# 第二讲：

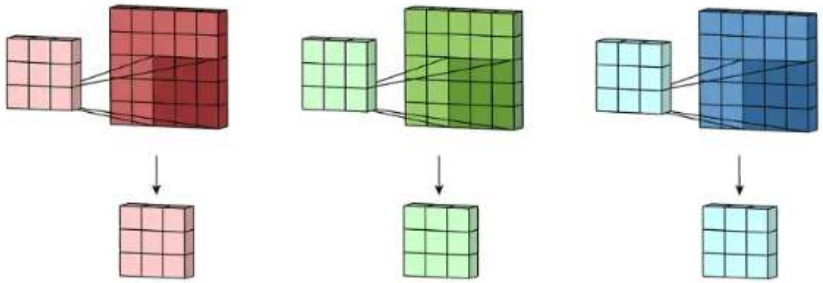
## 知识蒸馏与低秩分解

讲师： 王欢

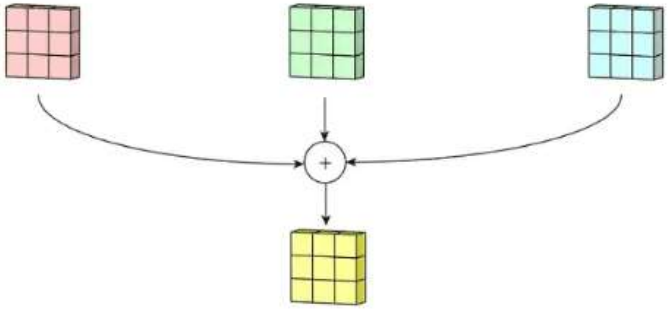
# 卷积

- 上次课，很重要的一种手段，通过改变卷积形状，这次课进一步看看其他方法减少计算
- 回看一下各种形式的卷积，有助于我们了解张量的概念

拥有多个通道的卷积，例如处理彩色图像时，分别对 R, G, B 这 3 个层处理的 3 通道卷积，如下图：



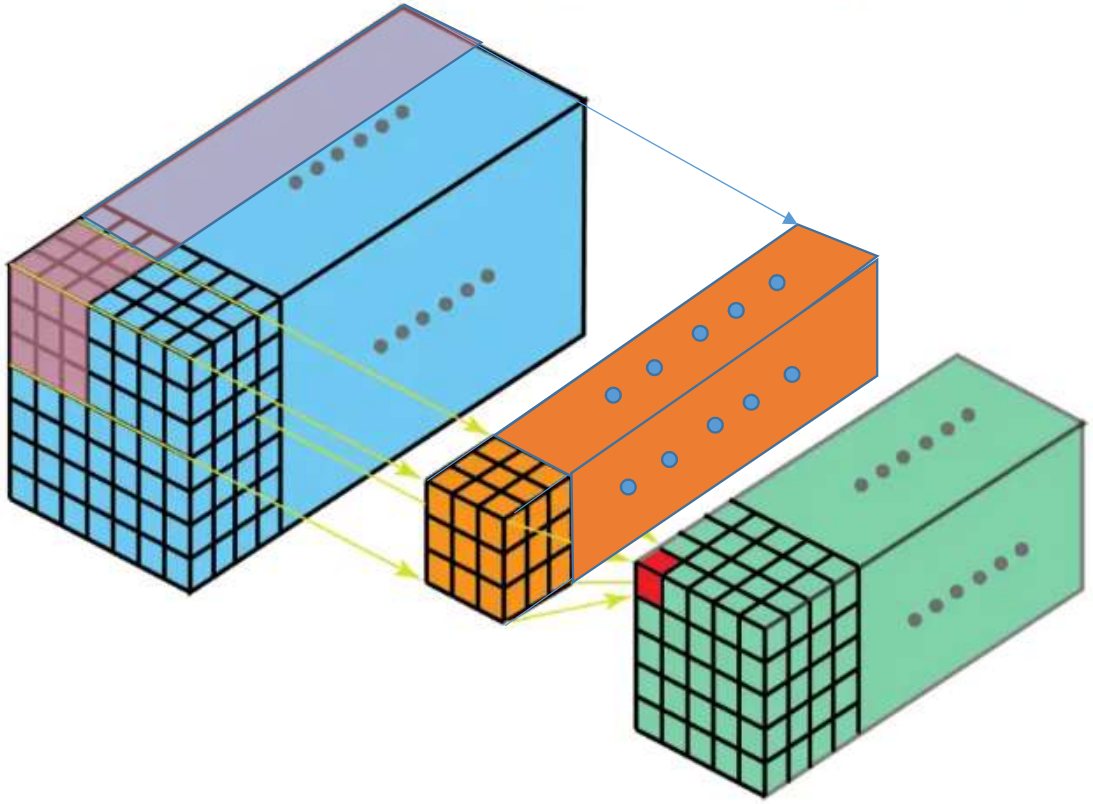
再将三个通道的卷积结果进行合并（一般采用元素相加），得到卷积后的结果，如下图：



(图片来源网络)

# 卷积-3D卷积

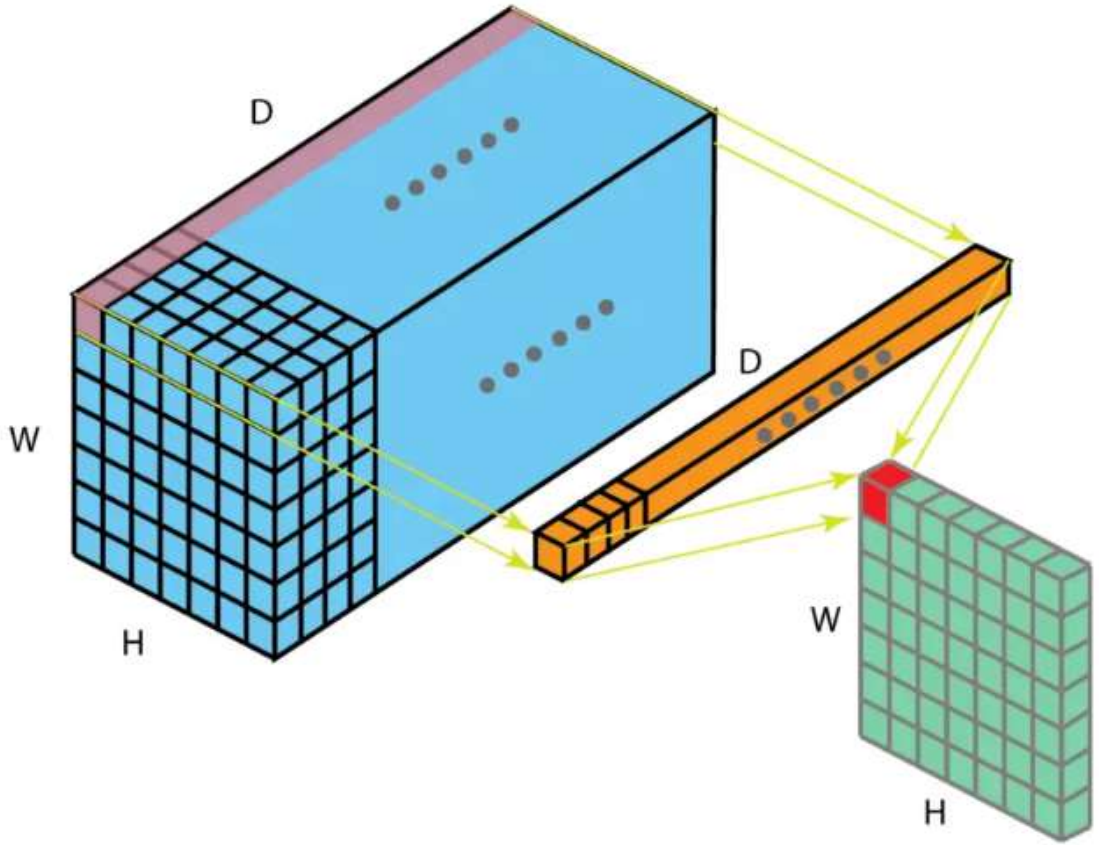
卷积有三个维度（高度、宽度、通道），沿着输入图像的 3 个方向进行滑动，最后输出三维的结果，如下图：



(图片来源网络)

# 卷积-1x1卷积

当卷积核尺寸为 1x1 时的卷积，也即卷积核变成只有一个数字。如下图：

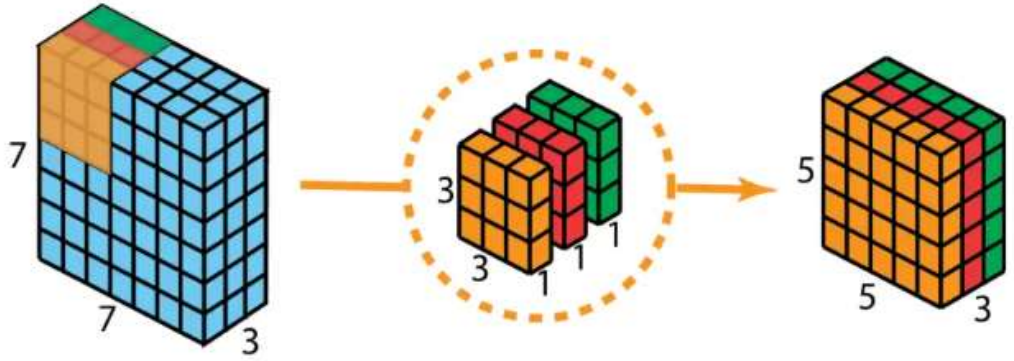



(图片来源网络)

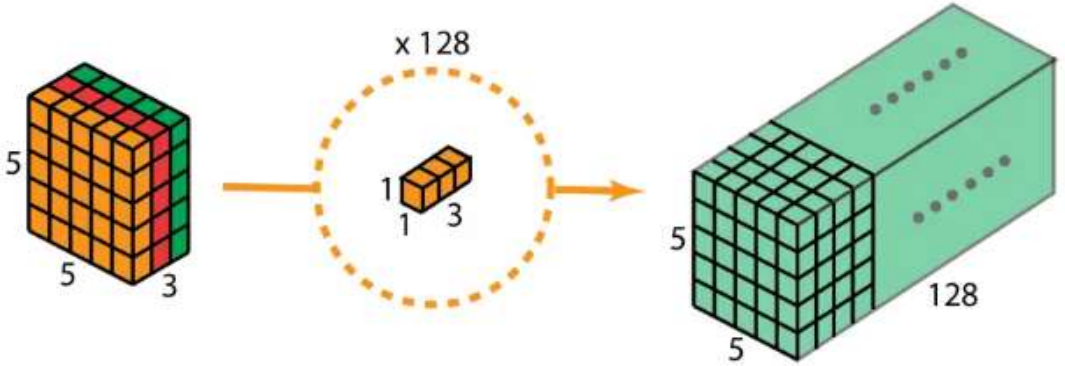
# 卷积-深度可分离卷积

深度可分离卷积由两步组成：深度卷积和 1x1 卷积。

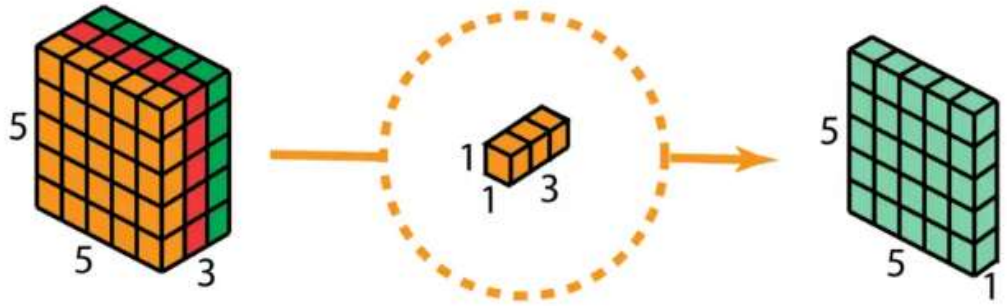
首先，在输入层上应用深度卷积。如下图，使用 3 个卷积核分别对输入层的 3 个通道作卷积计算，再堆叠在一起。



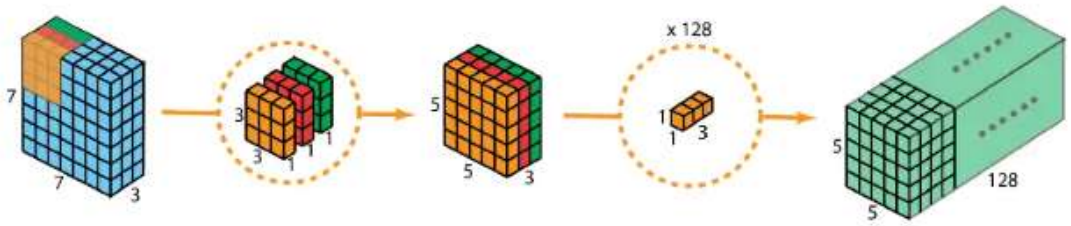

重复多次 1x1 的卷积操作（如下图为 128 次），则最后便会得到一个深度的卷积结果。




再使用 1x1 的卷积（3 个通道）进行计算，得到只有 1 个通道的结果



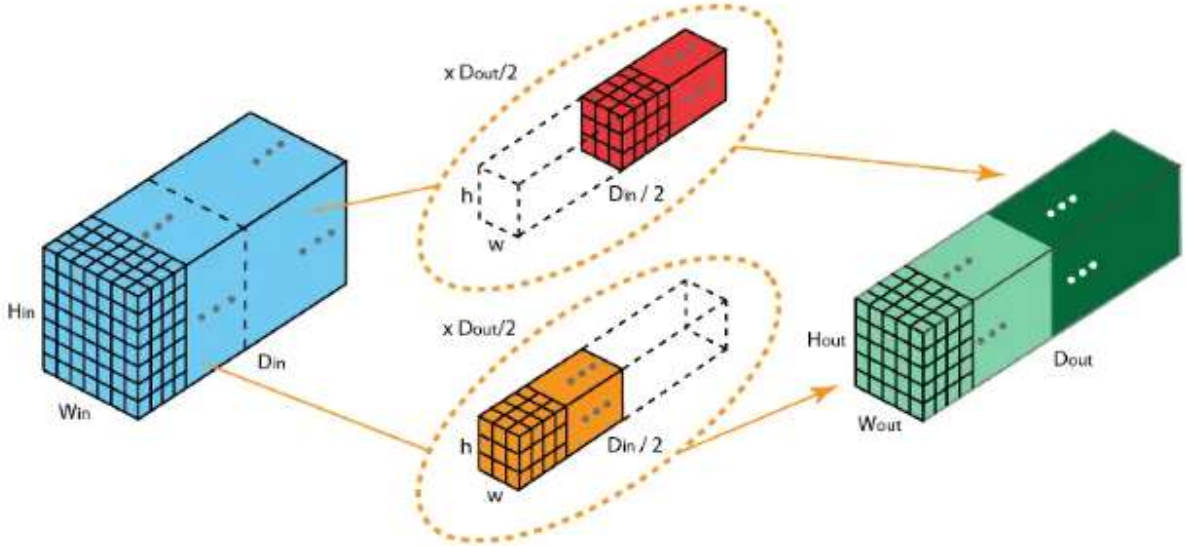

完整的过程如下：




（图片来源网络）

# 卷积-组卷积

在分组卷积中，卷积核被分成不同的组，每组负责对相应的输入层进行卷积计算，最后再进行合并。如下图，卷积核被分成前后两个组，前半部分的卷积组负责处理前半部分的输入层，后半部分的卷积组负责处理后半部分的输入层，最后将结果合并组合。

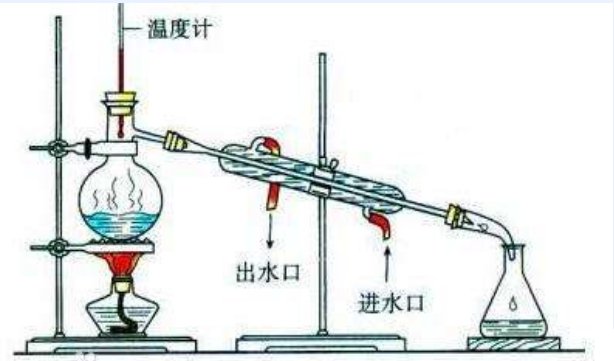



(图片来源网络)

# 蒸馏(酒)

蒸馏酒是把经过发酵的酿酒原料，经过一次或多次的蒸馏过程提取的高酒度酒液。蒸馏酒的制作原理是根据酒精的物理性质，采取使之汽化的方式，提取的高纯度酒液。因为酒精的汽化点是78.3℃，达到并保持这个温度就可以获得汽化酒精，如果再将汽化酒精输入管道冷却后，便是液体酒精。但是在加热过程中，原材料的水分和其他物质也会掺杂在酒精中，因而形成质量不同的酒液。所以大多数的名酒都采取多次蒸馏法等工艺来获取纯度高、杂质含量少的酒液。

蒸馏酒是一种含酒精的饮料，是由含酒精的液体里蒸馏出来的，与原来的液体中酒精含量多少无关，由蒸馏可得到酒精，其原理很简单。因为酒精变成气体比水变成气体所需的温度要低　。



水蒸气蒸馏装置

# 知识蒸馏

- ”好模型的目标不是拟合训练数据，而是学习如何泛化到新的数据“ – 知乎李如
- “神经网络用剩的logits不要扔，沾上鸡蛋液，裹上面包糠...”-某人

**背景**

- 训练和部署使用的模型存在着一定的不一致性:

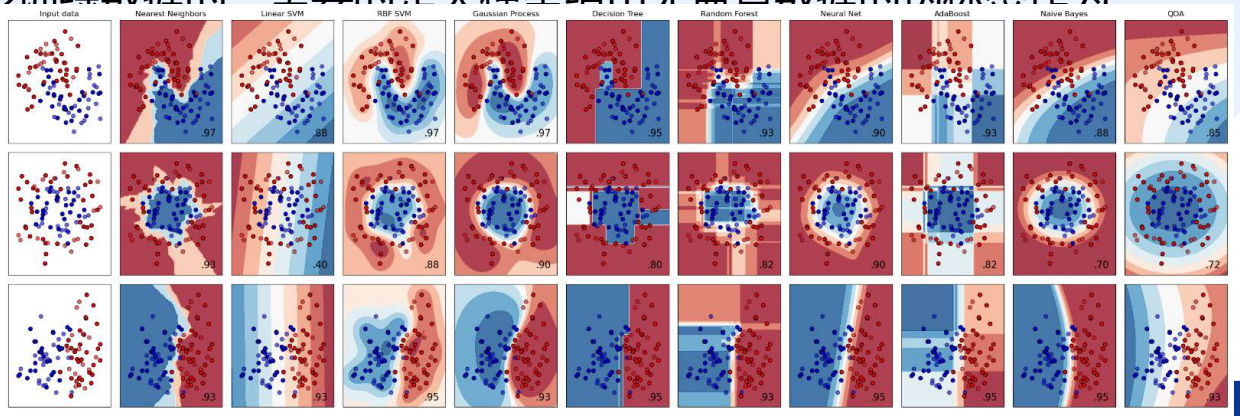
在训练过程中，我们需要使用复杂的模型，大量的计算资源，以便从非常大、高度冗余的数据集中提取出信息。在实验中，效果最好的模型往往规模很大，甚至由多个模型集成得到。而大模型不方便部署到服务中去，常见的瓶颈如下:

- 推断速度慢
- 对部署资源要求高(内存，显存等)
- 在部署时，我们对延迟以及计算资源都有着严格的限制。

因此，模型压缩（在保证性能的前提下减少模型的参数量）成为了一个重要的问题。而”模型蒸馏“属于模型压缩的一种方法。

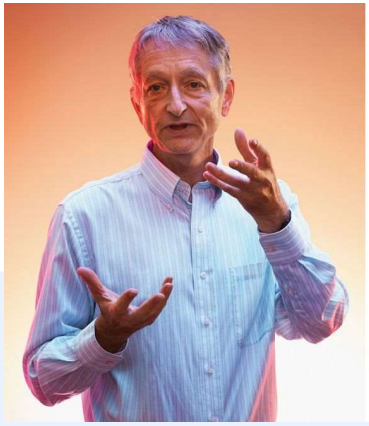
# 知识蒸馏

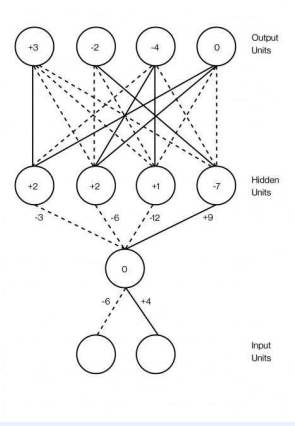
- 本质是通过一种映射关系，将老师学到的东西传递给学生网络。
- 疑问？我有训练数据了，训练数据的准确度肯定比你大模型的输出结构准确度高，为什么还需要从老师网络来学习知识？
- **回答：**大模型的输出对于logits不仅仅是类别属于哪一个，还有一个特点就是会给出不同类别之间的一个关系。真实标签只能告诉我们，某个图像样本是一辆宝马，不是一辆垃圾车，也不是一颗萝卜；而经过训练的softmax可能会告诉我们，它最可能是一辆宝马，不大可能是一辆垃圾车，但绝不可能是一颗萝卜-”暗知识“
- **半监督/无监督，**有些时候是没有那么多训练数据的，需要的是大模型给出无监督数据的伪标签作为冷启动也是不错的。




Classifier comparison — scikit-learn 1.1.0 documentation

# 知识蒸馏






- **理论依据-《** Distilling the Knowledge in a Neural Network**》**

**1. Teacher Model和Student Model**

- **Teacher—Student模型：** teacher是“知识”的输出者，student是“知识”的接受者。
- 知识蒸馏的过程分为2个阶段:
  - **原始模型训练:** 训练“Teacher模型”Net-T，它的特点是模型相对复杂，“Teacher模型”不作任何关于模型架构、参数量、是否集成方面的限制，唯一的要求就是，也就是上一课的ROC/mAP/mIoU指标好，无明显bias
  - **精简模型训练:** 训练“Student模型” Net-S，它是参数量较小、模型结构相对简单的单模型。同样的，比如分类模型，经过softmax后同样能输出对应类别的概率值。
- 论文中，作者将问题限定在分类问题下，而其他本质上属于分类问题的也同样适用，共同点是模型最后会有一个softmax层，其输出值对应了相应类别的概率值。

# 知识蒸馏的关键点

- **通过增加网络容量，获得泛化能力强的模型：**在某问题的所有数据上都能很好地反应输入和输出之间的关系，无论是训练数据，还是测试数据，还是任何属于该问题的未知数据。
- 我们在利用Net-T来蒸馏训练Net-S时，可以直接让Net-S去学习Net-T的泛化能力。
- **直白且高效的迁移泛化能力的方法就是：**使用softmax层输出的类别的概率来作为“soft target”。

【KD的训练过程和传统的训练过程的对比】

- **传统training过程(hard targets):** 对ground truth求极大似然，交叉熵和极大似然的关系
- **KD的training过程(soft targets):** 用large model的class probabilities作为soft targets

# 知识蒸馏的关键点

- 极大似然函数

假设有 $n$ 个样本：$(x^{(i)}, y^{(i)})$，则似然函数为 $\mathcal{L}((x^{(i)}, y^{(i)}); p) = \prod_{i=1}^{n} \prod_{k=1}^{K} p_k^{y_k}$

我们期望最大化似然估计，即最小化负对数似然函数： $\min -\sum_{i=1}^{n} \sum_{k=1}^{K} y_k \log p_k$

由于采用one-hot编码，故 $y_k = 0$ 的项乘积均为0，只需考虑 $y_k = 1$ 时。故上述函数可变形为：$\min -\sum_{i=1}^{n} \log P_k(y_k = 1)$

- KL散度-用于衡量两个分布P、Q的距离

对于离散型随机变量而言KL Divergence： $D_{KL}(P||Q) = \sum_x P(x)(\log P(x) - \log Q(x))$

- 熵与交叉熵

交叉熵则可以衡量使用Q的编码方案对具有分布P的x进行编码的最小比特数。其被定义为：

$H(P,Q) = -\sum_x P(x) \log Q(x)$ $H(P,Q) = -\sum P \log Q = -\sum P \log P + \sum P \log P - \sum P \log Q = H(P) + \sum P \log \frac{P}{Q} = H(P) + D_{KL}(P||Q)$

由于 H(P)与模型参数无关，可以视为常数。**故最小化KL距离等价于最小化交叉熵**。(蒸馏源于这样思考)

在深度学习中， P一般为真实标签的分布， Q 一般为模型预测输出的分布。

# 知识蒸馏的关键点

## 交叉熵损失函数

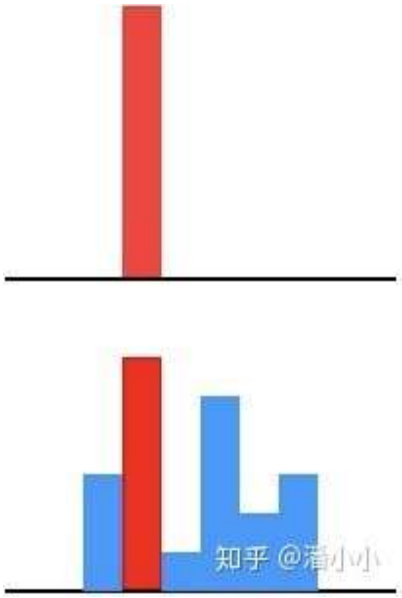
引入参数x的条件分布，我们希望能够最小化真实分布 $P(y|x)$ 与模型输出分布 $P(\hat{y}|x)$ 的距离，等价于最小化两者的交叉熵，其被定义为：

$$-\sum_{y} P(y|x)\log Q(\hat{y}|x) = -\sum_{k=1}^{K} y_k \log \hat{y_k}$$

由此可见，**最小化交叉熵和最小化负对数似然函数是等价的**

# KD的训练过程为什么更有效?

- softmax层的输出，除了正例之外，负标签也带有大量的信息，比如某些负标签对应的概率远远大于其他负标签（宝马，兔子和垃圾车）。而在传统的训练过程(hard target)中，所有负标签都被统一对待。也就是说，KD的训练方式使得每个样本给Net-S带来的信息量大于传统的训练方式。

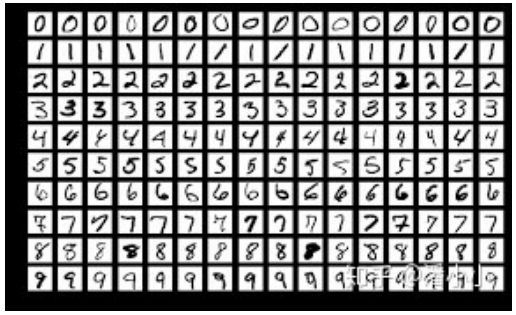


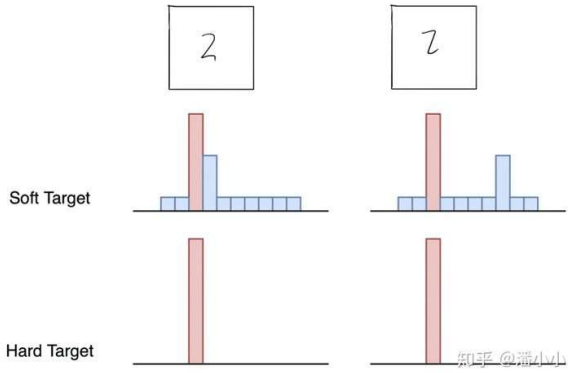

(图片来源网络)

# 一个例子：MNIST

在手写体数字识别任务MNIST中，输出类别有10个。

假设某个输入的“2”更加形似"3"，softmax的输出值中"3"对应的概率为0.1，而其他负标签对应的值都很小，而另一个"2"更加形似"7"，"7"对应的概率为0.1。这两个"2"对应的hard target的值是相同的，但是它们的soft target却是不同的，由此我们可见soft target蕴含着比hard target多的信息。并且soft target分布的熵相对高时，其soft target蕴含的知识就更丰富。

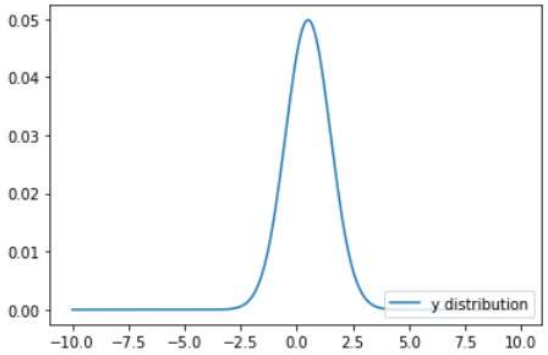


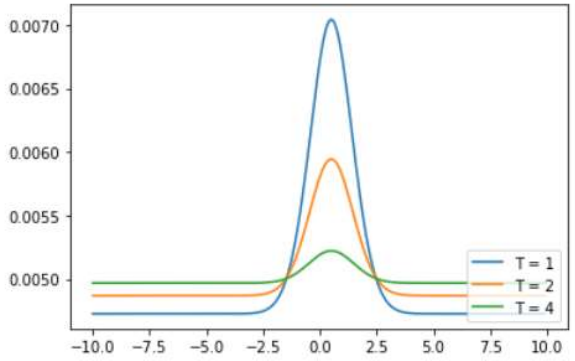



（图片来源网络）

# 带“温度”的softmax

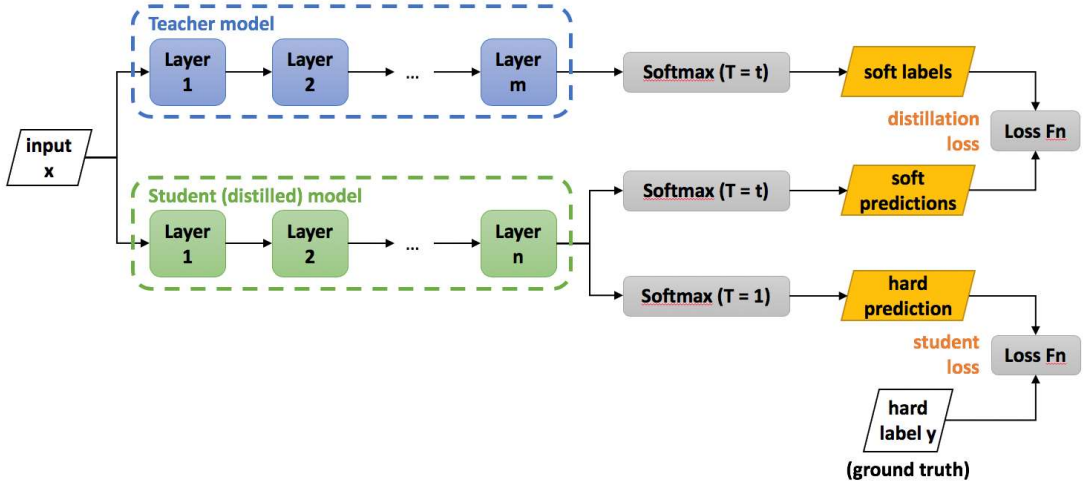
- 先回顾一下原始的softmax函数 $$q_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$
- 但要是直接使用softmax层的输出值作为soft target, 这又会带来一个问题: 当softmax输出的概率分布熵相对较小时，负标签的值都很接近0，对损失函数的贡献非常小，小到可以忽略不计。因此"温度"这个变量就派上了用场。
- 下面的公式时加了温度这个变量之后的softmax函数: $$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$
- 这里的T就是温度。
- 原来的softmax函数是T = 1的特例。 T越高，softmax的output probability distribution越趋于平滑，其分布的熵越大，负标签携带的信息会被相对地放大，模型训练将更加关注负标签。



# 知识蒸馏的具体方法

- **通用的知识蒸馏方法**
  - 第一步是训练Net-T；
  - 第二步是在高温T下，蒸馏Net-T的知识到Net-S
  - 训练Net-T的过程很简单，下面详细讲讲第二步:高温蒸馏的过程。高温蒸馏过程的目标函数由distill loss(对应soft target)和student loss(对应hard target)加权得到。示意图如。




知识蒸馏示意图(来自https://nervanasystems.github.io/distiller/knowledge_distillation.html)

# 知识蒸馏的具体方法

- Net-T 和 Net-S同时输入 transfer set (这里可以直接复用训练Net-T用到的training set), 用Net-T产生的softmax distribution (with high temperature) 来作为soft target，Net-S在相同温度T条件下的softmax输出和soft target的cross entropy就是Loss函数的第一部分

$$L_{soft}$$

- Net-S在T=1的条件下的softmax输出和ground truth的cross entropy就是Loss函数的第二部分

$$L_{hard}$$

- 第二部分Loss必要性其实很好理解: Net-T也有一定的错误率，使用ground truth可以有效降低错误被传播给Net-S的可能。打个比方，老师虽然学识远远超过学生，但是他仍然有出错的可能，而这时候如果学生在老师的教授之外，可以同时参考到标准答案，就可以有效地降低被老师偶尔的错误“带偏”的可能性
- **【注意】 在Net-S训练完毕后，做inference时其softmax的温度T要恢复到1.**

$$L = \alpha L_{soft} + \beta L_{hard}$$

$v_i$ : Net-T的logits

$z_i$ : Net-S的logits

$p_i^T$ : Net-T的在温度=T下的softmax输出在第i类上的值

$q_i^T$ : Net-S的在温度=T下的softmax输出在第i类上的值

$c_i$ : 在第i类上的ground truth值, $c_i \in \{0, 1\}$, 正标签取1，负标签取0.

$N$ : 总标签数量

$$L_{soft} = -\sum_j^N p_j^T \log(q_j^T) \quad p_i^T = \frac{\exp(v_i/T)}{\sum_k^N \exp(v_k/T)} \quad q_i^T = \frac{\exp(z_i/T)}{\sum_k^N \exp(z_k/T)}$$

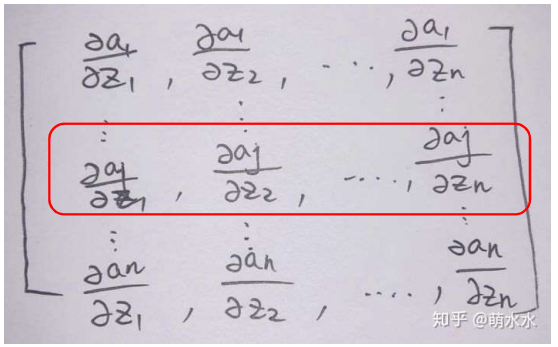
$$L_{hard} = -\sum_j^N c_j \log(q_j^1) \quad q_i^1 = \frac{\exp(z_i)}{\sum_k^N \exp(z_k)}$$

```python
class TeacherNet(nn.Module):
    """
    Network architecture taken from https://github.com/pytorch/examples/blob/master/mnist/main.py

    98.2% accuracy after 1 epoch
    """

    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        return x


class StudentNet(nn.Module):
    """
    Naive linear model

    92.8% accuracy after 5 epochs, single FC layer
    """

    def __init__(self):
        super().__init__()

        self.fc1 = nn.Linear(28 * 28, 16)
        self.fc2 = nn.Linear(16, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x
```

解释一下 $T^2$

```python
class KDMoudle(pl.LightningModule):
    def __init__(self, teacher, student, learning_rate, temperature, alpha):
        super().__init__()

        self.teacher = teacher
        self.teacher.requires_grad_(False)
        self.student = student

        self.learning_rate = learning_rate

        self.temperature = temperature
        self.alpha = alpha

    def forward(self, x):
        student_logits = self.student(x)
        teacher_logits = self.teacher(x)

        return student_logits, teacher_logits

    def training_step(self, batch, batch_index):
        x, y = batch
        student_logits, teacher_logits = self.forward(x)

        # # NOTE:第一组：直接用hard_loss训练student网络
        # loss = F.cross_entropy(student_logits, y)
        #
        # # NOTE:第二组：用soft_loss训练student网络
        # loss = nn.KLDivLoss()(F.log_softmax(student_logits / self.temperature),
        #                       F.softmax(teacher_logits / self.temperature)) * (
        #                self.alpha * self.temperature * self.temperature)

        # NOTE:第三组：用hard_loss+soft_loss训练student网络
        soft_loss = nn.KLDivLoss()(F.log_softmax(student_logits / self.temperature),
                                   F.softmax(teacher_logits / self.temperature)) * (
                            self.alpha * self.temperature * self.temperature)
        hard_loss = F.cross_entropy(student_logits, y) * (1.0 - self.alpha)
        loss = hard_loss + soft_loss

        # WHY:student_logits为什么用log_softmax 而 teacher_logits直接用softmax?

        self.log("student_train_loss", loss)
        return loss
```

# softmax交叉熵导数




1. 输入为z向量，维度为（1，n） $z = [z_1, z_2, z_3, \dots z_n]$
2. 经过softmax函数， $a = \left[\frac{e^{z_1}}{\sum_{k=1}^{n} e^{z_k}}, \frac{e^{z_2}}{\sum_{k=1}^{n} e^{z_k}}, \dots, \frac{e^{z_n}}{\sum_{k=1}^{n} e^{z_k}}\right]$ $a_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$
3. Softmax Loss损失函数定义为L，
4. 假设第 j 个类别是正确的y是onehot，只有j是1其他0，则 $L = -\Sigma_{i=1}^{n} y_i \ln(a_i)$ $L = -y_j \ln(a_j) = -\ln(a_j)$

目标是求标量L对向量 z的导数 。

1. 由链式法则， $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} * \frac{\partial a}{\partial z}$ $\frac{\partial L}{\partial \boldsymbol{a}} = [0, 0, 0, \dots, -\frac{1}{a_j}, \dots, 0]$ $\frac{\partial L}{\partial \boldsymbol{z}} = -\frac{1}{a_j} * \frac{\partial a_j}{\partial \boldsymbol{z}}$
2. i=j $\frac{\partial a_j}{\partial z_j} = \frac{e^{z_j}\sum_k^n e^{z_k} - e^{z_j}e^{z_j}}{(\sum_k^n e^{z_k})^2} = a_j - a_j^2$ i≠j， $\frac{\partial a_j}{\partial z_i} = \frac{0 - e^{z_j}e^{z_i}}{(\sum_k^n e^{z_k})^2} = -a_j a_i$ 所以

$$\frac{\partial L}{\partial z_j} = (a_j - a_j^2) * -\frac{1}{a_j} = a_j - 1$$

$$\frac{\partial L}{\partial z_i} = -a_j a_i * -\frac{1}{a_j} = a_i$$

$$\frac{\partial L}{\partial \boldsymbol{z}} = [a_1, a_2, \dots, a_j - 1, \dots a_n] = \boldsymbol{a} - \boldsymbol{y}$$

# softmax交叉熵导数

```
# NOTE:第三组：用hard_loss+soft_loss训练student网络
soft_loss = nn.KLDivLoss()(F.log_softmax(student_logits / self.temperature),
                           F.softmax(teacher_logits / self.temperature)) * (
                    self.alpha * self.temperature * self.temperature)
```

**Softmax带T情况，看看为什么这里需要一个T^2在loss里调和**

- 影响这两个地方的导数，
- i=j　　i≠j，

$$\frac{\partial a_j}{\partial z_j} = \frac{e^{z_j}\sum_k^n e^{z_k} - e^{z_j}e^{z_j}}{(\sum_k^n e^{z_k})^2} = a_j - a_j^2$$

$$\frac{\partial L}{\partial z_j} = (a_j - a_j^2) * -\frac{1}{a_j} = a_j - 1$$

$$\frac{a_j}{T} - 1$$

$$\frac{\partial a_j}{\partial z_i} = \frac{0 - e^{z_j}e^{z_i}}{(\sum_k^n e^{z_k})^2} = -a_j a_i$$

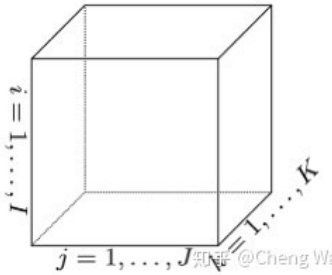
$$\frac{\partial L}{\partial z_i} = -a_j a_i * -\frac{1}{a_j} = a_i$$

$$\frac{a_i}{T}$$

- 所以 $\frac{\partial L}{\partial z} = \left[\frac{a_1}{T}, \frac{a_2}{T}, \ldots, \frac{a_j}{T} - 1, \ldots \frac{a_n}{T}\right] \cong \frac{1}{T}(\boldsymbol{a} - \boldsymbol{y})$
- Softmax Cross Entropy Loss的求导结果非常优雅，就等于预测值与Label的差
- 另外hard loss分支和softloss 在学习时需要调整下权重，DL任务很多时候需要简单推一下多目标之间的权重

# 低秩分解

## 相关概念解释



(张量，图片来源网络)

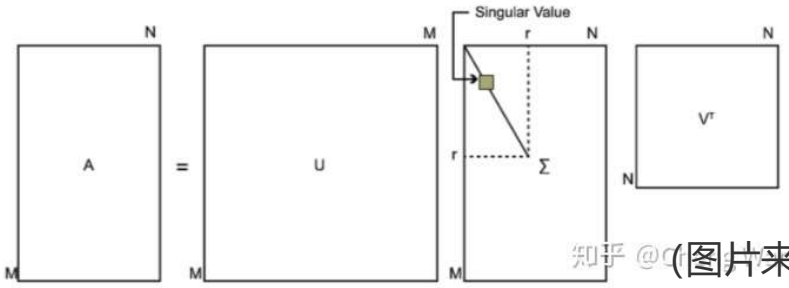
### 何为张量?

- 向量是一维，矩阵是二维，张量就代表三维及以上，或者说，我们可以将张量理解为高阶矩阵
- 三个向量外积是一个三阶张量的概念，我们引出秩一张量的概念。

### 秩一张量

- 为了大家更好的理解，我们可以先来回忆一下矩阵SVD的概念：
  - 上图是矩阵SVD分解的示意图，一个矩阵可以分解为左右奇异向量矩阵以及一个奇异值矩阵的乘积形式
  - 我们现在不妨以另外一种角度来看待矩阵的SVD分解：
  - 我们如果拿出第一个左奇异向量以及第一个右奇异向量，这两个向量做外积，我们就可以得到一个矩阵，同时这两个奇异向量对应同一个奇异值，我们尝试将奇异值理解为这两个向量外积得到的这个矩阵的在原始矩阵中所占的权重，以此类推我们就可以得到所有奇异值对应的左右奇异向量外积的结果矩阵，然后将这些矩阵加起来就得到了原始矩阵。

# SVD分解




(图片来源网络)

- **SVD用于低秩近似 Low-Rank Approximation**
  - 把SVD的计算结果可以拆写成index的形式

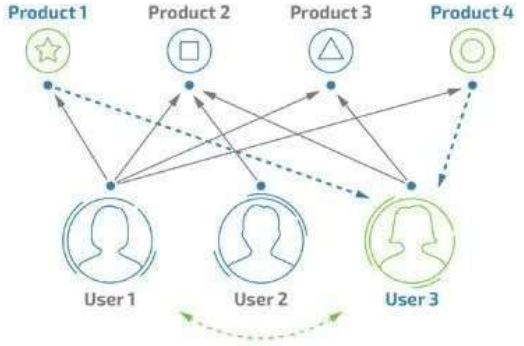
    其中 $u_j \in \mathbb{R}^{m\times 1}$ 是u的第j列， $v_j \in \mathbb{R}^{n\times 1}$ 是v的第j列。

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^T$$

- 上图是矩阵SVD分解的示意图，一个矩阵可以分解为左右奇异向量矩阵以及一个奇异值矩阵的乘积形式
- 我们现在不妨以另外一种角度来看待矩阵的SVD分解：
- 我们如果拿出第一个左奇异向量以及第一个右奇异向量，这两个向量做外积，我们就可以得到一个矩阵，同时这两个奇异向量对应同一个奇异值，我们尝试将奇异值理解为这两个向量外积得到的这个矩阵的在原始矩阵中所占的权重，以此类推我们就可以得到所有奇异值对应的左右奇异向量外积的结果矩阵，然后将这些矩阵加起来就得到了原始矩阵。每组 $u_j v_j^T \in \mathbb{R}^{m\times n}$ 都是一个秩=1的矩阵

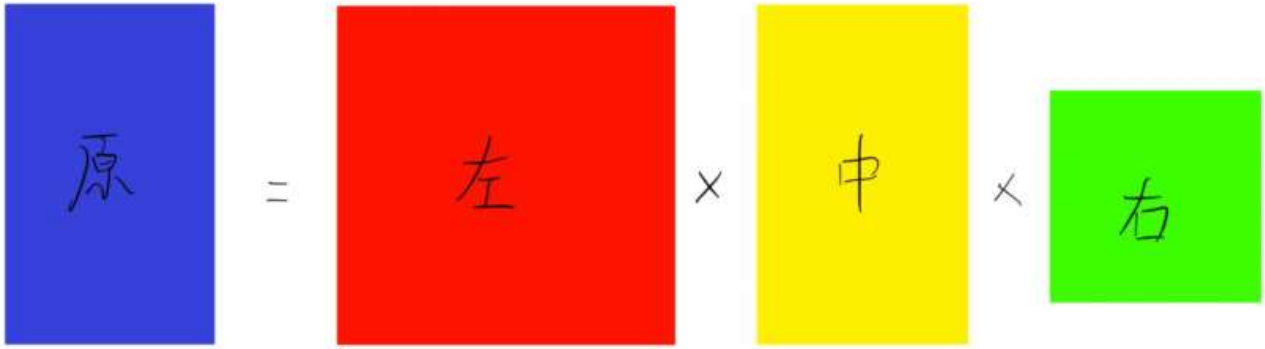
# SVD分解-协同过滤简单例子

| | 鳗鱼饭 | 日式炸鸡排 | 寿司饭 | 烤牛肉 | 三文鱼汉堡 | 鲁宾三明治 | 印度烤鸡 | 麻婆豆腐 | 宫保鸡丁 | 印度奶酪咖喱 | 俄式汉堡 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Bratt | 2 | 0 | 0 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rob | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| Drew | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 4 | 0 |
| Scott | 3 | 3 | 4 | 0 | 3 | 0 | 0 | 2 | 2 | 0 | 0 |
| Mary | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Brent | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 0 |
| Kyle | 4 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| Sara | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 4 |
| Shaney | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 0 |
| Brendan | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 | 5 | 0 |
| Leanna | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 0 | 4 | 5 | 0 |




```
u, sigma, v = svd(data)
k = select_K(sigma, 0.95)
sigmaK = np.mat(np.eye(k) * sigma[:k])

# 最后压缩之后得到的是item的矩阵，其中的每一个行向量对应一个item。
itemMat = data.T.dot(u[:,:k]).dot(sigmaK.I)
```

```
U:  (11, 4)
sigmaK:  (4, 4)
V:  (4, 11)
itemMat:  (11, 4)
```



# SVD分解

- 若向量 $a \in R^{n_1}, b \in R^{n_2}$ 则两个向量的外积 $(a \circ b = ab^T)_{n_1 \times n_2}$
- 生成一个矩阵。其中符号“ o ”表示外积运算。因此两个向量的外积运算从一维向量变为二维矩阵，增加了一个维度
- 举例说明，若 $x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, y = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ 则 $x \circ y = xy^T = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$
- 若向量 $a \in R^{n_1}, b \in R^{n_2}, c \in R^{n_3}$ ，则三个向量的外积 $a \circ b \circ c = (c * (ab^T))_{n_1 \times n_2 \times n_3}$
- 是一个三阶张量。这里的“*”运算为c中第一、二个元素与矩阵 $(ab^T)_{n_1 \times n_2}$
- 相乘得到两个面三个向量的外积的3维的张量，因此每做一次外积运算维度都增加1

秩=1的矩阵

$$X = \mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \ldots \circ \mathbf{a}^{(N)}$$

$$z = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

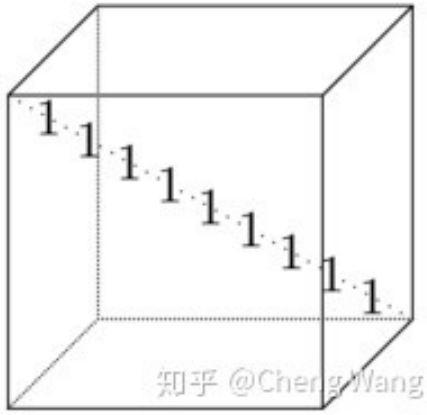
$$\mathcal{A} = x \circ y \circ z = z * (xy^T) = z * \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$\mathcal{A}(:,:,1) = 1 * \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$\mathcal{A}(:,:,2) = 0 * \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

SVD分解以后，奇异值矩阵中第一个奇异值为1，剩下的奇异值都为0。因此，奇异值为1对应的左右奇异向量的外积所得到的矩阵就是原始矩阵

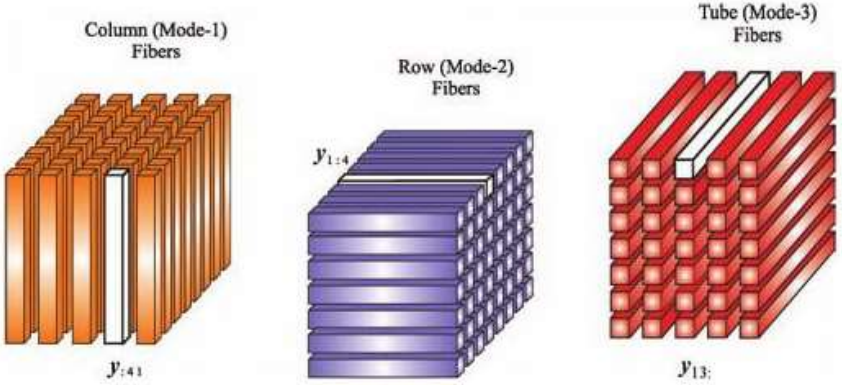
# 对角张量

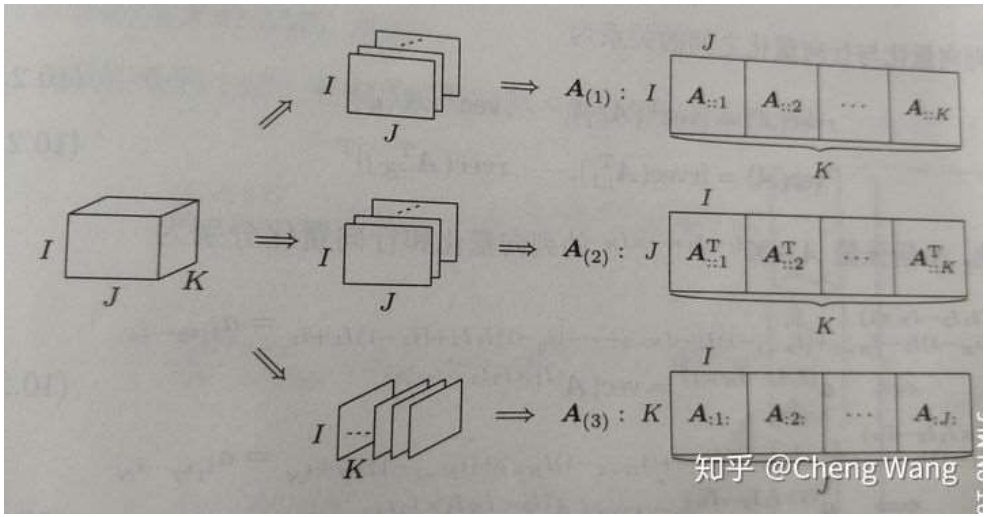



(图片来源网络)

# 张量矩阵化与向量化

- 以三阶张量，模式-n展开




(图片来源网络)

$$\mathbf{X}_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad \mathbf{X}_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}.$$

$$\mathbf{X}_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix},$$

$$\mathbf{X}_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix},$$

$$\mathbf{X}_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \cdots & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & \cdots & 21 & 22 & 23 & 24 \end{bmatrix}.$$

- mode-n product
- 在此我们仅考虑三阶张量与矩阵的乘积，由于该方法是Tucker定义的，现在常称为Tucker积。
- 三阶张量的Tucker积：考虑三阶张量 $\chi \in K^{I_1 \times I_2 \times I_3}$ 和矩阵 $A \in K^{J_1 \times I_1}$ $B \in K^{J_2 \times I_2}$ $C \in K^{J_3 \times I_3}$ 的乘积。三阶张量的Tucker模式-1积 $X \times_1 A$ ，模式-2积 $X \times_2 B$ ，模式-3积 $X \times_3 C$ 分别定义为：

M-1维度分别 $I_1 \times I_2 I_3$ 如果按照原来的顺序，这两个矩阵是无法直接相乘的，A右乘M-1。得到 $J_1 \times I_2 I_3$ 此处我们还需要将这个矩阵还原成张量 $J_1 \times I_2 \times I_3$

$$(\mathcal{X} \times_1 \boldsymbol{A})_{j_1 i_2 i_3} = \sum_{i_1=1}^{I_1} x_{i_1 i_2 i_3} a_{j_1 i_1}, \ \forall j_1, i_2, i_3$$

$$(\mathcal{X} \times_2 \boldsymbol{B})_{i_1 j_2 i_3} = \sum_{i_2=1}^{I_2} x_{i_1 i_2 i_3} b_{j_2 i_2}, \ \forall i_1, j_2, i_3$$

$$(\mathcal{X} \times_3 \boldsymbol{C})_{i_1 i_2 j_3} = \sum_{i_3=1}^{I_3} x_{i_1 i_2 i_3} c_{j_3 i_3}, \ \forall i_1, i_2, j_3$$



高维矩阵乘，实现对H，W，C方向的维度变换

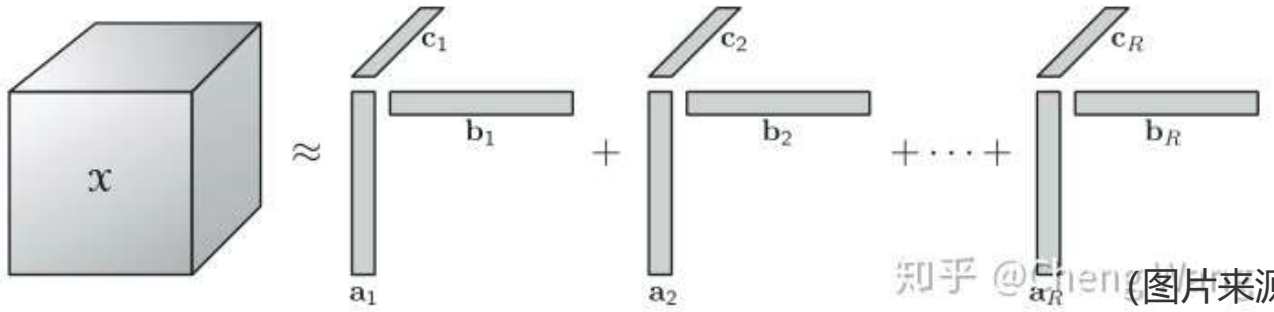
# CP分解-CANDECOMP/PARAFAC

CANDECOMP(canonical decomposition)和PARAFAC(parallel factors)

CP分解是将张量 $\chi \in R^{I \times J \times K}$ 分解为一系列的秩一张量之和。分解公式如下：

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$$

注意 $a_r, b_r, c_r$ 的维度分别是 $a_r \in R^I, b_r \in R^J, c_r \in R^K$，就是我们所说的CP秩，这个和前面SVD例子是一致。更直观的理解看下图：

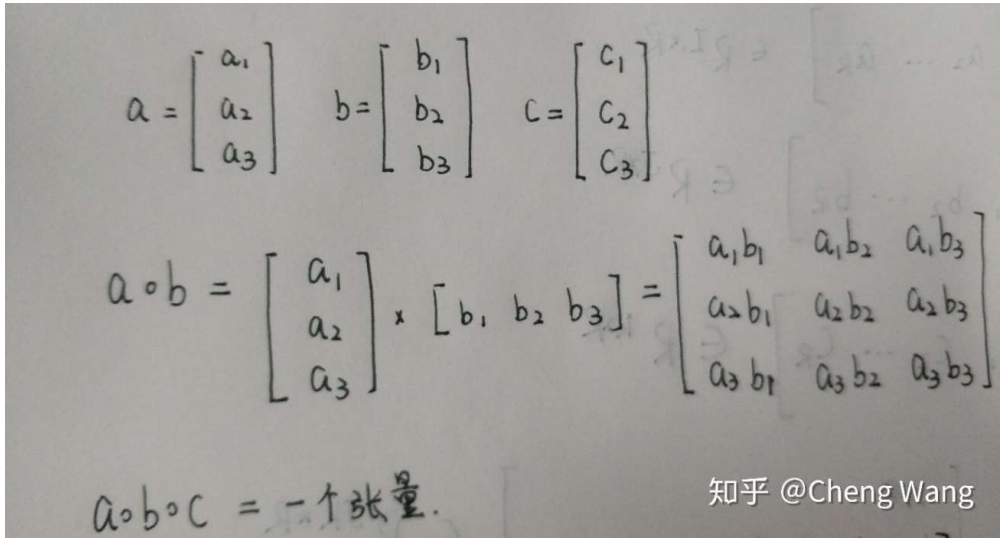


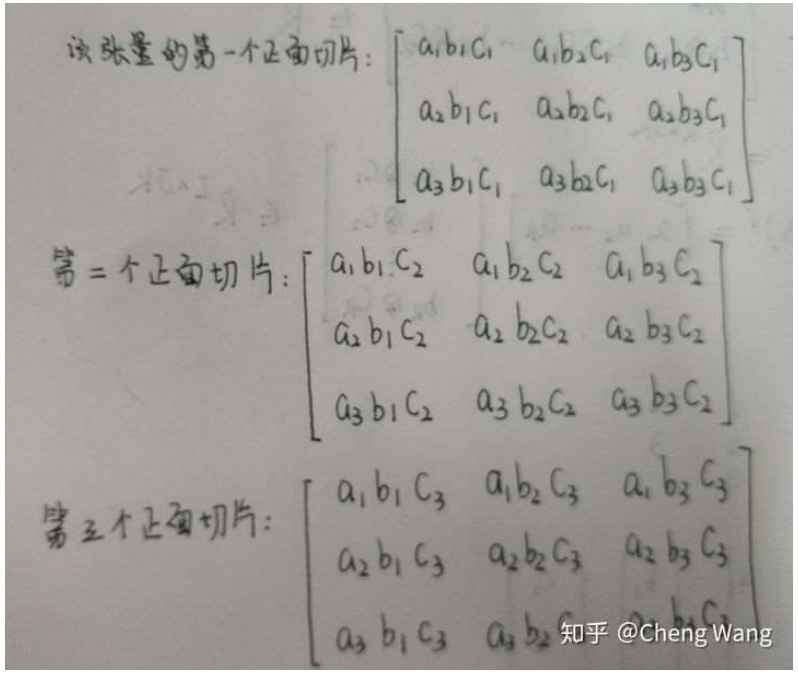



(图片来源网络)

# CP分解

- 在上图中 $a_1, b_1, c_1$ 代表三个向量，现在定义 $a_{1i}, b_{1j}, c_{1k}$ 代表这三个向量中不同位置的标量值。为了书写方便，我们将第一个下标1去掉，现在就变成了 $a_i, b_j, c_k$ 。假设 $I = J = K = 3$ 那么 $a \circ b \circ c$ 如下图所示：




**面向我们依次排放**

(图片来源网络)

# CP分解

也就是说，第一个因子张量的左上角位置元素为 $a_{11}b_{11}c_{11}$（第一下标为第一个秩1张量，第二下标表示分解向量元素），第二个因子张量左上角位置元素为 $a_{21}b_{21}c_{21}$，以此类推 $\chi_{111}=\sum_{R}^{r}a_{r1}b_{r1}c_{r1}$，我们就可以知道，张量 $\chi$ 其他位置的元素也可以用同样的方法推导。

$$x_{ijk} \approx \sum_{r=1}^{R} a_{ir}\, b_{jr}\, c_{kr} \text{ for } i = 1, \ldots, I,\ j = 1, \ldots, J,\ k = 1, \ldots, K.$$

接下来，我们介绍CP分解的另一种表示形式，在这种表示方式中，最小的单元是矩阵。

我们令 $A=[a_1,a_2,\cdots,a_R], B=[b_1,b_2,\cdots,b_R], C=[c_1,c_2,\cdots,c_R]$ ， $X_{(1)},X_{(2)},X_{(3)}$ 代表 $\chi$ 张量的模式-1，模式-2，模式-3分解，$\odot$ 代表Khatri-Rao积

那么，我们就可以定义：

$$\begin{aligned}
\mathbf{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^{\mathsf{T}}, \\
\mathbf{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^{\mathsf{T}}, \\
\mathbf{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^{\mathsf{T}}.
\end{aligned}$$

A，B，C注意类比SVD中的U，V

# CP分解

Khatri-Rao积的定义和Kronecker积定义Kronecker积也称为克罗内克积，是任意大小矩阵的运算，使用符号其表示为：若A为大小m*n的矩阵，B为大小p*q的矩阵，则A与B的克罗内克积是一个大小为mp*nq的矩阵，其表述为：

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}.$$

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$

**有规律，实际是a_mn*B的平铺**

我们这里只用到向量，如果A，B都是向量，则Kronecker积为m*p的向量，看第一列

# CP分解

- **Khatri-Rao积，照抄教科书**

**Khatri-Rao product**

The **Khatri-Rao product** of an $N \times M$ matrix $\mathbf{A}$ and a $P \times M$ matrix $\mathbf{B}$ is defined as the $NP \times M$ matrix

$$\mathbf{A} \square \mathbf{B} \triangleq \left[ \begin{array}{c|c|c|c} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_N \otimes \mathbf{b}_N \end{array} \right], \tag{A.96}$$

where $\mathbf{a}_j$ denotes the $j$th column of the matrix $\mathbf{A}$.



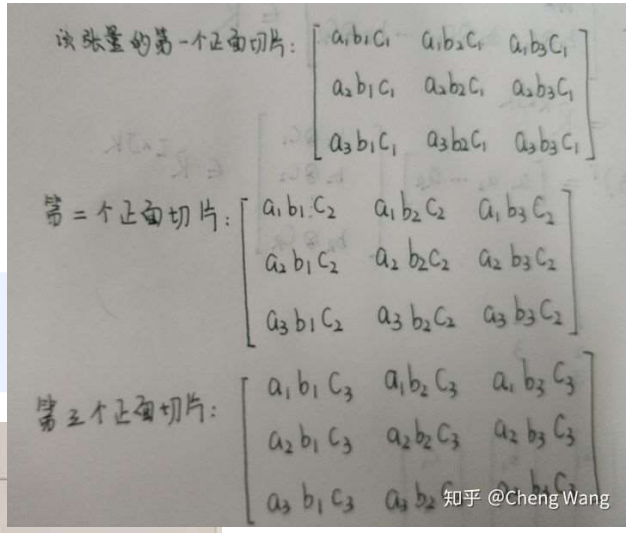
The first element is

$$\mathbf{a}_1 \otimes \mathbf{b}_1 = \left[ \begin{array}{c} a_{11}\mathbf{b}_1 \\ \vdots \\ a_{N1}\mathbf{b}_1 \end{array} \right] = \left[ \begin{array}{c} a_{11}b_{11} \\ a_{11}b_{21} \\ \vdots \\ a_{N1}b_{P-1,1} \\ a_{N1}b_{P1} \end{array} \right]. \tag{A.97}$$

简单翻译一下，
就是每一列都是N维向量ai和P维向量bi的Kronecker积

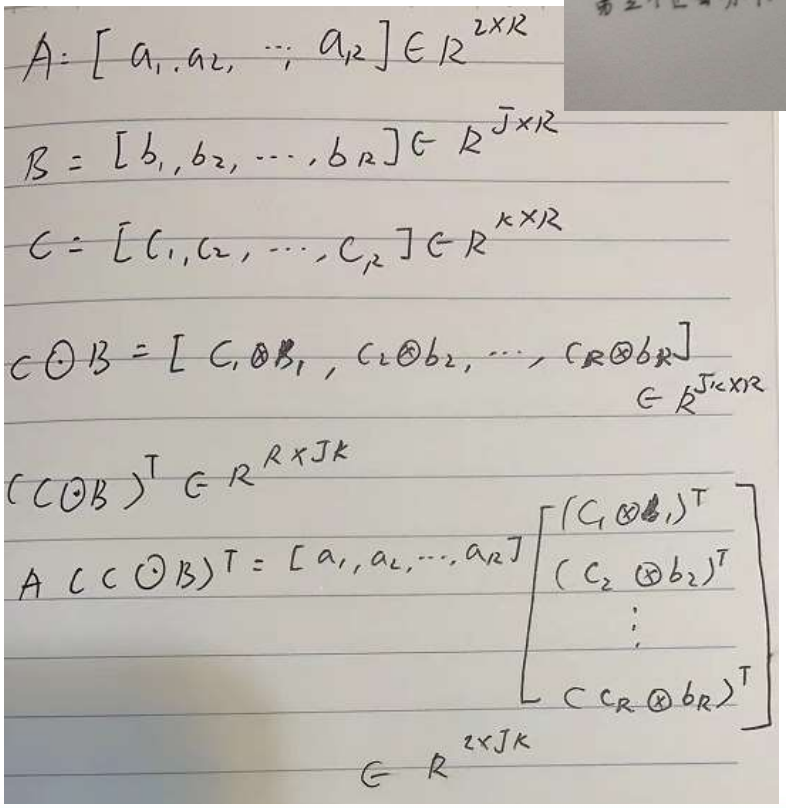
# CP分解

- 仅对 $X_{(1)}$ 进行简单推导，主要关注于等式右边计算过程中的矩阵维数变化：




（图片来源网络）

看看 $x_{111} = \sum_1^R a_{1r}b_{1r}c_{1r}$ 是否符合，
KR积的第一行的元素为 $b_{1r} * c_{1r}$
所以就是右边矩阵的第一列 ， 此处，
A的第一行表示为 $a_{1,r}$ ， 所以满足上式，
当然这只是一个简单验证其他元素类似
可以推断出来，还可以简单设置R=1，2…，如
右上图看看mode-1是怎么个规律



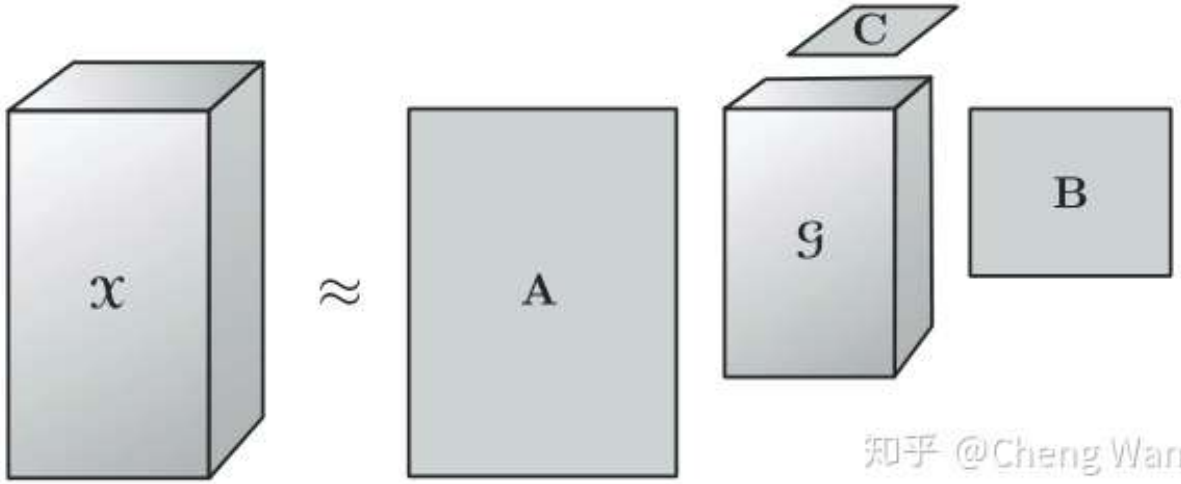
# Tucker分解

我们直接给出Tucker分解的定义，对于张量 $\chi \in R^{I\times J\times K}$ ，其Tucker分解形式如下：

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr}\, \mathbf{a}_p \circ \mathbf{b}_q \circ \mathbf{c}_r = [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!].$$

其中，$A \in R^{I\times P}, B \in R^{J\times Q}, C \in R^{K\times R}$ 是Tucker分解得到的因子矩阵，

$\mathcal{G} \in \mathbb{R}^{P\times Q\times R}$ 代表核心张量，它的元素代表不同因子矩阵之间相互作用的水平。Tucker分解示意图如下：



# Tucker分解

我们接下来再来看看张量中每一个元素的表示形式：

$$x_{ijk} \approx \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr}\, a_{ip}\, b_{jq}\, c_{kr} \quad \text{for} \quad i = 1,\ldots,I,\ \ j = 1,\ldots,J,\ \ k = 1,\ldots,K.$$

现在一起来梳理一下，（对照下面Tucker分解的公式）张量的Tucker分解本质就是将原始张量分解维一个核心张量以及对应不同维度的因子矩阵（三阶张量分解中就是三个因子矩阵），如果$P, Q, R$小于 $I, J, K$ 的话，核心张量就相当于原始张量的压缩形式，即对原始张量进行降维操作的结果。类似CP，Tucker的表示如下：还是简单推下 $X_{(1)}$ 形状

$$C \otimes B \in R^{JK \times QR}$$

$$\mathbf{X}_{(1)} \approx \mathbf{A}\mathbf{G}_{(1)}(\mathbf{C} \otimes \mathbf{B})^{\top}$$

$$(C \otimes B)^{T} \in R^{QR \times JK}$$

$$\mathbf{X}_{(2)} \approx \mathbf{B}\mathbf{G}_{(2)}(\mathbf{C} \otimes \mathbf{A})^{\top}$$

$$G_{(1)}(C \otimes B)^{T} \in R^{P \times JK}$$

$$\mathbf{X}_{(3)} \approx \mathbf{C}\mathbf{G}_{(3)}(\mathbf{B} \otimes \mathbf{A})^{\top}$$

$$AG_{(1)}(C \otimes B)^{T} \in R^{I \times JK}$$

**G(1),G(2),G(3) 分别为G的mode-n**

# Tucker分解

如果我们的核心张量是一个超对角张量的话（即，在对角张量那节介绍的对角张量的基础上约束 $P = Q = R$），那么Tucker分解就变成了CP分解。

# 全连接层的SVD

SVD分解使我们能够分解任何具有n行和m列的矩阵A：

$$A_{n\times m} = U_{n\times n} S_{n\times m} V^T_{m\times m}$$

S是一个对角矩阵，其对角线上有非负值（奇异值），并且通常被构造成奇异值按降序排列的。U和V是正交矩阵：

$$U^T U = V^T V = I$$

如果我们取最大的奇异值并将其余的归零，我们得到A的近似值：

$$\hat{A} = U_{n\times l} S_{l\times l} V^T_{m\times l}$$

$\hat{A}$ 具有作为Frobenius范数最接近于A的秩l矩阵的良好性质，所以如果l足够大，$\hat{A}$ 是A的良好近似。

$$||A||_F = \sqrt{tr(A^T A)} = \sqrt{\sum_{i,j} a_{ij}^2}$$

# 全连接层的SVD

一个全连接层通常是指矩阵乘法运算，也就是输入一个矩阵A然后增加一个偏差b：

$$Ax + b$$

我们可以取A的奇异值分解，只保留前l个奇异值。

$$\left(U_{n\times l}S_{l\times l}V^T_{m\times l}\right)x + b = U_{n\times l}\left(S_{l\times l}V^T_{m\times l}x\right) + b$$

这里我们将一个全连接层拆解成两个更好实现的结构：
第一个结果是有一个m*l的形状，没有偏置，其权重将取自 $S_{l\times l}V^T_{m\times l}$
第二个将有一个n*l的形状，设置偏置等于b，其权重将取自 $U_{n\times l}$ .

也就是权重总数从n*m下降到l*(n + m)。运算量从n*m下降为n*l+l*m（参考上一讲的fc的计算量）

# 全连接层的SVD

[py-faster-rcnn/compress_net.py at 781a917b378dbfdedb45b6a56189a31982da1b43 · rbgirshick/py-faster-rcnn (github.com)]

```
def compress_weights(W, l):
    """Compress the weight matrix W of an inner product (fully connected) layer
    using truncated SVD.
    Parameters:
    W: N x M weights matrix
    l: number of singular values to retain
    Returns:
    Ul, L: matrices such that W \approx Ul*L
    """

    # numpy doesn't seem to have a fast truncated SVD algorithm...
    # this could be faster
    U, s, V = np.linalg.svd(W, full_matrices=False)

    Ul = U[:, :l]
    sl = s[:l]
    Vl = V[:l, :]

    L = np.dot(np.diag(sl), Vl) # S*V
    return Ul, L

```

# 在卷积层上张量CP分解

二维卷积层是一个多维矩阵（张量），有四个维度：

cols x rows x input_channels x output_channels

我们将使用两种流行的的张量分解方法进行讲解：CP分解和Tucker分解（也称为高阶SVD）。

CP分解让我们将这种方法也推广到了张量上。

使用CP分解，我们的卷积核，也就是一个四维张量公式，可以近似为一个选定的R：

$$\sum_{r=1}^{R} K_r^x(i) K_r^y(j) K_r^s(s) K_r^t(t)$$

每个维度都降维到秩R的矩阵，我们希望R非常小来使分解是有效的，但是对保持近似高精度是足够大的。

# 带CP分解的卷积正向传递

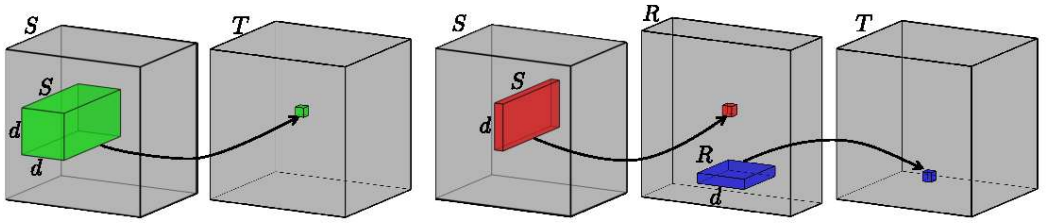
为了正向传递一个卷积层，我们给卷积层的输入为， $X(i,j,s)$

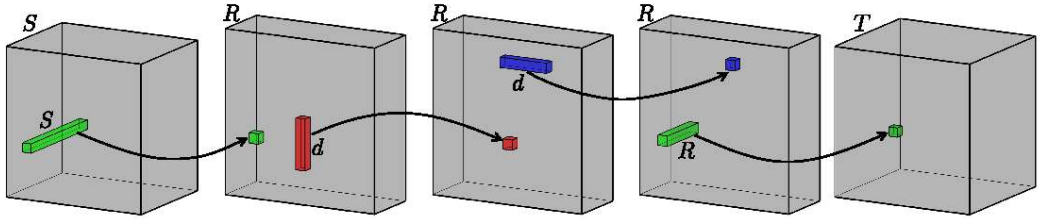
$$V(x,y,t) = \sum_i \sum_j \sum_s K(i,j,s,t) X(x-i, y-j, s)$$
$$= \sum_r \sum_i \sum_j \sum_s K_r^x(i) K_r^y(j) K_r^s(s) K_r^t(t) X(x-i, y-j, s)$$
$$= \sum_r K_r^t(t) \sum_i \sum_j K_r^x(i) K_r^y(j) \sum_s K_r^s(s) X(x-i, y-j, s)$$

我们可以使用下述方法：

- 首先做一个核为 $K_r(s)$ 的point wise卷积（1x1xS）。这可以把输入通道的数量从S降到R。下一步可以在较少数量的通道上完成卷积，实现更快的速度。
- 其次用 $K_r^x, K_r^y$ 在空间维度上执行分离的卷积。就像在mobilenet中一样，卷积是深度可分的，分别在每个通道中完成。与mobilenets不同，这里的卷积在空间维度上也是可分的。
- 最后做另一个point wise卷积来改变通道数量从R降到T，如果原始卷积层有偏置的话，在这一步的时候加上这些偏置。

# 带CP分解的卷积正向传递




```
conv2d layer shape:  (64, 32, 3, 3)
rank:  21
first shape:  (32, 21)
vertical shape:  (3, 21)
horizontal shape:  (3, 21)
last shape:  (64, 21)
pointwise_s_to_r_layer shape:  (21, 32, 1, 1)
depthwise_vertical_layer shape:  (21, 1, 3, 1)
depthwise_horizontal_layer shape:  (21, 1, 1, 3)
pointwise_r_to_t_layer shape:  (64, 21, 1, 1)
```

参数量：3*3*32*64=18432
21*32+21*3*2+64*21= 2142

计算量：N是输出的ftm的numel()
N*3*3*32*64=18432*N
N*(21*32+21*3*2+64*21)= 2142*N

```python
# Perform CP decomposition on the layer weight tensorly.
# last, first, vertical, horizontal = \
weight, (last, first, vertical, horizontal) = \
    parafac(layer.weight.data.numpy(), rank=rank, init='svd')

# print(weight)

print('conv2d layer shape: ',layer.weight.data.numpy().shape)
print('rank: ', rank)

print('first shape: ',first.shape)
print('vertical shape: ',vertical.shape)
print('horizontal shape: ',horizontal.shape)
print('last shape: ',last.shape)

pointwise_s_to_r_layer = torch.nn.Conv2d(in_channels=first.shape[0], \
        out_channels=first.shape[1], kernel_size=1, stride=1, padding=0,
        dilation=layer.dilation, bias=False)

depthwise_vertical_layer = torch.nn.Conv2d(in_channels=vertical.shape[1],
        out_channels=vertical.shape[1], kernel_size=(vertical.shape[0], 1),
        stride=1, padding=(layer.padding[0], 0), dilation=layer.dilation,
        groups=vertical.shape[1], bias=False)

depthwise_horizontal_layer = \
    torch.nn.Conv2d(in_channels=horizontal.shape[1], \
        out_channels=horizontal.shape[1],
        kernel_size=(1, horizontal.shape[0]), stride=layer.stride,
        padding=(0, layer.padding[0]),
        dilation=layer.dilation, groups=horizontal.shape[1], bias=False)

pointwise_r_to_t_layer = torch.nn.Conv2d(in_channels=last.shape[1], \
        out_channels=last.shape[0], kernel_size=1, stride=1,
        padding=0, dilation=layer.dilation, bias=True)
```

# 在卷积层上张量Tucker分解

Tucker分解也称为高阶奇异值分解（HOSVD high order）或者其他名称，是对张量进行奇异值分解的一种特殊形式

$$K(i,j,s,t)=\sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4}\sigma_{r_1r_2r_3r_4}K_{r1}^{x}(i)K_{r2}^{y}(j)K_{r3}^{s}(s)K_{r4}^{t}(t)$$

高阶SVD的原因是在HOSVD中 的分量必须是正交的，$\sigma_{r_1r_2r_3r_4}$ 是关键参数，被通常称为核心矩阵，用于定义不同的轴之相互作用关系。

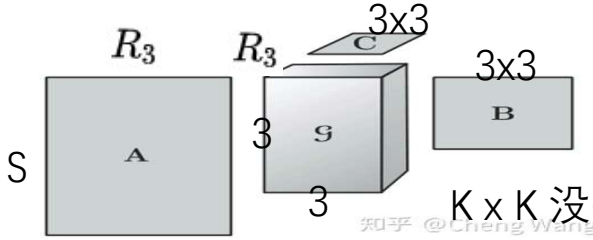
在上面描述的CP分解中，沿着空间维度 $K_r^x(i)K_r^y(j)$ 的分解可以利用空间可分离的卷积进行运算。这些filter通常是非常小尺寸的，通常是3x3或5x5，所以可分离的卷积并不能真正节省很大的计算量。

Trucker分解有用的性质是，它不必沿着所有的轴（模）来分解。我们可以沿着输入和输出通道进行分解：

$$K(i,j,s,t)=\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4}\sigma_{ijr_3r_4}(j)K_{r3}^{s}(s)K_{r4}^{t}(t)$$

# 在卷积层上张量Tucker分解

像CP分解一样，将分解方法插入到卷积公式




K x K 没有缩减维度

$$V(x,y,t)=\sum_i\sum_j\sum_s K(i,j,s,t)X(x-i,y-j,s)$$

$$V(x,y,t)=\sum_i\sum_j\sum_s\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4}\sigma_{(i)(j)r_3r_4}K_{r3}^s(s)K_{r4}^t(t)X(x-i,y-j,s)$$

$$V(x,y,t)=\sum_i\sum_j\sum_{r_4=1}^{R_4}\sum_{r_3=1}^{R_3}K_{r4}^t(t)\sigma_{(i)(j)r_3r_4}\sum_s K_{r3}^s(s)X(x-i,y-j,s)$$

我们可以发现用Tucker分解进行卷积的方法如下：

- 与 $K_{r3}^s(s)$ 进行point wise 卷积，通道的数量从S减少到 $R_3$
- 用 $\sigma_{(i)(j)r_3r_4}$ 进行常规的（非可分离的）卷积。这个卷积代替了原始层的S输入通道和T输出通道，替换成 $R_3$ 输入通道和 $R_4$ 输出通道。如果这些矩阵的秩小于S和T，这就是减少的原因。
- 用 $K_{r4}^t(t)$ 进行point wise卷积以回复原始卷积的T个输出通道。由于这是最后一次卷积，所以如果有偏置就在这个时候加上偏置。

# 在卷积层上张量Tucker分解

参数量：3*3*64*64=36864
64*29+64*29*3*3+64*29= 20416

计算量：N是输出的ftm的numel()
N*3*3*64*64=36864*N
N*(64*29*2+64*29*3*3)= 20416*N

```
print('conv2d layer shape: ',layer.weight.data.numpy().shape)
ranks = estimate_ranks(layer)
print(layer, "VBMF Estimated ranks", ranks)
core, [last, first] = \
    partial_tucker(layer.weight.data.numpy(), \
        modes=[0, 1], rank=ranks, init='svd')

print('first shape: ',first.shape)
print('core shape: ',core.shape)
print('last shape: ',last.shape)

# A pointwise convolution that reduces the channels from S to R3
first_layer = torch.nn.Conv2d(in_channels=first.shape[0], \
        out_channels=first.shape[1], kernel_size=1,
        stride=1, padding=0, dilation=layer.dilation, bias=False)

# A regular 2D convolution layer with R3 input channels
# and R3 output channels
core_layer = torch.nn.Conv2d(in_channels=core.shape[1], \
        out_channels=core.shape[0], kernel_size=layer.kernel_size,
        stride=layer.stride, padding=layer.padding, dilation=layer.dilation,
        bias=False)

# A pointwise convolution that increases the channels from R4 to T
last_layer = torch.nn.Conv2d(in_channels=last.shape[1], \
    out_channels=last.shape[0], kernel_size=1, stride=1,
    padding=0, dilation=layer.dilation, bias=True)
```

```
conv2d layer shape:  (64, 3, 3, 3)
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) VBMF Estimated ranks [14, 2]
first shape:  (3, 2)
core shape:  (14, 2, 3, 3)
last shape:  (64, 14)
first_layer shape:  (2, 3, 1, 1)
core_layer shape:  (14, 2, 3, 3)
last_layer shape:  (64, 14, 1, 1)
###################################################################
conv2d layer shape:  (64, 64, 3, 3)
Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) VBMF Estimated ranks [26, 29]
first shape:  (64, 29)
core shape:  (26, 29, 3, 3)
last shape:  (64, 26)
first_layer shape:  (29, 64, 1, 1)
core_layer shape:  (26, 29, 3, 3)
last_layer shape:  (64, 26, 1, 1)
###################################################################
conv2d layer shape:  (128, 64, 3, 3)
Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) VBMF Estimated ranks [39, 32]
first shape:  (64, 32)
core shape:  (39, 32, 3, 3)
last shape:  (128, 39)
first_layer shape:  (32, 64, 1, 1)
core_layer shape:  (39, 32, 3, 3)
last_layer shape:  (128, 39, 1, 1)
```

# 感谢 THANKS FOR WATCHING
# 您的观看

讲师：