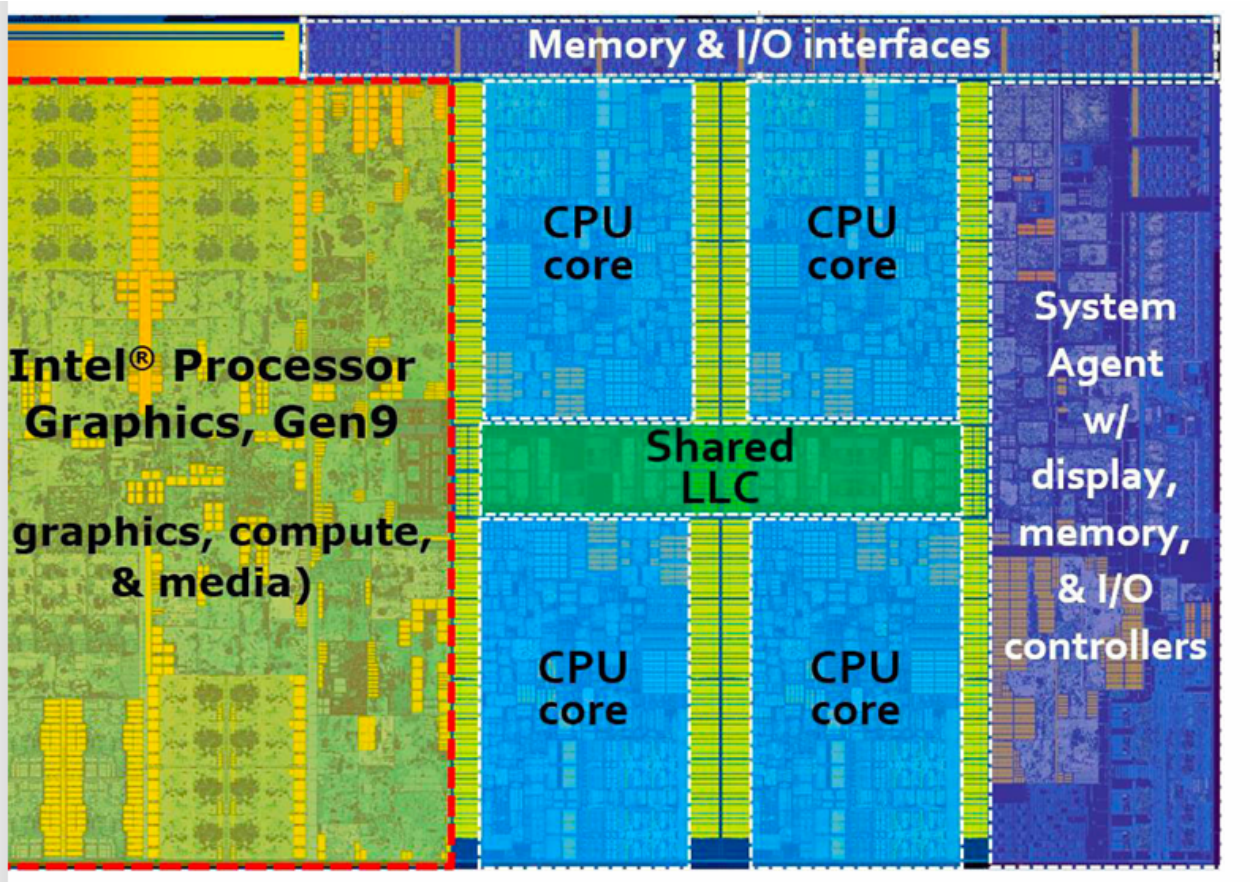
笔记：计算机视觉

第十二章 计算性能

31 深度学习硬件: CPU和GPU



1.提升CPU利用率

- 提升空间和时间的内存本地性
  - 时间：重用数据使得保持它们再缓存里
  - 空间：按序读写数据使得可以预读取

- 并行来利用所有核

# 核	6 / 64	2K / 4K
TFLOPS	0.2 / 1	10 / 100
内存大小	32 GB / 1 TB	16 GB / 32 GB
内存带宽	30 GB/s / 100 GB/s	400 GB/s / 1 TB/s
控制流	强	弱

2.提升GPU利用率

- 并行
  - 使用数千个线程
- 内存本地性
  - 缓存更小，架构更简单
- 少用控制语句

  - 支持有限

  - 同步开销大

  3.不要频繁在CPU和GPU之间传输数据：带宽限制，同步开销

  4.总结
- CPU：可以处理通用计算。性能优化考虑数据读写效率和多线程
- GPU：使用更多的小核和更好的内存带宽，适合能大规模并行的计算任务

32 深度学习硬件: TPU和其他

1.DSP：数字信号处理

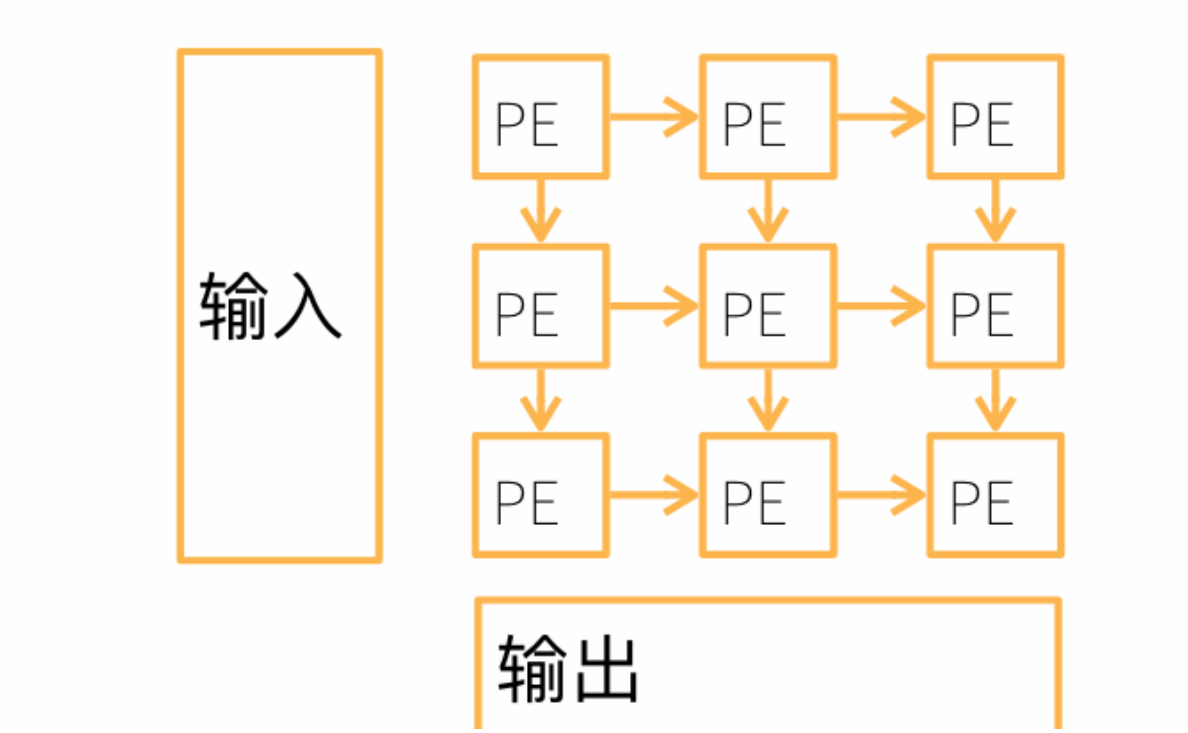
- 为数字信号处理算法设计：点积，卷积，FFT

  2.FPGA:可编程阵列
- 有大量可以编程逻辑单元和可配置的连接

  3.AI ASIC
- 大公司自己的芯片 e.g.Google TPU

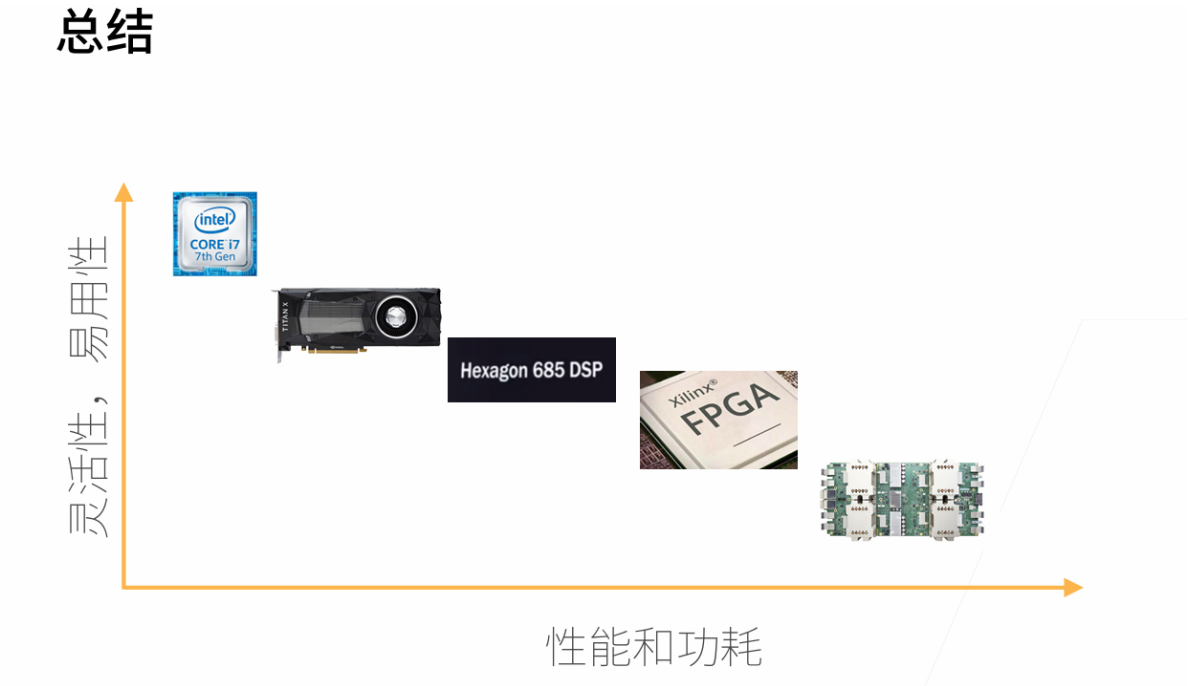
- 核心：systolic array

-



- 计算单元（PE）阵列，特别适合做矩阵乘法

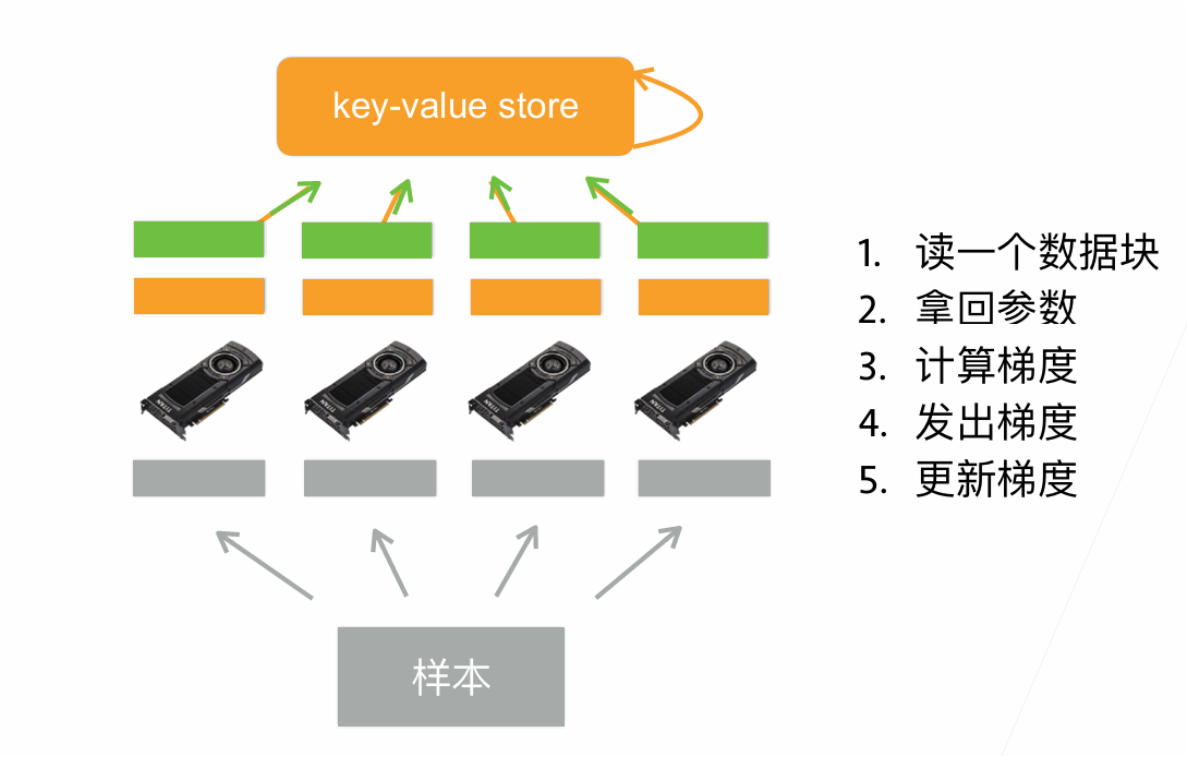
4.总结



33 单机多卡并行

1.将小批量计算分到多个GPU上

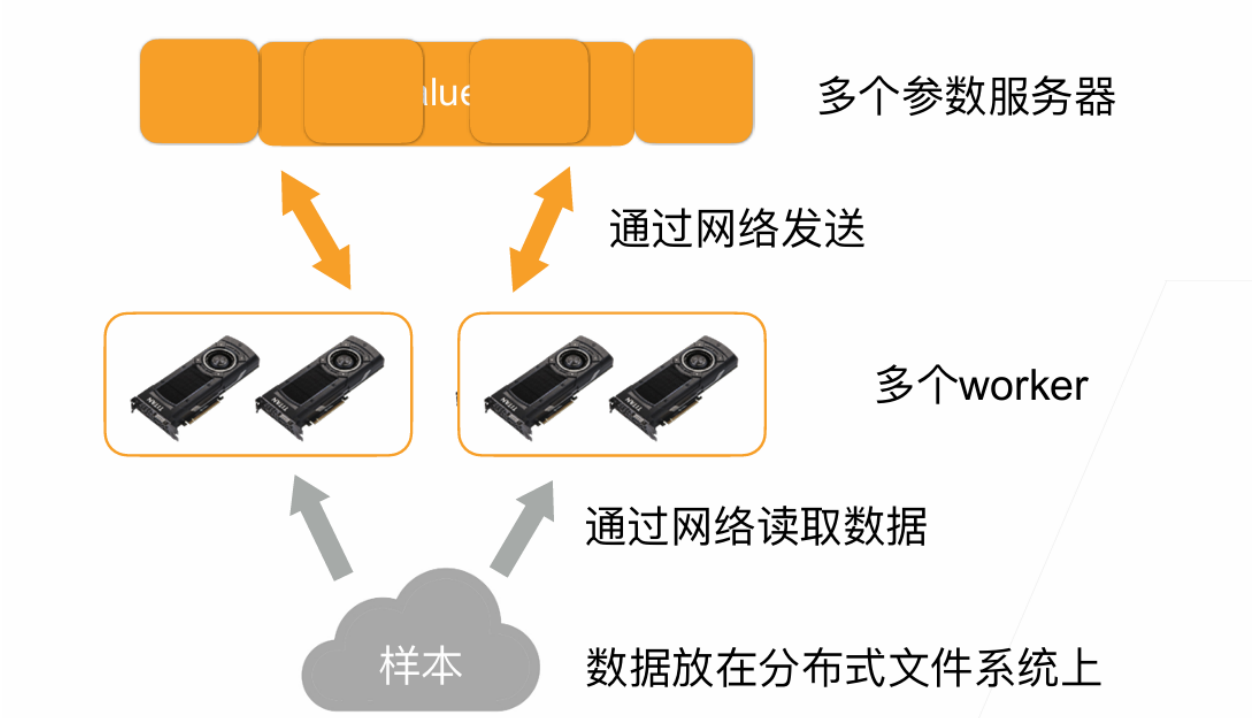
- 数据并行



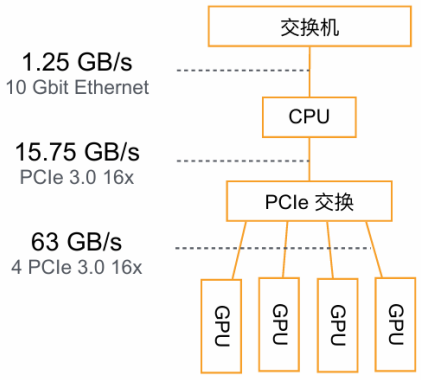
- 模型并行
- 通道并行（数据+模型）

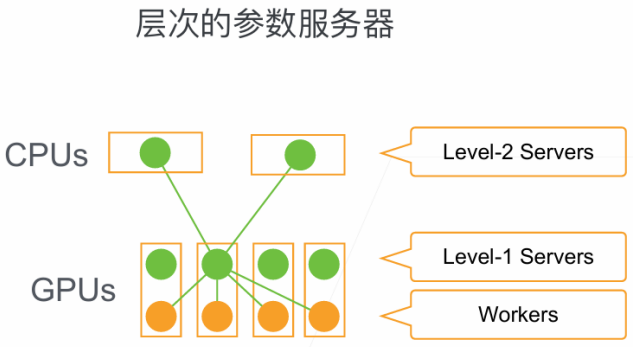
34 多GPU训练实现

35 分布式计算

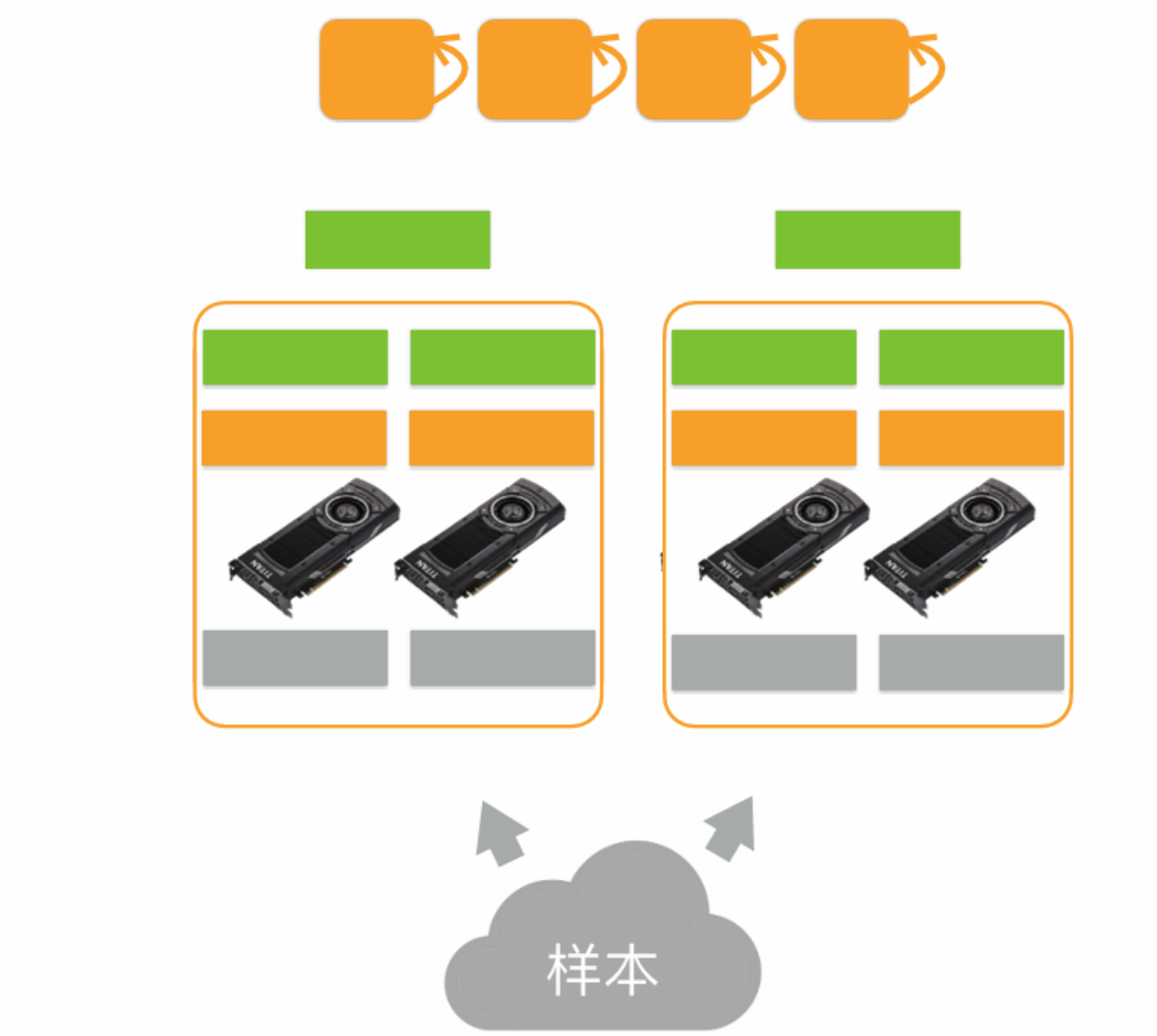

多个参数服务器
通过网络发送
多个worker
通过网络读取数据
样本
数据放在分布式文件系统上

GPU 机器架构


交换机
1.25 GB/s
10 Gbit Ethernet
CPU
15.75 GB/s
PCIe 3.0 16x
PCIe 交换
63 GB/s
4 PCIe 3.0 16x
GPU
GPU
GPU
GPU


层次的参数服务器
CPUs
GPUs
Level-2 Servers
Level-1 Servers
Workers

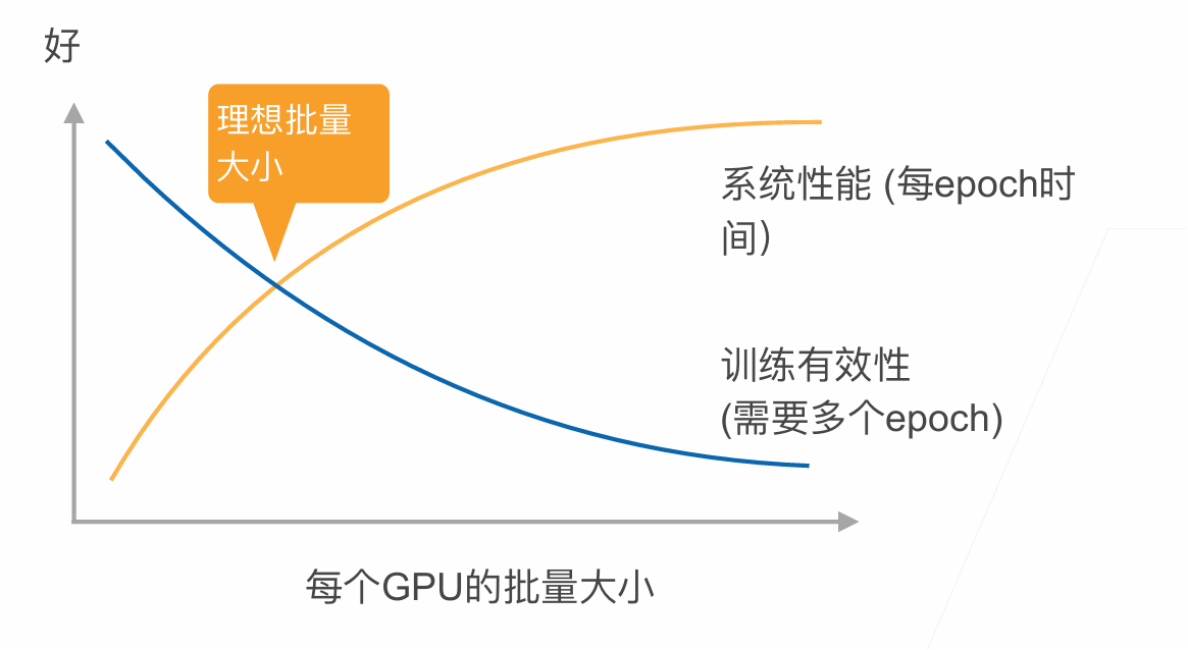
1.计算每一个小批量



- 每个计算服务器读取小批量中的一块
- 进一步将数据切分到每个GPU上
- 每个worker从参数服务器那里获取模型参数
- 复制参数到每个GPU上
- 每个GPU计算精度
- 将所有GPU上的梯度求和
- 梯度传回服务器，每个服务器对梯度求和并更新参数

  2.同步SGD：n个GPU，每个每次处理b个样本，同步SGD等价于单GPU运行批量大小为nb的SGD->也即得到相对单个的n倍加速

  3.性能的权衡



批量大小增加导致需要更多计算来得到给定的模型精度

第十三章 计算机视觉

13.1 图像增广

1.数据增强：增加一个已有数据集。使得有更多的多样性

- 在语音里加入各种不同的背景噪音
- 改变图片的颜色和形状

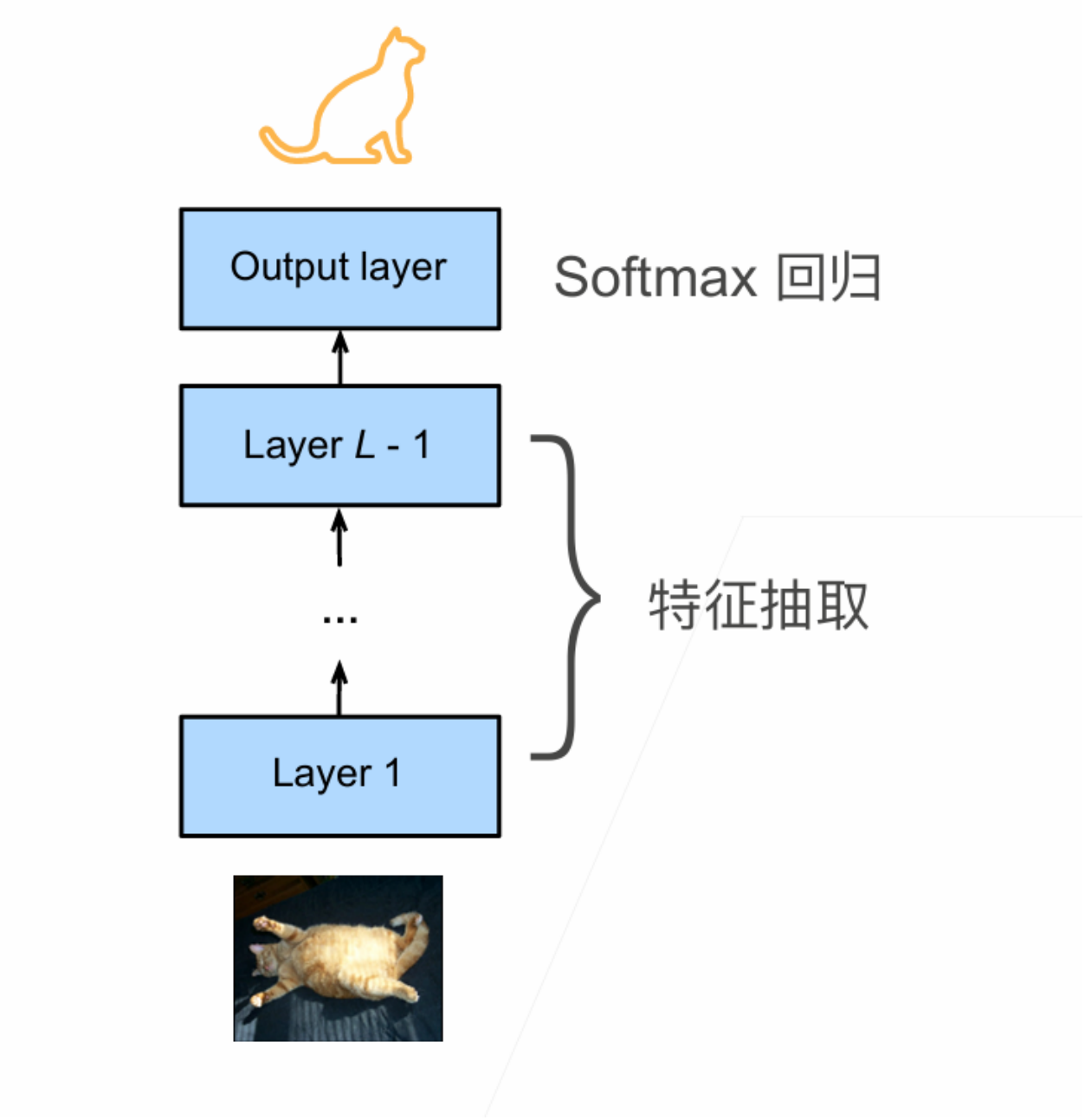
2.类型：

- 翻转：上下、左右
- 切割：然后变形到固定形状
  - 随机高宽比
  - 随机大小
  - 随机位置
- 颜色：色调、饱和度、明亮度、对比度

3.总结：数据增广通过变形数据来获取多样性从而使得模型泛化性能更好

13.2 微调

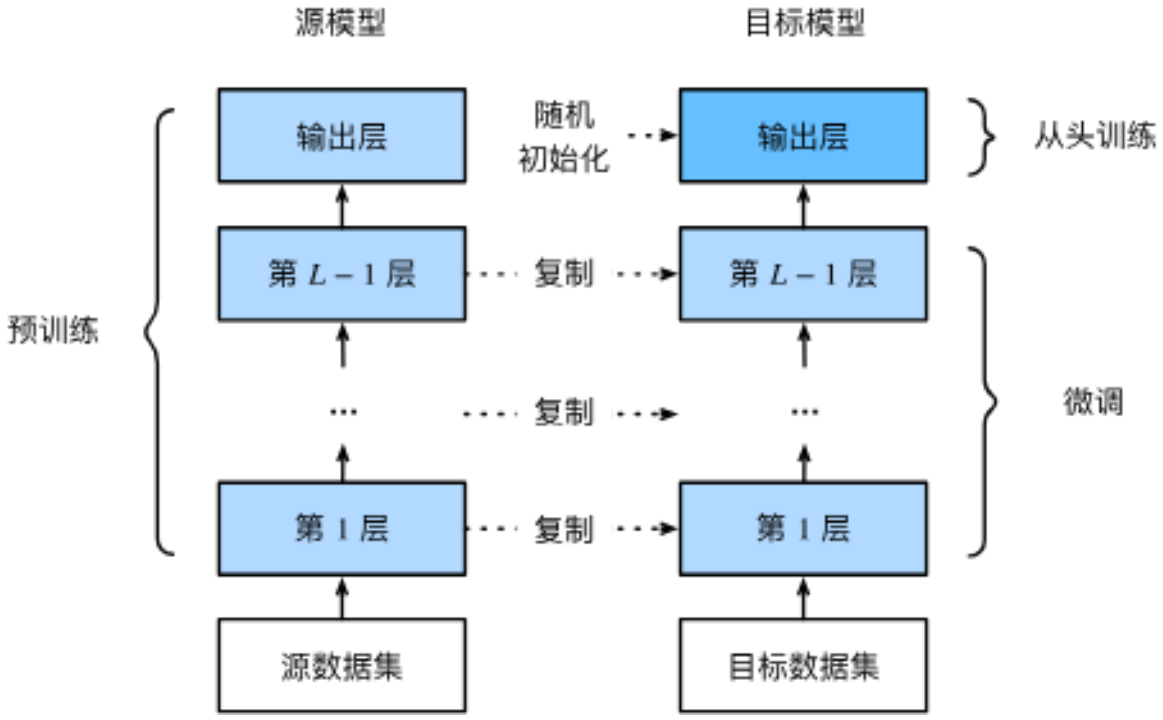
1.网络架构



一个神经网络一般可以分成两块：

- 特征抽取将原始像素变成容易线性分割的特征
- 线性分类器来做分类

2.



源模型的特征抽取部分可能仍可以对目标模型进行特征抽取，当然线性分类器用不了了了

->预训练的模型的特征抽取部分可以复制给目标模型，然后微调即可使用，输出层就只能从头训练了（随机初始化）

即：迁移学习，微调（fine-tuning)是其中一个常用技巧

3.训练：使用更小的学习率和更少的数据迭代（更强的正则化）

源数据集远复杂与目标数据集的话通常微调效果会更好，有助于泛化

4.trick

- 重用分类器权重
  - 源数据集可能也有目标数据中的部分符号->可以使用预训练好的模型分类器中对应标号对应的向量来做初始化
- 固定一些层
  - 神经网络学习有层次的特征：低层次的特征更加通用，高层次的特征更跟数据集相关->可以固定底部一些层的参数不参与更新（更强的正则）
- 对特征提取部分学习率用小一点，对线性分类层学习率用大的（乘以10）

5.总结：微调通过使用在大数据上得到的预训练好的模型来初始化权重来完成提升精度->

预训模型质量很是重要，微调通常速度更快精度更高

6.代码：调用net时加上pretrained=True

13.3 目标检测和边界框

1.边缘框(bounding box)：四个数字定义

- （左上x，左上y，右上x，右上y）
- （左上x，左上y，宽，高）

2.目标检测数据集

图片文件名，物体类别，边缘框

3.总结：物体检测识别图片里面的多个物体的类别和位置，位置通常用边缘框表示

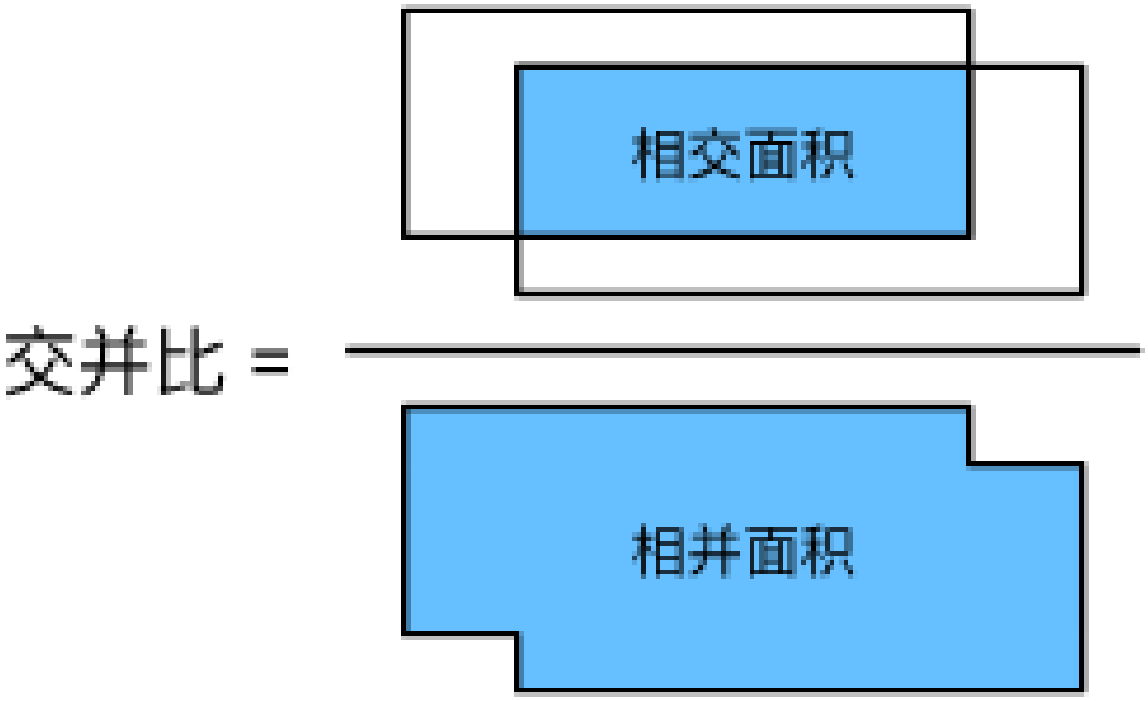
13.4 锚框

1.一类目标检测算法基于锚框（anchor box)：

- 提出多个被称为锚框的区域（边缘框）
- 预测每个锚框是否含有关注的物体
- 如果是，基于此预测从这个锚框到真实边缘框的偏移

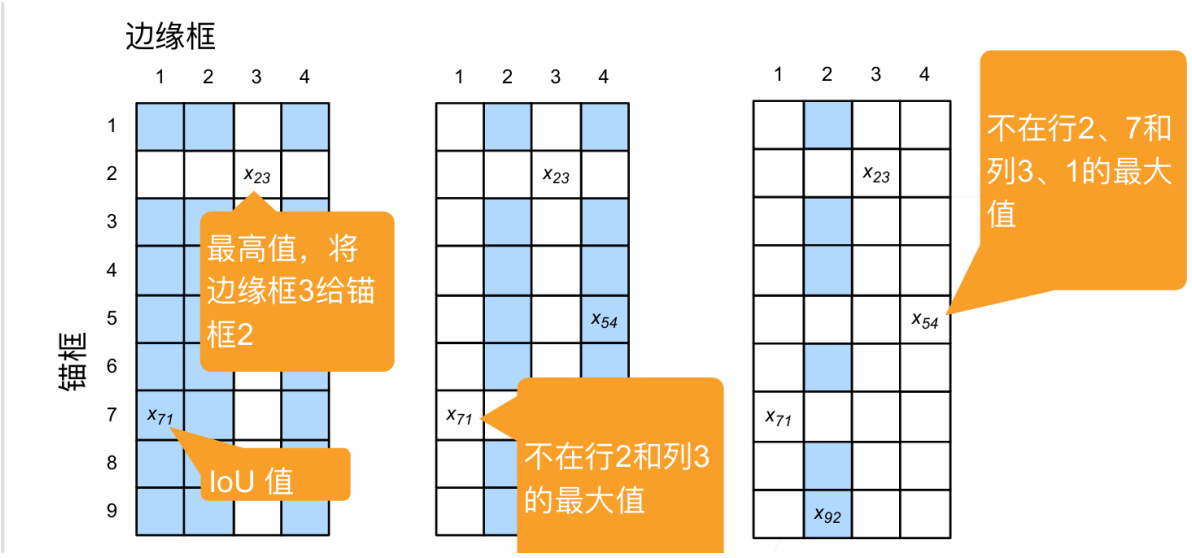
2.IoU-交并比：用来计算两个框之间的相似度

0~1.0：无重叠，1：重合



3.赋予锚框标号（每个锚框是一个训练样本）：

- 标注成背景
- 关联上一个真实边缘框

- 

4.使用非极大值抑制（NMS）输出

- 每个锚框预测一个边缘框，而NMS可以合并相似的预测
- 1.选中非背景类（不是背景是物体）的最大预测值（softmax回归，概率最大值），即计算每个类别的预测概率，最大的概率p所对应的类别就是预测的类别
- 2.去掉其他所有和这个被选中的的IoU值大于θ的预测
- 3.重复上述过程直到所有预测要么被选中要么被去掉

5.总结

- 首先生成大量锚框，并赋予编号，每个锚框作为一个样本来进行训练
- 在预测的时候，使用NMS来去掉冗余的预测

6.代码实现

以每一个像素为中心，生成不同高宽的锚框

锚框的宽度和高度分别是$w\sqrt{s}\sqrt{r}$和$h\sqrt{s}/\sqrt{r}$。我们只考虑组合：

$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1)$

其中w'、h'是锚框的宽和高，w、h是图片的宽和高，s是锚框的大小（占图片大小的百分比），r是锚框的宽高比与图片的宽高比之比

$$w \times \sqrt{\frac{size\ of\ box}{size\ of\ picture}} \times \sqrt{\frac{w'}{h'} / \frac{w}{h}} = w \times \sqrt{\frac{h' \times w'}{h \times w}} \times \sqrt{\frac{w' \times h}{h' \times w}} = w'$$

训练流程：

训练数据集是带有ground truth bounding box和class的，就像图片分类的带有class一样

- 用上述算法，**生成许多锚框。**(这里是对每个pixel作用)
  - multibox_prior(data,sizes,ratios)->Y(批量大小，锚框的数量，4）（1，图片像素数x（s+r-1），4个坐标量）

- 用ground truth 框（真实边界框）去标记所有1中生成的锚框(对应代码中的multibox_targe函数） 标记方法：计算所有锚框和ground-truth的IoU值，**给每一个锚框分配一个真实边界框**（小于阈值的为背景，大于的选一个最接近的ground-truth），即用最近的那个标记，详见3.
  - assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5)->anchors_bbox_map
- **标注类别**：锚框的类别与被分配的真实边界框的类别相同
- **标注偏移量**：对刚刚标记好的所有锚框预测偏移量(offset)（背景类偏移量为零（负类）新类别的整数索引递增一，对应offset_boxes函数）

  的偏移量。 这里介绍一种常见的变换。 给定框A和B，中心坐标分别为(x_a, y_a)和(x_b, y_b)，宽度分别为w_a和w_b，高度分别为h_a和h_b，可以将A的偏移量标记为：

  $$\left(\frac{\frac{x_b-x_a}{w_a}-\mu_x}{\sigma_x}, \frac{\frac{y_b-y_a}{h_a}-\mu_y}{\sigma_y}, \frac{\log\frac{w_b}{w_a}-\mu_w}{\sigma_w}, \frac{\log\frac{h_b}{h_a}-\mu_h}{\sigma_h}\right), \tag{13.4.3}$$

  其中常量的默认值为 $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1$， $\sigma_w = \sigma_h = 0.2$。 这种转换在下面的 `offset_boxes` 函数中实现。
  - offset_boxes(anchors, assigned_bb, eps=1e-6)->offset
  - multibox_target(anchors, labels（真实边框）)->(bbox_offset, bbox_mask, class_labels)
  - 返回的第二个元素是掩码（mask）变量，形状为（批量大小，锚框数的四倍）。 掩码变量中的元素与每个锚框的4个偏移量一一对应。 由于我们不关心对背景的检测，负类的偏移量不应影响目标函数。 通过元素乘法，掩码变量中的零将在计算目标函数之前过滤掉负类偏移量。
- 一个预测好的边界框根据其中某个带有预测偏移量的锚框生成->offset_inverse函数，该函数将锚框和偏移量预测作为输入，并应用逆偏移变换来返回预测的边界框坐标。
  - offset_inverse(anchors, offset_preds)->predicted_bbox
- 用**NMS**合并属于同一目标的类似的预测边界框（选）（挑一个预测最好的，然后把没我预测的好，但是和我预测的很近的框框删掉）
  - multibox_detection(cls_probs(预测的概率), offset_preds, anchors, nms_threshold=0.5,pos_threshold=0.009999999)->torch.stack(out)
  - 我们可以看到返回结果的形状是（批量大小，锚框的数量，6）。 最内层维度中的六个元素提供了同一预测边界框的输出信息。 第一个元素是预测的类索引，从0开始（0代表狗，1代表猫），值-1表示背景或在非极大值抑制中被移除了。

第二个元素是预测的边界框的置信度。 其余四个元素分别是预测边界框左上角和右下角的(x,y)轴坐标（范围介于0和1之间）

- tensor([[[0.00, 0.90, 0.10, 0.08, 0.52, 0.92],
  [1.00, 0.90, 0.55, 0.20, 0.90, 0.88],
  [-1.00, 0.80, 0.08, 0.20, 0.56, 0.95],
  [-1.00, 0.70, 0.15, 0.30, 0.62, 0.91]]])

- 最终得到 每一个 对象或物体 的 一个 最终预测边界框

13.5 多尺度目标检测

问题：为每个像素生成锚框太多了->均匀抽样一小部分像素，以他们为中心生成锚框
另外，在不同尺度下，可以生成不同数量和不同大小的锚框

- 比起较大的目标，较小的目标在图像上出现的可能性更多样。 例如，1×1、1×2和2×2的目标可以分别以4、2和1种可能的方式出现在2×2图像上。 因此，当使用较小的锚框检测较小的物体时，我们可以采样更多的区域（压缩阶段的开始），而对于较大的物体，我们可以采样较少的区域

13.5.1 多尺度锚框

1.读取图像，获得高和宽

```
%matplotlib inline
import torch
from d2l import torch as d2l

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

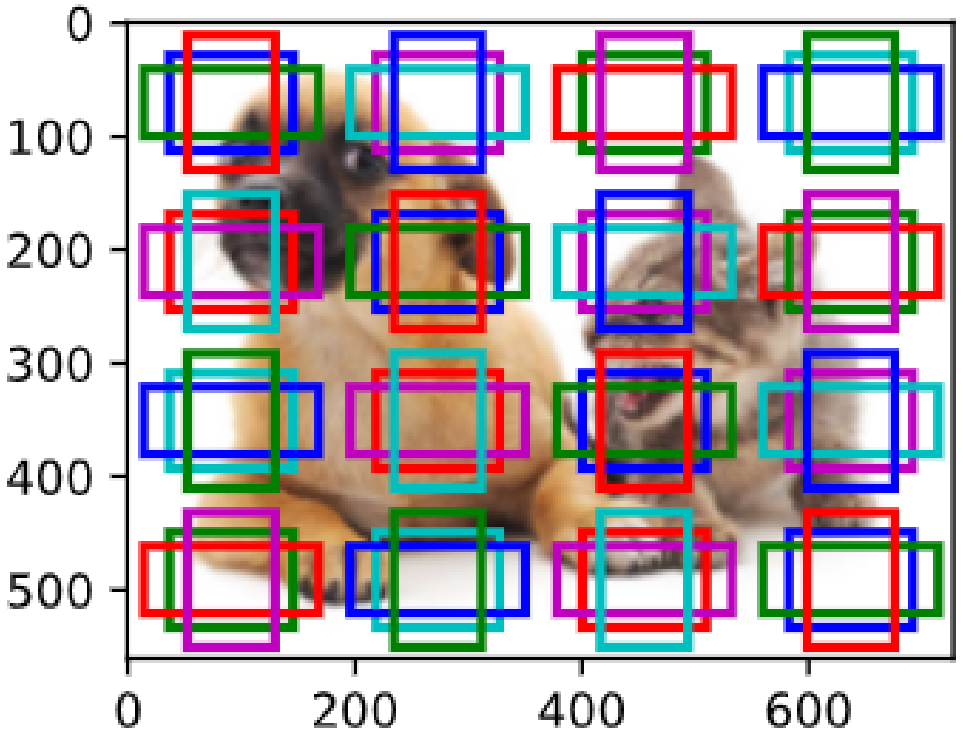
2.在特征图（`fmap`）上生成锚框（`anchors`），每个单位（像素）作为锚框的中心
在特征图上按像素生成->在原图上均匀生成
feature map：某个卷积层的输出

```
def display_anchors(fmap_w, fmap_h, s):
'''获得特征图的宽和高，得到有多少个像素,然后以每个像素为中心生成对应(size)的锚框'''
```

```
    d2l.set_figsize()
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])'''13.4实现的生成锚框的函数'''
    bbox_scale = torch.tensor((w, h, w, h))'''anchor出来的是介于0~1所以显示的时候需要乘以真实图片的高和宽'''
    d2l.show_bboxes(d2l.plt.imshow(img).axes, anchors[0] * bbox_scale)
```

3.探测小目标

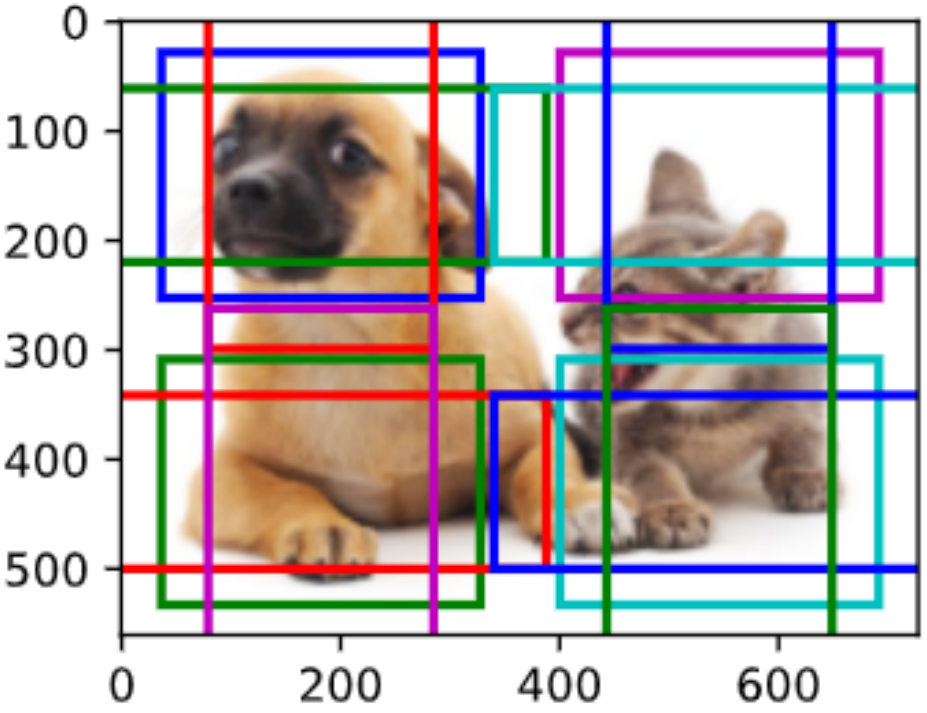
```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



压缩成了4x4，一共十六个像素，每个像素生成s+r-1=1+3-1=3个锚框

4.将特征图的高度和宽度减小一半，然后使用较大的锚框来检测较大的目标

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



5.将特征图的高度和宽度减小一半，然后将锚框的尺度增加到0.8

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```

6.总结，s的设置，越后面的段越大

13.5.2 多尺度检测

假设我们有c张hxw的特征图，这c张特征图是CNN基于输入图像的前向传播算法获得的中间输出。每张特征图上都有hw个不同的空间位置，相同空间位置可以看作含有c个单元。特征图在相同空间位置的c个单元在输入图像上的感受野相同： 它们表征了同一感受野内的输入图像信息。 因此，我们可以将特征图在同一空间位置的c个单元变换为使用此空间位置生成的a个锚框类别和偏移量。 **本质上，我们用输入图像在某个感受野区域内的信息，来预测输入图像上与该区域位置相近的锚框类别和偏移量**

注：在卷积神经网络中，对于某一层的任意元素x，其感受野（receptive field）是指在前向传播期间可能影响x计算的所有元素（来自所有先前层）

13.6 目标检测数据集

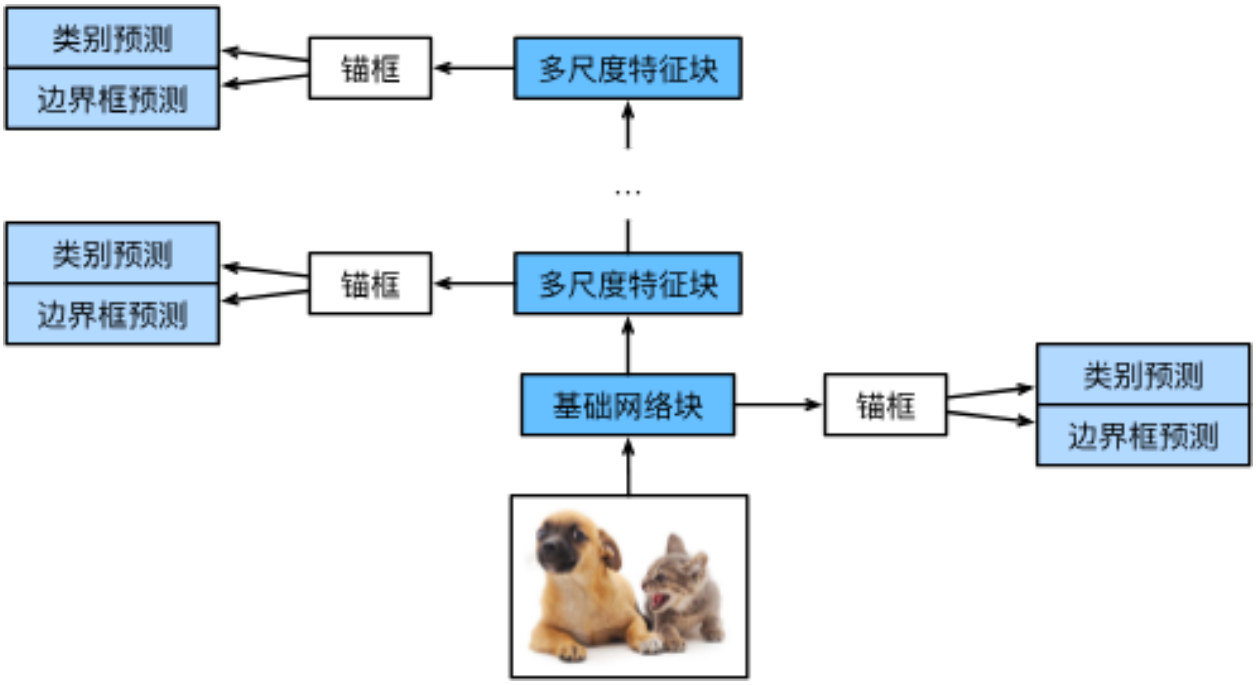
目标检测中的标签还包含真实边界框的信息

13.7 单发多框检测SSD

1.SSD模型

由13.5.2，当不同层的特征图在输入图像上分别拥有不同大小的感受野时，它们可以用于检测不同大小的目标->利用深层神经网络在多个层次上对图像进行分层表示，从而实现多尺度目标检测

接近顶部的多尺度特征图较小，但具有较大的感受野，它们适合检测较少但较大的物体



- 首先进入base network（CNN),进行抽取特征
- 然后进入右边的anchor box，对每一个像素生成锚框，然后预测类别和真实边界框
- 之后通过多个卷积层来减半高宽
- 在每段都生成锚框：越到后面压的越小，那么稍微取大一点的锚框，最后映射回原始的图片框的地方就很大了，所以底部端来拟合小物体，顶部段来拟合大物体
- 不在一开始就搞size很大的原因是，这样的重复就会很多

2.总结

- SSD通过单神经网络来检测模型
- 在多个段的输出上进行多尺度的检测
- 训练算法：首先把训练数据集中的图片分别经过上面那个ssd模型，得到一堆不同尺度下的锚框，和他们预测的类别与偏差，作为预测值。然后利用这些图片本来就标好

的真实边界框为每个锚框标注类别和偏移量，作为标签值。最后利用预测值和标签值之间的差别，计算损失函数，通过优化算法训练。

- 预测算法：把需要预测的图片放到训练好的模型中，输出生成的锚框的预测的类别和和偏差，然后计算每个锚框的概率，放入nms中去重，最后筛掉置信度低于阈值的，输出。

3.代码

1.类别预测层

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
'''这里采用卷积层进行预测而不是全连接层，类似于NiN。在NiN中已经经过global
pooling，高宽变成了1x1，所以输出通道数就是类别数。
我们这里需要预测的是每个像素生成的每个锚框的类别，所以总数量应该是总像素数（输入的高x宽）x每个像素的锚框数（s+r-1）x每个锚框预测的类别数（类别数+1）
这里卷积层kernal size=3，padding=1，也即输出保留输入的高宽，这样输出和输入在特征图宽和高上的空间坐标一一对应。
扫一遍我们对每个像素通过周围3x3的块来判断这个像素的类别，反应为一个数，然后通过通道数（在通道里）最后反映出这个像素的所有锚框在每个类别上的得分，index为i(q+1)+j（0≤j≤q）的通道代表了索引为i的锚框有关类别索引为j的预测
所以最后的输出每个维度都是信息，高、宽和通道
'''
def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)'''+1是加上背景类，对每一个锚框都需要预测是哪一类的概率'''
```

2.边界框预测层

```
'''预测和真实的bounding box的offset,offset是四个值（差）->num_anchors*4'''
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

3.连接多尺度的预测

```
def forward(x, block):
    return block(x)
```

```
Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))

->在不同的尺度下，特征图的形状或以同一单元为中心的锚框的数量可能会有所不同。因此，不同尺度下预测输出的形状可能会有所不同

```
def flatten_pred(pred):
        '''permute是调换顺序，这里把通道维放到了最后，这样就是对像素拉，把每个像素的每个锚框预测出来的类别放在一起。然后把4d的展成了2d的（batch size为一个维度，后面的三个展成一个维度'''
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
        '''几个张量先flatten然后再合并,方便后面处理'''
    return torch.cat([flatten_pred(p) for p in preds], dim=1)

concat_preds([Y1, Y2]).shape
```

torch.Size([2, 25300])（55x20x20+33x10x10）

4.高和宽减半块

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

torch.Size([2, 10, 10, 10])

同时改变通道数，和之前的神经网络架构中的stage一样

5.基本网络块

从抽特征到第一次对feature map做锚框中间的net

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i + 1]))
    '''接了三个block，通道数3到16，16到32，32到64，每次高宽减半'''
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

torch.Size([2, 64, 32, 32])

6.完整的单发多框检测模型由五个模块组成

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1, 1))
    else:'''i=2,3'''
        blk = down_sample_blk(128, 128)
    return blk
```

7.为每个块定义前向计算

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)'''生成锚框'''
    cls_preds = cls_predictor(Y)'''这里直接把Y传进去就可以了，因为函数不需要知道具体锚框长什么样，知道数量就可以了'''
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

8.超参数

0.2,0.37,0.54:0.17;0.272=$\sqrt{0.2 \times 0.54}$,从零到一逐步增大

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5'''常用组合'''
num_anchors = len(sizes[0]) + len(ratios[0]) - 1'''每一层两个size，三个ratio，
所以4个锚框
```

9.定义完整的模型

setattr(object, name, value):object -- 对象,name -- 字符串，对象属性,value -- 属性值

getattr(object, name[, default]): default -- 默认返回值，如果不提供该参数，在没有对应属性时，将触发 AttributeError

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(
                self, f'cls_{i}',
                cls_predictor(idx_to_in_channels[i], num_anchors,
                              num_classes))
            setattr(self, f'bbox_{i}',
                    bbox_predictor(idx_to_in_channels[i], num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(cls_preds.shape[0], -1,
                                      self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

10.创建一个模型实例，然后使用它 执行前向计算

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
```

```
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

output anchors: torch.Size([1, 5444, 4])1（所有的图片都和一张一样，因为是按像素生成的）,5444个，4个坐标

output class preds: torch.Size([32, 5444, 2])批量大小32，5444个，1+1类

output bbox preds: torch.Size([32, 21776])批量大小32，5444个x4个差距

11.读取香蕉检测数据集

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

read 1000 training examples

read 100 validation examples

12.初始化其参数并定义优化算法

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

13.定义损失函数和评价函数

由于我们不关心对背景的检测，负类的偏移量不应影响目标函数。 通过元素乘法，掩码变量中的零将在计算目标函数之前过滤掉负类偏移量。

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')
'''可以看作是分类loss和回归loss加一起'''
def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
        '''labels就是真实的类别和boundingbox，参见13.4'''
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),'''把batch size和个数合
一'''
                   cls_labels.reshape(-1)).reshape(batch_size,
-1).mean(dim=1)
```

```
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
'''下面的是评价函数'''
def cls_eval(cls_preds, cls_labels):
    return float(
        (cls_preds.argmax(dim=-1).type(cls_labels.dtype) ==
cls_labels).sum())'''类似之前分类问题的准确率'''

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) *
bbox_masks)).sum())'''平均绝对误差'''```
#### 14.训练模型
```python
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        anchors, cls_preds, bbox_preds = net(X)'''生成多尺度锚框，为每个锚框预
测类别和偏移量，对应图片分类的生成各类别的softmax概率值'''
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors,
Y)'''使用真实边界框为每个锚框标注类别和偏移量，在图片分类中直接用数据集中标注好的标
签'''
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)'''根据上面算的预测值和label算损失函数'''
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

15.预测目标

```
X=torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```
```

```
def predict(X):
    net.eval()'''预测模式'''
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)'''算每个的概率，
用于后面的nms'''
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

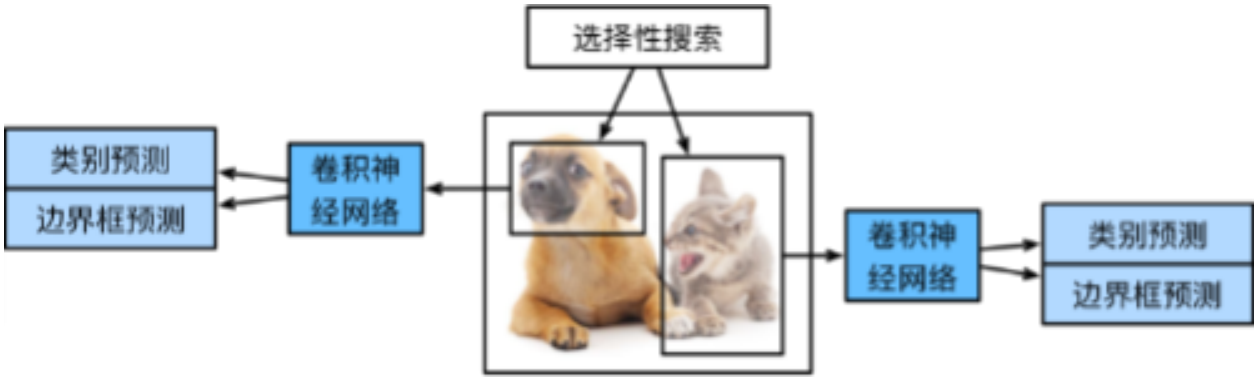
output = predict(X)
```

16.筛选所有置信度不低于 0.9 的边界框，做为最终输出

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```

13.8 区域卷积神经网络 R-CNN



1.R-CNN

- 使用启发式搜索算法来选取多个高质量提议区域（锚框也是一种选取方法）（多尺度下），每个提议区域标注类别和真实边缘框

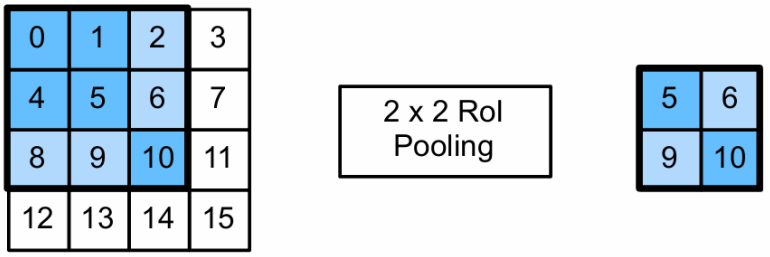
- 使用预训练好的模型（CNN），将每个提议区域变形为网络所需尺寸，并通过前向传播输出抽取的提议区域特征
- 将每个提议区域的特征连同其标注的类别作为一个样本。训练多个支持向量机对目标分类，其中每个支持向量机用来判断样本是否属于某一个类别
- 将每个提议区域的特征连同其标注的边界框作为一个样本，训练线性回归模型来预测真实边界框

  问题：第一、二步计算量太大

2.兴趣区域（RoI）池化层

利于做batch

- 给定一个锚框，均匀分割成 $n \times m$ 块，输出每块里的最大值
- 不管锚框多大，总是输出 nm 个值

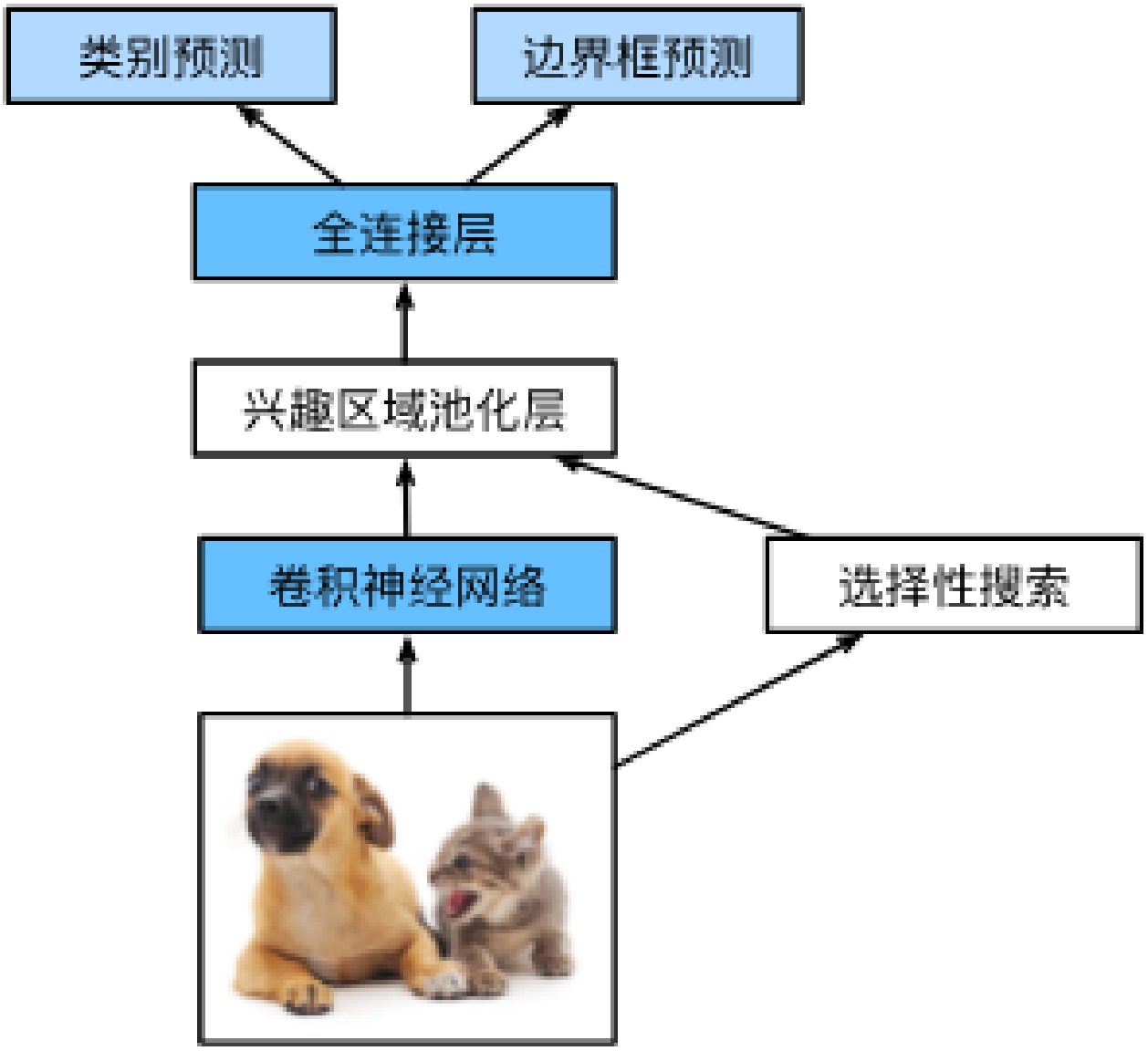


3.Fast R-CNN

关键：CNN不再是对每个锚框抽取，而是对整个图片

R-CNN的主要性能瓶颈在于，对每个提议区域，卷积神经网络的前向传播是独立的，而

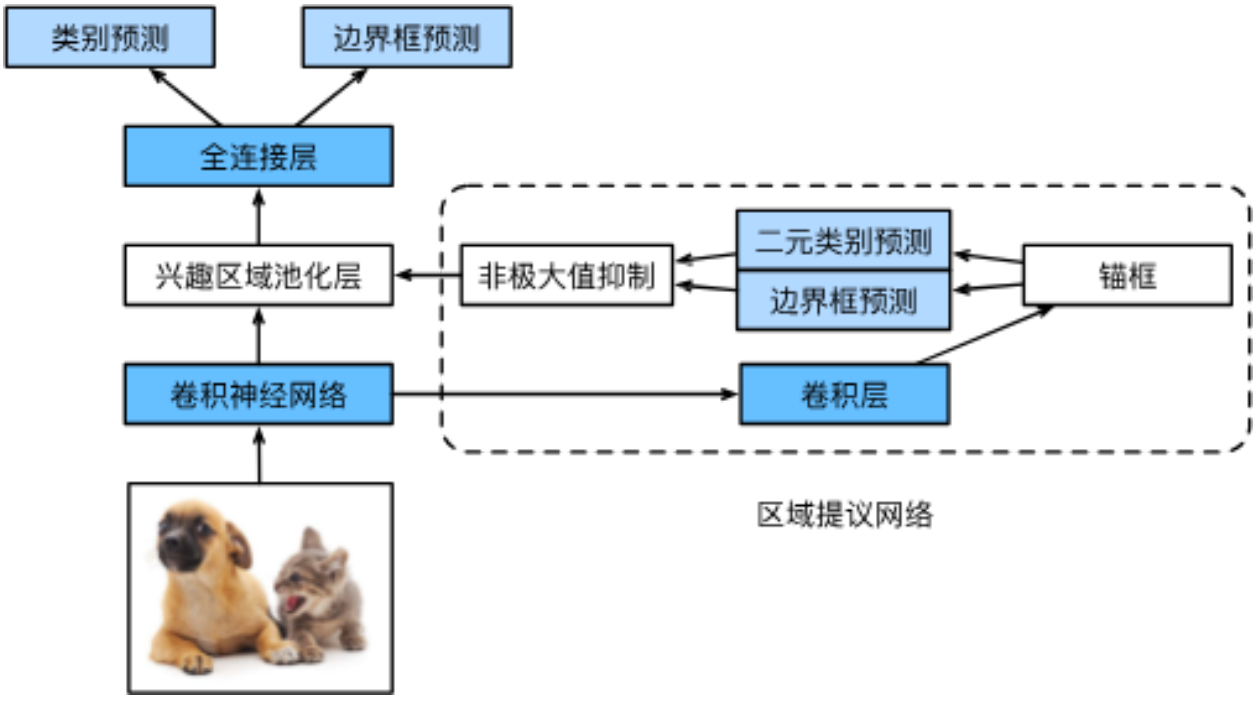
没有共享计算。 由于这些区域通常有重叠，独立的特征抽取会导致重复的计算



- 首先对图片用CNN进行处理，设输入为一张图像，将卷积神经网络的输出的形状记为1×c×h1×w1
- 同时在原始图片上进行选择性搜索（选择锚框），n个。（标出了形状各异的兴趣区域）
- 然后将选择好的锚框映射到处理过的图像上
- 这些感兴趣的区域需要进一步抽取出形状相同的特征（比如指定高度h2和宽度w2），以便于连结后输出。
    - 在R-CNN中，先生成区域，在用卷积修改尺寸和提取特征，最后能不同区域统一用svm和回归进行预测，缺点就是每个都要卷一次。在fast里面，就卷一次，直接对图片卷，然后把区域映射上去，相当于先框区域再压缩，缺点是尺寸不一样，所以要用RoI
- 再对锚框进行RoI pooling，变成一个形状

    - 将卷积神经网络的输出和提议区域作为输入，输出连结后的各个提议区域抽取的特征，形状为n×c×h2×w2
- 通过全连接层将输出形状变换为n×d
- 预测n个提议区域中每个区域的类别和边界框。更具体地说，在预测类别和边界框时，将全连接层的输出分别转换为形状为n×q（q是类别的数量）的输出和形状为n×4的输出。其中预测类别时使用softmax回归

4.Faster R-CNN

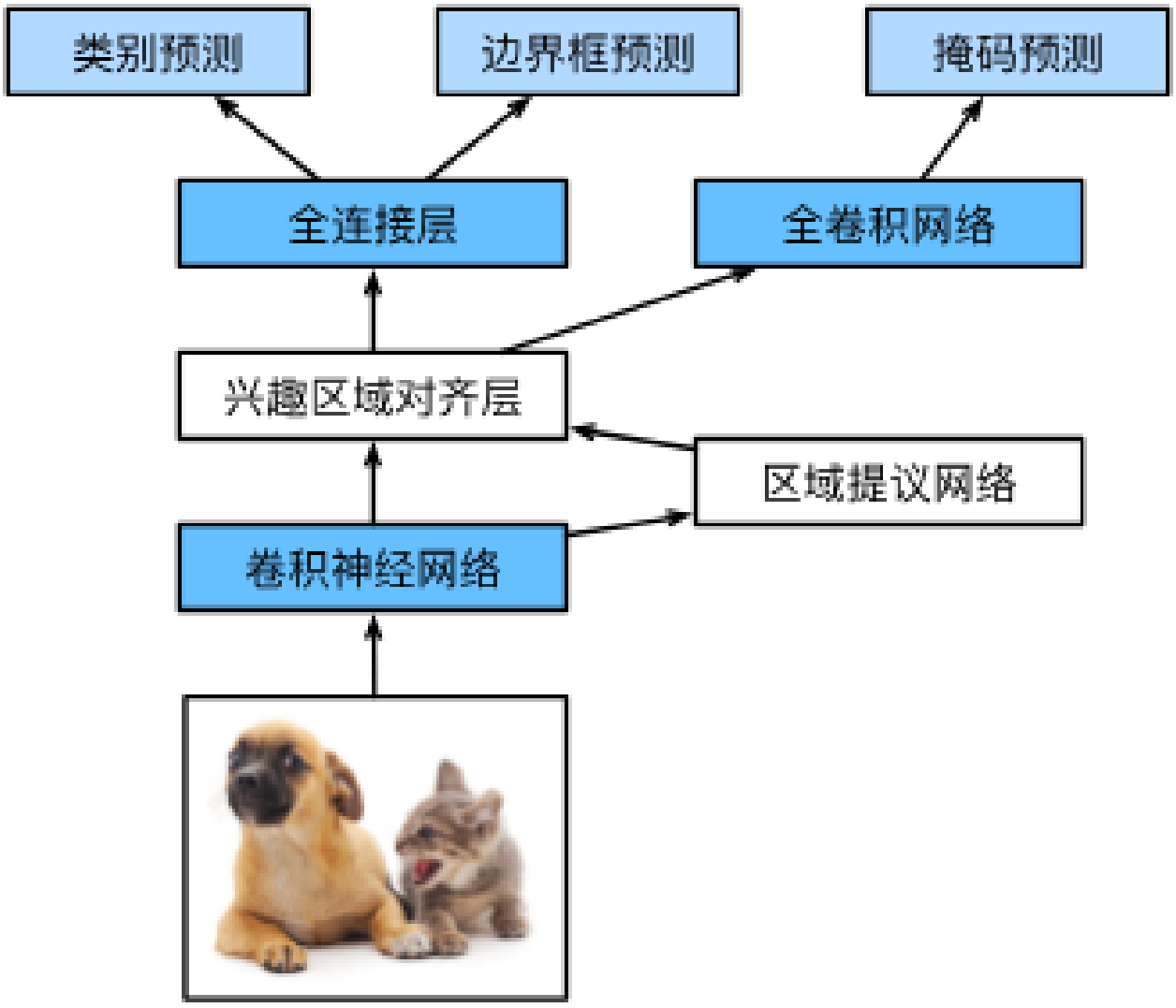


用Regional proposal network来代替原来的Selective search获得更好的锚框（减少提议区域的生成数量，生成不怎么重复的），一个比较糙的目标检测算法：two stage

- 图片先CNN
- 然后输出进入RPN，先再CNN，将输出通道数记为c。这样，卷积神经网络为图像抽取的特征图中的每个单元均得到一个长度为c的新特征
    - 在这个小network里先要筛一次，所以需要有特征，然后后面可以预测、筛选
- 然后生成锚框（13.4的生成方法）
- 使用长度为c的特征，两件事
    - binary category prediction：是否圈住
    - bounding box prediction

- NMS->从预测类别为目标的预测边界框中移除相似的结果。最终输出的预测边界框即是兴趣区域汇聚层所需的提议区域

5.Mask R-CNN



如果有像素级别的标号，使用FCN来利用这些信息（fully convolutional network）

RoI pooling->RoI align,直接中间切开，不管原有边缘，对于被切开的像素，进行加权，保留特征图上的空间信息，从而更适于像素级预测

6.总结

1.R-CNN基于锚框和CNN的目标检测算法

2.Faster R-CNN &Mask R-CNN是在追求高精度场景下的常用算法

44.物体检测算法：YOLO

1.you look only once

2.问题：SSD中锚框大量重叠，浪费了很多计算->YOLO将图片均匀分成SxS个锚框，每个锚框预测B个边缘框

3.总结：目标检测算法主要分为两个类型

- （1）two-stage方法，如R-CNN系算法（region-based CNN），其主要思路是先通过启发式方法（selective search）或者CNN网络（RPN)产生一系列稀疏的候选框，然后对这些候选框进行分类与回归，two-stage方法的优势是准确度高
- （2）one-stage方法，如Yolo和SSD，其主要思路是均匀地在图片的不同位置进行密集抽样，抽样时可以采用不同尺度和长宽比，然后利用CNN提取特征后直接进行分类与回归，整个过程只需要一步，所以其优势是速度快，但是均匀的密集采样的一个重要缺点是训练比较困难，这主要是因为正样本与负样本（背景）极其不均衡，导致模型准确度稍低

13.9 语义分割和数据集

1.语义分割（semantic segementation）将图片中的每个像素分类到对应的类别：语义区域的标注和预测是像素级的

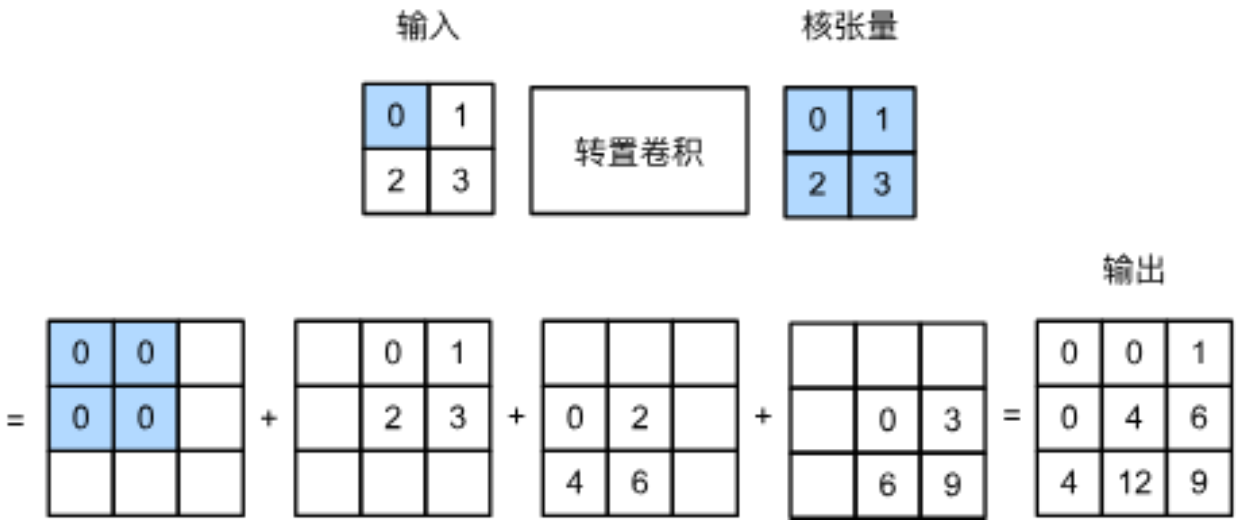
2.应用：背景虚化、路面分割

3.与实例分割的区别：

13.10 转置卷积

1.问题：之前的卷积不会增大输入的高宽，通常要么不变，要么减半。这对于像素级处理是不行的。转置卷积（transposed convolution）可以用来增大输入高宽

2.计算方法



0和kernal中的元素逐个做乘法，然后逐个写上去（保持核的大小）

$Y[i:i+h,j:j+w]+=X[i,j]\cdot K$

3.为什么是"转置"

- 对于卷积$Y=X\bigstar W$
  - 可以对W构造一个V，使得卷积等价于矩阵乘法Y'=VX'($n=n\times m\cdot m$)
  - Y'，X'是Y，X对应的向量版本
- 转置卷积等价于$Y'=V^TX'$($m=m\times n\cdot n$)
- ->如果卷积将输入从（h，w）变成（h'，w）
  - 同样超参数的转置卷积则从（h'，w'）变成（h，w）
- 转置卷积层能够交换卷积层的正向传播函数和反向传播函数

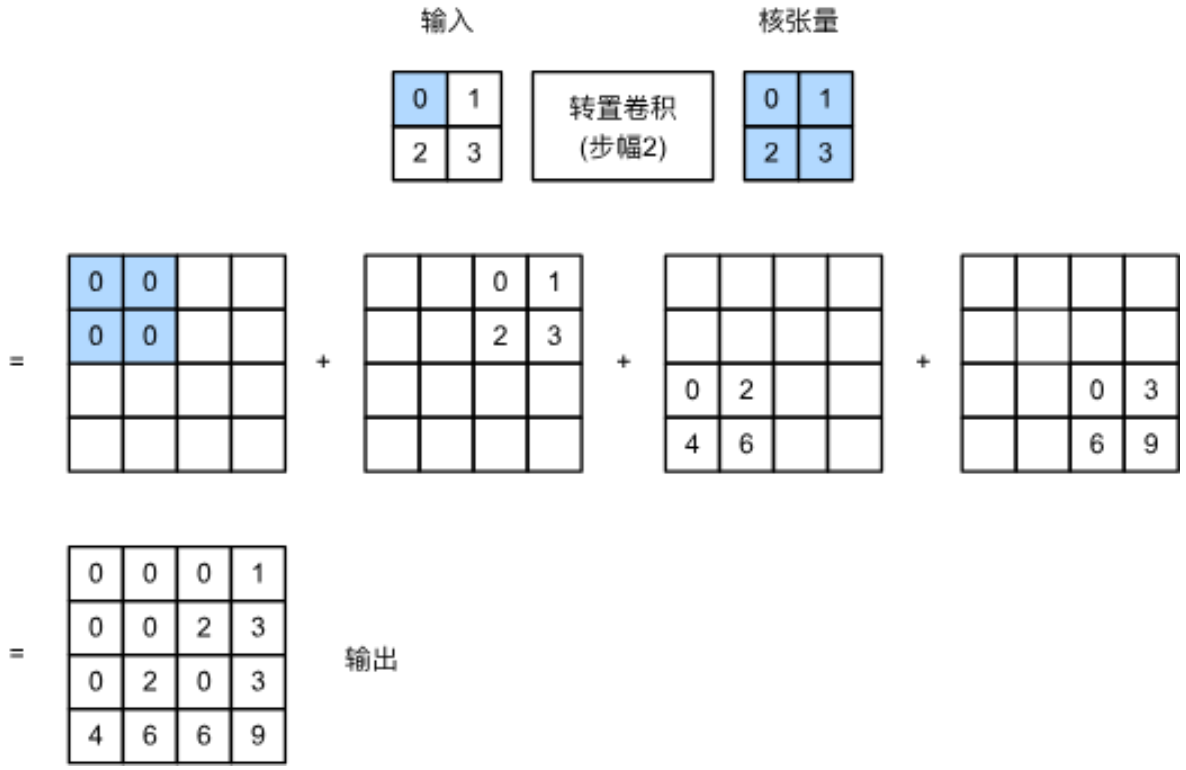
  4.填充

  tconv = nn.Conv2DTranspose(1, kernel_size=2, padding=1)

  转置卷积的输出中将删除第一和最后的行与列

  就是说转置卷积可以理解为就是卷积反着来，padding=1是加在输入层上。先想个卷积的过程，一个1x1的图像用(1, kernel_size=2, padding=1)卷积，得到1-2+2+1，也就是2x2的输出。所以逆过程2x2的经过转置卷积，得到1x1的输出。所以转置卷积填充是让输入变小。

  5.步幅



所以转置卷积步幅是让输入变大

6.通道

与卷积相同

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

True

7.

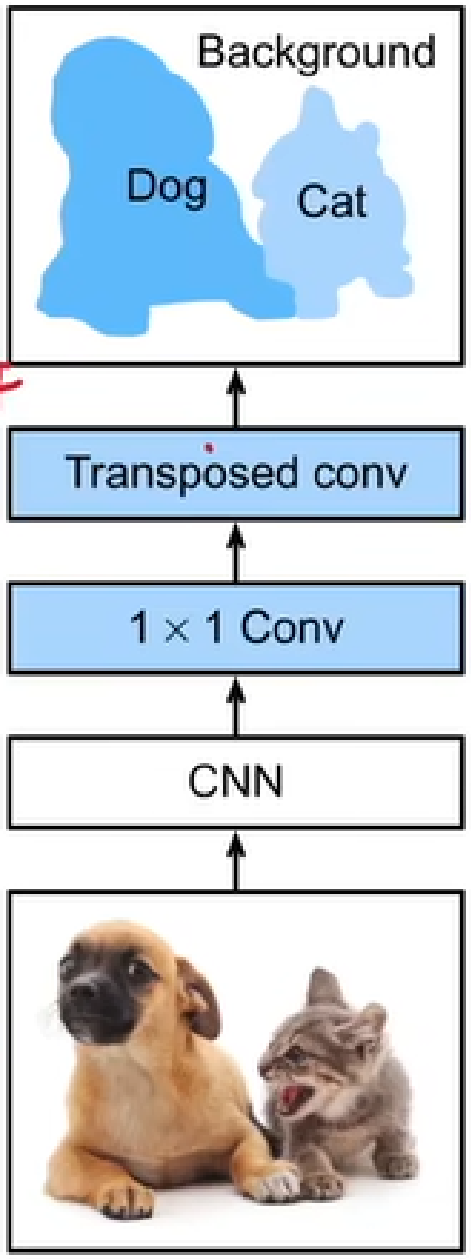
反卷积的意义是卷积之后的矩阵的每个元素有一个感受视野，反卷积希望通过这个元素还原感受视野里面的内容。并且由于卷积后的元素的感受视野有相交的情况，所以反卷积中也出现了结果中有些元素的值来源于卷积结果的一个或多个元素的现象，理论上通过反卷积，我们可以通过将特征图反卷积还原到原图大小从而获取到我们的卷积核在原图中提取的是什么信息

13.11 全卷积网络

1.FCN（fully convolutional network）

它用转置卷积层来替换CNN最后的全连接层，从而可以实现每个像素的预测

网络来
生工
替换
云，从
素的预



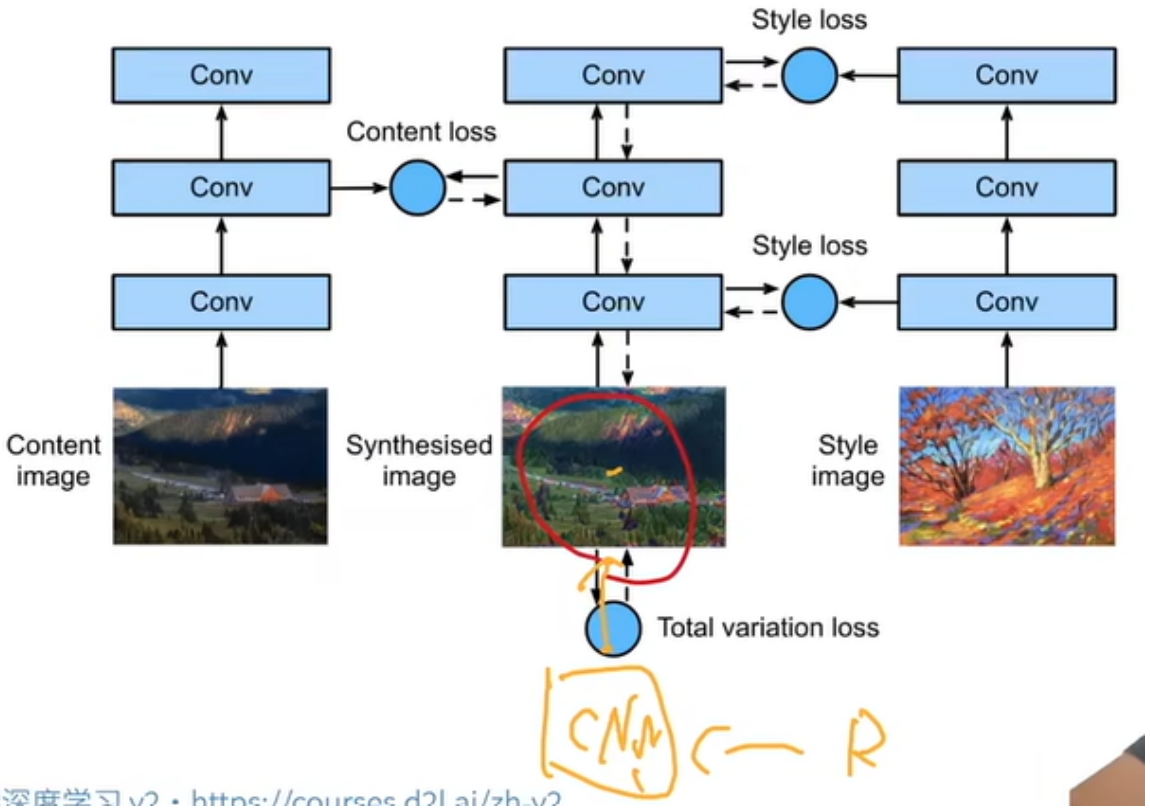
k（通道数）是k个类别，表示每个像素k个类别的概率

13.12 风格迁移

1.样式迁移：将样式中图片中的样式迁移到内容图片上，得到合成图片

2.基于CNN的样式迁移

- 奠基性工作



- 合成图像是风格迁移过程中唯一需要更新的变量，即风格迁移所需迭代的模型参数
- 全变分损失有助于减少合成图像中的噪点
- 假设该输出的样本数为1，通道数为c，高和宽分别为h和w，我们可以将此输出转换为矩阵X，其有c行和hw列。 这个矩阵可以被看作由c个长度为hw的向量$x_1, \ldots, x_c$组合而成的。其中向量xi代表了通道i上的风格特征。在这些向量的格拉姆矩阵$XX^T \in R^{c \times c}$中，i行j列的元素x_{ij}即向量x_i和x_j的内积。它表达了通道i和通道j上风格特征的相关性。我们用这样的格拉姆矩阵来表达风格层输出的风格
- 简而言之，用通道表示一个点的特征，用通道之间的统计关系表示图片的风格（gram matrix）
- 合成图像里面有大量高频噪点，即有特别亮或者特别暗的颗粒像素。 一种常见的去噪方法是全变分去噪（total variation denoising）： 假设xi,j表示坐标(i,j)处的像素值，降低全变分损失$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$能够尽可能使邻近的像素值相似。