Lab2: EEG classification

Author: 311553007 應耀德

1. Introduction

Lab Objective:

In this lab, you will need to implement simple EEG classification models which are EEGNet, DeepConvNet with BCI competition dataset. Additionally, you need to try different kinds of activation function including ReLU, Leaky ReLU, ELU.

Requirements:

1. Implement the EEGNet, DeepConvNet with three kinds of activation function including ReLU, Leaky ReLU, ELU.
2. In the experiment results, you have to show the highest accuracy (not loss) of two architectures with three kinds of activation functions.
3. To visualize the accuracy trend, you need to plot each epoch accuracy (not loss) during training phase and testing phase.

Data:

BCI Competition III - IIIb Cued motor imagery with online feedback (non-stationary classifier) with 2 classes (left hand, right hand) from 3 subjects.

[2 classes, 2 bipolar EEG channels]

Reference: http://www.bbci.de/competition/iii/desc_IIIb.pdf

- Train data: S4b_train.npz、X11b_train.npz
  Test data: S4b_test.npz、X11b_test.npz

```
Label 0 in train data: 540
Label 1 in train data: 540
Label 0 in test data: 540
Label 1 in test data: 540
```

Figure 1. Label distribution for data

- BCIDataset

```python
class BCIDataset(Dataset):
    _data_files: List[str]
    _data_loaded: bool = False

    _data: np.ndarray
    _labels: np.ndarray

    def __init__(
        self, data_files: List[str] = ["S4b_train.npz", "X11b_train.npz"]
    ):
        super(BCIDataset, self).__init__()
        for file in data_files:
            if not os.path.exists(file):
                raise FileNotFoundError(f"The data file {file} does not exist.")

        self._data_files = data_files

    def __len__(self):
        # lazy loading
        if not self._data_loaded:
            self._load_data()

        return self._labels.shape[0]

    def __getitem__(self, index: int) -> Tuple[torch.Tensor]:
        # lazy loading
        if not self._data_loaded:
            self._load_data()

        return torch.from_numpy(self._data[index]), self._labels[index]

    def _load_data(self):
        data = []
        labels = []

        for file in self._data_files:
            with np.load(file) as f:
                data.append(f["signal"])
                labels.append(f["label"])

        self._data = np.concatenate(data, axis = 0)
        self._labels = np.concatenate(labels, axis = 0)

        self._data = np.expand_dims(self._data, axis=1).swapaxes(-1, -2)
        self._labels -= 1

        mask = np.where(np.isnan(self._data))
        self._data[mask] = np.nanmean(self._data)

        self._data_loaded = True

train_dataset = BCIDataset(data_files = ["S4b_train.npz", "X11b_train.npz"])
test_dataset = BCIDataset(data_files = ["S4b_test.npz", "X11b_test.npz"])
```

Figure 2. Implementation of BCIDataset

Loss functions:

- Cross-Entropy Loss

$$l(\theta) = -\sum_{i=1}^{n} \left(y_i \log \hat{y}_{\theta,i} + (1 - y_i) \log \left(1 - \hat{y}_{\theta,i}\right) \right)$$

Where

- n: *number of data points*
- i: *index of data points*
- θ: *weights*
- $\hat{y}$: *predict result*
- y: *ground truth*

Optimizer:

- Adam (Adaptive Moment Estimation)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L_t}{\partial W_t}$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_2)(\frac{\partial L_t}{\partial W_t})^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \qquad W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where

- mt: $first\ moment\ at\ time\ step\ t$
- vt: $second\ moment\ at\ time\ step\ t$
- $\beta 1, \beta 2$: $betas$
- L: $loss$
- W: $model\ weights$
- η: $learning\ rate$
- ϵ: $term\ added\ to\ the\ denominator\ to\ improve\ numerical\ stability$

2. Experiment set up

Shared settings:

- Batch Size: 512
- Epoch Size: 500
- Optimizer: Adam, learning rate = 0.001, weight decay = 0.1
- Loss function: Cross-Entropy Loss

Base class:

To share the way to select the activation function, extract the same parts to base class.
self.activation is a property of lambda function to generate selected activation function.

```python
class Model(nn.Module):
    _activation_dict = {
        "ELU": lambda: nn.ELU(alpha = 1.0),
        "ReLU": lambda: nn.ReLU(),
        "LeakyReLU": lambda: nn.LeakyReLU(negative_slope = 0.01)
    }
    _activation_name: str

    def __init__(self, activation: str):
        super(Model, self).__init__()
        if activation not in self._activation_dict:
            raise NotImplementedError(f"The activation function {activation} is not implemented.")

        self.activation = None
        self._activation_name = activation

    @property
    def activation(self) -> nn.Module:
        return self._activation_dict[self._activation_name]()

    @activation.setter
    def activation(self, _):
        pass
```

Figure 3. Implementation of base class

EEGNet:

```python
class EEGNet(Model):
    first_conv: nn.Sequential
    depthwise_conv: nn.Sequential
    separable_conv: nn.Sequential
    classfier: nn.Sequential

    def __init__(self, activation: str = "ELU", dropout: float = 0.25):
        super(EEGNet, self).__init__(activation)

        self.first_conv = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size = (1, 51), stride = (1, 1), padding = (0, 25), bias = False),
            nn.BatchNorm2d(16, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.depthwise_conv = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size = (2, 1), stride = (1, 1), groups = 16, bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            self.activation,
            nn.AvgPool2d(kernel_size = (1, 4), stride = (1, 4), padding = 0),
            nn.Dropout(p = dropout)
        )
        self.separable_conv = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size = (1, 15), stride = (1, 1), padding = (0, 7), bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            self.activation,
            nn.AvgPool2d(kernel_size = (1, 8), stride = (1, 8), padding = 0),
            nn.Dropout(p = dropout)
        )
        self.classfier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(736, 2, bias = True)
        )

    def forward(self, x):
        x = self.first_conv(x)
        x = self.depthwise_conv(x)
        x = self.separable_conv(x)
        return self.classfier(x)

network = EEGNet(activation = "ELU")
print(network)
```

Figure 4. Implementation of EEGNet

```
EEGNet(
  (first_conv): 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)
  )
  (depthwise_conv): 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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (separable_conv): 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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (classfier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

Figure 5. An example architecture of EEGNet with ELU activation function

DeepConvNet:

```python
class DeepConvNet(Model):
    first_conv_block: nn.Sequential
    conv_blocks: nn.Sequential
    classfier: nn.Sequential

    def __init__(self, activation: str = "ELU", dropout: float = 0.5):
        super(DeepConvNet, self).__init__(activation)

        in_channels = 1
        out_channels = 25
        self.first_conv_block = nn.Sequential(
            # H = 2, W = 750
            nn.Conv2d(in_channels, out_channels, kernel_size = (1, 5), stride = (1, 1), bias = True),
            # H = 2, W = 750 - 5 + 1 = 746
            nn.Conv2d(out_channels, out_channels, kernel_size = (2, 1), stride = (1, 1), bias = True),
            # H = 1, W = 746
            nn.BatchNorm2d(out_channels, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            self.activation,
            nn.MaxPool2d((1, 2)),
            # H = 1, W = 373
            nn.Dropout(p = dropout)
        )

        kernels = (50, 100, 200)
        conv_blocks = []
        for kernel in kernels:
            in_channels = out_channels
            out_channels = kernel
            conv_blocks.append(nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size = (1, 5), stride = (1, 1), bias = True),
                nn.BatchNorm2d(out_channels, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
                self.activation,
                nn.MaxPool2d((1, 2)),
                nn.Dropout(p = dropout)
            ))
        # conv_blocks[0]: H = 1, W = (373 - 5 + 1) / 2 = 184 (floor)
        # conv_blocks[1]: H = 1, W = (184 - 5 + 1) / 2 = 90
        # conv_blocks[2]: H = 1, W = (90 - 5 + 1) / 2 = 43
        self.conv_blocks = nn.Sequential(*conv_blocks)

        self.classfier = nn.Sequential(
            nn.Flatten(),
            # Batch_Size, 200 * 1 * 43 = 8600
            nn.Linear(8600, 2, bias = True)
        )

    def forward(self, x):
        x = self.first_conv_block(x)
        x = self.conv_blocks(x)
        return self.classfier(x)

network = DeepConvNet(activation = "ELU")
print(network)
```

Figure 6. Implementation of DeepConvNet

```
DeepConvNet(
  (first_conv_block): 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)
  )
  (conv_blocks): Sequential(
    (0): 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)
    )
    (1): 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)
    )
    (2): 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)
    )
  )
  (classfier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=8600, out_features=2, bias=True)
  )
)
```

Figure 7. An example architecture of DeepConvNet with ELU activation function

Activation functions (ReLU, Leaky ReLU, ELU):

- ReLU

  ReLU is a good activation function for hidden units. It diminishes the vanishing gradient problem happened from Sigmoid and Tanh. But it still has a chance to cause this problem when the input x is negative due to too many dead neurons.

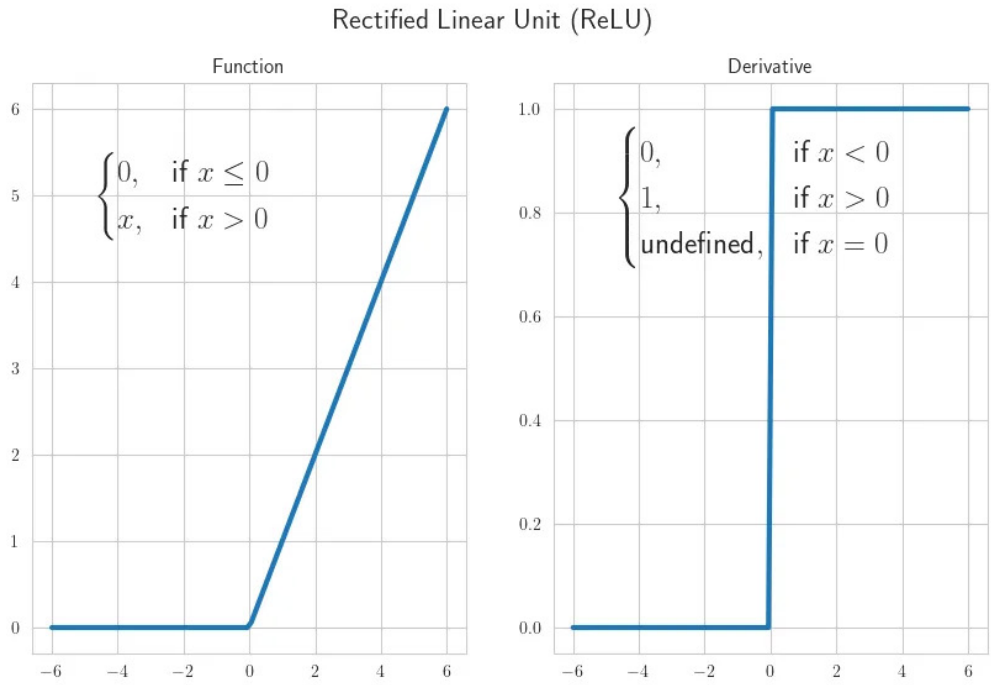

Figure 8. Visualization of ReLU function

  - The function of the forward pass

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

$$f(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$$

  - The function of the backward pass (derivative function)

$$f'(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x > 0 \\ \text{undefined}, & \text{if } x = 0 \end{cases}$$

- Leaky ReLU

  Leaky ReLU is an improved version of ReLU.

  Although Leaky ReLU keeps the negative value in limited scope to diminish the number of dead neurons, sometimes the value 0.01 is still too small to update parameters.

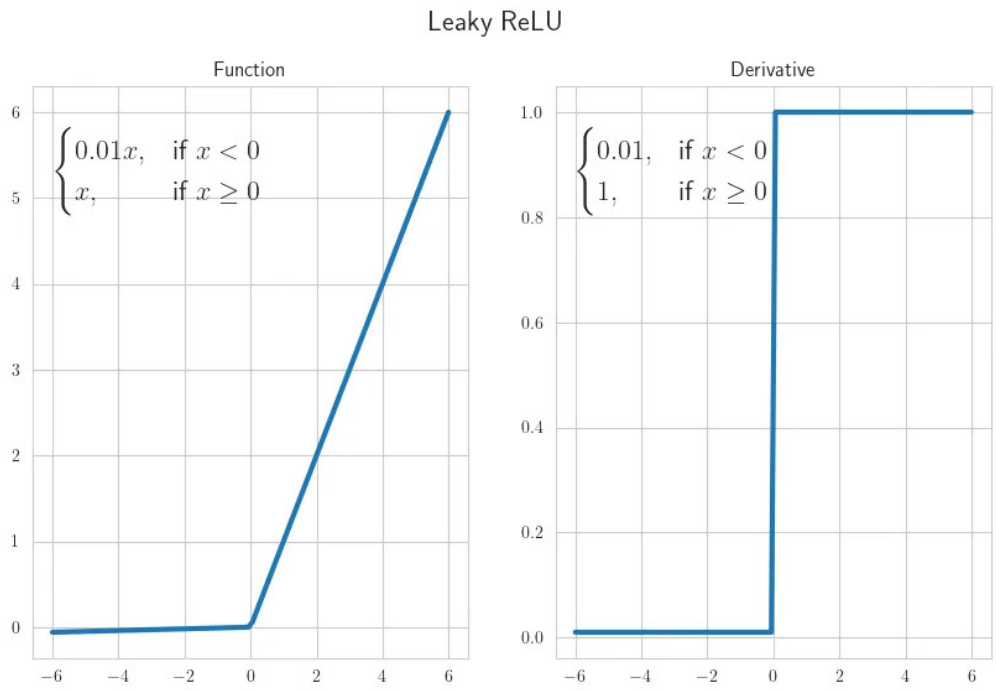

Figure 9. Visualization of Leaky ReLU function

- The function of the forward pass

$$f(x) = \begin{cases} 0.01x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$$

- The function of the backward pass (derivative function)

$$f'(x) = \begin{cases} 0.01, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

- ELU

  The default value of α hyperparameter is 1.0.

  ELU has all advantages of ReLU and succeed to solve the vanishing gradient problem.

  But there is no explicit evidence to prove that it is better than ReLU and Leaky ReLU.

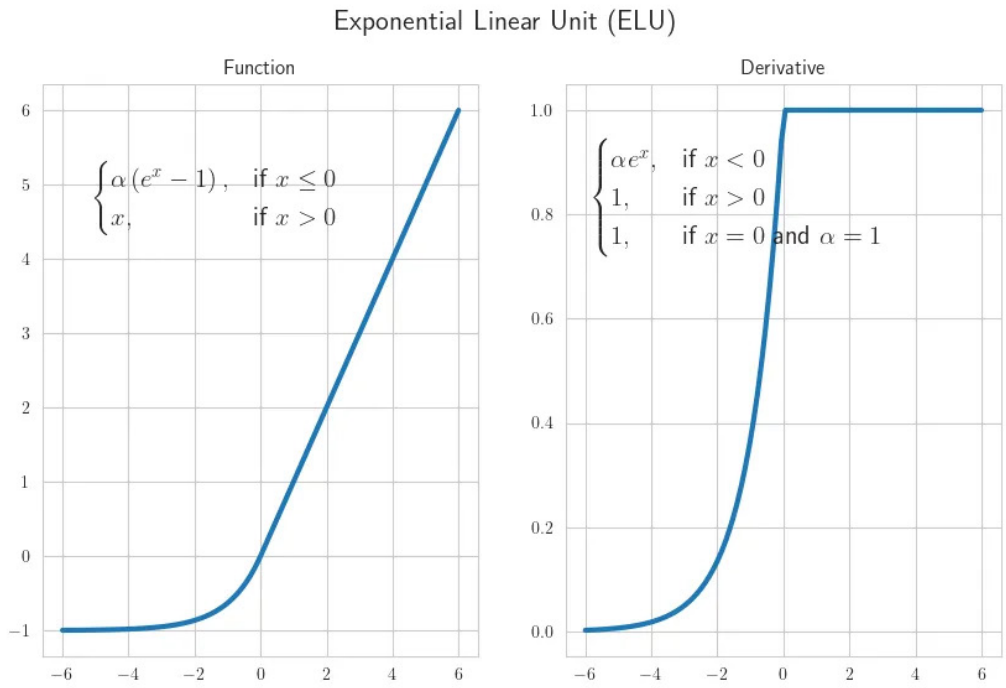

Figure 10. Visualization of ELU function

- The function of the forward pass

$$f(x) = \begin{cases} \alpha(e^x - 1), & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$$

- The function of the backward pass (derivative function)

$$f'(x) = \begin{cases} \alpha e^x, & \text{if } x < 0 \\ 1, & \text{if } x > 0 \\ 1, & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$$

3. Experimental results

Result:

Network	ReLU	Leaky ReLU	ELU
EEGNet	73.76%	74.15%	**75.53%**
DeepConvNet	**84.14%**	83.35%	80.59%

Best model (highest testing accuracy):

- **EEGNet (at epoch 325):**

```
EEGNet(
  (first_conv): 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)
  )
  (depthwise_conv): 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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (separable_conv): 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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (classfier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=736, out_features=2, bias=True)
  )
)
Accuracy of EEGNet with ELU: 0.7553013563156128
```

- **DeepConvNet (at epoch 221):**

```
DeepConvNet(
  (first_conv_block): 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): ReLU()
    (4): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Dropout(p=0.5, inplace=False)
  )
  (conv_blocks): Sequential(
    (0): 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): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
    (1): 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): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
    (2): 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): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (classfier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=8600, out_features=2, bias=True)
  )
)
Accuracy of DeepConvNet with ReLU: 0.8414248625437418
```

Comparison figures:

Note: To use the same hyperparameters, I choose an appropriate set of hyperparameters to fit the data although the model is still overfitting. (Increasing the weight decay can solve this problem but decreasing accuracy.)

I think we need more data to train the model.

- **EEGNet with 3 activation functions:**

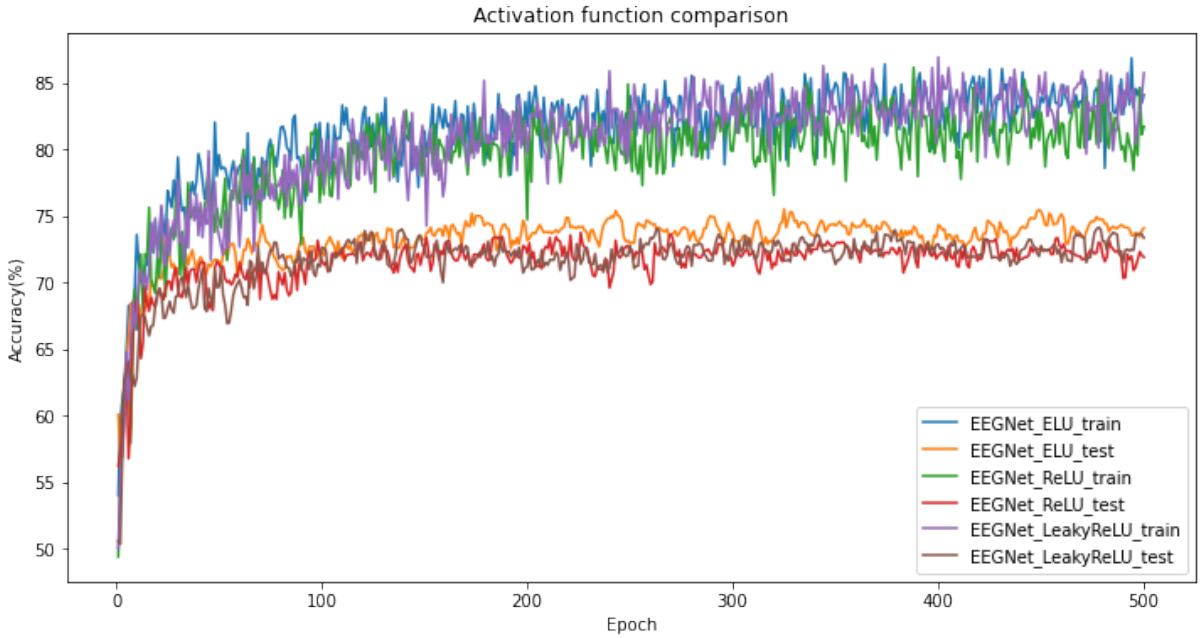

Figure 11. Accuracy comparison for EEGNet with 3 activation functions

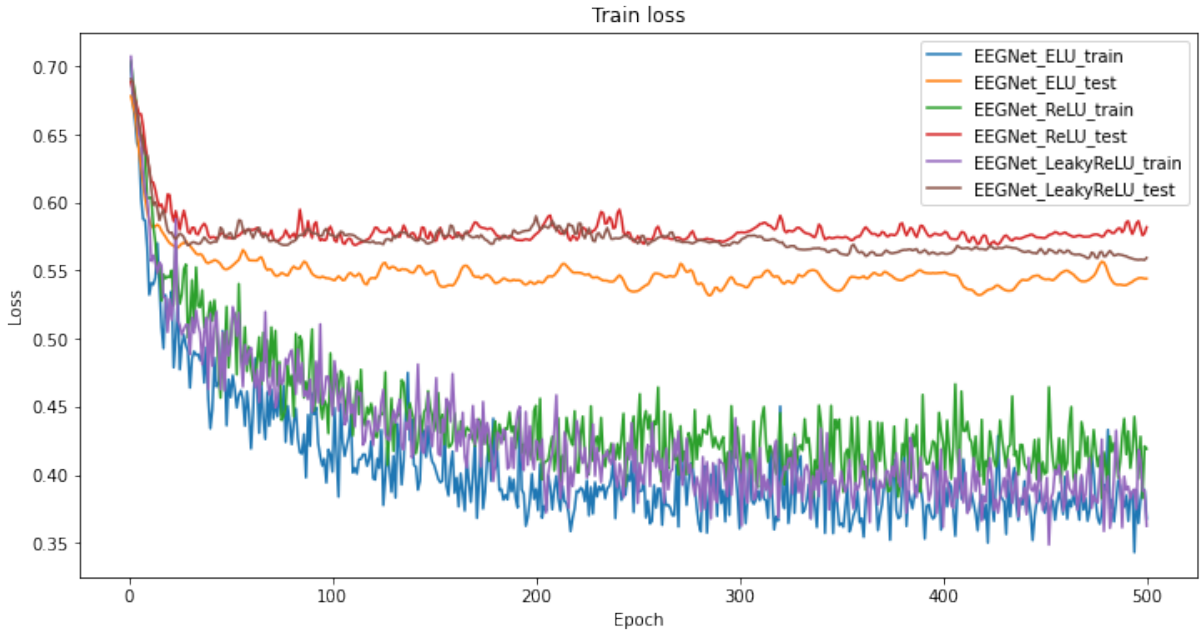

Figure 12. Loss comparison for EEGNet with 3 activation functions

- **<u>DeepConvNet with 3 activation functions:</u>**

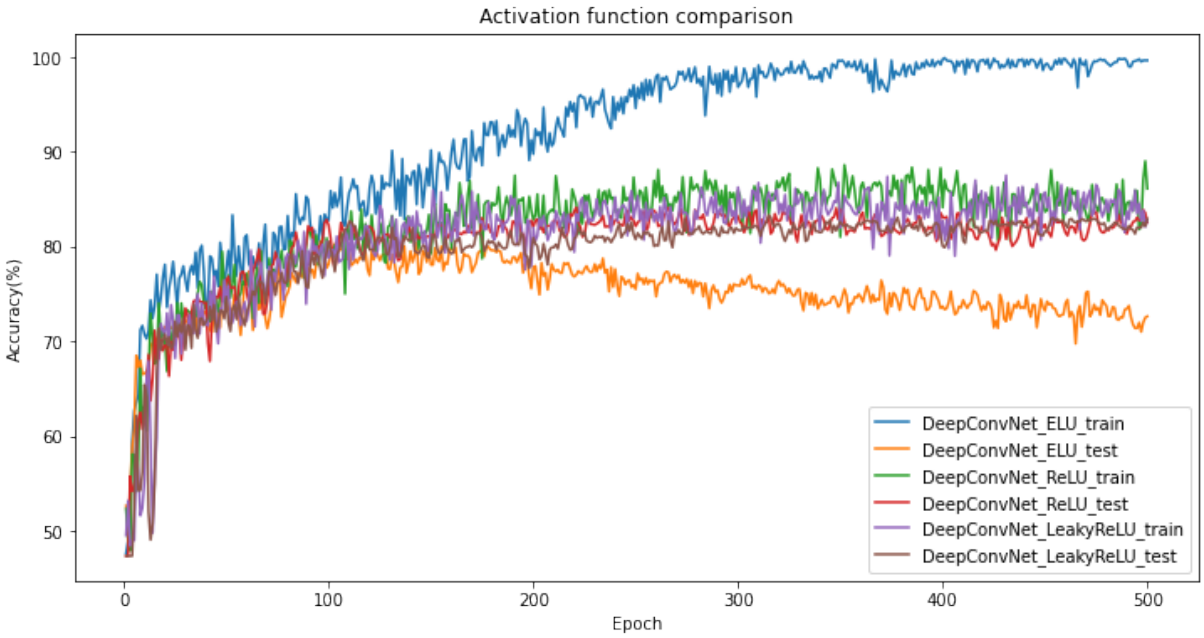

Figure 13. Accuracy comparison for DeepConvNet with 3 activation functions

Figure 14. Loss comparison for DeepConvNet with 3 activation functions

4. Discussion

What happened if I increase weight decay to diminish overfitting?

Use the same experiment set up except for changing the weight decay.

No matter the more weight decay I add, and the more accuracy is decreasing. The test accuracy is totally the same.

It seems that the model cannot generalize in a limited number of data (1080 samples).

Possible solutions:

1. Try adding more data (**higher probability)**
   Because the comparison figures in the experiment result part indicate the model can learn somethings from the training data but it cannot predict the testing data as well as the training data.
2. Study the EEGNet paper and find what the EEG signal means.

3. Change the model architecture
4. Finetune the pretrained model

- EEGNet, weight decay = 0.2

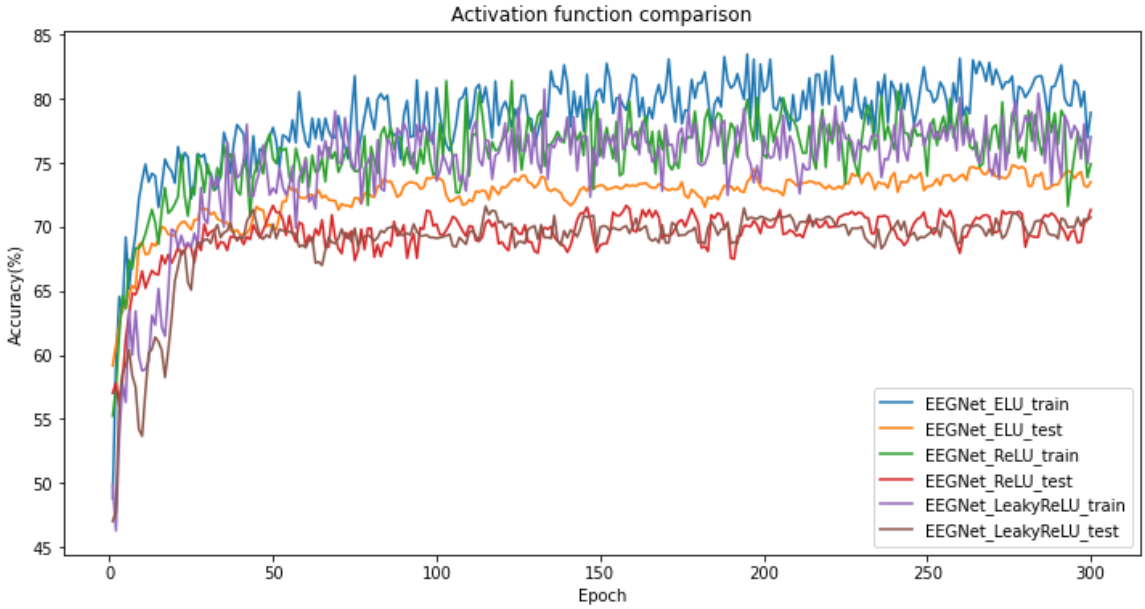

Figure 15. Accuracy comparison for EEGNet with weight decay = 0.2 with 3 activation functions

- EEGNet, weight decay = 0.3

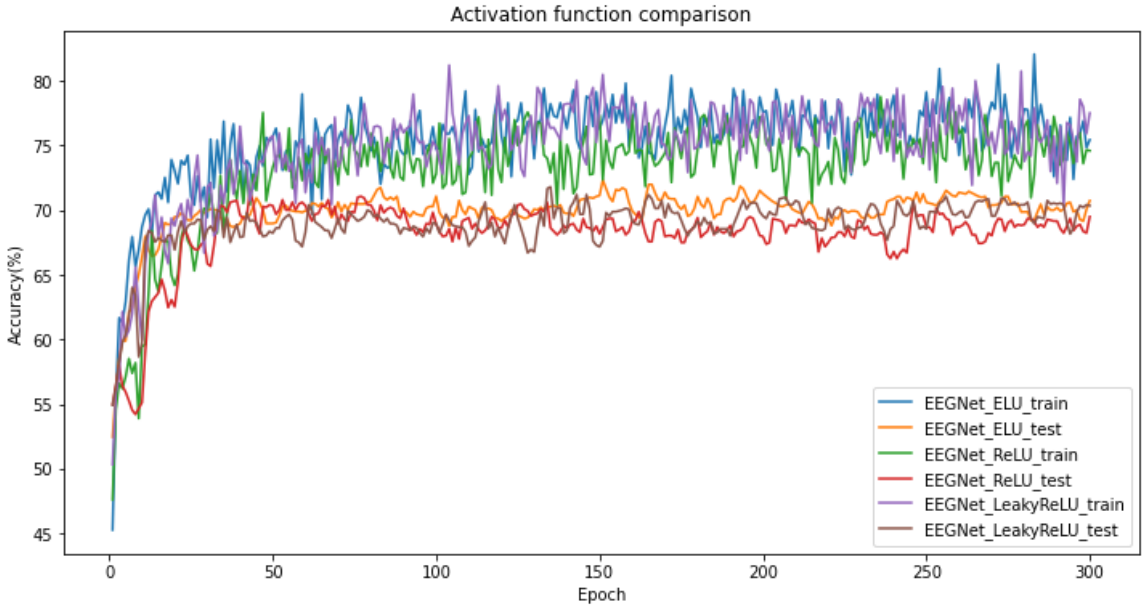

Figure 15. Accuracy comparison for EEGNet with weight decay = 0.3 with 3 activation functions

- DeepConvNet, weight decay = 0.2

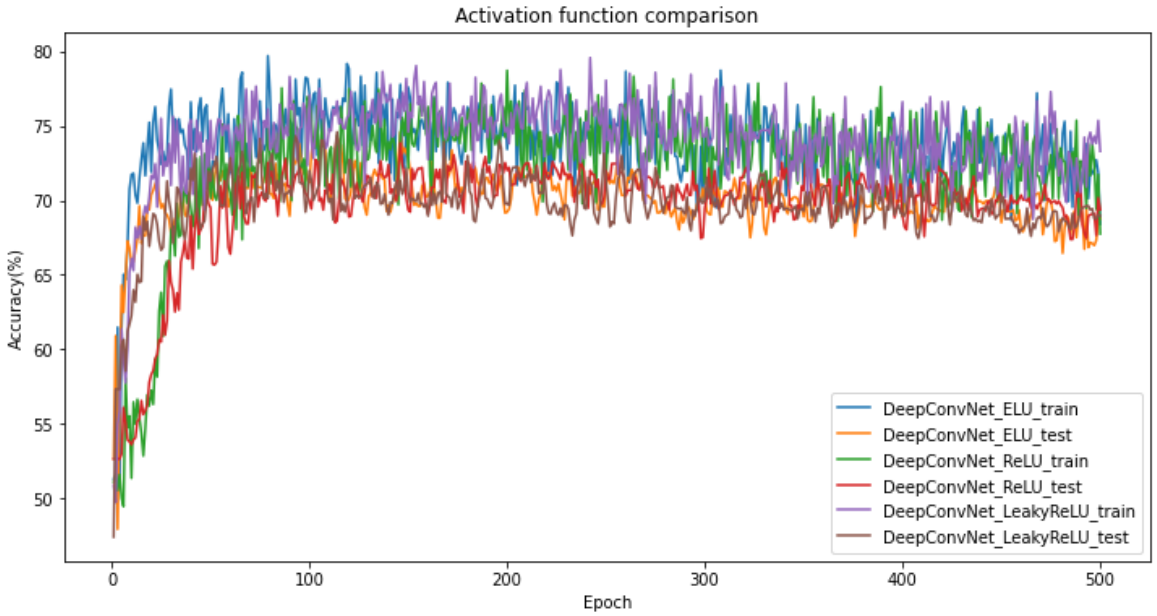

Figure 15. Accuracy comparison for DeepConvNet with weight decay = 0.2 with 3 activation functions

- DeepConvNet, weight decay = 0.3

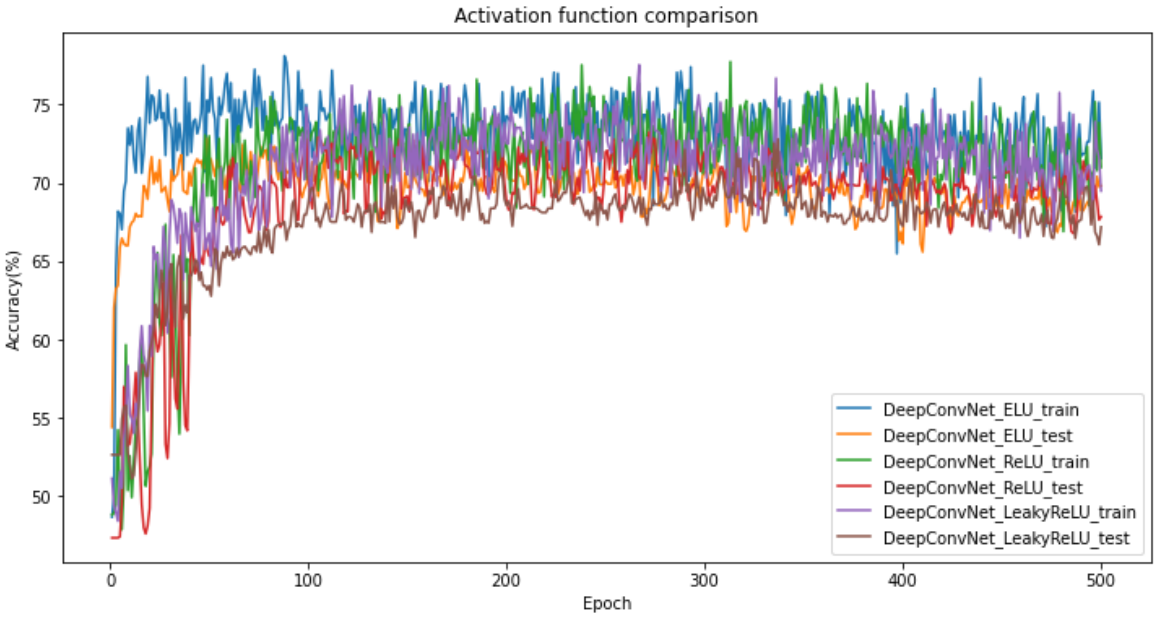

Figure 16. Accuracy comparison for DeepConvNet with weight decay = 0.3 with 3 activation functions

5. Reference

A. Activation functions: https://ml-explained.com/blog/activation-functions-explained

B. PyTorch document: https://pytorch.org/docs/stable/index.html

C. Lab 2 materials