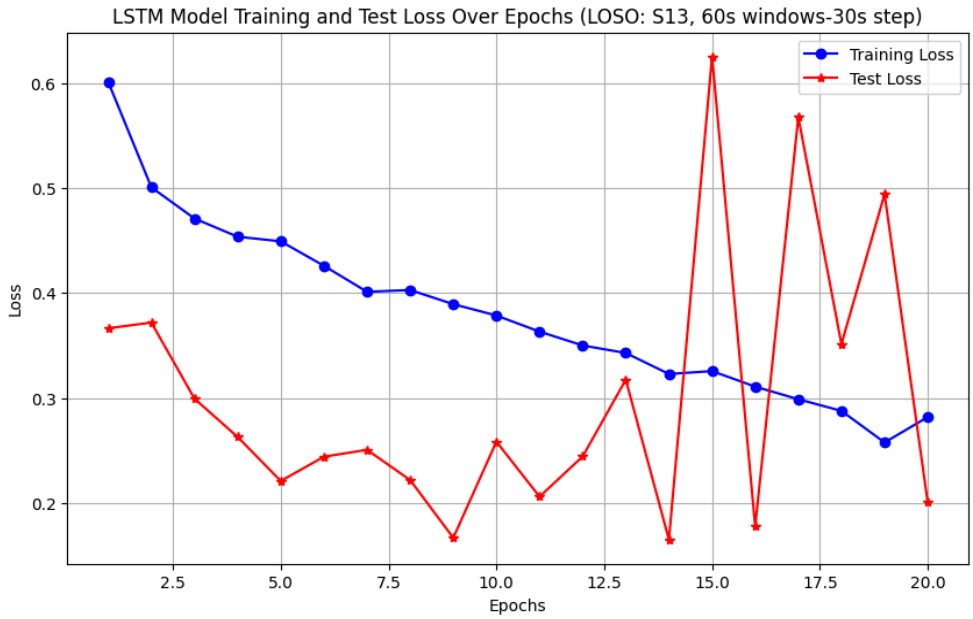


Figure 5. LSTM Model Training and Test Loss over Epochs (LOSO: S13)

In these graphs, it can be observed that for both models, both training and test losses decrease for the first 8 epochs of training, after which test loss begins to increase while training loss continues to decrease. This is an example of overfitting, and these graphs visually signal the turning point at which the parameters learnt become overtuned to performance on the training set, at the expense of generalization. In the experiment, it was observed that the RNN model would begin to overfit the training data slightly earlier (in terms of epochs) than the CNN model, on average at epoch 10.67 versus epoch 12.67. A common, simple regularization strategy is to implement early stopping, in which the model halts its training when its performance on a test or validation set begins to deteriorate consistently (Goodfellow).

5 Conclusion and Evaluation:

In the WESAD dataset paper, the highest performing machine learning model applied to the three-class classification task was AdaBoost, which when given all chest biosignals achieved a LOSO CV accuracy of 80.34 ± 0.43, and achieved a similar accuracy when processing all biosignals: 79.57 ± 0.93 (Schmidt). Both the multimodal CNN and multimodal RNN architectures proposed in this investigation were able to significantly outperform this benchmark

by achieving an average LOSO CV accuracy of 88.76% and 89.22%. These accuracy scores also imply that LSTM layers may be marginally better performing in biosignal processing than 1D CNN layers, although at the cost of significant training computational time and faster overfitting. In response to the research question, the findings of this investigation support the idea that deep learning algorithms such as CNNs and RNNs can be effective in the interpretation and classification of biosignal time-series data, and can serve as superior models to other machine learning models if evaluated solely on accuracy. The tradeoff of 'deep' neural networks is that they are naturally computationally expensive not just to train, but to deploy if there are significant situational restraints (Goodfellow).

The process of conducting this investigation featured roadblocks, one of which was determining the optimal values of certain hyperparameters, such as the learning rate and weight decay of certain parameters, to improve model generalization (Goodfellow 107). These hyperparameters were determined through a process of trial and error, but the random nature of model training as well as the time cost of testing a single set of hyperparameters across all subjects meant that this process of hyperparameter tuning was likely not sufficiently rigorous to find truly optimal values for test accuracy. However, the hyperparameters ultimately chosen for both models seemed to reach good performance on the unseen data.

Another roadblock came from extracting the data from WESAD, which features different biosignals sampled at vastly different frequencies. Without an easy way to debug and verify that the data in each window was synchronized, a significant time was spent encountering anomalous behaviors and investigating why the models performed so poorly on the test data. The bugs in the pre-processing script were eventually found and adjusted, with clear improvements in model performance that more closely resembled theory.

There are many ways in which this investigation can be extended. One way in which model performance can be improved is by taking advantage of models with pre-trained weights that can improve knowledge due to containing priors on similar tasks (Models and Pre-Trained Weights). A future investigation may investigate whether ensembling methods can further optimize the performance of deep learning models on the WESAD dataset, or other datasets that utilize

biosignals. Other types of deep learning architectures can be explored, for instance, transformers are known for their applications in natural language processing, but can be modified to analyze biosignals similarly to CNNs and LSTMs (Anwar).

Despite its limitations, this paper demonstrates the capabilities of deep learning architectures in tasks that involve analysing and classifying information derived from multiple biosignals and helps to improve our understanding of the extent to which these models can be successfully trained and applied.

Works Cited

Works Cited

Anwar, Ayman, et al. “Transformers in biosignal Analysis: A review.” Information Fusion, vol. 114, Feb. 2024, https://doi.org/10.1016/j.inffus.2024.102697.

Brodtman, Zack. “The Importance and Reasoning behind Activation Functions.” Medium, Towards Data Science, 16 Nov. 2021, towardsdatascience.com/the-importance-and-reasoning-behind-activation-functions-4dc00e74db41.

“Bidirectional LSTM in NLP.” GeeksforGeeks, GeeksforGeeks, 8 June 2023, www.geeksforgeeks.org/bidirectional-lstm-in-nlp/.

Brownlee, Jason. “A Gentle Introduction to K-Fold Cross-Validation.” MachineLearningMastery.Com, 3 Oct. 2023, machinelearningmastery.com/k-fold-cross-validation/.

Brownlee, Jason. “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.” MachineLearningMastery.Com, 12 Jan. 2021, machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

“CNN: Introduction to Pooling Layer.” GeeksforGeeks, GeeksforGeeks, 21 Apr. 2023, www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/.

Cowley, Benjamin, et al. “The Psychophysiology Primer: A Guide to Methods and a Broad Review with a Focus on Human–Computer Interaction.” Foundations and Trends® in Human–Computer Interaction, Now Publishers, Inc., 2 Nov. 2016, www.nowpublishers.com/article/Details/HCI-065.

“Crossentropyloss.” *CrossEntropyLoss - PyTorch 2.4 Documentation*,

pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html. Accessed 2 Sept. 2024.

Dahal, Prashant, and Akhlesh Kushwaha. “Electrocardiogram (ECG): Parts, Principle, Procedure, Types.” Microbe Notes, 4 Sept. 2023, microbenotes.com/electrocardiogram-ecg/.

DigitalSreeni. “165 - An Introduction to RNN and LSTM.” YouTube, YouTube, 8 Oct. 2020, www.youtube.com/watch?v=Mdp5pAKNNW4.

DiPietro, Robert, and Gregory D. Hager. “Deep learning: RNNS and LSTM.” Handbook of Medical Image Computing and Computer Assisted Intervention, 2020, pp. 503–519, https://doi.org/10.1016/b978-0-12-816176-0.00026-0.

Dzieżyc, Maciej, et al. “Can we ditch feature engineering? end-to-end deep learning for affect recognition from physiological sensor data.” Sensors, vol. 20, no. 22, 16 Nov. 2020, p. 6535, https://doi.org/10.3390/s20226535.

Feng, Weijiang, et al. “Audio visual speech recognition with multimodal recurrent neural networks.” 2017 International Joint Conference on Neural Networks (IJCNN), May 2017, https://doi.org/10.1109/ijcnn.2017.7965918.

Giannakakis, Giorgos, et al. "Review on psychological stress detection using biosignals." IEEE transactions on affective computing 13.1 (2019): 440-460.

Gokgoz, Ercan, and Abdulhamit Subasi. “Comparison of decision tree algorithms for EMG Signal Classification using DWT.” Biomedical Signal Processing and Control, vol. 18, Apr. 2015, pp. 138–144, https://doi.org/10.1016/j.bspc.2014.12.005.

Goodfellow, Ian, et al. Deep Learning. The MIT Press, 2016.

Holdsworth, Jim, and Mark Scapicchio. “What Is Deep Learning?” IBM, 17 June 2024, www.ibm.com/topics/deep-learning.

Ismail Fawaz, Hassan, et al. “Deep Learning for Time Series classification: A Review.” Data Mining and Knowledge Discovery, vol. 33, no. 4, 2 Mar. 2019, pp. 917–963, https://doi.org/10.1007/s10618-019-00619-1.

Kiranyaz, Serkan, et al. “1d convolutional neural networks and applications: A survey.” Mechanical Systems and Signal Processing, vol. 151, Apr. 2021, p. 107398, https://doi.org/10.1016/j.ymssp.2020.107398.

Liu, Shiwei, et al. “Multi-branch CNN and grouping cascade attention for medical image classification.” Scientific Reports, vol. 14, no. 1, 1 July 2024, https://doi.org/10.1038/s41598-024-64982-w.

“Models and Pre-Trained Weights.” Models and Pre-Trained Weights - Torchvision Main Documentation, 2017, pytorch.org/vision/master/models.html.

Sajno, Elena, et al. “Machine learning in biosignals processing for Mental Health: A Narrative Review.” Frontiers in Psychology, vol. 13, 13 Jan. 2023, https://doi.org/10.3389/fpsyg.2022.1066317.

Schmidt, Philip, et al. “Introducing WESAD, a multimodal dataset for wearable stress and affect detection.” Proceedings of the 20th ACM International Conference on Multimodal Interaction, 2 Oct. 2018, https://doi.org/10.1145/3242969.3242985.

Swapna, Mudrakola, et al. “Bio-Signals in Medical Applications and Challenges Using Artificial Intelligence.” MDPI, Multidisciplinary Digital Publishing Institute, 25 Feb. 2022, doi.org/10.3390/jsan11010017.

Thakur, Ayush. “Relu vs. Sigmoid Function in Deep Neural Networks.” W&B, 19 Aug. 2020,

wandb.ai/ayush-thakur/dl-question-bank/reports/ReLU-vs-Sigmoid-Function-in-Deep-Neural-Networks--VmlldzoyMDk0MzI.

"Utilizing the PPG/BVP Signal – Empatica Support." *Empatica Support Center*, support.empatica.com/hc/en-us/articles/204954639-Utilizing-the-PPG-BVP-signal. Accessed 1 Sept. 2024.

Wang, Weinan, et al. "PulseDB: A large, cleaned dataset based on mimic-III and vitaldb for benchmarking cuff-less blood pressure estimation methods." Frontiers in Digital Health, vol. 4, 8 Feb. 2023, https://doi.org/10.3389/fdgth.2022.1090854.

"What Are Convolutional Neural Networks?" IBM, 6 Oct. 2021, www.ibm.com/topics/convolutional-neural-networks.

Appendix

Appendix A: Imports and Modules

Packages installed in Python 3.12.1 virtual environment (IDE: Visual Studio Code)

```
Package           Version
----------------- -----------
appnope           0.1.4
asttokens         2.4.1
comm              0.2.2
contourpy         1.2.1
cycler            0.12.1
debugpy           1.8.5
decorator         5.1.1
executing         2.0.1
filelock          3.15.4
fonttools         4.53.1
fsspec            2024.6.1
ipykernel         6.29.5
ipython           8.26.0
jedi              0.19.1
jupyter_client    8.6.2
jupyter_core      5.7.2
kiwisolver        1.4.5
MarkupSafe        2.1.5
matplotlib        3.9.2
matplotlib-inline 0.1.7
mpmath            1.3.0
nest-asyncio      1.6.0
networkx          3.3
numpy             2.1.0
packaging         24.1
parso             0.8.4
pexpect           4.9.0
pillow            10.4.0
pip               24.2
platformdirs      4.2.2
prompt_toolkit    3.0.47
psutil            6.0.0
ptyprocess        0.7.0
pure_eval         0.2.3
Pygments          2.18.0
pyparsing         3.1.2
python-dateutil   2.9.0.post0
pyzmq             26.1.1
scipy             1.14.0
setuptools        73.0.1
six               1.16.0
stack-data        0.6.3
sympy             1.13.2
torch             2.4.0
tornado           6.4.1
traitlets         5.14.3
typing_extensions 4.12.2
wcwidth           0.2.13
```

Appendix B: Data Preprocessing

```python
# first, open the pkl file
import pickle
import numpy as np

subject = 2

file_path = f"WESAD/S{subject}/S{subject}.pkl"
with open(file_path, 'rb') as file:
    data = pickle.load(file, encoding='latin1')
```

```python
from scipy.stats import mstats

#downsampling frequency factors
downsampling_factor = {
    "chest/ECG": 10,
    "chest/ACC": 70,
    "chest/EMG": 70,
    "chest/EDA": 200,
    "chest/Temp": 200,
    "chest/Resp": 200,
    "wrist/ACC": 4,
    "wrist/BVP": 1,
    "wrist/EDA": 1,
    "wrist/TEMP": 1,
}

#3% winsorization + min-max normalization of signals in data
def preprocess(arrayobj, device, signal):
    arrayobj = mstats.winsorize(arrayobj, limits = [0.03,0.03])
    arrayobj = arrayobj[::downsampling_factor[f"{device}/{signal}"]]
    min_value, max_value = np.min(arrayobj), np.max(arrayobj)
    arrayobj = (arrayobj - min_value) / (max_value - min_value)
    return arrayobj


for device in data['signal'].keys():
    for signal in data['signal'][device].keys():
        print(f"Old range for {device}/{signal}:")
        print([np.min(data['signal'][device][signal]), np.max(data['signal'][device][signal])])
        data['signal'][device][signal] = preprocess(data['signal'][device][signal], device, signal)
        print(f"New range for {device}/{signal}:")
        print([np.min(data['signal'][device][signal]), np.max(data['signal'][device][signal])])
```

```
windows = []
frequencies = {
    "chest/ECG": 70,
    "chest/ACC": 10,
    "chest/EMG": 10,
    "chest/EDA": 3.5,
    "chest/Temp": 3.5,
    "chest/Resp": 3.5,
    "wrist/ACC": 8,
    "wrist/BVP": 64,
    "wrist/EDA": 4,
    "wrist/TEMP": 4,

}

# labels are collected at frequency of 700, so we are looking to divide into windows of 60 * 700 and steps of 30 * 700 Hz

labelfreq = 700
windowsize = 60
windowstep = 30
```

```
for i in range((len(data["label"]) // (windowstep * labelfreq)) - 2):
    datawindow = {}
    datawindow['signal'] = {}
    for device in data['signal'].keys():
        datawindow['signal'][device] = {}
        for signal in data['signal'][device].keys():


            fkey = f"{device}/{signal}"
            datawindow['signal'][device][signal] = np.array(data['signal'][device][signal][int(i
                                * frequencies[fkey] * windowstep):int(i * frequencies[fkey] * windowstep + frequencies[fkey] * windowsize)])
    # print(i)

    labels = data['label'][int(i * labelfreq * windowstep): int(i * labelfreq * windowstep + windowsize * labelfreq)]
    # print(labels)
    # print(np.bincount(labels))
    label = np.bincount(labels).argmax() # finds the most common element of an integer array

    datawindow['label'] = label
    if (label in [0, 4, 5, 6, 7]): # ignoring labels 0, 5, 6, 7. May consider removing label 4 as well, as I usually do not see meditation considered.
        continue


    datawindow['subject'] = data['subject']
    windows.append(datawindow)
print("done")
```

```
with open(f'windows/S{subject}windows.pkl', 'wb') as file:
    pickle.dump(windows, file)

print("List has been pickled and saved to f'windows/S{subject}windows.pkl'.")
```

```python
import pickle
import numpy as np
from scipy.stats import mstats


subjects = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17]
windows = []

for subject in subjects:

    file_path = f"WESAD/S{subject}/S{subject}.pkl"
    with open(file_path, 'rb') as file:
        data = pickle.load(file, encoding='latin1')

    downsampling_factor = {
        "chest/ECG": 10,
        "chest/ACC": 70,
        "chest/EMG": 70,
        "chest/EDA": 200,
        "chest/Temp": 200,
        "chest/Resp": 200,
        "wrist/ACC": 4,
        "wrist/BVP": 1,
        "wrist/EDA": 1,
        "wrist/TEMP": 1,
    }

    #3% winsorization + min-max normalization of signals in data
    def preprocess(arrayobj, device, signal):
        arrayobj = mstats.winsorize(arrayobj, limits = [0.03,0.03])
        arrayobj = arrayobj[::downsampling_factor[f"{device}/{signal}"]]
        min_value, max_value = np.min(arrayobj), np.max(arrayobj)
        arrayobj = (arrayobj - min_value) / (max_value - min_value)
        return arrayobj
```

```python
    for device in data['signal'].keys():
        for signal in data['signal'][device].keys():
            # print(f"Old range for {device}/{signal}:")
            # print([np.min(data['signal'][device][signal]), np.max(data['signal'][device][signal])])
            data['signal'][device][signal] = preprocess(data['signal'][device][signal], device, signal)
            # print(f"New range for {device}/{signal}:")
            # print([np.min(data['signal'][device][signal]), np.max(data['signal'][device][signal])])

    frequencies = {
        "chest/ECG": 70,
        "chest/ACC": 10,
        "chest/EMG": 10,
        "chest/EDA": 3.5,
        "chest/Temp": 3.5,
        "chest/Resp": 3.5,
        "wrist/ACC": 8,
        "wrist/BVP": 64,
        "wrist/EDA": 4,
        "wrist/TEMP": 4,

    }

    # labels are collected at frequency of 700, so we are looking to divide into windows of 60 * 700 and steps of 30 * 700 Hz

    labelfreq = 700
    # in seconds
    windowsize = 60
    windowstep = 30
```

```
    for i in range((len(data["label"]) // (windowstep * labelfreq)) - 2):
        datawindow = {}
        datawindow['signal'] = {}
        for device in data['signal'].keys():
            datawindow['signal'][device] = {}
            for signal in data['signal'][device].keys():


                fkey = f"{device}/{signal}"
                datawindow['signal'][device][signal] = np.array(data['signal'][device][signal][int(i * frequencies[fkey] * windowstep):
                                                                                               int(i * frequencies[fkey] * windowstep +
                                                                                                   frequencies[fkey]
                                                                                                   * windowsize)])
        # print(i)

        labels = data['label'][int(i * labelfreq * windowstep): int(i * labelfreq * windowstep + windowsize * labelfreq)]
        # print(labels)
        # print(np.bincount(labels))
        label = np.bincount(labels).argmax() # finds the most common element of an integer array

        datawindow['label'] = label
        if (label in [0, 4, 5, 6, 7]): # ignoring labels 0, 5, 6, 7. May consider removing label 4 as well, as I usually do not see meditation co
            continue


        datawindow['subject'] = data['subject']
        windows.append(datawindow)
    print("done")

with open(f'allwindows.pkl', 'wb') as file:
    pickle.dump(windows, file)

print("List has been pickled and saved to f'allwindows.pkl'.")
```

```
amusement = 0
stress = 0
baseline = 0
for window in windows:
    if (window['label'] == 1.):
        baseline += 1
    elif (window['label'] == 2.):
        stress += 1
    else:
        amusement += 1

print(baseline)
print(stress)
print(amusement)
```

Appendix C: Model Architecture and Training

Multimodal 1D CNN pytorch implementation

```
class MultiSignalTimeCNNnoACC(nn.Module):
    def __init__(self, input_dim_1, input_dim_2, input_dim_3, input_dim_4,
input_dim_5, input_dim_6, input_dim_7, input_dim_8, num_classes=3):
        super(MultiSignalTimeCNNnoACC, self).__init__()


        # CNN Branch for Signal 1
        self.cnn1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
```

```
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )

        # CNN Branch for Signal 2
        self.cnn2 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )

        # CNN Branch for Signal 3
        self.cnn3 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding = 3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )
        self.cnn4 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding = 3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )
        self.cnn5 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
```

```
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding = 3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )
        self.cnn6 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding = 3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )
        self.cnn7 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )
        self.cnn8 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=6, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Conv1d(in_channels=6, out_channels=12, kernel_size=7, padding=3),
            nn.Sigmoid(),
            nn.MaxPool1d(kernel_size=3),
            nn.Flatten()
        )

        # Calculate the size of the flattened CNN output
        flattened_size_1 = 12 * (((input_dim_1) // 3) // 3)
        flattened_size_2 = 12 * (((input_dim_2) // 3) // 3)
        flattened_size_3 = 12 * (((input_dim_3) // 3) // 3)
        flattened_size_4 = 12 * (((input_dim_4) // 3) // 3)
        flattened_size_5 = 12 * (((input_dim_5) // 3) // 3)
        flattened_size_6 = 12 * (((input_dim_6) // 3) // 3)
        flattened_size_7 = 12 * (((input_dim_7) // 3) // 3)
```

```
        flattened_size_8 = 12 * (((input_dim_8) // 3) // 3)



        # Total size after concatenating the flattened outputs
        total_flattened_size = flattened_size_1 + flattened_size_2 + flattened_size_3
+ flattened_size_4 + flattened_size_5 + flattened_size_6 + flattened_size_7 +
flattened_size_8

        # Fully connected layers
        self.fc1 = nn.Linear(total_flattened_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, num_classes)

        self.dropout = nn.Dropout(0.3)

    def forward(self, datawindow):
        signal = []
        for device in datawindow["signal"]:
            for s in datawindow["signal"][device]:
                if (s != "ACC"): # excluding ACC for now.
                    signal.append(np.zeros((len(datawindow["signal"][device][s]))))
                    for i in range(len(datawindow["signal"][device][s])):
                        signal[-1][i] = datawindow["signal"][device][s][i]
                    signal[-1] =
torch.from_numpy(signal[-1]).float().unsqueeze(0).unsqueeze(0)

        # print(signal)



        # Pass each signal through its respective CNN branch
        x1 = self.cnn1(signal[0])
        x2 = self.cnn2(signal[1])
        x3 = self.cnn3(signal[2])
        x4 = self.cnn4(signal[3])
        x5 = self.cnn5(signal[4])
        x6 = self.cnn6(signal[5])
        x7 = self.cnn7(signal[6])
        x8 = self.cnn8(signal[7])

        # Concatenate the features from each branch
```

```
        combined = torch.cat((x1, x2, x3, x4, x5, x6, x7, x8), dim=1)

        # Pass through fully connected layers
        x = torch.relu(self.fc1(combined))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)

        # x = torch.softmax(x, dim=1) # use only if not using cross entropy loss

        return x
```

Multimodal RNN pytorch implementation

```
class MultiSignalLSTMnoACC(nn.Module):
    def __init__(self, input_dim_1, input_dim_2, input_dim_3, input_dim_4,
input_dim_5, input_dim_6, input_dim_7, input_dim_8, num_classes=3):
        super(MultiSignalLSTMnoACC, self).__init__()

        hidden_dim = 64
        # LSTM Branch for Signal 1

        self.lstm1 = nn.LSTM(input_size=input_dim_1, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm2 = nn.LSTM(input_size=input_dim_2, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm3 = nn.LSTM(input_size=input_dim_3, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm4 = nn.LSTM(input_size=input_dim_4, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm5 = nn.LSTM(input_size=input_dim_5, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm6 = nn.LSTM(input_size=input_dim_6, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)
        self.lstm7 = nn.LSTM(input_size=input_dim_7, hidden_size=hidden_dim,
```

```
batch_first=True, bidirectional=True)
        self.lstm8 = nn.LSTM(input_size=input_dim_8, hidden_size=hidden_dim,
batch_first=True, bidirectional=True)

        # Fully connected layers
        total_hidden_size = hidden_dim * 8 # 8 signals, each with hidden_dim output
        # Fully connected layers
        self.fc1 = nn.Linear(total_hidden_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, num_classes)

        self.dropout = nn.Dropout(0.3)

    def forward(self, datawindow):
        signal = []
        for device in datawindow["signal"]:
            for s in datawindow["signal"][device]:
                if (s != "ACC"): # excluding ACC for now.
                    signal.append(np.zeros((len(datawindow["signal"][device][s]))))
                    for i in range(len(datawindow["signal"][device][s])):
                        signal[-1][i] = datawindow["signal"][device][s][i]
                    signal[-1] =
torch.from_numpy(signal[-1]).float().unsqueeze(0).unsqueeze(0)

        # print(signal)



        # Pass each signal through its respective CNN branch
        _, (h1, _) = self.lstm1(signal[0])
        _, (h2, _) = self.lstm2(signal[1])
        _, (h3, _) = self.lstm3(signal[2])
        _, (h4, _) = self.lstm4(signal[3])
        _, (h5, _) = self.lstm5(signal[4])
        _, (h6, _) = self.lstm6(signal[5])
        _, (h7, _) = self.lstm7(signal[6])
        _, (h8, _) = self.lstm8(signal[7])

        # Concatenate the features from each branch
        combined = torch.cat((h1[-1], h2[-1], h3[-1], h4[-1], h5[-1], h6[-1], h7[-1],
h8[-1]), dim=1)
```

```
        # Pass through fully connected layers
        x = torch.relu(self.fc1(combined))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)

        # x = torch.softmax(x, dim=1) # use only if not using cross entropy loss

        return x
```

Model Creation and Training

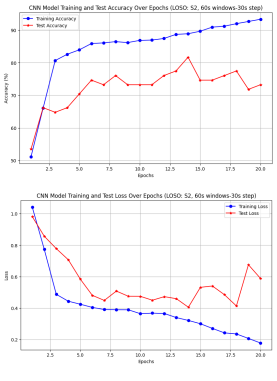
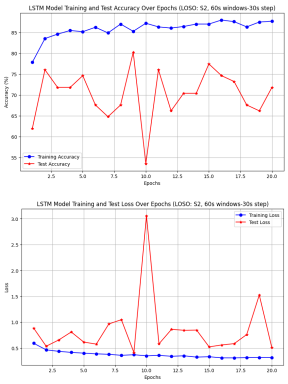
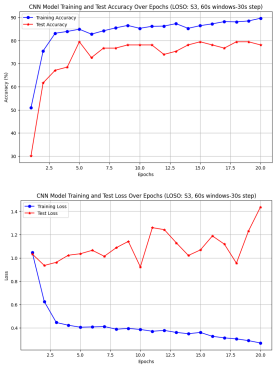
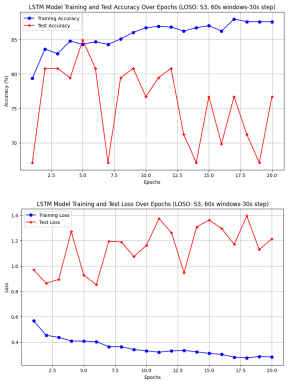
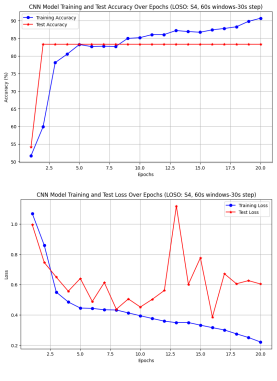
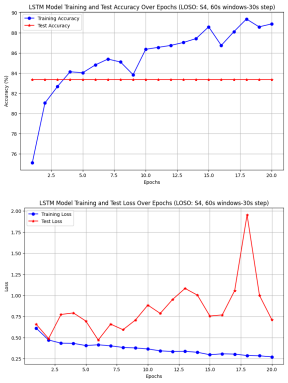
```
input_dim_1 = 4200
input_dim_2 = 600
input_dim_3 = 210
input_dim_4 = 210
input_dim_5 = 210
input_dim_6 = 3840
input_dim_7 = 240
input_dim_8 = 240
num_classes = 3

model = MultiSignalLSTMnoACC(input_dim_1=input_dim_1, input_dim_2=input_dim_2,
input_dim_3=input_dim_3, input_dim_4=input_dim_4, input_dim_5=input_dim_5,
input_dim_6=input_dim_6, input_dim_7=input_dim_7, input_dim_8=input_dim_8,
num_classes=num_classes)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)

batch_size = 1 # probably will break if you change this as of rn.
num_epochs = 20

train_accuracies = []
test_accuracies = []
```

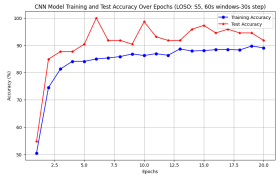
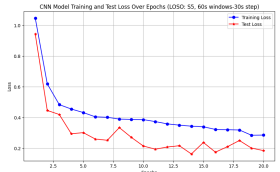
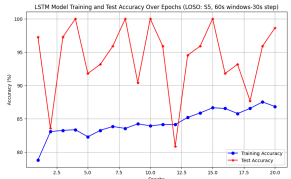
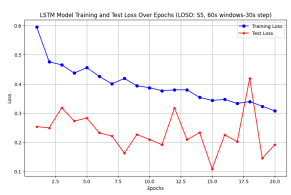
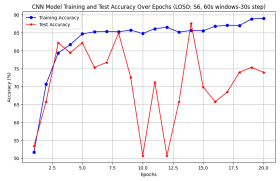
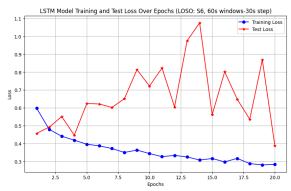
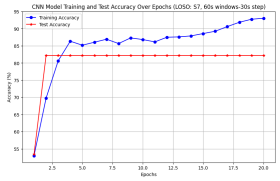
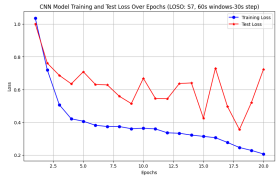
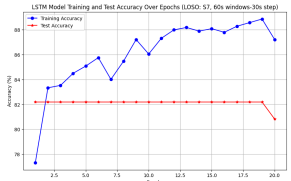
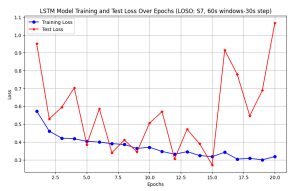
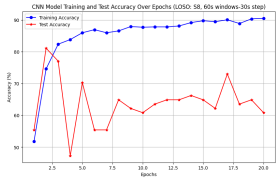
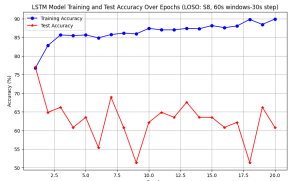
```
train_loss = []
test_loss = []

for epoch in range(num_epochs):
   model.train()
   running_loss = 0.0

   # Shuffle data at the beginning of each epoch (optional but recommended)
   indices = torch.randperm(len(trainY))

   correct = 0
   total = 0

   for i in range(0, len(trainY), batch_size):
       batch_indices = indices[i:i + batch_size]

       # Extract batch data
       batch_data = trainX[batch_indices]
       batch_labels = trainY[batch_indices]

       # Zero the parameter gradients
       optimizer.zero_grad()

       # Forward pass
       outputs = model(batch_data)

       # Compute loss
       loss = criterion(outputs, batch_labels)

       # Backward pass and optimize
       loss.backward()
       optimizer.step()

       running_loss += loss.item() * len(batch_labels)

       _, predicted = torch.max(outputs, 1)
       total += batch_size
       correct += (predicted == batch_labels).sum().item()

   epoch_loss = running_loss / len(trainY)
   print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Accuracy: {100 *
```

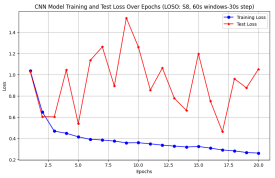
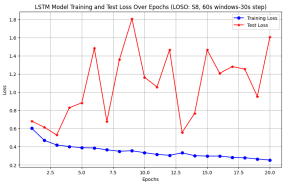
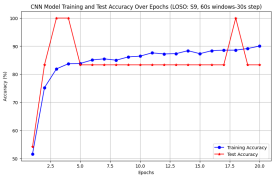
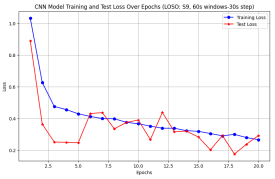
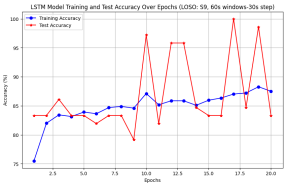
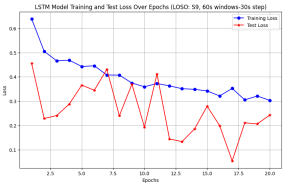
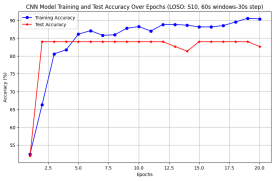
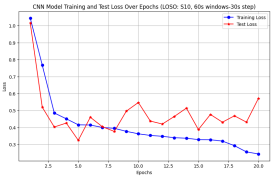
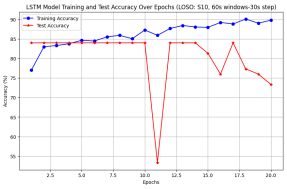
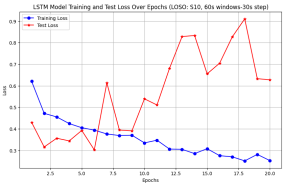
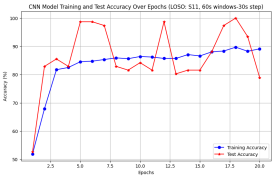
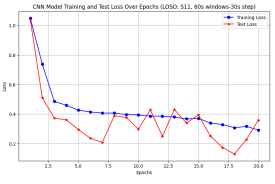
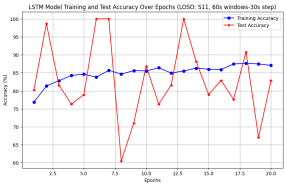
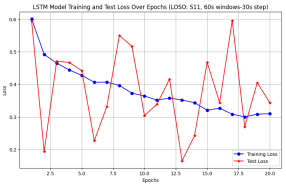
```
correct / total:.2f}%, ({correct} / {total})")
   train_loss.append(epoch_loss)
   train_accuracies.append(100 * correct / total)

   model.eval()
   with torch.no_grad():

       running_loss = 0.0

       indices = torch.randperm(len(testY))

       correct = 0
       total = 0
       for i in range(0, len(testY), batch_size):
           batch_indices = indices[i: i + batch_size]
           batch_data = testX[batch_indices]
           batch_labels = testY[batch_indices]

           outputs = model(batch_data)

           loss = criterion(outputs, batch_labels)
           running_loss += loss.item() * len(batch_labels)



           _, predicted = torch.max(outputs, 1)
           total += batch_size
           correct += (predicted == batch_labels).sum().item()

   epoch_loss = running_loss / len(testY)
   print(f"Test Loss: {epoch_loss:.4f}, Test Accuracy: {100 * correct / total:.2f}%,
({correct} / {total})")
   test_loss.append(epoch_loss)
   test_accuracies.append(100 * correct / total)
```

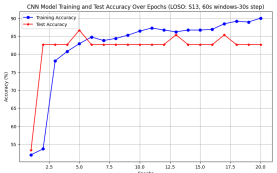
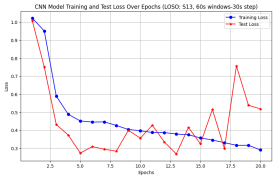
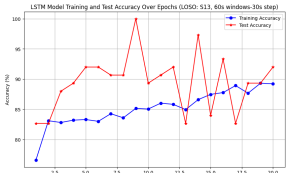
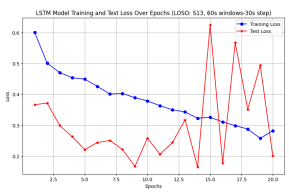
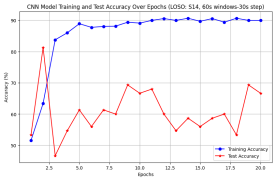
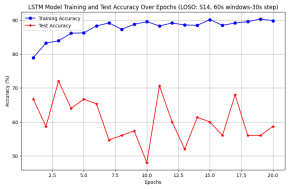
Appendix D: Data and Training Graphs

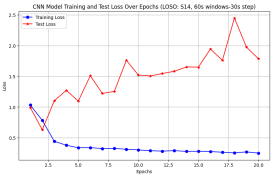
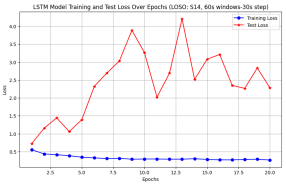
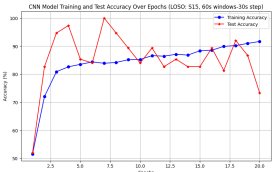
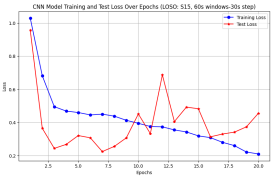
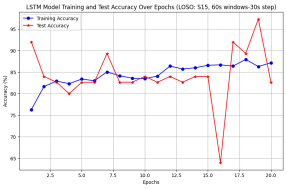
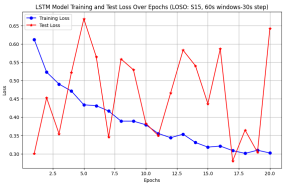
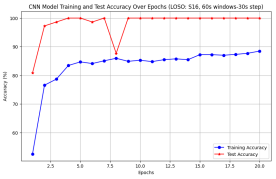
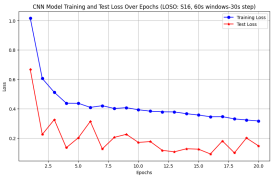
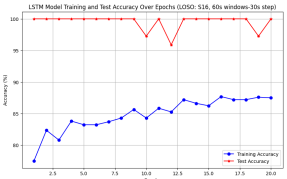
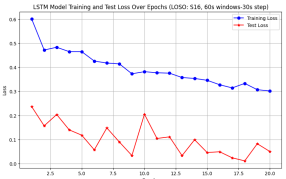
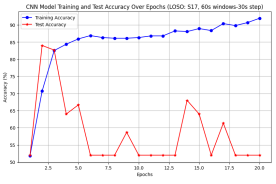
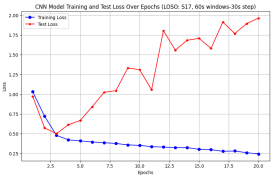
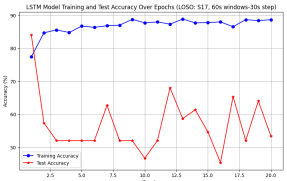
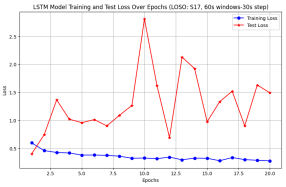
Table D1: LOSO CNN + LSTM (corrected model and results over 20 epochs)

Leave Out Subject	CNN peak Accuracy	LSTM peak Accuracy	CNN Time	LSTM Time
2	81.69 	80.28 	4m 59.0s	10m 8.1s
3	79.45 	84.93 	4m 41.1s	9m 59.0s
4	83.33 	83.33 	5m 22.9s	9m 4.6s
5	100.0	100.0	4m 31.9s	10m 17.7s

	 	 		
6	87.67  	93.15  	5m 28.4s	11m 25.0s
7	82.19  	82.19  	6m 24.8s	9m 24.8s
8	81.08 	77.02 	4m 58.0s	9m 19.3s

				
9	100.0  	100.0  	4m 36.1s	10m 49.9s
10	84.0  	84.0  	4m 47.9s	8m 59.5s
11	100.0  	100.0  	5m 16.2s	8m 38.5s
13	86.67	100.0	5m 17.6s	9m 40.6s

	 	 		
14	81.33 	72.0 	5m 10.4s	11m 16.3s
15	100.0	97.33	8m 13.6s	8m 42.3s
16	100.0	100.0	4m 46.8s	10m 49.5s

				
	 	 		
	 	 		
17	84.0  	84.0  	6m 43.8s	10m 35.5s
avg	88.76	89.22	5m 25.2s	9m 56.7s