THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

# Learning a Latent Space for EEGs with Computational Graphs

by

Radhakrishnan Thiyagarajan

A thesis submitted in partial fulfillment
of the requirements for the degree of
*Master of Engineering*

April 4, 2018

Professor Sam Keene, Advisor

THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Adviser and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

---

Professor Richard Stock - Date
Dean, School of Engineering

---

Professor Sam Keene - Date
Candidate's Thesis Adviser

# Acknowledgements

Thanks to my advisor, Dr. Sam Keene, for the guidance, the encouragement and the friendship that he has provided me with throughout my time at The Cooper Union.

Thanks to Chris Curro for teaching me deep learning, helping me improve my LaTeXing skills, and coauthoring my first paper.

Thanks to my project partner, Zhengqi Xi, who worked with me last year and helped me get this project off the group.

Thanks to the electrical engineering faculty and The Cooper Union for educating me in electrical engineering.

Thanks to my mentors, Shan Raju and Nirmala Chetty, for helping and advising me in multiple ways during the past four years.

Thanks to my friends, Abhinav Jain, Yingfu Ma, Tushar Nichakwade, Miles Barber and Garo Bedonian, for standing by me at all times.

Thanks to my parents, Thiyagarajan and Latha, and my younger brother, Abhiramakrishnan, for bearing with my temperament and encouraging me with their best wishes.

# Abstract

Despite the recent advances in data organization and structuring, electronic medical records (EMRs) can often contain unstructured raw data, temporally constrained measurements, multichannel signal data and image data, all of which are difficult to compare and contrast in large quantities due to their sizes and variation. We present a proof of concept system that can alleviate this problem by mapping raw data to a compressed 64-dimensional space where the Euclidean distance between data points can be used as a measure of similarity. Using EEGs as a case study, we optimize a deep neural network mapping from the spectrogram of EEG data to a latent space by using the triplet loss function. To verify that this clustering method learns a meaningful representation of the data, we apply a six-class k-NN classifier to the output, a binary (seizure-like and noise-like signal) k-NN classifier to the output and visualize the output latent space using the t-SNE dimensionality reduction technique. We achieved a 60.4% six-class classification accuracy, a 90.1% binary classification accuracy on the TUH EEG Cohorts dataset and observed distinct clusters in a reduced dimension latent space found using the t-SNE algorithm.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations & Symbols

| | |
|---|---|
| ANN | Artificial Neural Network |
| CNN | Convolutional Neural Network |
| DCNN | Deep Convolutional Neural Networks |
| EDF+ | European Data Format |
| EEG | Electroencephalography |
| EKG | Electrocardiogram |
| EMR | Electronic Medical Records |
| FC | Fully Connected Neural Network |
| LSTM | Long Short-Term Memory |
| MEG | Magnetoencephalography |
| MRI | Magnetic Resonance Imaging |
| RNN | Recurrent Neural Networks |
| SGD | Stochastic Gradient Descent |
| STFT | Short Term Fourier Transform |
| t-SNE | t-Distributed Stochastic Neighbor Embedding |

# 1. Introduction

The healthcare industry commonly stores diverse instrumentation signals such as EEGs, EKGs, MEGs, X-Rays, MRIs, and CAT scans in a variety of digital formats commonly referred to as Electronic Medical Records (EMRs). These records can also contain natural language notes from medical professionals. It is difficult to perform complex information retrieval on these records. Rich information retrieval may open up the ability to compare and contrast patient records en-masse leading to new understandings of disease pathologies. For example, while the healthcare industry possesses a large amount of data on Alzheimer's Disease, a common chronic neurodegenerative disorder, medical professionals are unable to find the underlying cause of this disease and why it worsens over time. If such data can be transformed into an accessible and patient invariant format such that different patients with similar cases can be found easily, medical professionals may be able to pinpoint the cause of the disease and discover better treatments. Towards this goal, Picone et al. [1] have demonstrated a system that can automatically discover, time-align and annotate EEG events in order to perform cohort retrieval, the task of efficiently finding a group of observations that share defining characteristics given an example observation. The signal event detection and classification work in Picone et al. [1] uses Hidden Markov Models, which have been shown to work well with sequential data. Although this model achieved 91.4% sensitivity and 8.5% specificity on the signal classification task, it is not possible to infer similarity of samples from the output of a classifier. Since similarity is a key factor in cohort retrieval, it is important to incorporate it into any cohort retrieval scheme.

In contrast to the work done by Picone et al. [1], we optimize a deep neural network using a triplet loss function that results in a reduced dimensionality latent space which minimizes the distance between similar signals and maximizes the distance between dissimilar signals. In doing so, we expect clusters of signals to form in the latent space characterized by features that have a meaningful interpretation of the original signals. At inference time, we can use the network to map new EEG signals onto this latent space for querying. New samples could be presented to this system and mapped into the latent space. After this mapping, clustering or other distance-based methods could be employed to find similar EEG records as part of a cohort retrieval system. It is hoped that this system can be used to discover significant relations between diseases in medicine.

We organize the rest of this thesis in the following way. In chapter 2, we discuss background information on machine learning needed to understand our work. In chapter 3, we review the literature relevant to deep metric learning and deep feature embedding techniques. In chapter 4, we describe our data and how we organized it for ease of access. In chapter 5, we describe our final models, their quantitative and qualitative characteristics, our experimental results and our analysis of those results.

# 2. Background

## 2.1 Machine Learning

Machine learning is the field of computer science that allows computers to acquire knowledge from data and make some sort of prediction or estimation without explicitly programming said knowledge [2]. Russell and Norvig [3] say that a machine learning algorithm is designed for a particular task or problem, and is said to be learning if it improves its performance at that task based on a metric. Machine learning is often used for tasks that require pattern recognition, especially in large amounts of data. Examples of such tasks include handwriting recognition, cybersecurity breach detection, medical disease diagnosis and autonomous cars control systems [4–7]. Due to the variety of tasks that could fit this broad definition, there are also a variety of approaches that could be used to solve each of those particular tasks.

### 2.1.1 Supervised Learning

Supervised learning is a task that tries to learn a model that maps from a domain of inputs to a range of outputs based on data on which the task has already been accomplished. Mathematically, if $\mathcal{S} = \{(\mathbf{X_i}, \mathbf{y_i}) \mid i = 1...N\} \subset \mathcal{P}$, then the machine learning task may be to find a function ,$\hat{\mathbf{y}} = f(\mathbf{X})$, an estimate of $\mathbf{y}$ for any $(\mathbf{X}, \mathbf{y}) \in \mathcal{P}$ where $\mathbf{X}$ is an input, $\mathbf{y}$ is an output (also called a label or a target), $\hat{\mathbf{y}}$ is an estimate of the output from the machine learning model, $\mathcal{S}$ is a set of $N$ examples of input-output pairs called training set, and $\mathcal{P}$ is the population of input-output pairs. For

example, in the case of the hand-written digits recognition using the data provided by LeCun [4], the input $\mathbf{X}$ may be the image of the digit, $\mathbf{y}$ will be the true label of the digit provided by the dataset, and $\hat{\mathbf{y}}$ is the discrete value that the algorithm infers restricted to the range of $\mathbf{y}$. This particular process is called classification since we are restricted to a discrete set of values that are predefined. In order to learn a mapping from $\mathbf{X}$ to $\mathbf{y}$, a loss function, $J(\mathbf{y}, \hat{\mathbf{y}})$, can be used to quantify how well our model performs on our sample set, $\mathcal{S}$. In supervised learning, $J(\mathbf{y}, \hat{\mathbf{y}}) \to 0$ as $\hat{\mathbf{y}} \to \mathbf{y}$. If the loss function's value is large, the model is doing poorly; if the loss function's value is small, it generally means that the algorithm is doing well on $\mathcal{S}$. The loss function's value can be fed back to the algorithm to iteratively change the algorithm until the loss function's value reaches convergence or until it is stopped by the developer before complete convergence. Iteratively changing the model while keeping track of this loss function will allow for the model's input-output mapping to improve over time.

### 2.1.2 Unsupervised Learning

The task of unsupervised learning is to discover some hidden patterns within the data without any prior knowledge or labels of any sort.

One common task that is often seen in unsupervised learning is cluster analysis. Cluster analysis or clustering is the machine learning task of grouping a set of observations or objects in a way such that the portion of data in the same group (a cluster) is more similar than the portion that is not in the same group. Whereas classification tries to find a decision boundary between predefined classes, clustering tries to find decision boundaries in the sample space provided without knowing how many distinctive clusters there are in the dataset. Mathematically, if $\mathcal{S} = \{\mathbf{X_i} \mid i = 1...N\} \subset \mathcal{P}$, then the machine learning task might be to find a function, $\hat{\mathbf{y}} = f(\mathbf{X})$, such that

$\hat{\mathbf{y}} \in \{\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_k}\}$ where $k$ is the number of clusters that the algorithm finds. Notice how there is no dependency on a label as opposed to the classification task presented in section 2.1.1. Clustering algorithms, such as the k-means algorithm developed by MacQueen et al. [8], may need a number, $k$, provided by a developer suggesting that there are $k$ clusters. However, the algorithm is not provided any knowledge on which cluster each data point in the dataset is from, which is why it is unsupervised. The clusters resulting from these types of algorithms are a matter of interest in many applications and can lead to natural ways of classifying things once the general trend is found in the input space.

Another common task called dimensionality reduction attempts to use a set of observations with $M$ attributes and decreasing it to $K$ attributes where $M > K$, such that the characteristics of the original observation are still represented in the reduced vector space. Mathematically, if $\mathcal{S} = \{(\mathbf{X_i} \in \mathbb{R}^{M\times 1}) \mid i = 1...N\} \subset \mathcal{P}$, then the machine learning task might be to find a function $\hat{\mathbf{y}} = f(\mathbf{X})$ where $\hat{\mathbf{y}} \in \mathbb{R}^{K\times 1}$ such that both $\mathbf{X}$ and $\hat{\mathbf{y}}$ accurately represent the original observation. A loss function, $J(\mathbf{X}, \hat{\mathbf{y}})$, may be defined to ensure that both $\mathbf{X}$ and $\hat{\mathbf{y}}$ represent the same observation. An example of dimensionality reduction is provided below. Figure 2.1 is a high-quality image of a kitten and you can clearly see that its a kitten in the image. We can down-sample this image to $64 \times 64$ pixels and can still see the kitten even though the image is distorted in figure 2.2. Despite using $3 \times 2^{12}$ pixels for the compressed image as opposed to $3 \times 2^{18}$ pixels for the original image, we are still able to see what the image represents. Hence, down-sampling is a crude example of dimensionality reduction.

Often, dimensionality reduction techniques are used to learn latent variables, which are variables that are not directly observed. For example, in a picture of a kitten, a latent variable might be the cuteness of a kitten. Cuteness is not something that can be directly observed. A kitten can be observed to have a certain length

**Figure 2.1:** Original $512 \times 512$ pixels photograph of a kitten [9]

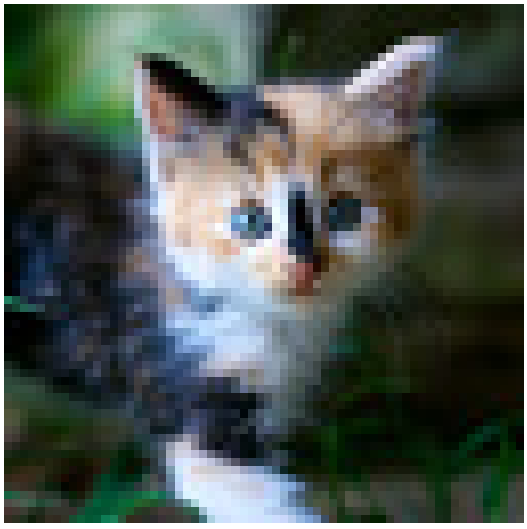

**Figure 2.2:** Down-sampled $64 \times 64$ pixels photograph of a kitten

of fur, a certain color of fur, eye diameter to head ratio, etc. but it is difficult to quantify a kitten's cuteness numerically. Whereas humans are able to claim a kitten is cute and can have a general consensus on it after looking at an image of a kitten, a machine might be able to learn what cuteness is by reducing the dimensionality of data and understanding it at a lower dimensional level but at a higher level of abstraction. These latent variables, along with other variables, can be used as input features to another machine learning algorithm which could be used learn more about the data. Hence, dimensionality reduction leads to processes known as feature extraction and feature engineering, the process of finding important variables derived directly or indirectly from raw input. Generally, the set of features that arise from feature extraction and engineering can form what is called a latent space (also called feature embedding, embedding space, latent space or space). The latent space is able to represent data that may have originally been incomparable by a machine as comparable points in this latent space.

### 2.1.3 Parametric Modeling and Optimization

Although the different types of learning methods have been described, we still need a way to train these models. One way to do this is to restrict our function $f$ representing the machine learning algorithm to a function that is parametric and differentiable. By changing the parameters of the function, we hope to make it better in the task that we designed it for. Therefore, we augment our original functional form of our machine learning model, $\hat{\mathbf{y}} = f(\mathbf{X})$, and refer to it as $\hat{\mathbf{y}} = f(\mathbf{X} \mid \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is a vector of parameters of the given function and $f$ is a differentiable function with respect to $\boldsymbol{\theta}$. Hence, our goal would be to find a $\hat{\boldsymbol{\theta}}$ that minimizes the loss function, $J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta}))$. The best way to approach this problem would be to calculate the gradient with respect to $\boldsymbol{\theta}$, set it equal to zero and solve for a $\hat{\boldsymbol{\theta}}$ like so

$$\nabla_{\boldsymbol{\theta}} J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta})) = 0. \tag{2.1}$$

However, in many cases, it is either not possible to find a closed form solution or the way to do so becomes intractable, especially when the training set is large. Often, the gradient calculation with respect to the model's parameters are estimated and are not exact. Hence, instead of attempting to find an analytic solution, we must use gradient descent, a numerical optimization algorithm, by finding the direction of steepest descent and moving the parameters in that direction in the following way

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta \, \nabla_{\boldsymbol{\theta}} J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta})) \tag{2.2}$$

where $\eta$ is the learning rate hyperparameter that controls the size of the step towards the direction of steepest descent as described by Bottou [10]. If this parameter is too large, the step towards the optimal $\boldsymbol{\theta}$ will overshoot, miss the optimal $\boldsymbol{\theta}$ and never

converge. If this parameter is too small, the step towards the optimal $\boldsymbol{\theta}$ will be more precise, but the time to find $\hat{\boldsymbol{\theta}}$ will be large. One way to solve this problem is to change the learning rate over time so that in the beginning large steps are taken and when the amount of change in between $\hat{\boldsymbol{\theta}}_t$ and $\hat{\boldsymbol{\theta}}_{t+1}$ is small, the learning rate is decreased as proposed by Xu [11] in order to get as close to the optimum solution as possible. Gradient descent is the name that is usually given to the algorithm which calculates the gradient based on the entire training set and updates the parameters using that gradient value. Another form of gradient descent, stochastic gradient descent (SGD) developed by Robbins and Monro [12] and Kiefer and Wolfowitz [13] calculates the gradient either for a single sample or a small batch of samples and takes small steps towards the optimal solution. SGD is computationally faster. Since large datasets cannot be held in RAM, it is faster to select mini-batches of data from the training set and calculate the gradient on said mini-batches. SGD is able to make more updates over a time period than gradient descent and results in a model as good as or better than that of gradient descent. Generally, this process is repeated multiple times on the training data and each repetition is called an *epoch*.

Unfortunately, as the number of parameters that the equation needs to train increases, the classic SGD algorithm becomes insufficient. The algorithm finds parameters which are local minimums in the function $J$ as opposed to global minimums. An example of a global minimum and local minimum is shown in figure 2.3. Consequently, the model that results is not as optimal as it can be. More sophisticated algorithms have been developed over time. One of these algorithms is called Adam and was developed by Kingma and Ba [14].

Adam, the short form for Adaptive Moment Estimation, is a first-order optimization algorithm that uses running averages of the gradient as well as the bias-corrected estimates to update the first two moments of the gradient to update the parameters

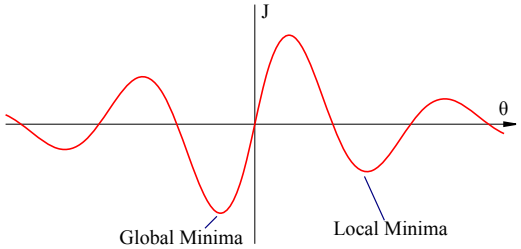


**Figure 2.3:** Illustration of global minimum vs. local minimum

$\boldsymbol{\theta}$. Moments are the measures of how the gradient has changed over the last couple of iterations, The moment allows the optimization algorithm to act like a ball on a hill trying to roll to the lowest height possible. The ball will continue to roll even if a local minimum is achieved and continue to try to find a global minimum. In the case that the minimum that it finds *is* the global minimum, the ball will continue to oscillate around that point until it loses momentum. Adam works in an analogous way. Therefore, we use Adam in our experiments as opposed to classic SGD for optimizing our models.

### 2.1.4 Bias-Variance Trade-off

One of the main goals of machine learning is to learn a general trend from a limited amount of sample data. In other words, we expect training accuracy, i.e. the predictive accuracy of the model on the training set, and the testing accuracy, i.e. the predictive accuracy of the model on unseen data from the real world, to be as close as possible. Unfortunately, this is not always the case. Figure 2.4 demonstrates this concept by fitting a first and a twentieth-degree polynomial to the same data originating from a sinusoid using the least squares loss function given by equation 2.3.

$$J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta})) = \sum_{i=0}^{N} (\mathbf{y}_i - f(\mathbf{X} \mid \boldsymbol{\theta}_i))^2 \tag{2.3}$$

It is evident from figure 2.4 that the twentieth-degree polynomial tries to go through as many points as possible and ends up over-fitting the data. On the other hand, the first-degree polynomial tries to best fit the data but is unable to do so due to the lack of complexity and, therefore, under-fits the original data. Neither of these graphs represents the true nature of the sinusoid. Therefore, we are faced with a trade-off between two sources of errors: bias and variance.

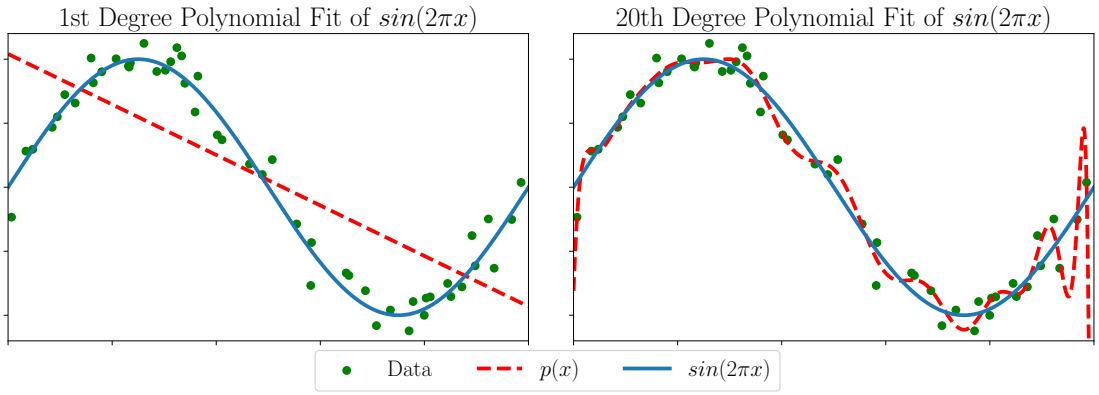


**Figure 2.4:** Example of an under-fit (left) and an over-fit (right) model

Bias is the error that arises from making overly simplistic assumptions about the underlying trend in the data and results from using too few parameters in the model we are training. Variance is the error that arises from making overly complicated assumptions about the underlying trend in the data and results from using too many parameters in the model we are training. These two errors comprise the bias-variance trade-off. In order to minimize this error, we need to make a compromise between these two sources of error and select a model that is neither too complex nor too simple.

There are two ways that we can approach this problem. Either we can start with a simplistic model and make it more complex or we can start with a complex model and make it simpler. The latter is easier and the general name given to the process

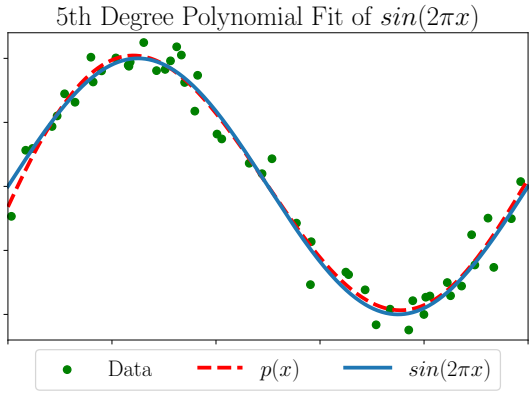


**Figure 2.5:** Example of a well-fit model

of making a relatively complex model simpler is called regularization. An example of a well-fit polynomial fit after simplifying the twentieth-degree polynomial is shown in figure 2.5.

In order for regularization to work, we need to know whether the model is generalizing well or not. Hence, we split the original training set, $\mathcal{S}$ into two mutually exclusive sets $\mathcal{S}_t$, the training set the model uses to learn, and $\mathcal{S}_v$, the validation set which we can use to see if the model is generalizing well to data that it has not seen before. If the validation accuracy is significantly lower than the training accuracy, we can infer that the model is not generalizing well.

One way we can regularize the model is by adding a penalty term to the loss function, $J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta}))$, based on the values of $\hat{\boldsymbol{\theta}}$. The resulting loss function would then be

$$J_t(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta})) = J(\mathbf{y}, f(\mathbf{X} \mid \boldsymbol{\theta})) + \lambda \, P(\boldsymbol{\theta}) \tag{2.4}$$

where $\lambda$ is a hyperparameter that controls how much we want to penalize a complex model and $P$ is a function such that $P(\boldsymbol{\theta}) \to \infty$ as $\theta_i \to \infty$. The two equations shown below are examples of such penalties.

$$P(\boldsymbol{\theta}) = \sum_{k=1}^{N} |\theta_k| \tag{2.5}$$

$$P(\boldsymbol{\theta}) = \sum_{k=1}^{N} \theta_k^2 \tag{2.6}$$

Equation 2.5 is known as the $L_1$ penalty and promotes sparsity. This means that the penalty forces any parameter that does not contribute to the model to zero thereby reducing the total number of parameters involved in the model. $L_2$ penalty, shown in equation 2.6, on the other hand, minimizes the contribution of a parameter but it does not force it to zero. Therefore, the number of parameters tends to be high, but the overall complex nature of the model is reduced. It is important to tune the hyperparameter $\lambda$ based on validation results in order to accurately penalize complex models to find a trade-off between bias and variance.

## 2.2 Neural Networks & Deep Learning

Artificial Neural Networks (ANNs), also called neural networks, are a type of computational system that was originally inspired by biological neural networks found in organisms that have a nervous system. Neural networks are at the cutting edge of difficult machine learning tasks and have surpassed human-level performance in these tasks. Neural networks have been used to solve a variety of problems including those in computer vision, speech recognition, machine translation, video game bots and medical diagnostics [6, 15–18].

### 2.2.1 Motivation for Neural Networks

Neural networks were originally inspired by biological neurons. Biological neurons generally consist of a cell body, dendrites, synapses and an axon. A dendrite is a part of a neuron that receives signals from other cells, including neurons, which were transmitted through a synapse as a chemical signal. These received signals are then propagated towards the cell body as an electrical signal. Once the cell body receives this electrical signal, more reactions happen within the cell body. If a certain action potential is reached, the neuron that received the signal fires and propagates the received signal along its axon towards other neurons and cells. A figure of a biological neuron is shown below in figure 2.6. The biological nervous system is, in essence, a network of these cells interconnected with each other in various ways.

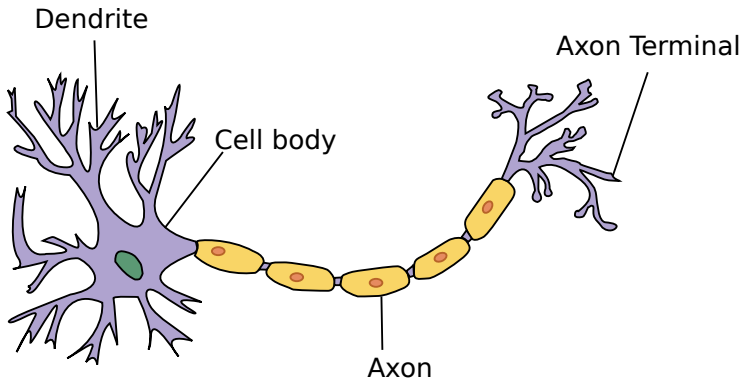


**Figure 2.6:** Illustration of a biological neuron[19]

An artificial neural network tries to mimic a nervous system through the simulation of neurons. A visual representation of a single artificial neuron or node is shown in figure 2.7. The inputs, $\{x_1...x_n\}$, are the set of numbers either given directly to the neuron or resulting from other neurons in the network. The neuron then multiplies each of these inputs by a corresponding weight $\{w_1...w_n\}$ and then sums these values together and adds a bias value $b$. The neuron then applies an activation function, $\sigma$,

and delivers its result through its axon to the next neuron or neurons in the network. Types of activation functions are discussed in section 2.2.6.

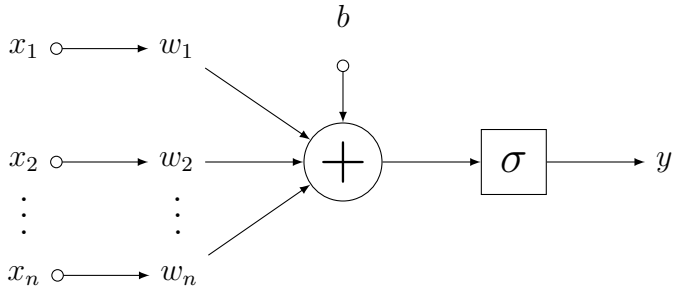


**Figure 2.7:** Block diagram of a single artificial neuron

As described in section 2.1.3, training a model is optimizing its learnable parameters, $\boldsymbol{\theta}$. The same concept applies to neural networks. The primary learnable parameters of the neural network are its weights. Depending on the architecture, i.e. the what parts the neural network is made up of, it may be possible to learn parameters presented in other parts of the neural network, such as the activation function. The process of learning any of these parameters for a neural network is called backpropagation.

Backpropagation begins with the forward propagation of a set of inputs. Forward propagation is when a neural network infers the output for a particular set of inputs. The network propagates the inputs, $\mathbf{X}$, through each appropriate neuron in a neural network sequentially depending on the architecture until the propagation reaches the output neuron, which will result in $\hat{\mathbf{y}}$. The output neurons' results can be compared to the true output $\mathbf{y}$ and, unless the network is already trained, a significant amount of error is expected between $\mathbf{y}$ and $\hat{\mathbf{y}}$, which can be quantified by a loss function, $J(\hat{\mathbf{y}}, \mathbf{y})$. A large loss indicates that the parameters need to be optimized which means that each of the unoptimized parameters contributes a small error to the total error in the output. Therefore, we need to trace back the steps in the neural network and find the

amount of error that each weight contributes for each input sample and adjust the respective weights. In other words, we are propagating the error backward towards every learnable parameter in the network, which is essentially the gradient of the loss function. Hence, equation 2.2 can be used to train the network.

### 2.2.2 Fully Connected Feed-Forward Networks

A fully connected neural network (FC network) is a neural network that propagates a signal layer by layer. The first layer in a neural network is commonly called the input layer since all the activations are inputs given to the network by the user. Depending on what the neural network is being used for, any of the next layers can be considered an output layer. In general, the last layer is considered the output layer. Any node not in either the input layer or the output layer are considered hidden nodes since these nodes' values do not matter to the input or the output; they are simply nodes that transfer information to the next layer or the next node. The neural network is considered to be a feed-forward network because the network does not propagate the information backward to previous nodes for any feedback. Figure 2.8 shows an illustration of a four-layer, fully connected, feed-forward neural network. The input layer contains four input nodes, both hidden layers contain five hidden nodes and the output layer contains three output nodes. The number of hidden layers and the number of nodes in each hidden layer are considered to be hyperparameters, parameters that can be controlled by the developer.

Fully connected feed-forward networks can be easily represented in mathematics as a series of matrix multiplications and activation functions. If there are $K$ layers in this type of neural network, there will be $K-1$ weight matrices and $K-1$ bias vectors. Each weight matrix will have an entry for a weight from a neuron in the

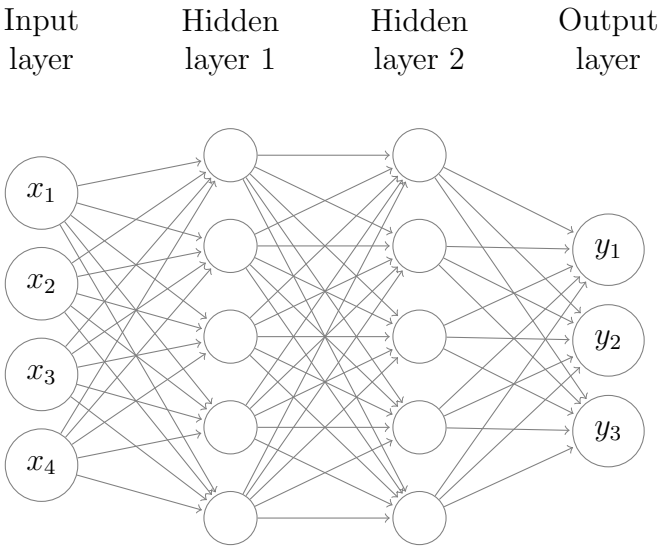


**Figure 2.8:** Graph diagram of a four-layer neural network

previous layer to a neuron in the current layer. In other words, if there are $N$ neurons in the previous layer and $M$ neurons in the current layer, there will be $N \times M$ entries in the weight matrix and $1 \times M$ entries in the bias vector. For example, the output of the neural network shown in figure 2.8 will be

$$\hat{\mathbf{y}} = \sigma(\sigma(\sigma(\mathbf{x} \times \mathbf{W_1} + \mathbf{b_1}) \times \mathbf{W_2} + \mathbf{b_2}) \times \mathbf{W_3} + \mathbf{b_3}) \tag{2.7}$$

where $\sigma$ is the activation function that acts on a matrix entry by entry, $\mathbf{x} \in \mathbb{R}^{1\times4}$ for an input with one sample, $\mathbf{W_1} \in \mathbb{R}^{4\times5}$, $\mathbf{b_1} \in \mathbb{R}^{1\times5}$, $\mathbf{W_2} \in \mathbb{R}^{5\times5}$, $\mathbf{b_2} \in \mathbb{R}^{1\times5}$, $\mathbf{W_3} \in \mathbb{R}^{5\times3}$, and $\mathbf{b_3} \in \mathbb{R}^{1\times3}$.

Backpropagation can be easily calculated for a simple FC network as demonstrated by Russell and Norvig [3].

### 2.2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of networks that have been successful in the field of image processing.

CNNs were also inspired by the biology of creatures that have vision. In 1965, Hubel and Wiesel [20] showed that cats contain neurons that respond to small regions of the field of vision. Assuming that the eyes are not moving, the concept of a receptive field, the particular region of vision that causes a single neuron to fire, was introduced. The concept of a receptive field is what led to what we know as a CNN.

CNNs attempt to learn shift-invariant characteristics in an input based on kernels, also called filters, which slide through the image and results in a filtered version of the original input image. The convolution computes the inner product of the kernel and the corresponding receptive field, saves the result to a new matrix known as the feature map, strides a particular length, and repeats the process until the entire input is convolved with the filter. Figure 2.9 shows an example of this process. The $3 \times 3$ kernel shown can be visualized as sliding over the input image represented as a $5 \times 5$ matrix. Note that a zero padding has been applied to the matrix so that the resulting feature map is the same size as the original image. However, it is also possible to forgo the padding and have the convolution result in a smaller sized feature map depending on the architecture of the neural network. The feature map is the result of the convolution and helps the network understand abstract patterns in the input, such as edges in the case of an image.

There are a few advantages in using CNNs over more traditional methods of image processing or FC networks. Suppose we are trying to classify the digits found in the MNIST dataset. Traditionally, a human is involved, hand-engineers filters that seem to work, and uses post-processing algorithms, which looks at the filtered

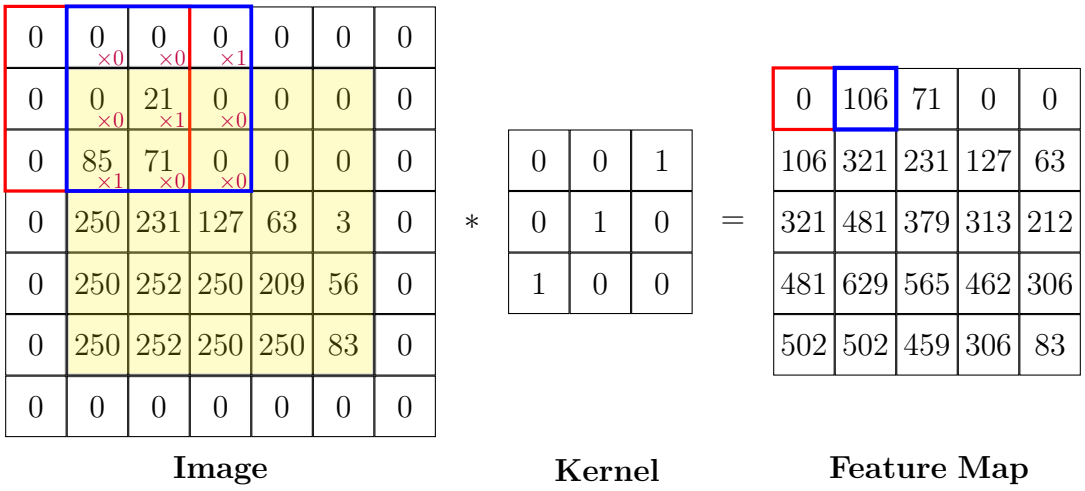


**Figure 2.9:** Illustration of a spatial convolution used in CNNs

image to finally classify it as a digit between zero to nine. However, the usage of neural networks, especially CNNs, has eliminated the need for hand engineering filters. Backpropagation has the ability to learn the weights just like it has the ability to learn the weights of a FC network. However, learning the weights of CNNs still take a long time due to the current hardware capabilities. The inference time of CNNs is also generally longer thant hat of traditional algorithms. Hence, if time is not of the essence, CNNs tend to be better than classical algorithms for image processing tasks.

It is also better to use CNNs over FC networks for a task where spatially invariant features may be involved. Assume again that we are trying to classify the digits found in the MNIST dataset. In a single digit, there are $28 \times 28$ pixels with values between 0 to 255. For a fully connected network, all of these pixels would be input nodes and therefore there would be $28 \times 28 = 784$ input neurons. Assuming that the first hidden layer has 250 neurons, the weight matrix will have $784 \times 250 = 196000$ weights that will have to be trained. This is an enormous amount of weights for a relatively small input size. However, if we use a convolutional layer with a $5 \times 5$ kernel on the same MNIST dataset, we would only have to train 25 weights which means that the overall

complexity of our network will go down and since we are learning a general trend, the average accuracy will go up.

Series of convolutions can be placed sequentially after one another and gives the network a chance of learning more complicated spatial features that affect the final output. Networks that use series of convolutions one after another are commonly called deep convolutional neural networks or DCNNs. This is where the idea of deep learning arises since we are trying to increase the depth, the number of layers in a neural network, to learn more meaningful representations of data.

### 2.2.4 Pooling

Pooling is a non-linear, sub-sampling operation in DCNNs which are used to reduce dimensions and the number of inputs to the subsequent layer. Pooling operations are non-parametric and they usually operate on the feature maps provided by convolutional layers before it. Two common types of pooling operations are used in neural networks: max-pooling and average pooling. Max-pooling [21] finds the maximum element in a receptive field while average pooling takes the average of elements in the receptive field. Pooling is used in situations where the absolute location of features do not matter as much as the relative location of features.

### 2.2.5 Dropout

Dropout is a special type of regularization used in neural networks. The technique was motivated by the fact that both biological and artificial neurons make decisions based on the decisions of its predecessors. In the case that one of the predecessors of a particular neuron fires incorrectly, the current neuron is also more likely to fire incorrectly. In order to prevent this phenomenon, Srivastava et al. [22] proposed this

method of regularization which randomly zeros out a neuron's output based on a Bernoulli trial probability of $p$ for the current iteration of training. As a result, when a future neuron fires, its dependency on the previous neurons effectively decreases and has a better chance of learning a meaningful representation of the dataset.

### 2.2.6 Activation Functions

Several common activation functions are used today in cutting edge deep learning work. Our paper uses two of these activation functions: sigmoids and ReLUs.

#### Sigmoid Activation

The sigmoid activation has a functional form as follows:

$$\sigma(x) = \frac{1}{1 + exp(-x)} \tag{2.8}$$

and the corresponding gradient is

$$\sigma'(x) = \frac{exp(x)}{(exp(x) + 1)^2}. \tag{2.9}$$

.

Brief analysis of the sigmoid's gradient shows that it suffers from the vanishing gradient problem, which refers to the function's gradient approaching zero as the input, $x$, increases or decreases. Because of this, sigmoid activations lead to extremely small gradient valules and tend to slow down learning. However, sigmoids are very useful for final layers, especially in classification problems, since they limit output values to less than one. They are also commonly used in probability since the cumulative distribution function (CDF) of various probability distributions are sigmoidal

in nature. As a result, the multi-dimensional analog of the sigmoid function, the softmax function, is used in neural network classification tasks to predict the probability that a given input is a particular class.

**ReLU**

Rectified Linear Units (ReLU) have demonstrated performance increases in both accuracy, generalization and training speed in neural networks as shown by Dahl et al. [23]. ReLU has a functional form of:

$$\rho(x) = max(x, 0) \tag{2.10}$$

and the corresponding gradient is

$$\rho'(x) = \begin{cases} 0 & x \leq 0 \\ & \\ 1 & x > 0 \end{cases}. \tag{2.11}$$

It is noticeable that the functional form of ReLU's gradient is a step function and only takes values of either 1 or 0. As a result, networks with ReLUs do not have to deal with vanishing gradient or the exploding gradient problems and leads to a faster learning network.

## 2.3 Basics About EEG Signals

The brain and the nervous system function on the basis of electrochemical reactions through biological neurons, as illustrated figure 2.6, to send various signals to cells all over the body. Electroencephalography or EEG is a non-invasive way of measuring that electrical activity resulting on the surface of the brain due to these

neurons. These signals can be used to diagnose seizures, brain tumors, head injuries, strokes, anesthesia overdose and many more ailments originating from the brain as described by Pruthi et al. [24].

According to Marghescu [25], electrodes that measure voltage with respect to ground are attached to certain locations on the scalp as described by the 10-20 system. An illustration of the 10-20 system's placement map is shown in figure 2.10. The letters and numbers on each of these electrodes can be used as an indication of location for each of the twenty-one channels in the EEG.

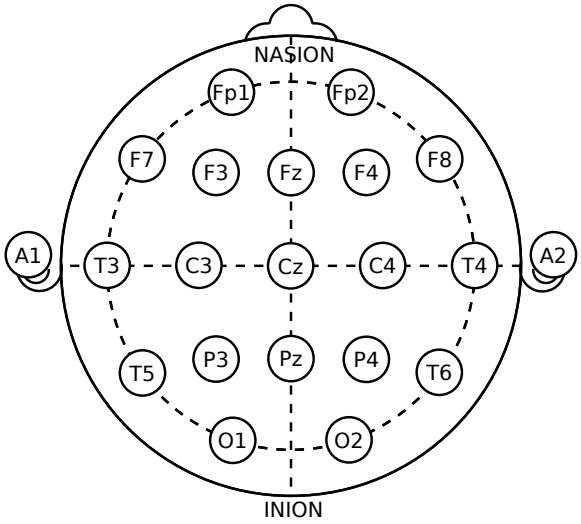


**Figure 2.10:** Illustration of the 10-20 system used to place EEG electrodes provided by Wikimedia-Commons [26]

Montages, voltage differences between certain probes, are used to extract information from these signals as opposed to the sole voltages sensed by the probes. Referential montages use the difference between a measuring electrode and a designated reference electrode. The reference electrode could be the ground which would mean the voltages sensed by the electrodes with respect to the ground can be used as the output of the EEG. In average reference montage, the outputs of all the electrodes are used as the reference voltage. Bipolar montages use the differences between two

adjacent electrodes, e.g. F3 and C3 dubbed "F3-C3", as the output of the EEG. In Laplacian montages, the output is the voltage difference between an electrode and the average of the neighboring electrodes.

A variety of different information can be extracted from these montages that can be used to classify the EEG activity. Frequency is one of these measurements. Rhythmic activity is considered to be constant in frequency, arhythmic activity is where no rhythms are present, dysrhythmic activity is a pattern that is rarely seen in healthy subjects. Frequency is also generally classified as delta, theta, alpha or beta waves. Delta waves have a frequency of 3Hz or below, theta waves have frequencies between 3.5Hz to 7.5Hz, alpha waves have frequencies between 7.5Hz and 13Hz, and beta waves have frequencies above 14Hz. Generally, only waves between 0Hz and 70Hz are considered since the rest of the signal is considered to be high-frequency noise in most cases and can be ignored.

The amplitude of these signals also tends to be useful. When a person is awake, beta waves usually dominate EEGs meaning that the average amplitude of the signals are small and the frequency is high. As a person starts to close their eyes, the average amplitude increases and the frequency starts to drop resulting in alpha waves. When a person starts to sleep, theta waves dominate, the average amplitude increases and the frequency decreases. Finally, in deep sleep, delta waves are observed in normal patients, the amplitude is generally large and the signal has very low frequencies. Therefore, we can logically deduce some things about the patient using amplitude and frequency together. Heuristically say that if a person's EEG contains high frequencies and high amplitudes, they may be experiencing a seizure.

However, such simplistic heuristics cannot always be used to extract useful information from patients. Therefore, a professional *or* an algorithm that considers the complexity of EEGs needs extract meaningful, diagnostic information.

# 3. Related Work

Deep metric learning is the task of using deep learning to learn a similarity metric using distance as a measure of similarity. Deep feature embedding learning is the task using deep learning to learn a set of features that aptly describes the original data and representing it in a vector space. Both of these tasks have been attempted many times before in a variety of different fields.

Schroff et al. [27] used a DCNN trained with a triplet loss function to create an embedding space for facial recognition and facial similarity search. This model was trained to minimize the distance between an anchor and a positive and maximize the distance between an anchor and a negative. Mathematically,

$$||f(x_i^a \mid \boldsymbol{\theta}) - f(x_i^p \mid \boldsymbol{\theta})||^2 + \alpha < ||f(x_i^a \mid \boldsymbol{\theta}) - f(x_i^n \mid \boldsymbol{\theta})||^2 \tag{3.1}$$

where $\hat{\mathbf{y}} = f(X \mid \boldsymbol{\theta}) \in \mathbb{R}^d$ represents the computational graph, $a$ is the anchor, $p$ is the positive which is the same class as the anchor, $n$ is the negative which is not the same class as the anchor and $\alpha$ is the margin parameter, a hyperparameter expressing the minimum distance between different clusters. Thus, the objective function becomes:

$$J = \sum_{i=1}^{N} \left[ \|f(x_i^a \mid \boldsymbol{\theta}) - f(x_i^p \mid \boldsymbol{\theta})\|^2 - \|f(x_i^a \mid \boldsymbol{\theta}) - f(x_i^n \mid \boldsymbol{\theta})\|^2 + \alpha \right]. \tag{3.2}$$

Schroff et al. [27] achieved 99.63% accuracy on the Labeled Faces in the Wild dataset and a 95.12% accuracy on the YouTube Faces DB dataset and it cut the error rate by 30% compared to the previous state-of-the-art published by Sun et al. [28].

Song et al. [29] provide a way of learning metrics through the use of what they describe as lifted structured feature embedding. Similar to Schroff et al. [27], an input is fed into a neural network to produce a feature embedding. However, this scheme considers both the local and global structure of the embedding space. As opposed to triplet approach, this method does not require partitioning data into tuples in any manner. Song et al. [29] find all the possible edges in a given mini-batch and describe whether they are similar or not using the Euclidean distance on the resulting embeddings and try to minimize a loss function based on those edges. They mathematically describe their loss function as the following:

$$\begin{aligned} \tilde{J}_{i,j} &= log\Big( \sum_{(i,k)\in\mathcal{N}} \exp\{\alpha - D_{i,k}\} + \sum_{(j,l)\in\mathcal{N}} \exp\{\alpha - D_{j,l}\} \Big) + D_{i,j} \\ J &= \frac{1}{2|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} \max\Big(0, \tilde{J}_{i,j}\Big)^2 \end{aligned} \tag{3.3}$$

where $D_{i,k} = ||f(\mathbf{X}_i) - f(\mathbf{X}_j)||^2$, $\alpha$ is the margin parameter, $\mathcal{P}$ is the set of positive pairs, $\mathcal{N}$ is the set of negative pairs, and $f$ is the network that produces the embeddings. This method achieved state of the art performance on standard datasets such as CUB200-2011, Cars196 and Stanford online products. However, this method represents a computational trade-off that may not necessary.

More ways of clustering raw data in the deep learning literature as those seen in Yang et al. [30], Weinberger and Saul [31], Wang et al. [32] and Rumelhart et al. [33]. However, little work has been done in trying to apply these methods to medical data to understand it better.

Choi et al. [34] proposed Med2Vec which both learned distributed representations for medical codes and visits from a large EHR dataset, and also allowed for meaningful inerpretations which were confirmed by clinicians using a two-layer pereptron. They use information such as demographics, diagnosis information and prescription infor-

mation to learn representations. Although the work done by Choi et al. [34] works towards building a latent space for EMRs, the model that they use is overly simplistic. Furthermore, it does not extract information directly from raw data. Hence, there is potential for loss of information.

Gøeg et al. [35] proposed a method for clustering models based on Systematized Nomenclature of Medicine - Clinical Terms (SNOMED CT) and used semantic similariy and aggregation techniques to hierarchically cluster EMRs. Similar to the work proposed by Choi et al. [34], their work relies on notes that were manually gathered by medical professionals and not the direct source of data itself.

Choi et al. [36] proposed a method for learning low-dimensional representations of a wide range of concepts in medicine using claims data, which is more widely available to the public than annotations by medical professionals. They define "medical relatedness" and "medical conceptual similarity" by using current standards in medicine as established by the NDF-RT and the hierarchical ICD9 groups from CCS. They qualitatively evaluate their system and show that the top 5 neighbors for each input, sans duplicates, are medically related. Although their system works well, it still suffers from the same pitfall as the ones shown above.

In fact, many more papers have attempted to cluster medical data and they have succeeded. However, they all seem to use only human annotations as input to their systems instead of both human annotations *and* raw data. It is evident that there is a motion towards finding representations of medical records and medical data, however, the ways that are currently utilized are insubstantial due to the fact that they are using the analysis of data provided by medical professionals. Hence, this paper tries to fill this void by attempting to cluster raw EEG data in order to improve current methods of clustering EMRs.

# 4. Data & Resources

## 4.1 Data

The data for this study was derived from the Temple University Hospital's EEG corpus which includes over 30,000 EEGs spanning the years from 2002 to the present as described and provided by Picone [37]. The original data consists of raw European Data Format (EDF+) files, a format commonly used for exchanging and storing multi-channel biological and physical signals, and the corresponding labels for each of these files in LBL files. Both EDF files and the LBL files were stored in session folders with a single patient's data and doctors' notes on that patient's EEGs. There are a total of 339 folders labeled from `session1` to `session339`. The label files are interpretable by Temple University's publicly available Python script [37], which transforms the label files into a readable format. Each channel is annotated as pertaining to one of six classes as described in table 4.1 with a granularity of one second. We assume that the data provided to us was time aligned correctly with the labels. For more details on the dataset see Obeid and Picone [38]

The EDF files contain raw signals with different channels from electrodesplaced in the standard 10-20 system and were decoded using Python's MNE package. A total of 22 montages were found in each label file. The power spectral density (PSD) plot of the signal was visualized using the `RawEDF.plot_psd()` function. The bandwidth of the signals was between 0 Hz to around 130 Hz. It was revealed that the signals contained power line noise at 60 Hz and 120 Hz as seen in the sample PSD plot in

**Table 4.1:** Set of classes for the TUH EEG Corpus. After consulting Harati et al. [39], it was determined that BCKG, ARTF and EYBL are noise-like signals, and the rest are all seizure-like signals, i.e. indications of common events that occur in seizures.

| Codename | Description |
|---|---|
| BCKG | Background noise |
| ARTF | Artifacts |
| EYBL | Eyeball movement |
| SPSW | Spikes and sharp waves |
| PLED | Periodic lateralized epileptiform discharges |
| GPED | Generalized periodic epileptiform discharges |

figure 4.1.

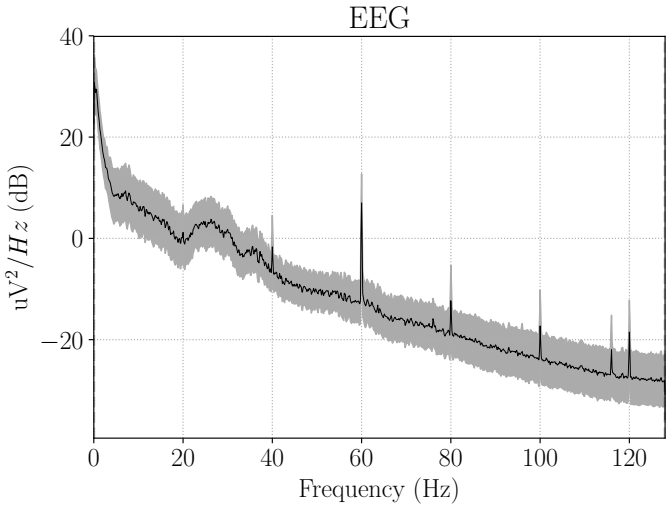


**Figure 4.1:** Power spectral density plot of raw signals using the MNE packagel

Hence, we apply notch filters at 60 Hz and 120 Hz to remove power line noise, and a bandpass filter with a 1 Hz to 70 Hz pass-band to remove any high-frequency noise as the bulk of the signal power was within this band. We apply the Short-Term Fourier Transform (STFT) provided by the MNE package with a window of

140 samples and a stride of two samples which results in the spectrogram represented as a $71 \times 125$ tensor for each one-second window of the signal as shown in figure 4.2.

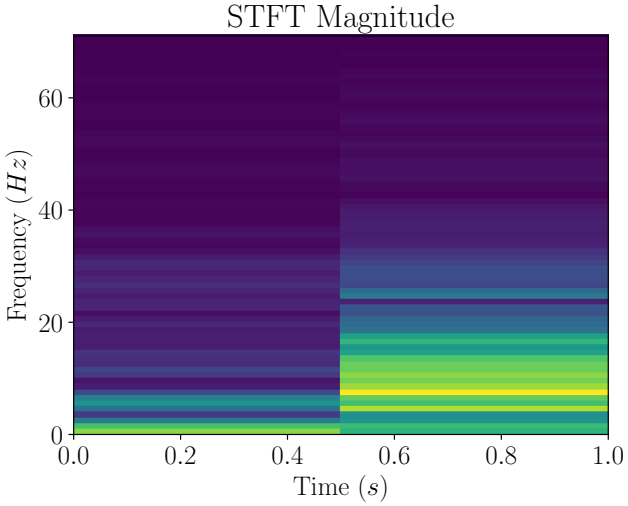


**Figure 4.2:** Spectrogram of a second of notch and bandpass filtered signal

Additionally, we globally normalize the signal power in order to standardize the input to the system that we designed and only use the real part of the spectrogram data. The raw time-domain data was not utilized in this experiment because the literature [24, 25, 40, 41] on EEGs indicated that the frequency domain is what contains data that is useful for our purpose.

We did not use the spatial information implicitly provided to us by the 10-20 system's spatial structure. This decision was made because the resulting tensor would have become a $22 \times 71 \times 125$ tensor of values and the amount of time taken to process this tensor would have been longer than the time taken to process a $71 \times 125$ tensor. Furthermore, even if it were computationally possible for us to process the larger tensor for each second of signal, there was no possible way to consistently label the tensor as each montage was labeled independently of the others. Hence, there was no possible way to consistently label all 22 montages with a single label. As a result, we only used a single channel's input as opposed to all 22 channels' inputs.

We also realized that the dataset is highly imbalanced. More than 80% of the data was labeled as noise-like signals. Since we were looking for anomalies in the dataset, it was necessary to use stratified sampling to help compensate for this imbalance which may result in a high accuracy. We split the $71 \times 125$ tensors into mutually exclusive training and validation sets. Each set is disjoint in both patient and sample acquisition, i.e. no single patient appears in both sets and no two windows from a single acquisition appear in both sets. We follow an 85/15% split for the training/validation set. Due to the large nature of the training set in this situation and the impossibility of training on every triplet possible, a random set of triplets were selected for each training iteration.

## 4.2 Resources

### Tensorflow

Tensorflow and TF-Slim, as described by Abadi et al. [42] and Silberman [43], are frameworks which provided a way to build scalable, computational graphs for machine learning. Tensorflow's use of automatic differentiation, as described by Rall [44], allowed for precise calculations of gradients for networks without error due to floating-point errors. Furthermore, automatic differentiation also helped in this project since the loss function depended on multiple example data points, i.e. the anchor, the positive and the negative, as opposed to a single example data point. TF-Slim's implementation of commonly used computational graph layers (e.g. convolutional layers) helped us define the network without explicitly defining and coding weight matrices.

## SciKit Learn

SciKit Learn is a Python module developed by Pedregosa et al. [45] that provided implementations of common machine learning algorithms and some commonly used accessory functions. In particular, it provided us the k-NN algorithm used throughout the paper to classify validation signals, the confusion matrix calculation function used to analyze our validation results, and the t-SNE reduction algorithm used to analyze the high dimesional latent space in a reduced, 2-dimensional space. SciKit Learn was also built with NumPy and matplotlib, and was easily compatible with the rest of our source code.

## MNE Package

The MNE package is a Python module developed by Gramfort et al. [46] that provided implementations for manipulating biological signal data. It has functions necessary to read, analyze, filter and convert raw data in EDF files to NumPy arrays. These functions allowed us to refine the data instead of processing the raw, time-domain signal.

# 5. Experiment & Results

After a considerable amount of research in clustering and metric learning, we chose triplet loss as our method of approaching the problem. Triplet loss is a well-established, simple and effective when it comes to learning a latent space. Other methods such as the one proposed by Song et al. [29] were overly complicated, especially given the size of our dataset. Triplet loss was relatively easy to implement given how we organized the transformed data and so, a network trained on triplet loss was the natural choice for this experiment.

## 5.1 Initial Experiments

Initially, we did not know whether this method would work on the STFT transformed signals as described in section 4.1 and we needed a simple way to test out the concept. Hence, to learn the latent space, we needed to start with a relatively simple network.

Although the network's architecture was not a priority since this was only a test run of the concept, CNNs were considered from the very beginning since they perform very well in image and video processing tasks as mentioned in section 2.2.3. Spectrograms inherently look like images as we saw in figure 4.2. Since we are using spectrograms of the EEG signals as the input, it made sense to use CNNs. Secondly, we assumed that the labels for each second of each channel of original signal were properly time-aligned in the time domain. In case that this assumption is invalid, CNNs still can perform better than other types of networks. CNNs tend to learn

the patterns in the spectrogram even if they were not time-aligned since they learn shift-invariant features. Hence, an important feature that starts in the first interval of the spectrogram may still be recognized even if it starts sixty intervals later.

**Table 5.1:** Network architecture for CNN

| layer | input | output | kernel |
|---|---|---|---|
| conv1 | $71 \times 125 \times 1$ | $71 \times 125 \times 32$ | $4 \times 4$ |
| pool1 | $71 \times 125 \times 32$ | $35 \times 62 \times 32$ | $3 \times 3$ |
| conv2 | $35 \times 62 \times 32$ | $35 \times 62 \times 64$ | $5 \times 5$ |
| pool2 | $35 \times 62 \times 64$ | $17 \times 30 \times 64$ | $2 \times 2$ |
| fc1 | $17 \times 30 \times 64$ | 256 | N/A |
| fc2 | 256 | 1024 | N/A |
| output | 1024 | 64 | N/A |

Eventually, we built our initial model described in table 5.1 in TensorFlow as described by Abadi et al. [42]. The code in listing A.1 was used to build the network described. We trained the initial model on a Lenovo Y700 laptop running Ubuntu 16.06 LTS with an Intel Core i7-6700HQ CPU running at 2.60 GHz, 8 GB of RAM and no discrete graphics card for Tensorflow's CUDA acceleration capabilities.

In our first attempt, the loss function converged to values very close to zero within the first few iterations. After stepping through the code, we discovered that the input data values were all in the $10^{-5}$ order of magnitude. As a result, the network was discovering a trivial solution that would satisfy the loss function and at the same time not solve the problem at hand. In order to avoid this, we amplified the input data by multiplying all inputs to the network by $10^4$. The network started to train normally and we noticed that the network's loss started to decrease. Semi-hard or hard triplets were chosen at run-time to train the network. Any "soft" triplets were skipped until semi-hard or hard triplets were found since they do not contribute to the learning of the space.

### 5.1.1 Hyperparameter Selection

Initially, we had selected random hyperparameters for the network. Once we realized that the network started to train as we intended it to, we needed to select the hyperparameters the learning rate, $\eta$, the margin parameter, $\alpha$, the regularization strength, $\lambda$ and the output dimension, $d$. More attention was given to $\eta$ since we wanted the network to converge fast but not oscillate before reaching convergence. Despite using the Adam optimizer, the oscillation described in section 2.1.3 can still occur due to small, deep valleys in the hypersurface created by the loss function. In most experiments that we've seen, $\lambda$ is typically a magnitude below the learning rate. We continued that convention and selected $\eta = 10^{-3}$ and $\lambda = 10^{-4}$. Regularization was not necessarily important at this point so not much attention was given to $\lambda$. The rest of the parameters were chosen to be "nice" numbers and were $\alpha = 1.0$, and $d = 128$.

We attempted running the network for a hundred thousand iterations. However, we noticed that after fifty thousand iterations, the triplet mining process was slowed down because the script found it difficult to find semi-hard or hard triplets. At one point, we observed that nearly two hundred thousand triplets were skipped due to their soft nature. Due to this phenomenon, we hypothesized that the network had started to converge at around sixty thousand iterations and decided that it was no longer needed to train. After that point, this particular network architecture was trained for sixty thousand iterations and the data pool was changed every ten thousand iterations to help the script find more semi-hard or hard triplets. This alleviated the stalling nature of triplet mining.

### 5.1.2 Measuring Performance

Although the loss was decreasing as expected, we needed a way of validating whether the space that we anticipated was actually forming. One way to do this was to follow the advice given by Schroff et al. [27] and use a k-NN classifier to quantify the quality of the embedding space produced. Assuming that the data provided was classified correctly, we run some training data through the network to find the embeddings of those samples. We then populate the k-NN space with those embeddings. New embeddings produced from the validation set are introduced to the k-NN algorithm and classified. Theoretically, if the embedding space develops distinct clusters based on the classes, it would classify the validation signals with a relatively high overall accuracy since the validation set's embeddings would be near the cluster.

We used SciKit Learn's implementation of the k-NN classifier to apply the test described. The default value of the number of neighbors, $k = 5$, was used to classify the validation embeddings produced. This test resulted in an 80% accuracy. At this point, we were not using stratified sampling to adjust for the data imbalance. Hence, we assumed that most of the data that was classified incorrectly were originating from classes with low numbers of samples points. It was clear that the experiment was a success and our initial hypothesis that triplet loss can be used learn a latent space for EEGs was valid since the resulting accuracy was relatively high considering that we only used a very simple, two-layer CNN.

### 5.1.3 Error in Dataset Organization

Unfortunately, the success that we experienced with the initial network was not long-lasting. When we explored the various sources of errors in the experiment, it was

obvious that the amount of time used to train the network and the overall accuracy of the network was qualitatively low and high, respectively. Therefore, we tried to inspect the code used so far for any possible errors that we may have introduced. We discovered that the data used to train the network was organized in a way such that the different classes of data was easily split, but the patients, i.e. the different sessions, were not. Therefore, when training, we were implicitly training and validating on a portion of the training set which was already seen.

Our goal was to make this system general so that the system can detect the presence of any of the six signals in any patient, not only for the patients provided in the dataset. Hence, it was necessary for us to reorganize the data so that it was split by session *and* class. We split the dataset again so that sessions from `session1` until `session300` were used as training data and the rest of the sessions were used as testing data while retaining all information about the signal including session, type of signal and time of the signal in the session. This ensured that the training set and the validation set were truly mutually exclusive. Furthermore, the new method of organization helped in conducting analysis of files with seizure-like signals and files with noise-like signals which will be discussed in section 5.3.

## 5.2 DCNN with Triplet Loss

As discussed in section 2.1.4, it is generally easier to cut down a model by using regularization than it is to increase the complexity of a model. Since the concept of using triplet loss to train a CNN for clustering EEG signals was validated, it made sense to proceed to the next step and experimentally increase the complexity of the network. The next logical step from two-layer CNN was to make a DCNN that used multiple layers of convolution to learn more complex shift-invariant features.

We continued the same convolutional layer followed by maxpool layer pattern with a fully connected layer at the end, as seen in the initial network shown in table 5.1. Our new network, specified in table 5.2, consisted of five convolutional layers each followed by a maxpool layer. We built the network using Tensorflow again and trained the model on a server with Intel Xeon ES-2620 24-core CPU with each core at 2.10 GHz, 128 GB of RAM and five Nvidia GeForce Titan X GPUs with 12 GiB of video memory for Tensorflow's GPU acceleration.

**Table 5.2:** Network architecture for simple CNN

| Layer | Input | Output | Kernel |
|---|---|---|---|
| conv1 | $71 \times 125 \times 1$ | $71 \times 125 \times 32$ | $5 \times 5$ |
| maxpool1 | $71 \times 125 \times 32$ | $34 \times 61 \times 32$ | $5 \times 5$ |
| conv2 | $34 \times 61 \times 32$ | $34 \times 61 \times 64$ | $3 \times 3$ |
| maxpool2 | $34 \times 61 \times 64$ | $16 \times 30 \times 64$ | $3 \times 3$ |
| conv3 | $16 \times 30 \times 64$ | $16 \times 30 \times 128$ | $2 \times 2$ |
| maxpool3 | $16 \times 30 \times 128$ | $8 \times 15 \times 128$ | $2 \times 2$ |
| conv4 | $8 \times 15 \times 128$ | $8 \times 15 \times 256$ | $1 \times 1$ |
| maxpool4 | $8 \times 15 \times 256$ | $4 \times 7 \times 256$ | $2 \times 2$ |
| conv5 | $4 \times 7 \times 256$ | $4 \times 7 \times 1024$ | $4 \times 4$ |
| maxpool5 | $4 \times 7 \times 1024$ | $1 \times 2 \times 1024$ | $4 \times 4$ |
| flatten | $1 \times 2 \times 1024$ | 2048 | N/A |
| fc1 | 2048 | 1024 | N/A |
| fc2 | 1024 | 512 | N/A |
| fc3 | 512 | 256 | N/A |
| output | 256 | 64 | N/A |

### 5.2.1 Hyperparameter Selection

The architecture of the network itself can be varied and may be considered a hyperparameter by itself. We can vary the number of layers, the number of neurons in each layer, the convolutional kernel size, the activation functions, parameter ini-

tialization methods and so on. However, this is something that is developed with experience in the field. There are ways to make some of those hyperparameters learnable. For example, the work done by Szegedy et al. [47] allows a single convolutional layer to have variable kernel sizes. When using backpropagation, the network can learn which kernels are used and how much they are used in determining the final output. He et al. [48] work on the parametrization of the ReLU activation function in order to help learn what type of activation function is optimal for a given task. However, such ways of improving the architecture are outside of the scope of this thesis and introduces more issues that increase the overall complexity of the network.

Following the results of the initial experiment, we started to train the new network with the same hyperparameters as the network shown in table 5.1 and decided to pivot on the hyperparameters as necessary in order to find the best model that both generalizes well to the validation set and forms a relatively compressed latent space. We did a manual grid search on a particular range for each of the hyperparameters, $\eta$, $\alpha$, $\lambda$ and $d$. We cross-validated the networks' ability to infer what new signals might be by using the validation set that we had set aside. After training the network and cross-validating, it was found that $\eta = 10^{-4}$ , $\alpha = 0.5$, $\lambda = 10^{-3}$ and $d = 64$ led to the best results.

### 5.2.2 Measuring Performance

We still used k-NN classification accuracy as a measure of the quality of the latent space produced. However, in order to make sure that the classification was being done correctly, we elected to change the number of neighbors that the k-NN algorithm used to a higher value. Increasing $k$ effectively smoothens the decision boundary since the algorithm uses more neighbors to make its decisions. After experimentally increasing

the number of neighbors the algorithm considered to make the classification decision in the latent space, we chose $k = 31$ as it had the highest level of accuracy and completed the task quickly with the amount of available memory. With those hyperparameters, we achieved a validation accuracy of 60.4%. The confusion matrix for that iteration of cross-validation is shown in figure 5.1.

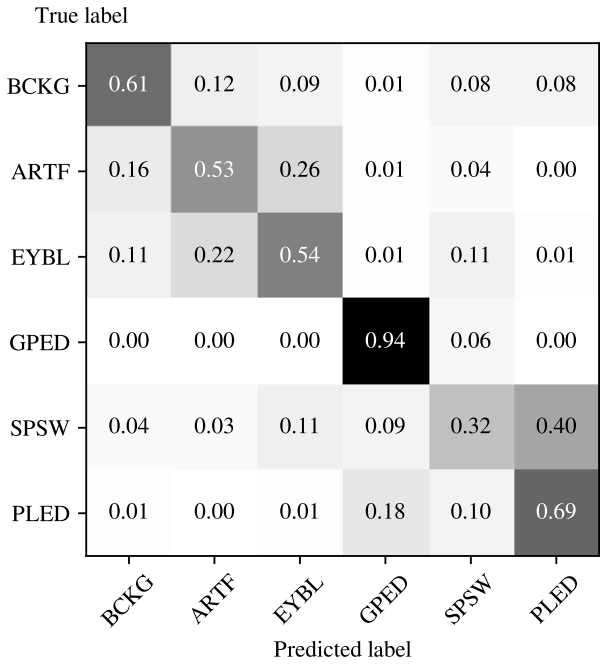


**Figure 5.1:** Confusion matrix for the DCNN clustering network with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations, and an accuracy of 60.4% with 31-NN classification

The classification accuracy provides a numerical value which could be used as a measure of the quality of the embedding that our system produces. However, this does not necessarily provide us information on whether clusters are forming, which is what we were hoping would happen from the beginning. In order to test how well the network was doing in clustering signals based on similarity, we decided to apply t-distributed stochastic neighbor embedding (t-SNE) which is an algorithm that reduces the dimensionality of high dimensional data. Although this algorithm

reduces the dimensionality, the overall structure of the latent space remains the same as the structure of the $d$-dimensional latent space. If clusters exist in the t-SNE plot, it would mean that the same clusters are highly likely to exist in the $d$-dimensional latent space. Hence, we used t-SNE to visualize the 64 dimension latent space in 2D. The t-SNE reduced two dimensional embedding after 5k iterations is shown in figure 5.2 and the same after 105k iterations is shown in figure 5.3.

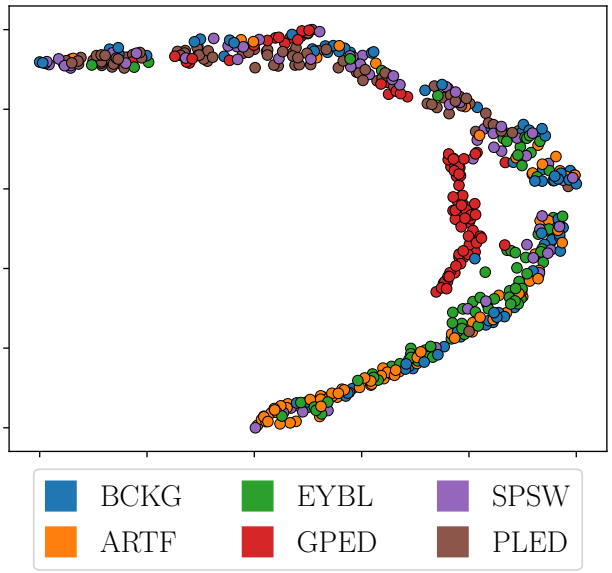


**Figure 5.2:** t-SNE reduced 2D visualization of validation set for the DCNN clustering network after five thousand iterations with $\alpha = 0.5$, $\eta = 10^{-5}$ after 5k iterations, and an accuracy of 26.6% with 31-NN classification after EEG signal was passed through the DCNN clustering network

We can see a clear difference of the t-SNE embedding after 100k iterations. Initially, figure 5.2 shows that the GPED signal is separating out from the crescent in the middle but, the rest of the classes are far off from forming their own clusters and are mixed together. However, figure 5.3 shows that the clusters are forming even in a 2-dimensional space. GPED, BCKG and ARTF have clearly split away from each other and have formed their own clusters.

We see a lot of qualatitative correlation between the confusion matrix in figure 5.1

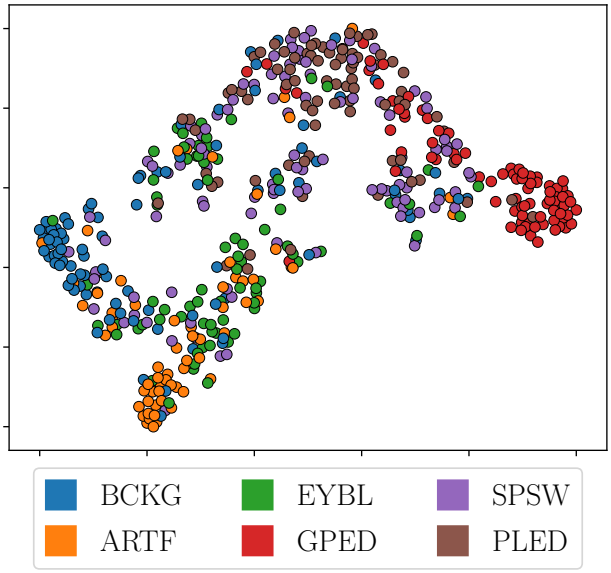


**Figure 5.3:** t-SNE reduced 2D visualization of validation set for the DCNN clustering network with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations, and an accuracy of 60.4% with 31-NN classification after EEG signal was passed through the DCNN clustering network

and t-SNE plot in figure 5.3. For example, according to the confusion matrix GPED was classified correctly 94% of the times that it was encountered in the validation set. This makes sense since the t-SNE plot shows a large cluster of GPED signals. Hence, we can conclude that the clustering algorithm is probably working well because of the amount of qualitative correlation between the t-SNE plot and the confusion matrix.

### 5.2.3 Comparison with a DCNN Classifier

Another way to measure the performance of the clustering network is to compare it with a baseline algorithm. Since we are evaluating the performance of a neural network as a method of clustering EEG signals, we decided to find out how the same architecture as a classifier would perform so that we could compare their performance. In order to keep the same architecture so that we are confident that the architecture

would not make a difference, we add an extra fully-connected layer to the network in table 5.2 as a classification layer with a softmax activation without changing anything else and trained the network on the softmax cross-entropy loss function used in neural network classifiers. We use the same exact hyperparameters as we used in training the clustering network and achieved a validation accuracy of 50.2%. The confusion matrix for the results of this network is shown in figure 5.4.

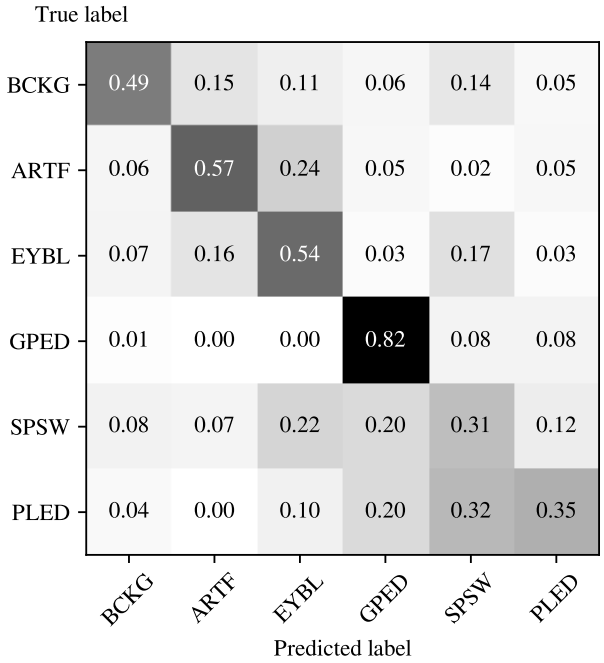


**Figure 5.4:** Confusion matrix for the baseline DCNN classifier with the same hyperparameters as figure 5.1 after 200k iterations and an accuracy of 50.2% with 31-NN classification

These results were perplexing. A classifier is particularly trained on the task of discriminating between different classes whereas our clustering network is trained on the triplet loss which hoped to group similar signals together. It was surprising that a network that was trained on classification did not do better than the network that was trained on clsutering. These results suggest that it may actually better to use the triplet loss in any situation since it provides more information about the original data,

can work with any number of classes and possibly detect new classes once trained, and still perform better than the DCNN on a classification task.

Furthermore, it is likely that this phenomenon occurred directly due to the differences in loss functions since each network had nearly identical architectural forms. The features learned by the DCNN trained on the triplet loss and the features learned by the penultimate layer in the DCNN classifier are probably different because of the difference in the way the networks are trained.

### 5.2.4 Binary Classification Using the Latent Space

We were curious as to how well our system would work if we only used it to classify a signal as either a seizure-like signal or noise-like signal. We considered BCKG, EYBL and ARTF as signals that are noise-like, and SPSW, GPED and PLED to be seizure-like signals as shown in table 4.1. We can pose this as a k-NN classification problem since our network has already been trained. We found the binary classification confusion matrix as shown in figure 5.5 and found the overall accuracy to be 90.2%.

Modifying the t-SNE to label seizure-like signals and noise-like signals, and plotting the decision boundary of a k-NN classifier as shown in figure 5.6 demonstrates that there is a boundary that separates seizure-like signal from noise-like signal clearly. Even though we trained on all types of triplets (e.g. PLED-PLED-GPED, GPED-GPED-BCKG etc.), we still found a clean separation between seizure-like signals and noise-like signals. This phenomena demonstrates that not only are the set of triplet classes that we train on separating, but their super classes, i.e. the general signal types, are separating which can possibly lead to a better, hierarchical taxonomy.

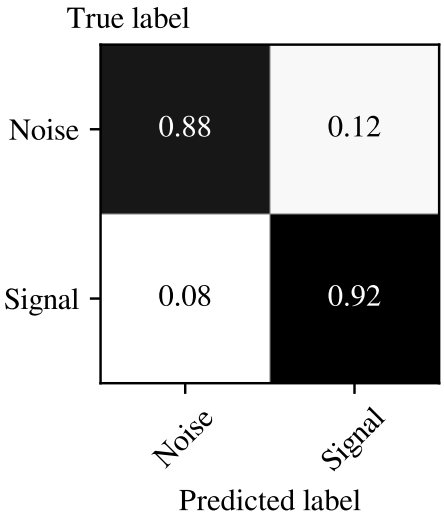


**Figure 5.5:** Binary confusion matrix for the DCNN clustering network with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations, and an accuracy of 90.2% with 31-NN classifier

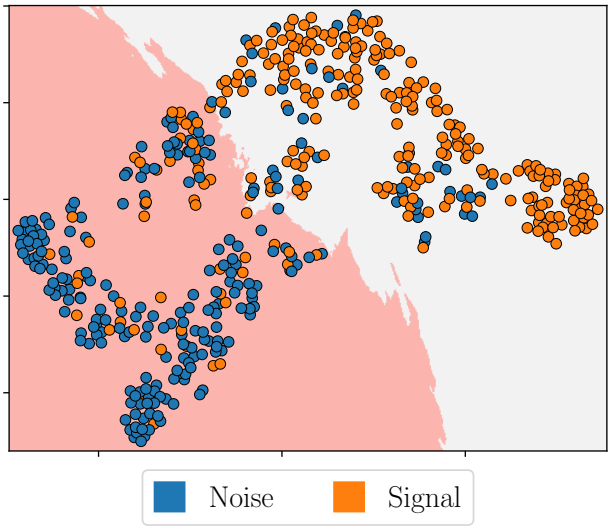


**Figure 5.6:** k-NN classifier decision boundary for t-SNE reduced 2D visualization of validation set for the DCNN clustering network after five thousand iterations with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

## 5.3 Analysis on Seizure-Like & Noise-Like Files

Although our network has done quite well on a rather noisy dataset, a thorough error analysis is certainly the most important step in order to determine how to pivot.

In order to do so, we analyzed how the network does on subsets of the data and tried to discover any patterns or explanations that might help us improve the network in order to get better results. We split the data into the following three subsets:

- sessions without sessions without seizure-like signals
- sessions with seizure-like signals
- sessions with seizure-like signals considering only seizure-like signals

We were able to separate the different signals based on their original session and what types of signals existed in that session. Everytime we computed a confusion matrix for validating the network on the stratified sampled dataset, we also computed a confusion matrix for a stratified sampled subset of the dataset for each of the above categories. In doing so, we obtained the following results.

The confusion matrix shown in figure 5.7 is on the data from sessions that do not contain any seizure-like signals. These sessions only contain BCKG, ARTF and EYBL signals. Hence, the bottom half of the confusion matrix is empty. The right half of the confusion matrix is not completely empty because the DCNN along with the k-NN classifier still predicts some of these signals to be GPED, SPSW or PLED since the network is still trained on the training set which contains all the classes. These incorrect predictions are expected to occur due to various sources of natural background noise, and incorrect true labels or shared characteristics that are closer to seizure-like signals as opposed to noise-like signals. This results in a 64.6% overall accuracy after 105k iterations.

As we had done before, we had also made a binary confusion matrix as well. Just like the confusion matrix presented in figure 5.7, the bottom half of the confusion matrix is empty and the top-right of the confusion matrix is not empty. The system achieved a 93% accuracy in detecting that a second of signal is noise-like and not a

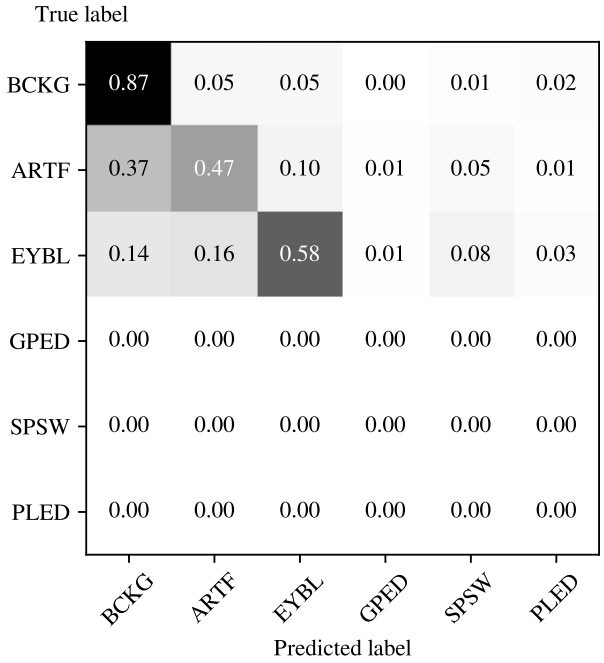


**Figure 5.7:** Confusion matrix of DCNN clustering network on files without seizures resulting in an accuracy of 64.6% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

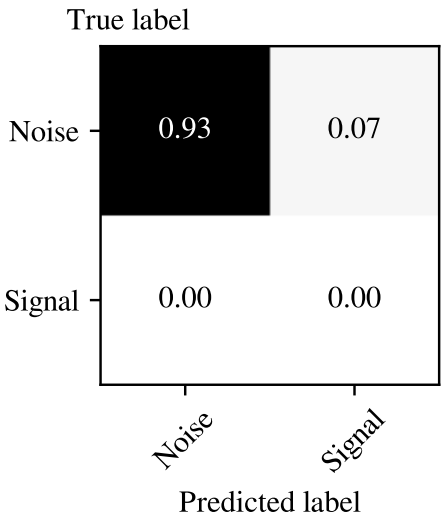


**Figure 5.8:** Binary classification confusion matrix of DCNN clustering network on files without seizures resulting in a binary accuracy of 93.0% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

seizure-like signal. In other words, given only noise-like signals, we are able to classify 93% of those signals as noise-like signals using our system and the 7% as not noise-like (i.e. seizure-like) signals.

The confusion matrix in figure 5.9 is on sessions that contain seizure-like signals. Sessions that contain seizure-like signals also contain noise-like signals since the entire session is not full of seizure-like signals. Therefore, all the types of signals are present in the confusion matrix. However, these sessions are mutually exclusive from the sessions that we looked at in figure 5.7 since those sessions do not contain any seizure-like signals at all. When looking at sessions that contain seizure-like signals, we obtained an overall accuracy of 56% after 105k iterations.

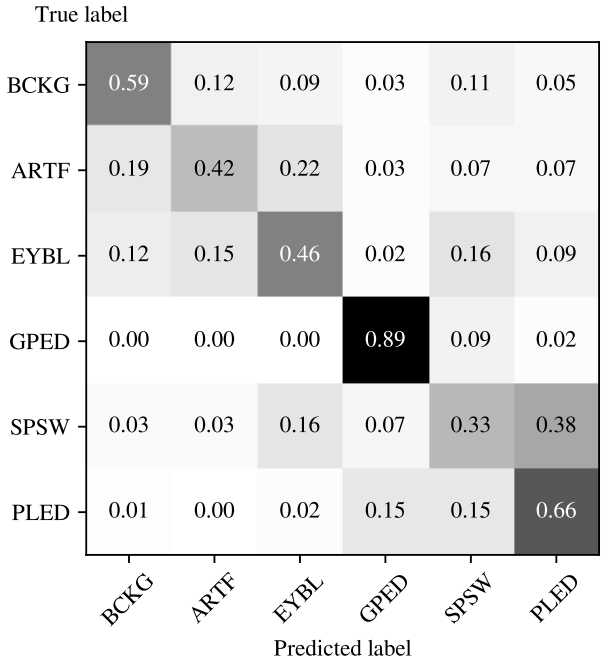


**Figure 5.9:** Confusion matrix of DCNN clustering network on files with seizures resulting in an accuracy of 56.0% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

Like before, we also constructed a binary classification confusion matrix. In this case, given that the session contains a seizure, we are able to classify the signal as

seizure-like or noise-like with an accuracy of 85%. The noise-like signals were able to be detected correctly with a 78% accuracy and the seizure-like signals were able to be detected correctly with a 91% accuracy.

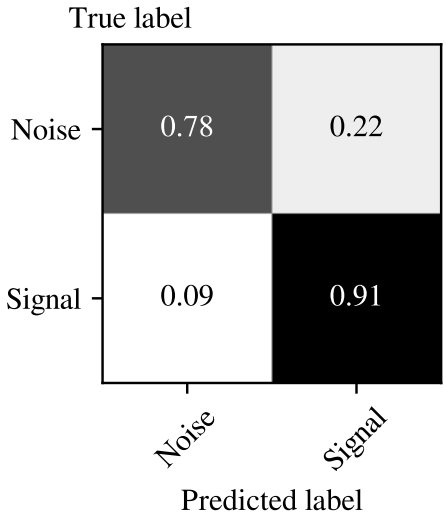


**Figure 5.10:** Confusion matrix of DCNN clustering network on files with seizures resulting in an accuracy of 85.0% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

Finally, the confusion matrix in figure 5.11 is on sessions that contain seizure-like signals but excluding noise-like signals to explore how the system performs on just signals that have seizures (i.e. GPED, SPSW, PLED). This is why the top half of the confusion matrix is empty and we see that most of the predictions are within the bottom right square of the confusion matrix, which is what we expected. Note that the signals that were tested to produce this confusion matrix are not necessarily mutually exclusive from the signals that we tested in figure 5.9 since the signals used to form that confusion matrix were the ones that contained seizures. The experiment results in an overall validation accuracy of 60.4% after 105k iterations. We also see that a lot of the SPSW are being classified as PLED. This is likely because of the high similarity between PLED and SPSW.

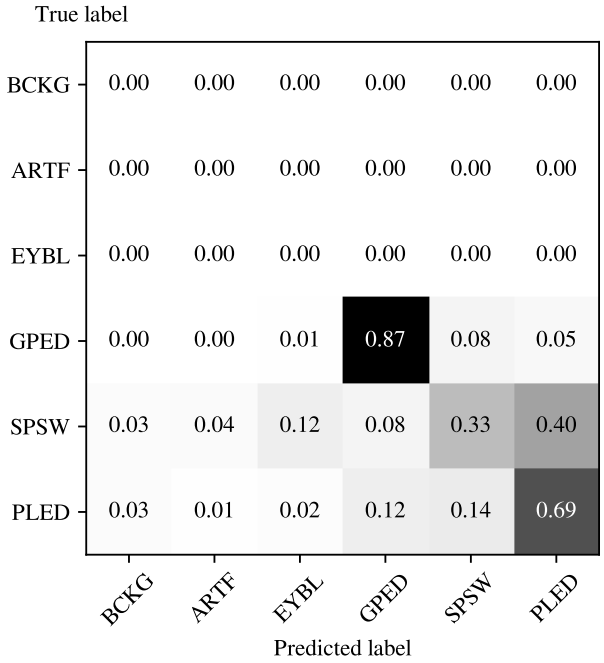


**Figure 5.11:** Confusion matrix of DCNN clustering network on files with ONLY seizure signals resulting in an accuracy of 63.0% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

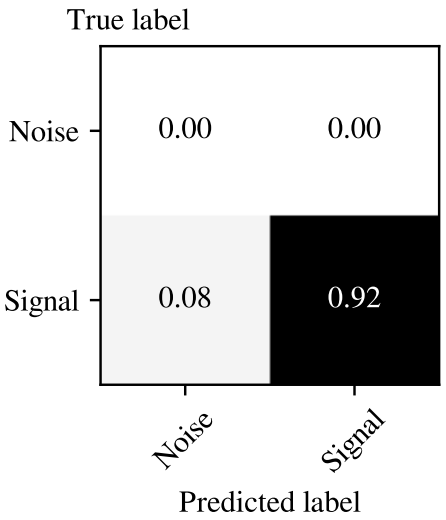


**Figure 5.12:** Confusion matrix of DCNN clustering network on files with ONLY seizure signals resulting in an accuracy of 91.8% with $\alpha = 0.5$, $\eta = 10^{-5}$ after 105k iterations and a 31-NN classifier

Similar to the last experiment, we also generated a binary classification confusion matrix. Given that the session contains a seizure and we are only looking at seizure signals in that particular session, we are able to observe that the signal presented to the system is seizure-like 92% of the time and mis-classified the signal as noise 8% of the time.

In doing the analysis on the subsets of the validation set, it is revealed that most of the error in attempting to recognize a signal as a one of the types of seizure-like signals arises because the signal is classified as one of the other types of seizure-like signals. For example, if a signal with PLED as the true label is presented to the system and the system makes an error in predicting the label of the signal, it is likely for the prediction to be SPSW or GPED as opposed to the noise-like signals. A possible reason for this phenomena may be because the given signal is more similar to SPSW or GPED. This phenomena is alright because the system is expected to cluster and place similar signals near each other. Logically this makes sense since the seizure-like signals are expected to be more similar to each other than noise-like signals. The binary confusion matrices supports this since it has a high true positive rate and a true negative rate.

Another common error that was seen in the various confusion matrices was the relatively high false classification rate of SPSW signals as PLED. This error could be attributed to the similarity between SPSW and PLED, however, it is also likely that amount of data present on SPSW is not enough. Furthermore, we may have also made an error when filtering the raw signal with a pass-band of 1 Hz to 70 Hz. SPSW by definition has high frequencies. It may be possible that some of these high frequencies are above 70 Hz. The assumption that the bulk of the signal is between that pass-band may be false in this case.

# 6. Summary & Future Work

## Summary of Results

We demonstrated an end-to-end system to learn embeddings in a Euclidean space for recognition and clustering using triplet loss. Our network managed to achieve a 60.4% six-class classification accuracy and 90.4% binary classification accuracy. Our work demonstrates that using deep metric learning and deep feature embedding networks, particularly those trained on the triplet loss, may be used to help learn more about EEG signals.

In particular, since our method involves clustering the EEG signals in an embedding space as opposed to directly classifying them, there are a lot more operations that can be done. For example, it may be possible to discover new types of EEGs with no extra training. In the case a new type of signal is discovered outside of the embedding, it might be possible to further train the current model in order for it to learn the new type of signal. Furthermore, the method used in this paper can be used to classifiy a given signal as either seizure-like or noise-like, help automated labeling systems to identify anomalies in EEGs and direct a physician's attention towards these anomalies possibly without the help of an expert in the field. The system can possibly be implemented in a seizure detection device for patients prone to seizures to automatically deploy counter measures and call emergency services in order to maximize survival rate.

# Future Work

Further analysis can be done to determine the usage of this system. For example, we can do an in-depth comparison between the features learned in baseline's latent spaces' penultimate layer and the features learned in the clustering network's latent spaces' final layer. Each network had different accuracies even though they both had identical functional forms. Therefore, a comparison between the two latent spaces speaks directly to the training method for selecting the parameters.

Since the TUH corpus includes natural language physician notes, it may be possible to incorporate these notes to improve the clustering provided by the network. Keywords such as "seizure" or "epilepsy" can be used to bias the network to push the sample towards a cluster containing seizures. We could look at the work provided by Rippel et al. [49] as an inspiration to further improve the clustering through adaptive density discrimination. Perhaps, more advanced versions of the triplet loss, such as the one described by Song et al. [29], to experiment and find out whether they may improve latent space learned by the neural network.

While our system is able to accurately classify the labels, further tests should be conducted to determine its ability to generalize to new labels. Optimally, the network should be able to detect new labels and classify them accordingly. One way to determine the networks' generalization property is to train on five labels and keep the sixth label as a holdout. A generalizing network will be able to cluster the data in a manner such that algorithms that recognize clusters (e.g. Affinity Propagation [50] and Mean Shift [51] clustering algorithms) will be able to detect the sixth without any prior information.

We also can hypothesize that it may be useful to augment the current convolutional architecture with a decoder network to create an autoencoder and train the

autoencoder with both triplet loss as well as the mean-squared-error loss. Autoencoders typically are used to reduce dimensionality of data without losing too much information about the input. Combining this with the triplet loss may help learner richer latent spaces involving features that contribute to high information gain. An extra hyperparameter will probably be introduced to control how much the triplet loss affects the encoding learned by the new network.

Finally, it might be beneficial for us to explore how the same components in this system performs with different types of data, such as MRIs and X-Rays. Although there are a few sources of errors, our system still has a relatively high accuracy and could serve as a stepping stone in directly analyzing, structuring and organizing medical data.

# 7. References


[1] J. Picone, I. Obeid, and S. M. Harabagiu, "Automatic Discovery and Processing of EEG Cohorts from Clinical Records," 2015.

[2] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2013.

[3] S. J. Russell and P. Norvig, *Artificial intelligence*. Prentice Hall, 2009.

[4] Y. LeCun, "The MNIST database of handwritten digits," 1998. [Online]. Available: yann.lecun.com/exdb/mnist/

[5] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cybersecurity intrusion detection," vol. 18, no. 2, pp. 1153–1176, 2016.

[6] E. Hosseini-Asl, G. Gimel'farb, and A. El-Baz, "Alzheimer's Disease Diagnostics by a Deeply Supervised Adaptable 3D Convolutional Network," 2016.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," 2016.

[8] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[9] E. Travers, "They only get cuter," Aug 2009. [Online]. Available: https://www.flickr.com/photos/evantravers/3817862182/

[10] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[11] W. Xu, "Towards optimal one pass large scale learning with averaged stochastic gradient descent," 2011.

[12] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[13] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, pp. 462–466, 1952.

[14] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[16] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," vol. 29, no. 6, pp. 82–97, 2012.

[17] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[19] Wikimedia-Commons, "File:neuron.svg — wikimedia commons, the free media repository," 2017, [Online; accessed 2-February-2018]. [Online]. Available: commons.wikimedia.org/w/index.php?title=File:Neuron.svg

[20] D. H. Hubel and T. N. Wiesel, "Receptive field and functional architecture in two non-striate visual areas (18 and 19) of the cat," vol. 28, no. 2, p. 229, 1965.

[21] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *International conference on artificial neural networks*. Springer, 2010, pp. 92–101.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," vol. 15, no. 1, pp. 1929–1958, 2014. [Online]. Available: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

[23] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8609–8613.

[24] S. Pruthi, M. Clarke, and J. W. Swanson, "EEG (electroencephalogram)," May 2014. [Online]. Available: mayoclinic.org/tests-procedures/eeg/about/pac-20393875

[25] D. Marghescu, "Introduction to EEGs," Aug 2013. [Online]. Available: medicine.mcgill.ca/physio/vlab/biomed_signals/eeg_n.htm

[26] Wikimedia-Commons, "File:21 electrodes of international 10-20 system for eeg.svg — wikimedia commons, the free media repository," 2015, [Online; ac-

cessed 2-February-2018]. [Online]. Available: commons.wikimedia.org/w/index.php?title=File:21_electrodes_of_International_$10$-$20$_system_for_EEG.svg

[27] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 815–823.

[28] Y. Sun, X. Wang, and X. Tang, "Deeply learned face representations are sparse, selective, and robust," 2014.

[29] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese, "Deep metric learning via lifted structured feature embedding," pp. 4004–4012, 2016.

[30] J. Yang, D. Parikh, and D. Batra, "Joint unsupervised learning of deep representations and image clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5147–5156.

[31] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," vol. 10, no. Feb, pp. 207–244, 2009.

[32] J. Wang, Y. Song, T. Leung, C. Rosenberg, J. Wang, J. Philbin, B. Chen, and Y. Wu, "Learning fine-grained image similarity with deep ranking," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1386–1393.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," vol. 323, no. 6088, p. 533, 1986.

[34] E. Choi, M. T. Bahadori, E. Searles, C. Coffey, M. Thompson, J. Bost, J. Tejedor-Sojo, and J. Sun, "Multi-layer representation learning for medical concepts," in

*Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1495–1504.

[35] K. R. Gøeg, R. Cornet, and S. K. Andersen, "Clustering clinical models from local electronic health records based on semantic similarity," *Journal of biomedical informatics*, vol. 54, pp. 294–304, 2015.

[36] Y. Choi, C. Y.-I. Chiu, and D. Sontag, "Learning low-dimensional representations of medical concepts," *AMIA Summits on Translational Science Proceedings*, vol. 2016, p. 41, 2016.

[37] J. Picone, "The tuh eeg corpus." [Online]. Available: https://www.isip.piconepress.com/projects/tuh_eeg/html/downloads.shtml

[38] I. Obeid and J. Picone, "The Temple University Hospital EEG Data Corpus," vol. 10, 2016.

[39] A. Harati, M. Golmohammadi, S. Lopez, I. Obeid, and J. Picone, "Improved EEG event classification using differential energy," in *Signal Processing in Medicine and Biology Symposium (SPMB), 2015 IEEE*. IEEE, 2015, pp. 1–4.

[40] "Normal eeg waveforms," Oct 2017. [Online]. Available: https://emedicine.medscape.com/article/1139332-overview

[41] "13.1 introduction." [Online]. Available: http://www.bem.fi/book/13/13.htm

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[43] N. Silberman, "TF-Slim: A Lightweight Library for Defining, Training and Evaluating Complex Models in TensorFlow," 2017.

[44] L. B. Rall, "Automatic differentiation: Techniques and applications," 1981.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] A. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, R. Goj, M. Jas, T. Brooks, L. Parkkonen *et al.*, "Meg and eeg data analysis with mne-python," *Frontiers in neuroscience*, vol. 7, p. 267, 2013.

[47] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich *et al.*, "Going deeper with convolutions." Cvpr, 2015.

[48] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[49] O. Rippel, M. Paluri, P. Dollar, and L. Bourdev, "Metric learning with adaptive density discrimination," 2015.

[50] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[51] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature

space analysis," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

# A. Code Sample

**Listing A.1:** Initial Model

```
1  def get_model(input, reuse=False):
2          with slim.arg_scope([layers.conv2d, layers.fully_connected], weights_initializer=layers.xavier_initializer(seed=
               ↪ random.random(), uniform=True), weights_regularizer=slim.l2_regularizer(0.05), reuse=reuse):
3                  net = tf.expand_dims(input, axis=3)
4                  net = layers.conv2d(net, num_outputs=32, kernel_size=4, scope='conv1', trainable=True)
5                  net = layers.max_pool2d(net, kernel_size=3, scope='maxpool1')
6                  net = layers.conv2d(net, num_outputs=64, kernel_size=5, scope='conv2', trainable=True)
7                  net = layers.max_pool2d(net, kernel_size=3, scope='maxpool2')
8                  net = layers.flatten(net, scope='flatten')
9                  net = layers.fully_connected(net, 256, scope='fc1', trainable=True)
10                 net = layers.fully_connected(net, 1024, scope='fc2', trainable=True)
11                 net = layers.fully_connected(net, num_output, activation_fn=None, weights_regularizer=None,
                       ↪ scope='output')
12                 return net
```

**Listing A.2:** Enlarged Model

```
1  def simple_model(inputs, reuse=False):
2          with slim.arg_scope([slim.layers.conv2d, slim.layers.fully_connected], weights_initializer=tf.contrib.layers.
               ↪ xavier_initializer(uniform=True), weights_regularizer=slim.l2_regularizer(l2_weight), reuse=reuse):
3                  net = tf.expand_dims(inputs, dim=3)
4                  net = slim.layers.conv2d(net, num_outputs=32, kernel_size=5, scope='conv1', trainable=True)
5                  net = slim.layers.max_pool2d(net, kernel_size=5, scope='maxpool1')
6                  net = slim.layers.conv2d(net, num_outputs=64, kernel_size=3, scope='conv2', trainable=True)
7                  net = slim.layers.max_pool2d(net, kernel_size=3, scope='maxpool2')
8                  net = slim.layers.conv2d(net, num_outputs=128, kernel_size=2, scope='conv3', trainable=True)
9                  net = slim.layers.max_pool2d(net, kernel_size=2, scope='maxpool3')
10                 net = slim.layers.conv2d(net, num_outputs=256, kernel_size=1, scope='conv4', trainable=True)
11                 net = slim.layers.max_pool2d(net, kernel_size=2, scope='maxpool4')
12                 net = slim.layers.conv2d(net, num_outputs=1024, kernel_size=4, scope='conv5', trainable=True)
13                 net = slim.layers.max_pool2d(net, kernel_size=4, scope='maxpool5')
14                 net = slim.layers.flatten(net, scope='flatten')
15                 net = slim.layers.fully_connected(net, 1024, scope='fc1', trainable=True)
16                 net = slim.layers.fully_connected(net, 512, scope='fc2', trainable=True)
17                 net = slim.layers.fully_connected(net, 256, scope='fc3', trainable=True)
18                 net = slim.layers.fully_connected(net, num_output, weights_regularizer=None, scope='output')
19                 return net
```

**Listing A.3:** Triplet Loss Function

```
def triplet_loss(alpha):
        anchor = tf.placeholder(tf.float32, shape=input_shape)
        positive = tf.placeholder(tf.float32, shape=input_shape)
        negative = tf.placeholder(tf.float32, shape=input_shape)
        anchor_out = get_model(anchor, reuse=True)
        positive_out = get_model(positive, reuse=True)
        negative_out = get_model(negative, reuse=True)
        with tf.variable_scope('triplet_loss'):
                pos_dist = distance_metric(anchor_out, positive_out, metric='euclidean')
                neg_dist = distance_metric(anchor_out, negative_out, metric='euclidean')
                basic_loss = tf.add(tf.subtract(pos_dist, neg_dist), alpha)
                loss = tf.reduce_mean(tf.maximum(basic_loss, 0.0), 0)
                return loss
```

**Listing A.4:** Runner Script

```
import random
import tensorflow as tf
from sklearn.svm import SVC
from BrainNet import BrainNet

for run in range(0, 1):
        batch_size = 5000
        alpha = 0.5
        learning_rate = 0.0001
        l2_weight = 0.001
        validation_size = 500

        print('Run: {:d}, Alpha: {:1.1f}, Learning Rate: {:3.2e}, L2−Weight: {:3.2e}, Batch Size: {:d}'.format(run
            ↪ + 1, alpha, learning_rate, l2_weight, batch_size))
        #path_to_files='/home/krishna/data',
        net = BrainNet(path_to_files='/home/krishna/data' alpha=alpha, validation_size=validation_size
            ↪ learning_rate=learning_rate, l2_weight=l2_weight batch_size=batch_size, debug=True, train_epoch
            ↪ =20)
        _, val_percent, val_conf_matrix = net.train_model()

        print('Validation Percentage: {:2.2f}\nConfusion Matrix:\n{}'.format(val_percent, val_conf_matrix))
```

**Listing A.5:** Accessory Functions

```
def norm_op(vector, axisss):
        return normalize(vector, axis=axisss, norm='l2')
        #return vector * 10e4

def plot_embedding(X, y, epoch, accuracy, num_to_label, title):
        x_min, x_max = np.min(X, 0), np.max(X, 0)
        X = (X - x_min) / (x_max - x_min)
        cmap = plt.get_cmap('gist_rainbow')
        color_map = [cmap(1.*i/6) for i in range(6)]
        legend_entry = []
        for ii, c in enumerate(color_map):
                legend_entry.append(matplotlib.patches.Patch(color=c, label=num_to_label[ii]))


        plt.figure(figsize=(4.0, 4.0))
        plt.scatter(X[:,0], X[:, 1], c=y, cmap=matplotlib.colors.ListedColormap(color_map), s=2)
        plt.legend(handles=legend_entry)
        plt.xticks([]), plt.yticks([])
        plt.title(title)
        plt.savefig('./%s Results/%s_tSNE_plot_epoch%s_%.3f%%.pdf' % (curr_time, curr_time, epoch, accuracy),
            ↪ bbox_inches='tight')

def compute_tSNE(X, y, epoch, accuracy, num_to_label, with_seizure=None, title="t-SNE Embedding of DCNN
    ↪ Clustering Network"):
        tsne = TSNE(n_components=2, init='random', random_state=0)
        X_tsne = tsne.fit_transform(X)
        plot_embedding(X_tsne, y, epoch=epoch, accuracy=accuracy, num_to_label=num_to_label, title=title)
        if with_seizure is None:
                np.savez('./%s Results/%s_tSNE_plot_epoch%s_%.3f%%' % (curr_time, curr_time, epoch, accuracy),
                    ↪ X_tsne, y)
        elif with_seizure == True:
                np.savez('./%s Results/%s_tSNE_plot_with_seizure_epoch%s_%.3f%%' % (curr_time, curr_time,
                    ↪ epoch, accuracy), X_tsne, y)
        elif with_seizure == False:
                np.savez('./%s Results/%s_tSNE_plot_without_seizure_epoch%s_%.3f%%' % (curr_time, curr_time,
                    ↪ epoch, accuracy), X_tsne, y)

```

```
33 def get_loss(loss_mem, loss_mem_skip):
34         plt.figure(figsize=(4.0, 4.0))
35         plt.plot(loss_mem_skip, 'ro−', markersize=2)
36         plt.xlabel("1000 Iterations")
37         plt.ylabel("Average Loss in 1000 Iterations")
38         plt.title("Iterations vs. Average Loss")
39         plt.savefig('./%s Results/%s_convergence_with_skip_plot.pdf' % (curr_time, curr_time), bbox_inches='tight')
40
41         plt.figure(figsize=(4.0, 4.0))
42         plt.plot(loss_mem, 'ro−', markersize=2)
43         plt.xlabel("1000 Iterations")
44         plt.ylabel("Average Loss in 1000 Iterations")
45         plt.title("Iterations vs. Average Loss")
46         plt.savefig('./%s Results/%s_convergence_plot.pdf' % (curr_time, curr_time), bbox_inches='tight')
47
48
49 def plot_confusion_matrix(cm, classes, normalize=True, cmap=plt.cm.Greys, accuracy = None, epoch=None,
       ↪ with_seizure=None, title = "Confusion Matrix on All Data"):
50         plt.figure(figsize=(4, 4))
51         plt.imshow(cm, interpolation='nearest', cmap=cmap)
52         ax = plt.gca()
53         #plt.colorbar()
54         tick_marks = np.arange(len(classes))
55         plt.xticks(tick_marks, classes, rotation=45)
56         plt.yticks(tick_marks, classes)
57         ax.yaxis.set_label_coords(−0.1,1.03)
58         h = ax.set_ylabel('True label', rotation=0, horizontalalignment='left')
59
60         if normalize:
61                 cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
62                 print("Normalized confusion matrix")
63         else:
64                 print('Confusion matrix, without normalization')
65
66         print(cm)
67
68         thresh = cm.max() / 2.
69         for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
70                 plt.text(j, i, '{0:.2f}'.format(cm[i, j]), horizontalalignment="center", verticalalignment="center",
```

```
                    ↪ color="white" if cm[i, j] > thresh else "black")
71
72          #plt.tight_layout()
73          plt.xlabel('Predicted label')
74          plt.title(title)
75          #plt.show()
76          if with_seizure is None:
77                  plt.savefig('./%s Results/%s_confusion_matrix_epoch%s_%.3f%%.pdf' % (curr_time, curr_time,
                        ↪ epoch, accuracy), bbox_inches='tight')
78          elif with_seizure == True:
79                  plt.savefig('./%s Results/%s_confusion_matrix_with_seizure_epoch%s_%.3f%%.pdf' % (curr_time,
                        ↪ curr_time, epoch, accuracy), bbox_inches='tight')
80          elif with_seizure == False:
81                  plt.savefig('./%s Results/%s_confusion_matrix_without_seizure_epoch%s_%.3f%%.pdf' % (curr_time,
                        ↪ curr_time, epoch, accuracy), bbox_inches='tight')
```

**Listing A.6:** Triplet Mining

```
def get_triplets(size=10):
    A = []
    P = []
    N = []

    for _ in range(size):
        choices = ['bckg', 'eybl', 'gped', 'spsw', 'pled', 'artf']
        neg_choices = list(choices)
        choice = random.choice(choices)
        neg_choices.remove(choice)

        if choice == 'bckg':
            a = np.load(random.choice(bckg))
            p = np.load(random.choice(bckg))
        elif choice == 'eybl':
            a = np.load(random.choice(eybl))
            p = np.load(random.choice(eybl))
        elif choice == 'gped':
            a = np.load(random.choice(gped))
            p = np.load(random.choice(gped))
        elif choice == 'spsw':
            a = np.load(random.choice(spsw))
            p = np.load(random.choice(spsw))
        elif choice == 'pled':
            a = np.load(random.choice(pled))
            p = np.load(random.choice(pled))
        else:
            a = np.load(random.choice(artf))
            p = np.load(random.choice(artf))

        neg_choice = random.choice(neg_choices)

        if neg_choice == 'bckg':
            n = np.load(random.choice(bckg))
        elif neg_choice == 'eybl':
            n = np.load(random.choice(eybl))
        elif neg_choice == 'gped':
```

```
38                  n = np.load(random.choice(gped))
39              elif neg_choice == 'spsw':
40                  n = np.load(random.choice(spsw))
41              elif neg_choice == 'pled':
42                  n = np.load(random.choice(pled))
43              else:
44                  n = np.load(random.choice(artf))
45
46              key = choice + choice + neg_choice
47
48              if key in count_of_triplets:
49                  count_of_triplets[key]+=1
50              else:
51                  count_of_triplets[key] = 1
52
53              a = norm_op(a, axisss=0)
54              p = norm_op(p, axisss=0)
55              n = norm_op(n, axisss=0)
56              A.append(a)
57              P.append(p)
58              N.append(n)
59
60
61      A = np.asarray(A)
62      P = np.asarray(P)
63      N = np.asarray(N)
64      return A, P, N
```

**Listing A.7:** Model Training

```
def train_model(outdir=None):
    loss = triplet_loss(alpha=alpha)
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    optim = optimizer.minimize(loss=loss)
    sess.run(tf.global_variables_initializer())

    count = 0
    ii = 0
    val_percentage = 0
    val_conf_matrix = 0
    epoch = -1
    while True:
        epoch += 1
        ii = 0
        count = 0
        temp_count = 0
        full_loss = 0
        while ii <= batch_size:
            ii += 1
            a, p, n = get_triplets()

            temploss = sess.run(loss, feed_dict={anchor: a, positive: p, negative: n})

            if temploss == 0:
                ii -= 1
                count += 1
                temp_count += 1
                continue

            full_loss += temploss

            if ((ii + epoch * batch_size) % 1000 == 0):
                loss_mem_skip.append(full_loss / (1000.0 + temp_count))
                loss_mem.append(full_loss / (1000.0))
                full_loss = 0
                temp_count = 0
                get_loss(loss_mem, loss_mem_skip)
```

```
38
39                  _, a, p, n = sess.run([optim, anchor_out, positive_out, negative_out], feed_dict={anchor: a, positive: p
                        ↪ , negative: n})
40
41                  d1 = np.linalg.norm(p − a)
42                  d2 = np.linalg.norm(n − a)
43
44                  if DEBUG:
45                      print("Epoch: %2d, Iter: %7d, IterSkip: %7d, Loss: %.4f, P_Diff: %.4f, N_diff: %.4f" % (epoch,
                            ↪ ii, count, temploss, d1, d2))
46              val_percentage, val_conf_matrix = validate(epoch)
47      sess.close()
48      return epoch, val_percentage, val_conf_matrix
```

**Listing A.8:** Validation Script

```
def validate(epoch):
    inputs, classes = get_sample(size=100, validation=True)
    vector_inputs = sess.run(inference_model, feed_dict={inference_input: inputs})
    del inputs

    tempClassifier = neighbors.KNeighborsClassifier(31)
    tempClassifier.fit(vector_inputs, classes)

    # All data (Files with Seizures & Files without Seizures)

    val_inputs, val_classes = get_sample(size=validation_size)
    vector_val_inputs = sess.run(inference_model, feed_dict={inference_input: val_inputs})
    del val_inputs

    pred_class = tempClassifier.predict(vector_val_inputs)

    percentage = len([i for i, j in zip(val_classes, pred_class) if i == j]) * 100.0 / validation_size

    if DEBUG:
        print("Validation Results: %.3f%% of of %d correct" % (percentage, validation_size))

    val_classes = list(map(lambda x: num_to_class[x], val_classes))
    pred_class = list(map(lambda x: num_to_class[x], pred_class))
    class_labels = [0, 1, 2, 3, 4, 5]
    class_labels = list(map(lambda x: num_to_class[x], class_labels))
    conf_matrix = confusion_matrix(val_classes, pred_class, labels=class_labels)
    np.set_printoptions(precision=2)

    np.save('./%s Results/%s_confusion_matrix_epoch%s_%.3f%%' % (curr_time, curr_time, epoch, percentage),
        ↪ conf_matrix)

    plot_confusion_matrix(conf_matrix, classes=class_labels, epoch=epoch, accuracy=percentage)

    compute_tSNE(vector_inputs, classes, epoch=epoch, accuracy=percentage, num_to_label=num_to_class)

    # Files with Seizures

```

```
37      val_inputs_seizure, val_classes_seizure = get_sample(size=validation_size)
38      vector_val_inputs_seizure = sess.run(inference_model, feed_dict={inference_input: val_inputs_seizure})
39      del val_inputs_seizure
40
41      pred_class_seizure = tempClassifier.predict(vector_val_inputs_seizure)
42
43      percentage_seizure = len([i for i, j in zip(val_classes_seizure, pred_class_seizure) if i == j]) * 100.0 /
            ↪ validation_size
44
45      if DEBUG:
46          print("Validation Results: %.3f%% of of %d correct" % (percentage_seizure, validation_size))
47
48      val_classes_seizure = list(map(lambda x: num_to_class[x], val_classes_seizure))
49      pred_class_seizure = list(map(lambda x: num_to_class[x], pred_class_seizure))
50      class_labels_seizure = [0, 1, 2, 3, 4, 5]
51      class_labels_seizure = list(map(lambda x: num_to_class[x], class_labels_seizure))
52      conf_matrix_seizure = confusion_matrix(val_classes_seizure, pred_class_seizure, labels=class_labels_seizure)
53      np.set_printoptions(precision=2)
54
55      np.save('./%s Results/%s_confusion_matrix_with_seizure_epoch%s_%.3f%%' % (curr_time, curr_time, epoch,
            ↪ percentage_seizure), conf_matrix_seizure)
56
57      plot_confusion_matrix(conf_matrix_seizure, classes=class_labels_seizure, epoch=epoch, accuracy=
            ↪ percentage_seizure, with_seizure=True, title = "Confusion Matrix on Files with Seizure")
58
59      #compute_tSNE(vector_inputs, classes, epoch=epoch, accuracy=percentage_seizure, num_to_label=num_to_class
            ↪ )
60
61      # Files without Seizures
62
63      val_inputs_without_seizure, val_classes_without_seizure = get_sample(size=validation_size)
64      vector_val_inputs_without_seizure = sess.run(inference_model, feed_dict={inference_input:
            ↪ val_inputs_without_seizure})
65      del val_inputs_without_seizure
66
67      pred_class_without_seizure = tempClassifier.predict(vector_val_inputs_without_seizure)
68
69      percentage_without_seizure = len([i for i, j in zip(val_classes_without_seizure, pred_class_without_seizure) if i
            ↪ == j]) * 100.0 / validation_size
```

```
70
71      if DEBUG:
72          print("Validation Results: %.3f%% of of %d correct" % (percentage_without_seizure, validation_size))
73
74      val_classes_without_seizure = list(map(lambda x: num_to_class[x], val_classes_without_seizure))
75      pred_class_without_seizure = list(map(lambda x: num_to_class[x], pred_class_without_seizure))
76      class_labels_without_seizure = [0, 1, 2, 3, 4, 5]
77      class_labels_without_seizure = list(map(lambda x: num_to_class[x], class_labels_without_seizure))
78      conf_matrix_without_seizure = confusion_matrix(val_classes_without_seizure, pred_class_without_seizure, labels=
            ↪ class_labels_without_seizure)
79      np.set_printoptions(precision=2)
80
81      np.save('./%s Results/%s_confusion_matrix_without_seizure_epoch%s_%.3f%%' % (curr_time, curr_time, epoch,
            ↪ percentage_without_seizure), conf_matrix_without_seizure)
82
83      plot_confusion_matrix(conf_matrix_without_seizure, classes=class_labels_without_seizure, epoch=epoch,
            ↪ accuracy=percentage_without_seizure, with_seizure=False, title = "Confusion Matrix on Files without
            ↪  Seizure")
84
85      #compute_tSNE(vector_inputs, classes, epoch=epoch, accuracy=percentage_without_seizure, num_to_label=
            ↪ num_to_class)
86
87      count_of_triplets = dict()
88
89      return percentage, conf_matrix
```