Deep Learning for Classification of Imagined Movement with Electroencephalogram Signals

Bassam Safadi
UCLA
samsafadi@g.ucla.edu

Chen-Hsin Yeh
UCLA
joshchyeh860503@g.ucla.edu

Hongseok Lee
UCLA
hslee.paul@g.ucla.edu

Jayanth Shreekumar
UCLA
jayshreekumar98@g.ucla.edu

Abstract

In this paper, we explored the use of Deep Learning models for classification of EEG signals. Our models range from standard CNN models and LSTM layers to more complex architectures such as transformers. Results indicate that LSTM-based architectures perform the best, with the spatio-temporal CNN+LSTM architecture achieving the highest accuracy of 74.7%.

1. Introduction

Our goal for this project was to predict the imagined movements from raw EEG data as accurately as possible over all subjects. This is a harder task than per-subject classification as it is harder to generalize across all subjects as they tend to have slightly different signals for the same action. We have attempted a wide variety of methods, models, and processing of the data in order to achieve this. Models ranged from convolutional neural networks to attention-based models, and we have attempted to process and augment the data using methods such as autoencoders and GANs. All the models were implemented in PyTorch[7] except for two models: spatio-temporal CNN and spatio-temporal CNN with attention which were coded in TensorFlow[1]. We discuss our methods in further depth in the following section.

1.1. The Dataset

The dataset[3] consisted of 2115 training samples and 443 test samples and each sample belonged to one of four classes: Moving the left arm, right arm, feet, and tongue. 22 electrodes were used to extract neural data and so each data sample had 22 channels and a length of 1000. As this might not be enough data to perform deep learning tasks, pre-processing was done as described in the next section.

1.2. Data Preprocessing

Our experiments were conducted on two sets of data, the original dataset consisting of 2115 training samples and 443 test samples, and a preprocessed dataset consisting of 8460 training samples and 1772 test samples.

Preprocessing consisted of four parts:

1. Trimming and discarding the second half of each sample as they were very similar and did not have distinguishing features. This leads to samples of length 500.

2. Maxpooling and thus downsampling the trimmed samples every two time-steps. This leads to samples of length 250.

3. Averaging the trimmed samples every two time-steps and adding Gaussian noise to it. This leads to samples of length 250. The outputs of 2 and 3 are concatenated across samples to obtain twice the number of original samples, each of length 250.

4. Subsampling the trimmed samples every two time-steps to obtain twice the number of samples. Finally, this is concatenated across samples with the output of 2 and 3 to obtain 4x the number of original samples, each of length 250.

2. Architectures

2.1. Convolutional Neural Network

The CNN that we utilized was made up of 5 layers: 4 convolutional layers and a fully connected layer. It also incorporated the ELU activation layers and max pooling layers along with batch normalization and dropout of 0.5 after

each CNN layer. The weights for all CNN networks discussed in this paper were initialized using a normal distribution of $\mathcal{N}(0, 0.02)$ for the convolutional layers and a normal distribution of $\mathcal{N}(1, 0.02)$ for the batchnorm layers. Also, early stopping was implemented which monitored validation loss with a patience of 10.

2.2. DeepConvNet

DeepConvNet[9] is a CNN network that has a single temporal convolutional layer followed by a convolution across all channels. Three standard conv-maxpool blocks are used after a maxpool layer is used on the output of the first block. ELU activation layers are used on all the blocks along with batchnorm layers and droput of 0.5 and finally, a dense layer followed by a softmax layer is utilized to obtain probabilities.

2.3. ShallowConvNet

ShallowConvNet[9] is proposed by Schirrmeister et al. as an alternative to DeepConvNet and it was designed specifically to utilize band power features. The temporal convolutional layer here has a larget kernel compared to DeepConvNet as well as log and square non-linearities rather than standard pooling layers. A single dropout layer and a single dropout layer was used, ending in a dense layer and softmax activation to obtain probabilities.

2.4. Spatio-Temporal CNN using multiple kernel scopes

This architecture has a temporal layer at first which opts four kernels with all different sizes(65,41,27,17). Followed by this temporal layer, four different outputs generated by each kernel are stacked. This stack propagates through a spatial layer, non-linearity + max-pool layer and the final classifier convolution layer defines its class. Using various kernel scopes enhances the generality of our classification.

2.5. Spatio-Temporal CNN with LSTM

Unlike the previous Spatio-Temporal CNN with multiple kernel scopes, this architecture starts with two subsequent spatial layers followed by 4 temporal layers. The output is then fed to a pooling layer,which connects to a LSTM layer. The LSTM used here is a bidirectional LSTM with a 76 hidden layers. The output of the LSTM layer gets flattened and propagates a dense layer to print the overall output.

2.6. Spatio-Temporal CNN with LSTM+Attention

Starting from the intuition that the last layer of the LSTM has more reliable data, an attention layer was added to the previous Spatio-Temporal CNN with LSTM architecture. The last layer of the LSTM output acts as a weight to the LSTM output by using their dot product as further data. This dot product goes in to a softmax classifier and a dense layer, providing the overall output. However, this architecture does not have significant improvement from the previous architecture without an attention stage.

2.7. Deep Convolutional Generative Adversarial Networks

Classification of EEG data is a challenging task due to the small dataset sizes that are available for use. Therefore, data augmentation can be a useful and much needed solution to this problem. We worked on two different GANs, the DCGAN as well as the WGAN with gradient penalty.

DCGANs [8] were proposed by Radford et al. in an attempt to generate high resolution images using deeper generative models. They modified the original GAN architecture in 3 significant ways:

- Using strided convolutions for spatial downsampling in the discriminator instead of pooling layers and using fractional-strided convolutions in the generator for upsampling.
- Eliminating the use of fully connected layers on top of the convolutional layers
- Utilizing batch normalization[5] for normalizing input and stabilizing training

2.8. Wasserstein Generative Adversarial Networks

As the DCGAN is tuned to work specifically on images, we also implemented the WGAN[2] as an alternative. It has been used previously on EEG data for emotion recognition[6]. The changes made for the WGAN when compared to the DCGAN are:

- Removed Sigmoid activation in the last layer of the discriminator.
- Wasserstein distance used as loss function rather than log loss.
- parameter clipping

Additionally, the WGAN training procedure was improved using gradient penalty [4], and so, we incorporated this into our model. Finally, to be able to generate samples belonging to specific classes, we built a conditional WGAN-GP model capable of this task.

2.9. Transformer

We implemented a Transformer [10] with blocks of self-attention and feed-forward networks as an encoder, and a simple feed-forward layer with a softmax activation as a classification head. The self-attention in the encoders are multiheaded, and experimented with different hidden sizes, numbers of heads, and numbers of blocks. We felt that the

transformer had potential to do well because of its success in sequence-type problems like natural language processing. Having attention weights between each of the elements of the sequence in order to give the classifier a holistic understanding of the sequence intuitively made sense, which is why we pursued it.

Initially, we treated every time sample with each of the 22 channels as one individual part of the entire sequence. However, we realized that the sample on its own does not have enough information. Hence, we split the sequence into small chunks (the length of which was experimented on). At first, we tried an encoder layer that would take the chunk of size $chunk\_length \times 22$ and convert it to an encoded vector. We tried both a feed-forward and convolutional encoder, but neither of them performed well, not even decreasing the training loss.

We thought this might be due to a lack of available training data, so instead of training the encoder as part of the transformer, we trained a separate autoencoder that uses convolutional layers to extract features from chunks of the sequence. We trained the autoencoder to reconstruct the chunk as best as possible from the features it extracts, then used the trained encoder portion of the autoencoder to extract features from the chunks of the overall sequence for the transformer. This allowed the transformer to at least decrease training loss, but it overfit very heavily even with extremely high dropout (0.8) and a very small model.

3. Results

The Spatio-Temporal CNN with LSTM performed the best out of our models with an accuracy of 74.7%. Following that, the Spatio-Temporal CNN using multiple kernel scopes and the Spatio-Temporal CNN with LSTM and Attention achieved test accuracies of 71.7% and 70.9% respectively. DeepConvNet followed with a testing accuracy of 69% when using the GAN augmented dataset. ShallowConvNet and Transformer achieve accuracies of 71% and 50% when trained with the preprocessed dataset. For a full list of results, refer to Table 1.

4. Discussion

4.1. CNNs

We expected DeepConvNet and ShallowConvNet, CNNs which were specifically modified for EEG inputs, to perform better than the standard CNN model. This is the case when we use the dataset without pre-processing steps but the CNN performs better when pre-processed data is used. We believe that the temporal convolution layer in the former networks required details that were trimmed out during the pre-processing stage causing this phenomenon. Additionally, the WGAN based data augmentation does not help the standard CNN but has an impact on the DeepConvNet architecture. This suggests that the WGAN is generating signals that do have temporal information.

4.2. DCGAN and WGAN

The DCGAN was specifically designed to handle images and is a poor model for generating data with temporal dependence. This was confirmed when the DCGAN failed to converge, generating very simple outputs like a periodic wave or a straight line that did not have information. The WGAN-GP performed much better, and we could generate realistic-looking samples of length 250. However, the main problem with GANs is that visual inspection is the best way to estimate its performance rather than relying on loss which tends to oscillate. However, choosing a GAN model over visual inspection for EEG data is quite challenging. Therefore, we simply used a WGAN model trained until 750 epochs to generate data. Note that there might have been models that had better generations, we simply could not figure out which WGAN model(trained until what epoch) to use to generate samples. Also, the amount of data to train the WGAN itself is far too small to capture a generalized distribution of the four action labels. We believe that more data might help the WGAN to perform better and generate more realistic signals.

Since the results of the WGAN were promising, we built a conditional WGAN GP that was capable of utilizing sample labels to generate data specific to a certain class. We generated a total of 256 samples per class, each of length 250 using cWGAN-GP. Note that we did not experiment with generating samples of the original length of 1000 due to computational constraints, but we believe this might be a good idea to explore further.

4.3. Transformer

Contrary to our expectations, the transformer ended up performing poorly despite a lot of experimentation with model architecture and data embedding. Based on how the model overfit the data extremely heavily, a reasonable conclusion is that the size of the dataset is not enough to train the transformer to generalize well. It is also possible that the relationship between different parts of the EEG sequence determined from the self-attention wasn't actually that important to the classification of the imagined movement.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensor-

Flow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 1

[2] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017. 2

[3] C. Brunner, R. Leeb, G. Müller-Putz, A. Schlögl, and G. Pfurtscheller. Bci competition 2008–graz data set a. *Institute for Knowledge Discovery (Laboratory of Brain-Computer Interfaces), Graz University of Technology*, 16:1–6, 2008. 1

[4] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved training of wasserstein gans. *Advances in neural information processing systems*, 30, 2017. 2

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 2

[6] Y. Luo, L.-Z. Zhu, Z.-Y. Wan, and B.-L. Lu. Data augmentation for enhancing eeg-based emotion recognition with deep generative models. *Journal of Neural Engineering*, 17(5):056021, 2020. 2

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 1

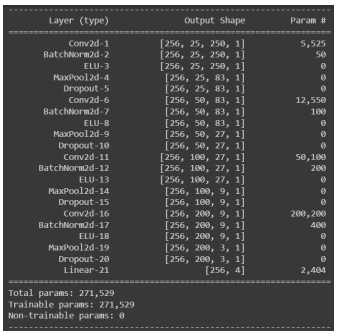
[8] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. 2

[9] R. T. Schirrmeister, J. T. Springenberg, L. D. J. Fiederer, M. Glasstetter, K. Eggensperger, M. Tangermann, F. Hutter, W. Burgard, and T. Ball. Deep learning with convolutional neural networks for eeg decoding and visualization. *Human brain mapping*, 38(11):5391–5420, 2017. 2

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 2

Table 1: Best testing accuracy (%) results over all subjects in BCI IV dataset 2a.

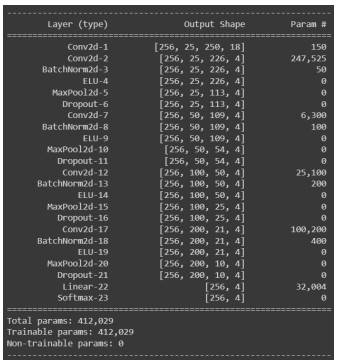
Model	**Raw Data**	**With Preprocessing**	**Using GAN**
Convolutional Neural Network	63.0	71.0	71.0
DeepConvNet	58.0	65.0	69.0
ShallowConvNet	61.0	67.0	69.0
Spatio-Temporal CNN using multiple kernel scopes	71.7	N/A	N/A
Spatio-Temporal CNN with LSTM	74.7	N/A	N/A
Spatio-Temporal CNN with LSTM+Attention	70.9	N/A	N/A
Transformer	31.0	50.0	N/A



Layer (type)	Output Shape	Param #
Conv2d-1	[256, 25, 250, 1]	5,525
BatchNorm2d-2	[256, 25, 250, 1]	50
ELU-3	[256, 25, 250, 1]	0
MaxPool2d-4	[256, 25, 83, 1]	0
Dropout-5	[256, 25, 83, 1]	0
Conv2d-6	[256, 50, 83, 1]	12,550
BatchNorm2d-7	[256, 50, 83, 1]	100
ELU-8	[256, 50, 83, 1]	0
MaxPool2d-9	[256, 50, 27, 1]	0
Dropout-10	[256, 50, 27, 1]	0
Conv2d-11	[256, 100, 27, 1]	50,100
BatchNorm2d-12	[256, 100, 27, 1]	200
ELU-13	[256, 100, 27, 1]	0
MaxPool2d-14	[256, 100, 9, 1]	0
Dropout-15	[256, 100, 9, 1]	0
Conv2d-16	[256, 200, 9, 1]	200,200
BatchNorm2d-17	[256, 200, 9, 1]	400
ELU-18	[256, 200, 9, 1]	0
MaxPool2d-19	[256, 200, 3, 1]	0
Dropout-20	[256, 200, 3, 1]	0
Linear-21	[256, 4]	2,404

Total params: 271,529
Trainable params: 271,529
Non-trainable params: 0

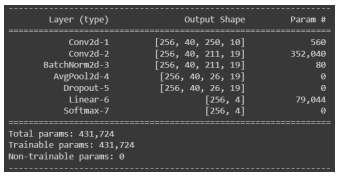
(a) Standard CNN Model, Adam, 0.0001



Layer (type)	Output Shape	Param #
Conv2d-1	[256, 25, 250, 18]	150
Conv2d-2	[256, 25, 226, 4]	247,525
BatchNorm2d-3	[256, 25, 226, 4]	50
ELU-4	[256, 25, 226, 4]	0
MaxPool2d-5	[256, 25, 113, 4]	0
Dropout-6	[256, 25, 113, 4]	0
Conv2d-7	[256, 50, 109, 4]	6,300
BatchNorm2d-8	[256, 50, 109, 4]	100
ELU-9	[256, 50, 109, 4]	0
MaxPool2d-10	[256, 50, 54, 4]	0
Dropout-11	[256, 50, 54, 4]	0
Conv2d-12	[256, 100, 50, 4]	25,100
BatchNorm2d-13	[256, 100, 50, 4]	200
ELU-14	[256, 100, 50, 4]	0
MaxPool2d-15	[256, 100, 25, 4]	0
Dropout-16	[256, 100, 25, 4]	0
Conv2d-17	[256, 200, 21, 4]	100,200
BatchNorm2d-18	[256, 200, 21, 4]	400
ELU-19	[256, 200, 21, 4]	0
MaxPool2d-20	[256, 200, 10, 4]	0
Dropout-21	[256, 200, 10, 4]	0
Linear-22	[256, 4]	32,004
Softmax-23	[256, 4]	0

Total params: 412,029
Trainable params: 412,029
Non-trainable params: 0

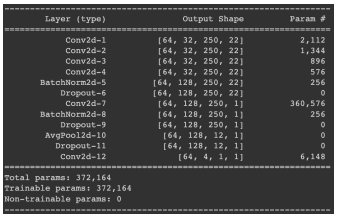
(b) DeepConvNet, Adam, 0.0001



Layer (type)	Output Shape	Param #
Conv2d-1	[256, 40, 250, 10]	560
Conv2d-2	[256, 40, 211, 19]	352,040
BatchNorm2d-3	[256, 40, 211, 19]	80
AvgPool2d-4	[256, 40, 26, 19]	0
Dropout-5	[256, 40, 26, 19]	0
Linear-6	[256, 4]	79,044
Softmax-7	[256, 4]	0

Total params: 431,724
Trainable params: 431,724
Non-trainable params: 0

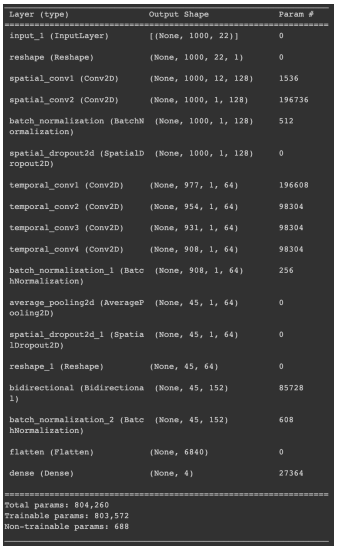
(c) ShallowConvNet, Adam, 0.0001



Layer (type)	Output Shape	Param #
Conv2d-1	[64, 32, 250, 22]	2,112
Conv2d-2	[64, 32, 250, 22]	1,344
Conv2d-3	[64, 32, 250, 22]	896
Conv2d-4	[64, 32, 250, 22]	576
BatchNorm2d-5	[64, 128, 250, 22]	256
Dropout-6	[64, 128, 250, 22]	0
Conv2d-7	[64, 128, 250, 1]	360,576
BatchNorm2d-8	[64, 128, 250, 1]	256
Dropout-9	[64, 128, 250, 1]	0
AvgPool2d-10	[64, 128, 12, 1]	0
Dropout-11	[64, 128, 12, 1]	0
Conv2d-12	[64, 4, 1, 1]	6,148

Total params: 372,164
Trainable params: 372,164
Non-trainable params: 0

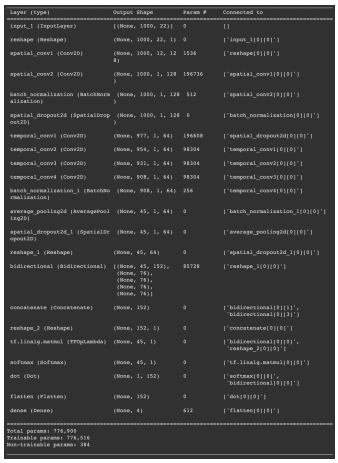
(d) Spatio-Temporal CNN Using Multiple Kernel Scopes, Adam, 0.001



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 1000, 22)]	0
reshape (Reshape)	(None, 1000, 22, 1)	0
spatial_conv1 (Conv2D)	(None, 1000, 12, 128)	1536
spatial_conv2 (Conv2D)	(None, 1000, 1, 128)	196736
batch_normalization (BatchNormalization)	(None, 1000, 1, 128)	512
spatial_dropout2d (SpatialDropout2D)	(None, 1000, 1, 128)	0
temporal_conv1 (Conv2D)	(None, 977, 1, 64)	196608
temporal_conv2 (Conv2D)	(None, 954, 1, 64)	98304
temporal_conv3 (Conv2D)	(None, 931, 1, 64)	98304
temporal_conv4 (Conv2D)	(None, 908, 1, 64)	98304
batch_normalization_1 (BatchNormalization)	(None, 908, 1, 64)	256
average_pooling2d (AveragePooling2D)	(None, 45, 1, 64)	0
spatial_dropout2d_1 (SpatialDropout2D)	(None, 45, 1, 64)	0
reshape_1 (Reshape)	(None, 45, 64)	0
bidirectional (Bidirectional)	(None, 45, 152)	85728
batch_normalization_2 (BatchNormalization)	(None, 45, 152)	608
flatten (Flatten)	(None, 6840)	0
dense (Dense)	(None, 4)	27364

Total params: 804,260
Trainable params: 803,572
Non-trainable params: 688

(e) Spatio-Temporal CNN Using LSTM, Adam, 0.0001



Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 1000, 22)]	0	[]
reshape (Reshape)	(None, 1000, 22, 1)	0	['input_1[0][0]']
spatial_conv1 (Conv2D)	(None, 1000, 12, 12 8)	1536	['reshape[0][0]']
spatial_conv2 (Conv2D)	(None, 1000, 1, 128)	196736	['spatial_conv1[0][0]']
batch_normalization (BatchNormalization)	(None, 1000, 1, 128)	512	['spatial_conv2[0][0]']
spatial_dropout2d (SpatialDropout2D)	(None, 1000, 1, 128)	0	['batch_normalization[0][0]']
temporal_conv1 (Conv2D)	(None, 977, 1, 64)	196608	['spatial_dropout2d[0][0]']
temporal_conv2 (Conv2D)	(None, 954, 1, 64)	98304	['temporal_conv1[0][0]']
temporal_conv3 (Conv2D)	(None, 931, 1, 64)	98304	['temporal_conv2[0][0]']
temporal_conv4 (Conv2D)	(None, 908, 1, 64)	98304	['temporal_conv3[0][0]']
batch_normalization_1 (BatchNormalization)	(None, 908, 1, 64)	256	['temporal_conv4[0][0]']
average_pooling2d (AveragePooling2D)	(None, 45, 1, 64)	0	['batch_normalization_1[0][0]']
spatial_dropout2d_1 (SpatialDropout2D)	(None, 45, 1, 64)	0	['average_pooling2d[0][0]']
reshape_1 (Reshape)	(None, 45, 64)	0	['spatial_dropout2d_1[0][0]']
bidirectional (Bidirectional)	((None, 45, 152), (None, 76), (None, 76), (None, 76), (None, 76))	85728	['reshape_1[0][0]']
concatenate (Concatenate)	(None, 152)	0	['bidirectional[0][1]', 'bidirectional[0][3]']
reshape_2 (Reshape)	(None, 152, 1)	0	['concatenate[0][0]']
tf.linalg.matmul (TFOpLambda)	(None, 45, 1)	0	['bidirectional[0][0]', 'reshape_2[0][0]']
softmax (Softmax)	(None, 45, 1)	0	['tf.linalg.matmul[0][0]']
dot (Dot)	(None, 1, 152)	0	['softmax[0][0]', 'bidirectional[0][0]']
flatten (Flatten)	(None, 152)	0	['dot[0][0]']
dense (Dense)	(None, 4)	612	['flatten[0][0]']

Total params: 776,905
Trainable params: 774,516
Non-trainable params: 384

(f) Spatio-Temporal CNN Using LSTM + Attention, Adam, 0.0001

GENERATOR

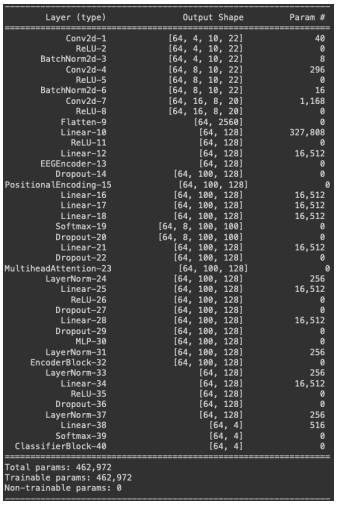
WGANGen(
(label_emb): Embedding(4, 4)
concatenate(labelLemb, input)
(fc1): Linear(in_features=104, out_features=4928, bias=True)
(bnorm1): BatchNorm1d(4928, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu1): LeakyReLU(negative_slope=0.3)
(reshape)input, (64, 352, 14))
(dropout1): Dropout(p=0.5, inplace=False)
(deconv1): ConvTranspose1d(352, 176, kernel_size=(5,), stride=(2,), padding=(1,), output_padding=(1,), bias=False)
(bnorm2): BatchNorm1d(176, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu2): LeakyReLU(negative_slope=0.3)
(dropout2): Dropout(p=0.5, inplace=False)
(deconv2): ConvTranspose1d(176, 88, kernel_size=(5,), stride=(2,), padding=(1,), output_padding=(1,), bias=False)
(bnorm3): BatchNorm1d(88, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu3): LeakyReLU(negative_slope=0.3)
(dropout3): Dropout(p=0.5, inplace=False)
(deconv3): ConvTranspose1d(88, 44, kernel_size=(5,), stride=(2,), padding=(2,), output_padding=(1,), bias=False)
(bnorm4): BatchNorm1d(44, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu4): LeakyReLU(negative_slope=0.3)
(dropout4): Dropout(p=0.5, inplace=False)
(deconv4): ConvTranspose1d(44, 22, kernel_size=(5,), stride=(2,), padding=(1,), output_padding=(1,), bias=False)
(tanh): TanH()
)

DISCRIMINATOR

WGANDisc(
(label_emb): Embedding(4, 4)
(scale): Linear(in_features=4, out_features=5500, bias=True)
(reshape)label_emb, (64, 22, 250))
concatenate(input, label_emb)
(conv1): Conv1d(44, 64, kernel_size=(2,), stride=(2,))
(bnorm1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu1): LeakyReLU(negative_slope=0.3)
(conv2): Conv1d(64, 128, kernel_size=(2,), stride=(2,))
(bnorm2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu2): LeakyReLU(negative_slope=0.3)
(conv3): Conv1d(128, 256, kernel_size=(2,), stride=(2,))
(bnorm3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu3): LeakyReLU(negative_slope=0.3)
(conv4): Conv1d(256, 512, kernel_size=(2,), stride=(2,))
(bnorm4): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu4): LeakyReLU(negative_slope=0.3)
(dropout1): Dropout(p=0.5, inplace=False)
(fc1): Linear(in_features=7680, out_features=1, bias=True)
)

The DCGAN will have the same architecture as WGAN but will have a final sigmoid activation layer in the discriminator.

(g) Wasserstein GAN, Adam, 0.0001, n_critic=10



Layer (type)	Output Shape	Param #
Conv2d-1	[64, 4, 10, 22]	40
ReLU-2	[64, 4, 10, 22]	0
BatchNorm2d-3	[64, 4, 10, 22]	8
Conv2d-4	[64, 8, 10, 22]	296
ReLU-5	[64, 8, 10, 22]	0
BatchNorm2d-6	[64, 8, 10, 22]	16
Conv2d-7	[64, 16, 8, 20]	1,168
ReLU-8	[64, 16, 8, 20]	0
Flatten-9	[64, 2560]	0
Linear-10	[64, 128]	327,808
ReLU-11	[64, 128]	0
Linear-12	[64, 128]	16,512
EEGEncoder-13	[64, 128]	0
Dropout-14	[64, 100, 128]	0
PositionalEncoding-15	[64, 100, 128]	0
Linear-16	[64, 100, 128]	16,512
Linear-17	[64, 100, 128]	16,512
Linear-18	[64, 100, 128]	16,512
Softmax-19	[64, 8, 100, 100]	0
Dropout-20	[64, 8, 100, 100]	0
Linear-21	[64, 100, 128]	16,512
Dropout-22	[64, 100, 128]	0
MultiheadAttention-23	[64, 100, 128]	0
LayerNorm-24	[64, 100, 128]	256
Linear-25	[64, 100, 128]	16,512
ReLU-26	[64, 100, 128]	0
Dropout-27	[64, 100, 128]	0
Linear-28	[64, 100, 128]	16,512
Dropout-29	[64, 100, 128]	0
MLP-30	[64, 100, 128]	0
LayerNorm-31	[64, 100, 128]	256
EncoderBlock-32	[64, 100, 128]	0
LayerNorm-33	[64, 128]	256
Linear-34	[64, 128]	16,512
ReLU-35	[64, 128]	0
Dropout-36	[64, 128]	0
LayerNorm-37	[64, 128]	256
Linear-38	[64, 4]	516
Softmax-39	[64, 4]	0
ClassifierBlock-40	[64, 4]	0

Total params: 462,972
Trainable params: 462,972
Non-trainable params: 0

(h) Transformer, Adam, 0.0001

Figure 1: Architectures, Optimizer, Learning Rate