In [35]:
```
# !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
```

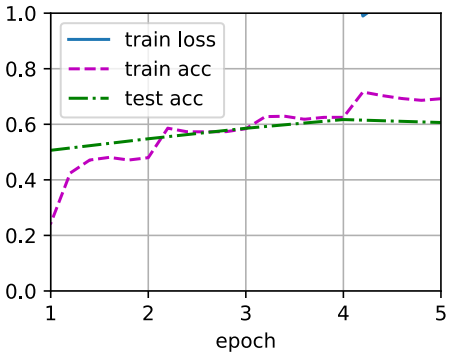
In [36]:
```
# define model
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

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', transform=train_augs)
test_imgs = torchvision.datasets.DTD(root='', download=True, split='test', transform=test_augs)

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 [37]:
```
def train_fine_tuning(net, learning_rate, num_epochs=5,
                      fine_tuning_type=None, train_iter=None, test_iter=None):
    # In case device setting went wrong.
    if torch.cuda.is_available():
        devices = d2l.try_all_gpus()
    else:
        print("No GPU detected, using CPU instead.")
        devices = ['cpu']

    loss = nn.CrossEntropyLoss(reduction="none")

    if fine_tuning_type == "full_ft":
        # Full-time tuning renew all the parameters or weights.
        backbone_params = [param for name, param in net.named_parameters() if 'classifier.6' not in name]
        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():
            # TODO: complete it
            if "lora" in name:
                param.requires_grad = True
                paras.append({'params': param})
            else:
                param.requires_grad = False
        trainer = torch.optim.SGD(paras, lr=learning_rate,
                                  weight_decay=0.0001)
    elif fine_tuning_type == "lp":
        # Linear Probing only update the last layer weight of the model, and keep other weights unchanged.
        param = [param for name, param in net.named_parameters() if 'classifier.6' in name]
        trainer = torch.optim.SGD(param, 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)
```

Problem 1. Full-Time Tuning with VGG11

In [39]:
```
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, test_iter=test_loader)
```

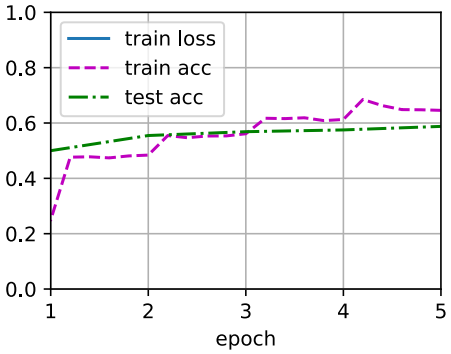
```
loss 1.043, train acc 0.692, test acc 0.606
172.2 examples/sec on [device(type='cuda', index=0)]
```



Problem 2. Linear Probing with VGG11

In [40]:
```
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_iter=test_loader)
```

```
loss 1.229, train acc 0.646, test acc 0.588
176.2 examples/sec on [device(type='cuda', index=0)]
```



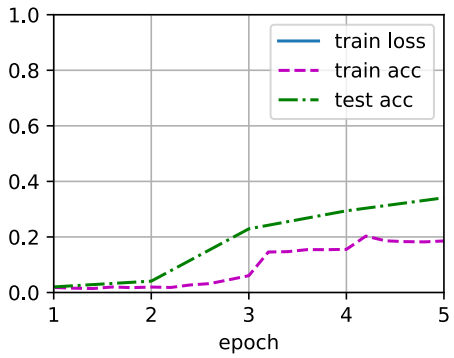
Problem 3. Applying LoRA on VGG11

In [58]:
```
import torch
import torch.nn as nn
import math
class LoRALayer(nn.Module):
    def __init__(self, original_layer, r=8):
        super(LoRALayer, self).__init__()
        self.original_layer = original_layer
        self.r = r
        self.in_features = original_layer.in_features
        self.out_features = original_layer.out_features

        self.lora_A = nn.Parameter(torch.zeros(self.r, self.in_features))
        self.lora_B = nn.Parameter(torch.zeros(self.out_features, self.r))
        # Initialize lora_A and lora_B
        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B)

    def forward(self, x):
        delta = x @ self.lora_A.t()
        delta = delta @ self.lora_B.t()

        return self.original_layer(x) + delta
```

In [59]:
```
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_iter=test_loader)
```

```
loss 3.276, train acc 0.186, test acc 0.341
492.5 examples/sec on [device(type='cuda', index=0)]
```



In [ ]: