


DLP HW2 EEG Classification

311551044 徐煜倫

DLP HW2 EEG Classification

Introduction

 <https://dog-tip-98e.notion.site/DLP-HW2-EEG-Classification-427d2ac0d669439eac64c930ec9cae2d?pvs=4>

1. Introduction

2. Experiment Setup

EEGNet

Spec

Code

Explain

Special Design

DeepConvNet

Spec

Code

Explain

2B. Explain The Activation functions

3. Experimental Results

The highest testing accuracy

EEGNet

DeepConvNet

Comparison figures

4. Discussion

1. Introduction

The primary objective of this lab is to develop and implement EEG classification models, specifically EEGNet and DeepConvNet, using the BCI (Brain-Computer Interface) competition dataset. The focus of the experiment is to explore the impact of different activation functions, including ReLU, Leaky ReLU, and ELU, on the performance of these models.

2. Experiment Setup

EEGNet

Spec

Following the spec to construct the model.

```
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)  
  )  
)
```

Code

```

1 class EEGNet(nn.Module):
2     func_map = {
3         "relu": nn.ReLU,
4         "leaky_relu": nn.LeakyReLU,
5         "elu": nn.ELU,
6     }
7
8     def __init__(self, act="elu", dropout: float = 0.5, **kwargs):
9         super(EEGNet, self).__init__()
10
11         act_func = self.func_map[act](**kwargs)
12         self.first_conv = Sequential(
13             Conv2d(
14                 in_channels=1,
15                 out_channels=16,
16                 kernel_size=(1, 51),
17                 stride=(1, 1),
18                 padding=(0, 25),
19                 bias=False,
20             ),
21             BatchNorm2d(
22                 num_features=16,
23                 eps=1e-05,
24                 momentum=0.1,
25                 affine=True,
26                 track_running_stats=True,
27             ),
28         )
29         self.depthwise_conv = Sequential(
30             Conv2d(
31                 in_channels=16,
32                 out_channels=32,
33                 kernel_size=(2, 1),
34                 stride=(1, 1),
35                 groups=16,
36                 bias=False,
37             ),
38             BatchNorm2d(
39                 num_features=32,
40                 eps=1e-05,
41                 momentum=0.1,
42                 affine=True,
43                 track_running_stats=True,
44             ),
45             act_func,
46             AvgPool2d(
47                 kernel_size=(1, 4),
48                 stride=(1, 4),
49                 padding=0,
50             ),
51             Dropout(p=dropout, inplace=True),
52         )
53         self.separable_conv = Sequential(
54             Conv2d(
55                 in_channels=32,
56                 out_channels=32,
57                 kernel_size=(3, 15),
58                 stride=(1, 1),
59                 padding=(0, 7),
60                 bias=False,
61             ),
62             BatchNorm2d(
63                 num_features=32,
64                 eps=1e-05,
65                 momentum=0.1,
66                 affine=True,
67                 track_running_stats=True,
68             ),
69             act_func,
70             AvgPool2d(
71                 kernel_size=(1, 8),
72                 stride=(1, 8),
73                 padding=0,
74             ),
75             Dropout(p=dropout, inplace=True),
76         )
77         self.classify = Sequential(
78             Linear(
79                 in_features=736,
80                 out_features=2,
81                 bias=True,
82             ),
83         )
84
85     def forward(self, x: Tensor) -> dict:
86         x = self.first_conv(x)
87         x = self.depthwise_conv(x)
88         x = self.separable_conv(x)
89         x = x.view(x.shape[0], -1)
90         x = self.classify(x)
91         return x
92

```

Explain

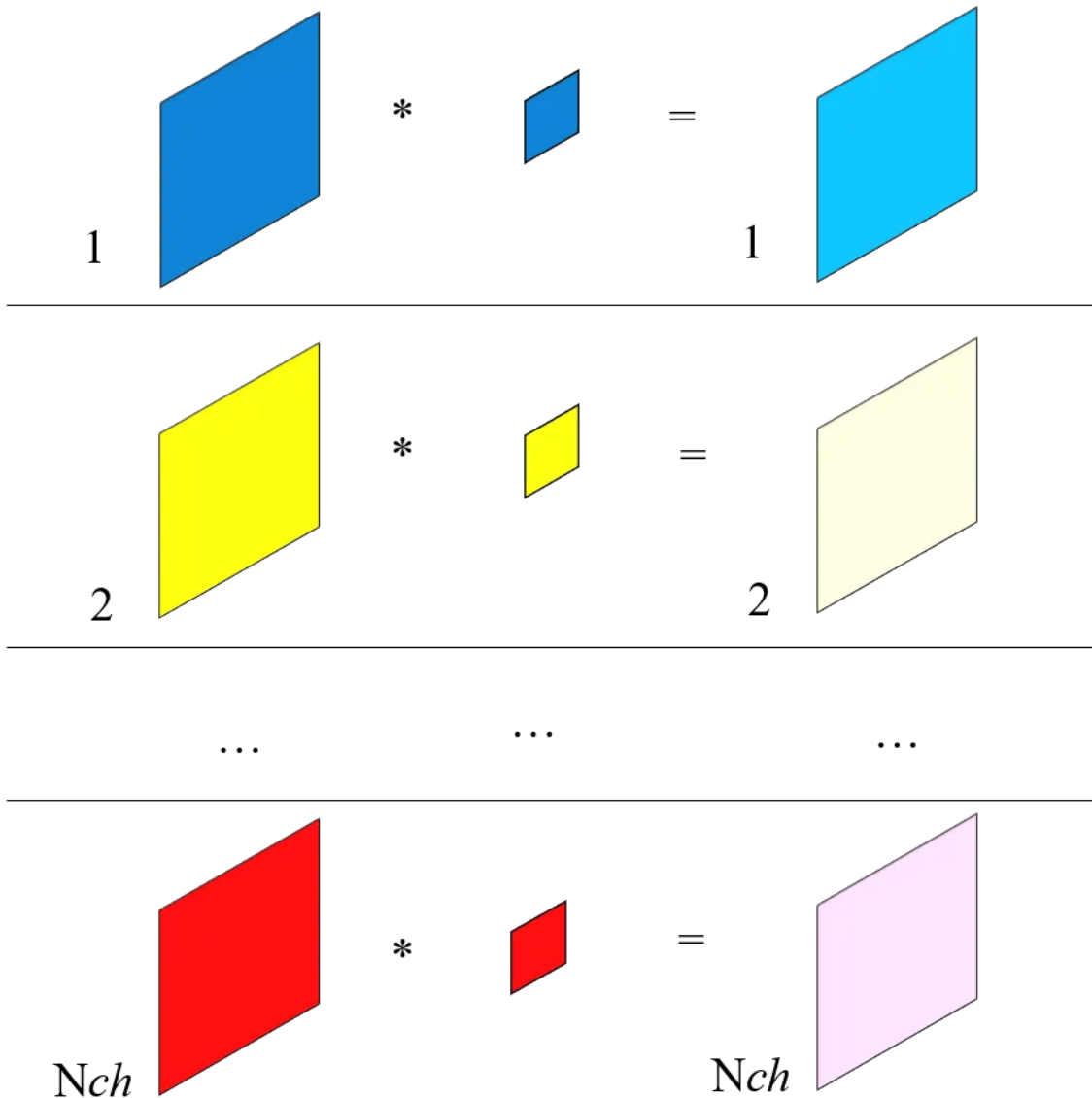
It is a PyTorch `nn.Module` called `EEGNet`. This is a convolutional neural network (CNN) model that's specifically designed for the task of electroencephalogram (EEG) signal classification.

Here are the main components of this network:

1. `act_func` : It's a dictionary to map the names of activation functions to their actual implementations in PyTorch. This is done to easily switch between different activation functions during initialization.
2. `first_conv` : This is the first convolutional layer, which applies a convolutional operation on the input data. The operation is followed by a 2D batch normalization which normalizes the activations of the convolutional layer.
3. `depthwise_conv` : This is the second block of the network which contains a depthwise convolutional layer. A depthwise convolution is a variant of the standard convolution, where each input channel is convolved with its own set of filters, as opposed to the standard convolution where each input channel is convolved with all filters. This is followed by batch normalization, an activation function (determined by `act_func`), average pooling, and dropout.
4. `separable_conv` : This is the third block, which contains a depthwise separable convolution. Depthwise separable convolutions consist of performing just the depthwise convolution followed by a pointwise convolution, i.e., a regular convolution with a 1x1 kernel. This sequence further reduces the computation and model size. This block also includes batch normalization, the activation function, average pooling, and dropout.
5. `classify` : This is the final linear layer used for classification. The number of out_features (2 in this case) represents the number of classes for classification.
6. `forward()` : This function defines the forward pass of the network, describing how an input tensor is transformed into the output tensor. Each of the defined blocks (`first_conv` , `depthwise_conv` , `separable_conv` , `classify`) is applied sequentially to the input tensor. The output tensor of one block is used as the input tensor to the next.

Special Design

輸入資料 $(W_{in} * H_{in} * Nch)$ Nch 個Kernel $(k * k)$ Depthwise_out $(W_{out} * H_{out} * Nch)$



source: <https://chih-sheng-huang821.medium.com/深度學習-mobilenet-depthwise-separable-convolution-f1ed016b3467>

For each channel of the input data, a $k*k$ kernel is created, and then each channel undergoes convolution separately with its corresponding kernel.

This step is different from the typical convolution process, where each kernel map is convolved with all channels. Here, the convolution is performed independently and separately for each channel and its corresponding kernel.

DeepConvNet

Spec

Following the spec to construct the model. It is worthy to mention that the last activation (softmax) does not need to implement, since we are using cross entropy for loss function.

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

Code

```

1  class DeepConvNet(nn.Module):
2      func_map = {
3          "relu": nn.ReLU,
4          "leaky_relu": nn.LeakyReLU,
5          "elu": nn.ELU,
6      }
7
8      def __init__(self, act: str = "elu", dropout: float = 0.5, **kwargs) -> None:
9          super(DeepConvNet, self).__init__()
10         act_func = self.func_map[act]()
11
12         channels = [25, 25, 50, 100, 200]
13         kernels = [(2, 1), (1, 5), (1, 5), (1, 5)]
14
15         self.convs = nn.ModuleList()
16         self.convs.append(Conv2d(1, 25, kernel_size=(1, 5)))
17
18         for i in range(len(channels) - 1):
19             conv = Sequential(
20                 Conv2d(
21                     in_channels=channels[i],
22                     out_channels=channels[i + 1],
23                     kernel_size=kernels[i],
24                     stride=1,
25                     padding=0,
26                 ),
27                 BatchNorm2d(num_features=channels[i + 1], eps=1e-5, momentum=0.1),
28                 act_func,
29                 MaxPool2d(kernel_size=(1, 2)),
30                 Dropout(p=dropout),
31             )
32             self.convs.append(conv)
33
34         self.classify = Sequential(
35             Linear(in_features=8600, out_features=2),
36         )
37
38     def forward(self, x: Tensor):
39         for conv in self.convs:
40             x = conv(x)
41         x = x.view(x.shape[0], -1)
42         x = self.classify(x)
43         return x

```

This is my code, I really like the modulelist design IMAO



```
1 loss = cross_entropy(output, target)
```

We don't need to add softmax in the classify block, since `cross_entropy` calculate this for us.

Explain

This model is defined as a Deep Convolutional Neural Network (ConvNet) in PyTorch. Let's explain the key components of this model layer by layer:

1. Activation Functions:

- The class `DeepConvNet` defines three activation functions: ReLU, Leaky ReLU, and ELU (Exponential Linear Unit). These activation functions introduce non-linearity in the network, allowing it to learn complex patterns and relationships from the input data.

2. Convolutional Layers:

- The model uses a series of convolutional layers to extract meaningful features from the input data. Each convolutional layer applies a set of learnable filters (kernels) to the input data, performing a convolution operation to produce feature maps. The number of channels in each layer gradually increases to capture more complex features.
- The `self.convs` variable is a `ModuleList` that stores all the convolutional layers.

3. Batch Normalization:

- Batch normalization is applied after each convolutional layer. It normalizes the output of the previous layer, helping to stabilize and speed up the training process. It is used to avoid internal covariate shift and improve the learning process.

4. Pooling Layers:

- The model utilizes max-pooling layers with a kernel size of (1, 2) after each convolutional layer (except the last one). Max-pooling downsamples the spatial dimensions of the feature maps while retaining the most important information, reducing the computational load and improving translation invariance.

5. Dropout:

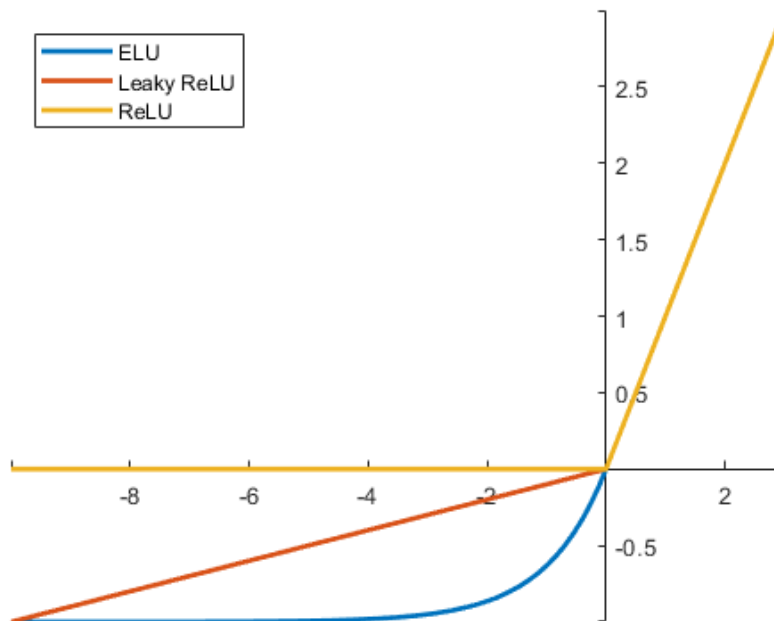
- Dropout is used as a regularization technique to prevent overfitting. It randomly sets a fraction of the input units to zero during training, forcing the network to learn more robust and redundant representations.

6. Fully Connected Layer (Classification Layer):

- After passing through the convolutional layers, the output is flattened to a 1D tensor, suitable for feeding into a fully connected (linear) layer.
- The `self.classify` layer is a fully connected layer that takes the flattened features and maps them to the final output classes. In this case, it projects the input to a 2-dimensional space representing the logits for binary classification.

In summary, this Deep ConvNet consists of a series of convolutional layers, each followed by batch normalization, activation function, max-pooling, and dropout. The output from the convolutional layers is then flattened and fed into a fully connected layer for final classification into two classes. The choice of activation function, number of channels, kernel sizes, and dropout rate are configurable parameters when creating an instance of this model.

2B. Explain The Activation functions



source:

<https://www.researchgate.net/publication/334389306/figure/fig8/AS:779352161677313@1562823443351/Illustration-of-output-of-ELU-vs-ReLU-vs-Leaky-ReLU-function-with-varying-input-values.ppm>

1. ReLU (Rectified Linear Activation):

- Function: ReLU is an activation function commonly used in neural networks.
- Purpose: It introduces non-linearity to the model, allowing it to learn complex patterns and make better predictions. ReLU sets all negative input values to zero and passes all positive input values as they are.
- Mathematically: $ReLU(x) = \max(0, x)$

2. Leaky ReLU (Leaky Rectified Linear Activation):

- Function: Leaky ReLU is a variation of the ReLU activation function.
- Purpose: To address the "dying ReLU" problem, which occurs when neurons output zero for all inputs and stop learning during training.
- Functionality: Leaky ReLU sets negative input values to a small, non-zero slope (alpha times the input), allowing a small gradient to flow through the neurons even when the input is negative.
- Mathematically: $LeakyReLU(x) = \max(\alpha * x, x)$, where alpha is a small positive constant.

3. ELU (Exponential Linear Unit):

- Function: ELU is another variant of the ReLU activation function.
- Purpose: To alleviate the vanishing gradient problem and provide better learning capability.
- Functionality: ELU maintains negative values, but with a smooth curve, which helps prevent dead neurons and improves model performance.
- Mathematically: $ELU(x) = x$ for $x \geq 0$, and $ELU(x) = \alpha * (\exp(x) - 1)$ for $x < 0$, where alpha is a small positive constant.

Each of these activation functions has its advantages and is used in different scenarios based on the nature of the data and the neural network architecture. Experimenting with different activation functions is common in deep learning to find the one that suits a particular problem best.

3. Experimental Results

The highest testing accuracy

EEGNet

```
(venv) alankindom@alankindom:~/桌面/repo/DLP/lab2$ python3 main.py --mode eval --checkpoint best-leaky_relu-EEGNet.pth --model EEGNet --act leaky_relu
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): LeakyReLU(negative_slope=0.01)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.5, inplace=True)
  )
  (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): LeakyReLU(negative_slope=0.01)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.5, inplace=True)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
Test Loss: 0.0015, Test Accuracy: 0.8731
```

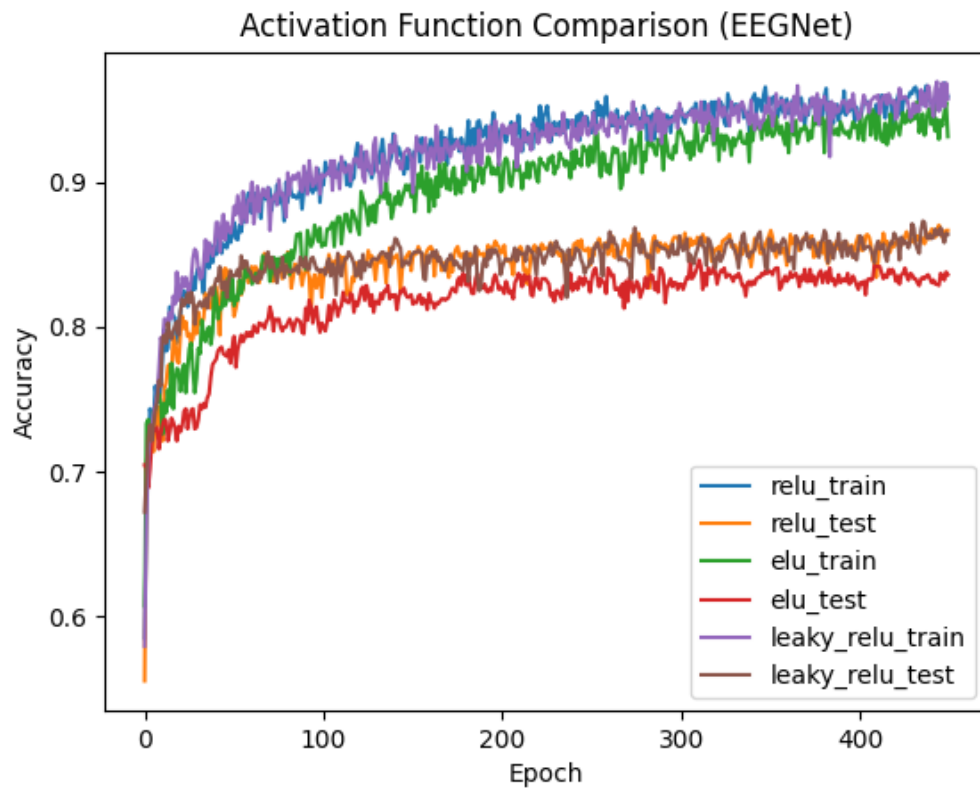
DeepConvNet

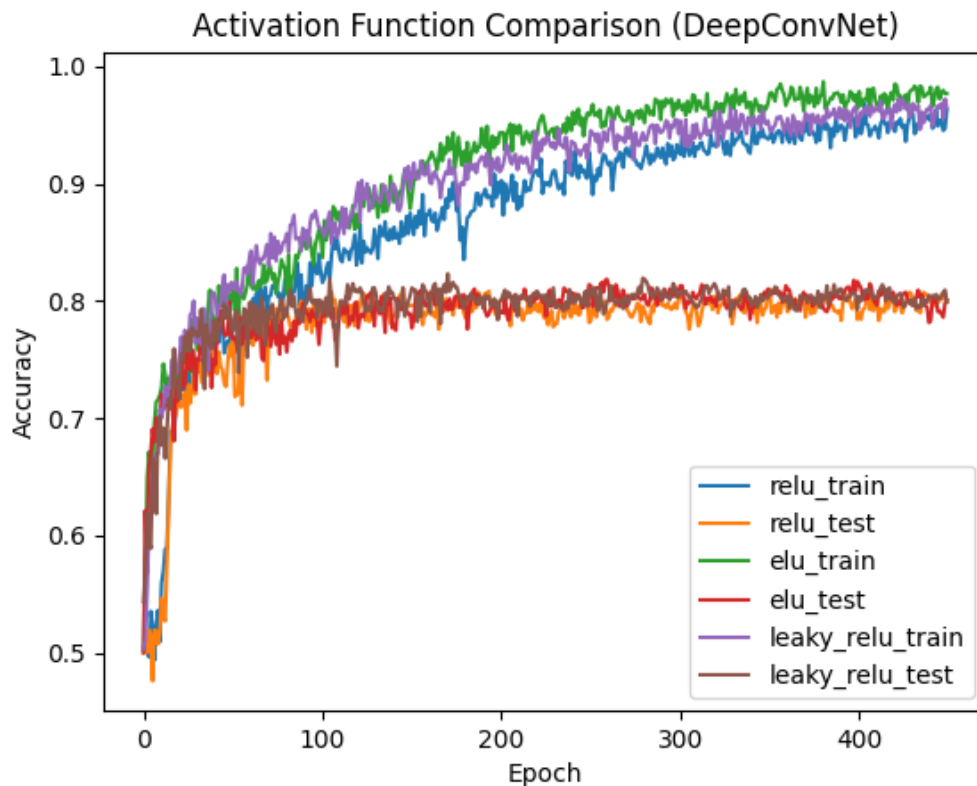
```

● (venv) alankindom@alankindom:~/桌面/repo/DLP/lab2$ python3 main.py --mode eval --checkpoint best-leaky_relu-DeepConvNet.pth --model DeepConvNet --act
leaky_relu
DeepConvNet(
  (convs): ModuleList(
    (0): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
    (1): Sequential(
      (0): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1))
      (1): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01)
      (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(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): LeakyReLU(negative_slope=0.01)
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
    (3): 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): LeakyReLU(negative_slope=0.01)
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
    (4): 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): LeakyReLU(negative_slope=0.01)
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.5, inplace=False)
    )
  )
  (classify): Linear(in_features=8600, out_features=2, bias=True)
)
Test Loss: 0.0021, Test Accuracy: 0.8231

```


Comparison figures





4. Discussion

These code snippets are essential for ensuring reproducibility in machine learning experiments and data processing pipelines. Reproducibility is crucial in research to verify results, compare models fairly, and identify potential issues in the code or data. By setting consistent random seeds, you can make sure that the random aspects of the program are controlled and remain the same across different runs.



```

1  def set_seed(seed: int = 890104):
2      torch.backends.cudnn.deterministic = True
3      torch.backends.cudnn.benchmark = False
4      torch.manual_seed(seed)
5      torch.cuda.manual_seed_all(seed)
6      np.random.seed(seed)
7      random.seed(seed)
8
9
10 def seed_worker(worker_id):
11     worker_seed = torch.initial_seed() % 2**32
12     np.random.seed(worker_seed)
13     random.seed(worker_seed)

```

These two code snippets are related to setting random seeds in a Python script using PyTorch and NumPy libraries. Controlling random seeds is crucial for ensuring reproducibility in machine learning or scientific experiments that involve random processes. By setting the same random seed, you can achieve consistent results across different runs, which is essential for debugging, comparing models, and ensuring research results are replicable.



890104 is my birthday

1. `set_seed(seed: int = 890104)`: This function sets the random seed for PyTorch, NumPy, and Python's built-in `random` module. It takes an optional `seed` parameter, which allows the user to specify the desired random seed. If no seed is provided, it defaults to `890104`.

Explanation of each line in the function:

- `torch.backends.cudnn.deterministic = True`: This line ensures that the CuDNN backend (used by PyTorch for GPU acceleration) operates deterministically. It disables some non-deterministic

algorithms that might produce slightly different results on different GPU runs, making the results reproducible.

- `torch.backends.cudnn.benchmark = False` : This line disables CuDNN's automatic benchmarking, which otherwise dynamically finds the best algorithms for the input size, leading to non-reproducible results. By setting it to `False`, you ensure that the same algorithms are used for each run.
 - `torch.manual_seed(seed)` : This sets the random seed for the PyTorch library, affecting operations that rely on random number generation, such as initializing model weights, data shuffling, and dropout layers.
 - `torch.cuda.manual_seed_all(seed)` : This sets the random seed for all available CUDA devices, i.e., GPUs. It ensures reproducibility when using GPU acceleration with PyTorch.
 - `np.random.seed(seed)` : This sets the random seed for the NumPy library. NumPy is often used for numerical computations in Python, and it's essential to control its randomness when combined with PyTorch.
 - `random.seed(seed)` : This sets the random seed for Python's built-in `random` module, which is used in various Python functions. By setting this seed, you ensure consistency in results involving random processes not covered by PyTorch or NumPy.
1. `seed_worker(worker_id)` : This function is used in a multi-worker setup, such as when employing multiple processes for data loading and parallel training in PyTorch. It sets the random seed for each worker process to avoid conflicts and ensure reproducibility in multi-process scenarios.

Explanation of the function:

- `torch.initial_seed() % 2**32` : PyTorch's `initial_seed()` function returns a random seed for the current PyTorch process. The `% 2**32` operation ensures that the seed is within the valid range for seeding NumPy.
- `np.random.seed(worker_seed)` : This sets the random seed for the NumPy library in the worker process, ensuring that each worker has its own unique random seed.
- `random.seed(worker_seed)` : This sets the random seed for Python's built-in `random` module in the worker process, providing each worker with a unique random seed as well.