Deeplearning_2023100124_

October 10, 2024

0.1 7.1 From Fully Connected Layers to Convolutions

When handling the high-dimensional data such as an image with MLP, there exists certain limitation of requiring large memory for params and collecting a large dataset.

Therefore we need CNN which takes data structure feature in the calculation.

Desideratas 1. Translation Invariance (Regardless of positional data) 2. Locality Principle (earliest layers should detect features of only local regions of data) 3. Deeper layers should capture longer-range features of the image.

0.2 7.2 Convolutions for Images

[97]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

[98]:
```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

[99]:
```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

[99]:
```
tensor([[19., 25.],
        [37., 43.]])
```

[100]:
```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))
    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

[101]:
```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

[102]:
```
K = torch.tensor([[1.0, -1.0]])
```

[103]:
```
Y = corr2d(X, K)
Y
```

[103]:
```
tensor([[-1., -1.],
        [-1., -1.],
        [-1., -1.]])
```

[104]:
```
corr2d(X.t(), K)
```

[104]:
```
tensor([[-3., -3.],
        [-3., -3.],
        [-3., -3.]])
```

7.2.3 finding edges in a image

X is dataset, K is filter(kernel) by applying K to X, the vertical edges will be detected.

[105]:
```
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

[105]:
```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

[106]:
```
K = torch.tensor([[1.0, -1.0]])
```

[107]:
```
Y = corr2d(X, K)
Y
```

```
[107]: tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

```
[108]: corr2d(X.t(), K)
```

```
[108]: tensor([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

Question.

1. How to find efficient filter(kernel) for detecting?
2. Is there no filter for detecting both vertical and horizontal edges at the same time?
3. By applying filters, the absolute of edge pixels will have high values, but the problem is they might have different sign. In applying filter, is one more step to make values as absolute needed? Or just remaining the values to have different signs.

```
[109]: # Construct a two-dimensional convolutional layer with 1 output channel and a
       # kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
       conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

       # The two-dimensional convolutional layer uses four-dimensional input and
       # output in the format of (example, channel, height, width), where the batch
       # size (number of examples in the batch) and the number of channels are both 1
       X = X.reshape((1, 1, 6, 8))
       Y = Y.reshape((1, 1, 6, 7))
       lr = 3e-2  # Learning rate

       for i in range(10):
           Y_hat = conv2d(X)
           l = (Y_hat - Y) ** 2
           conv2d.zero_grad()
           l.sum().backward()
           # Update the kernel
           conv2d.weight.data[:] -= lr * conv2d.weight.grad
           if (i + 1) % 2 == 0:
               print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 1.471
epoch 4, loss 0.348
epoch 6, loss 0.100
epoch 8, loss 0.034
epoch 10, loss 0.013
```

[110]:
```
conv2d.weight.data.reshape((1,2))
```

[110]:
```
tensor([[ 0.9835, -1.0058]])
```

Strict meaning of Convolution in mathmetic is to flip and move parallely, so the meaning of cross-correlation and convolution is different, strictly speaking, since cross-correlation does not calculate symmetricly.

However we will use them to refer each other. Futhermore, the term element refers to tensor in the layer or kernel data.

"receptive field" refers to all the elements that may affect the calculation of x during forward propagation.

When needing larger receptive field, design the network deeper.

0.3 7.3 Padding and Stride

Use padding to retain spacial dimension of the input.

Use convolution with stride to reduce the spacial dimension of the input.

[1]:
```
import torch
from torch import nn
```

Perimeter pixels are less used.

When convolutioning with nested kernels, the spacial dimension of the output get smaller than the spacial dimension of the input.

`spacial dimension of output = spacial dimension of input - spacial dimension of kernel + padded`

Thus to make the spacial dimension of the input same as the spacial dimension of output, set padding amount to be

`spacial dimension of kernel - 1`

If spacial dimension of kernel is even, padding amount should be odd. So in this matter, we usually allocate even padding at upper side of the input and odd padding at the bottom side of input. However it scarcely matters, because it is common to set odd spacial dimension of the kernel. That is why we use 3x3 kernels so often.

Question. By applying padding, in the output matrix, the perimeter pixels have less information about input, since they are calculated due to zero padded pixels. Don't they matter? Perimeter pixels of input pixels almost preserve their own value due to padding while center pixels don't.

I think this question is related to the question, why convolution layer is so effective. Why and how they work. In deep-learning, what kind of layer is evaluated as a good one. Are their purpose is to compact the values of previous layers, or extracting features of previous layers? Or all of those are one of the expected function of layer?

```
[4]: def comp_conv2d(conv2d, X):
         X = X.reshape((1,1)+X.shape) #convolution calculation requires 4th␣
      ↪dimension tensor as an input. (1,1) is for setting batchsize, channels as 1.
         Y = conv2d(X)
         return Y.reshape(Y.shape[2:])

     conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1) #"padding" property means␣
      ↪the number of padding which applied to up, down, left and right.
     X = torch.rand(size=(8,8))
     comp_conv2d(conv2d, X).shape
```

```
[4]: torch.Size([8, 8])
```

```
[5]: conv2d = nn.LazyConv2d(1, kernel_size=(5,3), padding = (2,1)) # by setting␣
      ↪padding as tuple, we can make padding width and height different.
     comp_conv2d(conv2d, X).shape
```

```
[5]: torch.Size([8, 8])
```

stride means convolution calculation is applied to input with specific spacing. the output dimension is equal to the equation below.

`floor(input dimension-kernel dimension+padding dimension+stride dimension)/stride dimension)`

```
[6]: conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
     comp_conv2d(conv2d, X).shape
```

```
[6]: torch.Size([4, 4])
```

```
[8]: conv2d = nn.LazyConv2d(1, kernel_size=(3,5), padding=(0,1), stride=(3,4))
     comp_conv2d(conv2d, X).shape
```

```
[8]: torch.Size([2, 2])
```

Question. what is the proper selection between "convolution with stride" and "pooling"

0.4 7.4 Multiple Input and Multiple Output Channels

```
[83]: import torch
      from d2l import torch as d2l
```

If input tensor has multiple channels just as an input of image, the kernel should have a same number of channels as an input tensor. To cross-correlate input tensor and kernels, cross-correlate each matrix of input and kernel and then sum them up.

[84]:
```
def corr2d_multi_in(X, K):
    return sum(d2l.corr2d(x,k) for x, k in zip(X, K))
```

[86]:
```
X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
               [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

[86]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

In practical cases, the models are designed for increasing channel depth and decreasing space dimension. Features of the input tensor are spread along the channels. And that is the reason why we sum up the result of cross-correlation. For example, to find the edge of the input image, we need to look up all of the 3 channels(red, green, blue).

The dimension of the kernel should be (channel_num_output)*(channel_num_input)*(kernel_height)*(kernel_widt

[87]:
```
def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

[88]:
```
K = torch.stack((K, K+1, K+2), 0)
K.shape
```

[88]:
```
torch.Size([3, 2, 2, 2])
```

[89]:
```
corr2d_multi_in_out(X,K)
```

[89]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

1x1 convolution can not detect a feature because it does not correlate adjacent pixels in each channels, instead it only correlate the pixels in the same spacial position of each channels. Thus it integrates the features in different channesl.

1x1 convolution is just same as fully connected layer.

The nonlinearity followed by convolutions makes the layer of convolutions not to be folded into one convolution. Since convolution is linear calculation, it is natural.

[90]:
```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w =X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h*w))
    K = K.reshape((c_o, c_i))
    Y =torch.matmul(K,X)
    return Y.reshape((c_o, h, w))
```

[91]:
```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2,3,1,1))
Y1 = corr2d_multi_in_out_1x1(X,K)
Y2 = corr2d_multi_in_out(X,K)
assert float(torch.abs(Y1-Y2).sum())<1e-6
```

0.5 7.5 Pooling

pooling layer contributes to expanding receptive field(make spacial resolution low), which is needed for detecting high-level feature, and also to detecting lower-level features through making layer to be invariant to translation.

[12]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

there is no parameter unlike convolutinal layer which has a kernel.

[16]:
```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0]-p_h+1, X.shape[1] - p_w+1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode=='max':
                Y[i,j] = X[i: i+p_h, j: j+p_w].max()
            elif mode=='avg':
                Y[i,j] = X[i: i+p_h, j:j+p_w].mean()
    return Y
```

[14]:
```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0,7.0,8.0]])
pool2d(X, (2,2))
```

[14]:
```
tensor([[4., 5.],
        [7., 8.]])
```

[17]:
```
pool2d(X, (2,2), 'avg')
```

[17]:
```
tensor([[2., 3.],
        [5., 6.]])
```

Question. I understand the point of the reason that the max pooling helps the network to detect the same feature even though there is a translation. 1. However is there no other option for delegating the effect of translation instead of max-pooling?

2. To back-propagate, the only maximum element's gradient would be calculated. To do this, should we stash the index of the maximum element while fore-propagate?

deeplearning frameworks default to matching poolling window sizes and stride.

```
[19]: X = torch.arange(16, dtype=torch.float32).reshape((1,1,4,4))
      X
```

```
[19]: tensor([[[[ 0.,  1.,  2.,  3.],
                [ 4.,  5.,  6.,  7.],
                [ 8.,  9., 10., 11.],
                [12., 13., 14., 15.]]]])
```

```
[21]: pool2d = nn.MaxPool2d(3)
      pool2d(X)
```

```
[21]: tensor([[[[10.]]]])
```

```
[22]: pool2d = nn.MaxPool2d((2,3), stride=(2,3), padding=(0,1))
      pool2d(X)
```

```
[22]: tensor([[[[ 5.,  7.],
                [13., 15.]]]])
```

Unlike convolutinal layer, the pooling doesn't sum up the output channel instead it just produce a tensor with same channel as input tensor.

```
[23]: X = torch.cat((X, X+1), 1)
      X
```

```
[23]: tensor([[[[ 0.,  1.,  2.,  3.],
                [ 4.,  5.,  6.,  7.],
                [ 8.,  9., 10., 11.],
                [12., 13., 14., 15.]],

               [[ 1.,  2.,  3.,  4.],
                [ 5.,  6.,  7.,  8.],
                [ 9., 10., 11., 12.],
                [13., 14., 15., 16.]]]])
```

```
[25]: pool2d = nn.MaxPool2d(3, padding=1, stride=2)
      pool2d(X)
```

```
[25]: tensor([[[[ 5.,  7.],
                [13., 15.]],

               [[ 6.,  8.],
                [14., 16.]]]])
```

The most popular pooling is "2x2 max pooling" for quartering the input tensor.

0.6 7.6 Convolutional Neural Networks

[113]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

LeNet is consists of two parts. 1. convolutinal layers (5x5 kernels and sigmoid activation layer + average pooling) 2. dense layers consisting of three fully connected layers

before going 1 -> 2, the result of 1 should be flattened to 2-dimension(on 1st dimension the index of the minibatch, on 2nd dimension actual data)

the result of the 2 should be same as the output class (classification)

[114]:
```
def init_cnn(module):
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

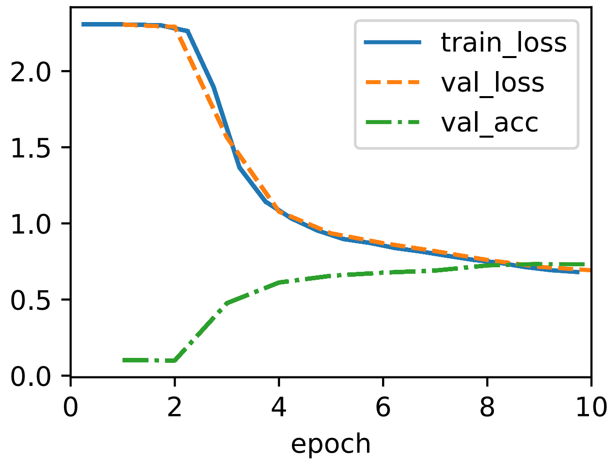
[115]:
```
@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:  torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
```

```
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
Sigmoid output shape:    torch.Size([1, 84])
Linear output shape:     torch.Size([1, 10])
```

[116]:
```python
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



0.7 8.2 Networks Using Blocks

the focus on designing network has shifted from neurons to block.

[38]:
```python
import torch
from torch import nn
from d2l import torch as d2l
```

Original network, stacking conv layer, pooling layer one by one reduces the resolution rapidly. Instead it is better design to stack multiple conv layer between pooling layer. Applying 2 Conv layer of 3 conv layer touches same pixels as applying 1 5x5 conv layer does. Thus applying conv layer twice is better than applying 5x5 conv layer once due to lower number of parameters, deeper network and non-linearity

Stacking 3x3 convolutions has become a gold standard.

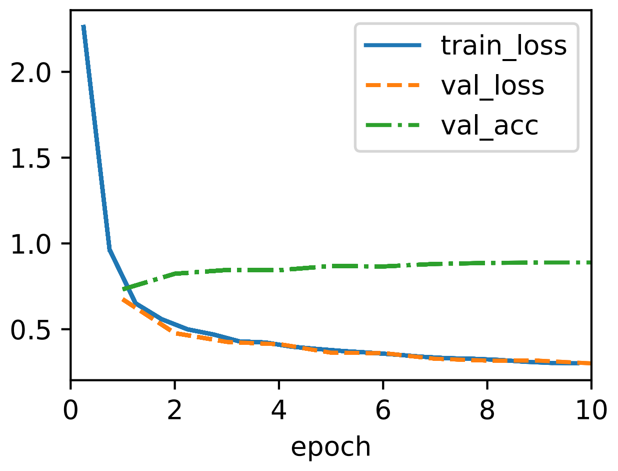
[45]:
```python
# vgg_block returns a stacked layer of conv layers number of num_convs, relu␣
↪and Max pooling.
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2)) #halving width and␣
↪height
    return nn.Sequential(*layers) # unpacking operator *
```

[46]:
```
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)
```

[47]:
```
VGG(arch=((1,64),(1,128),(2,256),(2,512),(2,512))).layer_summary((1,1,224,224))
```

```
Sequential output shape:         torch.Size([1, 64, 112, 112])
Sequential output shape:         torch.Size([1, 128, 56, 56])
Sequential output shape:         torch.Size([1, 256, 28, 28])
Sequential output shape:         torch.Size([1, 512, 14, 14])
Sequential output shape:         torch.Size([1, 512, 7, 7])
Flatten output shape:    torch.Size([1, 25088])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 10])
```

[50]:
```
model = VGG(arch=((1,16),(1,32),(2,64),(2,128),(2,128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224,224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



0.8 8.6 Residual Networks and ResNet

[117]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

When inspecting the network, maybe perhaps some layers might worsen the prediction.

To prevent this, the layers should include the result of the previous layer, thus it may only develop.

This is an idea of ResNet.

[123]:
```
class Residual(nn.Module):
    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)
```

the code above is for implementing res block. It is selectable to apply 1x1 conv to the x which is passed to activation function.

Question.

why remaining applying 1x1conv to be optional? what is the purpose of 1x1conv if selected?

-answer: used when input and output channel is not same.

[124]:
```
blk = Residual(3)
X = torch.randn(4,3,6,6)
blk(X).shape
```

[124]:
```
torch.Size([4, 3, 6, 6])
```

[125]:
```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

[125]:
```
torch.Size([4, 6, 3, 3])
```

[126]:
```
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))
```

[127]:
```
@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)
```

[133]:
```
@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)
```

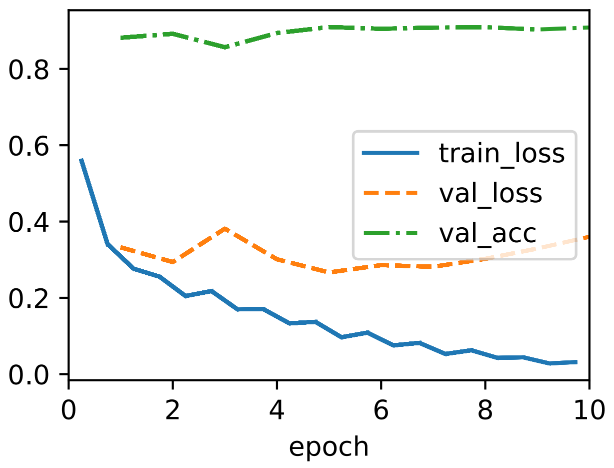
By modifying the num_channels and residual blocks, we can construct different ResNet models. The below is ResNet18.

[134]:
```
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)

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

```
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])
```

[135]:
```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus = 1)
data = d2l.FashionMNIST(batch_size=128, resize=(96,96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



It is possible to add more nonlinearity by increasing kernel width, increasing channel depth and increasing the number of layers.

However among them, increasing channel depth returns bad calculating time. if input channel depth is c0 and output channel depth is c1, it has O(c0*c1) time complexity.

To solve this, just group the input by channels. If the input tensor has 18 channels and we want to output 27 channels, group them by 3 so that one group calculates 6 input channels by 9 filters. Through this, we can decrease the calculation time by one-third. Also, requires less parameters (c0xc1 matrix -> (c0/k)x(c1/k))

The only problem remains is that by spliting the convolution, there is no information exchange between groups. To solve this, we wrap 3x3 convolution layer by two 1x1 convolution layers. By 1x1 conv, we can make groups to exchange informations and also reduce amount of calculation(bottleneck).

Question. 1. The text says that increasing kernel width add more nonlinearity, however the convolution calculation is just linear calculation. So how does increasing kernel width increase nonlinearity?

2. By 1x1 conv, we modify channel and exchange information along the channels. Then what is the proper value of 1x1 conv, in 3x3 conv the kernel has some kind of complexity and is able to detect some feature. However applying 1x1 conv is just as same as multiplying certain number to the sum of the tensor in specific (x,y).

[136]:
```
class ResNeXtBlock(nn.Module):
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,␣
↪strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3, stride=strides,␣
↪padding=1, groups=bot_channels//groups)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        if use_1x1conv:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,␣
↪stride=strides)
            self.bn4 =nn.LazyBatchNorm2d()
        else:
            self.conv4 = None

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return F.relu(Y+X)
```

[137]:
```
blk = ResNeXtBlock(32, 16, 1)
X = torch.randn(4,32,96,96)
blk(X).shape
```

```
[137]: torch.Size([4, 32, 96, 96])
```