HW5: Image Classification & Detection

CS4610/5335: Robotic Science and Systems (Spring 2023) | Robert Platt | Northeastern University

---

Please remember the following policies:

- Submissions should be made electronically via the Canvas. For this assignment, you should submit both a *.ipynb and *.pdf version of your completed Colab notebook as a single zipped file.
- You are welcome to discuss the programming questions (but not the written questions) with other students in the class. However, you must understand and write all code yourself. Also, you must list all students (if any) with whom you discussed your solutions to the programming questions.
- Please provide comments in your code to make it understandable to the graders.

Collaborators: [*Optional*]

```
%%capture
! pip install pytorch-lightning
! pip install torchsummary
```

```
## UPLOAD utils.py: click "Choose Files" then click on "utils.py"
from google.colab import files
uploaded = files.upload()

try:
  import utils
except ModuleNotFoundError:
  raise ModuleNotFoundError(
    'ERROR: you did not upload "utils.py" correctly, run the cell again.'
  )
```

Choose Files utils.py

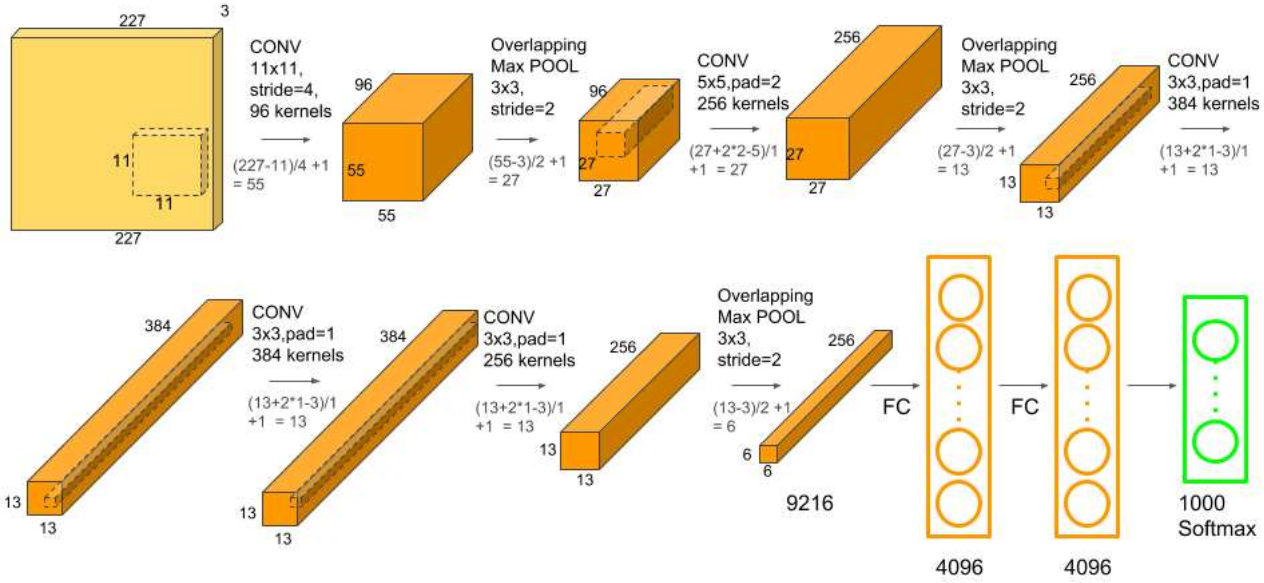
- **utils.py**(n/a) - 5104 bytes, last modified: 3/13/2023 - 100% done

```
Saving utils.py to utils.py
```

```
import torch
from torch import Tensor
import torch.nn as nn
import torchvision
import pytorch_lightning as pl
import matplotlib.pyplot as plt
from torchsummary import summary
```

Q1a. Implementing AlexNet

In this question, you will implement AlexNet in PyTorch using the model achitecture diagram provided below. For an introduction to creating neural networks with PyTorch, see this guide. If you want to look up specific pytorch modules, try searching the docs.




```
class AlexNet(nn.Module):
  def __init__(self):
    """
    You need to add ReLU activations after every internal convolution or linear layer.
    Do not add BatchNorm layers
    """
    super().__init__()

    self.conv1 = nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=4)
    self.relu1 = nn.ReLU(inplace=True)
    self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2)

    self.conv2 = nn.Conv2d(in_channels=96, out_channels=256, kernel_size=5, stride=1, padding=2)
    self.relu2 = nn.ReLU(inplace=True)
    self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2)

    self.conv3 = nn.Conv2d(in_channels=256, out_channels=384, kernel_size=3, stride=1, padding=1)
    self.relu3 = nn.ReLU(inplace=True)

    self.conv4 = nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3, stride=1, padding=1)
    self.relu4 = nn.ReLU(inplace=True)

    self.conv5 = nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3, stride=1, padding=1)
    self.relu5 = nn.ReLU(inplace=True)
    self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2)

    self.fc1 = nn.Linear(in_features=9216, out_features=4096)
    self.relu6 = nn.ReLU(inplace=True)
    self.dropout1 = nn.Dropout()

    self.fc2 = nn.Linear(in_features=4096, out_features=4096)
    self.relu7 = nn.ReLU(inplace=True)
    self.dropout2 = nn.Dropout()

    self.fc3 = nn.Linear(in_features=4096, out_features=1000)
    self.softmax = nn.Softmax(dim=1)
    # ...

  def forward(self, x):
      x = self.conv1(x)
      x = self.relu1(x)
      x = self.maxpool1(x)

      #print(x.shape)

      x = self.conv2(x)
      x = self.relu2(x)
```

```
        x = self.maxpool2(x)
        #print(x.shape)

        x = self.conv3(x)
        x = self.relu3(x)
        #print(x.shape)

        x = self.conv4(x)
        x = self.relu4(x)
        #print(x.shape)

        x = self.conv5(x)
        x = self.relu5(x)
        x = self.maxpool3(x)
        #print(x.shape)

        x = torch.flatten(x, 1)
        #print("Flattened output - ", x.shape)

        x = self.fc1(x)
        x = self.relu6(x)
        x = self.dropout1(x)
        #print(x.shape)

        x = self.fc2(x)
        x = self.relu7(x)
        x = self.dropout2(x)
        #print(x.shape)

        x = self.fc3(x)
        x = self.softmax(x)
        print(x.shape)

        return x
    pass


# To help you debug, you may want to print the shapes of tensors
# produced after each layer in the network
model = AlexNet()
img = torch.randn((1, 3, 227, 227), dtype=torch.float32)
model(img)

utils.check_q1a(AlexNet())
```

```
    torch.Size([1, 1000])
    torch.Size([32, 1000])
    PASSED
```

Q1b. Customizing AlexNet for Smaller Images

The original AlexNet was designed for 227x227 images. Modify the network to handle 37x37 images while keeping the model parameters constant. In other words, do not change the channel dimensions or kernel sizes. Instead modify the convolution padding or stride or remove maxpool operations.

```
import torch.nn.functional as F
class SmallAlexNet(nn.Module):
  def __init__(self):
    super().__init__()

    self.conv1 = nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=1)
    self.relu1 = nn.ReLU(inplace=True)
    #self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

    #self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2)
    self.conv2 = nn.Conv2d(in_channels=96, out_channels=256, kernel_size=5, stride=1, padding=2)
    self.relu2 = nn.ReLU(inplace=True)
    self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2)

    self.conv3 = nn.Conv2d(in_channels=256, out_channels=384, kernel_size=3, stride=1, padding=1)
    self.relu3 = nn.ReLU(inplace=True)

    self.conv4 = nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3, stride=1, padding=1)
    self.relu4 = nn.ReLU(inplace=True)
```

```
        self.conv5 = nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu5 = nn.ReLU(inplace=True)

        self.maxpool5 = nn.MaxPool2d(kernel_size=3, stride=2)

        #self.dropout1 = nn.Dropout(p=0.5)
        self.fc1 = nn.Linear(in_features=256*6*6, out_features=4096)
        self.relu6 = nn.ReLU(inplace=True)
        self.dropout1 = nn.Dropout()

        #self.dropout2 = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(in_features=4096, out_features=4096)
        self.relu7 = nn.ReLU(inplace=True)
        self.dropout2 = nn.Dropout()

        self.fc3 = nn.Linear(in_features=4096, out_features=1000)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):

        x = self.conv1(x)
        x = self.relu1(x)
        #print(x.shape)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)
        #print(x.shape)

        x = self.conv3(x)
        x = self.relu3(x)
        #print(x.shape)

        x = self.conv4(x)
        x = self.relu4(x)
        #print(x.shape)

        x = self.conv5(x)
        x = self.relu5(x)
        x = self.maxpool5(x)
        #print(x.shape)

        x = torch.flatten(x, 1)
        #print(x.shape)

        x = self.fc1(x)
        x = self.relu6(x)
        x = self.dropout1(x)

        x = self.fc2(x)
        x = self.relu7(x)
        x = self.dropout2(x)

        x = self.fc3(x)
        x = self.softmax(x)

        return x

    pass

model = SmallAlexNet()
#summary(model.cuda(), (3,37,37), device='cuda')
img = torch.randn((1, 3, 37, 37), dtype=torch.float32)
#img = img.to('cuda')
model(img)
utils.check_q1b(SmallAlexNet())
```

```
    PASSED
```

Q2. Implementing Basic Block

One of the most common network architectures for computer vision is the Residual Network (ResNet), first introduced by He et al. (2015). ResNets are constructed by stacking many residual blocks in a row. In this problem, you will implement the BasicBlock, which is used in the

ResNet18 and ResNet34 networks.

```
class BasicBlock(nn.Module):
  def __init__(self, c_in: int, c_out: int, stride):
    """
    x -> conv3x3(c_in, c_out, stride) -> bn -> relu -> conv3x3(c_out, c_out) -> bn -> + -> relu -> out
    |                                                                                 ^
    '------------------> conv1x1(c_in, c_out, stride) -> bn --------------------------'
    """

    super(BasicBlock, self).__init__()

    self.conv1 = nn.Conv2d(c_in, c_out, kernel_size=3, stride=stride, padding=1, bias=False)
    self.bn1 = nn.BatchNorm2d(c_out)
    self.relu1 = nn.ReLU(inplace=True)

    self.conv2 = nn.Conv2d(c_out, c_out, kernel_size=3, stride=1, padding=1, bias=False)
    self.bn2 = nn.BatchNorm2d(c_out)

    self.conv3 = nn.Conv2d(c_in, c_out, kernel_size=1, stride=stride, bias=False)
    self.bn3 = nn.BatchNorm2d(c_out)
    self.relu3 = nn.ReLU(inplace=True)

  def forward(self, x: Tensor) -> Tensor:

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu1(out)

    out = self.conv2(out)
    out = self.bn2(out)

    residual = self.conv3(x)
    residual = self.bn3(residual)

    final = residual + out
    final = self.relu1(final)

    return final
    pass

model = BasicBlock(3, 10, 2)
imgs = [torch.randn((1, 3, 64, 64), dtype=torch.float32),
        torch.randn((1, 3, 90, 90), dtype=torch.float32),
        torch.randn((1, 3, 845, 845), dtype=torch.float32)
        ]
for img in imgs:
  print(f"With an image of size {img.size()}, The model output is {model(img).shape}")
```

```
With an image of size torch.Size([1, 3, 64, 64]), The model output is torch.Size([1, 10, 32, 32])
With an image of size torch.Size([1, 3, 90, 90]), The model output is torch.Size([1, 10, 45, 45])
With an image of size torch.Size([1, 3, 845, 845]), The model output is torch.Size([1, 10, 423, 423])
```

Why are residual connections beneficial when designing deep networks?

*It creates shortcut paths makes it easier to bypass some of the layers in the network.

1. They help to prevent vanishing gradient.
2. Ease of training - provides easier optimization of deeper networks, During backpropagation, gradient becomes extremely small, this might lead to slow learning. But residual can help gradients providing direct paths for the gradients for the back propogation in the network.
3. Faster convergence - As this improves the gradients, networks with residual connections tend to converge faster during training. This is train network faster and improves the performace as well.

*

Why is there a conv1x1 layer in the residual pathway? What purpose does it serve? Hint: see what happens when it is removed.

- 1. Dimensionality Reduction - The feature maps in the input can be reduced using this. Infact 1x1 conv reduces the depth that is number of channels in the input. This allows it to match the output dimensions of the residual.
  2. Parameter reduction - Once, the dimensionality of feature map is reduced then this will help us also reduce the count of the parameters in the model. Resulting in more efficient model and less computational complexity. There is less chance that this model will face overfitting issues.Also, the time taken to train the models will be faster.

3. Non-linearity - Combination of 1x1 conv layer and activation function like Relu adds non-linearity to the residual connection. Model can better understand about complex representations.

-

Q3. Fine Tuning on Smaller Dataset

You are given MobileNetV2 pretrained on ImageNet and need to modify it so that it can train on a smaller dataset of flowers. To do this, you need to switch the network head to a linear layer with the proper output dimension, in this case 102 classes. Then you need to configure the optimizer to apply gradient descent on the head.

```
class MobileNetV2_finetune(pl.LightningModule):
  def __init__(self):
    super().__init__()
    pretrained_model = torchvision.models.mobilenet_v2(
        weights=torchvision.models.MobileNet_V2_Weights
    )

    # self.features = pretrained_model.features
    # self.classifier = nn.Sequential(
    #     nn.Dropout(0.2),
    #     nn.Linear(pretrained_model.last_channel, 102),
    #     nn.Softmax(dim=1)
    # )

    print(pretrained_model.last_channel)
    # MODIFY MODEL HEAD
    pretrained_model.classifier[1] = nn.Linear(in_features=pretrained_model.last_channel, out_features=102, bias=True)
    self.model = pretrained_model

    pass

  def forward(self, x):
    """Performs forward pass

    Arguments
    ---------
    x: Tensor
      image tensor of shape (B, 3, 128, 128)

    Returns
    -------
    Tensor
      logits (ranging from 0 to 1) tensor with shape (B, 102)
    """
    # MAKE SURE TO PERFORM SOFTMAX on output

    x = self.model(x)
    x = torch.softmax(x, dim=1)
    return x
    pass

  def configure_optimizers(self):

    # FIX THIS SO IT ONLY OPTIMIZES CERTAIN PARAMETERS

    #Optimizing only newly added layer while keeping the rest of model unchanged.
    trainable_params = self.model.classifier[1].parameters()
    #print("params",sum(p.numel() for p in trainable_params if p.requires_grad))
    optimizer = torch.optim.Adam(trainable_params, lr=1e-3)
    return optimizer

  def training_step(self, batch, batch_idx):
    x, y = batch
    y_pred = self(x)
    loss = nn.functional.nll_loss(y_pred, y)
    acc = (torch.argmax(y_pred, 1) == y).float().mean()
    self.log("train_acc", acc, prog_bar=True)
    return loss

  def validation_step(self, batch, batch_idx):
    x, y = batch
    y_pred = self(x)
    loss = nn.functional.nll_loss(y_pred, y)
```

```
    acc = (torch.argmax(y_pred, 1) == y).float().mean()
    self.log("val_acc", acc, prog_bar=True)
    return loss


model = MobileNetV2_finetune()
pl.seed_everything(42, workers=True)
train_dl, val_dl = utils.get_flowers_dataloaders(batch_size=512)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu')
trainer.fit(model=model, train_dataloaders=train_dl, val_dataloaders=val_dl)

# this may take about 5 min to run
# you should achieve a validation accuracy of ~0.68 if done correctly
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other t
  warnings.warn(msg)
INFO:lightning_fabric.utilities.seed:Global seed set to 42
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type        | Params
-------------------------------------
0 | model | MobileNetV2 | 2.4 M
-------------------------------------
2.4 M     Trainable params
0         Non-trainable params
2.4 M     Total params
9.418     Total estimated model params size (MB)
1280
```

Epoch 19: 100% 2/2 [00:14<00:00, 7.41s/it, v_num=1, train_acc=0.835, val_acc=0.678]

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=20` reached.
```

```
[link text](https://)# Q4. Data Augmentation to Improve Generalization This is good but I bet we can do better.  Try out some data augmentati see what performance you can get (see full list of supported transform (https://pytorch.org/vision/0.11/transforms.html)).  For two different transformations (or compositions of transformations), explain why you this transformation could improve generalization and what validation a you achieve (it's okay if it does not improve performance).

1.   *
 transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomResizedCrop(size=128, scale=(0.8, 1.0), ratio=(0.
,
    transforms.ColorJitter(brightness=0.2),
])

RandomHorizontalFlip - This will help the model learn to recognize obj regardless of their orientation. I have give the probability of 0.5.
RandomRotation - This will help the model learn to recognize objects f different viewpoints.
RandomResizedCrop - This transformation randomly crops the image and r to 128 x 128. Also changes the aspect ratio -  0.75 and 1.33 and the s between 0.8 and 1.0. Model can learn to recognize objects at different and aspect ratios. Every paramater here is invarient to scale, rotatio ColorJitter - This helps model learn to recognize objects under differ lighting conditions.
Accuracy improved to - val_acc=0.696
2.   Second transform is also similar with change in the values and so parameters -
transforms.Compose([

    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=20),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomResizedCrop(size=128, scale=(0.9, 1.1), ratio=(0.
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.
1),
])
```

link text# Q4. Data Augmentation to Improve Generalization This is good but I bet we can do better. Try out some data augmentations and see what performance you can get (see full list of supported transforms here). For two different transformations (or compositions of transformations), explain why you believe this transformation could improve generalization and what validation accuracy you achieve (it's okay if it does not improve performance).

1. * transforms.Compose([transforms.RandomHorizontalFlip(p=0.5), transforms.RandomRotation(degrees=15), transforms.RandomResizedCrop(size=128, scale=(0.8, 1.0), ratio=(0.75, 1.33)), transforms.ColorJitter(brightness=0.2),])

RandomHorizontalFlip - This will help the model learn to recognize objects regardless of their orientation. I have give the probability of 0.5. RandomRotation - This will help the model learn to recognize objects from different viewpoints. RandomResizedCrop - This transformation randomly crops the image and resizes it to 128 x 128. Also changes the aspect ratio - 0.75 and 1.33 and the scale between 0.8 and 1.0. Model can learn to recognize objects at different scales and aspect ratios. Every paramater here is invarient to scale, rotation. ColorJitter - This helps model learn to recognize objects under different lighting conditions. Accuracy improved to - val_acc=0.696

2. Second transform is also similar with change in the values and some added parameters - transforms.Compose([

   transforms.RandomHorizontalFlip(p=0.5), transforms.RandomRotation(degrees=20), transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)), transforms.RandomResizedCrop(size=128, scale=(0.9, 1.1), ratio=

```
flipping the image horizontally or rotating it by some degree will not
the class label of the image and provides additional training data. It
more label in the dataset. The random affine transformation and random
crop adds variations in the scale and position of objects. This makes
invariant to the changes in size and location. We can also play with
brightness, contrast, saturation, and hue of the input image, This hel
to perform better under different lighting conditions.

Second transform = val_acc=0.702

By applying these transformations to the training data, I was able to
the variation in the dataset, which can help the model learn to genera
better to unseen data.
```

(0.8, 1.2)), transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),])

flipping the image horizontally or rotating it by some degree will not change the class label of the image and provides additional training data. It adds one more label in the dataset. The random affine transformation and random resized crop adds variations in the scale and position of objects. This makes the model invariant to the changes in size and location. We can also play with brightness, contrast, saturation, and hue of the input image, This helps model to perform better under different lighting conditions.

Second transform = val_acc=0.702

By applying these transformations to the training data, I was able to increase the variation in the dataset, which can help the model learn to generalize better to unseen data.

```
# if you want to visualize dataset to think about useful transforms
_, val_dl = utils.get_flowers_dataloaders(batch_size=15)
imgs, _ = next(iter(val_dl))
plt.figure()
grid_img = torchvision.utils.make_grid(imgs, nrow=5, normalize=True).permute((1, 2, 0))
plt.imshow(grid_img)
plt.axis('off')
plt.show()
```

```
# try your first transform here
import torchvision.transforms as transforms

TRAIN_TFM = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomResizedCrop(size=128, scale=(0.8, 1.0), ratio=(0.75, 1.33)),
    transforms.ColorJitter(brightness=0.2),
])

model = MobileNetV2_finetune()
pl.seed_everything(42, workers=True)
train_dl, val_dl = utils.get_flowers_dataloaders(batch_size=512, train_tfm=TRAIN_TFM)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu')
trainer.fit(model=model, train_dataloaders=train_dl, val_dataloaders=val_dl)
```

```
INFO:lightning_fabric.utilities.seed:Global seed set to 42
```

```
# try your second transform here
TRAIN_TFM = transforms.Compose([

    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=20),
    #transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomResizedCrop(size=128, scale=(0.9, 1.1), ratio=(0.8, 1.2)),
    #transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
])

model = MobileNetV2_finetune()
pl.seed_everything(42, workers=True)
train_dl, val_dl = utils.get_flowers_dataloaders(batch_size=512, train_tfm=TRAIN_TFM)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu')
trainer.fit(model=model, train_dataloaders=train_dl, val_dataloaders=val_dl)
```

```
INFO:lightning_fabric.utilities.seed:Global seed set to 42
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
1280
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type        | Params
-------------------------------------
0 | model | MobileNetV2 | 2.4 M
-------------------------------------
2.4 M     Trainable params
0         Non-trainable params
2.4 M     Total params
9.418     Total estimated model params size (MB)
```

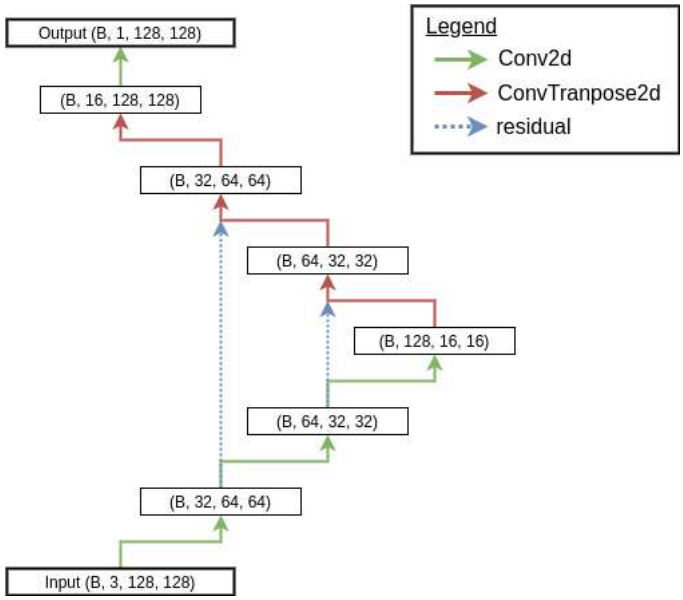
```
Epoch 19: 100%                                                    2/2 [00:15<00:00, 7.66s/it, v_num=6, train_acc=0.839, val_acc=0.702]
```

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=20` reached.
```

```
### This is formatted as code
```

Q5. Segmentation with UNet-style Architecture

Implement a UNet-style network as specified in the diagram below. Place a BatchNorm and ReLU after every conv or convtranspose layer.




```
class UNet(utils.SegmentationModule):
  def __init__(self):
    super().__init__()

    # IMPLEMENT LAYERS HERE
    #Encoder Layer
    self.conv1_encoder = nn.Conv2d(3,32, kernel_size=3, padding=1, stride=2)
    self.bn1_encoder = nn.BatchNorm2d(32)
```

```
        self.relu1 = nn.ReLU(inplace=True)

        self.conv2_encoder = nn.Conv2d(32,64, kernel_size=3, padding=1, stride=2)
        self.bn2_encoder = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU(inplace=True)

        self.conv3_encoder = nn.Conv2d(64,128, kernel_size=3, padding=1, stride=2)
        self.bn3_encoder = nn.BatchNorm2d(128)
        self.relu3 = nn.ReLU(inplace=True)

        self.conv4_encoder = nn.ConvTranspose2d(128,64, kernel_size=3, padding=1, stride=2, output_padding=1)
        self.bn4_encoder = nn.BatchNorm2d(64)
        self.relu4 = nn.ReLU(inplace=True)

        #Decoder Layers

        self.conv1_decoder = nn.ConvTranspose2d(64, 32, kernel_size=3, padding=1, stride=2, output_padding=1)
        self.bn1_decoder = nn.BatchNorm2d(32)
        self.relu5 = nn.ReLU(inplace=True)

        self.conv2_decoder = nn.ConvTranspose2d(32, 16, kernel_size=3, padding=1, stride=2, output_padding=1)
        self.bn2_decoder = nn.BatchNorm2d(16)
        self.relu6 = nn.ReLU(inplace=True)

        self.output = nn.Conv2d(16, 1, kernel_size=1)


    def forward(self, x):
        '''
        Arguments
        ---------
        x : Tensor
            image tensor of shape (B, 3, 128, 128)

        Returns
        -------
        Tensor
            image tensor of shape (B, 1, 128, 128) where every pixel is within (0,1)
            and describes the probability that the pixel is in the foreground
        '''
        encoder1 = self.conv1_encoder(x)
        encoder1 = self.bn1_encoder(encoder1)
        encoder1 = self.relu1(encoder1)
        #print(encoder1.shape)

        encoder2 = self.conv2_encoder(encoder1)
        encoder2 = self.bn2_encoder(encoder2)
        encoder2 = self.relu2(encoder2)
        #print(encoder2.shape)


        encoder3 = self.conv3_encoder(encoder2)
        encoder3 = self.bn3_encoder(encoder3)
        encoder3 = self.relu3(encoder3)
        #print(encoder3.shape)

        middle = self.conv4_encoder(encoder3)
        middle = self.bn4_encoder(middle)
        middle = self.relu4(middle)
        #print(middle.shape)

        # Concatenate enc1 and up1 along the channel dimension
        add_enc2_middle = encoder2 + middle
        #print("added = ",add_enc2_middle.shape)

        decoder1 = self.conv1_decoder(add_enc2_middle)
        decoder1 = self.bn1_decoder(decoder1)
        decoder1 = self.relu5(decoder1)

        add_enc2_decoder1 = encoder1 + decoder1
        #print("added = ",add_enc2_decoder1.shape)

        decoder2 = self.conv2_decoder(add_enc2_decoder1)
        decoder2 = self.bn2_decoder(decoder2)
        decoder2 = self.relu6(decoder2)
        #print(decoder2.shape)
```

```
        output = self.output(decoder2)
        output = torch.sigmoid(output)
        #print("Output shape = ", output.shape)
        return output

        pass


    # run this cell to debug the model using a dummy input
    model = UNet()
    dummy_input = torch.randn((30, 3, 128, 128), dtype=torch.float32)
    output = model(dummy_input)

    # MAKE SURE OUTPUT IMAGE HAS SAME HEIGHT AND WIDTH AS INPUT
    assert output.shape[2:] == dummy_input.shape[2:]

    # MAKE SURE THE OUTPUT IMAGE PIXELS VARY FROM 0 to 1
    assert output.min().item() > 0 and output.max().item() < 1

         torch.Size([30, 32, 64, 64])
         torch.Size([30, 64, 32, 32])
         torch.Size([30, 128, 16, 16])
         torch.Size([30, 64, 32, 32])
         added =  torch.Size([30, 64, 32, 32])
         added =  torch.Size([30, 32, 64, 64])
         torch.Size([30, 16, 128, 128])
         Output shape =  torch.Size([30, 1, 128, 128])


    # Train model once you have debugged the forward pass
    train_dl, val_dl = utils.get_voc_dataloaders(batch_size=256)

    model = UNet()
    pl.seed_everything(42, workers=True)
    trainer = pl.Trainer(max_epochs=10, accelerator='gpu',
                         logger=pl.loggers.TensorBoardLogger('./voc'),
                         log_every_n_steps=1)


    # this will take about 8 minutes
    trainer.fit(model=model, train_dataloaders=train_dl, val_dataloaders=val_dl)
```

```
Using downloaded and verified file: ./VOCtrainval_11-May-2012.tar
Extracting ./VOCtrainval_11-May-2012.tar to ./
Using downloaded and verified file: ./VOCtrainval_11-May-2012.tar
Extracting ./VOCtrainval_11-May-2012.tar to ./
INFO:lightning_fabric.utilities.seed:Global seed set to 42
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
   | Name          | Type            | Params
-------------------------------------------------
0  | acc_metric    | BinaryAccuracy  | 0
1  | criterion     | BCELoss         | 0
2  | conv1_encoder | Conv2d          | 896
3  | bn1_encoder   | BatchNorm2d     | 64
4  | relu1         | ReLU            | 0
5  | conv2_encoder | Conv2d          | 18.5 K
6  | bn2_encoder   | BatchNorm2d     | 128
7  | relu2         | ReLU            | 0
8  | conv3_encoder | Conv2d          | 73.9 K
9  | bn3_encoder   | BatchNorm2d     | 256
10 | relu3         | ReLU            | 0
11 | conv4_encoder | ConvTranspose2d | 73.8 K
12 | bn4_encoder   | BatchNorm2d     | 128
13 | relu4         | ReLU            | 0
14 | conv1_decoder | ConvTranspose2d | 18.5 K
15 | bn1_decoder   | BatchNorm2d     | 64
16 | relu5         | ReLU            | 0
17 | conv2_decoder | ConvTranspose2d | 4.6 K
18 | bn2_decoder   | BatchNorm2d     | 32
19 | relu6         | ReLU            | 0
20 | output        | Conv2d          | 17
-------------------------------------------------
190 K     Trainable params
0         Non-trainable params
190 K     Total params
0.763     Total estimated model params size (MB)


torch.Size([256, 32, 64, 64])
torch.Size([256, 64, 32, 32])
torch.Size([256, 128, 16, 16])
torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 32, 64, 64])
torch.Size([256, 16, 128, 128])
Output shape =  torch.Size([256, 1, 128, 128])
torch.Size([256, 32, 64, 64])
torch.Size([256, 64, 32, 32])
torch.Size([256, 128, 16, 16])
torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 32, 64, 64])
torch.Size([256, 16, 128, 128])
Output shape =  torch.Size([256, 1, 128, 128])
```

Epoch 9: 100% 2/2 [00:06<00:00, 3.10s/it, v_num=3, train_acc=0.751, val_acc=0.756]

```
torch.Size([256, 32, 64, 64])
torch.Size([256, 64, 32, 32])
torch.Size([256, 128, 16, 16])
torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 64, 32, 32])
added =  torch.Size([256, 32, 64, 64])
torch.Size([256, 16, 128, 128])
Output shape =  torch.Size([256, 1, 128, 128])
torch.Size([255, 32, 64, 64])
torch.Size([255, 64, 32, 32])
torch.Size([255, 128, 16, 16])
torch.Size([255, 64, 32, 32])
added =  torch.Size([255, 64, 32, 32])
```