Deep Learning Algorithms for EEG Motor Function Classification

Lukas Hager
105601835
ECE C147
l.h.hager@gmail.com

Samuel Perrott
005930926
ECE C147
sperrott@ucla.edu

Reetinav Das
005504665
ECE C147
reetinav@g.ucla.edu

Aryan
705796873
ECE C147
asingh03@g.ucla.edu

Abstract

This project employs a three-pronged approach to understanding, working with, and classifying EEG data. We begin by developing 7 distinct architectures based on convolutional neural networks (CNNs), recurrent neural networks (RNNs), and a particular random convolutional kernel model called ROCKET, amongst others, to determine which model has the highest classification accuracy. Our findings show that CNN architecture alone is most efficient with a test accuracy of 71%. *Subsequently, leveraging this vanilla CNN, we compare the classification accuracy between training the model on a single patient versus training it on all nine patients. Finally, we conduct an analysis of varying sample lengths to determine whether all 1000 time points in our dataset are necessary for constructing accurate models, or if most features can be extracted within shorter time intervals.*

1. Introduction

Electroencephalogram (EEG) technology has emerged as a powerful tool in neuroscience, offering insights into brain activity through relatively non-invasive measurement of electrical impulses generated by the brain [10].

To work with it, however, poses a set of challenges due its high dimensionality, and extensive noise composition. In the past, several CNN-based architectures have been employed to decode EEG data [8, 9]. We explored whether other deep learning architectures could achieve comparable performance. In order to do so, we designed a series of models using Keras, an open source deep-learning Python library, and trained them on a version of BCI's Competition IV dataset [1].

1.1. Data and Preprocessing

The data consists of 2115 training samples and 443 test samples relatively evenly distributed across 9 different patients. There are 4 possible label values corresponds to a patient imagining moving left, right, their foot, or their tongue. Each sample consists of 1000 millisecond steps of time series collected from 22 different electrodes.

We split the provided training data into two portions, 90% used for training and 10% for validation. For our training set, we added Guassian noise as a regularization factor. All our data was also maxpooled by a subsampling factor of 2 to make the time series shorter so that our RNN networks could capture more information. For specific architecture details, please refer to section 4.1 of the appendix "Model Architectures."

1.2. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of neural network particularly effective for processing data with a grid-like topology such as images, making them well-suited for a variety of tasks, including image recognition, natural language processing, and bio-signal analysis [7]. In the context of EEG signal processing, CNNs can be tailored to exploit the spatial and temporal characteristics of the data, enhancing the ability to classify or interpret neural signals.

1D Convolutions

In 2016, DeepMind researchers introduced *WaveNet*, a novel architecture that stacked 1D CNN layers, doubling the dilation rate each layer [11]. This allows lower layers to learn short-term patterns and higher layers to learn long-term patterns, effectively allowing CNNs to process time-series data.

2D Convolutions

When applying 2D convolutions to EEG data, the input is often reshaped or transformed to suit the requirements of a 2D convolutional layer. In this case, the input format is (samples, pooled time steps, 1, channels), where the data is treated as a single-row image with varying time steps and multiple channels. This allows one to process temporal sequences across multiple electrodes simultaneously, enabling the network to learn patterns that span both time and spatial dimensions of the electrode array.

3D Convolutions

3D convolutions extend the concept further by incorporating an additional spatial dimension into the analysis, making them especially suitable for data that includes spatial relationships along with time. For our EEG data, where spatial relationships between the 22 electrodes provide important context for brain activity, the input format for 3D convolutions is (samples, electrode x coord, electrode y coord, pooled time steps, 1). We used a rough approximation of the electrode's relative positions in a 7×7 grid to relate them spatially (see Figure 8). This augmented format treats the EEG data as a 3D volume, where the spatial arrangement of electrodes is preserved along two dimensions and the temporal dimension is considered as the third.

1.3. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are unique in that weights are shared across sequential layers, fostering an web of connections where the output of each computation can influence subsequent computations. This architecture enables the network to maintain a continuous contextual thread throughout the data processing sequence by storing information obtained from all previous time steps into a single input variable $\mathbf{h}_{(t-1)}$ in addition to $\mathbf{x}_t$. Thus, RNNs are particularly adept at handling time-series data, as their unique configuration allows for the temporal dimension of information to be conserved and integrated into the learning process. In our exploration, we evaluated both simple RNN structures alongside hybrid architectures that combined RNN layers with CNNs. We tested a simple RNN, although it did not perform well.

1.3.1 LSTM

Regular RNNs can suffer from vanishing gradients [4] over many cycles. This decreases the model's ability to learn long term relationships. Long-Short-Term-Memory (LSTM) networks address this by incorporating enhancements such as forget and input gates and having two inputs, $\mathbf{c}_{(t-1)}$ and $\mathbf{h}_{(t-1)}$, representing the long-term state and the short-term state, in addition $\mathbf{x}_{(t)}$. These LSTM cells allow the model to regulate the transfer of information better, ensuring that the earliest information in the time series is not lost since RNNs often suffer from short term memory [6].

We decided to test two LSTM variants. We created a 3 Conv2D + 1 LSTM and a 4 Conv2D + 2 LSTM. Our 3 Conv2D + LSTM model used a trim of 800 and a batch size of 64. One major change we did with this model compared to the others was to change our MaxPool padding mode from 'same' to 'valid'. This makes the MaxPool layers downscale the channels instead of keeping them the same, speeding up training time while roughly keeping the same level of performance. We used a fully connected layer after the convolutional layers to decrease the temporal size of the input tensor before passing it into the LSTM layer.

1.3.2 GRU

The *gated recurrent unit* (GRU) cell [2] is a simplified version of the LSTM cell with similar performance [5]. Unlike the standard LSTM, a GRU merges both state vectors into a single vector $\mathbf{h}_{(t)}$, uses a single gate controller for both forget and input gate, and has no output gate. Here, we utilize a hybrid CNN + GRU architecture.

1.4. ROCKET

We tested out a method from sktime called ROCKET, which provides exceptionally fast and accurate time series classification using random convolutional kernels to transform a dataset [3]. From there it can be fitted using a linear classifier. This was a promising idea because of how successful convolutional kernels are at doing time series classification, and our data was time series data. The creators of this module recommend to use SciKit's RidgeCV classifier, a linear classification model, after transforming the dataset, so that's what we used.

2. Results

The best performing models were a pure CNN at 71% and a CNN+LSTM at 72%. The pure CNN was able to achieve similar levels of test accuracy in 50 epochs, while the CNN+LSTM achieved it in 200 epochs, demonstrating the efficiency of a multilayer CNN. We found that a learning rate of 0.001 using the Adam optimizer led to most optimal results. We also implemented several regularization techniques such as L_2 regularization, batch normalization, and dropout layers after each convolutional layer. Please see Table 1 to observe how performance changes with model architecture and hyperparameters and Appendix 2 for layer dimensions and kernel sizes used. In general, our stacked CNN layers had increasing number of filters to compensate for the decreasing dimensionality of doing pooling, and the number of parameters was kept to around $700,000$ to ensure our models trained efficiently.

2.1. Subject-Specific Training

We also trained our top models on Participant 1 only. We wanted to see which of the two models would perform better when being tested on Participant 1. We found that the CNN + LSTM model overfit more as compared to the vanilla CNN (see Table 2).

2.2. Trim Variation

Finally, we assessed our model's performance by inputting data from various sized time intervals. To achieve this, we trained ten different models. The first model was trained using only the initial 100 data points across all channels, followed by the second model trained with the first 200 points, and so forth. Our last model incorporated all 1000 data points. We say that a model trained with only the first 100 time samples has Trim 100. Each model utilized a vanilla CNN architecture comprising four two-dimensional convolutional layers, where each model ran for 50 epochs with a learning rate of 0.001. See Table 4.

3. Discussion

3.1. Models

3.1.1 CNN

Our standalone CNN performed very well. One possible reason is that stacked CNNs tend to be less affected by changes over time compared to RNNs [6]. By aggregating spatially adjacent features over layers, the CNN can prioritize the most pertinent temporal information, potentially making it less susceptible to noise. Our vanilla CNN also outperformed the CNN+GRU architecture. This can be attributed to the fact that increased model complexity doesn't necessarily mean improved performance; in fact, overly complex architectures might overfit on the training data, diminishing their ability to generalize, as the ROCKET and RNN architectures do. Our CNN architecture performs well because it is able to effectively able to process and transform the time-series data as if it were spatial.

3.1.2 CNN+LSTM

Our CNN+LSTM model also performed well. This makes sense because this architecture allows us to combine spatial and temporal information: EEG signals are multidimensional data, typically consisting of multiple channels (electrodes) recording brain activity over time. CNNs can capture spatial information by convolving over different channels, while LSTMs can model temporal dependencies within and across these channels. Combining both CNNs and LSTMs allows for effective utilization of both spatial and temporal information in the EEG data.

3.1.3 ROCKET

The ROCKET (Random Convolutional Kernel Transform) coupled with the RidgeCV linear classifier ended up doing better than the baseline, achieving a training accuracy of 0.95 and a testing accuracy of 0.5. One major limiting factor for the ROCKET model was that we were constrained by our dataset size leading to overfitting as we saw high training accuracies, but poor testing accuracies in comparison. The issue of overfitting is likely due to the dimensionality expansion caused by running many convolutional kernels. This complexity likely leads to overfitting on small EEG datasets, explaining the performance we got.

3.2. RNN

Our standalone RNN model performed very poorly. One reason the RNN performed poorly is because it suffers from poor long term memory, whereas stacking multiple convolutional layers allows for information to be condensed and propagated through the model. Moreover, while RNNs are suitable for shorter sequential data like time series, EEG signals can have complex temporal dynamics that might not be effectively captured by a simple RNN architecture.

3.3. Subject Specific Training

We found that training on data from all nine participants led to better results than training only on one participant. Training on all nine participants helps the model generalize better, in that the model's performance improves because it has learned more generalizable patterns from the additional data. Moreover, training on all nine participants can be seen as a form of data augmentation, as it can help diversify the training data, which in turn can prevent overfitting and improve the model's ability to generalize to new patients. This can also help with noise reduction, allowing our model to hone in on more consistent and reliable patterns in our EEG data.

3.4. Trim Testing

We found that 4 Conv2D was able to reach similar levels of classification accuracy given only the first 300 time steps, instead of using all 1000. We hypothesize that this trend is due to a few reasons. For one, temporal dynamics might come in to play as the brain's response to stimuli or tasks can be time-sensitive. Certain cognitive processes might occur rapidly and be captured within the first few hundred time points of our EEG recording. Moreover, trimming might also serve as a form of noise reduction, as they may contain less noise compared to the rest of the signal. By focusing on the initial portion, you might be inadvertently excluding sections with higher noise levels, thereby improving signal quality. Initial portions may also encode information for future time signals as well. Finally, trimming reduces the dimensionality of our data, which can help with model training.

References

[1] Clemens Brunner, Robert Leeb, Gernot Müller-Putz, Alois Schlögl, and Gert Pfurtscheller. Data sets 2a: 4-class motor imagery. Institute for Knowledge Discovery (Laboratory

of Brain-Computer Interfaces), Graz University of Technology, 2008. EEG, cued motor imagery (left hand, right hand, feet, tongue), [22 EEG channels (0.5-100Hz; notch filtered), 3 EOG channels, 250Hz sampling rate, 4 classes, 9 subjects]. Dataset available online: `https://www.bbci.de/competition/iv/`. 1

[2] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014. 2

[3] Angus Dempster, François Petitjean, and Geoffrey I Webb. ROCKET: Exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5):1454–1495, 2020. 2

[4] Rian Dolphin. Lstm networks — a detailed explanation. 2020. 2

[5] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, and Jurgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, Oct. 2017. 2

[6] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2023. 2, 3

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 1

[8] Vernon J Lawhern, Amelia J Solon, Nicholas R Waytowich, Stephen M Gordon, Chou P Hung, and Brent J Lance. Eegnet: a compact convolutional neural network for eeg-based brain–computer interfaces. *Journal of Neural Engineering*, 15(5):056013, jul 2018. 1

[9] Robin Tibor Schirrmeister, Jost Tobias Springenberg, Lukas Dominique Josef Fiederer, Martin Glasstetter, Katharina Eggensperger, Michael Tangermann, Frank Hutter, Wolfram Burgard, and Tonio Ball. Deep learning with convolutional neural networks for eeg decoding and visualization. *Human Brain Mapping*, 38(11):5391–5420, Aug. 2017. 1

[10] Alois Schlögl, Mel Slater, and Gert Pfurtscheller. Presence research and eeg. 01 2002. 1

[11] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016. 1

4. Appendix

Architecture	Trim	Epochs	Batch Size	Train Accuracy	Validation Accuracy	Test Accuracy
4 Conv2D	550	50	64	0.88	0.67	0.71
3 Conv2D + 1 LSTM	800	200	64	0.79	0.71	0.72
4 Conv2D + 2 LSTM	550	150	32	0.72	0.69	0.68
9 Conv1D + 1 GRU	800	100	64	0.87	0.64	0.62
2 Conv3D + 1 GRU	550	100	64	0.98	0.51	0.52
ROCKET + RidgeCV	1000	—	—	0.95	—	0.50
4 RNN	600	100	64	0.88	0.34	0.35

Table 1. Model accuracies, architecture types, and hyperparameters. Note: all models used a learning rate of 0.001 with the Adam optimizer.

Architecture	Train Accuracy	Validation Accuracy	Accuracy (Subject 1)	Accuracy (Other Subjects)
4 Conv2D	0.96	0.46	0.58	0.38
4 Conv2D + 1 LSTM	0.92	0.67	0.64	0.35

Table 2. Comparison of Top Performing Models Trained on Subject 1.

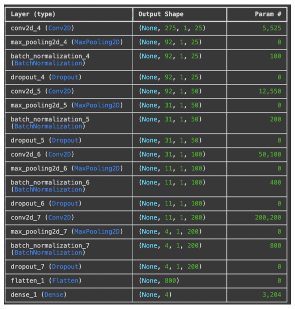
Training Feature		Notes
Convolutional Layer Regularizations	L_2 Regularization	$\lambda = 0.001$
	Activation	ReLU, elu, swish
	Dropout	0.6, 0.2, 0.5
Epochs	Varied	50, 100, 150
Batch Size	Varied	32, 64
Loss function	Cross Entropy Loss	4 classes
Optimizer	Adam	learning rate of 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.99$
Data Augmentations	Max Pooling	pooling window size of 2
	Averaging with Noise	averaging window size of 2, noise scale of 0.5
	Subsampling with Noise	subsampling window size of 2, noise scale of 0.5

Table 3. Hyperparameters and Regularizations Tested

Time Cutoff	**Testing Accuracy**
100	0.5192
200	0.6275
300	0.7201
400	0.7133
500	0.6997
600	0.7065
700	0.6885
800	0.6682
900	0.7088
1000	0.6795

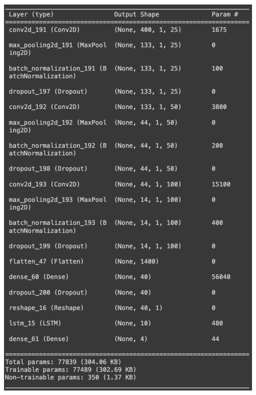
Table 4. Testing Accuracy for Different Trims with 4 Conv2D

4.1. Model Architectures



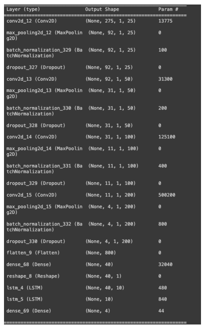
```
Layer (type)                          Output Shape                 Param #
conv2d_4 (Conv2D)                     (None, 275, 1, 25)             5,525
max_pooling2d_4 (MaxPooling2D)        (None, 92, 1, 25)                  0
batch_normalization_4                 (None, 92, 1, 25)                100
(BatchNormalization)
dropout_4 (Dropout)                   (None, 92, 1, 25)                  0
conv2d_5 (Conv2D)                     (None, 92, 1, 50)             12,550
max_pooling2d_5 (MaxPooling2D)        (None, 31, 1, 50)                  0
batch_normalization_5                 (None, 31, 1, 50)                200
(BatchNormalization)
dropout_5 (Dropout)                   (None, 31, 1, 50)                  0
conv2d_6 (Conv2D)                     (None, 31, 1, 100)            50,100
max_pooling2d_6 (MaxPooling2D)        (None, 11, 1, 100)                 0
batch_normalization_6                 (None, 11, 1, 100)               400
(BatchNormalization)
dropout_6 (Dropout)                   (None, 11, 1, 100)                 0
conv2d_7 (Conv2D)                     (None, 11, 1, 200)           200,200
max_pooling2d_7 (MaxPooling2D)        (None, 4, 1, 200)                  0
batch_normalization_7                 (None, 4, 1, 200)                800
(BatchNormalization)
dropout_7 (Dropout)                   (None, 4, 1, 200)                  0
flatten_1 (Flatten)                   (None, 800)                        0
dense_1 (Dense)                       (None, 4)                      3,204
```

Figure 1. 4 Conv2D Architecture



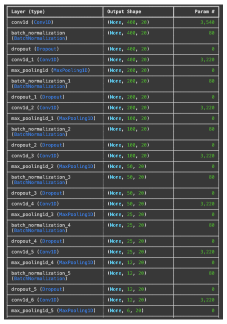
```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_191 (Conv2D)         (None, 400, 1, 25)        1675

 max_pooling2d_191 (MaxPool  (None, 133, 1, 25)        0
 ing2D)

 batch_normalization_191 (B  (None, 133, 1, 25)        100
 atchNormalization)

 dropout_197 (Dropout)       (None, 133, 1, 25)        0

 conv2d_192 (Conv2D)         (None, 133, 1, 50)        3800

 max_pooling2d_192 (MaxPool  (None, 44, 1, 50)         0
 ing2D)

 batch_normalization_192 (B  (None, 44, 1, 50)         200
 atchNormalization)

 dropout_198 (Dropout)       (None, 44, 1, 50)         0

 conv2d_193 (Conv2D)         (None, 44, 1, 100)        15100

 max_pooling2d_193 (MaxPool  (None, 14, 1, 100)        0
 ing2D)

 batch_normalization_193 (B  (None, 14, 1, 100)        400
 atchNormalization)

 dropout_199 (Dropout)       (None, 14, 1, 100)        0

 flatten_47 (Flatten)        (None, 1400)              0

 dense_60 (Dense)            (None, 40)                56040

 dropout_200 (Dropout)       (None, 40)                0

 reshape_16 (Reshape)        (None, 40, 1)             0

 lstm_15 (LSTM)              (None, 10)                480

 dense_61 (Dense)            (None, 4)                 44

=================================================================
Total params: 77839 (304.06 KB)
Trainable params: 77489 (302.69 KB)
Non-trainable params: 350 (1.37 KB)
```

Figure 2. 3 Conv2D + 1 LSTM Architecture



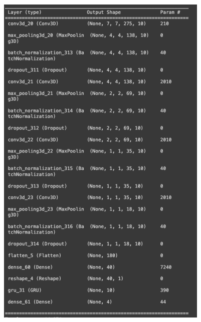
```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_12 (Conv2D)          (None, 275, 1, 25)        13775

 max_pooling2d_12 (MaxPoolin  (None, 92, 1, 25)        0
 g2D)

 batch_normalization_329 (Ba  (None, 92, 1, 25)        100
 tchNormalization)

 dropout_327 (Dropout)       (None, 92, 1, 25)         0

 conv2d_13 (Conv2D)          (None, 92, 1, 50)         31300

 max_pooling2d_13 (MaxPoolin  (None, 31, 1, 50)        0
 g2D)

 batch_normalization_330 (Ba  (None, 31, 1, 50)       200
 tchNormalization)

 dropout_328 (Dropout)       (None, 31, 1, 50)         0

 conv2d_14 (Conv2D)          (None, 31, 1, 100)        125100

 max_pooling2d_14 (MaxPoolin  (None, 11, 1, 100)       0
 g2D)

 batch_normalization_331 (Ba  (None, 11, 1, 100)       400
 tchNormalization)

 dropout_329 (Dropout)       (None, 11, 1, 100)        0

 conv2d_15 (Conv2D)          (None, 11, 1, 200)        500200

 max_pooling2d_15 (MaxPoolin  (None, 4, 1, 200)        0
 g2D)

 batch_normalization_332 (Ba  (None, 4, 1, 200)        800
 tchNormalization)

 dropout_330 (Dropout)       (None, 4, 1, 200)         0

 flatten_9 (Flatten)         (None, 800)               0

 dense_68 (Dense)            (None, 40)                32040

 reshape_8 (Reshape)         (None, 40, 1)             0

 lstm_4 (LSTM)               (None, 40, 10)            480

 lstm_5 (LSTM)               (None, 10)                840

 dense_69 (Dense)            (None, 4)                 44

=================================================================
```

Figure 3. 4 Conv2D + 2 LSTM Architecture



```
Layer (type)                       Output Shape            Param #
conv1d (Conv1D)                    (None, 400, 20)           3,540
batch_normalization                (None, 400, 20)              80
(BatchNormalization)
dropout (Dropout)                  (None, 400, 20)               0
conv1d_1 (Conv1D)                  (None, 400, 20)           3,220
max_pooling1d (MaxPooling1D)       (None, 200, 20)               0
batch_normalization_1              (None, 200, 20)              80
(BatchNormalization)
dropout_1 (Dropout)                (None, 200, 20)               0
conv1d_2 (Conv1D)                  (None, 200, 20)           3,220
max_pooling1d_1 (MaxPooling1D)     (None, 100, 20)               0
batch_normalization_2              (None, 100, 20)              80
(BatchNormalization)
dropout_2 (Dropout)                (None, 100, 20)               0
conv1d_3 (Conv1D)                  (None, 100, 20)           3,220
max_pooling1d_2 (MaxPooling1D)     (None, 50, 20)                0
batch_normalization_3              (None, 50, 20)               80
(BatchNormalization)
dropout_3 (Dropout)                (None, 50, 20)                0
conv1d_4 (Conv1D)                  (None, 50, 20)            3,220
max_pooling1d_3 (MaxPooling1D)     (None, 25, 20)                0
batch_normalization_4              (None, 25, 20)               80
(BatchNormalization)
dropout_4 (Dropout)                (None, 25, 20)                0
conv1d_5 (Conv1D)                  (None, 25, 20)            3,220
max_pooling1d_4 (MaxPooling1D)     (None, 12, 20)                0
batch_normalization_5              (None, 12, 20)               80
(BatchNormalization)
dropout_5 (Dropout)                (None, 12, 20)                0
conv1d_6 (Conv1D)                  (None, 12, 20)            3,220
max_pooling1d_5 (MaxPooling1D)     (None, 6, 20)                 0
```

Figure 4. 9 Conv1D + 1 GRU Architecture



```
 Layer (type)                Output Shape              Param #
=================================================================
 conv3d_20 (Conv3D)          (None, 7, 7, 275, 10)     210

 max_pooling3d_20 (MaxPoolin  (None, 4, 4, 138, 10)    0
 g3D)

 batch_normalization_313 (Ba  (None, 4, 4, 138, 10)    40
 tchNormalization)

 dropout_311 (Dropout)       (None, 4, 4, 138, 10)     0

 conv3d_21 (Conv3D)          (None, 4, 4, 138, 10)     2010

 max_pooling3d_21 (MaxPoolin  (None, 2, 2, 69, 10)     0
 g3D)

 batch_normalization_314 (Ba  (None, 2, 2, 69, 10)     40
 tchNormalization)

 dropout_312 (Dropout)       (None, 2, 2, 69, 10)      0

 conv3d_22 (Conv3D)          (None, 2, 2, 69, 10)      2010

 max_pooling3d_22 (MaxPoolin  (None, 1, 1, 35, 10)     0
 g3D)

 batch_normalization_315 (Ba  (None, 1, 1, 35, 10)     40
 tchNormalization)

 dropout_313 (Dropout)       (None, 1, 1, 35, 10)      0

 conv3d_23 (Conv3D)          (None, 1, 1, 35, 10)      2010

 max_pooling3d_23 (MaxPoolin  (None, 1, 1, 18, 10)     0
 g3D)

 batch_normalization_316 (Ba  (None, 1, 1, 18, 10)     40
 tchNormalization)

 dropout_314 (Dropout)       (None, 1, 1, 18, 10)      0

 flatten_5 (Flatten)         (None, 180)               0

 dense_60 (Dense)            (None, 40)                7240

 reshape_4 (Reshape)         (None, 40, 1)             0

 gru_31 (GRU)                (None, 10)                390

 dense_61 (Dense)            (None, 4)                 44

=================================================================
```

Figure 5. 2 Conv3D + 1 GRU Architecture

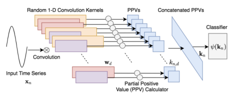
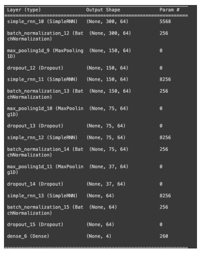
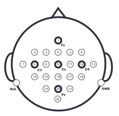


Figure 6. ROCKET + RidgeCV Architecture



```
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_10 (SimpleRNN)   (None, 300, 64)           5568

 batch_normalization_12 (Bat  (None, 300, 64)          256
 chNormalization)

 max_pooling1d_9 (MaxPooling  (None, 150, 64)          0
 1D)

 dropout_12 (Dropout)        (None, 150, 64)           0

 simple_rnn_11 (SimpleRNN)   (None, 150, 64)           8256

 batch_normalization_13 (Bat  (None, 150, 64)          256
 chNormalization)

 max_pooling1d_10 (MaxPoolin  (None, 75, 64)           0
 g1D)

 dropout_13 (Dropout)        (None, 75, 64)            0

 simple_rnn_12 (SimpleRNN)   (None, 75, 64)            8256

 batch_normalization_14 (Bat  (None, 75, 64)           256
 chNormalization)

 max_pooling1d_11 (MaxPoolin  (None, 37, 64)           0
 g1D)

 dropout_14 (Dropout)        (None, 37, 64)            0

 simple_rnn_13 (SimpleRNN)   (None, 64)                8256

 batch_normalization_15 (Bat  (None, 64)               256
 chNormalization)

 dropout_15 (Dropout)        (None, 64)                0

 dense_6 (Dense)             (None, 4)                 260

=================================================================
```

Figure 7. 4 RNN Architecture



7		2				
	8		3		1	
14		9		4		
	15		10		5	
19		16		11		6
	20		17		12	
22		21		18		13

(A) Electrode Placement (B) Electrode Numbering

Figure 8. EEG Electrode Placement and Numbering