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 [ ]:
```
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 [ ]:
```
#
device= torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
```

Trainer adjusted

In [ ]:
```
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.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
        self.avgtimer= timer.avg()

  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.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
```

In [ ]:

```
```

Residual Block

In [ ]:

```
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)
```

ResNet archetecture

In [ ]:

```
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.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
          nn.LazyLinear(num_classes)))
      self.net.apply(d2l.init_cnn)


    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))


    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)

    ###################################################################################################
    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 [ ]:

```
```

RestNet18

In [ ]:

```
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)
```

In [ ]:

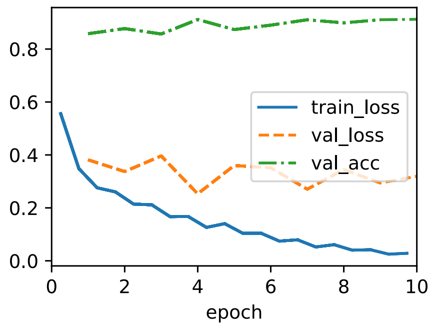
```
ResNet18().layer_summary((1, 1, 96, 96))
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/lazy.py:180: UserWarning: Lazy modules are a new feature under heavy d
evelopment so changes to the API or functionality can happen at any moment.
  warnings.warn('Lazy modules are a new feature under heavy development '
Sequential output shape:	 torch.Size([1, 64, 24, 24])
Sequential output shape:	 torch.Size([1, 64, 24, 24])
Sequential output shape:	 torch.Size([1, 128, 12, 12])
Sequential output shape:	 torch.Size([1, 256, 6, 6])
Sequential output shape:	 torch.Size([1, 512, 3, 3])
Sequential output shape:	 torch.Size([1, 10])
```

Hyperparameters	Time per epoch (s)	Total time (s)	Accuracy (%)
Lr =0.01 **Batch size**=128 **Epochs**=10	44.9	449.2	91.2

In [ ]:

```
model = ResNet18(lr=0.01)

data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))



trainer = Trainer(max_epochs=10, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)
```



In [ ]:

```
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[9]*100)}')
print(f'Average time: {trainer.epochtime.avg()} s')
print(f'Total time: {trainer.epochtime.sum()} s')
```

```
Epoch 1
Test accuracy: 85.80894470214844
Epoch time: 50.23906230926514 s
Epoch 2
Test accuracy: 87.72745513916016
Epoch time: 43.59075307846069 s
Epoch 3
Test accuracy: 85.69026947021484
Epoch time: 43.57801699638367 s
Epoch 4
Test accuracy: 91.15901947021484
Epoch time: 43.48850107192993 s
Epoch 5
Test accuracy: 87.3615493774414
Epoch time: 44.209349632263184 s
Epoch 6
Test accuracy: 89.0328369140625
Epoch time: 44.00176525115967 s
Epoch 7
Test accuracy: 91.06013488769531
Epoch time: 44.70082402229309 s
Epoch 8
Test accuracy: 89.89319610595703
Epoch time: 45.32575011253357 s
Epoch 9
Test accuracy: 91.07991027832031
Epoch time: 44.35497570037842 s
Epoch 10
Test accuracy: 91.2381362915039
Epoch time: 45.72406196594238 s
Test accuracy: 91.2381362915039
Average time: 44.921306014060974 s
Total time: 449.21306014060974 s
```

In [ ]:

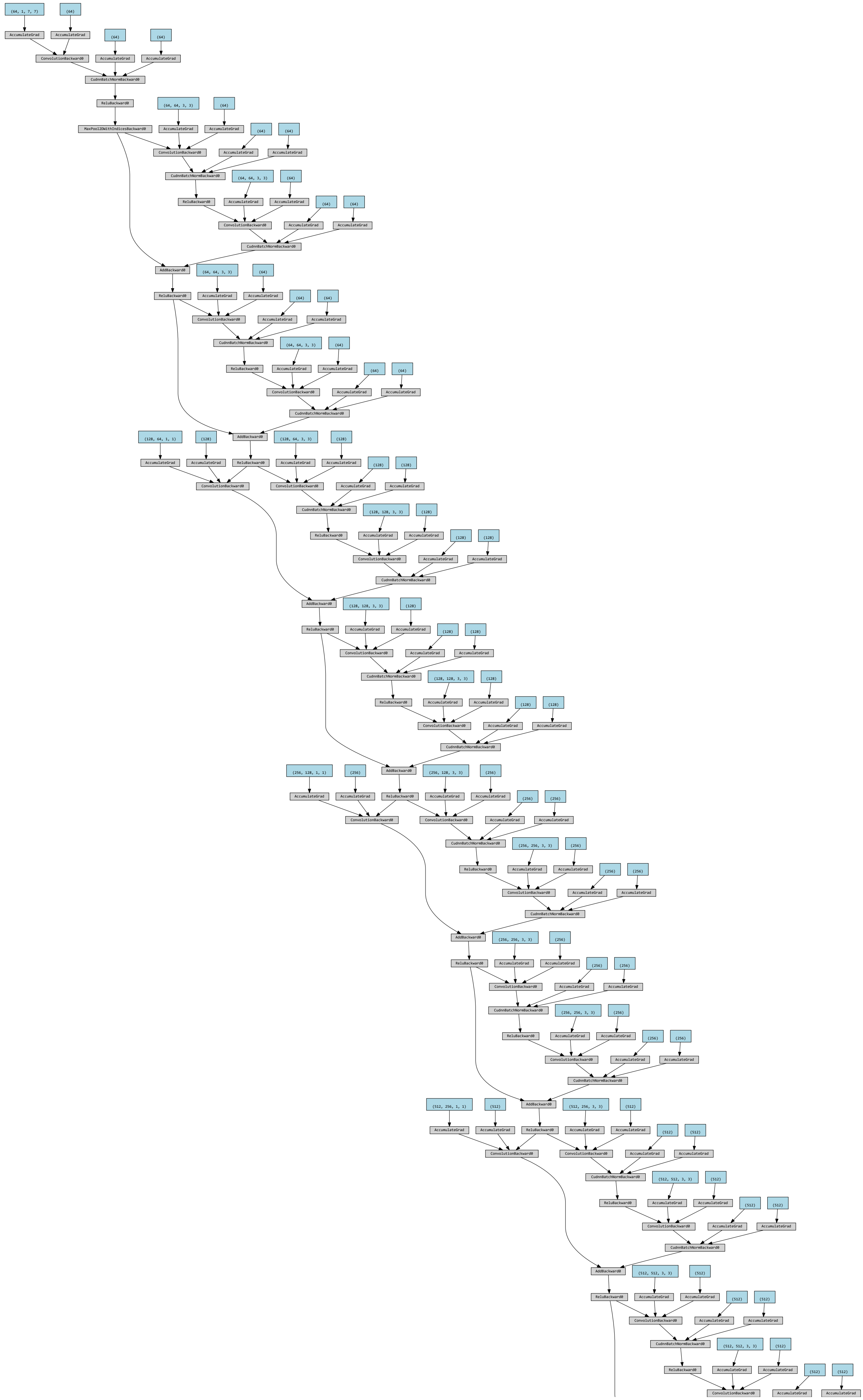
```
model.epoch_accuracy_vals[9]
```

In [ ]:

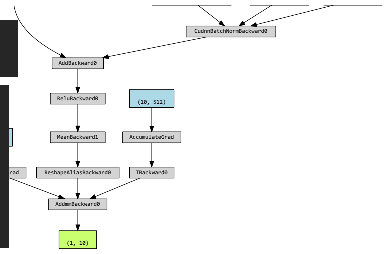
```
!pip install torchviz
from torchviz import make_dot
x = torch.zeros(1, 1, 96, 96, dtype=torch.float, requires_grad=False, device='cuda:0')
out = model(x)
make_dot(out)
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torchviz in /usr/local/lib/python3.10/dist-packages (0.0.2)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from torchviz) (2.0.1+cu118)
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (from torchviz) (0.20.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (3.12.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (1.11.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (3.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (3.1.2)
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist-packages (from torch->torchviz) (2.0.0)
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch->torchviz) (3.25.2)
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch->torchviz) (16.0.5)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->torchviz) (2.1.
2)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch->torchviz) (1.3.0)
```

Out[ ]:



Hyperparameters	Time per epoch (s)	Total time (s)	Accuracy (%)
Lr =OneCycle schedule(max = 0.1) **Batch size**=128 **Epochs**=10	45.6	456.4	93.5



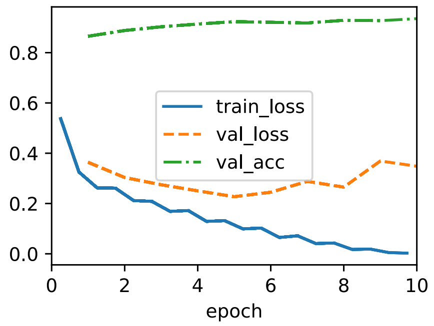
In [ ]:

```python
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=0.1,epochs=self.max_epochs,steps_per_epoch=len(sel
        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
```

In [ ]:

```
model = ResNet18(lr=0.01)

data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))



trainer = Trainer(max_epochs=10, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)
```



In [ ]:

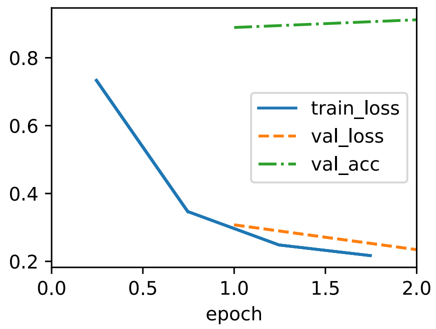
```
for x in range(len(model.epoch_accuracy_vals)):
  print(f'Epoch {x+1}')
  print(f'Test accuracy: {float(model.epoch_accuracy_vals[x]*100)}')
  print(f'Epoch time: {trainer.epochtime.times[x]} s')


print(f'Test accuracy: {float(model.epoch_accuracy_vals[9]*100)}')
print(f'Average time: {trainer.epochtime.avg()} s')
print(f'Total time: {trainer.epochtime.sum()} s')
```

```
Epoch 1
Test accuracy: 86.5308609008789
Epoch time: 46.48881125450134 s
Epoch 2
Test accuracy: 88.81526947021484
Epoch time: 44.54008674621582 s
Epoch 3
Test accuracy: 90.29866027832031
Epoch time: 44.00532841682434 s
Epoch 4
Test accuracy: 91.3765869140625
Epoch time: 44.13915419578552 s
Epoch 5
Test accuracy: 92.34573364257812
Epoch time: 45.677462100982666 s
Epoch 6
Test accuracy: 92.08860778808594
Epoch time: 46.17701554298401 s
Epoch 7
Test accuracy: 91.81171417236328
Epoch time: 47.61989903450012 s
Epoch 8
Test accuracy: 92.87974548339844
Epoch time: 44.736005544662476 s
Epoch 9
Test accuracy: 92.72151947021484
Epoch time: 46.16487669944763 s
Epoch 10
Test accuracy: 93.53244018554688
Epoch time: 46.83611083030701 s
Test accuracy: 93.53244018554688
Average time: 45.638475036621095 s
Total time: 456.38475036621094 s
```

Hyperparameters	Time per epoch (s)	Total time (s)	Accuracy (%)
Lr =OneCycle schedule(max = 0.1) **Batch size**=512 **Epochs**=2	42.7	85.5	91.2

In [ ]:

```
model = ResNet18(lr=0.01)

data = d2l.FashionMNIST(batch_size=512, resize=(96, 96))


trainer = Trainer(max_epochs=2, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)
```



In [ ]:

```
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: 88.92520904541016
Epoch time: 43.123722076416016 s
Epoch 2
Test accuracy: 91.20231628417969
Epoch time: 42.44218945503235 s
Test accuracy: 91.20231628417969
Average time: 42.78295576572418 s
Total time: 85.56591153144836 s
```

In [ ]:

```
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(sel
        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
```

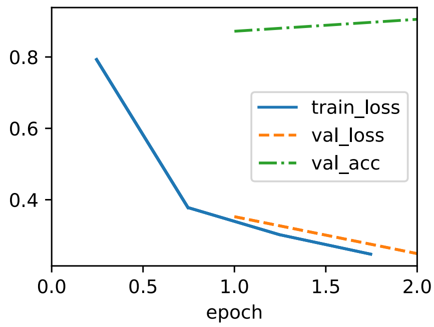
In [ ]:

```
```

Hyperparameters	Time per epoch (s)	Total time (s)	Accuracy (%)
Lr =OneCycle schedule(max = 0.4) **Batch size**=512 **Epochs**=2	42.7	85.4	90.6

In [ ]:

```
model = ResNet18(lr=0.04)

data = d2l.FashionMNIST(batch_size=512, resize=(96, 96))



trainer = Trainer(max_epochs=2, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)
```



In [ ]:

```
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: 87.2087631225586
Epoch time: 42.87779259681702 s
Epoch 2
Test accuracy: 90.55606842041016
Epoch time: 42.5186710357666 s
Test accuracy: 90.55606842041016
Average time: 42.69823181629181 s
Total time: 85.39646363258362 s
```

In [ ]:

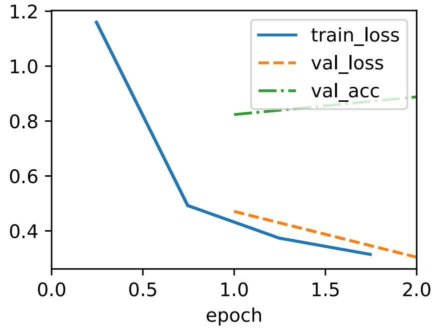
```
import albumentations as A
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset.

    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() ,transforms.RandomErasing()
                                    ])
        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)
```

Hyperparameters	Time per epoch (s)	Total time (s)	Accuracy (%)
Lr =OneCycle schedule(max = 0.4) **Batch size**=512 **Epochs**=2 With **random erasing**	45.3	90.7	88.7

In [ ]:

```
model = ResNet18(lr=0.04)

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

```
trainer = Trainer(max_epochs=2, num_gpus=1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)

trainer.fit(model,data)
```



In [ ]:

```
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: 82.35926055908203
Epoch time: 45.022716999053955 s
Epoch 2
Test accuracy: 88.79940032958984
Epoch time: 45.68494987487793 s
Test accuracy: 88.79940032958984
Average time: 45.35383343696594 s
Total time: 90.70766687393188 s
```

Adding Random erasing resulted in a slight reduction in accuracy as well as increasing test time by 2s per an epoch. Thus it wuill be removed.

BACKBONE

Now we will attempt to adjust the networks architecture to try and reduce training time. By firstly looking at the backbone to attempt to optimise the shortest path.Thus we eliminate the long branches first

In [ ]:

```
import albumentations as A
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 [ ]:

```
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)
        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 = F.relu(self.bn1(X))
        if self.conv3:
          Y = self.conv3(Y)
        Y=F.relu(self.bn2(Y))

        return Y
```

In [ ]:

```
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.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
          nn.LazyLinear(num_classes)))
      self.net.apply(d2l.init_cnn)


    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))


    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=2))
          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 [ ]:

```
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)
```

The Backbone only consists of a preparation layer 4 layers with two batch norms and two ReLU activation functions , three of the layers also have a 1x1 kernel with stride 2 and a dense block.

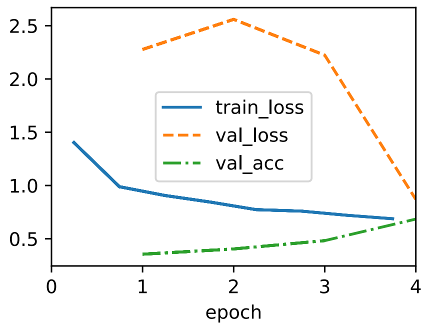
In [ ]:

```
ResNet18()
```

In [ ]:

```
model = ResNet18(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: 35.505516052246094
Epoch time: 30.642396450042725 s
Epoch 2
Test accuracy: 40.43830490112305
Epoch time: 29.988940715789795 s
Epoch 3
Test accuracy: 48.25080490112305
Epoch time: 31.428505182266235 s
Epoch 4
Test accuracy: 68.48230743408203
Epoch time: 30.13810396194458 s
Test accuracy: 68.48230743408203
Average time: 30.549486577510834 s
Total time: 122.19794631004333 s
```



Test accuracy is better than expected at just under 70 % and test time is 122.1 around half the time we were previously achieving.Furthermore following in Myrtle.ais footsteps we will remove repeated batch norm-ReU groups and assess.

In [ ]:

```
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)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()


    def forward(self, X):
        #Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn1(X))
        if self.conv3:
          Y = self.conv3(Y)


        return Y
```

In [ ]:

```
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`"""
```

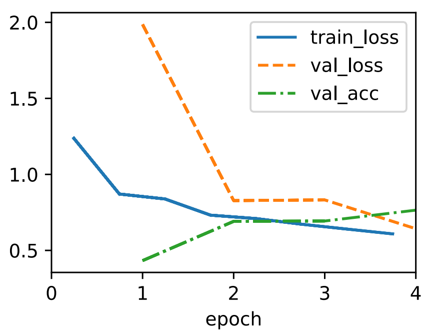
```
        if self.gpus:
            batch = [d2l.to(a, self.gpus[0]) for a in batch]
        return batch



    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
```

The repeating batch norm-ReLU groups were removed further shortening the backbone and resulting in an accuracy of this backbone achieved an accuracy of 76.4% in 4 epochs at 91.8s in total.

In [ ]:

```
model = ResNet18(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: 43.280677795410156
Epoch time: 24.515854358673096 s
Epoch 2
Test accuracy: 69.11994934082031
Epoch time: 22.652745962142944 s
Epoch 3
Test accuracy: 69.33824157714844
Epoch time: 22.00593876838684 s
Epoch 4
Test accuracy: 76.4809341430664
Epoch time: 22.632235288619995 s
Test accuracy: 76.4809341430664
Average time: 22.95169359445572 s
Total time: 91.80677437782288 s
```



continuing with archetecture experimentation below with 3x3 convolutions instead on 1x1 A serious shortcoming of this backbone network is that the downsampling convolutions have 1×1 kernels and a stride of two, so that rather than enlarging the receptive field they are simply discarding information. Furthermore, global max pooling is added to increase downsampling

In [ ]:

```
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 [ ]:

```
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 [ ]:
```
class ResNet18(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))
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/lazy.py:180: UserWarning: Lazy modules are a new feature under heavy d
evelopment so changes to the API or functionality can happen at any moment.
  warnings.warn('Lazy modules are a new feature under heavy development '
Sequential output shape:         torch.Size([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 [ ]:
```
model = ResNet18(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: 32.71656799316406
Epoch time: 30.77890968322754 s
Epoch 2
Test accuracy: 31.15694236755371
Epoch time: 29.658064126968384 s
Epoch 3
Test accuracy: 48.516197204589844
Epoch time: 30.458709001541138 s
Epoch 4
Test accuracy: 69.13890075683594
Epoch time: 30.121670484542847 s
Test accuracy: 69.13890075683594
Average time: 30.254338324069977 s
Total time: 121.01735329627991 s
```