Lab5A - Constructing a CNN Network (Guide)

For spatial data for example image or video data, Convolutional Neural Network (CNN or ConvNet) performs much better than standard neural network. In this practical, we shall learn how to build a CNN Network.

Objectives:

1. Learn how to build a convolutional neural network (CNN)
2. Learn how to build a network or layer using `sequential`

Remember to **enable the GPU** (Edit > Notebook setting > GPU) to ensure short training time.

In [1]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=True).
```

In [ ]:
```
!pip install torchinfo
```

In [2]:
```
cd "/content/gdrive/MyDrive/UCCD3074_Labs/UCCD3074_Lab5"
```

```
/content/gdrive/MyDrive/UCCD3074_Labs/UCCD3074_Lab5
```

Import the required libraries

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F

import torchvision
import torchvision.transforms as transforms

import torch.optim as optim
from torch.utils.data import DataLoader

from torchinfo import summary
```

```
from cifar10 import CIFAR10

%load_ext autoreload
%autoreload 2
```

SECTION 1. DEFINING A CNN MODULE WITH `torch.nn.Module`

In this section, we create a CNN network using `nn.Module`. The `Module` is the main building block, it defines the base class for all neural network and you MUST subclass it.

1.1 Build the network

Exercise. Build the following CNN. You will need to following modules:

- To define a conv2d layer: `torch.nn.Conv2d(in_channel, out_channel, kernel_size, stride=1, padding=0)`
    - `in_channel` : number of channels in the input tensor.
    - `out_channel` : number of channels in the output tensor. This is equivalent to the number of filters in the current convolutional layer.
    - `kernel_size` : size of the filter (`f`).
    - `stride` : stride (`s`). Default value is 1.
    - `padding` : padding (`p`). Default value is 0.
- To define a max pooling layer: `torch.nn.functional.max_pool2d (x, kernel_size, stride=None, padding=0)`
    - `x` : input tensor of shape `(b, c, h, w)` . This is required as this is a `functional` operation.
    - `kernel_size` : size of the filter (`f`).
    - `stride` : stride (`s`). Default value is `kernel_size` .
    - `padding` : padding (`p`). Default value is 0.
- To define a linear layer: `torch.nn.Linear (in_features, out_features)`
    - `in_features` : size of each input sample. This is equivalent to the number of units or signals in the previous layer.
    - `out_features` : size of the output sample. This is equivalent to the number of units / neurons in the current layer.
- To define the global average pooling: `torch.mean (x, dim)`

- `x` : the input tensor
- `dim` : the dimensions to reduce. For the input tensor is `(b, c, h, w)` , to compute the mean of the spatial dimensions `h` and `w` , set `dim = [2, 3]` . This will compute the mean for the spatial dimensions and output a tensor of shape `(b, c, 1, 1)` . Then, use `torch.squeeze` to remove the two empty dimensions to get a tensor of shape `(b, c)` .

- Alternatively, the global average pooling can be defined using the following command: `torch.nn.AdaptiveAvgPool2d (output_size)`

  - `output_size` : the target output size (`o`). The layer will configure the kernel size as `(input_size+target_size-1)//target_size` to generate an output tensor of shape `output_size` .

Network Architecture

Layer	Name	Description	OutputShape
-	Input	-	(?, 3, 32, 32)
1	conv1	Conv2d (k=32, f=3, s=1, p=1)	(?, 32, 32, 32)
		relu	(?, 32, 32, 32)
2	conv2	Conv2d (k=32, f=3, s=1, p=1)	(?, 32, 32, 32)
		relu	(?, 32, 32, 32)
	pool1	maxpool (f=2, s =2, p=0)	(?, 32, 16, 16)
3	conv3	Conv2d (k=64, f=3, s=1, p=1)	(?, 64, 16, 16)
		relu	(?, 64, 16, 16)
4	conv4	Conv2d (k=64, f=3, s=1, p=1)	(?, 64, 16, 16)
		relu	(?, 64, 16, 16)
	global_pool	AdaptiveAvgPool (o=(1,1))	(?, 64, 1, 1)
		view	(?, 64)
5	fc1	Linear (#units=10)	(?, 10)

Notes: `k` : number of filters, `f` : filter or kernel size, `s` : stride, `p` : padding, `o` : output shape

In [4]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

In [5]:

```
class Net1(nn.Module):
    def __init__(self):
        # call super constructor
        super().__init__()

        # create the conv1 layer
        self.conv1 = nn.Conv2d(3,  32, kernel_size=3, stride=1, padding=1)

        # create the conv2 layer
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1)

        # create the conv3 layer
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)

        # create the conv4 layer
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)

        # create the global pooling layer
        self.global_pool = nn.AdaptiveAvgPool2d((1,1))

        # fully connected layer
        self.fc1   = nn.Linear(64, 10)

    def forward(self, x):

        # conv1 layer
        x = F.relu(self.conv1(x))

        # conv2 layer
        x = F.relu(self.conv2(x))

        # pooling layer
        x = F.max_pool2d(x, 2, 2)

        # conv3 layer
        x = F.relu(self.conv3(x))

        # conv4 layer
        x = F.relu(self.conv4(x))

        # global pooling
        # x = torch.mean(x, [2, 3])
        x = self.global_pool(x)

        # remove the spatial dimension
```

```python
        x = x.squeeze()

        # fc1 layer
        x = self.fc1(x)

        return x
```

Create the network and test it

In [6]:
```python
net1 = Net1()
out = net1(torch.randn(4, 3, 32, 32))
```

Display the network

In [7]:
```python
print(net1)
```

```
Net1(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (global_pool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc1): Linear(in_features=64, out_features=10, bias=True)
)
```

In [8]:
```python
summary(net1, input_size=(4, 3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
            Conv2d-2           [-1, 32, 32, 32]           9,248
            Conv2d-3           [-1, 64, 16, 16]          18,496
            Conv2d-4           [-1, 64, 16, 16]          36,928
 AdaptiveAvgPool2d-5             [-1, 64, 1, 1]               0
            Linear-6                   [-1, 10]             650
================================================================
Total params: 66,218
Trainable params: 66,218
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.75
Params size (MB): 0.25
Estimated Total Size (MB): 1.01
----------------------------------------------------------------
```

1.2 Load the dataset

Exercise. Load the dataset

1. Load the dataset. Define the following transformation pipeline to
    - Convert an image (numpy array with range (0, 255)) to a tensor, and
    - Normalize the tensor with mean = 0.5 and std = 1.0

In [9]:

```
# transform the model
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (1., 1., 1.))
])

# Load the dataset
trainset = CIFAR10(train=True,  transform=transform, download=True, num_samples=10000)
testset  = CIFAR10(train=False,  transform=transform, download=True, num_samples=2000)

print('Size of trainset:', len(trainset))
print('Size of testset:', len(testset))
```

```
Files already downloaded and verified
Files already downloaded and verified
Size of trainset: 10000
Size of testset: 2000
```

1. Create the dataloader for train set and test set. Use a batch size of 16, enable shuffle, apply the transformation pipeline defined above and use 2 cpu workers to load the datasets.

In [10]:
```
trainloader = DataLoader(trainset, batch_size=16, shuffle=True, num_workers=2)
testloader  = DataLoader(testset, batch_size=16, shuffle=True, num_workers=2)
```

1.3 Train the model

Exercise. Complete the training function below

In [11]:
```
def train(net, trainloader, num_epochs=15, lr=0.1, momentum=0.9):

    loss_iterations = int(np.ceil(len(trainloader)/3))

    # transfer model to GPU
    net = net.to(device)

    # set the optimizer. Use the SGD optimizer. Use the lr and momentum settings passed by the user
    optimizer = optim.SGD(net.parameters(), lr=lr, momentum=momentum)

    # set to training mode
    net.train()

    # variables
    best_loss = np.inf
    saturate_count = 0

    # train the network
    for e in range(num_epochs):

        running_loss = 0
        running_count = 0

        # for all batch samples
        for i, (inputs, labels) in enumerate(trainloader):

            # Clear all the gradient to zero
```

```python
            optimizer.zero_grad()

            # transfer data to GPU
            inputs = inputs.to(device)
            labels = labels.to(device)

            # forward propagation to get h
            outs = net(inputs)

            # compute loss
            loss = F.cross_entropy(outs, labels)

            # backpropagation to get gradients of all parameters
            loss.backward()

            # update parameters
            optimizer.step()

            # get the loss
            running_loss += loss.item()
            running_count += 1

             # display the averaged loss value
            if i % loss_iterations == loss_iterations-1 or i == len(trainloader) - 1:
                train_loss = running_loss / running_count
                running_loss = 0.
                running_count = 0.
                print(f'[Epoch {e+1:2d} Iter {i+1:5d}/{len(trainloader)}]: train_loss = {train_loss:.4f}')

    print("Training completed.")
```

Now, train the model for 15 epochs.

In [12]:
```python
train(net1, trainloader, num_epochs=15, lr=0.01, momentum=0.9)
```

```
[Epoch  1 Iter   209/625]: train_loss = 2.3033
[Epoch  1 Iter   418/625]: train_loss = 2.2374
[Epoch  1 Iter   625/625]: train_loss = 2.1177
[Epoch  2 Iter   209/625]: train_loss = 2.0776
[Epoch  2 Iter   418/625]: train_loss = 2.0541
[Epoch  2 Iter   625/625]: train_loss = 1.9898
[Epoch  3 Iter   209/625]: train_loss = 1.9296
[Epoch  3 Iter   418/625]: train_loss = 1.8712
[Epoch  3 Iter   625/625]: train_loss = 1.8263
[Epoch  4 Iter   209/625]: train_loss = 1.8003
[Epoch  4 Iter   418/625]: train_loss = 1.7818
[Epoch  4 Iter   625/625]: train_loss = 1.7367
[Epoch  5 Iter   209/625]: train_loss = 1.7340
[Epoch  5 Iter   418/625]: train_loss = 1.7119
[Epoch  5 Iter   625/625]: train_loss = 1.6874
[Epoch  6 Iter   209/625]: train_loss = 1.6954
[Epoch  6 Iter   418/625]: train_loss = 1.6175
[Epoch  6 Iter   625/625]: train_loss = 1.6409
[Epoch  7 Iter   209/625]: train_loss = 1.6300
[Epoch  7 Iter   418/625]: train_loss = 1.5704
[Epoch  7 Iter   625/625]: train_loss = 1.5864
[Epoch  8 Iter   209/625]: train_loss = 1.5217
[Epoch  8 Iter   418/625]: train_loss = 1.5524
[Epoch  8 Iter   625/625]: train_loss = 1.5273
[Epoch  9 Iter   209/625]: train_loss = 1.5390
[Epoch  9 Iter   418/625]: train_loss = 1.4959
[Epoch  9 Iter   625/625]: train_loss = 1.4553
[Epoch 10 Iter   209/625]: train_loss = 1.4317
[Epoch 10 Iter   418/625]: train_loss = 1.4425
[Epoch 10 Iter   625/625]: train_loss = 1.3974
[Epoch 11 Iter   209/625]: train_loss = 1.3811
[Epoch 11 Iter   418/625]: train_loss = 1.3577
[Epoch 11 Iter   625/625]: train_loss = 1.3808
[Epoch 12 Iter   209/625]: train_loss = 1.3373
[Epoch 12 Iter   418/625]: train_loss = 1.3626
[Epoch 12 Iter   625/625]: train_loss = 1.3055
[Epoch 13 Iter   209/625]: train_loss = 1.3144
[Epoch 13 Iter   418/625]: train_loss = 1.2533
[Epoch 13 Iter   625/625]: train_loss = 1.2920
[Epoch 14 Iter   209/625]: train_loss = 1.2453
[Epoch 14 Iter   418/625]: train_loss = 1.2586
[Epoch 14 Iter   625/625]: train_loss = 1.2454
[Epoch 15 Iter   209/625]: train_loss = 1.2000
[Epoch 15 Iter   418/625]: train_loss = 1.1960
[Epoch 15 Iter   625/625]: train_loss = 1.2186
Training completed.
```

3. Evaluate the model

Now let's evaluate the model. Remember that a 2-layered neural network only achieves an accuracy of around 38%. With a CNN architecture, you should be able to achieve a higher accuracy of more than 50%.

Exercise. Complete the function `evaluate` below to evaluate the model on the test loader

In [15]:
```
def evaluate(net, testloader):

    # set to evaluation mode
    net.eval()

    # running_correct
    running_corrects = 0

    # Repeat for all batch data in the test set
    for inputs, targets in testloader:

        # transfer to the GPU
        inputs = inputs.to(device)
        targets = targets.to(device)

        # # disable gradient computation
        with torch.no_grad():

            # perform inference
            outputs = net(inputs)

            # predict as the best result
            _, predicted = torch.max(outputs, 1)

            running_corrects += (targets == predicted).double().sum()


    print('Accuracy = {:.2f}%'.format(100*running_corrects/len(testloader.dataset)))
```

Now, let's evaluate our model.

In [16]:
```
evaluate(net1, testloader)
```

```
Accuracy = 52.65%
```

---

2. CREATING A CNN NETWORK DIRECTLY USING torch.nn.Sequential

In this section, we shall learn how to create a network using `torch.nn.Sequential`. `Sequential` is a container of `Modules` that can be stacked together and run at the same time.

```
net = nn.Sequential(
    nn.Conv2d(....),
    nn.ReLU(),
    ....
)

x = ... # get the input tensor
output = net(x)  # perform inference
```

We can see immediately that it is a very convenient way to build a network.

Limitations: Note that you cannot add functional operations (e.g., `torch.relu`) into a `Sequential` model. If the `nn` module version does not exist for the function, then you have to create your own `nn` module for the function.

Exercise. Reimplement the network above using `torch.nn.Sequential`.

Since you cannot use functional operations for `Sequential` models, you use their corresponding module versions:

- `torch.nn.functional.max_pool2d` --> `torch.nn.MaxPool2d`
- `torch.nn.functional.relu` --> `torch.nn.ReLU`
- `torch.view` --> `nn.Flatten`

In [17]:
```
net2 = nn.Sequential(

    nn.Conv2d(3,  32, kernel_size=3, stride=1, padding=1),  # conv1
    nn.ReLU(), # relu
    nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),  # conv2
    nn.ReLU(), # relu

    nn.MaxPool2d(kernel_size=2, stride=2, padding=0),       # max pool

    nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),  # conv3
```

```
    nn.ReLU(),  # relu
    nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),  # conv4
    nn.ReLU(), # relu

    nn.AdaptiveAvgPool2d((1,1)), # global average pooling
    nn.Flatten(),   # flatten

    nn.Linear(64, 10) # fc1
)
```

In [18]:

```
summary(net2, input_size=(4, 3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
              ReLU-2           [-1, 32, 32, 32]               0
            Conv2d-3           [-1, 32, 32, 32]           9,248
              ReLU-4           [-1, 32, 32, 32]               0
         MaxPool2d-5           [-1, 32, 16, 16]               0
            Conv2d-6           [-1, 64, 16, 16]          18,496
              ReLU-7           [-1, 64, 16, 16]               0
            Conv2d-8           [-1, 64, 16, 16]          36,928
              ReLU-9           [-1, 64, 16, 16]               0
AdaptiveAvgPool2d-10             [-1, 64, 1, 1]               0
          Flatten-11                   [-1, 64]               0
           Linear-12                   [-1, 10]             650
================================================================
Total params: 66,218
Trainable params: 66,218
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.56
Params size (MB): 0.25
Estimated Total Size (MB): 1.83
----------------------------------------------------------------
```

Train the model

In [19]:

```
train(net2, trainloader, num_epochs=15, lr=0.01, momentum=0.9)
```

```
[Epoch  1 Iter   209/625]: train_loss = 2.3037
[Epoch  1 Iter   418/625]: train_loss = 2.2817
[Epoch  1 Iter   625/625]: train_loss = 2.1324
[Epoch  2 Iter   209/625]: train_loss = 2.1070
[Epoch  2 Iter   418/625]: train_loss = 2.0628
[Epoch  2 Iter   625/625]: train_loss = 2.0199
[Epoch  3 Iter   209/625]: train_loss = 1.9433
[Epoch  3 Iter   418/625]: train_loss = 1.8955
[Epoch  3 Iter   625/625]: train_loss = 1.8435
[Epoch  4 Iter   209/625]: train_loss = 1.8034
[Epoch  4 Iter   418/625]: train_loss = 1.7735
[Epoch  4 Iter   625/625]: train_loss = 1.7826
[Epoch  5 Iter   209/625]: train_loss = 1.7399
[Epoch  5 Iter   418/625]: train_loss = 1.7113
[Epoch  5 Iter   625/625]: train_loss = 1.7396
[Epoch  6 Iter   209/625]: train_loss = 1.6783
[Epoch  6 Iter   418/625]: train_loss = 1.6685
[Epoch  6 Iter   625/625]: train_loss = 1.6208
[Epoch  7 Iter   209/625]: train_loss = 1.6188
[Epoch  7 Iter   418/625]: train_loss = 1.6320
[Epoch  7 Iter   625/625]: train_loss = 1.5998
[Epoch  8 Iter   209/625]: train_loss = 1.5504
[Epoch  8 Iter   418/625]: train_loss = 1.5874
[Epoch  8 Iter   625/625]: train_loss = 1.5513
[Epoch  9 Iter   209/625]: train_loss = 1.5319
[Epoch  9 Iter   418/625]: train_loss = 1.4825
[Epoch  9 Iter   625/625]: train_loss = 1.5077
[Epoch 10 Iter   209/625]: train_loss = 1.4713
[Epoch 10 Iter   418/625]: train_loss = 1.4456
[Epoch 10 Iter   625/625]: train_loss = 1.4627
[Epoch 11 Iter   209/625]: train_loss = 1.4206
[Epoch 11 Iter   418/625]: train_loss = 1.4193
[Epoch 11 Iter   625/625]: train_loss = 1.3976
[Epoch 12 Iter   209/625]: train_loss = 1.3967
[Epoch 12 Iter   418/625]: train_loss = 1.3494
[Epoch 12 Iter   625/625]: train_loss = 1.3385
[Epoch 13 Iter   209/625]: train_loss = 1.3304
[Epoch 13 Iter   418/625]: train_loss = 1.3122
[Epoch 13 Iter   625/625]: train_loss = 1.3069
[Epoch 14 Iter   209/625]: train_loss = 1.2950
[Epoch 14 Iter   418/625]: train_loss = 1.2506
[Epoch 14 Iter   625/625]: train_loss = 1.2692
[Epoch 15 Iter   209/625]: train_loss = 1.2458
[Epoch 15 Iter   418/625]: train_loss = 1.2469
[Epoch 15 Iter   625/625]: train_loss = 1.2405
Training completed.
```

Evaluate the model on the test set

In [20]:
```
evaluate(net2, testloader)
```

```
Accuracy = 55.20%
```

---

3. Using a function to create a customizable BLOCK module

In the following, we group the convolutional layers in the network above into 2 blocks:

- `conv1` and `conv2` --> `block_1`
- `conv3` and `conv4` --> `block_2` .

This is how the network looks:

NETWORK (with block)

Block	Layer	Name	Description	OutputShape
input	-	-	-	(?, 3, 32, 32)
block_1 (blk_cin=3, blk_cout=32)	1	conv1	Conv2d (in_channels=3, out_channels=32,f=3,s=1,p=1)	(?, 32, 32, 32)
	-	ReLU	relu	(?, 32, 32, 32)
	2	conv2	Conv2d (in_channels=32, out_channels=32,f=3,s=1,p=1)	(?, 32, 32, 32)
	-	ReLU	relu	(?, 32, 32, 32)
-	-	pool1	maxpool (f=2,s=2,p=0)	(?, 32, 16, 16)
block_2 (blk_cin=32, blk_cout=64)	3	conv1	Conv2d (in_channels=32, out_channels=64,f=3,s=1,p=1)	(?, 64, 16, 16)
	-	ReLU	relu	(?, 64, 16, 16)
	4	conv2	Conv2d (in_channels=64, out_channels=64, f=3,s=1,p=1)	(?, 64, 16, 16)
	-	ReLU	relu	(?, 64, 16, 16)

Block	Layer	Name	Description	OutputShape
	-	global_pool	AdaptiveAvgPool, o=(1,1)	(?, 64, 1, 1)
	-	-	view	(?, 64)
	5	fc1	Linear(#units=10)	(?, 10)
	-	-	view	(?, 10)

Notes: *k*: number of filters, *f*: filter or kernel size, *s*: stride, *p*: padding, *o*: output shape

Comparing `block_1` and `block_2`, we find that they have similar structure

`conv (blk_cin, blk_cout)  --> relu --> conv (blk_cout, blk_cout) --> relu`

where

- `blk_cin=3` and `blk_cout=32` for `block1`
- `blk_cin=32` and `blk_cout=64` for `block2`

This means that we can use the same function to create `block1` and `block2`. we shall do precisely this next.

The BUILD_BLOCK function

Rather than declaring each layer individually, you create a function `build_block` to create a block of layers. The function receives `in_ch`, the number of channels in the input tensor and `out_ch`, the number of channels in the output tensor. In the following, complete the `build_block` function by returning a `nn.Sequential` module that builds and returns the following block

BLOCK (blk_cin, blk_cout)
Conv2d (in_channels=blk_cin, output_channels=blk_cout, f=3, s=1,p=1)
ReLU ()
Conv2d (in_channels=blk_cout, output_channels=blk_cout, f=3, s=1,p=1)
ReLU ()

In [21]:
```
def build_block(blk_cin, blk_cout):
    block = nn.Sequential(
        nn.Conv2d(blk_cin,  blk_cout, kernel_size=3, stride=1, padding=1),
        nn.ReLU(),
        nn.Conv2d(blk_cout, blk_cout, kernel_size=3, stride=1, padding=1),
        nn.ReLU(),
    )
    return block
```

In [22]:
```
block1 = build_block(3, 32)
print(block1)
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU()
)
```

Constructing the network with BUILD_BLOCK

Now, construct `NETWORK (with block)`. Use the `build_block` function to construct `block1` and `block2`

In [23]:
```
class Net3(nn.Module):
    def __init__(self):
        super().__init__()

        # define block 1
        self.conv_block1 = build_block(3, 32)

        # define block 2
        self.conv_block2 = build_block(32, 64)

        # define global_pool
        self.global_pool = nn.AdaptiveAvgPool2d((1,1))

        # define fc1
        self.fc1 = nn.Linear(64, 10)

    def forward(self, x):
        # block 1
        x = self.conv_block1(x)

        # max pool
```

```python
        x = F.max_pool2d(x, kernel_size=2, stride=2, padding=0)

        # block 2
        x = self.conv_block2(x)

        # global pool
        x = self.global_pool(x)

        # view
        x = x.view(x.size(0), -1)

        # fc1
        x = self.fc1(x)

        return x
```

In [24]:
```python
net3 = Net3()
summary(net3, input_size=(4, 3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
              ReLU-2           [-1, 32, 32, 32]               0
            Conv2d-3           [-1, 32, 32, 32]           9,248
              ReLU-4           [-1, 32, 32, 32]               0
            Conv2d-5           [-1, 64, 16, 16]          18,496
              ReLU-6           [-1, 64, 16, 16]               0
            Conv2d-7           [-1, 64, 16, 16]          36,928
              ReLU-8           [-1, 64, 16, 16]               0
 AdaptiveAvgPool2d-9             [-1, 64, 1, 1]               0
           Linear-10                   [-1, 10]             650
================================================================
Total params: 66,218
Trainable params: 66,218
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.50
Params size (MB): 0.25
Estimated Total Size (MB): 1.76
----------------------------------------------------------------
```

Train the model

In [25]:
```
train(net3, trainloader, num_epochs=15, lr=0.01, momentum=0.9)
```

```
[Epoch  1 Iter   209/625]: train_loss = 2.3043
[Epoch  1 Iter   418/625]: train_loss = 2.2616
[Epoch  1 Iter   625/625]: train_loss = 2.1163
[Epoch  2 Iter   209/625]: train_loss = 2.0972
[Epoch  2 Iter   418/625]: train_loss = 2.0199
[Epoch  2 Iter   625/625]: train_loss = 1.9518
[Epoch  3 Iter   209/625]: train_loss = 1.9106
[Epoch  3 Iter   418/625]: train_loss = 1.8369
[Epoch  3 Iter   625/625]: train_loss = 1.8175
[Epoch  4 Iter   209/625]: train_loss = 1.8142
[Epoch  4 Iter   418/625]: train_loss = 1.7458
[Epoch  4 Iter   625/625]: train_loss = 1.7503
[Epoch  5 Iter   209/625]: train_loss = 1.7100
[Epoch  5 Iter   418/625]: train_loss = 1.6953
[Epoch  5 Iter   625/625]: train_loss = 1.6882
[Epoch  6 Iter   209/625]: train_loss = 1.6642
[Epoch  6 Iter   418/625]: train_loss = 1.6411
[Epoch  6 Iter   625/625]: train_loss = 1.6486
[Epoch  7 Iter   209/625]: train_loss = 1.6068
[Epoch  7 Iter   418/625]: train_loss = 1.6026
[Epoch  7 Iter   625/625]: train_loss = 1.5702
[Epoch  8 Iter   209/625]: train_loss = 1.5390
[Epoch  8 Iter   418/625]: train_loss = 1.5445
[Epoch  8 Iter   625/625]: train_loss = 1.5442
[Epoch  9 Iter   209/625]: train_loss = 1.4911
[Epoch  9 Iter   418/625]: train_loss = 1.5032
[Epoch  9 Iter   625/625]: train_loss = 1.4851
[Epoch 10 Iter   209/625]: train_loss = 1.4244
[Epoch 10 Iter   418/625]: train_loss = 1.4538
[Epoch 10 Iter   625/625]: train_loss = 1.4037
[Epoch 11 Iter   209/625]: train_loss = 1.3941
[Epoch 11 Iter   418/625]: train_loss = 1.3626
[Epoch 11 Iter   625/625]: train_loss = 1.3757
[Epoch 12 Iter   209/625]: train_loss = 1.3141
[Epoch 12 Iter   418/625]: train_loss = 1.3364
[Epoch 12 Iter   625/625]: train_loss = 1.3011
[Epoch 13 Iter   209/625]: train_loss = 1.2824
[Epoch 13 Iter   418/625]: train_loss = 1.2968
[Epoch 13 Iter   625/625]: train_loss = 1.2925
[Epoch 14 Iter   209/625]: train_loss = 1.2376
[Epoch 14 Iter   418/625]: train_loss = 1.1726
[Epoch 14 Iter   625/625]: train_loss = 1.2761
[Epoch 15 Iter   209/625]: train_loss = 1.1994
[Epoch 15 Iter   418/625]: train_loss = 1.2135
[Epoch 15 Iter   625/625]: train_loss = 1.1916
Training completed.
```

Evaluate the model

In [26]:
```
evaluate(net3, testloader)
```

```
Accuracy = 52.65%
```

---

4. Create a customizable BLOCK module

We can also build a block by constructing a module called `BLOCK` where we can specify the `blk_cin` and `blk_cin` when constructing the block.

The BLOCK module

Implement the BLOCK module.

BLOCK (blk_cin, blk_cout)
Conv2d (in_channels=blk_cin, output_channels=blk_cout, f=3, s=1,p=1)
ReLU ()
Conv2d (in_channels=blk_cout, output_channels=blk_cout, f=3, s=1,p=1)
ReLU ()

In [27]:
```
class BLOCK(nn.Module):
    def __init__(self, blk_cin, blk_cout):

        super().__init__()

        self.conv1 = nn.Conv2d(blk_cin,  blk_cout, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(blk_cout, blk_cout, kernel_size=3, stride=1, padding=1)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))

        return x
```

Construcing network with the BLOCK module

Now, let's build the network using the `BLOCK` module that we have just created.

In [28]:
```
class Net4(nn.Module):
    def __init__(self):
        super().__init__()

        # define block 1
        self.conv_block1 = BLOCK(3, 32)

        # define block 2
        self.conv_block2 = BLOCK(32, 64)

        # define global_pool
        self.global_pool = nn.AdaptiveAvgPool2d((1,1))

        # define fc1
        self.fc1 = nn.Linear(64, 10)

    def forward(self, x):
        # block 1
        x = self.conv_block1(x)

        # max pool
        x = F.max_pool2d(x, kernel_size=2, stride=2, padding=0)

        # block 2
        x = self.conv_block2(x)

        # global pool
        x = self.global_pool(x)

        # view
        x = x.view(x.size(0), -1)

        # fc1
        x = self.fc1(x)

        return x
```

In [29]:
```
net4 = Net4()
```

Train the model

In [30]:
```
train(net4, trainloader, num_epochs=15, lr=0.01, momentum=0.9)
```

```
[Epoch  1 Iter   209/625]: train_loss = 2.3015
[Epoch  1 Iter   418/625]: train_loss = 2.2380
[Epoch  1 Iter   625/625]: train_loss = 2.1052
[Epoch  2 Iter   209/625]: train_loss = 2.0804
[Epoch  2 Iter   418/625]: train_loss = 2.0293
[Epoch  2 Iter   625/625]: train_loss = 1.9657
[Epoch  3 Iter   209/625]: train_loss = 1.9291
[Epoch  3 Iter   418/625]: train_loss = 1.8748
[Epoch  3 Iter   625/625]: train_loss = 1.8325
[Epoch  4 Iter   209/625]: train_loss = 1.7820
[Epoch  4 Iter   418/625]: train_loss = 1.7922
[Epoch  4 Iter   625/625]: train_loss = 1.7768
[Epoch  5 Iter   209/625]: train_loss = 1.7579
[Epoch  5 Iter   418/625]: train_loss = 1.7392
[Epoch  5 Iter   625/625]: train_loss = 1.7319
[Epoch  6 Iter   209/625]: train_loss = 1.6866
[Epoch  6 Iter   418/625]: train_loss = 1.6857
[Epoch  6 Iter   625/625]: train_loss = 1.6418
[Epoch  7 Iter   209/625]: train_loss = 1.6643
[Epoch  7 Iter   418/625]: train_loss = 1.6227
[Epoch  7 Iter   625/625]: train_loss = 1.6382
[Epoch  8 Iter   209/625]: train_loss = 1.6152
[Epoch  8 Iter   418/625]: train_loss = 1.5588
[Epoch  8 Iter   625/625]: train_loss = 1.5742
[Epoch  9 Iter   209/625]: train_loss = 1.5690
[Epoch  9 Iter   418/625]: train_loss = 1.5363
[Epoch  9 Iter   625/625]: train_loss = 1.5349
[Epoch 10 Iter   209/625]: train_loss = 1.4852
[Epoch 10 Iter   418/625]: train_loss = 1.4738
[Epoch 10 Iter   625/625]: train_loss = 1.4711
[Epoch 11 Iter   209/625]: train_loss = 1.4782
[Epoch 11 Iter   418/625]: train_loss = 1.4293
[Epoch 11 Iter   625/625]: train_loss = 1.3731
[Epoch 12 Iter   209/625]: train_loss = 1.4311
[Epoch 12 Iter   418/625]: train_loss = 1.4043
[Epoch 12 Iter   625/625]: train_loss = 1.3351
[Epoch 13 Iter   209/625]: train_loss = 1.3347
[Epoch 13 Iter   418/625]: train_loss = 1.3368
[Epoch 13 Iter   625/625]: train_loss = 1.2996
[Epoch 14 Iter   209/625]: train_loss = 1.2829
[Epoch 14 Iter   418/625]: train_loss = 1.2901
[Epoch 14 Iter   625/625]: train_loss = 1.2729
[Epoch 15 Iter   209/625]: train_loss = 1.2537
[Epoch 15 Iter   418/625]: train_loss = 1.2475
[Epoch 15 Iter   625/625]: train_loss = 1.2435
Training completed.
```

Evaluate the model

In [31]:
```
evaluate(net4, testloader)
```

```
Accuracy = 55.00%
```

--- End of Practical ---