# 312554012 王偉誠 Lab3 : EEG classification

## 1. Introduction

For this assignment, we will utilize Convolutional Neural Networks (CNNs) to tackle the brainwave pattern classification problem. Specifically, we will implement EEGNet and DeepConvNet using PyTorch for EEG classification. PyTorch's torch.nn package will be employed, which includes modules for constructing CNNs, extensible classes, and all necessary components. Once the model is established, we only need to define the forward function, as PyTorch handles backpropagation automatically. During the forward process, different activation functions (including ReLU, LeakyReLU, ELU) will be employed to observe their respective impacts on the results.

## 2. Experiment set up:

### A. The detail of your model

**EEGNet**

```
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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25)
  )
  (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): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

Just implement the given architecture as provided by TA.

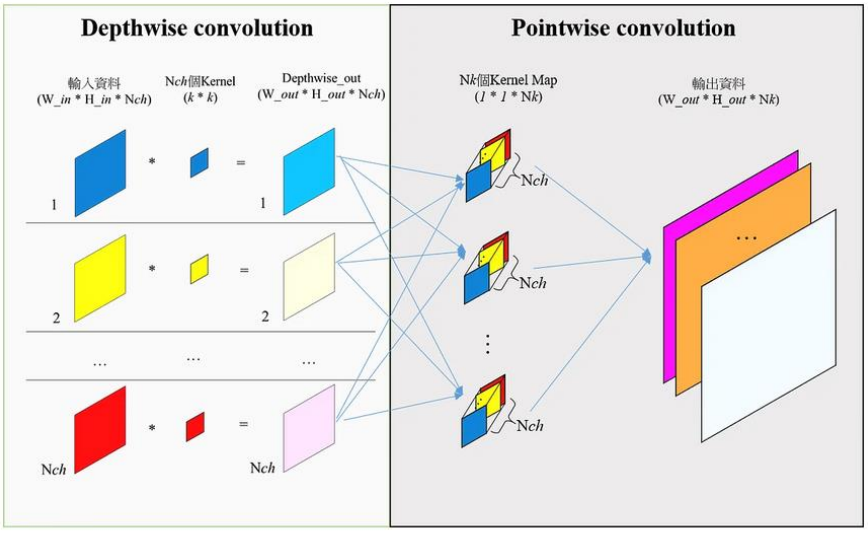
```python
class EEGNet(nn.Module):
    def __init__(self, activation_func, device):
        super(EEGNet, self).__init__()
        self.device = device
        self.firstconv = nn.Sequential(
            nn.Conv2d(
                1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False
            ),
            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_func,
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
            nn.Dropout(p=0.25),
        )
        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_func,
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
            nn.Dropout(p=0.25),
        )
        self.classify = nn.Linear(736, 2)

    def forward(self, x):
        x = x.to(self.device)
        x = self.firstconv(x)
        x = self.depthwiseConv(x)
        x = self.separableConv(x)
        x = x.view(x.shape[0], -1)
        x = self.classify(x)
        return x      You, 21 小時前 • add lab2 …
```

In this implementation, I have made the activation function and device settings configurable as parameters. This allows for easy experimentation with different activation functions (ReLU, LeakyReLU, ELU) during execution, as well as the flexibility to choose between GPU and CPU for data synchronization and processing.

- Sequential(): Executes the contents within the parentheses in sequential order.
- Conv2d(): This core component of CNN allows setting parameters like kernel size and stride. It detects various features in an image and outputs channels that serve as input for the next layer.
- BatchNorm2d(): This layer applies normalization to the input data, transforming it to a normalized space with a mean of 0 and a standard deviation of 1. This enhances network accuracy.
- MaxPool(): Pooling is a technique to reduce the data volume of an image while preserving important information. Max Pooling calculates the maximum value in each pool, ensuring that the network can detect specific features regardless of their position in the image.

- Dropout(p=0.25): In this case, 0.25 indicates that 25% of neurons in the layer will be randomly deactivated during each iteration of training. The specific neurons to deactivate vary randomly in each iteration, creating seemingly different structures in the neural network at each iteration to prevent overfitting.
- Linear(): This layer serves as the final layer in the network and acts as a fully connected layer, computing the scores for each class.

EEGNet incorporates a technique called Depthwise Separable Convolution. In a conventional convolution operation, each kernel map convolves with all the channels of the input data. However, Depthwise Separable Convolution takes a different approach. It creates a separate k x k kernel for each channel of the input data. Then, each channel convolves independently with its corresponding kernel. The process can be represented with the following schematic diagram:




Depthwise separable convolution

From the schematic diagram above, we can deduce that the computational complexity of Depthwise Separable Convolution, as compared to conventional convolution, can be estimated as follows. By doing so, it significantly reduces the computational burden on CNNs. As the size and number of Kernel Maps increase, Depthwise Separable Convolution can achieve even more substantial computational savings. In PyTorch, this can be implemented using the 'groups' parameter.

Input data: $W_{in}$ * $H_{in}$ * Nch，Kernel Map: k * k * Nk
Output data: $W_{out}$ * $H_{out}$ * Nk
一般卷積計算量: $W_{in}$ * $H_{in}$ * Nch* k * k * Nk

$$\frac{\text{Depthwise separable convolution 計算量}}{\text{一般卷積計算量}}$$

$$= \frac{W_{in} * H_{in} * Nch * k * k + Nch * Nk * W\_in * H\_in}{W\_in * H\_in * Nch * k * k * Nk}$$

$$= \frac{1}{Nk} + \frac{1}{k * k}$$

Ref: [https://chih-sheng-huang821.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-mobilenet-depthwise-separable-convolution-f1ed016b3467](https://chih-sheng-huang821.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-mobilenet-depthwise-separable-convolution-f1ed016b3467)

**DeepConvNet**

| Layer | # filters | size | # params | Activation | Options |
|---|---|---|---|---|---|
| Input | | (C, T) | | | |
| Reshape | | (1, C, T) | | | |
| Conv2D | 25 | (1, 5) | 150 | Linear | mode = valid, max norm = 2 |
| Conv2D | 25 | (C, 1) | 25 * 25 * C + 25 | Linear | mode = valid, max norm = 2 |
| BatchNorm | | | 2 * 25 | | epsilon = 1e-05, momentum = 0.1 |
| Activation | | | | ELU | |
| MaxPool2D | | (1, 2) | | | |
| Dropout | | | | | p = 0.5 |
| Conv2D | 50 | (1, 5) | 25 * 50 * C + 50 | Linear | mode = valid, max norm = 2 |
| BatchNorm | | | 2 * 50 | | epsilon = 1e-05, momentum = 0.1 |
| Activation | | | | ELU | |
| MaxPool2D | | (1, 2) | | | |
| Dropout | | | | | p = 0.5 |
| Conv2D | 100 | (1, 5) | 50 * 100 * C + 100 | Linear | mode = valid, max norm = 2 |
| BatchNorm | | | 2 * 100 | | epsilon = 1e-05, momentum = 0.1 |
| Activation | | | | ELU | |
| MaxPool2D | | (1, 2) | | | |
| Dropout | | | | | p = 0.5 |
| Conv2D | 200 | (1, 5) | 100 * 200 * C + 200 | Linear | mode = valid, max norm = 2 |
| BatchNorm | | | 2 * 200 | | epsilon = 1e-05, momentum = 0.1 |
| Activation | | | | ELU | |
| MaxPool2D | | (1, 2) | | | |
| Dropout | | | | | p = 0.5 |
| Flatten | | | | | |
| Dense | N | | | softmax | max norm = 0.5 |

Just implement the given architecture as provided by TA. As mentioned before, I have made the activation function and device settings configurable as parameters for ease of use. One important thing to note is that since I am using the nn.CrossEntropyLoss as the loss function, there is no need to add a softmax activation after the final fully connected (fc) layer. The nn.CrossEntropyLoss criterion already incorporates LogSoftMax and NLLLoss in a single class, handling the necessary computations during the training process.

```python
You, 21 小時前 | 1 author (You)
class DeepConvNet(nn.Module):
    def __init__(self, activation_func, device) -> None:
        super(DeepConvNet, self).__init__()
        self.device = device        You, 21 小時前 • add lab2 ...
        self.conv1 = nn.Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
        self.conv2 = nn.Sequential(
            nn.Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1)),
            nn.BatchNorm2d(
                25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
            ),
            activation_func,
            nn.MaxPool2d(
                kernel_size=(1, 2),
                stride=(1, 2),
                padding=0,
                dilation=1,
                ceil_mode=False,
            ),
            nn.Dropout(p=0.5),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1)),
            nn.BatchNorm2d(
                50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
            ),
            activation_func,
            nn.MaxPool2d(
                kernel_size=(1, 2),
                stride=(1, 2),
                padding=0,
                dilation=1,
                ceil_mode=False,
            ),
            nn.Dropout(p=0.5),
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1)),
            nn.BatchNorm2d(
                100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
            ),
            activation_func,
            nn.MaxPool2d(
                kernel_size=(1, 2),
                stride=(1, 2),
                padding=0,
                dilation=1,
                ceil_mode=False,
            ),
            nn.Dropout(p=0.5),
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1)),
            nn.BatchNorm2d(
                200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
            ),
            activation_func,
            nn.MaxPool2d(
                kernel_size=(1, 2),
                stride=(1, 2),
                padding=0,
                dilation=1,
                ceil_mode=False,
            ),
            nn.Dropout(p=0.5),
        )
        self.classifier = nn.Linear(8600, 2)

    def forward(self, x):
        x = x.to(self.device)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)

```

Certainly, let's take a closer look at the formulas for LogSoftMax and NLLLoss:

**nn.LogSoftmax**

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

**nn.NLLLoss**

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

Upon closer examination of nn.CrossEntropyLoss, we find that it can be viewed as a combination of nn.LogSoftmax and nn.NLLLoss. Therefore, in the aforementioned implementation, there is no need to add a softmax activation after the final fully connected (fc) layer.

The formula and description of nn.CrossEntropyLoss are as follows:

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)` [SOURCE]

This criterion computes the cross entropy loss between input and target.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C-1]$ where $C$ is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where $x$ is the input, $y$ is the target, $w$ is the weight, $C$ is the number of classes, and $N$ spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Note that this case is equivalent to the combination of `LogSoftmax` and `NLLLoss`.

During the training phase, I utilized the Adam optimizer for gradient descent updates. Specifically, I set the learning rate to 0.001 and the weight decay to 0.01. Weight decay involves adding a penalty term during parameter updates to mitigate overfitting. For the loss function, I employed Cross Entropy, and the training process was executed for a total of 500 epochs.

The training procedure involves the following steps performed iteratively: clearing the optimizer gradients, forwarding the input data through the network, computing the loss, calling the backward function to calculate gradients, and finally, invoking the step function to perform the gradient descent update.

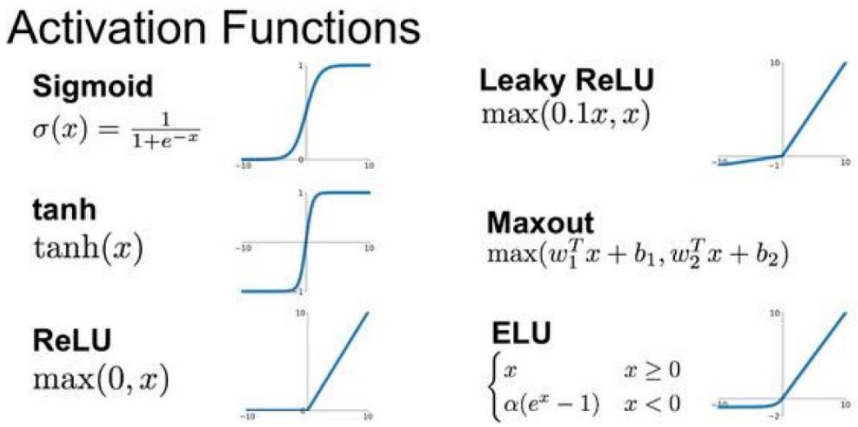
Similarly, during the testing phase, the same steps were executed, except that there was no need to calculate gradients or perform updates. The focus was solely on forwarding the input data through the trained network and evaluating the model's performance on the test dataset.

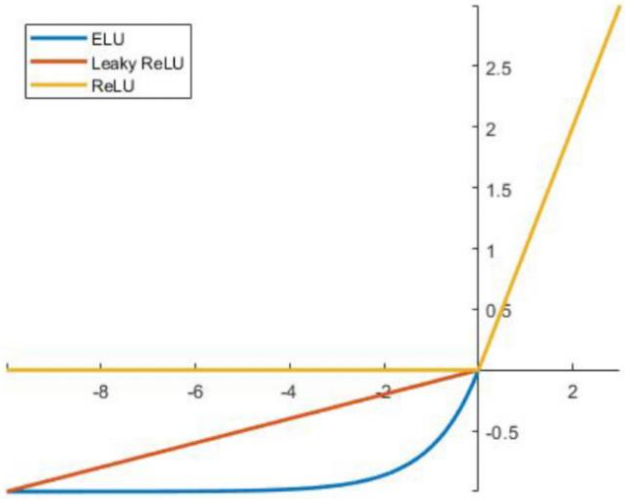
```python
model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
```

```python
# training process
total_loss = 0
total_train = 0
correct_train = 0
model.train()
for i, (data, label) in enumerate(train_loader):
    data = data.to(device, dtype=torch.float)
    label = label.to(device, dtype=torch.long)

    # clear gradient
    optimizer.zero_grad()

    # forward propagation
    output = model(data)

    # calculate cross entropy (loss function)
    loss = criterion(output, label)
    total_loss += loss

    # get predictions from the maximum value
    prediction = torch.max(output.data, 1)[1]

    # total number of labels
    total_train += len(label)

    # total correct predictions
    correct_train += (prediction == label).float().sum()

    # Calculate gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

```python
# testing process
total_test = 0
correct_test = 0
model.eval()
for i, (data, label) in enumerate(test_loader):
    data = data.to(device, dtype=torch.float)
    label = label.to(device, dtype=torch.long)

    # no need to calculate gradient and loss function

    # forward propagation
    output = model(data)      You, 21 小時前 • add lab2

    # get predictions from the maximum value
    prediction = torch.max(output.data, 1)[1]

    # total number of labels
    total_test += len(label)

    # total correct predictions
    correct_test += (prediction == label).float().sum()
```

**B. Explain the activation function (ReLU, Leaky ReLU, ELU)**




The figure above depicts the graphs of various activation functions, including a comparison among ReLU, Leaky ReLU, and ELU. It is evident that the main difference between ELU and Leaky ReLU when compared to ReLU lies in their behavior when the input value is less than zero. ELU retains a non-zero gradient even for negative values, whereas Leaky ReLU sets the gradient directly to zero.

As a consequence, ReLU may encounter an issue known as the "dying ReLU problem," wherein some neurons output zero and become inactive, making it challenging for them to recover and produce meaningful outputs again. In extreme cases, if all inputs to ReLU are negative, then all outputs will be zero, perpetuating this problem. Two primary causes of this phenomenon are improper parameter initialization and a high learning rate, which result in overly large parameter updates during training.

LeakyReLU addresses the dying ReLU problem by introducing a non-zero slope for negative input values. When the input value is greater than zero, LeakyReLU behaves identically to ReLU with a gradient of one. However, for negative values,

LeakyReLU maintains a small non-zero gradient, allowing for effective learning and avoiding the issue of neuron inactivity.

ELU (Exponential Linear Unit) is an activation function that tends to converge values to zero more quickly, resulting in more accurate outputs. Like Leaky ReLU, it can also address the dying ReLU problem. However, one important consideration when using ELU is that it comes with a higher computational cost compared to Leaky ReLU. LeakyReLU do not ensure a noise-robust deactivation state. ELUs saturate to a negative value with smaller inputs and thereby decrease the forward propagated variation and information.

## 3. Experimental results

### A. The highest testing accuracy

learning rate = 0.001
weight decay = 0.01
epochs = 500

**EEGNet**

```
ReLU has max accuracy 88.33333587646484% at epoch 306
Accuracy of EEGNet with ReLU activation function: 88.33333587646484
```

```
LeakyReLU has max accuracy 87.31481170654297% at epoch 130
Accuracy of EEGNet with LeakyReLU activation function: 87.31481170654297
```

```
ELU has max accuracy 83.05555725097656% at epoch 487
Accuracy of EEGNet with ELU activation function: 83.05555725097656
```

**DeepConvNet**

```
ReLU has max accuracy 84.72222137451172% at epoch 435
Accuracy of DeepConvNet with ReLU activation function: 84.72222137451172
```

```
LeakyReLU has max accuracy 85.55555725097656% at epoch 464
Accuracy of DeepConvNet with LeakyReLU activation function: 85.55555725097656
```

```
ELU has max accuracy 82.96295928955078% at epoch 447
Accuracy of DeepConvNet with ELU activation function: 82.96295928955078
```

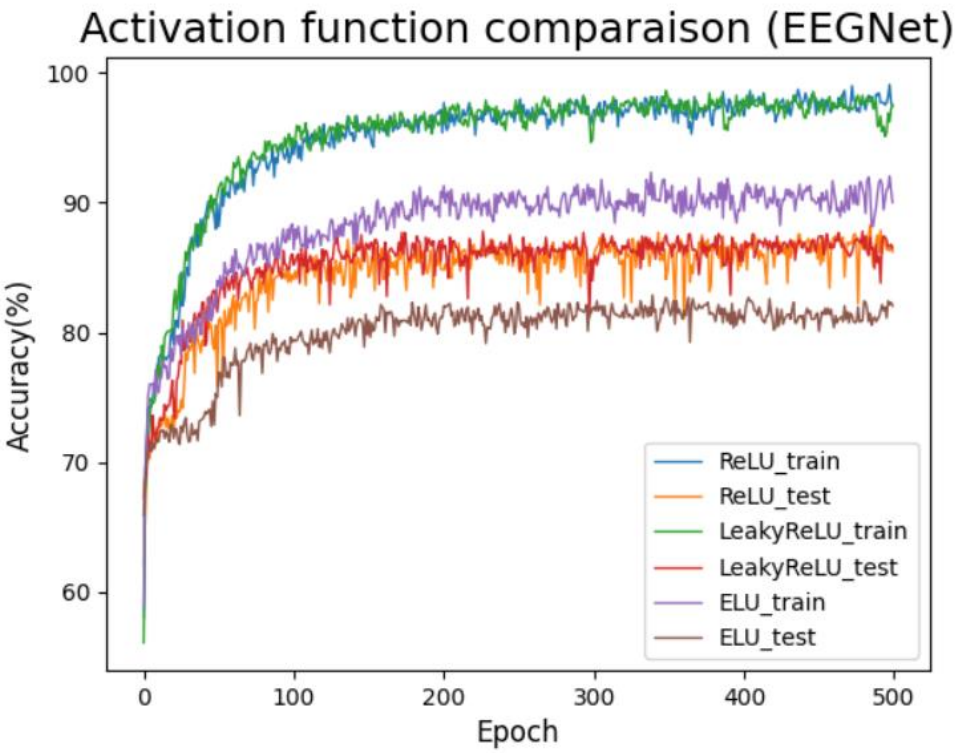
Testing accuracy comparison:

| | **ReLU** | **LeakyReLU** | **ELU** |
|---|---|---|---|
| **EEGNet** | 88.33% | 87.31% | 83.05% |
| **DeepConvNet** | 84.72% | 85.55% | 82.96% |

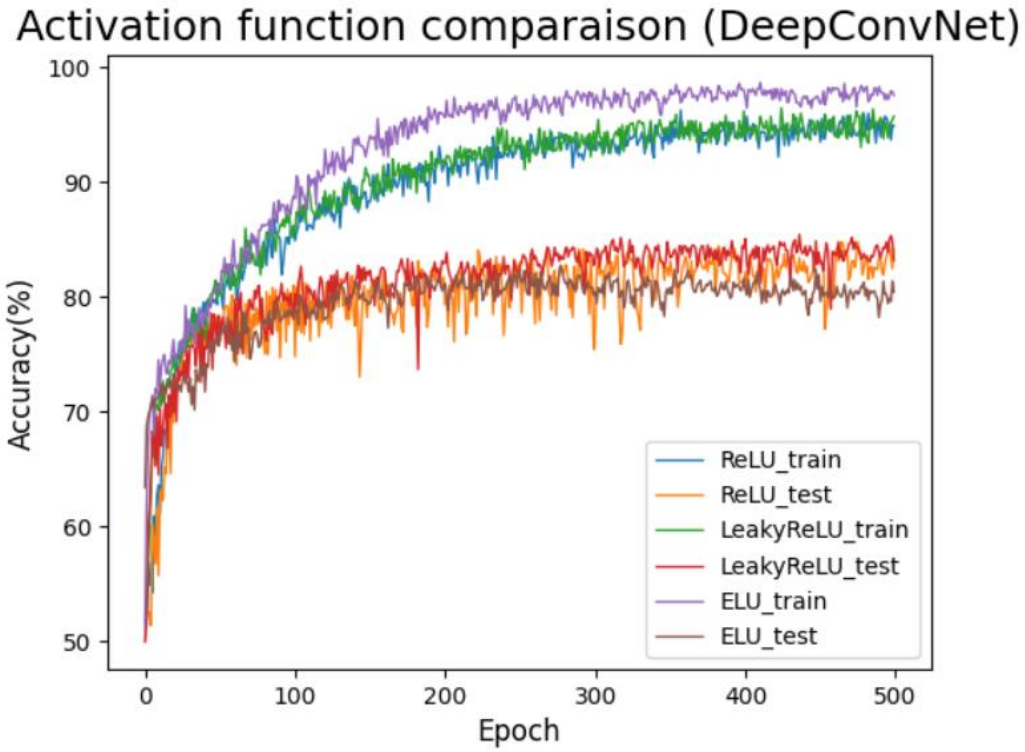
Using EEGNet with ReLU activation and EEGNet with LeakyReLU activation both result in testing accuracy exceeding 87%.

**B. Comparison figures**

**EEGNet**

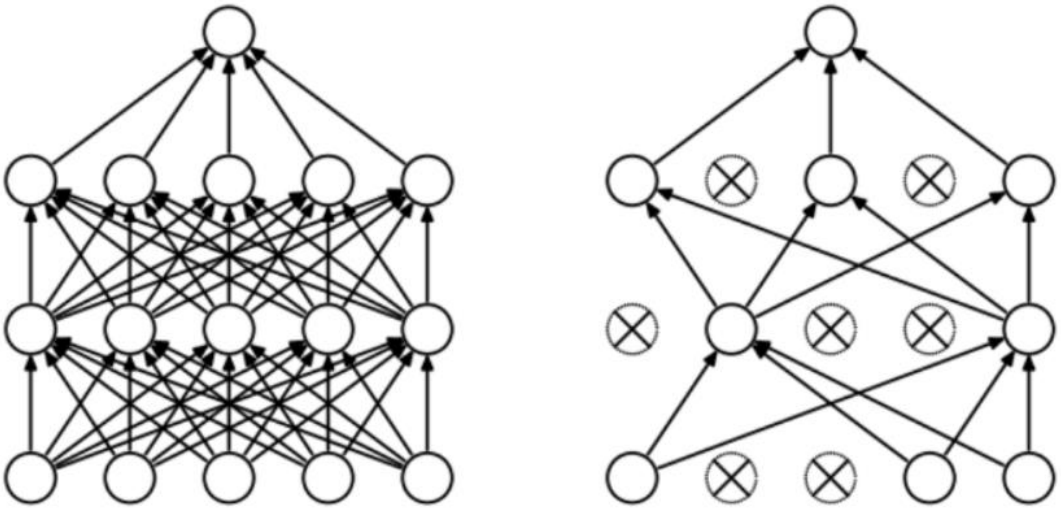



**DeepConvNet**



## 4. Discussion

### A. Dropout



(a) Standard Neural Net (b) After applying dropout.

If we incorporate a Dropout Layer during the training process, the Neural Network depicted in the diagram (a) will transform into diagram (b). When initializing the Dropout Layer, a "probability" is provided, indicating the likelihood that each neuron will "deactivate." By "deactivate," we mean that regardless of the input passed to the Neural Network, the output (activation) of these neurons will always be zero. When a neuron's output is zero, regardless of the magnitude of the connected weights, the result of their multiplication will always be zero.

Furthermore, during the Backpropagation process, if the input to a weight (the activation from the previous layer) is zero, it will result in a gradient of zero. As a consequence, the weight will not be updated. Therefore, in diagram (b), we represent the deactivated neurons as (×) and remove the lines (weights) connecting them. The entire Neural Network appears as if these neurons do not exist.

When we input the next batch of data into the Neural Network, the Dropout Layer will restore the deactivated neurons and randomly select a new batch of neurons to deactivate. This process is repeated, and in each iteration, a portion of neurons will not contribute to the Neural Network's output, and consequently, the weights connected to them will not be updated.

Dropout is effective in addressing the overfitting problem due to its ensemble-like concept. In Ensemble Learning, multiple models are trained and their predictions are considered together to produce the final prediction, often through averaging or majority voting, which improves the overall performance

of the model.

With Dropout, during each iteration, certain neurons are "deactivated," effectively removing them from the Neural Network. Consequently, each iteration yields a Neural Network with a "different structure." Throughout the training process, it appears as though we are training a single model. However, in reality, it combines various structures of models.

As a result, the output of the Neural Network is not solely that of a single model but rather a synthesis of the outputs from multiple Neural Networks. This ensemble effect enhances the model's generalization capabilities, as it leverages diverse structures to make predictions.

Moreover, Dropout prevents neurons from becoming overly sensitive to specific features, ensuring that the model is not overly reliant on particular attributes. This robustness allows the model to perform well even when the input contains some flaws or noise, further enhancing the model's generalization ability.

**B. Weight Decay**

Weight decay is a regularization method that introduces a penalty term to the loss function during training. This penalty term is determined by multiplying the square of all parameters by a smaller value (λ). Consequently, the modified loss function and the weight updates using gradient descent can be represented as follows:

$$w_i^{t+1} \longleftarrow w_i^t - \eta \cdot \frac{\partial L'}{\partial w_i} \text{ , where } L'(\boldsymbol{W}) = L(\boldsymbol{W}) + \lambda \cdot \sum_{n=1}^{N} (w_n^t)^2$$

$$\Longrightarrow w_i^{t+1} \longleftarrow w_i^t - \eta \cdot (\frac{\partial L}{\partial w_i} + 2\lambda w_i^t)$$

As we can observe from the above, during parameter updates, an additional term of $\eta \cdot 2\lambda w_i^t$ is subtracted. This effectively adds a penalty term to the parameter update process, preventing the model from overly relying on specific weights during fitting. Consequently, it achieves the effect of regularization, imposing constraints on the weights to mitigate overfitting.

Regularization through weight decay serves as a means to avoid overfitting by penalizing large weights during the parameter update process. By doing so, the model's dependence on particular weights is reduced, promoting a more balanced and generalized representation of the data. This constraint on the weights aids in preventing overfitting and contributes to a more robust and accurate model.

**C. Execution time of GPU vs. CPU**

To record the execution time of the model on both GPU and CPU, I utilize torch.cuda.Event() in PyTorch.

**EEGNet + LeakyReLU (GPU)**

```
LeakyReLU has max accuracy 88.14814758300781% at epoch 350
execution time: 33.843511718755
Accuracy of EEGNet with LeakyReLU activation function: 88.14814758300781
```

**EEGNet + LeakyReLU (CPU)**

```
epoch  490 :
trainig accuracy:  tensor(96.3889)   loss:  tensor(0.5567, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.2593)

epoch  500 :
trainig accuracy:  tensor(97.4074)   loss:  tensor(0.5503, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.9074)

LeakyReLU has max accuracy 87.03704071044922% at epoch 213
execution time: 905.8438125s
```

**DeepConvNet + LeakyReLU (GPU)**

```
LeakyReLU has max accuracy 85.27777862548828% at epoch 491
execution time: 49.8872421875s
Accuracy of DeepConvNet with LeakyReLU activation function: 85.27777862548828
```

**DeepConvNet + LeakyReLU (CPU)**

```
epoch  490 :
trainig accuracy:  tensor(94.7222)   loss:  tensor(0.6814, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.1667)

epoch  500 :
trainig accuracy:  tensor(94.4444)   loss:  tensor(0.7221, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.7222)

LeakyReLU has max accuracy 85.18518829345703% at epoch 392
execution time: 2097.917s
```

As evident from the comparison chart above, the execution time of the model can vary significantly depending on whether the GPU is used or not. This highlights the substantial acceleration achieved through parallel computing on the GPU in the context of CNNs.

## 5. Extra

### A. Implement any other classification model

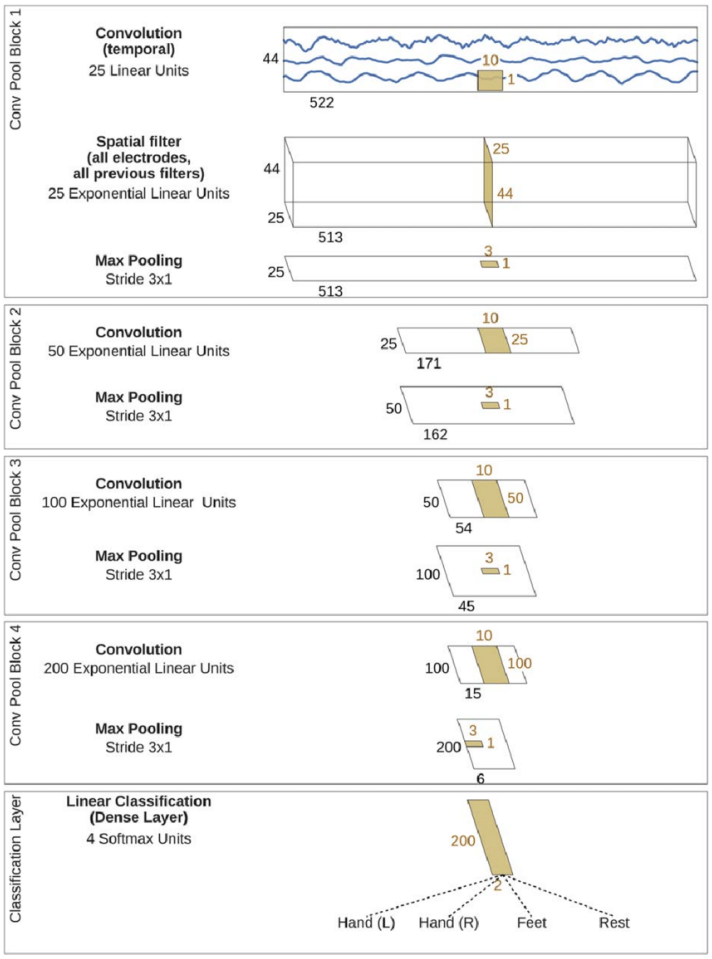
Compared to DeepConvNet, there is a simpler model that can achieve similar accuracy in EEG classification while significantly reducing computational complexity, known as ShallowConvNet.

DeepConvNet consists of five convolutional layers along with normalization, pooling, and dropout layers at each stage. On the other hand, ShallowConvNet comprises only two convolutional layers with normalization and pooling layers, resulting in a much smaller computational workload.

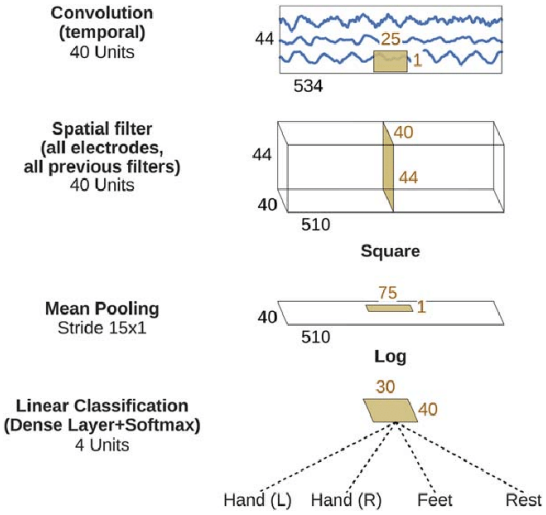
Below is a comparison chart between DeepConvNet and ShallowConvNet:

| | DeepConvNet | ShallowConvNet |
|---|---|---|
| Activation functions | ELU | Square, ReLU |
| Pooling mode | Max | Mean |
| Regularization and intermediate normalization | Dropout + batch normalization + a new tied loss function (explanations see text) | Only batch normalization, only dropout, neither of both, nor tied loss |
| Factorized temporal convolutions | One 10 × 1 convolution per convolutional layer | Two 6 × 1 convolutions per convolutional layer |
| Splitted vs one-step convolution | Splitted convolution in first layer (see the section "Deep ConvNet for raw EEG signals") | One-step convolution in first layer |

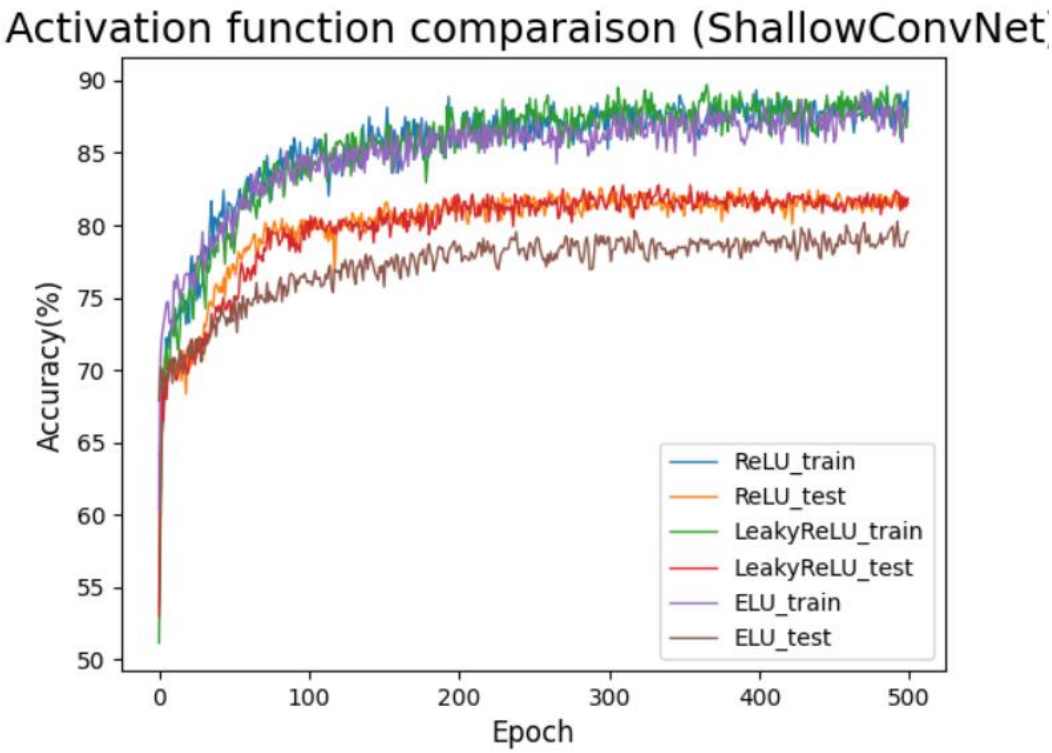
## DeepConvNet




## ShallowConvNet



learning rate = 0.001
weight decay = 0.01
epochs = 500




Comparison between DeepConvNet and ShallowConvNet:

**DeepConvNet**

```
ReLU has max accuracy 84.72222137451172% at epoch 435
Accuracy of DeepConvNet with ReLU activation function: 84.72222137451172
```

```
LeakyReLU has max accuracy 85.55555725097656% at epoch 464
Accuracy of DeepConvNet with LeakyReLU activation function: 85.55555725097656
```

```
ELU has max accuracy 82.96295928955078% at epoch 447
Accuracy of DeepConvNet with ELU activation function: 82.96295928955078
```

**ShallowConvNet**

```
ReLU has max accuracy 82.6851806640625% at epoch 267
Accuracy of ShallowConvNet with ReLU activation function: 82.6851806640625
```

```
LeakyReLU has max accuracy 83.05555725097656% at epoch 337
Accuracy of ShallowConvNet with LeakyReLU activation function: 83.05555725097656
```

```
ELU has max accuracy 82.22222137451172% at epoch 335
Accuracy of ShallowConvNet with ELU activation function: 82.22222137451172
```

Testing accuracy comparison:

| | **ReLU** | **LeakyReLU** | **ELU** |
|---|---|---|---|
| **DeepConvNet** | 84.72% | 85.55% | 82.96% |
| **ShallowConvNet** | 82.68% | 82.05% | 82.22% |

**DeepConvNet (GPU)**

```
ReLU has max accuracy 84.72222137451172% at epoch 437
execution time: 50.72158984375s
Accuracy of DeepConvNet with ReLU activation function: 84.72222137451172
```

```
LeakyReLU has max accuracy 85.27777862548828% at epoch 491
execution time: 49.8872421875s
Accuracy of DeepConvNet with LeakyReLU activation function: 85.27777862548828
```

```
ELU has max accuracy 81.48148345947266% at epoch 99
execution time: 49.72224609375s
Accuracy of DeepConvNet with ELU activation function: 81.48148345947266
```

**ShallowConvNet (GPU)**

```
ReLU has max accuracy 83.24073791503906% at epoch 271
execution time: 31.44002734375s
Accuracy of ShallowConvNet with ReLU activation function: 83.24073791503906
```

```
LeakyReLU has max accuracy 83.05555725097656% at epoch 213
execution time: 27.2671171875s
Accuracy of ShallowConvNet with LeakyReLU activation function: 83.05555725097656
```

```
ELU has max accuracy 81.66666412353516% at epoch 238
execution time: 31.51015234375s
Accuracy of ShallowConvNet with ELU activation function: 81.66666412353516
```

Model execution time comparison:

| | **ReLU** | **LeakyReLU** | **ELU** |
|---|---|---|---|
| **DeepConvNet** | 50.72s | 49.88s | 49.72s |
| **ShallowConvNet** | 31.44s | 27.26s | 31.51s |

From the comparison chart above, it is evident that while ShallowConvNet does not achieve a higher accuracy than DeepConvNet, it comes remarkably close (with a difference of approximately 2%-3%). Additionally, the model execution time is reduced by nearly half, indicating that ShallowConvNet can train a decent model within a limited timeframe. This makes it well-suited for real-time applications, where quick and efficient processing is crucial.

Reference: Deep learning with convolutional neural networks for EEG decoding and visualization (**https://doi.org/10.1002/hbm.23730**)