Real Time Machine Learning - Homework 3

Name : Tarun Reddy Challa

Student ID : 801318301

GitHub Link : https://github.com/tarunreddy03/RTML

```
!pip install d2l==1.0.0b0
!pip install ptflops
import torch
from torch import nn
from torch.nn import functional as F
import torchvision
import torchvision.datasets as datasets
from torchvision import transforms
from d2l import torch as d2l
import ptflops
from ptflops import get_model_complexity_info
```

```
print(torch.__version__)
print(torchvision.__version__)
```

Saved successfully!

```
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->d2l==1.0.0b0) (2.1
Requirement already satisfied: torch>=1.11 in /usr/local/lib/python3.9/dist-packages (from linear-operator>=0.2.0->gpyto
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.7->matplotlib
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.9/dist-packages (from ipykernel->jupyter->d2l==1
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.9/dist-packages (from ipykernel->jupyter->d2l==1.0
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.9/dist-packages (from ipykernel->jupyter->d2l==1
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.9/dist-packages (from ipywidgets->jup
Requirement already satisfied: ipython-genutils~=0.2.0 in /usr/local/lib/python3.9/dist-packages (from ipywidgets->jupyt
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.9/dist-packages (from ipywidgets->jup
Requirement already satisfied: pygments in /usr/local/lib/python3.9/dist-packages (from jupyter-console->jupyter->d2l==1
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 (f
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==
Requirement already satisfied: lxml in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.0.0b0) (4
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.0.
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.0.0b0
Requirement already satisfied: defusedxml in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.0.0
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d
Requirement already satisfied: bleach in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.0.0b0)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.9/dist-packages (from nbconvert->jupyter->d2l==1.
Requirement already satisfied: pyzmq>=17 in /usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0b0
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==1.0.0
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l=
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l==
Requirement already satisfied: Send2Trash>=1.5.0 in /usr/local/lib/python3.9/dist-packages (from notebook->jupyter->d2l=
Collecting qtpy>=2.0.1
  Downloading QtPy-2.3.0-py3-none-any.whl (83 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 83.6/83.6 KB 5.7 MB/s eta 0:00:00
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.9/dist-packages (from scikit-learn->gpytor
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.9/dist-packages (from scikit-learn->gpytorch->d2l
Requirement already satisfied: pexpect in /usr/local/lib/python3.9/dist-packages (from ipython>=5.0.0->ipykernel->jupyte
Requirement already satisfied: backcall in /usr/local/lib/python3.9/dist-packages (from ipython>=5.0.0->ipykernel->jupyt
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.9/dist-packages (from ipython>=5.0.0->ipykerne
Requirement already satisfied: decorator in /usr/local/lib/python3.9/dist-packages (from ipython>=5.0.0->ipykernel->jupy
Requirement already satisfied: pickleshare in /usr/local/lib/python3.9/dist-packages (from ipython>=5.0.0->ipykernel->ju
Collecting jedi>=0.10
  Downloading jedi-0.18.2-py2.py3-none-any.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 19.5 MB/s eta 0:00:00
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.9/dist-packages (from jupyter-core>=4.7->nbco
Requirement already satisfied: fastjsonschema in /usr/local/lib/python3.9/dist-packages (from nbformat>=5.1->nbconvert->
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.9/dist-packages (from nbformat>=5.1->nbconvert-
Requirement already satisfied: wcwidth in /usr/local/lib/python3.9/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.9/dist-packages (from terminado>=0.8.3->notebook->ju
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch>=1.11->linear-ope
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/python3.9/dist-packages (from argon2-cffi->noteboo
Requirement already satisfied: webencodings in /usr/local/lib/python3.9/dist-packages (from bleach->nbconvert->jupyter->
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /usr/local/lib/python3.9/dist-packages (from jedi>=0.10->ipython>=
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.9/dist-packages (from jsonschema>=2.6->nbformat>=
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in /usr/local/lib/python3.9/dist-packages (
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from argon2-cffi-bindings->argon2-
Requirement already satisfied: pycparser in /usr/local/lib/python3.9/dist-packages (from cffi>=1.0.1->argon2-cffi-bindin
Building wheels for collected packages: gym
  Building wheel for gym (setup.py) ... done
  Created wheel for gym: filename=gym-0.21.0-py3-none-any.whl size=1616823 sha256=e2dd32f53bb277241a569113bfb53cb5cf427f
  Stored in directory: /root/.cache/pip/wheels/b3/50/6c/0a82c1358b4da2dbd9c1bb17e0f89467db32812ab236dbf6d5
Successfully built gym
Installing collected packages: qtpy, matplotlib-inline, jedi, gym, linear-operator, gpytorch, qtconsole, jupyter, d2l
  Attempting uninstall: gym
    Found existing installation: gym 0.25.2
    Uninstalling gym-0.25.2:
      Successfully uninstalled gym-0.25.2
Successfully installed d2l-1.0.0b0 gpytorch-1.9.1 gym-0.21.0 jedi-0.18.2 jupyter-1.0.0 linear-operator-0.3.0 matplotlib-
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting ptflops
  Downloading ptflops-0.6.9.tar.gz (12 kB)
  Preparing metadata (setup.py) ... done
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)
Building wheels for collected packages: ptflops
  Building wheel for ptflops (setup.py) ... done
  Created wheel for ptflops: filename=ptflops-0.6.9-py3-none-any.whl size=11712 sha256=10f84dbbf963e2089bbcd79a9d8db7d9a
```

Saved successfully!

PROBLEM 1

1.1 Train the based line VGG model we need for FashinMNIST on CIFAR-10 and report your classification accuracy for validation set, as well as training loss and training accuracy. For this training resize the network input to 64*64 resolution.

```
class CIFAR10(d2l.DataModule):
    def __init__(self, resize, batch_size=64):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize), transforms.ToTensor()])
        self.train = datasets.CIFAR10(
            root=self.root, train=True, transform=trans, download=True)
        self.val = datasets.CIFAR10(
            root=self.root, train=False, transform=trans, download=True)


data = CIFAR10(resize = (64, 64))
print("Training Images   = ", len(data.train))
print("Validation Images = ", len(data.val))

data.train[0][0].shape
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data/cifar-10-python.tar.gz
100%  170498071/170498071 [00:04<00:00, 33409737.36it/s]
Extracting ../data/cifar-10-python.tar.gz to ../data
Files already downloaded and verified
Training Images   =  50000
Validation Images =  10000
```



```
# Define text labels for CIFAR10 classes
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Define a custom DataLoader class for CIFAR10
class CIFAR10DataLoader(torch.utils.data.DataLoader):
    def __init__(self, cifar10, batch_size, train=True, num_workers=0):
        self.dataset = cifar10.train if train else cifar10.val
        super().__init__(self.dataset, batch_size, shuffle=train, num_workers=num_workers)

    def __iter__(self):
        for X, y in super().__iter__():
            yield X, [labels[int(i)] for i in y]

# Load CIFAR10 data
cifar10 = d2l.data.CIFAR10()
train_loader = CIFAR10DataLoader(cifar10, batch_size=32, train=True)
test_loader = CIFAR10DataLoader(cifar10, batch_size=32, train=False)

# Get a batch of data from the training loader
X, Y = next(iter(train_loader))
print(X.shape, X.dtype, Y.shape, Y.dtype)
```

```
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:554: UserWarning: This DataLoader will create 4 worker processes i
  warnings.warn(_create_warning_msg(
torch.Size([64, 3, 64, 64]) torch.float32 torch.Size([64]) torch.int64
```

```
def vgg_block(num_convs, out_channels):
  layers = []
  for _ in range(num_convs):
    layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
    layers.append(nn.ReLU())
  layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
  return nn.Sequential(*layers)



class VGG(d2l.Classifier):
  def __init__(self, arch, lr=0.1, num_classes=10):
    super().__init__()
    self.save_hyperparameters()
    conv_blks = []
    for (num_convs, out_channels) in arch:
      conv_blks.append(vgg_block(num_convs, out_channels))
    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))
        self.net.apply(d2l.init_cnn)

    @d2l.add_to_class(d2l.Classifier)
    def layer_summary(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)

    model = VGG(arch=((1, 64), (1, 128), (1, 256), (1, 512), (1, 512)))
    model.layer_summary((64, 3, 64, 64))

    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 developr
      warnings.warn('Lazy modules are a new feature under heavy development '
    Sequential output shape:         torch.Size([64, 64, 32, 32])
    Sequential output shape:         torch.Size([64, 128, 16, 16])
    Sequential output shape:         torch.Size([64, 256, 8, 8])
    Sequential output shape:         torch.Size([64, 512, 4, 4])
    Sequential output shape:         torch.Size([64, 512, 2, 2])
                                     Size([64, 2048])
                                     Size([64, 4096])
                                     Size([64, 4096])
    Dropout output shape:    torch.Size([64, 4096])
    Linear output shape:     torch.Size([64, 4096])
    ReLU output shape:       torch.Size([64, 4096])
    Dropout output shape:    torch.Size([64, 4096])
    Linear output shape:     torch.Size([64, 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
```

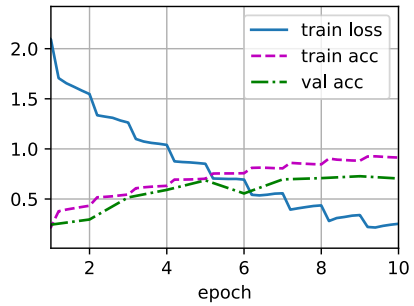
Saved successfully!

```
# Function to Evaluate the Model's Accuracy
def evaluate_accuracy(net, data_iter, device):
    net.eval()
```

```
    metric = d2l.Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            X, y = X.to(device), y.to(device)
            metric.add(d2l.accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]

# Training Function
def train(net, train_iter, val_iter, num_epochs, lr, device):
    net.apply(d2l.init_weights)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'val acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
```

Saved successfully!

```
                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 = evaluate_accuracy(net, val_iter, device)
        animator.add(epoch + 1, (None, None, val_acc))

lr, num_epochs = 0.1, 10
device = d2l.try_gpu()
train(model.net, model.get_dataloader(True), model.get_dataloader(False), num_epochs, lr, device)
```



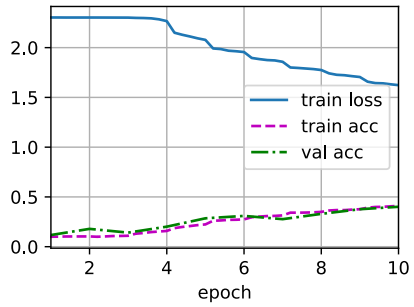
Conclusion: The network input was adjusted to 64*64 and the CIFAR10 dataset was loaded first. Then the baseline VGG model was used to train CIFAR10. The image below displays the computational complexity, the number of parameters, and the training loss, training accuracy, and validation accuracy values. The graph shows that although the training loss is significantly declining, which is a very positive indication, the training and validation accuracy are only slightly diverging, which suggests that the method is not generalizable.

1.2 Use Table 1 in the VGG paper (Simonyan and Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19. Train them on CIFAR-10, compare the accuracies, computational complexity and model size.

a. VGG-16

```
model_VGG16 = VGG(arch=((2,64), (2,128), (3,256), (3,512), (3,512)))
model_VGG16.layer_summary((64, 3, 64, 64))

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))
```

```
lr, num_epochs = 0.01, 10
train(model_VGG16.net, data.get_dataloader(True), data.get_dataloader(False), num_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))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

    def forward(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 depreca
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.cache/torch/hub/checkpoints/vgg19-dcbb9e9d.
100%                                      548M/548M [00:02<00:00, 171MB/s]
```

```
model_VGG19 = VGG(arch=((2, 64), (2, 128), (4, 256), (4, 512), (4, 512)))
model_VGG19.layer_summary((64, 3, 64, 64))

macs, params = ptflops.get_model_complexity_info(model_VGG19.net, (3, 64, 64))
print('{:<30}  {:<8}'.format('Computational complexity: ', macs))
print('{:<30}  {:<8}'.format('Number of parameters: ', params))

lr, num_epochs = 0.01, 10
train(model_VGG19.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```

Conclusion: Designing and using the VGG network is very straightforward. However, there are some problems with its ability to extend to test data like that shown in the graphs. The generalization to the test data is much superior though because it scales, like in VGG-19. VGG-19, on the other hand, necessitates a significantly higher number of factors and a longer training period. Very little has changed in this situation.

PROBLEM 2

1.Use the CIFAR-10 to train a baseline classifier based on the GoogleNet model we did in lectures for 64*64 input resolution. Report your classification accuracy for the validation set, as well as training loss and training accuracy.

```
class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def _init_(self, c1, c2, c3, c4, **kwargs):
        super(Inception, 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)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)


class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, 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)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)

class GoogleNet(d2l.Classifier):
  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))

@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))

@d2l.add_to_class(GoogleNet)
def b3(self):
    return nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                         Inception(128, (128, 192), (32, 96), 64),
```

```
                        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

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

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

@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)
```

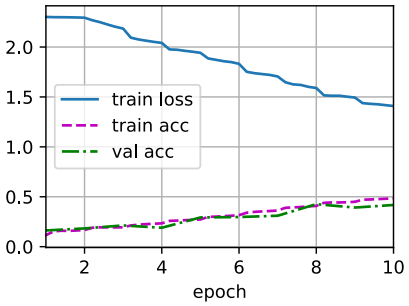
Saved successfully!

```
, 64, 64))

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

```
     Sequential output shape:         torch.Size([64, 64, 16, 16])
     Sequential output shape:         torch.Size([64, 192, 8, 8])
     Sequential output shape:         torch.Size([64, 480, 4, 4])
     Sequential output shape:         torch.Size([64, 832, 2, 2])
     Sequential output shape:         torch.Size([64, 1024])
     Linear output shape:     torch.Size([64, 10])
     Sequential(
       5.98 M, 100.000% Params, 129.76 MMac, 100.000% MACs,
       (0): Sequential(
         9.47 k, 0.158% Params, 9.83 MMac, 7.576% MACs,
         (0): Conv2d(9.47 k, 0.158% Params, 9.7 MMac, 7.475% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
         (1): ReLU(0, 0.000% Params, 65.54 KMac, 0.051% MACs, )
         (2): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.051% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
       )
       (1): Sequential(
         114.94 k, 1.921% Params, 29.54 MMac, 22.766% MACs,
         (0): Conv2d(4.16 k, 0.070% Params, 1.06 MMac, 0.821% MACs, 64, 64, kernel_size=(1, 1), stride=(1, 1))
         (1): ReLU(0, 0.000% Params, 16.38 KMac, 0.013% MACs, )
         (2): Conv2d(110.78 k, 1.851% Params, 28.36 MMac, 21.857% MACs, 64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (3): ReLU(0, 0.000% Params, 49.15 KMac, 0.038% MACs, )
         (4): MaxPool2d(0, 0.000% Params, 49.15 KMac, 0.038% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
       )
       (2): Sequential(
         552.43 k, 9.232% Params, 35.42 MMac, 27.293% MACs,
         (0): Inception(
           163.7 k, 2.736% Params, 10.49 MMac, 8.083% MACs,
           (b1_1): Conv2d(12.35 k, 0.206% Params, 790.53 KMac, 0.609% MACs, 192, 64, kernel_size=(1, 1), stride=(1, 1))
           (b2_1): Conv2d(18.53 k, 0.310% Params, 1.19 MMac, 0.914% MACs, 192, 96, kernel_size=(1, 1), stride=(1, 1))
           (b2_2): Conv2d(110.72 k, 1.850% Params, 7.09 MMac, 5.461% MACs, 96, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (b3_1): Conv2d(3.09 k, 0.052% Params, 197.63 KMac, 0.152% MACs, 192, 16, kernel_size=(1, 1), stride=(1, 1))
           (b3_2): Conv2d(12.83 k, 0.214% Params, 821.25 KMac, 0.633% MACs, 16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
           (b4_1): MaxPool2d(0, 0.000% Params, 12.29 KMac, 0.009% MACs, kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
           (b4_2): Conv2d(6.18 k, 0.103% Params, 395.26 KMac, 0.305% MACs, 192, 32, kernel_size=(1, 1), stride=(1, 1))
         )
         (1): Inception(
           388.74 k, 6.496% Params, 24.9 MMac, 19.186% MACs,
           (b1_1): Conv2d(32.9 k, 0.550% Params, 2.11 MMac, 1.623% MACs, 256, 128, kernel_size=(1, 1), stride=(1, 1))
           (b2_1): Conv2d(32.9 k, 0.550% Params, 2.11 MMac, 1.623% MACs, 256, 128, kernel_size=(1, 1), stride=(1, 1))
           (b2_2): Conv2d(221.38 k, 3.700% Params, 14.17 MMac, 10.919% MACs, 128, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (b3_1): Conv2d(8.22 k, 0.137% Params, 526.34 KMac, 0.406% MACs, 256, 32, kernel_size=(1, 1), stride=(1, 1))
           (b3_2): Conv2d(76.9 k, 1.285% Params, 4.92 MMac, 3.793% MACs, 32, 96, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
           (b4_1): MaxPool2d(0, 0.000% Params, 16.38 KMac, 0.013% MACs, kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
           (b4_2): Conv2d(16.45 k, 0.275% Params, 1.05 MMac, 0.811% MACs, 256, 64, kernel_size=(1, 1), stride=(1, 1))
         )
         (2): MaxPool2d(0, 0.000% Params, 30.72 KMac, 0.024% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
       )
       (3): Sequential(
         2.81 M, 46.946% Params, 45.0 MMac, 34.681% MACs,
         (0): Inception(
```

```
    376.18 k, 6.287% Params, 6.03 MMac, 4.644% MACs,
    (b1_1): Conv2d(92.35 k, 1.543% Params, 1.48 MMac, 1.139% MACs, 480, 192, kernel_size=(1, 1), stride=(1, 1))
    (b2_1): Conv2d(46.18 k, 0.772% Params, 738.82 KMac, 0.569% MACs, 480, 96, kernel_size=(1, 1), stride=(1, 1))
    (b2_2): Conv2d(179.92 k, 3.007% Params, 2.88 MMac, 2.219% MACs, 96, 208, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b3_1): Conv2d(7.7 k, 0.129% Params, 123.14 KMac, 0.095% MACs, 480, 16, kernel_size=(1, 1), stride=(1, 1))
    (b3_2): Conv2d(19.25 k, 0.322% Params, 307.97 KMac, 0.237% MACs, 16, 48, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (b4_1): MaxPool2d(0, 0.000% Params, 7.68 KMac, 0.006% MACs, kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
    (b4_2): Conv2d(30.78 k, 0.514% Params, 492.54 KMac, 0.380% MACs, 480, 64, kernel_size=(1, 1), stride=(1, 1))
  )
```

```
lr, num_epochs = 0.01, 10
train(model_GoogleNet.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```



now trained on the CIFAR10 dataset. Although the training loss on the graph is better, the training and validation accuracy are still insufficient and do not demonstrate generalization.

2.2 With Batch Normalization

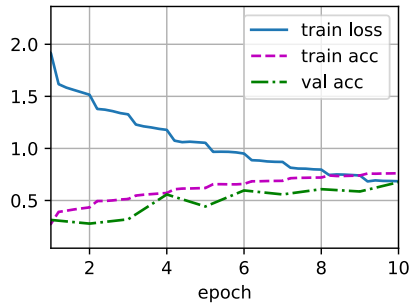
```
class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        self.bn1_1 = nn.BatchNorm2d(c1)
        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.bn2_1 = nn.BatchNorm2d(c2[0])
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        self.bn2_2 = nn.BatchNorm2d(c2[1])
        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.bn3_1 = nn.BatchNorm2d(c3[0])
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        self.bn3_2 = nn.BatchNorm2d(c3[1])
        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)
        self.bn4_2 = nn.BatchNorm2d(c4)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)

class GoogleNet_BatchNorm(d2l.Classifier):
  def b1(self):
    return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3), nn.BatchNorm2d(64),
        nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

@d2l.add_to_class(GoogleNet_BatchNorm)
def b2(self):
    return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=1), nn.BatchNorm2d(64), nn.ReLU(),
        nn.LazyConv2d(192, kernel_size=3, padding=1), nn.BatchNorm2d(192), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

@d2l.add_to_class(GoogleNet_BatchNorm)
```

```
def b3(self):
    return nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                         Inception(128, (128, 192), (32, 96), 64),
                         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

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

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

@d2l.add_to_class(GoogleNet_BatchNorm)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet_BatchNorm, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                             self.b5(), nn.LazyLinear(num_classes))
```

Saved successfully!

```
model = GoogleNet_BatchNorm()
model.layer_summary((64, 3, 64, 64))

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))
```

```
lr, num_epochs = 0.01, 10
train(model.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```

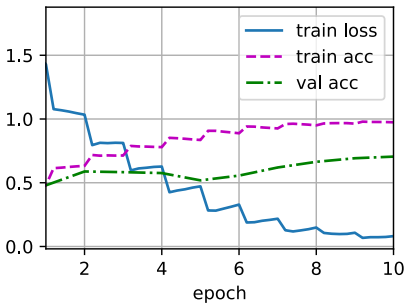


Conclusion: After the convolution layer, batch normalization is added to rescale the result and get it ready to be an input for the following layer. Adding batch normalization helps improve training loss and offers better generalization over higher epochs because GoogleNet has so many layers.

3.1 The baseline model we did in homework is called ResNet-18. Train that for CIFAR-10 and report and compare your validation accuracy against GoogleNet and VGGNet architectures you did. Can you compare the training time, model size and computation complexity across these three networks for CIFAR-10? Use 64*64 resolution across all training.

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
```

```
            else:
                self.conv3 = None
            self.bn1 = nn.LazyBatchNorm2d()
            self.bn2 = nn.LazyBatchNorm2d()

        def forward(self, X):
            Y = F.relu(self.bn1(self.conv1(X)))
            Y = self.bn2(self.conv2(Y))
            if self.conv3:
                X = self.conv3(X)
            Y += X
            return F.relu(Y)

    class ResNet(d2l.Classifier):
        def b1(self):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    @d2l.add_to_class(ResNet)
    def block(self, num_residuals, num_channels, first_block=False):
        blk = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                blk.append(Residual(num_channels))
        return nn.Sequential(*blk)

    @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)


    model = ResNet18()
    model.layer_summary((64, 3, 64, 64))

    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))

    lr, num_epochs = 0.01, 10
    train(model.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```




```
    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)

    model = ResNet18()
    model.layer_summary((64, 3, 64, 64))

    macs, params = ptflops.get_model_complexity_info(model.net, (3, 64, 64))
```

Saved successfully!

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

```
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 128, 8, 8])
Sequential output shape:         torch.Size([64, 256, 4, 4])
Sequential output shape:         torch.Size([64, 512, 2, 2])
Sequential output shape:         torch.Size([64, 10])
Sequential(
  11.18 M, 100.000% Params, 148.76 MMac, 100.000% MACs,
  (0): Sequential(
    9.6 k, 0.086% Params, 9.96 MMac, 6.696% MACs,
    (0): Conv2d(9.47 k, 0.085% Params, 9.7 MMac, 6.520% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (1): BatchNorm2d(128, 0.001% Params, 131.07 KMac, 0.088% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(0, 0.000% Params, 65.54 KMac, 0.044% MACs, )
    (3): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.044% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (b2): Sequential(
    148.22 k, 1.325% Params, 37.95 MMac, 25.507% MACs,
    (0): Residual(
      74.11 k, 0.663% Params, 18.97 MMac, 12.754% MACs,
      (conv1): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      74.11 k, 0.663% Params, 18.97 MMac, 12.754% MACs,
      (conv1): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b3): Sequential(
    525.95 k, 4.702% Params, 33.66 MMac, 22.627% MACs,
    (0): Residual(
      230.27 k, 2.059% Params, 14.74 MMac, 9.907% MACs,
      (conv1): Conv2d(73.86 k, 0.660% Params, 4.73 MMac, 3.177% MACs, 64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(8.32 k, 0.074% Params, 532.48 KMac, 0.358% MACs, 64, 128, kernel_size=(1, 1), stride=(2, 2))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      295.68 k, 2.644% Params, 18.92 MMac, 12.721% MACs,
      (conv1): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b4): Sequential(
    2.1 M, 18.780% Params, 33.61 MMac, 22.591% MACs,
    (0): Residual(
      919.3 k, 8.219% Params, 14.71 MMac, 9.887% MACs,
      (conv1): Conv2d(295.17 k, 2.639% Params, 4.72 MMac, 3.175% MACs, 128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(590.08 k, 5.276% Params, 9.44 MMac, 6.346% MACs, 256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(33.02 k, 0.295% Params, 528.38 KMac, 0.355% MACs, 128, 256, kernel_size=(1, 1), stride=(2, 2))
```

Saved successfully!

Conclusion: ResNet-18 was not used to train the CIFAR10 dataset. The training loss of the ResNet-18 model was excellent. The training loss on the ResNet-18 mode was the lowest of all the prior models. Even while the accuracy and training loss weren't terrible, they weren't great either. Though it was improving, generalization still wasn't good enough. While VGG models showed the best generalization, this had the downside of having insufficient training loss. In comparison to the other two models, GoogleNet seems to have a good balance between training loss and training and validation accuracy.

3.2 Build two new versions of ResNet (ResNet-26, and ResNet-32). Train them on CIFAR-10. Plot the training loss, training accuracy and validation accuracy. Compare the classification accuracy, computation complexity, and model size across these three versions of ResNet (18, 26, 32). How does the complexity grow as you increase the network depth?

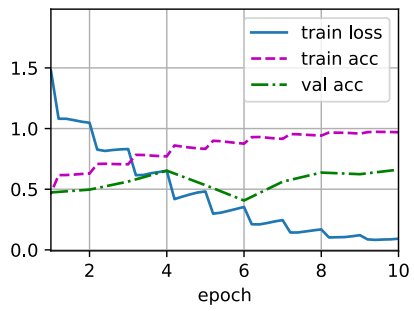
```
class ResNet26(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (4, 256), (2, 512)),
                         lr, num_classes)
```

```
model = ResNet26()
model.layer_summary((64, 3, 64, 64))

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))
```

```
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 128, 8, 8])
Sequential output shape:         torch.Size([64, 256, 4, 4])
Sequential output shape:         torch.Size([64, 512, 2, 2])
Sequential output shape:         torch.Size([64, 10])
Sequential(
  13.55 M, 100.000% Params, 186.56 MMac, 100.000% MACs,
  (0): Sequential(
    9.6 k, 0.071% Params, 9.96 MMac, 5.340% MACs,
    (0): Conv2d(9.47 k, 0.070% Params, 9.7 MMac, 5.199% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (1): BatchNorm2d(128, 0.001% Params, 131.07 KMac, 0.070% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(0, 0.000% Params, 65.54 KMac, 0.035% MACs, )
    (3): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.035% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (b2): Sequential(
    148.22 k, 1.094% Params, 37.95 MMac, 20.339% MACs,
    (0): Residual(
      74.11 k, 0.547% Params, 18.97 MMac, 10.170% MACs,
      (conv1): Conv2d(36.93 k, 0.273% Params, 9.45 MMac, 5.067% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.273% Params, 9.45 MMac, 5.067% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      .001% Params, 32.77 KMac, 0.018% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      .001% Params, 32.77 KMac, 0.018% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (1): Residual(
      74.11 k, 0.547% Params, 18.97 MMac, 10.170% MACs,
      (conv1): Conv2d(36.93 k, 0.273% Params, 9.45 MMac, 5.067% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.273% Params, 9.45 MMac, 5.067% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.018% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.018% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b3): Sequential(
    525.95 k, 3.882% Params, 33.66 MMac, 18.043% MACs,
    (0): Residual(
      230.27 k, 1.700% Params, 14.74 MMac, 7.899% MACs,
      (conv1): Conv2d(73.86 k, 0.545% Params, 4.73 MMac, 2.534% MACs, 64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.089% Params, 9.45 MMac, 5.063% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(8.32 k, 0.061% Params, 532.48 KMac, 0.285% MACs, 64, 128, kernel_size=(1, 1), stride=(2, 2))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.009% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.009% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      295.68 k, 2.183% Params, 18.92 MMac, 10.143% MACs,
      (conv1): Conv2d(147.58 k, 1.089% Params, 9.45 MMac, 5.063% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.089% Params, 9.45 MMac, 5.063% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.009% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.009% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b4): Sequential(
    4.46 M, 32.943% Params, 71.41 MMac, 38.275% MACs,
    (0): Residual(
      919.3 k, 6.786% Params, 14.71 MMac, 7.884% MACs,
      (conv1): Conv2d(295.17 k, 2.179% Params, 4.72 MMac, 2.531% MACs, 128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(590.08 k, 4.356% Params, 9.44 MMac, 5.061% MACs, 256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(33.02 k, 0.244% Params, 528.38 KMac, 0.283% MACs, 128, 256, kernel_size=(1, 1), stride=(2, 2))
```

Saved successfully!

```
lr, num_epochs = 0.01, 10
train(model.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```



```
class ResNet32(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((3, 64), (4, 128), (6, 256), (3, 512)),
                         lr, num_classes)

model = ResNet32()
model.layer_summary((64, 3, 64, 64))

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))
```

```
     Sequential output shape:         torch.Size([64, 64, 16, 16])
     Sequential output shape:         torch.Size([64, 64, 16, 16])
     Sequential output shape:         torch.Size([64, 128, 8, 8])
     Sequential output shape:         torch.Size([64, 256, 4, 4])
     Sequential output shape:         torch.Size([64, 512, 2, 2])
     Sequential output shape:         torch.Size([64, 10])
     Sequential(
       21.3 M, 100.000% Params, 300.07 MMac, 100.000% MACs,
       (0): Sequential(
         9.6 k, 0.045% Params, 9.96 MMac, 3.320% MACs,
         (0): Conv2d(9.47 k, 0.044% Params, 9.7 MMac, 3.232% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
         (1): BatchNorm2d(128, 0.001% Params, 131.07 KMac, 0.044% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
         (2): ReLU(0, 0.000% Params, 65.54 KMac, 0.022% MACs, )
         (3): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.022% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
         (4): Sequential(
           56.96 k, 0.267% Params, 18.92 MMac, 18.969% MACs,
           (0): Residual(
             74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
             (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
             (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           )
           (1): Residual(
             74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
             (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
             (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           )
           (2): Residual(
             74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
             (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
             (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           )
         )
         (b3): Sequential(
           1.12 M, 5.246% Params, 71.51 MMac, 23.831% MACs,
           (0): Residual(
             230.27 k, 1.081% Params, 14.74 MMac, 4.911% MACs,
             (conv1): Conv2d(73.86 k, 0.347% Params, 4.73 MMac, 1.575% MACs, 64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
             (conv2): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (conv3): Conv2d(8.32 k, 0.039% Params, 532.48 KMac, 0.177% MACs, 64, 128, kernel_size=(1, 1), stride=(2, 2))
             (bn1): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
             (bn2): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           )
           (1): Residual(
             295.68 k, 1.388% Params, 18.92 MMac, 6.306% MACs,
             (conv1): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (conv2): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (bn1): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
             (bn2): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           )
           (2): Residual(
```

Saved successfully!

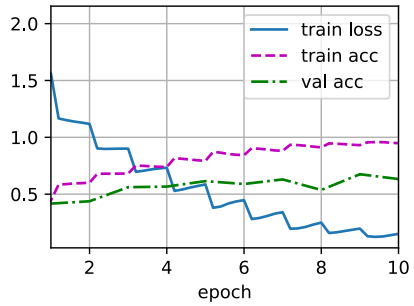
```
class ResNet32(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((3, 64), (4, 128), (6, 256), (3, 512)),
                         lr, num_classes)

model = ResNet32()
model.layer_summary((64, 3, 64, 64))

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 developr
  warnings.warn('Lazy modules are a new feature under heavy development '
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 128, 8, 8])
Sequential output shape:         torch.Size([64, 256, 4, 4])
Sequential output shape:         torch.Size([64, 512, 2, 2])
Sequential output shape:         torch.Size([64, 10])
Sequential(
  21.3 M, 100.000% Params, 300.07 MMac, 100.000% MACs,
  (0): Sequential(
    9.6 k, 0.045% Params, 9.96 MMac, 3.320% MACs,
    (0): Conv2d(9.47 k, 0.044% Params, 9.7 MMac, 3.232% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (1): BatchNorm2d(128, 0.001% Params, 131.07 KMac, 0.044% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(0, 0.000% Params, 65.54 KMac, 0.022% MACs, )
    (3): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.022% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (b2): Sequential(
    222.34 k, 1.044% Params, 56.92 MMac, 18.969% MACs,
    (0): Residual(
      74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
      (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
    )
    (1): Residual(
      74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
      (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
    )
    (2): Residual(
      74.11 k, 0.348% Params, 18.97 MMac, 6.323% MACs,
      (conv1): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.173% Params, 9.45 MMac, 3.150% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.011% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
    )
  )
  (b3): Sequential(
    1.12 M, 5.246% Params, 71.51 MMac, 23.831% MACs,
    (0): Residual(
      230.27 k, 1.081% Params, 14.74 MMac, 4.911% MACs,
      (conv1): Conv2d(73.86 k, 0.347% Params, 4.73 MMac, 1.575% MACs, 64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(8.32 k, 0.039% Params, 532.48 KMac, 0.177% MACs, 64, 128, kernel_size=(1, 1), stride=(2, 2))
      (bn1): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      295.68 k, 1.388% Params, 18.92 MMac, 6.306% MACs,
      (conv1): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 0.693% Params, 9.45 MMac, 3.148% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.001% Params, 16.38 KMac, 0.005% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
```

Saved successfully!

```
lr, num_epochs = 0.01, 10
train(model.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```



Conclusion: As the number of layers is increased from ResNet-18 to ResNet-32, the network depth also increases. In actuality, this advances the model. Out of all the models, ResNet-32 has the lowest training loss and the highest training and validation accuracy. Also, it is noted that generalization is taking place. With the CIFAR10 dataset, ResNet-32 appears to be the strongest model so far.

Extra Question: Build a compariable densenet and resnet, with same complexity and same computational. Macs and Params are the same. Show which one has higher accuracy on CIfar-10

```
def conv_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1))

class DenseBlock(nn.Module):
    def _init_(self, num_convs, num_channels):
        super(DenseBlock, self)._init_()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate input and output of each block along the channels
            X = torch.cat((X, Y), dim=1)
        return X

def transition_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))

class DenseNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

@d2l.add_to_class(DenseNet)
def _init_(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
           lr=0.1, num_classes=10):
    super(DenseNet, self)._init_()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, num_convs in enumerate(arch):
        self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                          growth_rate))
        # The number of output channels in the previous dense block
        num_channels += num_convs * growth_rate
        # A transition layer that halves the number of channels is added
        # between the dense blocks
        if i != len(arch) - 1:
            num_channels //= 2
            self.net.add_module(f'tran_blk{i+1}', transition_block(
                num_channels))
    self.net.add_module('last', nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)
```

```
class DenseNet(DenseNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)

model = DenseNet18()
model.layer_summary((64, 3, 64, 64))
```

```
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))
```

```
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 64, 16, 16])
Sequential output shape:         torch.Size([64, 128, 8, 8])
Sequential output shape:         torch.Size([64, 256, 4, 4])
Sequential output shape:         torch.Size([64, 512, 2, 2])
Sequential output shape:         torch.Size([64, 10])
Sequential(
  11.18 M, 100.000% Params, 148.76 MMac, 100.000% MACs,
  (0): Sequential(
    9.6 k, 0.086% Params, 9.96 MMac, 6.696% MACs,
    (0): Conv2d(9.47 k, 0.085% Params, 9.7 MMac, 6.520% MACs, 3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (1): BatchNorm2d(128, 0.001% Params, 131.07 KMac, 0.088% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(0, 0.000% Params, 65.54 KMac, 0.044% MACs, )
    (3): MaxPool2d(0, 0.000% Params, 65.54 KMac, 0.044% MACs, kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (b2): Sequential(
    148.22 k, 1.325% Params, 37.95 MMac, 25.507% MACs,
    (0): Residual(
      74.11 k, 0.663% Params, 18.97 MMac, 12.754% MACs,
      (conv1): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      74.11 k, 0.663% Params, 18.97 MMac, 12.754% MACs,
      (conv1): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(36.93 k, 0.330% Params, 9.45 MMac, 6.355% MACs, 64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(128, 0.001% Params, 32.77 KMac, 0.022% MACs, 64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b3): Sequential(
    525.95 k, 4.702% Params, 33.66 MMac, 22.627% MACs,
    (0): Residual(
      230.27 k, 2.059% Params, 14.74 MMac, 9.907% MACs,
      (conv1): Conv2d(73.86 k, 0.660% Params, 4.73 MMac, 3.177% MACs, 64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(8.32 k, 0.074% Params, 532.48 KMac, 0.358% MACs, 64, 128, kernel_size=(1, 1), stride=(2, 2))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): Residual(
      295.68 k, 2.644% Params, 18.92 MMac, 12.721% MACs,
      (conv1): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv2): Conv2d(147.58 k, 1.320% Params, 9.45 MMac, 6.349% MACs, 128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (bn2): BatchNorm2d(256, 0.002% Params, 16.38 KMac, 0.011% MACs, 128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (b4): Sequential(
    2.1 M, 18.780% Params, 33.61 MMac, 22.591% MACs,
    (0): Residual(
      919.3 k, 8.219% Params, 14.71 MMac, 9.887% MACs,
      (conv1): Conv2d(295.17 k, 2.639% Params, 4.72 MMac, 3.175% MACs, 128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (conv2): Conv2d(590.08 k, 5.276% Params, 9.44 MMac, 6.346% MACs, 256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (conv3): Conv2d(33.02 k, 0.295% Params, 528.38 KMac, 0.355% MACs, 128, 256, kernel_size=(1, 1), stride=(2, 2))
```

Saved successfully!

```
lr, num_epochs = 0.01, 10
train(model.net, data.get_dataloader(True), data.get_dataloader(False), num_epochs, lr, d2l.try_gpu())
```