```python
"""
EECS 445 - Introduction to Machine Learning
Fall 2022 - Project 2
Challenge
    Constructs a pytorch model for a convolutional neural network
    Usage: from model.challenge import Challenge
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from math import sqrt
from utils import config
from torchvision import transforms
from torch.utils.data import DataLoader

learning_rate = 0.00005
num_epochs = 50
batch_size = 10
IMG_SIZE = (64, 64)
num_classes = 2


class Challenge(nn.Module):
    def __init__(self):
        """Define the architecture, i.e. what layers our network contains.
        At the end of __init__() we call init_weights() to initialize all model parameters (weights and biases)
        in all layers to desired distributions."""
        super().__init__()

        # TODO: define each layer of your network
        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=3,
                      out_channels=64,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      # (1(32-1)- 32 + 3)/2 = 1
                      padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64,
                      out_channels=64,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_2 = nn.Sequential(
            nn.Conv2d(in_channels=64,
                      out_channels=128,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128,
                      out_channels=128,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_3 = nn.Sequential(
            nn.Conv2d(in_channels=128,
                      out_channels=256,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
```

```
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256,
                      out_channels=256,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256,
                      out_channels=256,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_4 = nn.Sequential(
            nn.Conv2d(in_channels=256,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_5 = nn.Sequential(
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.classifier = nn.Sequential(
            nn.Linear(2048, 4096),
            nn.ReLU(True),
```

```
            nn.Dropout(p=0.65),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.65),
            nn.Linear(4096, num_classes),
        )

        self.init_weights()

    def init_weights(self):
        """Initialize all model parameters (weights and biases) in all layers to desired distributions"""
        # TODO: initialize the parameters for your network
        torch.manual_seed(42)

        for m in [self.block_1, self.block_2, self.block_3, self.block_4, self.block_5, self.classifier]:
            if isinstance(m, torch.nn.Conv2d) or isinstance(m, torch.nn.Linear):

                nn.init.kaiming_uniform_(
                    m.weight, mode='fan_in', nonlinearity='leaky_relu')
                if m.bias is not None:
                    m.bias.detach().zero_()

    def forward(self, x):
        """This function defines the forward propagation for a batch of input examples, by
            successively passing output of the previous layer as the input into the next layer (after applying
            activation functions), and returning the final output as a torch.Tensor object

            You may optionally use the x.shape variables below to resize/view the size of
            the input matrix at different points of the forward pass"""
        N, C, H, W = x.shape

        # TODO: forward pass
        x = self.block_1(x)
        x = self.block_2(x)
        x = self.block_3(x)
        x = self.block_4(x)
        x = self.block_5(x)
        # x = self.avgpool(x)
        # x = x.view(x.size(0), -1)
        x = x.reshape(N, 2048)  # 512*2*2
        # x = x.reshape(512, 4096)
        x = self.classifier(x)
        # probyas = F.softmax(logits, dim=1)
        # probas = nn.Softmax(logits)
        return x
        # return logits
```