In [1]:
```
# !pip install d2l
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
```

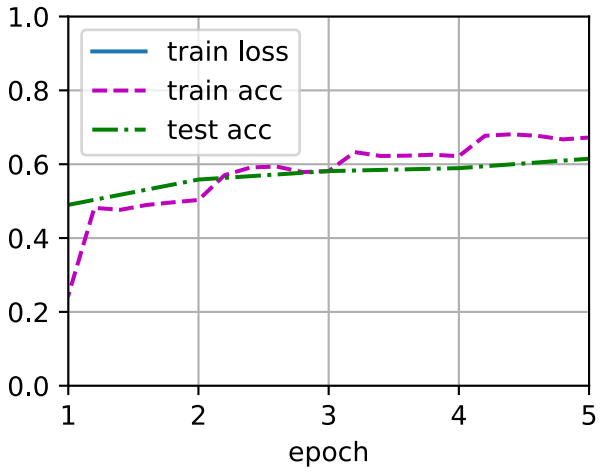
In [2]:
```
# define model
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize([256, 256]),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])

train_imgs = torchvision.datasets.DTD(root='', download=True, split='train', transf
test_imgs = torchvision.datasets.DTD(root='', download=True, split='test', transfor

train_loader = DataLoader(train_imgs, batch_size=64, shuffle=True)
test_loader = DataLoader(test_imgs, batch_size=64, shuffle=False)

NUM_CLASSES = len(train_imgs.classes)
```

In [5]:
```
import torchvision.models as models
def train_fine_tuning(net, learning_rate, num_epochs=5,
                      fine_tuning_type=None, train_iter=None, test_iter=None):
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")

    if fine_tuning_type == "full_ft":
        # Unfreeze all model parameters for full fine-tuning
        backbone_params = [p for name, p in net.named_parameters() if 'classifier'
        trainer = torch.optim.SGD([{'params': backbone_params},
                                   {'params': net.classifier[-1].parameters(),
                                    'lr': learning_rate * 10}],
                                  lr=learning_rate, weight_decay=0.0001)

    elif fine_tuning_type == "lora":
        paras = []
        for name, param in net.named_parameters():
            # Fine-tune only LoRA layers (typically identified by 'lora' or specifi
            if "lora" in name or "adapter" in name:  # Update based on your LoRA im
                param.requires_grad = True
```

```
                paras.append({'params': param, 'lr': learning_rate})
            else:
                param.requires_grad = False
        trainer = torch.optim.SGD(paras, lr=learning_rate,
                                  weight_decay=0.0001)

    elif fine_tuning_type == "lp":
        paras = []
        for name, param in net.named_parameters():
            # Fine-tune only the classifier / last layer during linear probing
            if "classifier.6" in name:  # Only update last FC layer
                paras.append({"params": param, "lr": learning_rate * 10})
            else:
                param.requires_grad = False
        trainer = torch.optim.SGD(paras, lr=learning_rate,
                                  weight_decay=0.0001)

    else:
        raise ValueError("Unknown fine tuning type")

    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

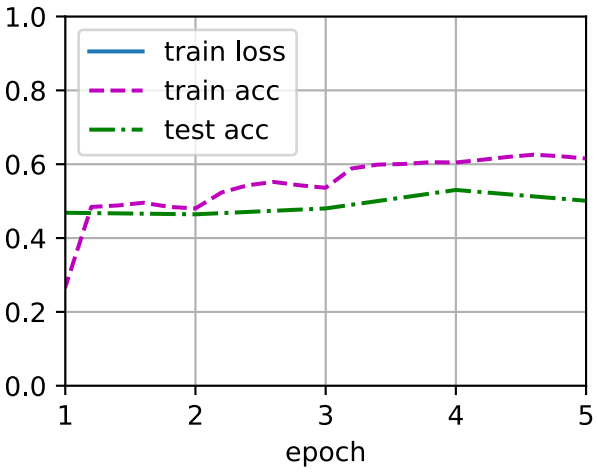
In [6]:
```
net = torchvision.models.vgg11(pretrained=True)
net.classifier[-1] = nn.Linear(net.classifier[-1].in_features, NUM_CLASSES)
nn.init.xavier_uniform_(net.classifier[-1].weight);
train_fine_tuning(net, 1e-4, fine_tuning_type='full_ft', train_iter=train_loader, t
```

```
loss 1.090, train acc 0.672, test acc 0.615
99.9 examples/sec on [device(type='cuda', index=0)]
```



In [7]:
```
net = torchvision.models.vgg11(pretrained=True)
net.classifier[-1] = nn.Linear(net.classifier[-1].in_features, NUM_CLASSES)
nn.init.xavier_uniform_(net.classifier[-1].weight);
train_fine_tuning(net, 1e-3, fine_tuning_type='lp', train_iter=train_loader, test_i
```

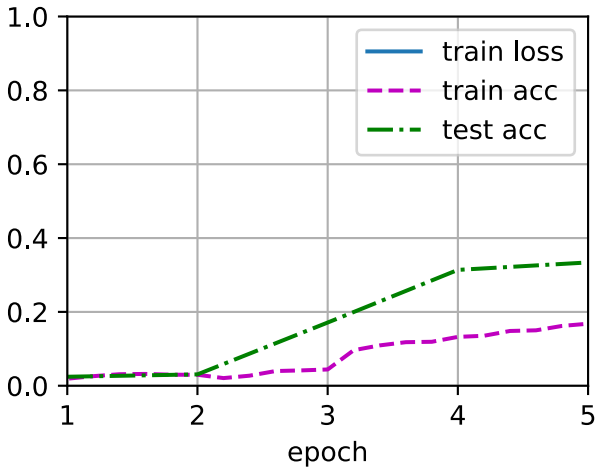
```
loss 3.673, train acc 0.615, test acc 0.501
397.3 examples/sec on [device(type='cuda', index=0)]
```



In [8]:
```python
import torch
import torch.nn as nn
import math

class LoRALayer(nn.Module):
    def __init__(self, original_layer, r=4):
        super(LoRALayer, self).__init__()
        self.original_layer = original_layer
        self.r = r
        in_features = original_layer.in_features  # Correct input dim
        out_features = original_layer.out_features  # Correct output dim

        # LoRA A: Projects from input_dim -> r
        self.lora_A = nn.Linear(in_features, r, bias=False)

        # LoRA B: Projects from r -> output_dim
        self.lora_B = nn.Linear(r, out_features, bias=False)


        nn.init.kaiming_uniform_(self.lora_A.weight, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B.weight)

    def forward(self, x):
        # Apply LoRA projection and add to original output
        return self.original_layer(x) + self.lora_B(self.lora_A(x))
```

In [9]:
```python
net = torchvision.models.vgg11(pretrained=True)
net.classifier[-1] = nn.Linear(net.classifier[-1].in_features, NUM_CLASSES)
nn.init.xavier_uniform_(net.classifier[-1].weight);

for name, module in net.named_modules():
    if name in ['classifier.0', 'classifier.3']:
        parent = net
        *parent_names, target_name = name.split('.')
        for pname in parent_names:
            parent = getattr(parent, pname)
        original_layer = getattr(parent, target_name)
        lora_layer = LoRALayer(original_layer, r=4)
        setattr(parent, target_name, lora_layer)
train_fine_tuning(net, 1e-3, fine_tuning_type='lora', train_iter=train_loader, test
```

```
loss 3.336, train acc 0.168, test acc 0.334
384.7 examples/sec on [device(type='cuda', index=0)]
```



In [12]:
```
for name, module in net.named_modules():
  print(name)
  print(module)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): LoRALayer(
      (original_layer): Linear(in_features=25088, out_features=4096, bias=True)
      (lora_A): Linear(in_features=25088, out_features=4, bias=False)
      (lora_B): Linear(in_features=4, out_features=4096, bias=False)
    )
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): LoRALayer(
      (original_layer): Linear(in_features=4096, out_features=4096, bias=True)
      (lora_A): Linear(in_features=4096, out_features=4, bias=False)
      (lora_B): Linear(in_features=4, out_features=4096, bias=False)
    )
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=47, bias=True)
  )
)
features
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU(inplace=True)
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): ReLU(inplace=True)
  (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU(inplace=True)
  (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
  (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (12): ReLU(inplace=True)
  (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): ReLU(inplace=True)
  (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (19): ReLU(inplace=True)
  (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
features.0
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.1
ReLU(inplace=True)
features.2
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
features.3
Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.4
ReLU(inplace=True)
features.5
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
features.6
Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.7
ReLU(inplace=True)
features.8
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.9
ReLU(inplace=True)
features.10
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
features.11
Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.12
ReLU(inplace=True)
features.13
Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.14
ReLU(inplace=True)
features.15
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
features.16
Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.17
ReLU(inplace=True)
features.18
Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
features.19
ReLU(inplace=True)
features.20
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
avgpool
AdaptiveAvgPool2d(output_size=(7, 7))
classifier
```

```
Sequential(
  (0): LoRALayer(
    (original_layer): Linear(in_features=25088, out_features=4096, bias=True)
    (lora_A): Linear(in_features=25088, out_features=4, bias=False)
    (lora_B): Linear(in_features=4, out_features=4096, bias=False)
  )
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): LoRALayer(
    (original_layer): Linear(in_features=4096, out_features=4096, bias=True)
    (lora_A): Linear(in_features=4096, out_features=4, bias=False)
    (lora_B): Linear(in_features=4, out_features=4096, bias=False)
  )
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=47, bias=True)
)
classifier.0
LoRALayer(
  (original_layer): Linear(in_features=25088, out_features=4096, bias=True)
  (lora_A): Linear(in_features=25088, out_features=4, bias=False)
  (lora_B): Linear(in_features=4, out_features=4096, bias=False)
)
classifier.0.original_layer
Linear(in_features=25088, out_features=4096, bias=True)
classifier.0.lora_A
Linear(in_features=25088, out_features=4, bias=False)
classifier.0.lora_B
Linear(in_features=4, out_features=4096, bias=False)
classifier.1
ReLU(inplace=True)
classifier.2
Dropout(p=0.5, inplace=False)
classifier.3
LoRALayer(
  (original_layer): Linear(in_features=4096, out_features=4096, bias=True)
  (lora_A): Linear(in_features=4096, out_features=4, bias=False)
  (lora_B): Linear(in_features=4, out_features=4096, bias=False)
)
classifier.3.original_layer
Linear(in_features=4096, out_features=4096, bias=True)
classifier.3.lora_A
Linear(in_features=4096, out_features=4, bias=False)
classifier.3.lora_B
Linear(in_features=4, out_features=4096, bias=False)
classifier.4
ReLU(inplace=True)
classifier.5
Dropout(p=0.5, inplace=False)
classifier.6
Linear(in_features=4096, out_features=47, bias=True)
```

In [ ]: