2021 43rd Annual International Conference of the
IEEE Engineering in Medicine & Biology Society (EMBC)
Oct 31 - Nov 4, 2021. Virtual Conference

EEG-GNN: Graph Neural Networks for Classification of Electroencephalogram (EEG) Signals

Andac Demir[1], Toshiaki Koike-Akino[2], Ye Wang[2], Masaki Haruna[3], Deniz Erdogmus[1]

***Abstract*— Convolutional neural networks (CNN) have been frequently used to extract subject-invariant features from electroencephalogram (EEG) for classification tasks. This approach holds the underlying assumption that electrodes are equidistant analogous to pixels of an image and hence fails to explore/exploit the complex functional neural connectivity between different electrode sites. We overcome this limitation by tailoring the concepts of convolution and pooling applied to 2D grid-like inputs for the functional network of electrode sites. Furthermore, we develop various graph neural network (GNN) models that project electrodes onto the nodes of a graph, where the node features are represented as EEG channel samples collected over a trial, and nodes can be connected by weighted/unweighted edges according to a flexible policy formulated by a neuroscientist. The empirical evaluations show that our proposed GNN-based framework outperforms standard CNN classifiers across ErrP, and RSVP datasets, as well as allowing neuroscientific interpretability and explainability to deep learning methods tailored to EEG related classification problems. Another practical advantage of our GNN-based framework is that it can be used in EEG channel selection, which is critical for reducing computational cost, and designing portable EEG headsets.**

***Index Terms*— Graph neural networks (GNN), Convolutional neural networks (CNN), electroencephalogram (EEG) classification.**

I. Introduction

Deep convolutional neural networks (CNN) have been frequently used in extraction of task relevant features from physiological data, such as electroencephalogram (EEG) and electromyogram (EMG) signals, to devise more robust human-machine interfaces (HMI). However a generic CNN is good at learning features from grid-like data structures like images, where each pixel is equidistant to neighboring pixels. Feeding EEG data to a CNN typically uses two methodologies:

1) Applying 2D convolutions to each EEG trial, which is presented a pseudo-image $\mathbb{R}^{C\times T}$, where C denotes number of EEG channels, and T denotes number of discretized time samples, which effectively treats the EEG channels and time samples like spatial dimensions for CNN processing.
2) Applying 1D convolutions along only the time axis of the EEG trial, while treating the EEG channels as separate channels of the CNN processing.

In the 2D input case, arbitrarily stacking the time samples for each of the EEG channels into a single row ignores standard coordinates of EEG electrodes on a spherical head model. Neglecting functional neural connectivity between different parts of the brain oversimplifies the EEG feature extraction process. On the other hand, CNNs using 1D convolutions with the EEG channels used as CNN channels underperform compared to the 2D convolutional case, since CNNs are only good at learning local spatial patterns, and not an effective approach to explore long-term temporal dependencies for not being sensitive to the order of timesteps.

The main contributions of this work over the existing studies are as follows:

- EEG-GNN properly maps the network of the brain as a graph, where each electrode used to collect EEG data according to intl. 10-5 system represents a node in the graph and time samples acquired from an electrode corresponds to that node's feature vector.
- Adjacency matrix of this graph can be constructed flexibly, e.g., i) every pair of nodes is connected by an unweighted edge, ii) every pair of nodes is connected by an edge weighted by the functional neural connectivity factor, which is the Pearson correlation coefficient between the feature vectors of the two nodes, iii) a sparse adjacency matrix can be designed under the constraint only nodes that are closer than a heuristic distance are connected, or iv) a sparse adjacency matrix can be constructed via k-nearest neighbors (k-NNG).
- One of the major drawbacks to using CNNs to classify EEG data is they fail to provide a brain connectivity mapping by identifying Regions of Interests (ROIs), whereas EEG-GNN can learn and visualize the connectivity between salient nodes, which addresses a critical issue of neuroscientific interpretability.

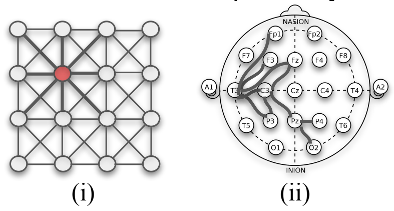


Fig. 1: (i) is 2D CNN convolution, and the graph structure is analogous to pixels of an image. We know functional neural connectivity between electrode sites is arbitrary like in (ii). Cognitive activity of one hemisphere or lobe is more salient than the others for a specific EEG classification task, and there are intricate relationships between different electrode sites, which need to be explored.

[1]A. Demir, and D. Erdogmus are with Cognitive Systems Laboratory, Electrical and Computer Engineering Department, Northeastern University, Boston, MA 02115, USA (e-mail: demir.a@ece.neu.edu).

[2]T. Koike-Akino and Y. Wang are with Mitsubishi Electric Research Laboratories (MERL), Cambridge, MA 02139, USA (e-mail: {koike, yewang}@merl.com).

[3]M. Haruna is with Advanced Technology R&D Center, Mitsubishi Electric Corporation (MELCO), Amagasaki, Hyogo, Japan.

II. Preliminary to GNNs

A graph is a topological space which arises from a triplet of vertices, edges and weights, $G = (V, E, W)$. Vertices are a set of n labels where n is the total number of nodes: $V(G) = \{1, \ldots, n\}$, where $n := |V|$. Edges, e_{ij}, are ordered pairs of labels (i, j). Weights, w_{ij}, are associated to edges. w_{ij} represents strength of the influence of node j on node i.

The adjacency matrix of G is a typically sparse matrix $\mathbf{A}$ with entries given by $\mathbf{A}_{ij} = w_{ij}$ for all i, j. If G is symmetric, then $\mathbf{A}$ is symmetric: $\mathbf{A} = \mathbf{A}^{\mathbf{T}}$. For the particular case in which G is unweighted, weights are interpreted as units: $\mathbf{A}_{ij} = 1$, if node i and node j are connected, and otherwise 0. In an unweighted graph, $\mathbf{A}$ has the same sparsity pattern, except that now all nonzero entries will be 1.

The degree of a node i, denoted as d_i, is the sum of weights in all of the incident edges that connect i to its neighbors. The degrees of all nodes are grouped into the degree matrix $\mathbf{D}$. This is a diagonal matrix whose diagonal entry is the d_i. $\mathbf{D}$ can also be written in terms of the adjacency matrix, because the diagonal of $\mathbf{D}$ corresponds to the sum of the rows of $\mathbf{A}$.

Having defined adjacency and degree matrices, we can now define the Laplacian matrix, $\mathbf{L}$, of a graph,

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{1}$$

The Laplacian can also be written explicitly in terms of the weights of the graph. Since $\mathbf{D}$ is diagonal, off-diagonal entries are simply given by opposite values of the corresponding entries of $\mathbf{A}$. $\mathbf{L}_{ij} = -\mathbf{A}_{ij} = -w_{ij}$. Assuming the graph has no self loops, the diagonal entries of $\mathbf{A}$ are null, hence the diagonal entries of $\mathbf{L}$ are simply the diagonal entries of $\mathbf{D}$: $\mathbf{L}_{ij} = d_i$.

Normalized versions of $\mathbf{A}$ and $\mathbf{L}$ are also utilized as matrix representations of graphs. Both normalized adjacency and Laplacian are defined by pre-emposed multiplication by the inverse of square root of $\mathbf{D}$,

$$\tilde{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{1/2} \quad \text{and} \quad \tilde{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{1/2} \tag{2}$$

These pre-emposed multiplications by $\mathbf{D}^{-1/2}$ result in representations in which weights are expressed relative to the degrees of individual nodes. Both $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{L}}$ will be symmetric if the graph is symmetric. Further observed, given these definitions $\tilde{\mathbf{L}}$ can also be written as,

$$\tilde{\mathbf{L}} = \mathbf{I} - \tilde{\mathbf{A}} \tag{3}$$

Graph shift operator $\mathbf{S}$ is a stand in operator for any of the matrix representations of the graph. We can set it $\mathbf{S} = \mathbf{A}$, $\mathbf{S} = \mathbf{L}$, $\mathbf{S} = \tilde{\mathbf{A}}$ or $\mathbf{S} = \tilde{\mathbf{L}}$. The specific choice of $\mathbf{S}$ matters in practice, but most of results and analysis hold for any choice of $\mathbf{S}$.

III. Methodology

The path towards scalable machine learning on graphs begins from the generalization of the convolutional operator in CNNs to signals supported on graphs. Once the convolutional operator is generalized to graphs, we can easily generate graph filter banks, combine these graph filter banks with pointwise non-linearities, and then stack them into layers to create GNNs. Graph convolutions follow a neighborhood aggregation strategy collecting the node features, $\mathbf{X} \in \mathbb{R}^{|V| \times F}$ (where $|V|$ denotes number of nodes, and F denotes the size of a node feature vector) within each node's K-hop neighbors to learn a representation vector of a node, h_v, or the entire graph, h_G.

$$h_G = \sum_{k=0}^{K-1} \sigma(\mathbf{S}^k \mathbf{X} \mathbf{W}_k), \tag{4}$$

where σ is the pointwise non-linearity, $\mathbf{S} \in \mathbb{R}^{|V| x |V|}$ is the graph shift operator, and $\mathbf{W}_k \in \mathbb{R}^{F \times G}$ is the graph filter, where G denotes the number of filters applied to each node feature vector. Multiplication by $\mathbf{W}$ gives the linear combination of features at each node. h_G iteratively updated K times to capture structural information from K-hop neighbors. We compare the discriminative power of various different GNN implementations for classifying EEG, and benchmark their performance against the state-of-the-art CNN models.

*a) **GraphSAGE**:* GraphSAGE learns an aggregated neighborhood embedding, $h^k_{N(v)}$, via $\texttt{AGGREGATE}_k$ functions, which distill high dimensional information from a node's K-hop neighbors, h^{k-1}_u, $\forall u \in N(v)$, in which $N(v)$ is the set of node v's K-hop neighbors and u denotes the nodes uniformly sampled from $N(v)$ [1]. The aggregated neighborhood embedding is then concatenated with the current node representation, h^{k-1}_v, and modulated by a trainable weight matrix (by feeding through a fully connected layer), W_k, which learns to propagate different levels of information for different search depths, k, of a node, and incorporates information about the graph structure. The output is passed through a ReLU non-linearity, σ. Algorithm 1 summarizes the procedure for the GraphSAGE algorithm.

Algorithm 1 GraphSAGE forward propagation - learning graph embedding, h_G.

```
1: $h^0_v \leftarrow \mathbf{X}_v, \forall v \in V$  Initialize a representation vector for each node
2: for $k = 1 \ldots K$ do
3:     for $v \in V$ do
4:         $h^k_{N(v)} \leftarrow \texttt{AGGREGATE}_k(\{h^{k-1}_u,\ \forall u \in N(v)\})$
5:         $h^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \texttt{CONCAT}(h^{k-1}_v,\ h^k_{N(v)})\right)$
6:     $h^k_v \leftarrow h^k_v / \left\| h^k_v \right\|_2$
7: $h_G \leftarrow \texttt{READOUT}(\{h^K_v,\ \forall v \in V\})$
```

AGGREGATE and READOUT functions must be invariant to permutation of input node representations such as summation or graph level max/mean pooling. We apply an element-wise mean-pooling as AGGREGATE, where each K-hop neighbor's representation vector is fed through a fully connected layer with ReLU non-linearity, and then an element-wise mean-pooling operation is applied to each of the computed node representations. READOUT returns the sum of node representations after graph convolution at the final iteration.

$$\texttt{AGGREGATE}_k = \texttt{mean}(\{\sigma(\mathbf{W}_{\text{pool}} h^k_{u_i} + b),\ \forall u_i \in N(v)\}) \tag{5}$$

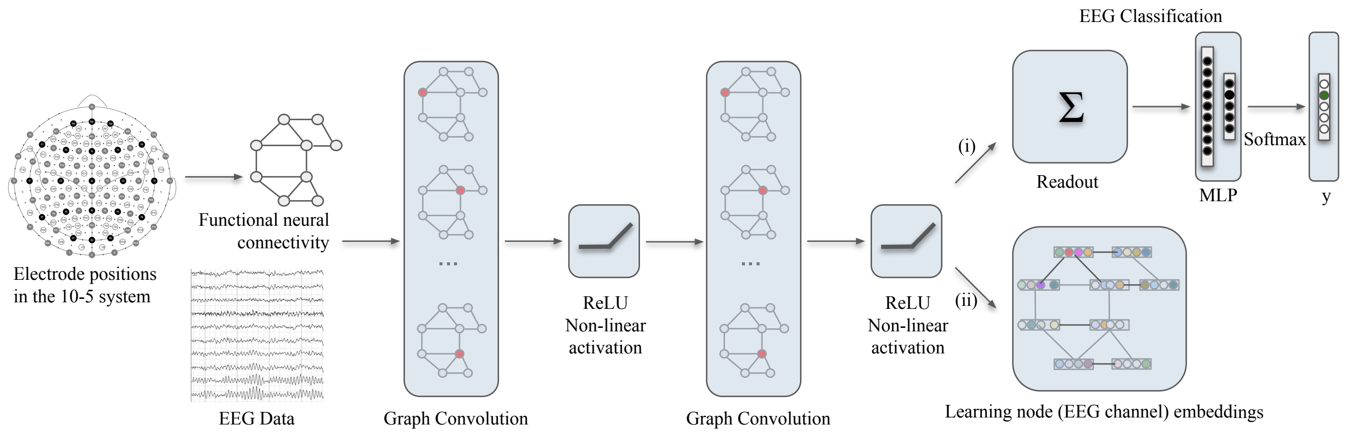

EEG Classification
Electrode positions in the 10-5 system
Functional neural connectivity
EEG Data
Graph Convolution
ReLU Non-linear activation
Graph Convolution
ReLU Non-linear activation
(i)
(ii)
Readout
MLP
Softmax
y
Learning node (EEG channel) embeddings

Fig. 2: An EEG-GNN with graph convolutional operators and ReLU non-linearity applied on EEG signals mapped onto the graph structure of the functional neural connectivity between EEG electrode sites. Graph convolution layer is based on neighborhood aggregation approach, and encapsulates each EEG channel's hidden state vector by aggregating information from its neighboring electrode sites. As the number of graph convolution layers increases, we can get the hidden state vector of further neighborhoods. For 2 graph convolution layers, each node (EEG electrode site) aggregates information from its nearest neighbor and 2-hop neighbors. Following (i), we aggregate the node representations from the final iteration of graph convolution via `READOUT` function to learn the representation vector of the entire graph. Then, the graph representation vector is classified by a multi-layer perceptron (MLP) using softmax activation at the output layer. Optionally, following (ii), we transform the output from the final iteration of graph convolution with ReLU non-linearity, and learn an embedding for each EEG channel. This is particularly useful for tasks that require EEG channel selection instead of task classification.

*b) **Graph Isomorphism Network (GIN):*** GIN is conceptually inspired by Weisfeiler-Lehman (WL) isomorphism test, which reduces graphs to their canonical forms to check whether they are topologically identical. Graph isomorphism requires there is a bijective function $f : G_1 \to G_2$ mapping a graph G_1 to another graph G_2, while preserving adjacencies. GIN proposes that if G_1 and G_2 are non-isomorphic, then their embeddings h_{G_1} and h_{G_2} cannot be identical [2], [3]. It also proves if `AGGREGATE` and `READOUT` functions are both permutation invariant and injective, then GNN is at most as powerful as WL isomorphism test at recognizing different graph structures [2].

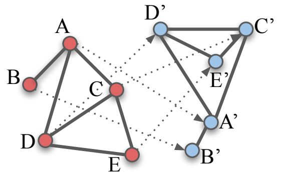


Fig. 3: An injective surjective mapping (bijection) between 2 topologically identical graphs. Adjacencies are preserved.

Nodes in different graph structures get the same embedding, when `mean` and `max` operators are used to aggregate information from neighboring nodes, as illustrated in Fig. 5, because these operators are non-injective. GNNs can only be as powerful as WL isomorphism test, if and only if `AGGREGATE` function maps an identical embedding for two nodes from two graphs, only when these two nodes have same subtree structures and same feature vectors in neighboring nodes [4]. According to universal approximation theorem, an MLP with one hidden layer can approximate any measurable functions. Hence, GIN approximates the solution to formulate an injective and permutation invariant aggregation operator via training an MLP with a single hidden layer as revealed in Equation 6, where λ is a learnable parameter. Graph representation concatenates the summations of node embeddings at the same iteration.

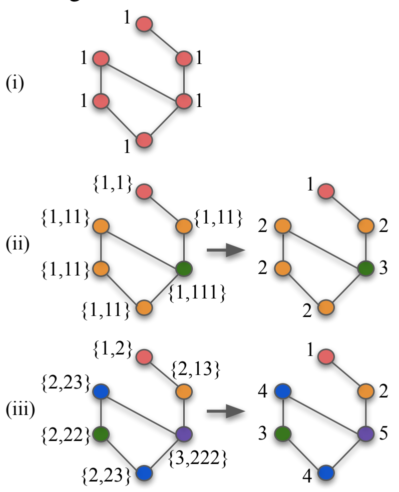


Fig. 4: **WL Subtree Kernel Method** (i) All vertices are assigned the initial color label 1, (ii) A signature string is constructed for each vertex by concatenating its own color label and its neighbors' color labels. Then, all the signature strings are compressed into their new integer color labels in the lexicographical order, (iii) This process is repeated until convergence to produce a canonical form of the graph. Vertices with the same color label have structurally similar roles in the graph.

Theorem 3.1 (Universal Approximation Theorem): Let σ be a non-linear activation function, and $x \in I_n$. Given the output of a one hidden layer MLP with weights w_j, α_j and

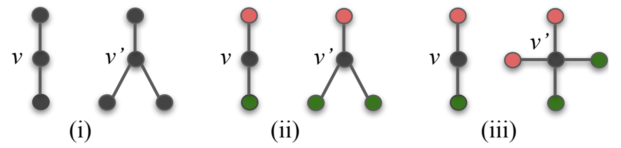


Fig. 5: (i) v and v' would have the same node embedding, if `AGGREGATE` function in (i) is `mean` or `max`, in (ii) is `max`, and in (iii) is `mean` or `max`.

biases b_j of the form,

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + b)$$

Then for any $f \in C(I_n)$, and $\varepsilon > 0$, there is a $G(x)$ such that

$$|G(x) - f(x)| \leq \varepsilon, \quad \forall x \in I_n.$$

$$h_v^k \leftarrow \mathtt{MLP}^k \left((1+\lambda^k) \cdot h_v^{k-1} + \sum_u h_u^{k-1}, \forall u \in N(v) \right)$$
$$h_G \leftarrow \mathtt{CONCAT} \left(\mathtt{READOUT}(\{h_v^k, \ \forall v \in V\}) \mid k = 0, 1 \dots K \right) \tag{6}$$

Since composition of injective functions is also injective, node representations can be evaluated with a single MLP,

$$h_v \leftarrow \mathtt{MLP} \left((1+\lambda^k) \cdot h_v^{k-1} + \sum_u h_u^{k-1}, \forall u \in N(v) \right) \tag{7}$$

Theorem 3.2: Let $f : G \to \mathbb{R}^d$ be a GNN, and G_1, G_2 represent 2 graphs. f is as powerful as the WL isomorphism test if `AGGREGATE` and `READOUT` functions are injective. Since `READOUT` projects distinct node representations in the domain to distinct graph representations in the codomain, we only need to prove aggregating neighborhood features must be injective [2]. Given node representations h_v^k derived by injective functions ψ and ϕ, and node labels l_v^k assigned by WL isomorphism test operator with injective function φ,

$$h_v^k \leftarrow \psi \left(h_v^{k-1}, \ \phi \left(\left\{ h_u^{k-1}, \forall u \in N(v) \right\} \right) \right)$$
$$l_v^k \leftarrow \ \varphi \left(l_v^{k-1}, \left\{ l_u^{k-1}, \forall u \in N(v) \right\} \right)$$

We can prove by induction, $\exists g \in \{g(a) = g(b) \implies a = b, \ \forall a, b \in \mathbb{R}\}$ that satisfies $h_v^k = g(l_v^k)$.
<u>Base case</u> for $k = 0$: $h_v^0 = g(l_v^0), \ \forall v \in G_1, G_2$, because initial node representations are identical for f and WL subtree kernel method.
<u>Inductive step</u> for $k = k-1$:

$$h_v^k \leftarrow \psi \left(g \left(l_v^{k-1} \right), \ \phi \left(\left\{ g \left(l_u^{k-1} \right), \forall u \in N(v) \right\} \right) \right)$$

Since the composition of injective functions is injective, this can be rewritten using another injective function ω,

$$h_v^k \leftarrow \omega \left(l_v^{k-1}, \ \left\{ l_u^{k-1}, \forall u \in N(v) \right\} \right)$$

which is identical to,

$$h_v^k \leftarrow \omega \circ \varphi^{-1} \varphi \left(l_v^{k-1}, \ \left\{ l_u^{k-1}, \forall u \in N(v) \right\} \right) = \omega \circ \varphi^{-1} \left(l_v^k \right)$$

Since the composition $\omega \circ \varphi^{-1}$ is also injective, f is as powerful as WL isomorphism test to decide the multisets $\left\{ l_v^k \right\}$ are unique for non-isomorphic G_1 and G_2 at each iteration k.

Pooling in CNNs downsamples the number of features in the input, which reduces the number of model parameters, and helps to avoid overfitting. SortPool, EdgePool, SagPool and Set2Set propose different pooling operations that play a similar role in GNNs.

*c) **SortPool:*** The key insight here is sorting vertices based on their structural importance established by WL isomorphism test in order to sequentially extract features from a graph in a consistent order [5]. After K iterations of graph convolution, SortPool layer gets an input of size $|V| \times K \cdot |h_v|$, where $|V|$ denotes the number of nodes, and $|h_v|$ denotes the size of a node's embedding vector, which is assumed to be identical for all the nodes. The output of SortPool layer has size $\rho \times K \cdot |h_v|$, where ρ is a heuristically selected parameter, and $\rho < |V|$. SortPool layer sorts the color labels in descending lexicographical order according to WL subtree kernel method, and chooses the first ρ number of nodes. Then, the sorted output is reshaped as a row vector of size $\rho \cdot K \cdot |h_v| \times 1$ and applied 1D convolutional layers with kernel size $K \cdot |h_v|$ followed by MaxPool. Final MaxPool output is fed through a dense layer with softmax activation.

*d) **EdgePool:*** Given a weighted graph, softmax function is applied to all edge weights to compute edge scores, denoted as s_{ij}. According to these scores, edges are iteratively contracted unless nodes have already been part of a contracted edge [6], [7]. Edges between contracted nodes are preserved. If there is an isolated node (a node with degree 0) after edge contraction, edges between the isolated node and contracted nodes are reconstructed. In the process of edge contraction, node features are combined, and then multiplied by the edge score, which allows the gradient to backpropagate through edge scores. Let $e = \{v_i, v_j\}$ is the edge contracting 2 nodes, then $h_{v_{ij}} = s_{ij}(h_{v_i} + h_{v_j})$.

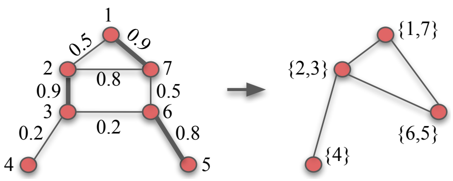


Fig. 6: After EdgePool operator, number of nodes is downsampled by a fixed ratio of 2.

*e) **SagPool:*** SagPool uses both node features and graph topology for pooling via a masking operator [8], as illustrated in Figure 7. First, the graph convolution layer computes self-attention scores, $Z \in \mathbb{R}^{|V| \times 1}$,

$$Z = \sigma(\mathbf{SXW}_{att}) \tag{8}$$

where $\mathbf{W}_{att} \in \mathbb{R}^{F \times 1}$ denotes the learnable pooling parameters. Top-rank function [9] selects the indices of most useful $\rho \times |V|$ number of nodes according to Z for a heuristic parameter $\rho \in (0, 1]$. The pooled output is the element-wise multiplication of selected node features and attention scores,

$$\text{idx} = \text{top-rank}\Big(Z, \lceil \rho |V| \rceil\Big), \quad Z_{\text{mask}} = Z_{\text{idx}} \\ X_{\text{out}} = X_{\text{idx}} \odot Z_{\text{mask}} \tag{9}$$

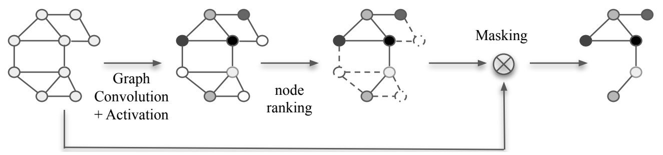


Fig. 7: Downsampling with SagPool operator.

*f) **Set2Set**:* It's an extension of Seq2Seq framework, which uses LSTM's with attention mechanism for graph level pooling. The pooled output, denoted as q_t^*, is the concatenation of the weighted sum of node embeddings, and the query vector, which reads from the memory of an LSTM [10].

Algorithm 2 Set2Set Pooling Operator

```
1: $q_0^* \leftarrow \mathbf{zeros}(1, \text{input dim.})$ Initialize a zero vector for the query vector $q$
2: $\text{hidden}_0 \leftarrow \mathbf{zeros}(1, \text{output dim.})$ Initialize a zero vector for the hidden state of LSTM cell
3: $\text{cell}_0 \leftarrow \mathbf{zeros}(1, \text{output dim.})$ Initialize a zero vector for the cell state of LSTM cell
4: $h_i$ Embedding of node $i$
5: for $i \in \{1, 2, \ldots |V|\}$ do
6:     $q_t, \text{hidden}_t, \text{cell}_t = \mathbf{LSTM}(q_{t-1}^*, \text{hidden}_{t-1}, \text{cell}_{t-1})$
7:     $\alpha_i = \frac{h_i^T q_i}{\sum_i h_j^T q_t}$ (attention params.)
8:     $r_t = \sum_i \alpha_i h_i$ (attention readout)
9:     $q_t^* = [q_t, r_t]$
10: $h_G = \mathbf{ReLU}(\mathbf{Dense}(q_t^*))$
```

IV. Datasets & Software

*a) **ErrP**:* The detection of error-related potentials (ErrP) to improve the accuracy of P300-based BCI speller [11].[1] The dataset was recorded from 16 healthy subjects participating in an offline P300 spelling task. Spelling task had a fast mode (each item was flashed 4 times), and a slow mode (each item was flashed 8 times). Each subject performed 340 trials. If there is an inconsistency between subject's intention and BCI system, elicited ErrP should be detected. EEG data were recorded at a downsampled rate of 200 Hz from 56 channels. A trial has 250 discretized time samples, and is associated with a binary class label: erroneous (inferred item is different from the intent of subject) or correct feedback.

*b) **RSVP**:* A BCI system to type based on rapid serial visual presentation (RSVP) paradigm [12].[2] The dataset was collected from 10 healthy subjects, and consists of 41,400 trials of 16 channel EEG data. A g.USBamp biosignal amplifier with active electrodes was used to record trials during RSVP keyboard operations. Each trial has 128 discretized time samples, and is associated with one of the 4 labels: emotion elicitation, resting-state, or motor imagery/execution task.

We use PyTorch Geometric v1.8.0 [13] to implement GNN variants. All models were trained with a minibatch size of 256 for 400 epochs on NVIDIA Tesla K80 12GB GPU, and optimized by Adam with an initial learning rate of 0.001, which decays into half every 50 epochs. EEG trials from all subjects were shuffled, first 80% of EEG trials was used for training, and the last 20% was used for validation. If there was an improvement in classification accuracy on the validation dataset, model checkpoints were saved.

V. Results & Discussion

*a) **Edge Formation**:* We present 6 different methods to connect electrode sites by edges e.g., i) each pair of electrode sites is connected (complete graph), ii) graph is complete, and allows self-loops, iii) electrode sites x and y are connected by an edge, if the distance between is among the k-th lowest distance that connects an electrode to x (k-NNG), iv) k-NNG allows self-loops, v) each pair of electrode site is connected, if the Euclidian distance between is lower than a heuristic threshold, vi) distance thresholding for the edge formation procedure allows self-loops. Empirical results in Table II and III indicate that connecting electrode sites with only their nearest neighbor provides a classification accuracy as good as a complete graph, while also saving training time, and model size. Therefore, we conclude that using fewer electrodes, and a sparse adjacency matrix does not cause a drop in classification performance, but has a positive impact on the system efficiency.

*b) **Parameter Regularization**:* Although GNNs have a high expressive power learning over graph structured data, there is no theoretical guarantee on their generalizability. Empirical results show that they are prone to overfitting for small datasets, and suffer from issues of poor convergence. In our experiments, we compare the performance of L1, L2, and elastic net regularization across different values of hyper-parameters to avoid overfitting. While the L1 penalty imposes sparsity by adding the absolute value of the magnitude of model weights modulated by the hyperparameter α to the loss objective, L2 penalty imposes feature selection, and shrinks weights towards 0, by adding the square of the magnitude of weights modulated by β. Elastic net performs concurrent regularization with both the L1 and L2 penalty. In Table IV, we finetune α and β, and report classification performance for neural connectivity between electrode sites constructed according to k-NNG ($k = 1$). Fast convergence, and sub-optimal optimization of model parameters were common issues (as also reported by other practitioners training GNNs in different domains [14]), despite our tests across smaller learning rates, parameter regularization, data augmentation, and node feature selection to reduce model bias, and variance. GNN models have one pitfall in that they tend to overfit quite heavily in small datasets.

*c) **Data Augmentation**:* Training data is augmented by adding white Gaussian noise (AWGN) to every channel of the original signal. The variance of this zero-mean Gaussian

[1]https://www.kaggle.com/c/inria-bci-challenge/
[2]http://hdl.handle.net/2047/D20294523

random variable affects the average noise power, and is specified to obtain SNR levels of 10, 5, and 2 dB. Desired SNR level is the difference between signal power and noise power in dB. This provides a three-fold increase in the number of EEG trials in the training datasets.

d) ***Temporal Compression****:* GNN models that classify EEG datasets defined over spatial-temporal graphs become more susceptible to overfitting as the number of features associated with each node increases due to a higher number of model parameters [4]. This turns out to be even more problematic while learning over EEG datasets, because number of samples per trial, which is equivalent to number of node features, is 250 for ErrP, and 128 for RSVP. We apply 1D convolutions with kernel size of 1×3 and stride 2 along the time axis of a trial, and then adopt batch normalization. This downsamples number of node features by 2, and also helps to capture temporal dependencies. We stack several layers of 1D convolutions, and reduce the number of node features to 32.

We benchmark the performance of EEG-GNN against the state-of-the-art CNN classifiers in Table I. Compared to a standard CNN classifier of similar model size, GNN models significantly improve the accuracy by 2.0% for ErrP, whereas the improvement for RSVP is 0.4%. Compared to AutoBayes classifiers [15], which detect the conditional relationship between data features, task labels, nuisance variation labels (subject IDs), and potential latent variables in DNN architectures to identify the best inference strategy, GNN models still have higher classification performance, while reducing model size more than 20x. Although GNN models don't take advantage from adversarial learning using variations in subject IDs, they perform 0.8% better for ErrP, and nearly same for RSVP. Identification of best Bayesian network trained in an adversarial setting helps with getting a non-dispersive distribution of the accuracies across different subjects and hence provides more robustness against subject variation; however GNN classifiers demonstrate an accuracy equivalent to the best Bayesian network selected by AutoBayes framework.

TABLE I: Task classification performance of GNN compared to a CNN classifier, AutoBayes models, and the ensemble of AutoBayes.

Method	ErrP		RSVP	
	Acc.	# Model Params.	Acc.	# Model Params.
EEG-GNN	76.73 ± 0.40	106,562	93.49 ± 0.10	83,138
Standard CNN	74.72 ± 0.31	127,335	93.07 ± 0.15	268,865
Best of AutoBayes	75.91 ± 0.44	3,407,390	93.42 ± 0.15	2,005,917

We further benchmark the performance of EEG-GNN against the standard CNN classifiers by exploring different network configurations in Fig. 8. Pareto front connects the model configurations with superior performance proceeding from low to high complexity regime. It is observed that EEG-GNN performs significantly higher in low complexity regime than the standard CNN classifier. However, the performance gap reduces in high complexity regime. These results highlight the problem that scaling up GNN models to exploit multi-hop neighborhoods does not necessarily improve the performance. Vanishing gradient problem is encountered while training deeper GNN models. Using skip connections between graph convolutional layers can mitigate the effects of vanishing gradients, and allow to capture the representations of higher order graphs.

VI. Conclusion & Future Work

In this paper, we presented several GNN models along with various regularization strategies to model the functional neural connectivity between EEG electrode sites, and demonstrated GNN models outperform CNN models of different size and inference strategies in classification tasks across ErrP and RSVP datasets. There are many interesting directions for prospective research. For instance, we are currently learning models over unweighted graphs, but we can also use the Pearson correlation coefficient between EEG channels to represent the edge weights. Another interesting approach is to entirely eliminate a hand-engineered design of adjacency matrix, and learn a graph shift operator matrix along with model weights during training, in order to better capture the data topology. This would require parameterizing the graph shift operator matrix, transferring it to graph convolutional operators, and then appropriately taking gradients in backpropagation.

References

[1] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[2] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[3] B.-H. Kim and J. C. Ye, "Understanding graph isomorphism network for rs-fmri functional connectivity analysis," *Frontiers in neuroscience*, vol. 14, p. 630, 2020.

[4] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, 2020.

[5] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[6] F. Diehl, T. Brunner, M. T. Le, and A. Knoll, "Towards graph pooling by edge contraction," in *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*, 2019.

[7] F. Diehl, "Edge contraction pooling for graph neural networks," *arXiv preprint arXiv:1905.10990*, 2019.

[8] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3734–3743.

[9] H. Gao and S. Ji, "Graph u-nets," in *international conference on machine learning*. PMLR, 2019, pp. 2083–2092.

[10] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," *arXiv preprint arXiv:1511.06391*, 2015.

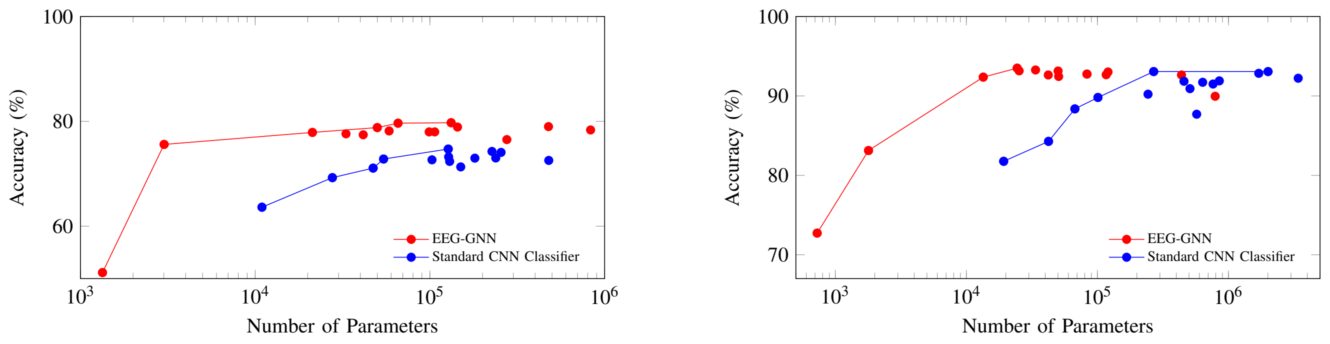
[11] P. Margaux, M. Emmanuel, D. Sébastien, B. Olivier, and M. Jérémie, "Objective and subjective evaluation of online error correction during p300-based spelling," *Advances in Human-Computer Interaction*, vol. 2012, 2012.

[12] U. Orhan, K. E. Hild, D. Erdogmus, B. Roark, B. Oken, and M. Fried-Oken, "Rsvp keyboard: An eeg based typing interface," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 645–648.

[13] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[14] R. Mercado, T. Rastemo, E. Lindelöf, G. Klambauer, O. Engkvist, H. Chen, and E. J. Bjerrum, "Graph networks for molecular design," *Machine Learning: Science and Technology*, vol. 2, no. 2, p. 025023, 2021.

[15] A. Demir, T. Koike-Akino, Y. Wang, and D. Erdogmus, "Autobayes: Automated bayesian graph exploration for nuisance-robust inference," *IEEE Access*, vol. 9, pp. 39 955–39 972, 2021.



(a) ErrP Dataset

(b) RSVP Dataset

Fig. 8: Accuracy vs. Space Complexity

TABLE II: Performance of datasets: Edge index matrix construction using a complete graph (all), a complete graph containing self-loops (all with self-loops), computing graph edges to the nearest k neighbors (k-NNG), and k-NNG containing self-loops (k-NNG w. self-loops).

Dataset	Model	All	All w. Self-Loops	k-NNG			k-NNG w. Self-Loops		
				k=1	k=2	k=4	k=1	k=2	k=4
ErrP	GraphSage	74.44 ± 0.75	75.94 ± 1.42	74.04 ± 0.98	74.89 ± 1.88	75.29 ± 0.70	74.34 ± 0.62	74.47 ± 0.88	76.33 ± 0.69
	Set2Set	75.38 ± 0.54	74.62 ± 0.17	75.66 ± 0.82	74.37 ± 0.83	75.88 ± 1.18	75.38 ± 0.90	73.27 ± 0.50	74.53 ± 1.04
	SortPool	72.90 ± 0.61	74.83 ± 1.71	73.52 ± 0.53	74.99 ± 0.16	74.56 ± 0.39	75.23 ± 0.85	74.34 ± 0.70	75.08 ± 0.53
	EdgePool	73.03 ± 0.96	73.05 ± 0.72	73.98 ± 0.54	75.60 ± 0.72	74.19 ± 0.53	75.11 ± 1.17	74.56 ± 1.80	76.24 ± 1.28
	SagPool	74.71 ± 1.09	75.57 ± 0.80	73.52 ± 0.80	75.66 ± 1.74	74.96 ± 0.59	74.53 ± 0.54	75.78 ± 2.17	74.86 ± 1.32
	GIN0	75.48 ± 0.60	76.09 ± 1.15	75.26 ± 1.95	73.79 ± 0.56	75.14 ± 0.59	76.24 ± 0.85	74.99 ± 1.00	74.44 ± 0.62
RSVP	GraphSage	93.27 ± 0.05	93.25 ± 0.35	93.05 ± 0.11	93.47 ± 0.06	93.22 ± 0.08	93.09 ± 0.33	93.08 ± 0.20	93.23 ± 0.17
	Set2Set	93.33 ± 0.09	93.19 ± 0.22	92.94 ± 0.11	93.34 ± 0.17	93.24 ± 0.26	93.30 ± 0.24	93.32 ± 0.10	93.26 ± 0.07
	SortPool	93.24 ± 0.20	93.36 ± 0.16	93.39 ± 0.28	93.38 ± 0.27	93.29 ± 0.21	93.05 ± 0.34	93.31 ± 0.14	93.35 ± 0.24
	EdgePool	92.89 ± 0.04	93.02 ± 0.26	93.32 ± 0.06	93.47 ± 0.49	93.39 ± 0.23	93.31 ± 0.06	93.49 ± 0.17	93.43 ± 0.22
	SagPool	93.45 ± 0.19	93.34 ± 0.09	93.14 ± 0.23	92.99 ± 0.16	93.36 ± 0.02	93.24 ± 0.20	93.03 ± 0.07	93.07 ± 0.11
	GIN0	93.26 ± 0.07	93.23 ± 0.01	93.18 ± 0.10	93.07 ± 0.19	93.23 ± 0.11	93.14 ± 0.34	93.10 ± 0.18	93.22 ± 0.10

TABLE III: Performance of datasets: Edge index construction using a distance threshold for the formation of edges, and distance threshold containing self-loops.

Dataset	Model	Distance			Distance w. Self-Loops		
		d=0.3	d=0.4	d=0.5	d=0.3	d=0.4	d=0.5
ErrP	GraphSage	75.05 ± 0.45	74.86 ± 1.50	75.81 ± 1.99	75.78 ± 1.02	74.89 ± 0.50	74.89 ± 0.61
	Set2Set	74.68 ± 2.02	74.07 ± 0.71	75.02 ± 0.57	75.57 ± 0.38	76.15 ± 1.11	74.25 ± 1.20
	SortPool	75.94 ± 0.95	74.16 ± 0.26	74.34 ± 1.59	74.99 ± 1.62	73.79 ± 0.84	74.62 ± 0.45
	EdgePool	74.71 ± 0.75	74.83 ± 0.41	74.47 ± 1.53	75.84 ± 0.55	75.23 ± 0.74	74.56 ± 1.23
	SagPool	76.06 ± 0.74	74.40 ± 1.32	76.15 ± 0.53	73.88 ± 0.67	74.47 ± 1.20	74.80 ± 1.28
	GIN0	76.06 ± 0.91	74.65 ± 0.66	74.83 ± 0.85	76.03 ± 1.05	75.75 ± 0.51	75.97 ± 0.46

TABLE IV: Performance of datasets: Hyperparameter selection for L1, L2, and ElasticNet Regularization of GNN models.

Dataset	Model	L1 Regularization			L2 Regularization			ElasticNet Regularization
		$\alpha = 0.1$	$\alpha = 0.01$	$\alpha = 0.001$	$\beta = 0.2$	$\beta = 0.4$	$\beta = 0.8$	Best of $\alpha \& \beta$
ErrP	GraphSage	76.55 ± 0.87	76.30 ± 1.00	74.07 ± 0.23	75.54 ± 0.44	74.56 ± 1.20	74.93 ± 1.41	73.39 ± 0.22
	Set2Set	74.86 ± 0.20	74.65 ± 1.48	74.56 ± 1.65	74.68 ± 0.67	76.18 ± 0.19	74.22 ± 0.96	74.50 ± 0.34
	SortPool	75.14 ± 0.33	74.89 ± 1.78	74.80 ± 1.74	73.73 ± 1.36	75.26 ± 0.71	74.65 ± 0.74	73.94 ± 0.82
	EdgePool	73.64 ± 0.17	74.96 ± 1.02	76.15 ± 1.06	74.99 ± 0.46	74.96 ± 1.05	75.23 ± 0.23	73.76 ± 0.67
	SagPool	75.60 ± 0.40	76.58 ± 0.54	75.11 ± 0.30	74.13 ± 0.23	74.19 ± 1.13	75.23 ± 1.04	73.58 ± 0.36
	GIN0	75.60 ± 0.42	75.02 ± 0.55	75.17 ± 1.01	74.40 ± 1.35	76.73 ± 0.40	74.56 ± 1.06	74.04 ± 0.85
RSVP	GraphSage	92.51 ± 0.19	93.49 ± 0.10	93.07 ± 0.32	92.64 ± 0.28	92.60 ± 0.11	92.76 ± 0.13	92.89 ± 0.12
	Set2Set	93.03 ± 0.17	93.12 ± 0.13	93.22 ± 0.27	92.93 ± 0.08	92.97 ± 0.11	93.03 ± 0.30	91.47 ± 0.20
	SortPool	93.14 ± 0.12	93.11 ± 0.20	92.91 ± 0.13	92.71 ± 0.31	93.10 ± 0.13	92.74 ± 0.08	92.90 ± 0.12
	EdgePool	93.49 ± 0.10	93.29 ± 0.10	93.12 ± 0.04	93.12 ± 0.01	93.13 ± 0.19	92.74 ± 0.19	92.57 ± 0.16
	SagPool	93.09 ± 0.18	93.18 ± 0.21	93.38 ± 0.30	92.74 ± 0.09	93.08 ± 0.17	92.91 ± 0.34	92.87 ± 0.23
	GIN0	92.87 ± 0.10	93.26 ± 0.12	93.13 ± 0.21	92.93 ± 0.06	93.07 ± 0.13	92.49 ± 0.13	92.85 ± 0.15