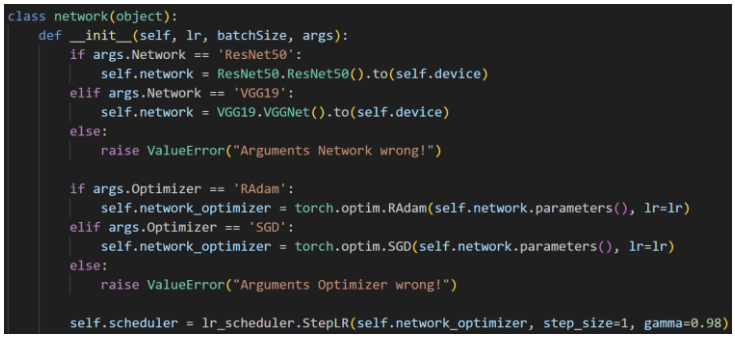
1. Introduction (10%):

In this homework, we aim to construct a classifier (either VGG19 or ResNet50) to classify 100 species of butterflies and moths. In this section, we will introduce the overall structure.

In the dataloader.py file, we implement a dataloader that includes loading images from a folder and transforming the dataset (normalization, flipping, and rotation). Detailed introductions are provided in Section 2-B and Section 3.

In the VGG19.py and ResNet.py files, we respectively implement the VGG19 and ResNet50 networks. Detailed introductions are provided in Section 2-A.

In the main.py file, for convenience, we encapsulated training, evaluation, testing, and network (VGG19 or ResNet50) into a class object. Next, we'll introduce this class.

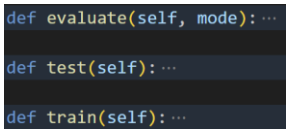


```
class network(object):
    def __init__(self, lr, batchSize, args):
        if args.Network == 'ResNet50':
            self.network = ResNet50.ResNet50().to(self.device)
        elif args.Network == 'VGG19':
            self.network = VGG19.VGGNet().to(self.device)
        else:
            raise ValueError("Arguments Network wrong!")

        if args.Optimizer == 'RAdam':
            self.network_optimizer = torch.optim.RAdam(self.network.parameters(), lr=lr)
        elif args.Optimizer == 'SGD':
            self.network_optimizer = torch.optim.SGD(self.network.parameters(), lr=lr)
        else:
            raise ValueError("Arguments Optimizer wrong!")

        self.scheduler = lr_scheduler.StepLR(self.network_optimizer, step_size=1, gamma=0.98)
```

The network class include the network (VGG19 or ResNet50), network optimizer.



```
def evaluate(self, mode): ···

def test(self): ···

def train(self): ···
```

We also have functions: evaluate(to evaluate the network performance on both training and validation datasets), test(to evaluate the network performance on the testing dataset after training), and train(to train the network for one epoch).

Terminal Usage:

For example, in terminal you can execute:

python main.py --Epoch=150 --BatchSize=50 --Optimizer="SGD" --Network="VGG19" --lr=4e-3

args:

Epoch(int): Training total epoch.

BatchSize(int): The batch size used during training.

Optimizer(str): Select the training optimizer “SGD” or “RAdam”.

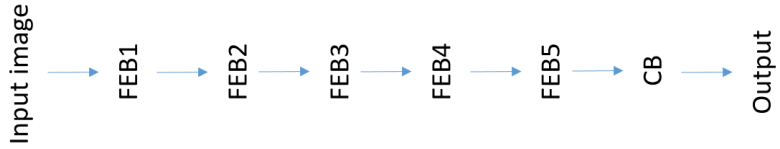
Network(str): Select the network “VGG19” or “ResNet50”.

lr(float): learning rate used during training.

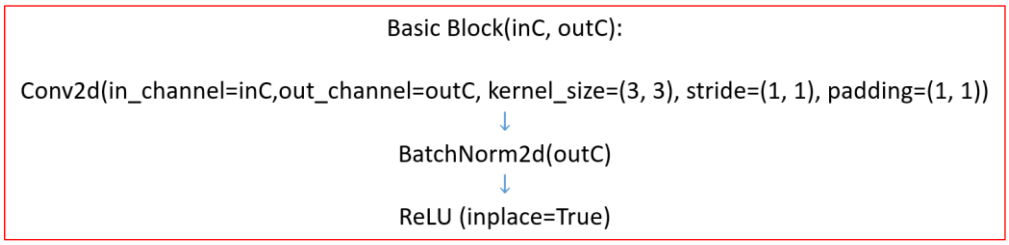
2. Implementation on Details (30%):

A. The details of your model (VGG19, ResNet50):

The VGG19 model is composed of 5 feature extractor blocks (FEBs) and 1 classification block (CB).



Each FEB consists of several basic blocks:



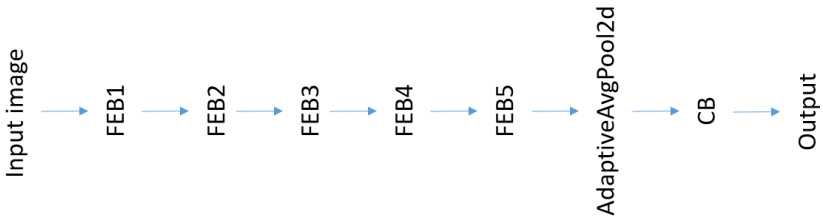
The structure details of VGG19 are as follows:

- Feature extractor block1(Input size: [batch_size, 3, 244, 244], output size: [batch_size, 64, 112, 112])
    1. Basic Block(inC=3, outC=64)
    2. Basic Block(inC=64, outC=64)
    3. nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Feature extractor block2(Input size: [batch_size, 64, 112, 112], output size: [batch_size, 128, 56, 56])
    1. Basic Block(inC=64, outC=128)
    2. Basic Block(inC=128, outC=128)
    3. nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Feature extractor block3(Input size: [batch_size, 128, 56, 56], output size: [batch_size, 256, 28, 28])
    1. Basic Block(inC=128, outC=256)
    2. Basic Block(inC=256, outC=256)
    3. Basic Block(inC=256, outC=256)
    4. Basic Block(inC=256, outC=256)
    5. nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Feature extractor block4(Input size: [batch_size, 256, 28, 28], output size: [batch_size, 512,14,14])
    1. Basic Block(inC=256, outC=512)
    2. Basic Block(inC=512, outC=512)
    3. Basic Block(inC=512, outC=512)
    4. Basic Block(inC=512, outC=512)
    5. nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Feature extractor block5(Input size: [batch_size, 512,14,14], output size: [batch_size, 512, 7, 7])
    1. Basic Block(inC=512, outC=512)

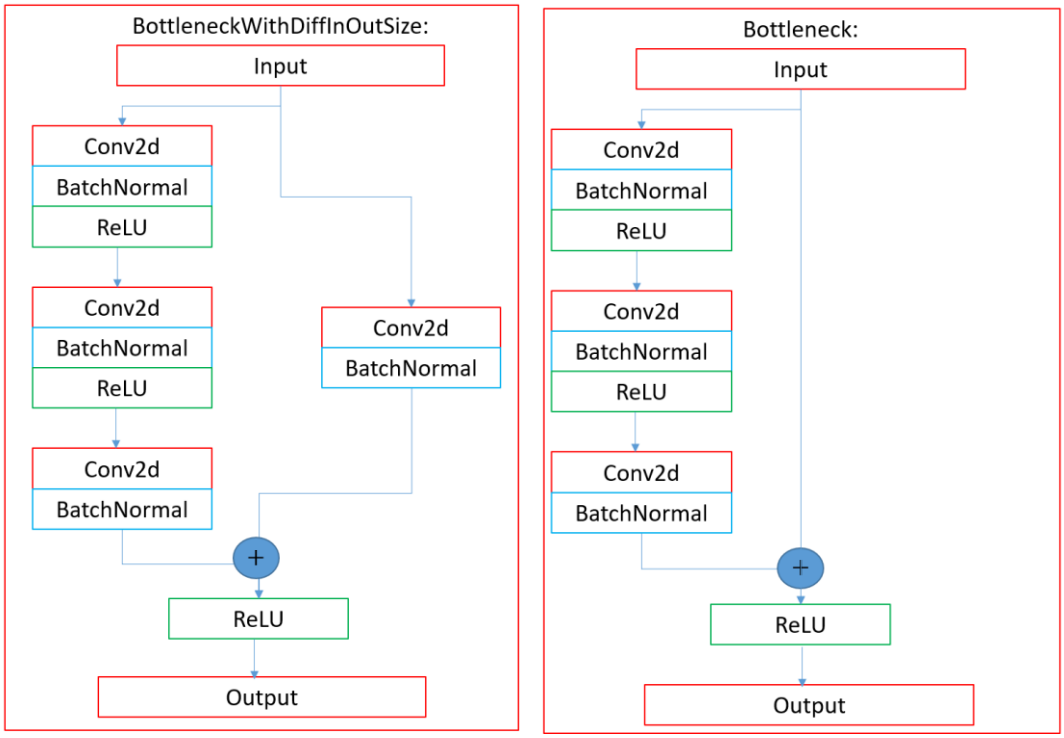
2. Basic Block(inC=512, outC=512)
3. Basic Block(inC=512, outC=512)
4. Basic Block(inC=512, outC=512)
5. nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

- Classification block(Input size: [batch_size, 512*7*7], output size: [batch_size, 100])
  1. nn.Dropout(0.5, inplace=False),
  2. nn.Linear(7*7*512, 4096, bias=True)
  3. nn.ReLU(inplace=True)
  4. nn.Dropout(0.5, inplace=False)
  5. nn.Linear(4096, 1000, bias=True)
  6. nn.ReLU(inplace=True)
  7. nn.Linear(1000,100, bias=True)

The ResNet50 model is composing by 5 feature extractor blocks(FEB) and 1 Classification block(CB).



Each FEB consists of 1 BottleneckWithDiffInOutSize and several Bottleneck:



The abstract structure of ResNet50 is as follows. (For the sake of concise formatting,

we will provide the detailed structure at the 6.Appendix at the end of the report.):

- Feature extractor block1(Input size: [batch_size, 3, 244, 244], output size: [batch_size, 64, 56, 56])
    1. nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3)),
    2. nn.BatchNorm2d(64),
    3. nn.ReLU(inplace=True),
    4. nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
- Feature extractor block2(Input size: [batch_size, 64, 56, 56], output size: [batch_size,256, 56, 56])
    1. BottleneckWithDiffInOutSize
    2. Bottleneck*2
- Feature extractor block3(Input size: [batch_size, 256, 56, 56], output size: [batch_size, 512, 28, 28])
    1. BottleneckWithDiffInOutSize
    2. Bottleneck*3
- Feature extractor block4(Input size: [batch_size, 512, 28, 28], output size: [batch_size, 1024, 14, 14])
    1. BottleneckWithDiffInOutSize
    2. Bottleneck*5
- Feature extractor block5(Input size: [batch_size, 1024, 14, 14], output size: [batch_size, 2048, 7, 7])
    1. BottleneckWithDiffInOutSize
    2. Bottleneck*2
- Classification block(Input size: [batch_size, 2048], output size: [batch_size, 100])
    1. nn.Dropout(0.6, inplace=False)
    2. nn.Linear(2048,100)

B. The details of your Dataloader:

We manually construct the dataloader instead of using torch.utils.data.DataLoader.

```
class BufferflyMothLoader(data.Dataset):
    def __init__(self, root, mode):
```

In the __init__, the input parameter:

- root: the path to the dataset
- mode: train, valid or test

In __init__, we get the path to the dataset and labels.

```
def __getitem__(self, index,task):
```

In the class function __getitem__, the input parameter:

- index(List): the index that we want to return the image corresponding to.
- task: Use 'train' or 'eval' to determine how to preprocess the image.

In this function, we do the following things:

1. Load the images and labels corresponding to the index and task.
2. Preprocess the images.
3. Convert the image and label to tensor and return the image and labels.

```
class network(object):
    def __init__(self, lr, batchSize, args):
        self.dataset_train = BufferflyMothLoader(root='dataset', mode='train')
        self.dataset_valid = BufferflyMothLoader(root='dataset', mode='valid')
        self.dataset_test = BufferflyMothLoader(root='dataset', mode='test')
```

In the main.py file, we initialize dataset_train, dataset_valid, and dataset_test as dataset loaders. During each epoch of training, we first randomly shuffle the indices [0, len(dataset)]. For each batch of an epoch, we use the __getitem__ function to retrieve the images and labels corresponding to the batch indices.
During each epoch evaluation of training and testing, we also use the __getitem__ function to retrieve the images and labels corresponding to the batch indices.

3. Data Preprocessing (20%):

A. How you preprocessed your data?

We preprocess the image when calling BufferflyMothLoader class function __getitem__, in this function, we do the following things:
For the training process, we

1. Scale the image pixel value from [0,255] to [0,1]
2. Randomly decide whether to flip the image.
3. Randomly select the angle for rotating the image.

For the evaluation and testing process, we

1. Normalize the image pixel value from [0,255] to [0,1]

B. What makes your method special?

During the training process, our method will randomly determine whether to flip the image and select the angle for rotation. Therefore, for the same image, different decisions will be made at different training epochs. This method can enhance the generalization strength of model.

4. Experimental results (10%):

A. The highest testing accuracy:

For the VGG19, we set the following hyperparameter:

- Learining rate: 4e-3
- Optimizer: SGD
- Training Epoch: 150
- BatchSize: 50

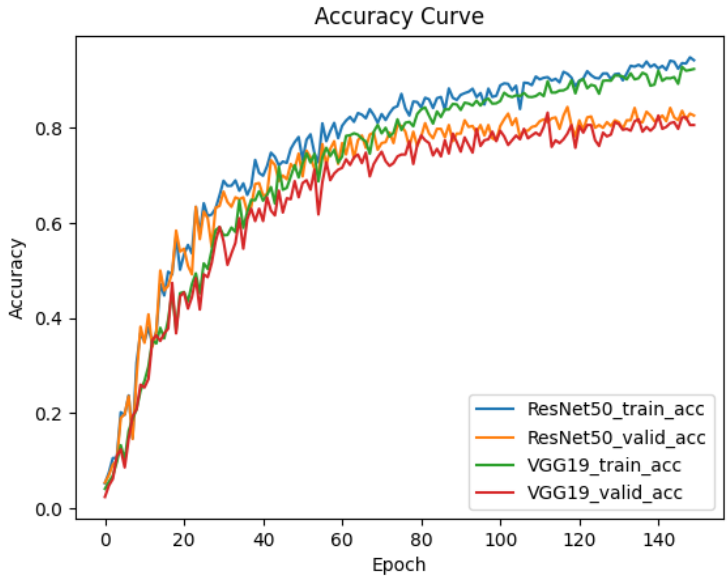
```
Test batch  0, batch predict accuracy: 46/50(0.92)
Test batch  1, batch predict accuracy: 42/50(0.84)
Test batch  2, batch predict accuracy: 44/50(0.88)
Test batch  3, batch predict accuracy: 47/50(0.94)
Test batch  4, batch predict accuracy: 43/50(0.86)
Test batch  5, batch predict accuracy: 43/50(0.86)
Test batch  6, batch predict accuracy: 44/50(0.88)
Test batch  7, batch predict accuracy: 42/50(0.84)
Test batch  8, batch predict accuracy: 38/50(0.76)
Test batch  9, batch predict accuracy: 42/50(0.84)
Total predict accuracy: 431/500(0.86)
```

For the ResNet50, we set the following hyperparameter:

- Learining rate: 1e-3
- Optimizer: RAdam
- Training Epoch: 150
- BatchSize: 80

```
Test batch  0, batch predict accuracy: 71/80(0.89)
Test batch  1, batch predict accuracy: 68/80(0.85)
Test batch  2, batch predict accuracy: 77/80(0.96)
Test batch  3, batch predict accuracy: 72/80(0.90)
Test batch  4, batch predict accuracy: 73/80(0.91)
Test batch  5, batch predict accuracy: 69/80(0.86)
Test batch  6, batch predict accuracy: 16/20(0.80)
Total predict accuracy: 446/500(0.89)
```

B. Comparison figures:

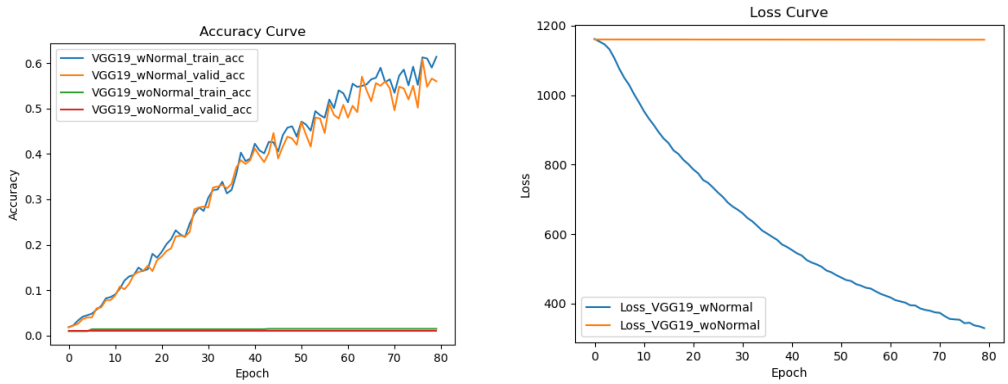


5. Discussion (30%):

- We can observe that ResNet50 has better performance and learning speed than VGG19. This is because ResNet50 has a deeper layer and a bottleneck network structure.

- The necessity of BatchNorm2d and ReLU layers in the VGG19 model.

In this part, we employ an experimental perspective to illustrate the necessity of the BatchNorm2d layer. We provide the learning curve (accuracy on training and validation datasets) and loss curve of VGG19 w/wo the BatchNorm2d layer after the Conv2d layer.



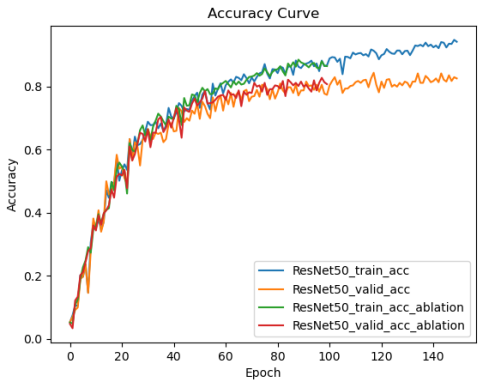
The labels with '_wNormal' indicate the results of the VGG19 network with the BatchNorm2d layer after Conv2d, while 'woNormal' indicates the results without the BatchNorm2d layer after Conv2d. We can observe that without the BatchNorm2d layer, the training performance would be poor.

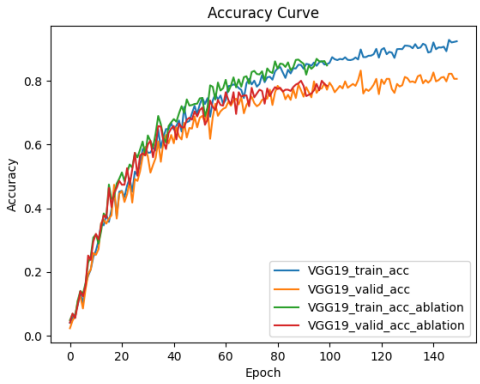
- Preprocessing with more strong method

In this part, we employ a more robust method for preprocessing the image to increase the diversity of the training data. In addition to random flipping and rotating, we randomly select one of the following techniques.

1. ImageFilter.BLUR: Applying image blurring.
2. ImageFilter.CONTOUR: Generate image contour.
3. ImageFilter.FIND_EDGES: Detect the edge of image.
4. ImageFilter.EMBOSS: Image embossing.
5. ImageFilter.SMOOTH: Smooth the image.
6. None

We conduct experiments using the VGG19 and ResNet50 models with the same hyperparameters in Section 4. The only difference is that the strong method is trained for only 100 epochs. We compare the learning curves between the strong method and the original method.

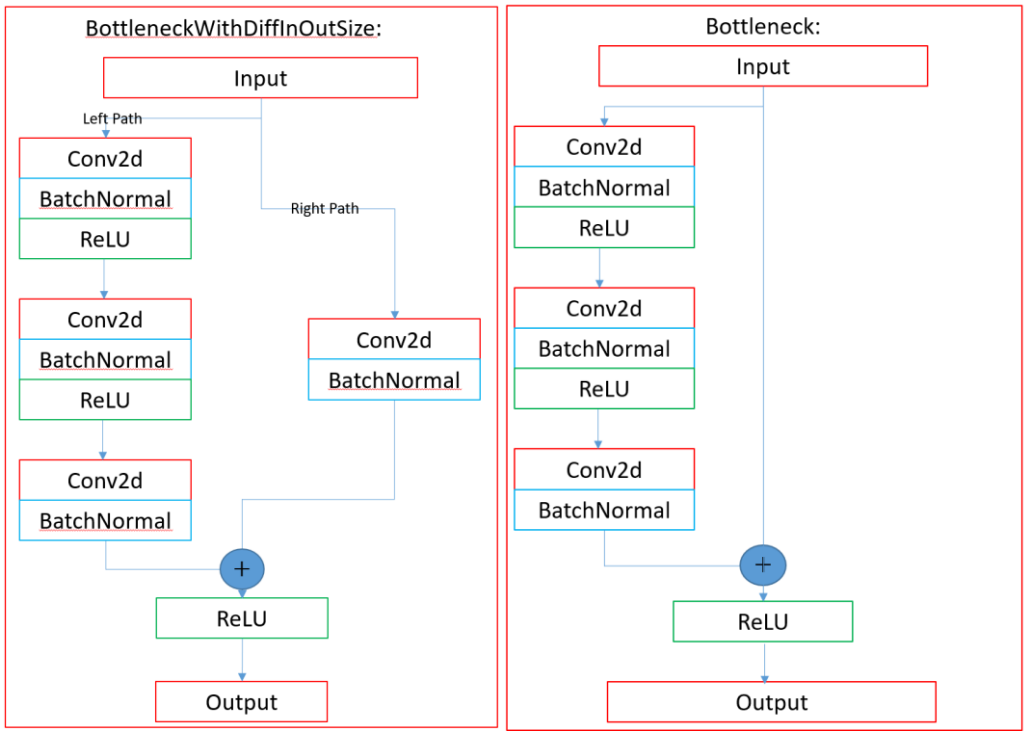




The labels with '_ablation' are trained using strong image preprocessing. We observe that using strong image preprocessing results in slightly better performance compared to not using the strong method.

6. Appendix:

In this part, we provide the detail structure of ResNet50:



- Feature extractor block1(Input size: [batch_size, 3, 244, 244], output size: [batch_size, 64, 56, 56])
    1. nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3)),
    2. nn.BatchNorm2d(64),
    3. nn.ReLU(inplace=True),
    4. nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
- Feature extractor block2(Input size: [batch_size, 64, 56, 56], output size: [batch_size,256, 56, 56])
    1. BottleneckWithDiffInOutSize
        - LeftPath:

- (Conv2d kernel=1*1 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
- (Conv2d kernel=3*3 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
- (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
- RightPath
- (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)

2. Bottleneck
    - (Conv2d kernel=1*1 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
    - (Conv2d kernel=3*3 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
    - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
3. Bottleneck
    - (Conv2d kernel=1*1 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
    - (Conv2d kernel=3*3 out_channel=64, stride=1) + nn.BatchNorm2d(64)+ nn.ReLU(inplace=True)
    - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)

- Feature extractor block3(Input size: [batch_size, 256, 56, 56], output size: [batch_size, 512, 28, 28])
    1. BottleneckWithDiffInOutSize
        - LeftPath:
        - (Conv2d kernel=1*1 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=3*3 out_channel=128, stride=2) + nn.BatchNorm2d(128) nn.ReLU(inplace=True)
        - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
        - RightPath
        - (Conv2d kernel=1*1 out_channel=512, stride=2) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
    2. Bottleneck
        - (Conv2d kernel=1*1 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=3*3 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
    3. Bottleneck
        - (Conv2d kernel=1*1 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=3*3 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
    4. Bottleneck
        - (Conv2d kernel=1*1 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=3*3 out_channel=128, stride=1) + nn.BatchNorm2d(128)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
- Feature extractor block4(Input size: [batch_size, 512, 28, 28], output size: [batch_size, 1024, 14, 14])
    1. BottleneckWithDiffInOutSize
        - LeftPath:
        - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
        - (Conv2d kernel=3*3 out_channel=256, stride=2) + nn.BatchNorm2d(256) nn.ReLU(inplace=True)
        - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)

- RightPath
- (Conv2d kernel=1*1 out_channel=1024, stride=2) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)

2. Bottleneck
   - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)
3. Bottleneck
   - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)
4. Bottleneck
   - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)
5. Bottleneck
   - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)
6. Bottleneck
   - (Conv2d kernel=1*1 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=256, stride=1) + nn.BatchNorm2d(256)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=1024, stride=1) + nn.BatchNorm2d(1024)+ nn.ReLU(inplace=True)

- Feature extractor block5(Input size: [batch_size, 1024, 14, 14], output size: [batch_size, 2048, 7, 7])

1. BottleneckWithDiffInOutSize
   - LeftPath:
   - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=512, stride=2) + nn.BatchNorm2d(512) nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=2048, stride=1) + nn.BatchNorm2d(2048)+ nn.ReLU(inplace=True)
   - RightPath
   - (Conv2d kernel=1*1 out_channel=2048, stride=2) + nn.BatchNorm2d(2048)+ nn.ReLU(inplace=True)
2. Bottleneck
   - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=2048, stride=1) + nn.BatchNorm2d(2048)+ nn.ReLU(inplace=True)
3. Bottleneck
   - (Conv2d kernel=1*1 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=3*3 out_channel=512, stride=1) + nn.BatchNorm2d(512)+ nn.ReLU(inplace=True)
   - (Conv2d kernel=1*1 out_channel=2048, stride=1) + nn.BatchNorm2d(2048)+ nn.ReLU(inplace=True)

- Classification block(Input size: [batch_size, 2048], output size: [batch_size, 100])
    1. nn.Dropout(0.6, inplace=False)
    2. nn.Linear(2048,100)