Lab2 EEGNet

0856029 Yu-Shao Liu

April 12, 2020

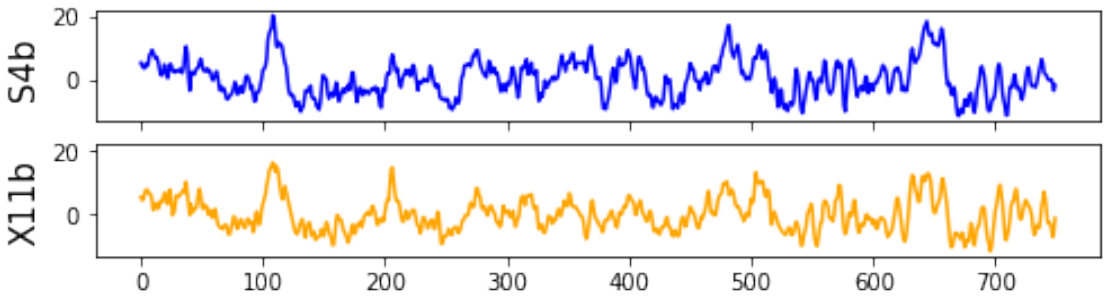
1 Introduction

This lab is aim at building a light-weighted CNN model called EEGNet, which is designed for classification problem on EEG. EEGNet is inspired by Xception and uses depthwise separable convolution instead of 3x3 convolution filter to capture features and to decrease parameters. EEGNet is still the state-of-the-art in EEG classification problem. We have presented this work at CECNL Lab. See more details at: Link

1.1 EEG paradigm

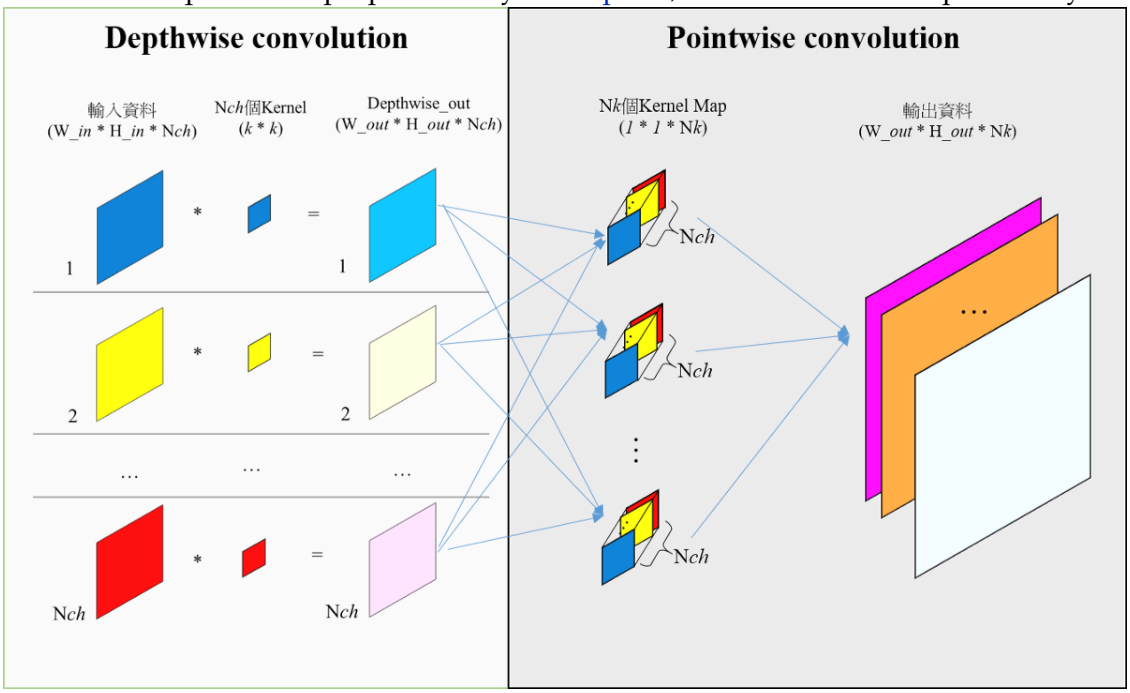
Both training and testing data are extracted from BCI Competition III - IIIb Cued motor imagery with online feedback (nonstationary classifier) with 2 classes (left hand, right hand) from 3 subjects [2 classes, 2 bipolar EEG channels] Ref

- The processed data including 2 channels, each including 1080 records and 750 timestamps.
- Take one record as example:



1.2 Depthwise Separable Convolution

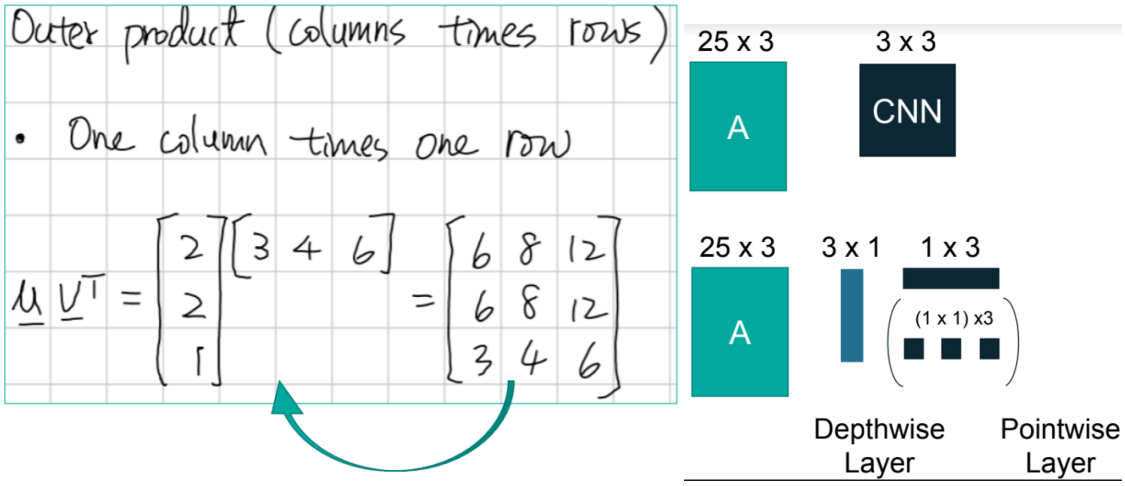
This technique is proposed by Xception, which is inspired by Inception.



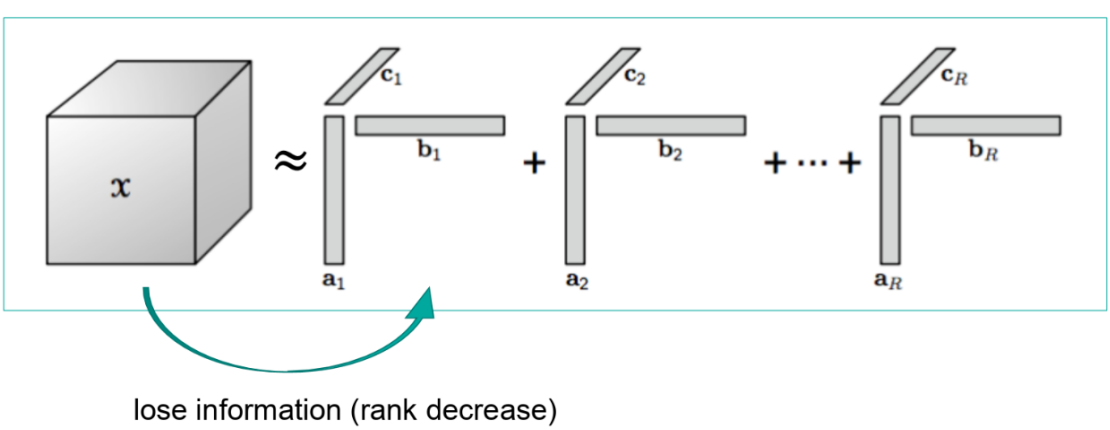
1.2.1 Low rank approximation

We could also see Depthwise Separable convolution in Linear Algebra or in Low rank approximation aspects.

Rank-1 Matrix Decomposition Considering a n by n convolution filter as a matrix, and we could apply rank-1 matrix decomposition on a matrix and get the sum of outer products of n by 1 vectors. These n by 1 vectors could be regarded as depthwise and pointwise convolution filters. Replacing the n by n convolution filters with these rank-1 vectors, we could decrease the parameters from n^2 to 2n. Obviously, this kind of parameter reduction might decrease the capability of models.



Rank-1 Tensor Decomposition We often use high dimensional convolution filters in Computer Vision, and we need tensors for representing those high dimensional filters. Rank-1 decomposition for tensor is an optimization problem, and we could take this as extension of singular value decomposition in high dimensional domain.



- Practically, some deep learning frameworks like keras have implemented **separable layers** which combines depthwise convolution and pointwise convolution together, which is a depthwise layer and immediately follows by a pointwise convolution layer. We would use "separable convolution" as a term later.
- Ref: https://medium.com/@chih.sheng.huang821/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92
  mobilenet-depthwise-separable-convolution-f1ed016b3467

2 Experimental Setup

We compare EEGNet with a Deep Convolution Network mentioned in EEGNet paper.

2.1 EEGNet architechture

- My EEGNet has the same neurons and layers with TA's EEGNet
  - Our architecture has different number of neurons with original paper, I guess TA took differenet some EEG data with different sample rate.

```
EEGNet(
  (firstConv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU6()
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.2, inplace=False)
  )
  (separableConv): Sequential(
    (0): Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0, 7), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU6()
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

2.2 DeepConvNet architechture

- C = 2, T = 750 and N = 2.

```
DeepConvNet(
  (conv0): Sequential(
    (0): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
    (1): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1))
    (2): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): ELU(alpha=1.0)
    (4): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Dropout(p=0.5, inplace=False)
  )
  (conv1): Sequential(
    (0): Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (conv2): Sequential(
    (0): Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (conv3): Sequential(
    (0): Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (classify): Sequential(
    (0): Linear(in_features=8600, out_features=2, bias=True)
  )
)
```

3 EEGNet Details

3.1 Activation Function

3.1.1 ELU

ELU is mentioned in many papers using depthwise separable convolution like Xception, MobileNetV2, and also EEGNet. Most of these works pointed out that ReLU might much more easily suffer from the "dead neuron" problem due to smaller sizes of depthwise and pointewise layers. They recommended that it is better to use ELU activation right after pointwise convolution.

However, in my experiments, ELU performed the worst in all activation functions. I guess the reason about this phenomenon is that our dataset is so small that EEGNet does not need to learn weights in sparse space to represent the information of whole training data. Without sparsity of weights, the dead neuron problem would not be too harmful.

3.1.2 ReLU

ReLU gives good enough results.

3.1.3 ReLU6

Some empirical results found that clipping mechanism for ReLU function could make training process much more stable and get better accuracy. Ref1 Ref2

3.1.4 Leaky ReLU

Not bad, but dead neurons problem didn't happen harmfully in our network so that Leaky ReLU did not outperform ReLU and ReLU6

3.2 Default Hyperparameters

- Batch size= 64
- Learning rate = 0.0025
- Epochs = 300
- Optimizer: Adam
- Loss function: torch.nn.CrossEntropyLoss()

4 Experimental results

4.1 Highest accuracy

- EEGNet (default dropout rate) (I tried different dropoute rates at discussion part)

	EEG_elu_train	EEG_relu_train	EEG_relu6_train	EEG_leaky_relu_train	EEG_elu_test	EEG_relu_test	EEG_relu6_test	EEG_leaky_relu_test
best_acc	99.814815	99.814815	99.537037	99.722222	87.12963	87.12963	87.314815	88.240741

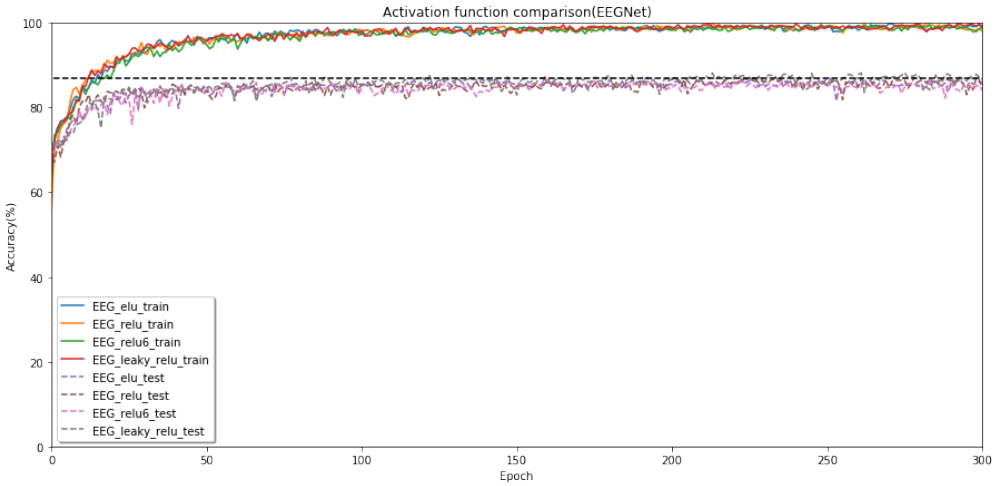
- DeepConvNet

	DCN_elu_train	DCN_relu_train	DCN_relu6_train	DCN_leaky_relu_train	DCN_elu_test	DCN_relu_test	DCN_relu6_test	DCN_leaky_relu_test
best_acc	98.055556	97.037037	96.851852	97.222222	81.851852	80.833333	81.111111	82.777778

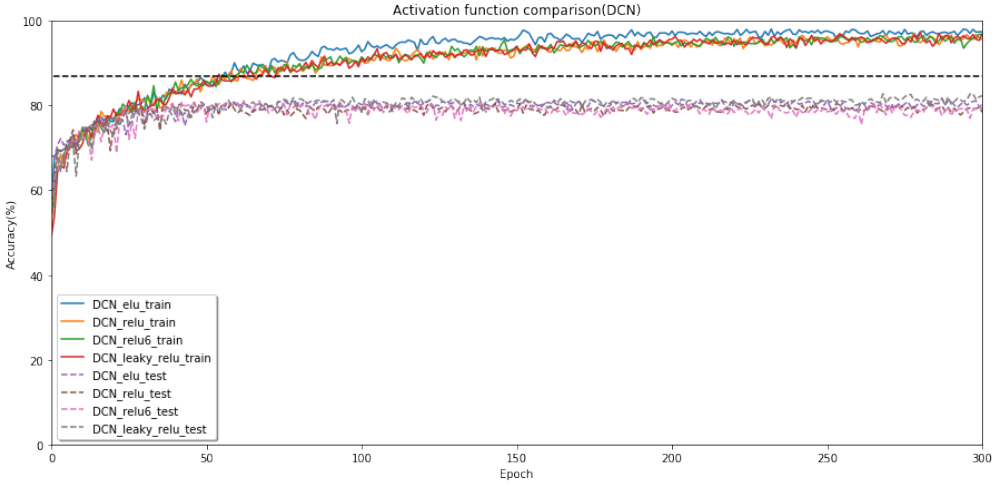
4.2 Result Comparison

There is a black dotted line at 87%

- EEGNet



- DeepConvNet



5 Discussion and extra experiments

5.1 Hyperparameter Tuning

Hyperparameter Tuning is only applied to EEGNet because EEGNet is much more pratical then the DeepConvNet, and there are less parameter in EEGNet so that this tuning process could be faster.

- Though dropout rates don't belong to hyperparameters, I tried to tune dropout rates together to see whether it could improve accuracy

5.1.1 Ray Tune

Ray Tune is a state-of-the-art distributed hyperparameter tuning framework, supporting many famous tuning algorithm including grid search, Population Based Training (PBT), and also supporting some early stopping algorithm like Asynchronous Successive Halving. Algorithm (ASHA)

- Using Early stopping algorithm often stops too early to get better accuracy, so I omit this.

```
== Status ==
Memory usage on this node: 31.0/31.0 GiB: ***LOW MEMORY*** less than 10% of the memory on this node is available for use. This can cause unexpect
ed crashes. Consider reducing the memory used by your application or reducing the Ray object store size by setting `object_store_memory` when cal
ling `ray.init`.
Using AsyncHyperBand: num_stopped=28
Bracket: Iter 64.000: 0.762962962962963 | Iter 16.000: 0.7171296296296297 | Iter 4.000: 0.712037037037037 | Iter 1.000: 0.6972222222222222
Resources requested: 3/12 CPUs, 1/2 GPUs, 0.0/5.62 GiB heap, 0.0/1.9 GiB objects
Result logdir: /home/ailurus/ray_results/TrainEEG
Number of trials: 120 (108 ERROR, 1 RUNNING, 11 TERMINATED)
+---------------+------------+-------+---------+---------+--------+----------+--------+------------------+
| Trial name    | status     | loc   |   drop1 |   drop2 |     lr |      acc |   iter |   total time (s) |
|---------------+------------+-------+---------+---------+--------+----------+--------+------------------|
| TrainEEG_00009 | ERROR      |       |    0.65 |    0.65 | 0.003  | 0.707407 |      1 |          1.44206 |
| TrainEEG_00010 | ERROR      |       |    0.6  |    0.68 | 0.003  |          |        |                  |
| TrainEEG_00011 | ERROR      |       |    0.65 |    0.68 | 0.003  | 0.697222 |      1 |          1.3442  |
| TrainEEG_00012 | ERROR      |       |    0.6  |    0.65 | 0.0022 |          |        |                  |
| TrainEEG_00013 | ERROR      |       |    0.65 |    0.65 | 0.0022 |          |        |                  |
| TrainEEG_00014 | ERROR      |       |    0.6  |    0.68 | 0.0022 |          |        |                  |
| TrainEEG_00015 | ERROR      |       |    0.65 |    0.68 | 0.0022 |          |        |                  |
| TrainEEG_00016 | ERROR      |       |    0.6  |    0.65 | 0.0025 |          |        |                  |
| TrainEEG_00017 | ERROR      |       |    0.65 |    0.65 | 0.0025 |          |        |                  |
| TrainEEG_00018 | ERROR      |       |    0.6  |    0.68 | 0.0025 |          |        |                  |
| TrainEEG_00119 | RUNNING    |       |    0.65 |    0.68 | 0.003  |          |        |                  |
| TrainEEG_00000 | TERMINATED |       |    0.6  |    0.65 | 0.0022 | 0.65463  |      1 |          1.69158 |
| TrainEEG_00001 | TERMINATED |       |    0.65 |    0.65 | 0.0022 | 0.781481 |    100 |          6.93045 |
| TrainEEG_00002 | TERMINATED |       |    0.6  |    0.68 | 0.0022 | 0.700926 |      4 |          1.58215 |
| TrainEEG_00003 | TERMINATED |       |    0.65 |    0.68 | 0.0022 | 0.681481 |      1 |          1.57006 |
| TrainEEG_00004 | TERMINATED |       |    0.6  |    0.65 | 0.0025 | 0.664815 |      1 |          1.52389 |
| TrainEEG_00005 | TERMINATED |       |    0.65 |    0.65 | 0.0025 | 0.706481 |      4 |          1.58193 |
| TrainEEG_00006 | TERMINATED |       |    0.6  |    0.68 | 0.0025 | 0.667593 |      1 |          1.60679 |
| TrainEEG_00007 | TERMINATED |       |    0.65 |    0.68 | 0.0025 | 0.700926 |      4 |          1.70325 |
| TrainEEG_00008 | TERMINATED |       |    0.6  |    0.65 | 0.003  | 0.691667 |      4 |          1.73266 |
| TrainEEG_00046 | TERMINATED |       |    0.6  |    0.68 | 0.003  | 0.686111 |      1 |          1.29691 |
+---------------+------------+-------+---------+---------+--------+----------+--------+------------------+
```

- Ray Tune running Example

```
== Status ==
Memory usage on this node: 4.5/88.5 GiB
Using FIFO scheduling algorithm.
Resources requested: 3/4 CPUs, 1/1 GPUs, 0.0/52.83 GiB heap, 0.0/18.21 GiB objects
Result logdir: /home/ubuntu/ray_results/TrainEEG
Number of trials: 800 (1 RUNNING, 799 TERMINATED)
+---------------+------------+-----------------+--------------+---------+---------+--------+----------+--------+-----------------+
| Trial name    | status     | loc             |   activation |   drop1 |   drop2 |     lr |      acc |   iter |   total time (s) |
|---------------+------------+-----------------+--------------+---------+---------+--------+----------+--------+-----------------|
| TrainEEG_00799 | RUNNING    | 10.54.87.6:3614 |            3 |    0.65 |    0.68 | 0.0039 | 0.836111 |    256 |         12.6855  |
| TrainEEG_00000 | TERMINATED |                 |            0 |    0.6  |    0.65 | 0.0022 | 0.815741 |    280 |         13.4511  |
| TrainEEG_00001 | TERMINATED |                 |            1 |    0.6  |    0.65 | 0.0022 | 0.860185 |    280 |         13.4434  |
| TrainEEG_00002 | TERMINATED |                 |            2 |    0.6  |    0.65 | 0.0022 | 0.873148 |     85 |          5.22564 |
| TrainEEG_00003 | TERMINATED |                 |            3 |    0.6  |    0.65 | 0.0022 | 0.855556 |    280 |         14.1445  |
| TrainEEG_00004 | TERMINATED |                 |            0 |    0.65 |    0.65 | 0.0022 | 0.828704 |    280 |         13.9425  |
| TrainEEG_00005 | TERMINATED |                 |            1 |    0.65 |    0.65 | 0.0022 | 0.872222 |    260 |         13.3257  |
| TrainEEG_00006 | TERMINATED |                 |            2 |    0.65 |    0.65 | 0.0022 | 0.872222 |    209 |         10.7938  |
| TrainEEG_00007 | TERMINATED |                 |            3 |    0.65 |    0.65 | 0.0022 | 0.844444 |    280 |         13.7365  |
| TrainEEG_00008 | TERMINATED |                 |            0 |    0.6  |    0.68 | 0.0022 | 0.799074 |    280 |         13.9066  |
| TrainEEG_00009 | TERMINATED |                 |            1 |    0.6  |    0.68 | 0.0022 | 0.862963 |    280 |         14.7089  |
| TrainEEG_00010 | TERMINATED |                 |            2 |    0.6  |    0.68 | 0.0022 | 0.846296 |    280 |         13.725   |
| TrainEEG_00011 | TERMINATED |                 |            3 |    0.6  |    0.68 | 0.0022 | 0.843519 |    280 |         13.4502  |
| TrainEEG_00012 | TERMINATED |                 |            0 |    0.65 |    0.68 | 0.0022 | 0.84537  |    280 |         13.8676  |
| TrainEEG_00013 | TERMINATED |                 |            1 |    0.65 |    0.68 | 0.0022 | 0.871296 |    264 |         12.878   |
| TrainEEG_00014 | TERMINATED |                 |            2 |    0.65 |    0.68 | 0.0022 | 0.856481 |    280 |         13.6226  |
| TrainEEG_00015 | TERMINATED |                 |            3 |    0.65 |    0.68 | 0.0022 | 0.851852 |    280 |         14.0215  |
| TrainEEG_00016 | TERMINATED |                 |            0 |    0.6  |    0.65 | 0.0025 | 0.814815 |    280 |         13.5289  |
| TrainEEG_00017 | TERMINATED |                 |            1 |    0.6  |    0.65 | 0.0025 | 0.87037  |    174 |          8.819   |
| TrainEEG_00018 | TERMINATED |                 |            2 |    0.6  |    0.65 | 0.0025 | 0.859259 |    280 |         13.3686  |
+---------------+------------+-----------------+--------------+---------+---------+--------+----------+--------+-----------------+
... 780 more trials not shown (780 TERMINATED)
```

- Ray Tune result Example

```
| TrainEEG_00770 | TERMINATED |       |            2 |    0.6  |    0.65 | 0.0035 | 0.857407 |    280 |         13.8584  |
| TrainEEG_00771 | TERMINATED |       |            3 |    0.6  |    0.65 | 0.0035 | 0.842593 |    280 |         13.5983  |
| TrainEEG_00772 | TERMINATED |       |            0 |    0.65 |    0.65 | 0.0035 | 0.806481 |    280 |         13.6223  |
| TrainEEG_00773 | TERMINATED |       |            1 |    0.65 |    0.65 | 0.0035 | 0.857407 |    280 |         13.6689  |
| TrainEEG_00774 | TERMINATED |       |            2 |    0.65 |    0.65 | 0.0035 | 0.873148 |    174 |          9.20229 |
| TrainEEG_00775 | TERMINATED |       |            3 |    0.65 |    0.65 | 0.0035 | 0.833333 |    280 |         13.327   |
| TrainEEG_00776 | TERMINATED |       |            0 |    0.6  |    0.68 | 0.0035 | 0.824074 |    280 |         13.782   |
| TrainEEG_00777 | TERMINATED |       |            1 |    0.6  |    0.68 | 0.0035 | 0.85463  |    280 |         13.4194  |
| TrainEEG_00778 | TERMINATED |       |            2 |    0.6  |    0.68 | 0.0035 | 0.857407 |    280 |         13.4881  |
| TrainEEG_00779 | TERMINATED |       |            3 |    0.6  |    0.68 | 0.0035 | 0.852778 |    280 |         13.1828  |
| TrainEEG_00780 | TERMINATED |       |            0 |    0.65 |    0.68 | 0.0035 | 0.784259 |    280 |         13.4409  |
| TrainEEG_00781 | TERMINATED |       |            1 |    0.65 |    0.68 | 0.0035 | 0.85     |    280 |         13.4271  |
| TrainEEG_00782 | TERMINATED |       |            2 |    0.65 |    0.68 | 0.0035 | 0.853704 |    280 |         14.3059  |
| TrainEEG_00783 | TERMINATED |       |            3 |    0.65 |    0.68 | 0.0035 | 0.863889 |    280 |         13.6275  |
| TrainEEG_00784 | TERMINATED |       |            0 |    0.6  |    0.65 | 0.0039 | 0.807407 |    280 |         14.0187  |
| TrainEEG_00785 | TERMINATED |       |            1 |    0.6  |    0.65 | 0.0039 | 0.853704 |    280 |         13.3928  |
| TrainEEG_00786 | TERMINATED |       |            2 |    0.6  |    0.65 | 0.0039 | 0.844444 |    280 |         13.2157  |
| TrainEEG_00787 | TERMINATED |       |            3 |    0.6  |    0.65 | 0.0039 | 0.864815 |    280 |         13.6354  |
| TrainEEG_00788 | TERMINATED |       |            0 |    0.65 |    0.65 | 0.0039 | 0.799074 |    280 |         13.3651  |
| TrainEEG_00789 | TERMINATED |       |            1 |    0.65 |    0.65 | 0.0039 | 0.828704 |    280 |         13.4807  |
| TrainEEG_00790 | TERMINATED |       |            2 |    0.65 |    0.65 | 0.0039 | 0.863889 |    280 |         13.7216  |
| TrainEEG_00791 | TERMINATED |       |            3 |    0.65 |    0.65 | 0.0039 | 0.872222 |    164 |          8.50717 |
| TrainEEG_00792 | TERMINATED |       |            0 |    0.6  |    0.68 | 0.0039 | 0.816667 |    280 |         13.4541  |
| TrainEEG_00793 | TERMINATED |       |            1 |    0.6  |    0.68 | 0.0039 | 0.87037  |    209 |         10.4741  |
| TrainEEG_00794 | TERMINATED |       |            2 |    0.6  |    0.68 | 0.0039 | 0.873148 |    227 |         11.1149  |
| TrainEEG_00795 | TERMINATED |       |            3 |    0.6  |    0.68 | 0.0039 | 0.871296 |    204 |         10.102   |
| TrainEEG_00796 | TERMINATED |       |            0 |    0.65 |    0.68 | 0.0039 | 0.816667 |    280 |         13.4023  |
| TrainEEG_00797 | TERMINATED |       |            1 |    0.65 |    0.68 | 0.0039 | 0.848148 |    280 |         13.5914  |
| TrainEEG_00798 | TERMINATED |       |            2 |    0.65 |    0.68 | 0.0039 | 0.866667 |    280 |         13.3319  |
| TrainEEG_00799 | TERMINATED |       |            3 |    0.65 |    0.68 | 0.0039 | 0.84537  |    280 |         13.8336  |
+---------------+------------+-------+--------------+---------+---------+--------+----------+--------+-----------------+


Best config is: {'activation': 1, 'args': Namespace(ray_address=None, smoke_test=False, use_gpu=True), 'drop1': 0.6, 'drop2': 0.65, 'lr': 0.0025}
```

5.2 Tuning Results

5.2.1 Adam with weight decay

Due to the learning rate is set to be 0.0028, weight decay should be less than learning rate to improve optimization process.

```
colname = ['lr=.0028', 'lr=.0028, wd=.01', 'lr=.0028, wd=.0001']
max_list = [df8.max(), df8_01.max(), df8_0001.max()]
pd.DataFrame(max_list, index=colname).T.iloc[4:]
```

	lr=.0028	lr=.0028, wd=.01	lr=.0028, wd=.0001
EEG_elu_test	83.796296	71.944444	85.555556
EEG_relu_test	87.962963	81.481481	89.259259
EEG_relu6_test	88.055556	73.888889	88.333333
EEG_leaky_relu_test	87.777778	72.314815	88.425926

5.2.2 Best hyperparameters

By Ray Tune, we get best learning rate between [0.0022 to 0.0028]. I apply the weight decay rate to be 0.001 to see whether it could be much better.

Surprisingly, ELU performs usually bad, while ReLU6 often outperform ReLU.

```
import pandas as pd
df2 = pd.read_csv('eeg_0022.csv', index_col=0)
df5 = pd.read_csv('eeg_0025.csv', index_col=0)
df8 = pd.read_csv('eeg_0028.csv', index_col=0)
df8_01 = pd.read_csv('eeg_0028_wd_01.csv', ir
df8_0001 = pd.read_csv('eeg_0028_wd_0001.cs
```

```
colname = ['lr=.0022','lr=.0025', 'lr=.0028']#, 'lr
max_list = [df2.max(), df5.max(), df8.max()]#, d
pd.DataFrame(max_list, index=colname).T.iloc
```

	lr=.0022	lr=.0025	lr=.0028
EEG_elu_test	85.555556	85.185185	83.796296
EEG_relu_test	86.759259	88.148148	87.962963
EEG_relu6_test	88.703704	89.259259	88.055556
EEG_leaky_relu_test	88.240741	87.685185	87.777778

```
colname = ['lr=.0025', 'lr=.0025, wd=.0001']
max_list = [df5.max(), df5_0001.max()]
pd.DataFrame(max_list, index=colname).T.iloc[
```

	lr=.0025	lr=.0025, wd=.0001
EEG_elu_test	85.185185	84.259259
EEG_relu_test	88.148148	86.574074
EEG_relu6_test	89.259259	88.611111
EEG_leaky_relu_test	87.685185	89.166667

5.3 Conclusion

EEGNet is an excellent network with efficient training time and good enough performance. Actually, the first version of EEGNet posted on arxiv at 2016 does not include Depthwise Separable Convolution. It is until the version3 and version4 at 2018 that EEGNet take some experiences from MobileNet and Xception so that this work become better. In EEG classification problem, or multi-channel time series classification, EEGNet is still state-of-the-art. However, I think EEGNet could be extended more, but maybe there are less researcher in BCI discipline working with it. Maybe some related works of Stock Market prediction, or some tricks in MobileNetV3 could apply on EEGNet to improve it. I would try this in my final project.

6 Code

- For recording accuracy, we provide a `train` function and a `test` function in `main.py`
- In tuning process, it's not necessary to record accuracy of each epoch so that we provide simpler `train` function and `test` function in `EEGNet.py`

6.1 main.py

[ ]:
```
from functools import reduce

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim
import pandas as pd
from torch.cuda import device
from torch.utils.data import TensorDataset, DataLoader

from dataloader import read_bci_data
from EEGNet import EEGNet
from DCN2 import DeepConvNet

def get_bci_dataloaders():
    train_x, train_y, test_x, test_y = read_bci_data()
    datasets = []
    for train, test in [(train_x, train_y), (test_x, test_y)]:
        train = torch.stack(
            [torch.Tensor(train[i]) for i in range(train.shape[0])]
        )
        test = torch.stack(
            [torch.Tensor(test[i:i+1]) for i in range(test.shape[0])]
        )
        datasets += [TensorDataset(train, test)]

    return datasets

def get_data_loaders(train_dataset, test_dataset):
    #train_dataset, test_dataset = get_bci_dataloaders()
    kwargs = {'num_workers': 1, 'pin_memory': True} if torch.cuda.is_available()␣
 ↪else {}
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=len(test_dataset),␣
 ↪shuffle=True)
    return train_loader, test_loader


def showResult(title='', **kwargs):
```

```
    plt.figure(figsize=(15, 7))
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy(%)')

    for label, data in kwargs.items():
        plt.plot(range(len(data)), data, '--' if 'test' in label else '-',␣
 ↪label=label)
    plt.ylim(0, 100)
    plt.xlim(0, 300)
    points = [(-5, 87), (310, 87)]
    (xpoints, ypoints) = zip(*points)

    plt.plot(xpoints, ypoints, linestyle='--', color='black')

    plt.legend(loc='best', fancybox=True, shadow=True)
    plt.show()



def main():
    torch.backends.cudnn.enabled = True
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    nets1 = {
        "EEG_elu": EEGNet(nn.ELU).to(device),
        "EEG_relu": EEGNet(nn.ReLU).to(device),
        "EEG_relu6": EEGNet(nn.ReLU6).to(device),
        "EEG_leaky_relu": EEGNet(nn.LeakyReLU).to(device)
    }

    nets2 = {
            "DCN_elu": DeepConvNet(nn.ELU).to(device),
            "DCN_relu": DeepConvNet(nn.ReLU).to(device),
            "DCN_relu6": DeepConvNet(nn.ReLU6).to(device),
            "DCN_leaky_relu": DeepConvNet(nn.LeakyReLU).to(device)
    }

    #nets = nets1
    nets = nets2

    # Training setting
    loss_fn = nn.CrossEntropyLoss()
    learning_rates = {0.0025}

    optimizer = torch.optim.Adam
    optimizers = {
```

```
        key: optimizer(value.parameters(), lr=learning_rate, weight_decay=0.0001)
        for key, value in nets.items()
        for learning_rate in learning_rates
    }

    epoch_size = 300
    batch_size = 64
    acc = train(nets, epoch_size, batch_size, loss_fn, optimizers)
    df = pd.DataFrame.from_dict(acc)

    #df.to_csv('eeg_0025_0001.csv')
    #print(df)
    display(df)
    return df

# This train is for demo and recording accuracy
def train(nets, epoch_size, batch_size, loss_fn, optimizers):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    trainDataset, testDataset = get_bci_dataloaders()
    trainLoader, testLoader = get_data_loaders(trainDataset, testDataset)

    accuracy = {
        **{key + "_train": [] for key in nets},
        **{key + "_test": [] for key in nets}
    }
    for epoch in range(epoch_size + 1):
        train_correct = {key: 0.0 for key in nets}
        test_correct = {key: 0.0 for key in nets}
        for step, (x, y) in enumerate(trainLoader):
            x = x.to(device)
            y = y.to(device).long().view(-1)

            for key, net in nets.items():
                net.train(mode=True)
                y_hat = net(x)
                loss = loss_fn(y_hat, y)
                loss.backward()
                train_correct[key] += (torch.max(y_hat, 1)[1] == y).sum().item()

            for optimizer in optimizers.values():
                optimizer.step()
                optimizer.zero_grad()

        with torch.no_grad():
            for step, (x, y) in enumerate(testLoader):
                x = x.to(device)
```

```
                y = y.to(device).long().view(-1)
                for key, net in nets.items():
                    net.eval()
                    y_hat = net(x)
                    test_correct[key] += (torch.max(y_hat, 1)[1] == y).sum().
↪item()

        for key, value in train_correct.items():
            accuracy[key + "_train"] += [(value * 100.0) / len(trainDataset)]

        for key, value in test_correct.items():
            accuracy[key + "_test"] += [(value * 100.0) / len(testDataset)]

        if epoch % 100 == 0:
            print('epoch : ', epoch, ' loss : ', loss.item())
            display(pd.DataFrame.from_dict(accuracy).iloc[[epoch]])
            print('')
        torch.cuda.empty_cache()
    showResult(title='Activation function comparison(EEGNet)'.format(epoch + 1),␣
↪**accuracy)
    #showResult(title='Activation function comparison(DCN)'.format(epoch + 1),␣
↪**accuracy)
    return accuracy

if __name__ == '__main__':
    df1 = main()
    df1.max()
    display(pd.DataFrame(df1.max(), columns=['best_acc']).T)
```

6.2 EEGNet.py

[ ]:
```
import torch
import torch.nn as nn
import torch.optim
import torch.nn.functional as F
from torch.cuda import device
from torch.utils.data import DataLoader

EPOCH_SIZE = 512
TEST_SIZE = 256

class EEGNet(nn.Module):
    def __init__(self, activation=None, dropout1=0.25, dropout2=0.25):
        if not activation:
            activation = nn.ELU
        super(EEGNet, self).__init__()
```

```
        self.firstConv = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0,␣
↪25), bias=False),
            nn.BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,␣
↪track_running_stats=True)
        )

        self.depthwiseConv = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16,␣
↪bias=False),
            nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,␣
↪track_running_stats=True),
            activation(),
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
            nn.Dropout(p=dropout1)
        )

        self.separableConv = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0,␣
↪7), bias=False),
            nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,␣
↪track_running_stats=True),
            activation(),
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
            nn.Dropout(p=dropout2)
        )

        self.classify = nn.Sequential(
            nn.Linear(in_features=736, out_features=2, bias=True)
        )

    def forward(self, x):
        x = self.firstConv(x)
        x = self.depthwiseConv(x)
        x = self.separableConv(x)
        x = x.view(-1, self.classify[0].in_features)
        x = self.classify(x)
        return x

# This train func is for tuning only so that accuracy recoding is removed
def train(model, optimizer, train_loader, device=torch.device("cpu")):
    model.train()
    loss_fn = nn.CrossEntropyLoss()
    for batch_idx, (data, target) in enumerate(train_loader):
        if batch_idx * len(data) > EPOCH_SIZE:
```

```
            return
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target.squeeze().long())
        loss.backward()
        optimizer.step()

# This test func is for tuning only so that accuracy recoding is removed
def test(model, test_loader, device=torch.device("cpu")):
    model.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
         for batch_idx, (data, target) in enumerate(test_loader):
            if batch_idx * len(data) > TEST_SIZE:
                break
            data, target = data.to(device), target.squeeze().to(device)
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    return correct / total
```

6.3 DCN.py

[ ]:
```
from functools import reduce
import torch
import torch.nn as nn
import torch.optim as optim

class DeepConvNet(nn.Module):
    def __init__(self, activation=None, deepconv=[25,50,100,200], dropout=0.5):
        super(DeepConvNet, self).__init__()

        if not activation:
            activation = nn.ELU

        self.deepconv = deepconv
        self.conv0 = nn.Sequential(
            nn.Conv2d(
                1, deepconv[0], kernel_size=(1, 5),
                stride=(1,1), padding=(0,0), bias=True
            ),
```

```
            nn.Conv2d(
                deepconv[0], deepconv[0], kernel_size=(2,1),
                stride=(1,1), padding=(0,0), bias=True
            ),
            nn.BatchNorm2d(deepconv[0]),
            activation(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=dropout)
        )

        for idx in range(1, len(deepconv)):
            setattr(self, 'conv'+str(idx), nn.Sequential(
                nn.Conv2d(
                    deepconv[idx-1], deepconv[idx], kernel_size=(1,5),
                    stride=(1,1), padding=(0,0), bias=True
                ),
                nn.BatchNorm2d(deepconv[idx]),
                activation(),
                nn.MaxPool2d(kernel_size=(1, 2)),
                nn.Dropout(p=dropout)
            ))


        flatten_size =  deepconv[-1] * reduce(
            lambda x,_: round((x-4)/2), deepconv, 750)
        self.classify = nn.Sequential(
            nn.Linear(flatten_size, 2, bias=True),
        )

    def forward(self, x):
        for i in range(len(self.deepconv)):
            x = getattr(self, 'conv'+str(i))(x)
        # flatten
        x = x.view(-1, self.classify[0].in_features)
        x = self.classify(x)
        return x
```

6.4 Tuning.py

This code should run in command line interface to get arguments

[ ]:

[ ]:
```
from __future__ import print_function

import argparse
import os
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from dataloader import read_bci_data
from EEGNet import (EEGNet, train, test)

import ray
from ray import tune
from ray.tune.schedulers import ASHAScheduler

activation_list = [nn.ELU, nn.ReLU, nn.ReLU6, nn.LeakyReLU]


def gen_bci_dataloaders():
    train_x, train_y, test_x, test_y = read_bci_data()
    datasets = []
    for train, test in [(train_x, train_y), (test_x, test_y)]:
        train = torch.stack(
            [torch.Tensor(train[i]) for i in range(train.shape[0])]
        )
        test = torch.stack(
            [torch.Tensor(test[i:i+1]) for i in range(test.shape[0])]
        )
        datasets += [TensorDataset(train, test)]

    return datasets

def get_data_loaders():
    train_dataset, test_dataset = gen_bci_dataloaders()
    kwargs = {'num_workers': 1, 'pin_memory': True} if torch.cuda.is_available()␣
 ↪else {}
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=len(test_dataset),␣
 ↪shuffle=True)
    return train_loader, test_loader

# Change these values if you want the training to run quicker or slower.
EPOCH_SIZE = 512
TEST_SIZE = 256

# Training settings
parser = argparse.ArgumentParser(description="PyTorch MNIST Example")
parser.add_argument(    "--use-gpu",    action="store_true",    default=True,    ␣
 ↪help="enables CUDA training")
parser.add_argument(
    "--ray-address", type=str, help="The Redis address of the cluster.")
```

```
parser.add_argument(
    "--smoke-test", action="store_true", help="Finish quickly for testing")


class TrainEEG(tune.Trainable):
    def _setup(self, config):
        self.device = torch.device("cuda" if torch.cuda.is_available() else␣
 ↪"cpu")
        self.train_loader, self.test_loader = get_data_loaders()
        self.model = EEGNet(
                activation=activation_list[config.get("activation", 1)]).to(self.
 ↪device)
                #dropout1=config.get("drop1", 0.6),
                #dropout2=config.get("drop2", 0.65)
                #).to(self.device)

        self.model = self.model.to(self.device)
        self.optimizer = optim.Adam(
                self.model.parameters(),
                lr=config.get("lr", 0.0025))

    def _train(self):
        train(
            self.model, self.optimizer, self.train_loader, device=self.device)
        acc = test(self.model, self.test_loader, self.device)
        return {"mean_accuracy": acc}

    def _save(self, checkpoint_dir):
        checkpoint_path = os.path.join(checkpoint_dir, "model.pth")
        torch.save(self.model.state_dict(), checkpoint_path)
        return checkpoint_path

    def _restore(self, checkpoint_path):
        self.model.load_state_dict(torch.load(checkpoint_path))



if __name__ == "__main__":
    return
    args = parser.parse_args()
    ray.init(address=args.ray_address, num_cpus=3 if args.smoke_test else None)
    #sched = ASHAScheduler(metric="mean_accuracy")


    analysis = tune.run(
        TrainEEG,
        #scheduler=sched,
```

```
        stop={
            "mean_accuracy": 0.87,
            "training_iteration": 3 if args.smoke_test else 300,
        },
        resources_per_trial={
            "cpu": 3,
            "gpu": int(args.use_gpu)
        },
        num_samples=1 if args.smoke_test else 10,
        checkpoint_at_end=True,
        checkpoint_freq=10,
        config={
            "activation": tune.grid_search([0, 1, 2, 3]),
            #"args": args,
            #"drop1": tune.grid_search([0.6,0.65]),
            #"drop2": tune.grid_search([0.65,0.68]),
            "lr": tune.grid_search([0.0025,0.0026])
            #"lr": tune.grid_search([0.0022,0.0025, 0.003, 0.01])
            #"lr": tune.uniform(0.0022, 0.005)
        })

    print("Best config is:", analysis.get_best_config(metric="mean_accuracy"))
```