BrainNet: Learning an Embedding Space for EEGs

Krishna Thiyagarajan
Department of Electrical Engineering
Albert Nerken School of Engineering
The Cooper Union
New York, New York 10003
Email: krisht@cooper.edu

Zhengqi Xi
Department of Mechanical Engineering
Albert Nerken School of Engineering
The Cooper Union
New York, New York 10003
Email: xi@cooper.edu

***Abstract*—Despite the advances in medical data organization and structuring, electronic medical records (EMRs) can often contain unstructured raw data, temporally constrained measurements, multi-channel signal data and image data all of which are often difficult to compare and contrast in large quantities. We present a system that can learn a function from EEG data to a relatively compressed n-dimensional Euclidean space where the distances between the data points can be used as a measure of similarity. We utilize a deep, convolutional encoder that attempts to optimize the n-diemnsional Euclidean space by using triplet loss. It is our hope that such a system can be expanded to other types of unstructured raw medical data to make it easier to explore, query and find nontrivial correlations between common ailments.**

On the data provided by Temple University, TUH EEG Cohorts, our network achieves a 80% accuracy on classification of EEG signals using our embedding along with kNN classification algorithms.

***Keywords*—EEG, Triplet, Clustering, EMR**

I. Introduction

Electronic Medical Records are a very common method of storing patient data and medical history. These can be used to recover information, images, tests and other patient related data. However, a lot of these EMRs contain data that is in its raw form. Examples may include physical signals (e.g. EEG, EKG, MEG etc.), images (e.g. X-rays, MRIs, CAT) and textual notes on patients by medical professionals. Because of the crudeness of the data, it is often difficult to compare and contrast patient files in order to come up with a meaning for why some diseases form. For example, Alzheimers Disease is a very common chronic neuro-degenerative disease. We have a lot of data on the disease, however, we are unable to find the underlying cause or "meaning" as some would say to why this disease forms and worsens over time. We only know the symptoms and not the cause. However, if such data can be harnessed and can be made comparable between different patients, it would be very useful for medical professionals to pinpoint the cause of the disease and discover better ways of treatment.

With the ultimate goal of organizing unstructured medical data, we present a system for analyzing raw EEG cohorts for signal recognition (is this signal in this class) and clustering (finding similar signals) and possible relations between different classes of diseases. Our method seeks to learn a metric or embedding space using a deep convolutional neural network on one second EEG signals sampled at $250\ Hz$. The network is trained such that the euclidean distance between any two signals is a measurement of how similar the signals. Signals that have the same class and have similar features are expected to be close together and have smaller distances while those that are different are expected to have larger distances between each other in the metric space constructed by the computational graph.

We can then use the network to transform the EEG signals into an embedding space in which more queries can be performed. For example, we can reduce recognizing signals to a kNN classification problem and reduce clustering signals based on their features into a k-means clustering problem.

Previous signal event detection and classification approaches have used hidden Markov models. These

models in general are simple enough and works well enough for handling sequential data. Although these models achieve 91.4% sensitivity and 8.5% specificity with respect to recognizing the different type of signals, it is difficult to determine the underlying cause of the spikes and the abnormalities.

In contrast to the hidden Markov Models, our solution uses a deep convolutional neural network which optimizes a triplet loss objective function and directly trains the embedding space that the network outputs. Triplets of anchors, positives and negatives were chosen such that the anchor and positive were of the same class and the anchor and the negative were of different classes.

We organize the rest of the paper in the following way. In II, we review literature and inspiration from which our model was constructed. In III, we describe our data and how we organized it for easy triplet mining. In **??**, we describe the model(s) that we used. In IV, we describe our experiments and quantitative and qualitative results about the entire network.

II. Related Work

The concept of producing an embedding space by using triplet loss to cluster data points has already been well established in different contexts.

One of the implementations that inspired our network was done by [1]. FaceNet's system used a convolutional neural network with a triplet loss objective to optimize an embedding space so that facial recognition, similarity and clustering could be achieved. The paper found anchors, positives and negatives from its dataset and optimized the distance between them by minimizing the distance between the anchors and positives and maximizing the distance between the anchor and the negative. Mathematically,

$$||f(x_i^a) - f(x_i^p)||^2 + \alpha < ||f(x_i^a) - f(x_i^n)||^2$$

where $f(x) \in \mathbb{R}^d$ 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 α is a hyper parameter expressing the minimum distance between. Thus, the objective function becomes

$$J = max(0, \sum_i^N ||f(x_i^a) - f(x_i^p)||^2 + \alpha - ||f(x_i^a) - f(x_i^n)||^2)$$

which directly optimizes the embedding space. This paper was important because it was the first paper to achieve a working model of a triplet loss objective. This network is considered a breakthrough paper on triplet loss to cluster human faces for facial recognition.

Another paper authored by [2] describes a similar image metric learning scheme using lifted structured embedding. They proposed a method where each positive pair compares the distances against all the negative pairs weighted by the minimum distance expected between them (α). The advantage of this was that the loss function they used has a differentiable loss which incorporates "online hard negative mining" functionality. This way, they were able to define an end to end system based on structured prediction which not only looks at the local structure but also the global structure of the embedding space being constructed by the deep neural network through their loss function. As opposed to the FaceNet paper, this paper's methods doesn't require preparation of data in tuples. They tested and showed state of the art performance on standard datasets such as CUB200-2011, Cars196 and Stanford online products. However, their loss function seems overly complicated since they look at each positive pair against each negative pairs.

A recurrent solution was proposed by [3] under the name Joint Unsupervised Learning (JULE) to again cluster and represent images as embeddings depending on their features. However, it is known that recurrent networks are rather hard to optimize and are very unstable when it comes to training. One cannot determine whether the given recurrent network will actually converge to a solution or not. Thus, this is not a very elegant solution to generate embeddings for images (or EEGs for that matter).

A method for learning an embedding using Mahalanobis distance was proposed by [4].

Although many different solutions were proposed to the same type of problem, [1] tended to be the most developer friendly. First of all, finding triplets in the 6 classes was found to be relatively easy. Secondly, using these triplets to train was also straightforward. Finally, we didn't need all of the different possible triplets at once like [2] proposed. Therefore, we employed Facenets base methods to approach this problem.

The goal of this particular project is to try and predict various ailments that can be diagnosed with EEG studies.

III. Data

Data for this project was mostly derived from the Temple University Hospital's EEG corpus (TUH EEG corpus). The data consists of raw European Data Format (EDF+) files, a format for exchanging and storing multichannel biological and physical signals, and their corresponding labels of each in label (LBL) files. The label files can be interpreted by Temple Unversity's python script on their website which can be used to transform the label files into data files indicting the montage of interest, the label of interest for each second of data recorded. There are a total of 6 EEG labels provided by the TUH EEG corpus: BCKG, ARTF, EYBL, SPSW, PLED and GPED. BCKG is the label for background noise, ARTF is the label for artifacts, EYBL is the label for eyeball movement, SPSW is the label for spikes & sharp waves, PLED is the label for periodic lateralized epileptiform discharges and GPED is the label for generalized periodic epileptiform discharges. All of these labels are of interest for our purposes since the purpose of this project is to learn a compressed embedding which may lead to more meaning when looking at EEG signals.

The EDF files themselves were deciphered using the python MNE package. They contain raw signals with different channels placed in the standard 10-20 system. A total of 22 montages, i.e. voltage differences between channels of interest, were contained in the label file. A notch filter at 60 Hz and 120 Hz were applied to get rid of any power line noise in the signal and a band pass filter between 1 Hz and 70 Hz with a transition band at the high cutoff frequency of 10 Hz was applied to remove any high frequency noise. Short Term Fourier Transform (STFT) with a window size of 140 samples and a time step of 2 samples was taken which resulted in a $71 \times 125 \times 1$ dimensional matrix which was then stored in the NumPy NPZ format in different files depending on the labels. Doing this was useful as our method required triplets of anchors, positive and negative signals as described previously. We were able to "pre-mine" for triplets before the process began which made the training process easier. The raw time domain data was not utilized in this case because literature indicated that the frequency domain is what contains data that is useful for our purpose and not the time domain. Notice that we did not maintain the entire structure of the original 22 montages supplied by the original labels. This decision was made because it would take a lot more computations to handle and the labels were on each montage not on a time slice as a whole. Thus, such a large tensor will be of little value for our purposes.

Most of the data (more than 80%) was labeled as background noise. Therefore, the data contains a natural bias towards background noise as opposed to true labels such as SPSWs, PLEDs nad GPEDs. Stratified sampling was then utilized to alleviate this natural bias by trying to equalize the number of data points available in each category.

These files were then split into 70% training, 20% validation and 10% testing sets on each class using random selection for future use.

IV. Experimental Design & Results

The below model in table IV was built in TensorFlow [5]. The code for the training is attached for reference.

Initially, we were only validating the idea of whether triplet loss on a deep convolutional network could be used to put these signals that are in the time domain to a higher level feature domain that serves as the embedding so that clustering, classifying and other common machine learning tasks could be attempted on them. Therefore, we did not consider the network's architecture to be a priority. However, we do need a network to begin with, so we started out with the architecture below:

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

TABLE I. NETWORK ARCHITECTURE ON WHICH PROPOSAL WAS TESTED ON

In our first attempt, it turned out that the network was discovering a trivial (or close to trivial) solution. The loss would converge to zero within the first three iterations. It was discovered that the input data magnitudes were in the 10^{-5} order of magnitude and therefore, the network was discovering a trivial solution that would satisfy the objective 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. This allowed the network to start training normally and no particular side effects were seen.

Since α was a hyper parameter, we manually selected some αs and trained the network. In these experiments, $\alpha = 0.5$ was optimal for this particular network.

We attempted running the network for a total of 10 epochs with each epoch having 10 thousand iterations. However, we noticed that after the 5th epoch, the triplet mining process was slowed down by a lot. The amount of semi-hard or hard triplets that was found compared to the soft triplets found were different in magnitudes. At one point, we observed that there were nearly 200 thousand iterations skipped due to soft triplets and around 5 thousand iterations were ran in the same epoch with semi-hard or hard triplets. Due to this ratio, we hypothesized that the network had started to converge at around 6 epochs and decided that it was no longer needed to train. After that experiment, the network ran for 10 thousand iterations for 6 epoch, where each epoch was from a different set of sample data.

After this run, we realized that classifying using the embedding space of certain signals is a measure of how well the produced embedding is. The higher percentage of classification accuracy we have, the better the embedding. Therefore, we used SciKit Python's implementation of k nearest neighbors algorithm to classify validation data that we had prepared in order to test accuracy. This resulted in a 80% accuracy. Most of the data that was incorrect was originating from the classes with low numbers of sample points which is expected.

All in all, our experiments resulted in an 80% accuracy in classification using kNN classification given by SciKit Python within a total of 60 thousand training iterations on a relatively simple two layer convolutional neural network. Since 80% is a reasonably high success rate for these conditions, it is expected that more exploration in this area can lead to even better results with slightly more complex models and better hyper parameter selection.

V. SUMMARY AND FUTURE WORK

We provide a method to directly learn an embedding space of EEG signals in a Euclidean space for recognition and clustering. This method is useful in the field because it can be used to determine the causes of certain ailments as opposed to their symptoms. Furthermore, this type of embedding space can be queried when needed to determine certain similarities of a certain patients EEG data with the data of another person's EEG data. Doing this will help recognize treatments that may be more effective than the ones currently being used in medicine. Our end-to-end system can be used for all of the above purposes.

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 classify the sixth label as a new type of label at inference time. '

It would also be wise to attempt to incorporate a recurrent relationship between the neurons in the network in order for a better prediction of the embedding. After all, the data that we are dealing with is time dependent and recurrent neural networks are

top of the line for that even if they can be unstable at times.

Future work can be done to use this system in other types of raw medical data, such as MRIs. Medical researchers can attempt to use a triplet loss clustering network to cluster these images in order to recognize deeper meaning while using a (relatively) low dimensional embedding space.

ACKNOWLEDGMENT

We would like to thank Christopher Curro and Professor Sam Keene at the Cooper Union for encouraging us int his direction of deep learning.

REFERENCES

[1] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," *CoRR*, vol. abs/1503.03832, 2015. [Online]. Available: http://arxiv.org/abs/1503.03832

[2] H. O. Song, S. Jegelka, V. Rathod, and K. Murphy, "Learnable structured clustering framework for deep metric learning," *CoRR*, vol. abs/1612.01213, 2016. [Online]. Available: http://arxiv.org/abs/1612.01213

[3] J. Yang, D. Parikh, and D. Batra, "Joint unsupervised learning of deep representations and image clusters," *CoRR*, vol. abs/1604.03628, 2016. [Online]. Available: http://arxiv.org/abs/1604.03628

[4] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," *J. Mach. Learn. Res.*, vol. 10, pp. 207–244, Jun. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1577069.1577078

[5] 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," *arXiv preprint arXiv:1603.04467*, 2016.

BrainNet Code

Krishna Thiyagarajan, Zhengqi Xi

May 10, 2017

```
import os

import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow.contrib.slim as slim
import random
import numpy as np
from sklearn import neighbors

saver = None
sess = None

loss_mem = []

#Garbage to make it pull properly


bckg_num = 0
artf_num = 1
eybl_num = 2
gped_num = 3
spsw_num = 4
pled_num = 5


def get_loss(loss_mem):
    print(loss_mem)
    plt.figure(1)
    plt.subplot(211)
    plt.plot(loss_mem, 'r--')
    plt.xlabel("100 Iterations")
    plt.ylabel("Average Loss in 100 Iterations")
    plt.title("Iterations vs. Average Loss")
    plt.show()


def save_model(sess, saver):
    save_path = saver.save(sess, "./latest_weights.ckpt")
    print("Model saved in file: %s" % save_path)


class BrainNet:
```

```
44      def __init__(self, sess, input_shape=[None, 71, 125],
        num_output=64, num_classes=6, restore_dir=None):
45
46          path = os.path.abspath('/media/krishna/My Passport/
        DataForUsage/labeled')
47
48
49          self.ARTF = [...]
50          self.ARTF_VAL = self.ARTF[:len(self.ARTF)/2]
51          self.ARTF = self.ARTF[len(self.ARTF)/2:]
52
53          self.BCKG = [...]
54          self.BCKG_VAL = self.ARTF[:len(self.BCKG)/2]
55          self.BCKG = self.ARTF[len(self.BCKG)/2:]
56
57          self.SPSW = [...]
58          self.SPSW_VAL = [...]
59
60          self.PLED = [...]
61          self.PLED_VAL = [...]
62
63          self.GPED = [...]
64          self.GPED_VAL = [...]
65
66          self.EYBL = [...]
67          self.EYBL_VAL = [...]
68
69          self.sess = sess
70          self.num_classes = num_classes
71          self.num_output = num_output
72          self.input_shape = input_shape
73          self.inference_input = tf.placeholder(tf.float32, shape=
        input_shape)
74          self.inference_model = self.get_model(self.inference_input,
         reuse=False)
75          if restore_dir is not None:
76              dir = tf.train.Saver()
77              dir.restore(self.sess, restore_dir)
78
79          print(len(self.ARTF))
80          print(len(self.BCKG))
81          print(len(self.EYBL))
82          print(len(self.SPSW))
83          print(len(self.PLED))
84          print(len(self.GPED))
85          self.load_files()
86
87      def triplet_loss(self, alpha):
88          self.anchor = tf.placeholder(tf.float32, shape=self.
        input_shape)
89          self.positive = tf.placeholder(tf.float32, shape=self.
        input_shape)
90          self.negative = tf.placeholder(tf.float32, shape=self.
        input_shape)
91          self.anchor_out = self.get_model(self.anchor, reuse=True)
92          self.positive_out = self.get_model(self.positive, reuse=
        True)
```

```
93          self.negative_out = self.get_model(self.negative, reuse=
        True)
94          with tf.variable_scope('triplet_loss'):
95              pos_dist = tf.reduce_sum(tf.square(self.anchor_out -
        self.positive_out))
96              neg_dist = tf.reduce_sum(tf.square(self.anchor_out -
        self.negative_out))
97
98              basic_loss = tf.maximum(0., alpha + pos_dist - neg_dist
        )
99              loss = tf.reduce_mean(basic_loss)
100             return loss
101
102     def load_files(self, validate=False):
103         print("Loading New Source Files...")
104         if not validate:
105             self.bckg = np.load(random.choice(self.BCKG))['arr_0']
106             self.eybl = np.load(random.choice(self.EYBL))['arr_0']
107             self.artf = np.load(random.choice(self.ARTF))['arr_0']
108             self.gped = np.load(random.choice(self.GPED))['arr_0']
109             self.pled = np.load(random.choice(self.PLED))['arr_0']
110             self.spsw = np.load(random.choice(self.SPSW))['arr_0']
111         else:
112             self.bckg = np.load(random.choice(self.BCKG_VAL))['
        arr_0']
113             self.eybl = np.load(random.choice(self.EYBL_VAL))['
        arr_0']
114             self.artf = np.load(random.choice(self.ARTF_VAL))['
        arr_0']
115             self.gped = np.load(random.choice(self.GPED_VAL))['
        arr_0']
116             self.pled = np.load(random.choice(self.PLED_VAL))['
        arr_0']
117             self.spsw = np.load(random.choice(self.SPSW_VAL))['
        arr_0']
118
119     def get_triplets(self):
120
121         choices = ['bckg', 'eybl', 'gped', 'spsw', 'pled', 'artf']
122         neg_choices = choices
123
124         choice = random.choice(choices)
125
126         if choice in neg_choices: neg_choices.remove(choice)
127
128         if choice == 'bckg':
129             ii = random.randint(0, len(self.bckg) - 1)
130             a = self.bckg[ii]
131
132             jj = random.randint(0, len(self.bckg) - 1)
133             p = self.bckg[jj]
134
135         elif choice == 'eybl':
136             ii = random.randint(0, len(self.eybl) - 1)
137             a = self.eybl[ii]
138
139             jj = random.randint(0, len(self.eybl) - 1)
```

```
            p = self.eybl[jj]

        elif choice == 'gped':
            ii = random.randint(0, len(self.gped) - 1)
            a = self.gped[ii]

            jj = random.randint(0, len(self.gped) - 1)
            p = self.gped[jj]

        elif choice == 'spsw':
            ii = random.randint(0, len(self.spsw) - 1)
            a = self.spsw[ii]

            jj = random.randint(0, len(self.spsw) - 1)
            p = self.spsw[jj]

        elif choice == 'pled':
            ii = random.randint(0, len(self.pled) - 1)
            a = self.pled[ii]

            jj = random.randint(0, len(self.pled) - 1)
            p = self.pled[jj]

        else:
            ii = random.randint(0, len(self.artf) - 1)
            a = self.artf[ii]

            jj = random.randint(0, len(self.artf) - 1)
            p = self.artf[jj]

        neg_choice = random.choice(neg_choices)

        if neg_choice == 'bckg':
            ii = random.randint(0, len(self.bckg) - 1)
            n = self.bckg[ii]
        elif neg_choice == 'eybl':
            ii = random.randint(0, len(self.eybl) - 1)
            n = self.eybl[ii]
        elif neg_choice == 'gped':
            ii = random.randint(0, len(self.gped) - 1)
            n = self.gped[ii]
        elif neg_choice == 'spsw':
            ii = random.randint(0, len(self.spsw) - 1)
            n = self.spsw[ii]
        elif neg_choice == 'pled':
            ii = random.randint(0, len(self.pled) - 1)
            n = self.pled[ii]
        else:
            ii = random.randint(0, len(self.artf) - 1)
            n = self.artf[ii]

        a = np.expand_dims(a, 0) * 10e4
        p = np.expand_dims(p, 0) * 10e4
        n = np.expand_dims(n, 0) * 10e4

        return np.vstack([a, p, n])

```

```
197     def get_model(self, input, reuse=False):
198         with slim.arg_scope([slim.layers.conv2d, slim.layers.
        fully_connected],
199                             weights_initializer=tf.contrib.layers.
        xavier_initializer(seed=random.random(),
200
                        uniform=True),
201                             weights_regularizer=slim.l2_regularizer
        (0.05), reuse=reuse):
202             net = tf.expand_dims(input, axis=3)
203             net = slim.layers.conv2d(net, num_outputs=32,
        kernel_size=4, scope='conv1', trainable=True)
204             net = slim.layers.max_pool2d(net, kernel_size=3, scope=
        'maxpool1')
205             net = slim.layers.conv2d(net, num_outputs=64,
        kernel_size=5, scope='conv2', trainable=True)
206             net = slim.layers.max_pool2d(net, kernel_size=3, scope=
        'maxpool2')
207             net = slim.layers.flatten(net, scope='flatten')
208             net = slim.layers.fully_connected(net, 256, scope='fc1'
        , trainable=True)
209             net = slim.layers.fully_connected(net, 1024, scope='fc2
        ', trainable=True)
210             net = slim.layers.fully_connected(net, self.num_output,
         activation_fn=None, weights_regularizer=None,
211                                               scope='output')
212
213             return net
214
215     def train_model(self, learning_rate, keep_prob, batch_size,
        train_epoch, outdir=None):
216         loss = self.triplet_loss(alpha=0.5)
217         self.optimizer = tf.train.AdamOptimizer(learning_rate=
        learning_rate)
218         self.optim = self.optimizer.minimize(loss=loss)
219         self.sess.run(tf.global_variables_initializer())
220
221         count = 0
222         ii = 0
223
224         for epoch in range(0, train_epoch):
225             ii = 0
226             count = 0
227             full_loss = 0
228             while ii <= batch_size:
229                 ii += 1
230                 feeder = self.get_triplets()
231
232                 anchor = feeder[0]
233                 anchor = np.expand_dims(anchor, 0)
234                 positive = feeder[1]
235                 positive = np.expand_dims(positive, 0)
236                 negative = feeder[2]
237                 negative = np.expand_dims(negative, 0)
238
239                 temploss = self.sess.run(loss, feed_dict={self.
        anchor: anchor, self.positive: positive,
```

```
240                                                                 self.
    negative: negative})
241
242             if temploss == 0:
243                 print(temploss)
244                 ii -= 1
245                 count += 1
246                 continue
247
248             full_loss += temploss
249
250             if ii % 100 == 0:
251                 loss_mem.append(full_loss /(100+ count))
252                 full_loss = 0
253
254             _, anchor, positive, negative = self.sess.run([self
    .optim,
255                                                            self
    .anchor_out,
256                                                            self
    .positive_out,
257                                                            self
    .negative_out], feed_dict={self.anchor: anchor,
258
                                self.positive: positive,
259
                                self.negative: negative})
260
261             d1 = np.linalg.norm(positive - anchor)
262             d2 = np.linalg.norm(negative - anchor)
263
264             print("Epoch: ", epoch, "Iteration:", ii, ", Loss:
    ", temploss, ", Positive Diff: ", d1,
265                   ", Negative diff: ", d2)
266             print("Iterations skipped: ", count)
267         self.validate()
268         self.load_files()
269
270     def get_sample(self, size = 1):
271         data_list=[]
272         class_list=[]
273
274         for ii in range(0, size):
275             choice = random.choice(['bckg', 'eybl', 'gped', 'spsw',
     'pled', 'artf'])
276
277             if choice == 'bckg':
278                 ii = random.randint(0, len(self.bckg) - 1)
279                 data_list.append(self.bckg[ii])
280                 class_list.append(bckg_num)
281
282             elif choice=='eybl':
283                 ii = random.randint(0, len(self.eybl) - 1)
284                 data_list.append(self.eybl[ii])
285                 class_list.append(eybl_num)
286             elif choice =='gped':
287                 ii = random.randint(0, len(self.gped) - 1)
```

```
288                 data_list.append(self.gped[ii])
289                 class_list.append(gped_num)
290             elif choice == 'spsw':
291                 ii = random.randint(0, len(self.spsw) -1)
292                 data_list.append(self.spsw[ii])
293                 class_list.append(spsw_num)
294             elif choice=='pled':
295                 ii = random.randint(0, len(self.spsw) - 1)
296                 data_list.append(self.pled[ii])
297                 class_list.append(pled_num)
298             else:
299                 ii = random.randint(0, len(self.artf) -1)
300                 data_list.append(self.artf[ii])
301                 class_list.append(artf_num)
302
303
304         return data_list, class_list
305
306     def validate(self):
307         self.load_files(True)
308
309         for _ in range(0, 5):
310             inputs, classes = self.get_sample(size=1000)
311
312             vector_inputs = self.sess.run(self.inference_model,
    feed_dict={self.inference_input: inputs})
313
314             knn = neighbors.KNeighborsClassifier()
315             knn.fit(vector_inputs, classes)
316
317             val_inputs, val_classes = self.get_sample(size=100)
318
319             vector_val_inputs = self.sess.run(self.inference_model,
     feed_dict={self.inference_input: val_inputs})
320
321             pred_class = knn.predict(vector_val_inputs)
322
323             percentage = len([i for i, j in zip(val_classes,
    pred_class) if i==j])
324
325             print("Validation Results: %d out of 100 correct" %
    percentage )
326
327
328
329 if __name__ == "__main__":
330     try:
331         sess = tf.Session()
332         model = BrainNet(sess=sess, restore_dir='previous_run/
    latest_weights.ckpt')
333         model.validate()
334     except (KeyboardInterrupt, SystemError, SystemExit):
335         save_model(sess, saver)
336         get_loss(loss_mem)
337
338 #
339 # sess = tf.Session()
```

```
340 # model = BrainNet(sess=sess, restore_dir='./latest_weights.ckpt')
```

Listing 1: BrainNet Code