The following additional libraries are needed to run this notebook. Note that running on Colab is experimental, please report a Github issue if you have any problem.

In [ ]:
```
!pip install d2l==v1.0.0-alpha1.post0
!pip install pytorch-ignite
!pip install torchviz
```

In [2]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
from torchvision import transforms
import torchvision
import time
import numpy as np
```

In [3]:
```
device= torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [4]:
```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset.

    Defined in :numref:`sec_fashion_mnist`"""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),transforms.ToTensor()
                                    ])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans , download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
    def text_labels(self, indices):
        """Return text labels.

        Defined in :numref:`sec_fashion_mnist`"""
        labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                  'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
        return [labels[int(i)] for i in indices]


    def get_dataloader(self, train):
        """Defined in :numref:`sec_fashion_mnist`"""
        data = self.train if train else self.val
        return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                           num_workers=self.num_workers)


    def visualize(self, batch, nrows=1, ncols=8, labels=[]):
        """Defined in :numref:`sec_fashion_mnist`"""
        X, y = batch
        if not labels:
            labels = self.text_labels(y)
        d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
```

In [5]:
```
class Trainer(d2l.HyperParameters):
  def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        """Defined in :numref:`sec_use_gpu`"""
        self.save_hyperparameters()
        self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]
        self.avg_accuracy = []
        self.avgtimer=None
        self.epochtime = None


  def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

  def fit(self, model, data):
        timer=d2l.Timer()
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.sched = torch.optim.lr_scheduler.OneCycleLR(self.optim ,
                                                         max_lr=0.4,
                                                         epochs=self.max_epochs,
                                                         steps_per_epoch=len(self.train_dataloader),
                                                         pct_start= 0.25,
                                                         anneal_strategy = "linear")
        self.epoch = 0
        self.train_batch_idx = 0
```

```
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            timer.start()
            self.fit_epoch()
            timer.stop()
            self.model.epoch_accuracy()
        self.epochtime=timer

    def fit_epoch(self):
        raise NotImplementedError

    def prepare_batch(self, batch):
        """Defined in :numref:`sec_linear_scratch`"""
        return batch

    def fit_epoch(self):
        """Defined in :numref:`sec_linear_scratch`"""
        self.model.train()
        for batch in self.train_dataloader:
            loss = self.model.training_step(self.prepare_batch(batch))
            self.optim.zero_grad()
            with torch.no_grad():
                loss.backward()
                if self.gradient_clip_val > 0:  # To be discussed later
                    self.clip_gradients(self.gradient_clip_val, self.model)
                self.optim.step()
                self.sched.step()
            self.train_batch_idx += 1
        if self.val_dataloader is None:
            return
        self.model.eval()
        for batch in self.val_dataloader:
            with torch.no_grad():
                accuracy=self.model.validation_step(self.prepare_batch(batch))

            self.val_batch_idx += 1




    def prepare_batch(self, batch):
        """Defined in :numref:`sec_use_gpu`"""
        if self.gpus:
            batch = [d2l.to(a, self.gpus[0]) for a in batch]
        return batch



    def prepare_model(self, model):
        """Defined in :numref:`sec_use_gpu`"""
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        if self.gpus:
            model.to(self.gpus[0])
        self.model = model


    def clip_gradients(self, grad_clip_val, model):
        """Defined in :numref:`sec_rnn-scratch`"""
        params = [p for p in model.parameters() if p.requires_grad]
        norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
        if norm > grad_clip_val:
            for param in params:
                param.grad[:] *= grad_clip_val / norm
```

Global maxpooling before the output layer

In [13]:

```
class varied_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)

        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3,
                                   stride=1)
        self.bn1 = nn.LazyBatchNorm2d()
        self.maxpool = nn.MaxPool2d(kernel_size=2)


    def forward(self, X):
        #Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn1(self.conv1(X)))
        Y=self.maxpool(Y)
        return Y
```

In [14]:

```
class ResNet(d2l.Classifier):

    def __init__(self, arch, lr=0.1, num_classes=10):
      super().__init__()
      self.save_hyperparameters()
      self.net = nn.Sequential(self.b1())
      self.averageaccuracy =[]
      self.epoch_accuracy_vals = []


#Adding blocks
      for i, b in enumerate(arch):
          self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))


##Final block

      self.net.add_module('last', nn.Sequential(
          nn.MaxPool2d(4), nn.Flatten(),
          nn.LazyLinear(num_classes)))
      self.net.apply(d2l.init_cnn)


    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=3, stride=1, padding=1),
            nn.LazyBatchNorm2d(), nn.ReLU())


    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(varied_Residual(num_channels, use_1x1conv=True, strides=1))
          else:
              blk.append(varied_Residual(num_channels))
      return nn.Sequential(*blk)

  ##############################################################################################################
    def epoch_accuracy(self):
        epoch_acc= torch.mean(torch.stack(self.averageaccuracy))
        self.averageaccuracy=[]
        self.epoch_accuracy_vals.append(epoch_acc)



    def validation_step(self, batch):
              Y_hat = self(*batch[:-1])
              self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
              self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
              accuracy = self.accuracy(Y_hat, batch[-1])
              self.averageaccuracy.append(accuracy)


    def accuracy(self, Y_hat, Y, averaged=True):
        """Compute the number of correct predictions.

        Defined in :numref:`sec_classification`"""
        Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
        preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
        compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
        return d2l.reduce_mean(compare) if averaged else compare


    def loss(self, Y_hat, Y, averaged=True):
        """Defined in :numref:`sec_softmax_concise`"""
        Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
        Y = d2l.reshape(Y, (-1,))
        return F.cross_entropy(
            Y_hat, Y, reduction='mean' if averaged else 'none')


    def layer_summary(self, X_shape):
        """Defined in :numref:`sec_lenet`"""
        X = d2l.randn(*X_shape)
        for layer in self.net:
            X = layer(X)
            print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

In [15]:

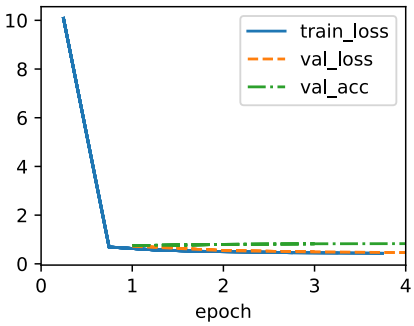
```
class ResNet9(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(( (1, 128), (1, 256), (1, 512)),
                         lr, num_classes)
ResNet18()
ResNet18().layer_summary((1,1,96,96))
```

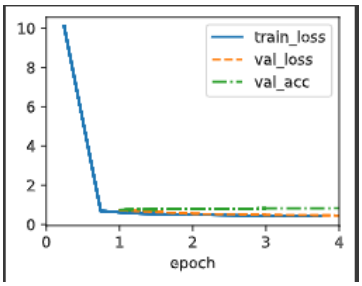
```
Sequential output shape:         torch.Size([1, 64, 96, 96])
Sequential output shape:         torch.Size([1, 128, 47, 47])
Sequential output shape:         torch.Size([1, 256, 22, 22])
Sequential output shape:         torch.Size([1, 512, 10, 10])
Sequential output shape:         torch.Size([1, 10])
```

In [16]:

```
model = ResNet9(lr=0.04)

data = FashionMNIST(batch_size=512, resize=(96, 96))



trainer = Trainer(max_epochs=4, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)

for x in range(len(model.epoch_accuracy_vals)):
  print(f'Epoch {x+1}')
  print(f'Test accuracy: {float(model.epoch_accuracy_vals[x]*100)}')
  print(f'Epoch time: {trainer.epochtime.times[x]} s')


print(f'Test accuracy: {float(model.epoch_accuracy_vals[len(model.epoch_accuracy_vals)-1]*100)}')
print(f'Average time: {trainer.epochtime.avg()} s')
print(f'Total time: {trainer.epochtime.sum()} s')
```

```
Epoch 1
Test accuracy: 74.5289535522461
Epoch time: 23.869215965270996 s
Epoch 2
Test accuracy: 79.73287963867188
Epoch time: 23.9197735786438 s
Epoch 3
Test accuracy: 82.34202575683594
Epoch time: 24.023970365524292 s
Epoch 4
Test accuracy: 82.8670654296875
Epoch time: 24.04045271873474 s
Test accuracy: 82.8670654296875
Average time: 23.963353157043457 s
Total time: 95.85341262817383 s
```





Epoch 1 Test accuracy: 74.5289535522461 Epoch time: 23.869215965270996 s Epoch 2 Test accuracy: 79.73287963867188 Epoch time: 23.9197735786438 s Epoch 3 Test accuracy: 82.34202575683594 Epoch time: 24.023970365524292 s Epoch 4 Test accuracy: 82.8670654296875 Epoch time: 24.04045271873474 s Test accuracy: 82.8670654296875 Average time: 23.963353157043457 s Total time: 95.85341262817383 s

#Conversion of ReLU activation functions to CELU and reordering of maxpooling and batch norms

In [17]:

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

```
                                   stride=1)

        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        self.maxpool = nn.MaxPool2d(kernel_size=2)

    def forward(self, X):
        X = nn.CELU(self.bn1(self.maxpool(self.conv1(X))),alpha=0.075)

        Y = nn.CELU(self.bn2(self.conv2(X)),alpha=0.075)
        Y = nn.CELU(self.bn3(self.conv3(Y)),alpha=0.075)
        Y += X
        return F.celu(Y)
```

In [18]:

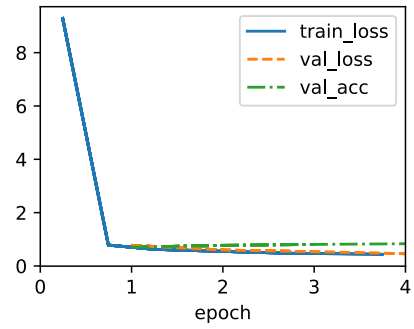
```
class ResNet(d2l.Classifier):

    def __init__(self, arch, lr=0.1, num_classes=10):
      super().__init__()
      self.save_hyperparameters()
      self.net = nn.Sequential(self.b1())
      self.averageaccuracy =[]
      self.epoch_accuracy_vals = []


#Adding blocks
      for i, b in enumerate(arch):
          if i == 1:
            self.net.add_module(f'b{i+2}',nn.Sequential(
                nn.LazyConv2d(256, kernel_size=3, padding=1,stride=1),
                nn.MaxPool2d(kernel_size=2),
                nn.LazyBatchNorm2d(),
                nn.CELU(alpha=0.075)
                ))


          self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))


##Final block
      self.net.add_module('last', nn.Sequential(
          nn.MaxPool2d(4), nn.Flatten(),nn.LazyLinear(num_classes)))
      self.net.apply(d2l.init_cnn)


    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=3, stride=1, padding=1),
            nn.LazyBatchNorm2d(), nn.ReLU())


    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(varied_Residual(num_channels, use_1x1conv=True, strides=1))
          else:
              blk.append(varied_Residual(num_channels))
      return nn.Sequential(*blk)

  ##########################################################################################################################
    def epoch_accuracy(self):
        epoch_acc= torch.mean(torch.stack(self.averageaccuracy))
        self.averageaccuracy=[]
        self.epoch_accuracy_vals.append(epoch_acc)



    def validation_step(self, batch):
              Y_hat = self(*batch[:-1])
              self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
              self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
              accuracy = self.accuracy(Y_hat, batch[-1])
              self.averageaccuracy.append(accuracy)


    def accuracy(self, Y_hat, Y, averaged=True):
        """Compute the number of correct predictions.

        Defined in :numref:`sec_classification`"""
        Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
        preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
        compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
        return d2l.reduce_mean(compare) if averaged else compare


    def loss(self, Y_hat, Y, averaged=True):
        """Defined in :numref:`sec_softmax_concise`"""
        Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
        Y = d2l.reshape(Y, (-1,))
```

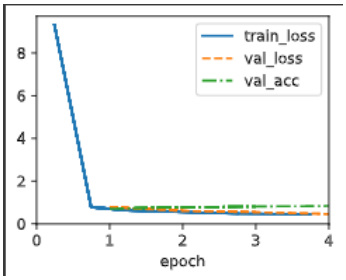
```
        return F.cross_entropy(
            Y_hat, Y, reduction='mean' if averaged else 'none')


    def layer_summary(self, X_shape):
        """Defined in :numref:`sec_lenet`"""
        X = d2l.randn(*X_shape)
        for layer in self.net:
            X = layer(X)
            print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

In [19]:

```
class ResNet9(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(( (1, 128), (1, 256), (1, 512)),
                         lr, num_classes)
model = ResNet9(lr=0.04)

data = FashionMNIST(batch_size=512, resize=(96, 96))



trainer = Trainer(max_epochs=4, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)

for x in range(len(model.epoch_accuracy_vals)):
  print(f'Epoch {x+1}')
  print(f'Test accuracy: {float(model.epoch_accuracy_vals[x]*100)}')
  print(f'Epoch time: {trainer.epochtime.times[x]} s')


print(f'Test accuracy: {float(model.epoch_accuracy_vals[len(model.epoch_accuracy_vals)-1]*100)}')
print(f'Average time: {trainer.epochtime.avg()} s')
print(f'Total time: {trainer.epochtime.sum()} s')
```

```
Epoch 1
Test accuracy: 70.29756927490234
Epoch time: 23.873865604400635 s
Epoch 2
Test accuracy: 77.23345947265625
Epoch time: 23.99230980873108 s
Epoch 3
Test accuracy: 80.84846496582031
Epoch time: 23.99388575553894 s
Epoch 4
Test accuracy: 83.31629180908203
Epoch time: 24.057832956314087 s
Test accuracy: 83.31629180908203
Average time: 23.979473531246185 s
Total time: 95.91789412498474 s
```





Epoch 1 Test accuracy: 70.29756927490234 Epoch time: 23.873865604400635 s Epoch 2 Test accuracy: 77.23345947265625 Epoch time: 23.99230980873108 s Epoch 3 Test accuracy: 80.84846496582031 Epoch time: 23.99388575553894 s Epoch 4 Test accuracy: 83.31629180908203 Epoch time: 24.057832956314087 s Test accuracy: 83.31629180908203 Average time: 23.979473531246185 s Total time: 95.91789412498474 s

In [ ]:

In [ ]: