Homework-3

Name: Hanumantha Rao Vakkalanka

Email: hvakkala@uncc.edu

Student ID: 801333188

Course: ECGR 5106 Real Time ML

Lab Number: Spring 2023

```
#Install the d2l library version 1.0.0b0
!pip install d2l==1.0.0b0

#Install the ptflops library
!pip install ptflops

#Import the torch library for PyTorch functionality
import torch

#Import the nn module from PyTorch for building neural networks
from torch import nn

#Import the functional module from nn to use activation functions, etc.
from torch.nn import functional as F
```

```
Looking in indexes: https://pypi.org/simple,
https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: d2l==1.0.0b0 in
/usr/local/lib/python3.9/dist-packages (1.0.0b0)
Requirement already satisfied: gpytorch in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (1.9.1)
Requirement already satisfied: numpy in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (1.22.4)
Requirement already satisfied: pandas in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (1.3.5)
Requirement already satisfied: requests in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (2.25.1)
Requirement already satisfied: gym==0.21.0 in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (0.21.0)
Requirement already satisfied: scipy in
```

```
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (1.10.1)
Requirement already satisfied: jupyter in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (1.0.0)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (3.5.3)
Requirement already satisfied: matplotlib-inline in
/usr/local/lib/python3.9/dist-packages (from d2l==1.0.0b0) (0.1.6)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.9/dist-packages (from gym==0.21.0->d2l==1.0.0b0)
(2.2.1)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.9/dist-packages (from gpytorch->d2l==1.0.0b0) (1.2.1)
Requirement already satisfied: linear-operator>=0.2.0 in
/usr/local/lib/python3.9/dist-packages (from gpytorch->d2l==1.0.0b0) (0.3.0)
Requirement already satisfied: ipykernel in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (5.3.4)
Requirement already satisfied: nbconvert in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (6.5.4)
Requirement already satisfied: ipywidgets in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (7.7.1)
Requirement already satisfied: qtconsole in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (5.4.0)
Requirement already satisfied: jupyter-console in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (6.1.0)
Requirement already satisfied: notebook in
/usr/local/lib/python3.9/dist-packages (from jupyter->d2l==1.0.0b0) (6.3.0)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(4.39.0)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(0.11.0)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0) (23.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(1.4.4)
Requirement already satisfied: pillow>=6.2.0 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(8.4.0)
Requirement already satisfied: pyparsing>=2.2.1 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.9/dist-packages (from matplotlib->d2l==1.0.0b0)
(2.8.2)
Requirement already satisfied: traitlets in
/usr/local/lib/python3.9/dist-packages (from matplotlib-inline->d2l==1.0.0b0)
(5.7.1)
Requirement already satisfied: pytz>=2017.3 in
```

```
/usr/local/lib/python3.9/dist-packages (from pandas->d2l==1.0.0b0) (2022.7.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0)
(1.26.14)
Requirement already satisfied: idna<3,>=2.5 in
/usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in
/usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0) (4.0.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0)
(2022.12.7)
Requirement already satisfied: torch>=1.11 in
/usr/local/lib/python3.9/dist-packages (from
linear-operator>=0.2.0->gpytorch->d2l==1.0.0b0) (1.13.1+cu116)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.9/dist-packages (from
python-dateutil>=2.7->matplotlib->d2l==1.0.0b0) (1.15.0)
Requirement already satisfied: tornado>=4.2 in
/usr/local/lib/python3.9/dist-packages (from
ipykernel->jupyter->d2l==1.0.0b0) (6.2)
Requirement already satisfied: ipython>=5.0.0 in
/usr/local/lib/python3.9/dist-packages (from
ipykernel->jupyter->d2l==1.0.0b0) (7.9.0)
Requirement already satisfied: jupyter-client in
/usr/local/lib/python3.9/dist-packages (from
ipykernel->jupyter->d2l==1.0.0b0) (6.1.12)
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in
/usr/local/lib/python3.9/dist-packages (from
ipywidgets->jupyter->d2l==1.0.0b0) (3.0.5)
Requirement already satisfied: ipython-genutils~=0.2.0 in
/usr/local/lib/python3.9/dist-packages (from
ipywidgets->jupyter->d2l==1.0.0b0) (0.2.0)
Requirement already satisfied: widgetsnbextension~=3.6.0 in
/usr/local/lib/python3.9/dist-packages (from
ipywidgets->jupyter->d2l==1.0.0b0) (3.6.2)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0
in /usr/local/lib/python3.9/dist-packages (from
jupyter-console->jupyter->d2l==1.0.0b0) (2.0.10)
Requirement already satisfied: pygments in
/usr/local/lib/python3.9/dist-packages (from
jupyter-console->jupyter->d2l==1.0.0b0) (2.6.1)
Requirement already satisfied: entrypoints>=0.2.2 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (0.4)
Requirement already satisfied: lxml in /usr/local/lib/python3.9/dist-packages
(from nbconvert->jupyter->d2l==1.0.0b0) (4.9.2)
Requirement already satisfied: defusedxml in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (0.7.1)
Requirement already satisfied: jupyter-core>=4.7 in
```

```
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (5.2.0)
Requirement already satisfied: nbformat>=5.1 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (5.7.3)
Requirement already satisfied: bleach in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (6.0.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (2.1.2)
Requirement already satisfied: beautifulsoup4 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (4.6.3)
Requirement already satisfied: mistune<2,>=0.8.1 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (0.8.4)
Requirement already satisfied: pandocfilters>=1.4.1 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (1.5.0)
Requirement already satisfied: jupyterlab-pygments in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (0.2.2)
Requirement already satisfied: jinja2>=3.0 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (3.1.2)
Requirement already satisfied: nbclient>=0.5.0 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (0.7.2)
Requirement already satisfied: tinycss2 in
/usr/local/lib/python3.9/dist-packages (from
nbconvert->jupyter->d2l==1.0.0b0) (1.2.1)
Requirement already satisfied: terminado>=0.8.3 in
/usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0)
(0.17.1)
Requirement already satisfied: prometheus-client in
/usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0)
(0.16.0)
Requirement already satisfied: argon2-cffi in
/usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0)
(21.3.0)
Requirement already satisfied: pyzmq>=17 in
/usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0)
(23.2.1)
Requirement already satisfied: Send2Trash>=1.5.0 in
/usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0)
(1.8.0)
Requirement already satisfied: qtpy>=2.0.1 in
/usr/local/lib/python3.9/dist-packages (from
qtconsole->jupyter->d2l==1.0.0b0) (2.3.0)
```

```
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.9/dist-packages (from
scikit-learn->gpytorch->d2l==1.0.0b0) (3.1.0)
Requirement already satisfied: joblib>=1.1.1 in
/usr/local/lib/python3.9/dist-packages (from
scikit-learn->gpytorch->d2l==1.0.0b0) (1.2.0)
Requirement already satisfied: decorator in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (4.4.2)
Requirement already satisfied: setuptools>=18.5 in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (57.4.0)
Requirement already satisfied: pexpect in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (4.8.0)
Requirement already satisfied: pickleshare in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (0.7.5)
Requirement already satisfied: backcall in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (0.2.0)
Requirement already satisfied: jedi>=0.10 in
/usr/local/lib/python3.9/dist-packages (from
ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (0.18.2)
Requirement already satisfied: platformdirs>=2.5 in
/usr/local/lib/python3.9/dist-packages (from
jupyter-core>=4.7->nbconvert->jupyter->d2l==1.0.0b0) (3.1.0)
Requirement already satisfied: fastjsonschema in
/usr/local/lib/python3.9/dist-packages (from
nbformat>=5.1->nbconvert->jupyter->d2l==1.0.0b0) (2.16.3)
Requirement already satisfied: jsonschema>=2.6 in
/usr/local/lib/python3.9/dist-packages (from
nbformat>=5.1->nbconvert->jupyter->d2l==1.0.0b0) (4.3.3)
Requirement already satisfied: wcwidth in
/usr/local/lib/python3.9/dist-packages (from
prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->jupyter-console->jupyter->d2l==
1.0.0b0) (0.2.6)
Requirement already satisfied: ptyprocess in
/usr/local/lib/python3.9/dist-packages (from
terminado>=0.8.3->notebook->jupyter->d2l==1.0.0b0) (0.7.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.9/dist-packages (from
torch>=1.11->linear-operator>=0.2.0->gpytorch->d2l==1.0.0b0) (4.5.0)
Requirement already satisfied: argon2-cffi-bindings in
/usr/local/lib/python3.9/dist-packages (from
argon2-cffi->notebook->jupyter->d2l==1.0.0b0) (21.2.0)
Requirement already satisfied: webencodings in
/usr/local/lib/python3.9/dist-packages (from
bleach->nbconvert->jupyter->d2l==1.0.0b0) (0.5.1)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in
```

```
/usr/local/lib/python3.9/dist-packages (from
jedi>=0.10->ipython>=5.0.0->ipykernel->jupyter->d2l==1.0.0b0) (0.8.3)
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0
in /usr/local/lib/python3.9/dist-packages (from
jsonschema>=2.6->nbformat>=5.1->nbconvert->jupyter->d2l==1.0.0b0) (0.19.3)
Requirement already satisfied: attrs>=17.4.0 in
/usr/local/lib/python3.9/dist-packages (from
jsonschema>=2.6->nbformat>=5.1->nbconvert->jupyter->d2l==1.0.0b0) (22.2.0)
Requirement already satisfied: cffi>=1.0.1 in
/usr/local/lib/python3.9/dist-packages (from
argon2-cffi-bindings->argon2-cffi->notebook->jupyter->d2l==1.0.0b0) (1.15.1)
Requirement already satisfied: pycparser in
/usr/local/lib/python3.9/dist-packages (from
cffi>=1.0.1->argon2-cffi-bindings->argon2-cffi->notebook->jupyter->d2l==1.0.0
b0) (2.21)
Looking in indexes: https://pypi.org/simple,
https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: ptflops in
/usr/local/lib/python3.9/dist-packages (0.6.9)
Requirement already satisfied: torch in
/usr/local/lib/python3.9/dist-packages (from ptflops) (1.13.1+cu116)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.9/dist-packages (from torch->ptflops) (4.5.0)
```

```python
# Import the torchvision library for computer vision tasks
import torchvision

# Import the datasets module from torchvision and name it as datasets
import torchvision.datasets as datasets

# Import the transforms module from torchvision for image transformations
from torchvision import transforms

# Import the torch module from d2l library and name it as d2l for deep
learning tasks
from d2l import torch as d2l

# Import the ptflops library for calculating FLOPS (floating point
operations) of a PyTorch model
import ptflops

# Import the get_model_complexity_info function from ptflops library to get
FLOPS
from ptflops import get_model_complexity_info
```

```
# Print the version of the PyTorch library
print(torch.__version__)

# Print the version of the torchvision library
print(torchvision.__version__)

# Import the tensorflow library for deep learning tasks
import tensorflow as tf

# Check the name of the GPU device being used by tensorflow (if any)
tf.test.gpu_device_name()
```

```
1.13.1+cu116
0.14.1+cu116
```

{"type":"string"}

PROBLEM 1(1)

```
class CIFAR10(d2l.DataModule):
    # Define the constructor of the class which takes in the resize and 
batch_size arguments
    def __init__(self, resize, batch_size=64):
        # Call the constructor of the parent class
        super().__init__()
        # Save the hyperparameters
        self.save_hyperparameters()
        # Define a transform composed of resizing and converting the image to 
tensor
        trans = transforms.Compose([transforms.Resize(resize), 
transforms.ToTensor()])
        # Load the CIFAR10 training dataset
        self.train = datasets.CIFAR10(
            root=self.root, train=True, transform=trans, download=True)
        # Load the CIFAR10 validation dataset
        self.val = datasets.CIFAR10(
            root=self.root, train=False, transform=trans, download=True)
```

```python
#Create an instance of the CIFAR10 data module with resized images of (128,
128) and default batch size of 64
data = CIFAR10(resize = (64, 64))

#Print the number of Train_Images in the data module
print("Train_Images = ", len(data.train))

#Print the number of Val_Images in the data module
print("Val_Images = ", len(data.val))

#Print the shape of the first image in the training dataset
data.train[0][0].shape
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
../data/cifar-10-python.tar.gz
```

{"model_id":"fc1766b125d7470c9ab6c90b0199e7d7","version_major":2,"version_min
or":0}

```
Extracting ../data/cifar-10-python.tar.gz to ../data
Files already downloaded and verified
Train_Images =  50000
Val_Images =  10000

torch.Size([3, 64, 64])
```

```python
# Assigning labels to numerical values so its is easier to read
@d2l.add_to_class(CIFAR10)
def txt_labels(self, indices):

  #A list of labels for different categories
  labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']

  #Mapping the numerical label to its corresponding text label
  return [labels[int(i)] for i in indices]

# Function to load either Training or Validation data
@d2l.add_to_class(CIFAR10)
def get_dloader(self, train):

  #Using the training data if train is True, otherwise validation data is
used
  data = self.train if train else self.val

  #Creating a dloader object to load the data in batches
  return torch.utils.data.dloader(data, self.batch_size, shuffle = train,
                                            num_workers = self.num_workers)

# Getting a batch of data and printing its shape and data type
X, Y = next(iterator(data.train_dloader()))
print(X.shape, X.dtype, Y.shape, Y.dtype)
```

```
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dloader.py:554:
UserWarning: This dloader will create 4 worker processes in total. Our
suggested max number of worker in current system is 2, which is smaller than
what this dloader is going to create. Please be aware that excessive worker
creation might get dloader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(

torch.Size([64, 3, 64, 64]) torch.float32 torch.Size([64]) torch.int64
```

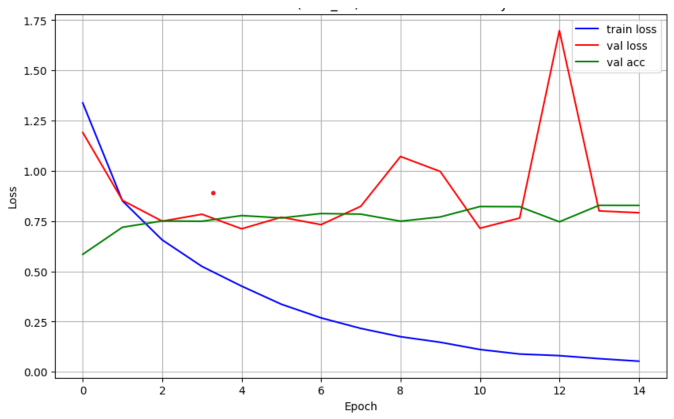
```
#This function defines a VGG block that consists of multiple convolutional
layers followed by a max pooling layer
def vgg_block(no_convs, out_channels):
  layers = []
  for _ in range(no_convs):
    #A convolutional layer is added to the list of layers. The out_channels
parameter specifies the number of output channels, and the kernel_size and
padding parameters specify the size and padding of the convolutional kernel,
respectively.
    layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
    layers.append(nn.ReLU())
  layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
  #The list of layers is converted to a sequential module and returned.
  return nn.Sequential(*layers)

class VGG(d2l.Classifier):

  #Initializing the VGG model with architecture, learning rate, and number of
classes as hyperparameters
  def __init__(self, arch, lr=0.1, num_classes=10):
    super().__init__()
    self.save_hyperparameters()
    #Creating convolutional blocks using the architecture provided
    conv_blks = []
    for (no_convs, out_channels) in arch:
      conv_blks.append(vgg_block(no_convs, out_channels))

      #Defining the overall network architecture by stacking the
convolutional blocks, flattening the output, and adding fully connected
layers with dropout and ReLU activationsless
    self.net = nn.Sequential(
        *conv_blks, nn.Flatten(),
        nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
        nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
        nn.LazyLinear(num_classes))
    #Applying the init_cnn initialization to all layers of the network
    self.net.apply(d2l.init_cnn)

#Defining a method to print the output shape of each layer in the network
@d2l.add_to_class(d2l.Classifier)
def layer_sum(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(type(layer).__name__, 'output shape:\t', X.shape)
#Creating a VGG model with the specified architecture and calling the
layer_sum method to print the output shape of each layer
model = VGG(arch=((1, 64), (1, 128), (1, 256), (1, 512), (1, 512)))
model.layer_sum((128, 3, 64, 64))


#Calculating the computational complexity and number of parameters of the VGG
model and printing the results
macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
print('{:<30}  {:<8}'.format('Computational complexity: ', macs))
print('{:<30}  {:<8}'.format('Number of parameters: ', params))
```

```
/usr/local/lib/python3.9/dist-packages/torch/nn/modules/lazy.py:180:
UserWarning: Lazy modules are a new feature under heavy development so
changes to the API or functionality can happen at any moment.
  warnings.warn('Lazy modules are a new feature under heavy development '

Sequential output shape:       torch.Size([128, 64, 32, 32])
Sequential output shape:       torch.Size([128, 128, 16, 16])
Sequential output shape:       torch.Size([128, 256, 8, 8])
Sequential output shape:       torch.Size([128, 512, 4, 4])
Sequential output shape:       torch.Size([128, 512, 2, 2])
Flatten output shape:    torch.Size([128, 2048])
Linear output shape:     torch.Size([128, 4096])
ReLU output shape:       torch.Size([128, 4096])
Dropout output shape:    torch.Size([128, 4096])
Linear output shape:     torch.Size([128, 4096])
ReLU output shape:       torch.Size([128, 4096])
Dropout output shape:    torch.Size([128, 4096])
Linear output shape:     torch.Size([128, 10])
Sequential(
  29.13 M, 100.000% Params, 298.04 MMac, 100.000% MACs,
  (0): Sequential(
    1.79 k, 0.006% Params, 7.86 MMac, 2.639% MACs,
    (0): Conv2d(1.79 k, 0.006% Params, 7.34 MMac, 2.463% MACs, 3, 64,
kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(0, 0.000% Params, 262.14 KMac, 0.088% MACs, )
    (2): MaxPool2d(0, 0.000% Params, 262.14 KMac, 0.088% MACs, kernel_size=2,
stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (1): Sequential(
    73.86 k, 0.254% Params, 75.89 MMac, 25.463% MACs,
    (0): Conv2d(73.86 k, 0.254% Params, 75.63 MMac, 25.375% MACs, 64, 128,
kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(0, 0.000% Params, 131.07 KMac, 0.044% MACs, )
    (2): MaxPool2d(0, 0.000% Params, 131.07 KMac, 0.044% MACs, kernel_size=2,
stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (2): Sequential(
    295.17 k, 1.013% Params, 75.69 MMac, 25.397% MACs,
    (0): Conv2d(295.17 k, 1.013% Params, 75.56 MMac, 25.353% MACs, 128, 256,
kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(0, 0.000% Params, 65.54 KMac, 0.022% MACs, )
    (2): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.022% MACs, kernel_size=2,
stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (3): Sequential(
    1.18 M, 4.052% Params, 75.6 MMac, 25.364% MACs,
    (0): Conv2d(1.18 M, 4.052% Params, 75.53 MMac, 25.342% MACs, 256, 512,
kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(0, 0.000% Params, 32.77 KMac, 0.011% MACs, )
    (2): MaxPool2d(0, 0.000% Params, 32.77 KMac, 0.011% MACs, kernel_size=2,
stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (4): Sequential(
    2.36 M, 8.102% Params, 37.77 MMac, 12.674% MACs,
    (0): Conv2d(2.36 M, 8.102% Params, 37.76 MMac, 12.668% MACs, 512, 512,
kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(0, 0.000% Params, 8.19 KMac, 0.003% MACs, )
    (2): MaxPool2d(0, 0.000% Params, 8.19 KMac, 0.003% MACs, kernel_size=2,
stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (5): Flatten(0, 0.000% Params, 0.0 Mac, 0.000% MACs, start_dim=1,
end_dim=-1)
  (6): Linear(8.39 M, 28.815% Params, 8.39 MMac, 2.816% MACs,
in_features=2048, out_features=4096, bias=True)
  (7): ReLU(0, 0.000% Params, 4.1 KMac, 0.001% MACs, )
  (8): Dropout(0, 0.000% Params, 0.0 Mac, 0.000% MACs, p=0.5, inplace=False)
  (9): Linear(16.78 M, 57.617% Params, 16.78 MMac, 5.631% MACs,
in_features=4096, out_features=4096, bias=True)
  (10): ReLU(0, 0.000% Params, 4.1 KMac, 0.001% MACs, )
  (11): Dropout(0, 0.000% Params, 0.0 Mac, 0.000% MACs, p=0.5, inplace=False)
  (12): Linear(40.97 k, 0.141% Params, 40.97 KMac, 0.014% MACs,
in_features=4096, out_features=10, bias=True)
)
Computational complexity:       298.04 MMac
Number of parameters:           29.13 M
```

```python
# Function to Evaluate the Model's Accuracy
def ev_acc_gpu(net, data_iterator, device=None):
    if isinstance(net, nn.Module):
        # Set the network to evaluation mode
        net.eval()
        # If device is not specified, use the device of the first parameter
of the network
        if not device:
            device = next(iterator(net.parameters())).device

    # Create an accumulator to store the number of correct predictions and
the number of predictions
    metric = d2l.Accumulator(2)

    with torch.no_grad():
        # iteratorate over each batch in the data iteratorator
        for X, y in data_iterator:
            if isinstance(X, list):
                # Move each tensor in the list to the specified device
                X = [x.to(device) for x in X]
            else:
                # Move the tensor to the specified device
                X = X.to(device)
            # Move the label tensor to the specified device
            y = y.to(device)
            # Compute the network's prediction for the input batch and add
the number of correct predictions and
            # total number of predictions to the accumulator
            metric.add(d2l.accuracy(net(X), y), y.numel())
    # Return the accuracy (number of correct predictions / total number of
predictions)
    return metric[0] / metric[1]

# Training Function
def train(net, train_iterator, val_iterator, no_epochs, lr, device):
    # Initialize the network's weights using Xavier uniform initialization
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
     # Move the network to the specified device
    net.to(device)

    # Create a stochastic gradient descent optimizer to update the network's
weights

    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
```

```
  # Create a cross-entropy loss function to compute the network's loss
    loss = nn.CrossEntropyLoss()

    # Create an animator to visualize the training progress
    animator = d2l.Animator(xlabel='epoch', xlim=[1, no_epochs],
                            legend=['train loss', 'train acc', 'val acc'])
    timer, num_batches = d2l.Timer(), len(train_iterator)

    # Create a timer to measure the training time for each batch and compute
the number of batches
    for epoch in range(no_epochs):

        # Create an accumulator to store the sum of training loss, sum of
training accuracy, and number of examples
        metric = d2l.Accumulator(3)

        # Set the network to training mode
        net.train()
        # iteratorate over each batch in the training data iteratorator
        for i, (X, y) in enumerate(train_iterator):
            timer.start()
            optimizer.zero_grad()
             # Move the input and label tensors to the specified device
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()

             # Update the network's weights using the gradients
            optimizer.step()

            # Compute the training loss, training accuracy, and number of
examples for the input batch and add them to the accumulator
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y),
X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))


        val_acc = ev_acc_gpu(net, val_iterator)
        animator.add(epoch + 1, (None, None, val_acc))

lr, no_epochs = 0.1, 10
```

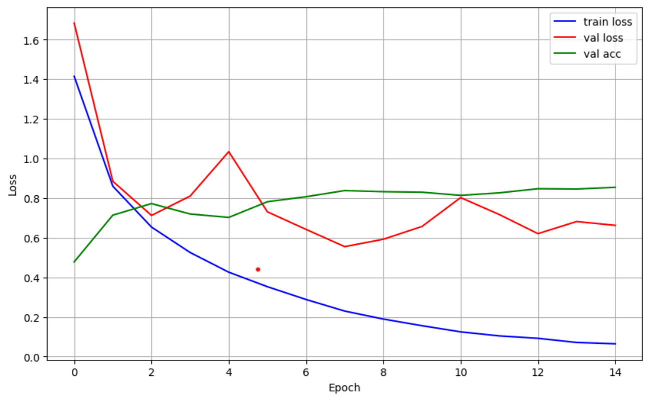
```
train(model.net, data.get_dloader(True), data.get_dloader(False), no_epochs,
lr, d2l.try_gpu())
```



Problem 1(2)

For VGG-16

```
#creates an instance of the VGG class, which represents a VGG network
model_VGG16 = VGG(arch=((2,64), (2,128), (3,256), (3,512), (3,512)))
model_VGG16.layer_sum((64, 3, 64, 64))

#the get_model_complexity_info() function from the ptflops package to compute
the number of floating-point operations (FLOPs) and parameters of the VGG
network
macs, params = ptflops.get_model_complexity_info(model_VGG16.net, (3, 64,
64))
print('{:<30}  {:<8}'.format('Computational complexity: ', macs))
print('{:<30}  {:<8}'.format('Number of parameters: ', params))

# set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

# the train() function with the VGG network model, training dataset,
validation dataset, number of epochs, learning rate, and device (GPU).
train(model_VGG16.net, data.get_dloader(True), data.get_dloader(False),
no_epochs, lr, d2l.try_gpu())
```



b. VGG-19

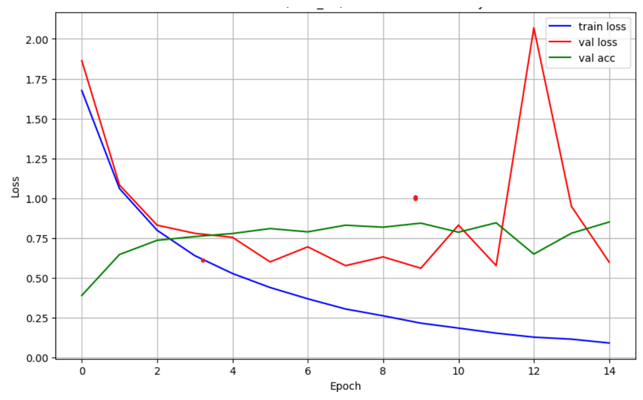
```
import torch.nn as nn
from torchvision import models
class VGG19(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG19, self).__init__()
        self.features = models.vgg19(pretrained=True).features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

          # Construct the VGG network using the convolutional blocks and
fully connected layers

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096), # Fully connected layer with 4096
output neurons
            nn.ReLU(inplace=True),  # ReLU activation
            nn.Dropout(),
            nn.Linear(4096, 4096),  # Fully connected layer with 4096 output
neurons
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),  # Fully connected layer with
'num_class' output neurons
        )

    def fwd(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

model_VGG19 = VGG19()
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
```

```
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior
is equivalent to passing `weights=VGG19_Weights.IMAGENET1K_V1`. You can also
use `weights=VGG19_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to
/root/.cache/torch/hub/checkpoints/vgg19-dcbb9e9d.pth
```

{"model_id":"f1170354c0314a229976f20b03f79834","version_major":2,"version_min
or":0}

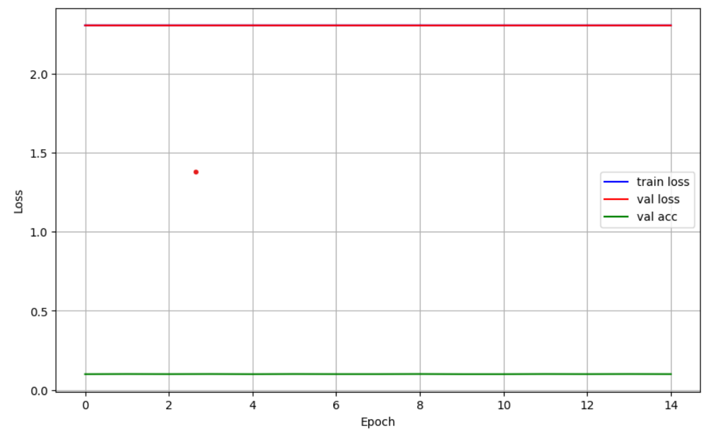
```
#creates an instance of the VGG class, which represents a VGG network
model_VGG19 = VGG(arch=((2,64), (2,128), (4,256), (4,512), (4,512)))

model_VGG19.layer_sum((64, 3, 64, 64))



# prints a summary of the VGG network's layers, given an input tensor of
shape (64, 3, 64, 64)
macs, params = ptflops.get_model_complexity_info(model_VGG19.net, (3, 64,
64))

# print the computed FLOPs and parameters of the VGG network.
print('{:<30}  {:<8}'.format('Computational complexity: ', macs))
print('{:<30}  {:<8}'.format('Number of parameters: ', params))

#set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

# line calls the train() function with the VGG network model, training
dataset, validation dataset, number of epochs, learning rate, and device
(GPU)
train(model_VGG19.net, data.get_dloader(True), data.get_dloader(False),
no_epochs, lr, d2l.try_gpu())
```



Problem 2.1

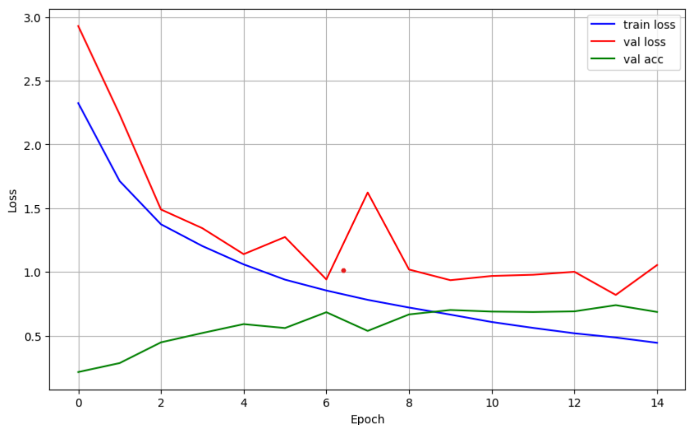
```
# Define a class named Inception_module which is a subclass of nn.Module
class Inception_module(nn.Module):

    # c1--c4 are the number of output channels for each branch
    def _init_(self, c1, c2, c3, c4, **kwargs):
        # Call the constructor of the parent class nn.Module
        super(Inception_module, self)._init_(**kwargs)
        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    # Define the fwd function for the Inception_module module
    def fwd(self, x):
        b1 = F.relu(self.b1_1(x))    # Perform convolution operation on input 
x and apply ReLU activation function
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))    # Perform convolution 
operations on input x and apply ReLU activation functions
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))    # Perform convolution 
operations on input x and apply ReLU activation functions
        b4 = F.relu(self.b4_2(self.b4_1(x)))    # Perform convolution 
operations on input x and apply ReLU activation function
        return torch.cat((b1, b2, b3, b4), dim=1)    # Concatenate the output 
of all the branches and return the output

#Define a class named Inception_module
class Inception_module(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception_module, self).__init__(**kwargs)
        # Define Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        # Define Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        # Define Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        # Define Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)
    # Define fwd function for the class
    def fwd(self, x):
       # fwd pass through Branch 1
        b1 = F.relu(self.b1_1(x))
       # fwd pass through Branch 2
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
       # fwd pass through Branch 3
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
       # fwd pass through Branch 4
        b4 = F.relu(self.b4_2(self.b4_1(x)))
       # Concatenate the outputs from all the branches and return
        return torch.cat((b1, b2, b3, b4), dim=1)



class GoogleNet(d2l.Classifier):
    # Define the first block of the network
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    # Define the second block of the network
    @d2l.add_to_class(GoogleNet)
    def b2(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
            nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    # Define the third block of the network
    @d2l.add_to_class(GoogleNet)
    def b3(self):
        return nn.Sequential(Inception_module(64, (96, 128), (16, 32), 32),
                             Inception_module(128, (128, 192), (32, 96), 64),
                             nn.MaxPool2d(kernel_size=3, stride=2,
padding=1))

    # Define the fourth block of the network
    @d2l.add_to_class(GoogleNet)
    def b4(self):
        return nn.Sequential(Inception_module(192, (96, 208), (16, 48), 64),
                             Inception_module(160, (112, 224), (24, 64), 64),
                             Inception_module(128, (128, 256), (24, 64), 64),
                             Inception_module(112, (144, 288), (32, 64), 64),
                             Inception_module(256, (160, 320), (32, 128),
128),
                             nn.MaxPool2d(kernel_size=3, stride=2,
padding=1))
```

```
    # Define the fifth block of the network
    @d2l.add_to_class(GoogleNet)
    def b5(self):
        return nn.Sequential(Inception_module(256, (160, 320), (32, 128),
128),
                             Inception_module(384, (192, 384), (48, 128),
128),
                             nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())

    # Define the network architecture and initialize it
    @d2l.add_to_class(GoogleNet)
    def __init__(self, lr=0.1, num_classes=10):
        super(GoogleNet, self).__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                                 self.b5(), nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)




#create an instance of the GoogleNet class called model_GoogleNet
model_GoogleNet = GoogleNet()

#use the layer_sum function to print out a summary of the layers in the
model_GoogleNet network
model_GoogleNet.layer_sum((64, 3, 64, 64))

#use ptflops library to calculate the computational complexity and number of
parameters in the network
macs, params = ptflops.get_model_complexity_info(model_GoogleNet.net, (3, 64,
64))

#print the computational complexity and number of parameters of the network
using the values obtained from the previous line
print('{:<30} {:<8}'.format('Computational complexity: ', macs))
print('{:<30} {:<8}'.format('Number of parameters: ', params))

#set the learning rate and number of epochs for training the model
lr, no_epochs = 0.01, 10

#call the train function to train the model_GoogleNet network using the
training data and validation data loaders
train(model_GoogleNet.net, data.get_dloader(True), data.get_dloader(False),
no_epochs, lr, d2l.try_gpu())
```



2.2 With Batch Normalization

```
class Inception_module(nn.Module):
    # c1--c4 are the number of output channels for each branch

    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception_module, self).__init__(**kwargs)

        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)  # 1x1 Convolutional
layer with c1 output channels
        self.bn1_1 = nn.BNorm2d(c1)  # Batch Normalization layer with c1
output channels

        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)  # 1x1 Convolutional
layer with c2[0] output channels
        self.bn2_1 = nn.BNorm2d(c2[0])  # Batch Normalization layer with
c2[0] output channels
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)  # 3x3
Convolutional layer with c2[1] output channels and padding of 1
        self.bn2_2 = nn.BNorm2d(c2[1])  # Batch Normalization layer with
c2[1] output channels

        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)  # 1x1 Convolutional
layer with c3[0] output channels
        self.bn3_1 = nn.BNorm2d(c3[0])  # Batch Normalization layer with
c3[0] output channels
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)  # 5x5
Convolutional layer with c3[1] output channels and padding of 2
        self.bn3_2 = nn.BNorm2d(c3[1])  # Batch Normalization layer with
c3[1] output channels

        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)  # Max
pooling layer with kernel size of 3x3, stride of 1 and padding of 1
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)  # 1x1 Convolutional
layer with c4 output channels
        self.bn4_2 = nn.BNorm2d(c4)  # Batch Normalization layer with c4
output channels

    def fwd(self, x):
        b1 = F.relu(self.b1_1(x))  # Pass input through the 1st branch and
apply ReLU activation function
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))  # Pass input through
the 2nd branch, apply ReLU activation function, pass output through 3x3
Convolutional layer and apply ReLU activation function
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))  # Pass input through
the 3rd branch, apply ReLU activation function, pass output through 5x5
Convolutional layer and apply ReLU activation function
        b4 = F.relu(self.b4_2(self.b4_1(x)))  # Pass input through the 4th
branch, apply Max pooling and then apply 1x1 Convolutional layer with ReLU
activation function
        return



# Define a class called GoogleNet_BNorm, which inherits from d2l.Classifier
class GoogleNet_BNorm(d2l.Classifier):

  # Define a function called b1, which returns a sequence of convolutional
and pooling layers
  def b1(self):
    return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
nn.BNorm2d(64),
        nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

  # Define a function called b2, which returns a sequence of convolutional
and pooling layers

  @d2l.add_to_class(GoogleNet_BNorm)
  def b2(self):
    return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=1), nn.BNorm2d(64), nn.ReLU(),
        nn.LazyConv2d(192, kernel_size=3, padding=1), nn.BNorm2d(192),
nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

  # Define a function called b3, which returns a sequence of Inception_module
blocks and pooling layers

  @d2l.add_to_class(GoogleNet_BNorm)
  def b3(self):
    return nn.Sequential(Inception_module(64, (96, 128), (16, 32), 32),
                         Inception_module(128, (128, 192), (32, 96), 64),
                         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

  # Define a function called b4, which returns a sequence of Inception_module
blocks and pooling layers

  @d2l.add_to_class(GoogleNet_BNorm)
  def b4(self):
    return nn.Sequential(Inception_module(192, (96, 208), (16, 48), 64),
                         Inception_module(160, (112, 224), (24, 64), 64),
                         Inception_module(128, (128, 256), (24, 64), 64),
                         Inception_module(112, (144, 288), (32, 64), 64),
                         Inception_module(256, (160, 320), (32, 128), 128),
                         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

  # Define a function called b5, which returns a sequence of Inception_module
blocks and pooling layers

  @d2l.add_to_class(GoogleNet_BNorm)
  def b5(self):
    return nn.Sequential(Inception_module(256, (160, 320), (32, 128), 128),
                         Inception_module(384, (192, 384), (48, 128), 128),
                         nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())



#Create an instance of the GoogleNet_BNorm class
model = GoogleNet_BNorm()

#Print a summary of the layers of the model for an input tensor of size (64,
3, 64, 64)
model.layer_sum((64, 3, 64, 64))

#Calculate the model's computational complexity (in MACs) and number of
parameters using the ptflops package
macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
print('{:<30} {:<8}'.format('Computational complexity: ', macs))
print('{:<30} {:<8}'.format('Number of parameters: ', params))

#Set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

#Train the model using the train function with training and validation data
loaders, learning rate, and number of epochs
train(model.net, data.get_dloader(True), data.get_dloader(False), no_epochs,
lr, d2l.try_gpu())
```

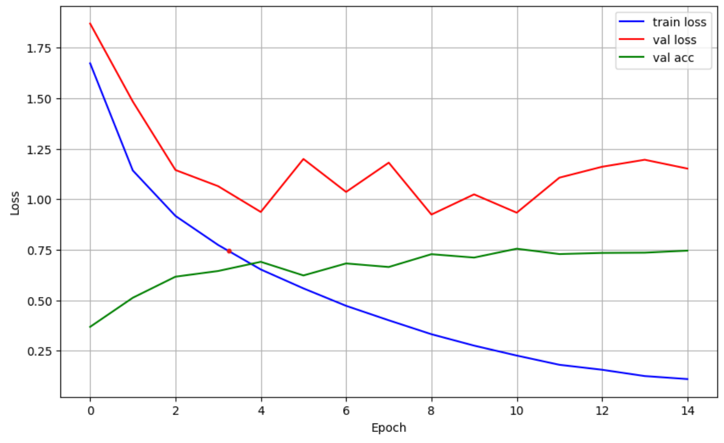


3.1

```
class Res(nn.Module):
    """The Res block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()  # initialize parent class
        # First convolutional layer with 3x3 kernel, stride and padding = 
strides, num_channels input channels
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1, 
stride=strides)
        # Second convolutional layer with 3x3 kernel and num_channels 
input/output channels, stride and padding = 1
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        # Optional convolutional layer with 1x1 kernel, num_channels 
input/output channels, and stride = strides
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, 
stride=strides)
        else:
            self.conv3 = None
        # Batch normalization layer after the first convolutional layer
        self.bn1 = nn.LazyBNorm2d()
        # Batch normalization layer after the second convolutional layer
        self.bn2 = nn.LazyBNorm2d()


    def fwd(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))  # 1st convolution -> batch norm 
-> ReLU
        Y = self.bn2(self.conv2(Y))  # 2nd convolution -> batch norm
        if self.conv3:
            X = self.conv3(X)
        Y += X  # Res connection
        return F.relu(Y)  # final ReLU activation
```

```
#Define a ResNet model as a subclass of d2l.Classifier.
return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),  # 1st
convolutional layer
            nn.LazyBNorm2d(), nn.ReLU(),  # batch normalization -> ReLU
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))  # max pooling
layer


#Define the first block of the ResNet model.
@d2l.add_to_class(ResNet)
def block(self, num_Ress, num_channels, first_block=False):
    blk = []
    for i in range(num_Ress):
        if i == 0 and not first_block:
            blk.append(Res(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Res(num_channels))
    return nn.Sequential(*blk)


#Define a block of Res layers that will be used in the ResNet model.
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, b in enumerate(arch):
        self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
    self.net.add_module('last', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)
#Define the initializer of the ResNet model.
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes)




#Create an instance of the GoogleNet_BNorm class
model = GoogleNet_BNorm()

#Print a summary of the layers of the model for an input tensor of size (64,
3, 64, 64)
model.layer_sum((64, 3, 64, 64))
```

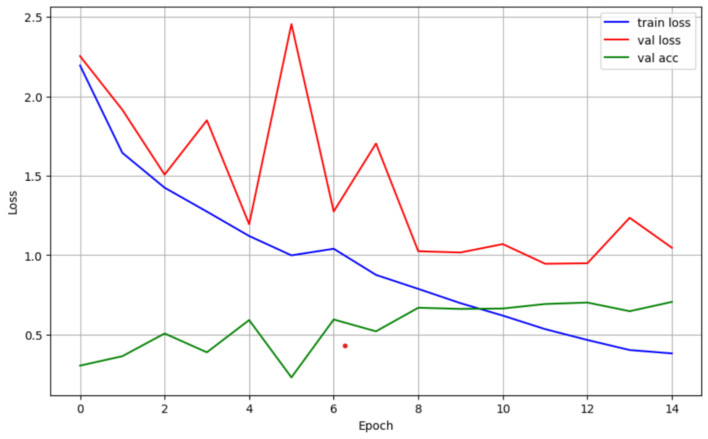
```
#Calculate the model's computational complexity (in MACs) and number of
parameters using the ptflops package
macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
print('{:<30} {:<8}'.format('Computational complexity: ', macs))
print('{:<30} {:<8}'.format('Number of parameters: ', params))

#Set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

#Train the model using the train function with training and validation data
loaders, learning rate, and number of epochs
train(model.net, data.get_dloader(True), data.get_dloader(False), no_epochs,
lr, d2l.try_gpu())
```



3.2

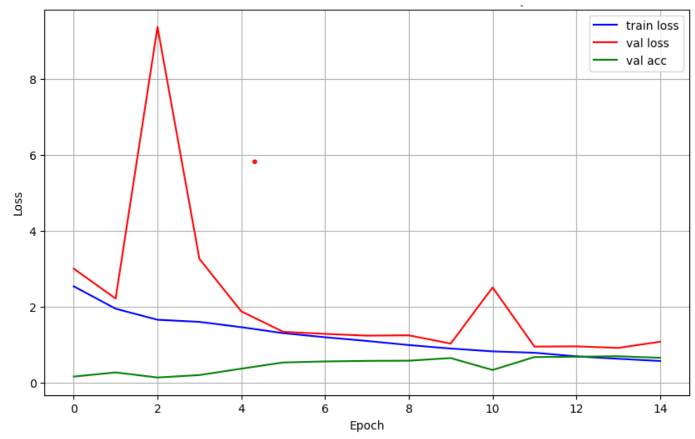
```
#Defines a class ResNet26 that inherits from the ResNet class.
class ResNet26(ResNet):
    #Defines a constructor method for the ResNet26 class that takes two
optional arguments
    def __init__(self, lr=0.1, num_classes=10):
       #Calls the constructor of the parent ResNet class
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes)

#Create an instance
model = ResNet26()

#Print a summary of the layers of the model for an input tensor of size (64,
3, 64, 64)
model.layer_sum((64, 3, 64, 64))

#Calculate the model's computational complexity (in MACs) and number of
```

```
parameters using the ptflops package
macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
print('{:<30} {:<8}'.format('Computational complexity: ', macs))
print('{:<30} {:<8}'.format('Number of parameters: ', params))

#Set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

#Train the model using the train function with training and validation data
loaders, learning rate, and number of epochs
train(model.net, data.get_dloader(True), data.get_dloader(False), no_epochs,
lr, d2l.try_gpu())
```



```
#Defines a class ResNet32 that inherits from the ResNet class.
class ResNet32(ResNet):
    #Defines a constructor method for the ResNet26 class that takes two
optional arguments
    def __init__(self, lr=0.1, num_classes=10):
       #Calls the constructor of the parent ResNet class
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes)


#Create an instance
model = ResNet32()

#Print a summary of the layers of the model for an input tensor of size (64,
3, 64, 64)
model.layer_sum((64, 3, 64, 64))

#Calculate the model's computational complexity (in MACs) and number of
parameters using the ptflops package
macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
```

```
print('{:<30} {:<8}'.format('Computational complexity: ', macs))
print('{:<30} {:<8}'.format('Number of parameters: ', params))

#Set the learning rate and number of epochs for training
lr, no_epochs = 0.01, 10

#Train the model using the train function with training and validation data
loaders, learning rate, and number of epochs
train(model.net, data.get_dloader(True), data.get_dloader(False), no_epochs,
lr, d2l.try_gpu())
```


train loss
val loss
val acc
Loss
Epoch