Lab2 : EEG Classification

contributed by < `K12010Kai` >

Introduction

Target

CNNs

- In this lab, we need to implement **Convolutional Neural Networks**(LeCun et al.,1998) on **DNNs** using by PyTorch framework.

- Deep Learning, GoodFellow et al., MIT Press, 2016. Section 9. CNNs are defined as follow :

  Convolutional networks are simply neural networks that **use convolution** in place of general matrix multiplication in at least one of their layers.

Convolution Operations

- If we conduct convolution operation on input image I with kernel K, the output pixel $S(i,j)$ can be computed by :

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I[m,n]K[i-m,j-n]$$

or

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I[i-m,j-n]K[m,n]$$

, and the latter formula can be viewed as **kernel flipping** operations

- However, the reason that we use convolution operations is to obtain the **commutative property** with "kernel flip". In most CNN implementation, the **cross-correlation** operations are used instead of convolutions:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I[i+m,j+n]K[m,n]$$

EEGNet & DeepConvNet

- The CNNs we are going to implement are **EEGNet** & **DeepConvNet**. Their structures are shown below:
  - EEGNet

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 16, 1, 750]             32
       BatchNorm2d-2            [-1, 16, 1, 750]             32
            Conv2d-3            [-1, 32, 2, 750]            512
       BatchNorm2d-4            [-1, 32, 2, 750]             64
            ELU-5               [-1, 32, 2, 750]              0
         AvgPool2d-6            [-1, 32, 1, 187]              0
           Dropout-7            [-1, 32, 1, 187]              0
            Conv2d-8            [-1, 32, 1, 187]            512
       BatchNorm2d-9            [-1, 32, 1, 187]             64
            ELU-10              [-1, 32, 1, 187]              0
         AvgPool2d-11           [-1, 32, 1, 23]               0
           Dropout-12           [-1, 32, 1, 23]               0
           Linear-13            [-1, 2]                    1474
================================================================
Total params: 2,682
Trainable params: 2,682
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.71
Params size (MB): 0.01
Estimated Total Size (MB): 0.73
----------------------------------------------------------------
```

  - DeepConvNet

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 25, 2, 750]             25
            Conv2d-2            [-1, 25, 1, 750]            625
       BatchNorm2d-3            [-1, 25, 1, 750]             50
         MaxPool2d-4            [-1, 25, 1, 187]              0
           Dropout-5            [-1, 25, 1, 187]              0
            Conv2d-6            [-1, 50, 1, 187]          6,250
       BatchNorm2d-7            [-1, 50, 1, 187]            100
         MaxPool2d-8            [-1, 50, 1, 47]               0
           Dropout-9            [-1, 50, 1, 47]               0
           Conv2d-10            [-1, 100, 1, 47]          25,100
      BatchNorm2d-11            [-1, 100, 1, 47]            200
         MaxPool2d-12           [-1, 100, 1, 11]              0
          Dropout-13            [-1, 100, 1, 11]              0
           Conv2d-14            [-1, 200, 1, 11]          50,200
      BatchNorm2d-15            [-1, 200, 1, 11]            400
         MaxPool2d-16           [-1, 200, 1, 2]               0
          Dropout-17            [-1, 200, 1, 2]               0
           Linear-18            [-1, 2]                    1,202
================================================================
Total params: 84,152
Trainable params: 84,152
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.14
Params size (MB): 0.32
Estimated Total Size (MB): 2.47
----------------------------------------------------------------
```

Experiment Setup

- Use PyTorch framework to implement

EEGNet

DeepConvNet

Activation Functions

ReLU

- **Rectified Linear Unit**

- Definition:

$$Rectifier(x) = \max(0, x)$$

- Properties[1]:
  - Its cheap to compute:
    - The gradient is constrained
      - 1st Direction : 1 when the "neuron" is **active** and 0 instead
      - 2nd direction : 0 almost everywhere
  - It converges faster:
    - Since it's a "linear" units, it will not **saturate** when input it gets large
      - Saturation of an activation function means that the gradient of the function **vanishes** when the value approaches $+\infty$ or $-\infty$
  - **Its sparsely activated**:
    - Since ReLU is 0 for all negative inputs, it's likely for any given unit to **not activate at all**. This is often desirable
    - When the network works, not all neurons active simultaneously. It's more likely that neurons are actually processing **meaningful aspects** of the problem
    - For example, in a model detecting cats in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is about a building.
- Problems of ReLU
  - Dying ReLU
    - If ReLU neuron is "Dead" if it's stuck in the negative side and always outputs 0
    - The dying problem is likely to occur when **learning rate is too high** or there is a large negative bias
  - Not zero-centered
  - Non-differentiable at 0

Leaky ReLU

- Variant of ReLU
- Definition:

$$LeakyReLU(x) = \max(0, x) + \min(0, x) \cdot (exp(x) - 1)$$

- Properties
  - Its gradient vanishing problem
  - According to the paper(Clevert et al., 2016)[2]:

    ELUs minimize the vanishing gradient effect as rectified linear units (ReLUs) and leaky ReLUs do. The corresponding gradient problem is alleviated because the positive part of these functions is the identity, therefore their derivative is one and not contractive.

Experimental Results

Experiment Parameters

- Batch Size: `64`
- Training Epochs: `300`
- Activation Functions: `ReLU` / `Leaky ReLU`
- Optimizer: `torch.nn.optim.SGD`
  - Learning Rate: `1e-2`
  - Weight Decay: `1e-2`
  - Momentum: `0.9`
- Scheduler: `torch.optim.lr_scheduler.ReduceLROnPlateau`

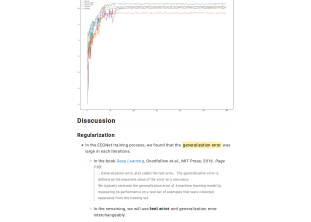
Accuracy

EEGNet

```
Test Accuracy:
- - - - - - - - - - - - - - - - - - - - - -
With ELU              : 85.4814814814814%
With Leaky ReLU       : 88.6555555555556%
With ReLU             : 87.1481481481481%
```

DeepConvNet

```
Test Accuracy:
- - - - - - - - - - - - - - - - - - - - - -
With ELU              : 80.3703703703703%
With Leaky ReLU       : 79.5185185185185%
With ReLU             : 80.0925925925926%
```



EEGNet

DeepConvNet

Discussion

Regularization

- In the EEGNet training process, we found that the **generalization error** was large in each iterations.
  - In the book Deep Learning, GoodFellow et al., MIT Press, 2016. Page 110:

    Generalization error, also called the test error. The generalization error is defined as the expected value of the error on a new input.

    We typically estimate the generalization error of a machine learning model by measuring its performance on a test set of examples that were collected separately from the training set.

  - In the remaining, we will use **test error** and generalization error interchangeably.
- The **Regularization** techniques aim to reducing the training error and testing error :
  - As in the book Deep Learning, GoodFellow et al., MIT Press, 2016. Section 8.2.2

    Regularization is defined as any modification we make to a learning algorithm that is intended to **reduce its generalization error but not its training error**

- We use weight-decay when using optimizers from `torch.nn.optim`
- We conduct experiments on the regularization term
  - Experiment Parameters
    - Model: EEGNet
    - Activation functions: `Leaky ReLU`
    - Optimizer: SGD
      - **Weight Decay**: `1e-2`
  - Result:

```
Test Accuracy:
- - - - - - - - - - - - - - - - - - - - - -
With weight-decay     : 80.6555555555556%
Without weight-decay  : 81.7592592592592%
```

  - Experiment Conclusion :
    - We can see that the effect of regularization is significant
  - Note:
    - By the way, the value tuning of weight-decay is also important

Optimizer

- Different optimizer has different learning strategy
- So the choices of optimizer is also important
- We compare SGD, Adam, Adagrad in an experiment
  - Control Setting
    - Model: EEGNet
    - Activation Function: `Leaky ReLU`
  - Individual Parameters:
    - SGD
      - Learning Rate : `1e-2`
      - Weight Decay : `1e-2`
      - Momentum : `0.9`
    - Adam
      - Learning Rate : `1e-2`
      - Weight Decay : `1e-2`
    - Adagrad
      - Learning Rate : `1e-2`
      - Weight Decay : `1e-2`
  - Result:

```
Test Accuracy:
- - - - - - - - - - - - - - - - - - - - - -
With SGD              : 87.4074074074074%
With Adam             : 85.1111111111111%
With Adagrad          : 79.2592592592592%
```

  - We can see that, using different optimizer can lead to different performance, and SGD is classic though powerful.

Reference

- Deep Learning, GoodFellow et al., MIT Press, 2016.
- https://pytorch.org/docs/
- https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7
- https://zh.wikipedia.org/wiki/Regularization_(mathematics)

1. https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7 ↩

2. https://arxiv.org/abs/1511.07289 ↩