神经网络基础

1 模型训练&微调相关

预训练+微调的模型应用流程使得我们有机会只花费很小的算力就能获得一个完美适配下游任务的模型。pytorch官方、HuggingFace、modelscope、paddlepaddle社区等处均有提供大量预训练模型供大家下载使用。

1.1 预训练和微调任务有什么区别？两者的目的是什么？

- 数据集大小：预训练使用的数据集通常非常大，而微调使用的数据集相对较小。
- 训练目标：预训练旨在学习通用的语言特征，减少对大量标注数据的依赖。通过预训练，可以在没有大量标注数据的情况下训练出强大的模型，而微调则专注于学习特定任务的特征。
- 训练阶段：预训练是模型训练的第一阶段，微调是在预训练模型的基础上进行的第二阶段。
- 模型参数：在预训练阶段，模型的所有参数都被训练；在微调阶段，可能只有部分参数或新添加的参数被训练。

1.2 使用预训练模型的好处

利用训练好的 SOTA 模型权重去做特征提取，只需在小数据集上对整个模型或模型的最后几层做微调就能获得效果不错的模型，可以节省我们训练模型和调参的时间。

1.3 什么是微调（fine-tune）

指稍微调整参数即可得到优秀的性能，是迁移学习的一种实现方式。

微调和从头训练的本质区别在于模型参数的初始化，train from scratch 通常指对网络各类参数进行随机初始化（当然随机初始化也存在一定技巧），随机初始化模型通常不具有任何预测能力，通常需要大量的数据或者特定域的数据进行从零开始的训练，这样需要训练到优秀的模型通常是稍困难的。而微调的网络，网络各类参数已经在其他数据集（例如 ImageNet 数据集）完成较好调整的，具备了较优秀的表达能力。因此，我们只需要以较小的学习速率在自己所需的数据集领域进行学习即可得到较为优秀的模型。

通常情况下，我们无须再重新设计网络结构，预训练模型提供了优秀的结构，只需稍微修改部分层即可（通常为最前面几层或最后面几层）。在小数据集上，通常微调的效果比从头训练要好很多，原因在于数据量较小的前提下，训练更多参数容易导致过度拟合。

1.4 为什么只微调最后几层

1. 模型 中更靠近底部的层（定义模型时先添加到模型中的层）编码的是更加通用的可复用特征，而更靠近顶部的层（最后添加到模型中的层）编码的是更专业化的特征。微调这些更专业化的特征更加有用，它更代表了新数据集上的有用特征。

2. 训练的参数越多，过拟合的风险越大。很多 SOTA 模型拥有超过千万的参数，在一个不大的数据集上训练这么多参数是有过拟合风险的，除非你的数据集像 Imagenet 那样大。

通常情况下，CNN/bert等模型的最后一层都是用于将hidden state映射为输出维度的linear层，只微调该输出层一般也能获得不错的效果。（对于大模型，微调最前面的embedding层有时候也能获得不错的结果）

1.5 微调有哪些不同方法？

- 微调最后一层。我们利用预训练模型本身的特征提取能力来获取输入数据的特征向量，只训练一层全连接层来适配下游任务。
- 全量微调。预训练模型和下游任务的差异大到仅训练最后一层无法满足任务需求，此时只能对整个模型的所有参数进行微调。在选取合适的预训练模型前提下，通常使用较少的数据量以很小的学习率微调几轮模型就能获得满意的结果。
- 模型参数量过大，无法进行全量微调：
  - 仅微调bias
  - 参考prompt-tuning等方法，微调embedding层
  - 参考lora等方法，进行低秩分解
  - ...

1.6 batchsize的一些抽象见解(仅供参考)

batchsize越大，模型更新一次参数所能“看见”的视野就越宏观，参数优化的方向就相对更明确。但是如果过大，模型所能”看见“的微观细节就越少，优化的方向就越模糊。

batchsize越小，模型所能“看见”的视野越小，细节越多，损失下降的方向就更具针对性，但是也可能会导致损失的波动就越大，捡了芝麻丢了西瓜。

一般经验来讲，设置一个平庸的bs，譬如32/64/128等，基本可以保证正常训练；但也可以选择在训练前期设置一个大的bs，搭配较大的lr，先朝着宏观正确的方向训练一段时间（warm up），再调小bs和lr，精细调整，可能会有更好的效果。

1.7 深度网络模型中权重能不能都初始化为 0

- 如果所有权重都初始化为 0，那么在前向传播和反向传播的过程中，每一层的所有神经元将接收到相同的输入和梯度更新，导致它们更新后的权重依然相同。这种情况下，无论网络有多少层或多少神经元，每一层的所有神经元都会执行相同的操作，相当于网络没有多个神经元的学习能力。必须用随机值初始化权重，以确保神经网络能够学习到更丰富的特征。
- 激活函数的非线性：大多数神经网络中使用的激活函数都是非线性的。如果权重初始化为 0，那么无论输入数据如何，激活函数的输入总是 0，这将使得激活函数的输出在一开始是固定值，从而无法通过梯度下降进行有效的学习。

1.8 怎么解决数据集中正负样本不均衡的现象

- 过采样少数类，增加少数类的样本数量，可以通过简单地复制现有样本或使用更复杂的技术。
- 欠采样多数类：减少多数类的样本数量，以减少类别不平衡的影响。
- 调整权重：在损失函数中为不同类别的样本分配不同的权重，使得模型更关注少数类样本。FocalLoss：专为解决类别不平衡设计的损失函数，它降低了易分类样本的权重，并增加了难分类样本的权重。
- Bagging：使用集成方法，如随机森林，可以提高模型对少数类样本的预测能力。Boosting：一些提升方法，如AdaBoost，可以通过在训练过程中增加对少数类样本的关注来解决不平衡问题。
- 调整分类阈值，使得模型对少数类样本更加敏感。

注：以上方法都要谨慎使用。不管是过采样少数类还是欠采样多数类，都有可能会改变数据的分布。而其他让模型更关注少数类的方法，可能会导致过拟合，使得模型在测试集中的表现不佳。也有人会尝试构造少数类新样本，这也同样可能改变原始数据分布。

1.9 超参数

1.9.1 神经网络中包含哪些超参数?

通常可以将超参数分为三类：网络参数、优化参数、正则化参数。

- 网络参数：可指网络层与层之间的交互方式（相加、相乘或者串接等）、卷积核数量和卷积核尺寸、网络层数（也称深度）和激活函数等。
- 优化参数：一般指学习率、批样本数量（batch size）、不同优化器的参数以及部分损失函数的可调参数。
- 正则化：权重衰减系数，丢弃比率（dropout）

1.9.2 为什么要进行超参数调优?

本质上，这是模型优化寻找最优解和正则项之间的关系。

网络模型优化调整的目的是为了寻找到全局最优解（或者相比更好的局部最优解），而正则项又希望模型尽量拟合到最优（泛用性、鲁棒性等方面的最优）。

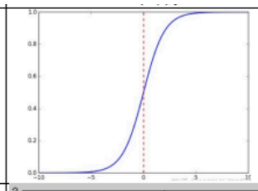
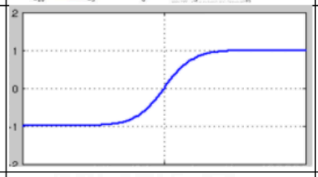
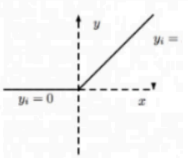
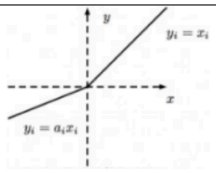
两者通常情况下，存在一定的对立，但两者的目标是一致的，即最小化期望风险。模型优化希望最小化经验风险，而容易陷入过拟合，正则项用来约束模型复杂度。

所以如何平衡两者之间的关系，得到最优或者较优的解就是超参数调整优化的目的。

1.9.3 通常需要调整哪些超参数?

- 学习率：可以说是最重要的超参之一，用于控制模型训练时网络梯度更新的量级
- 损失函数相关超参：
  - 损失函数本身的超参：例如对比学习infoNCE中的温度超参τ等，一般论文中会给出特定的建议值
  - 同时优化多目标时，不同目标的加权系数
  - 辅助损失
- batchsize：参考前文batchsize相关。如果网络中存在batchnorm，过小收敛很慢，甚至不收敛，因为数据样本越少，统计量越不具代表性；过大可能会使得梯度方向基本稳定，容易陷入局部最优解。通常在[1,1024]之间，大家习惯性设置为2的幂次。
- momentum：sgd的超参，见后文adam&sgd相关内容。通常在想要极致优化模型时才会考虑使用带动量的sgd，精调momentum。
- adam相关超参、权重衰减系数、dropout比率、网络参数：在非科研活动中通常不会考虑调整，甚至建议不要动这些超参。

2 激活函数

Sigmoid	$\sigma(x)=\frac{1}{1+e^{-x}}$ $\sigma'(x)=\sigma(x)[1-\sigma(x)]$		1. Sigmoid函数的输出在(0,1)之间，输出范围有限，优化稳定，可以用作输出层。 2. 连续函数，便于求导。	1. 会有梯度弥散 2. 不是关于原点对称 3. 计算exp比较耗时
Tanh	$a=tanh(z)=\frac{e^z-e^{-z}}{e^z+e^{-z}}$ $tanh(x)=2sigmoid(2x)-1$		1. 解决了原点对称问题 2. **比sigmoid更快**	具有软饱和性，从而造成**梯度消失**，在两边一样有趋近于 0 的情况。
ReLU	$f(x)=\begin{cases}x, if x\geq 0\\0, if x<0\end{cases}$ $f(x)=max(0,x)$		1. 收敛速度比 s型 和 tanh 快 2. 在x>0区域上，不会出现梯度饱和、梯度消失的问题。 3. 计算复杂度低，只要一个阈值就可以得到激活值。	梯度消失**没有完全解决**，在x<0时，梯度为0，相当于神经元死亡而且不会复活
Leaky ReLU	ai是（1，+∞）区间内的固定参数 $y_i=\begin{cases}x_i & \text{if } x_i\geq 0\\ \frac{x_i}{a_i} & \text{if } x_i<0,\end{cases}$		解决了神经死亡问题；	

2.1 为什么需要激活函数？

激活函数是用来加入非线性因素的，因为线性模型的表达能力不够，多层不带激活函数的线性变换等价于单层线性变换。

2.2 为什么 ReLU 常用于神经网络的激活函数？

1. 在前向传播和反向传播过程中，ReLU 相比于 Sigmoid 等激活函数计算量小；
2. 避免梯度消失问题。对于深层网络，Sigmoid 函数反向传播时，很容易就会出现梯度消失问题（在 Sigmoid 接近饱和区时，变换太缓慢，导数趋于 0，这种情况会造成信息丢失），从而无法完成深层网络的训练。
3. 可以缓解过拟合问题的发生。Relu 会使一部分神经元的输出为 0，这样就造成了网络的稀疏性，并且减少了参数的相互依存关系，缓解了过拟合问题的发生。
4. 相比 Sigmoid 型函数，ReLU 函数有助于随机梯度下降方法收敛。

3 梯度消失&梯度爆炸

梯度消失：靠近输出层的 hidden layer 梯度大，参数更新快，所以很快就会收敛；而靠近输入层的 hidden layer 梯度小，参数更新慢，几乎就和初始状态一样，随机分布。

另一种解释：当反向传播进行很多层的时候，由于每一层都对前一层梯度乘以了一个小数，因此越往前传递，梯度就会越小，训练越慢。

梯度爆炸：前面 layer 的梯度通过训练变大，而后面 layer 的梯度指数级增大。

3.1 梯度爆炸引发的问题？

- 在深度多层感知机(MLP)网络中，梯度爆炸会引起网络不稳定，最好的结果是无法从训练数据中学习，而最坏的结果是出现无法再更新的 NaN 权重值。
- 在 RNN 中，梯度爆炸会导致网络不稳定，无法利用训练数据学习，最好的结果是网络无法学习长的输入序列数据。

3.2 如何确定出现梯度爆炸

- 模型不稳定，导致更新过程中的损失出现显著变化；
- 训练过程中模型权重变成 NaN 值；
- 训练过程中模型梯度快速变大；
- 训练过程中，每个节点和层的误差梯度值持续超过 1.0。

（通常观察到损失从零点几快速增大到1以上甚至几十几百时，或者损失变为nan，代表出现了梯度爆炸；当损失变为0或一个不合理的极小值时，代表出现了梯度消失。需要立即停止训练）

3.3 解决方案

- 换用relu/leakyrelu：让激活函数的导数为1
- BatchNormalization：BN层将每个隐层神经元，把逐渐向非线性函数映射后向取值区间极限饱和区靠拢的输入分布强制拉回到均值为0方差为1的比较标准的正态分布，使得非线性变换函数的输入值落入对输入比较敏感的区域，以此避免梯度消失问题。
- 添加残差结构
- LSTM：通过门控机制改善RNN结构的梯度消失问题
- 逐层预训练+整体微调

4 正则化

正则化的大致目标是通过增加模型结构或训练数据的复杂度来提高模型的鲁棒性，获得更优更稳定的结果

4.1 L1正则化（Lasso） & L2正则化（Ridge）

见西瓜书结构风险最小化相关内容，鼠鼠还没见西瓜书以外的地方用到过

4.2 dropout

背景：如果模型的参数太多，数据量又太小，则容易产生过拟合。为了解决过拟合，就需要利用**集成学习**的思想，同时训练多个网络，然后多个网络取均值。

介绍：训练时，让某个神经元的激活值以一定的概率 P 停止工作，这样可以使模型的泛化性更强。Dropout 效果跟 bagging 效果类似（bagging 是减少方差 variance，而 boosting 是减少偏差 bias）。加入 dropout 会使神经网络训练时间长，模型预测时不需dropout，需要手动设置`model.eval()`关闭。

为啥它能解决过拟合：减少了神经元之间复杂的共适应关系，迫使网络更加鲁棒

训练和预测时的区别：训练时随机删除一些神经元，在测试模型时将所有的神经元加入

4.3 批量归一化 & 层归一化

BatchNorm：计算训练阶段 mini_batch 数量激活函数前结果的均值和方差，然后对其进行归一化，最后对其进行缩放和平移。

作用：调整了数据的分布，不考虑激活函数，它让每一层的输出归一化到了均值为 0 方差为 1 的分布，这保证了梯度的有效性，可以解决反向传播过程中的梯度问题。

训练时：是对每一个 batch 的训练数据进行归一化，也即是用每一批数据的均值和方差。

测试时：都是对单个样本进行测试。这个时候的均值和方差是全部训练数据的均值和方差，这两个数值是通过移动平均法求得，是一个确定值。

- BN 的优点：可以解决“Internal Covariate Shift”；解决梯度消失的问题（针对 sigmoid），加快收敛速度
- BN 的缺点：batch size 小的时候估算的统计值是不合理的；不适用于 RNN，因为 RNN 的输入是动态，长度不一致，导致计算的估值不合理.

LayerNorm：与batchnorm类似，但是是对每个样本的所有特征统一做归一化。

- LN 的优点：不依赖于 batch size；适用于 RNN
- LN 的缺点：不适用于 CNN；针对具有多个连续特征的数据，进行特征之间的缩放，可能会导致量纲差异消失（故而适合文本序列这种没有量纲差异的数据）。

  注：BN 通常放在激活函数之前，这是因为它能够更有效地稳定训练过程。由于激活函数可能会产生非线性变换，这可能会使得归一化的效果不如在激活前进行的那么有效。

4.4 数据增强

通过对现有数据进行各种变换和处理来生成新的训练样本，从而增加数据集的多样性和数量。

常见的数据增强有：图像裁剪，图像旋转等（见pytorch transform相关api）；子图抽取，构建超图；...

4.5 早停法

通过提前停止训练来避免过拟合。实际中很难通过loss/acc等指标判断何时停止训练，所以一般是通过经常保存checkpoint来实现早停法。

5 梯度更新（Optimizer）

实际中SGD和Adam使用更频繁。

5.1 一阶优化&二阶优化

一阶优化方法：SGD -> SGDM ->NAG -> AdaGrad -> AdaDelta / RMSProp(加速梯度下降) -> Adam -> Nadam

二阶优化方法：牛顿法、拟牛顿法（如BFGS）

二阶优化方法需要计算 Hessian 矩阵的逆，计算量大，在深度学习中并不常用。因此一般使用的是一阶梯度优化。

5.2 Adam

> ” 如果你不知道用什么优化器，那就用Adam。——沐神 “

Adam = Adaptive + Momentum。它用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率，在经过偏置的校正后，每一次迭代后的学习率都有个确定的范围，使得参数较为平稳。

（形式上类似于一阶指数平滑，并尝试消除序列最前面的几个数据平滑效果差的缺点）

5.3 SGD

动量：动量累加了历史梯度更新方向，所以在每次更新时，要是当前时刻的梯度与历史时刻梯度方向相似，这种趋势在当前时刻则会加强；要是不同，则当前时刻的梯度方向减弱。动量方法限制了梯度更新方向的随机性，使其沿正确方向进行。

一般先用Adam快速下降，然后用精细调参的SGD（Momentum）取得更好的最终结果。

5.4 反向传播

反向传播的目的是更新模型参数，利用损失值 loss 对参数的导数（可以自行了解计算图相关知识），并沿着负梯度方向进行更新。

> 模型的参数通常存在于全连接层 $wx+b$ 中的权重w和偏置b、卷积层的卷积核、注意力机制qkv的三（或3*num_heads）个矩阵、亦或自己通过`nn.Parameter`定义的，参与模型前向计算的矩阵或张量等。

反向传播公式推导：太占地方了就没写，听说有些地方会考察，建议有时间还是熟悉下

6 损失函数

6.1 0-1损失

$$L(y,f(x))=\begin{cases}1,y\neq f(x)\\0,y=f(x)\end{cases}$$

6.2 感知机损失

$$L(y,f(x))=\max(0,-yf(x))$$

6.3 Hinge损失

$$L(y,f(x))=\max(0,1-yf(x))$$

- 目标值y（-1或+1），$f(x)$是分类器输出的预测值，当y和$f(x)$的符号相同（表示预测正确）并且$|f(x)|\geq 1$时，损失为0；当y和$f(x)$符号相反时，损失随着$f(x)$的增大线性增大
- 用于支持向量机

6.4 平方损失

$$L(y,f(x))=\frac{1}{2}\|f(x)-y\|^2$$

- L2损失
- 目标和预测之间的欧氏距离
- 用于回归分析

6.5 绝对损失

$$L(y, f(x)) = |y - f(x)|$$

- L1损失
- 差距不会被平方缩放

6.6 交叉熵损失

$$L(y, f(x)) = -\sum_i y_i \log \hat{y}_i$$

- 用于分类问题
- 最小化交叉熵损失相当于最小化预测分布和标签分布之间的KL散度（KL divergence）
  - KL散度也称为相对熵（Relative Entropy）

$$\begin{aligned} D_{KL}(P\|Q) &= \sum_{x\in\mathcal{X}} P(x)\log(\frac{P(x)}{Q(x)}) \\ &= \sum_{x\in\mathcal{X}} P(x)\log P(x) - \sum_{x\in\mathcal{X}} P(x)\log Q(x) \\ &= P(x)\text{的熵} + P(x)\text{和}Q(x)\text{的交叉熵} \end{aligned}$$

6.7 为什么用交叉熵

在机器学习中，我们需要评估标签值 y 和预测值 $\hat{y}$ 之间的差距（即两个概率分布之间的相似性），使用 KL 散度 $D_{KL}(y\|\hat{y})$ 即可，但因为样本标签值的分布通常是固定的，即 $P(y)$ 不变。因此，为了计算方便，在优化过程中，只需要关注交叉熵就可以了。所以，在机器学习中一般直接用交叉熵做损失函数来评估模型。

6.8 为什么不能使用均方差做为分类问题的损失函数？

回归问题通常用均方差损失函数，可以保证损失函数是个凸函数，即可以得到最优解。而分类问题如果用均方差的话，损失函数的表现不是凸函数，就很难得到最优解。而交叉熵函数可以保证区间内单调。

分类问题的最后一层网络，需要分类函数，Sigmoid 或者 Softmax，如果再接均方差函数的话，其求导结果复杂，运算量比较大。用交叉熵函数的话，可以得到比较简单的计算结果，一个简单的减法就可以得到反向误差。

使用 MSE 损失函数会导致对类别之间的差异进行了平方和处理，存在无界输出，例如预测房价或股票价格。不符合分类问题的特性。且会有**梯度消失**的问题。

卷积神经网络

找个视频或动图更好理解

1 三个特点

1. 局部连接：相邻层中神经元之间的连接是局部的，这样可以有效地减少整个网络的参数数量。
2. 权值共享：卷积神经网络中每一个卷积核都可以被用于处理整张输入图片中所有的位置。
3. 池化操作：在每个卷积层之后加入池化层，可以对数据进行下采样，进而降低数据量和计算复杂度。

2 网络设计中，为什么卷积核设计尺寸都是奇数？

1. 保证像素点中心位置，避免位置信息偏移

2. 填充边缘时能保证两边都能填充，原矩阵依然对称

3 卷积层有哪些基本参数？

1. 卷积核大小 (Kernel Size)：
   定义了卷积的感受野 在过去常设为 5，如 LeNet-5；现在多设为 3，通过堆叠 3×33×3 的卷积核来达到更大的感受域。

2. 卷积核步长 (Stride)：
   常见设置为 1，可以覆盖所有相邻位置特征的组合；当设置为更大值时相当于对特征组合降采样。

3. 填充方式 (Padding)

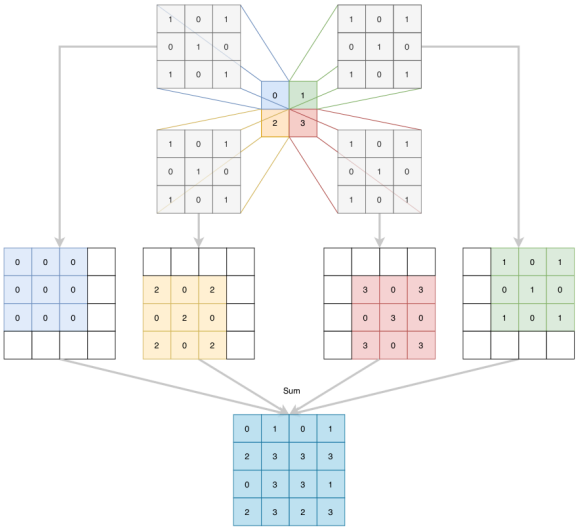
4. 输入通道数 ： 指定卷积操作时卷积核的深度

5. 输出通道数 ： 指定卷积核的个数

6. 感受野：CNN 每一层输出的特征图上的像素点在原始图像上映射的区域大小。

4 各种卷积

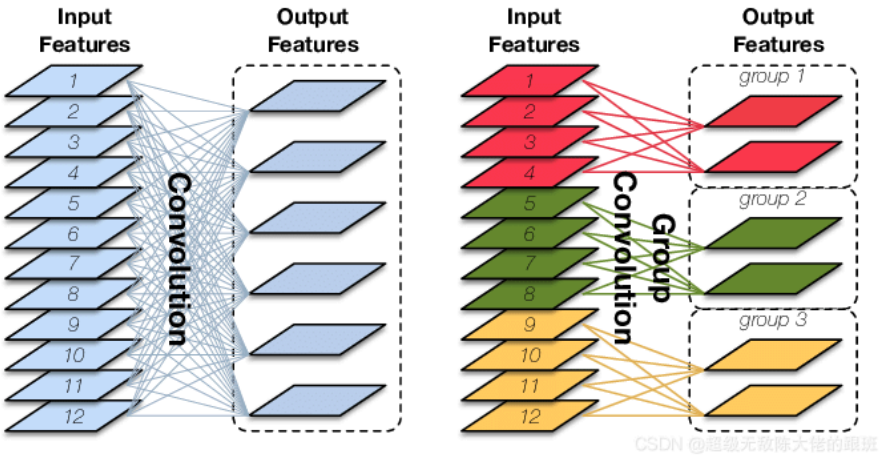
4.1 反卷积（转置卷积）



转置卷积是标准卷积的逆运算，用于还原输入矩阵的形状

4.2 分组卷积

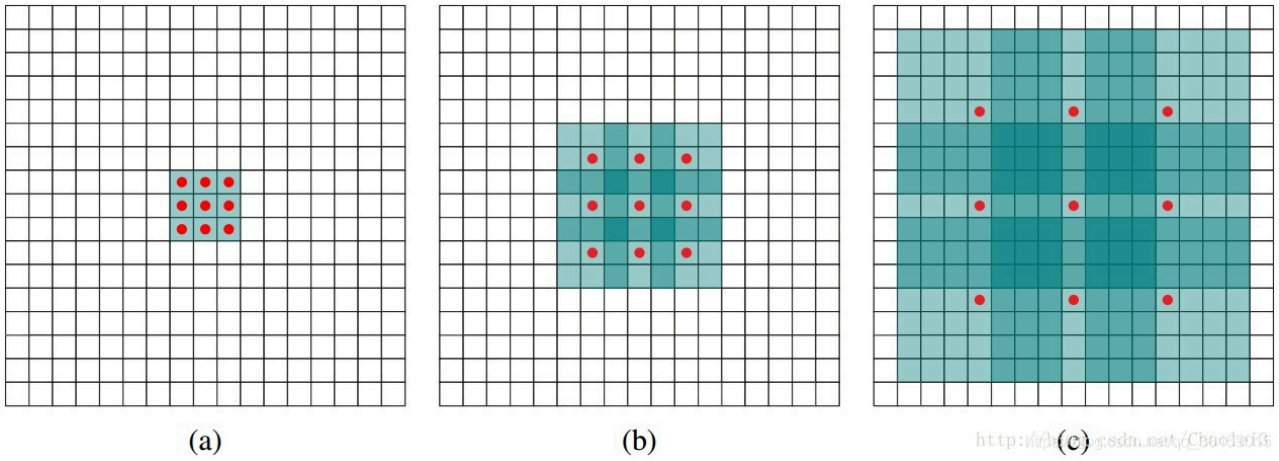
Group convolution是将输入层的不同特征图进行**分组**，然后采用不同的卷积核再对各个组进行卷积，这样会降低卷积的计算量。因为一般的卷积都是在所有的输入特征图上做卷积，可以说是全通道卷积，这是一种通道密集连接方式（channel dense connection）。而group convolution相比则是一种通道稀疏连接方式（channel sparse connection）。



4.3 深度可分离卷积

当分组数量等于通道数量、输出通道数量等于输入通道数量时，分组卷积就成了深度可分离卷积，参数量进一步锐减。

4.4 空洞卷积



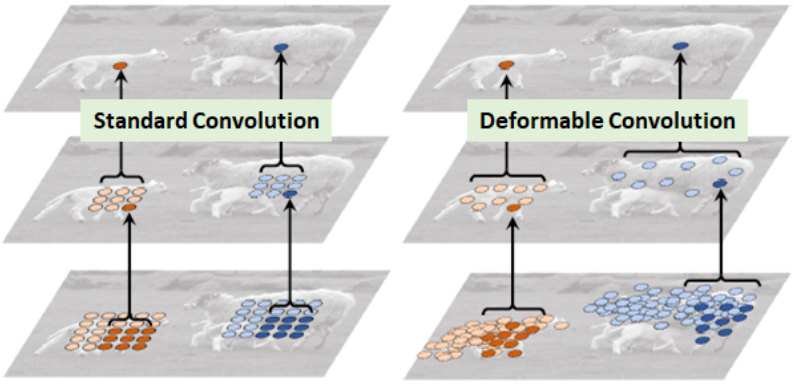
空洞卷积（dilated convolution）是针对图像语义分割问题中下采样会降低图像分辨率、丢失信息而提出的一种卷积思路。利用添加空洞扩大感受野，让原本3x3的卷积核，在相同参数量和计算量下拥有5x5（dilated rate =2）或者更大的感受野，从而无需下采样。

4.5 3d卷积

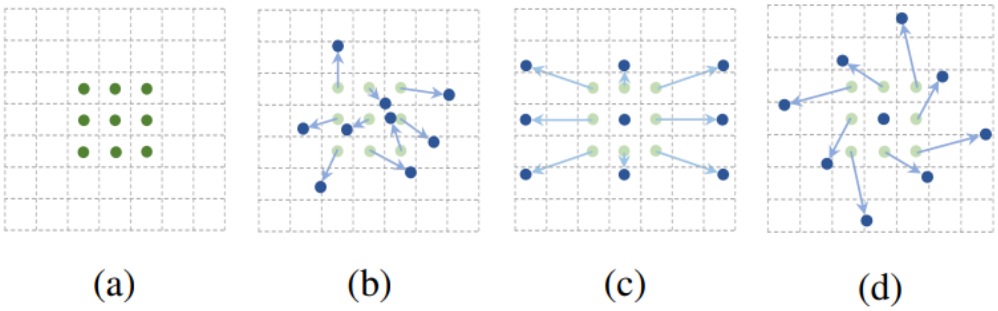
标准卷积是2d卷积，3d卷积就是卷积核移动的方向多了一个维度，其他均与2d卷积类似。通常用于视频数据、3d物体数据等。

4.6 可变形卷积

可变形卷积顾名思义就是卷积的位置是可变形的，并非在传统的$N \times N$的网格上做卷积，这样的好处就是更准确地提取到我们想要的特征（传统的卷积仅仅只能提取到矩形框的特征）



可变卷积实际上就是在每一个卷积采样点加上了一个偏移量：

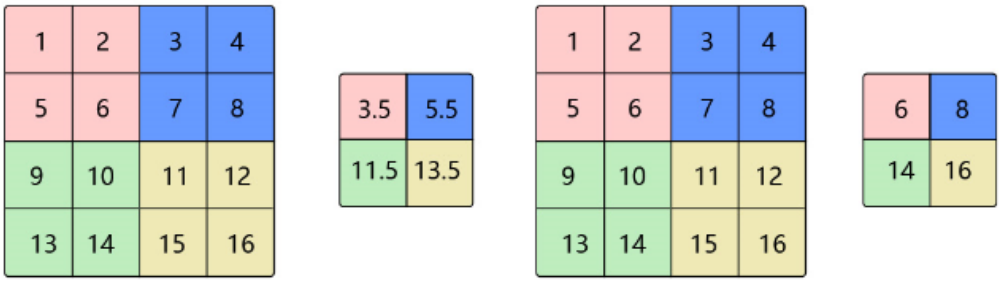


5 池化

在图像处理中，由于图像中存在较多冗余信息，可用某一区域子块的统计信息（如最大值或均值等）来刻画该区域中所有像素点呈现的空间分布模式，以替代区域子块中所有像素点取值，这就是卷积神经网络中池化(pooling)操作。

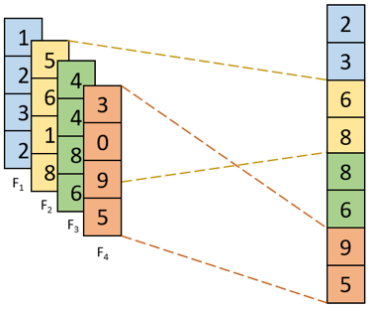
池化操作对卷积结果特征图进行约减，实现了下采样，同时保留了特征图中主要信息。比如：当识别一张图像是否是人脸时，我们需要知道人脸左边有一只眼睛，右边也有一只眼睛，而不需要知道眼睛的精确位置，这时候通过池化某一片区域的像素点来得到总体统计特征会显得很有用。

池化的几种常见方法包括：平均池化、最大池化、K-max池化。



- **平均池化：** 计算区域子块所包含所有像素点的均值，将均值作为平均池化结果。如上图左侧，这里使用大小为2×2的池化窗口，每次移动的步幅为2，对池化窗口覆盖区域内的像素取平均值，得到相应的输出特征图的像素值。池化窗口的大小也称为池化大小，用$k_h \times k_w$表示。在卷积神经网络中用的比较多的是窗口大小为2×2，步幅为2的池化。

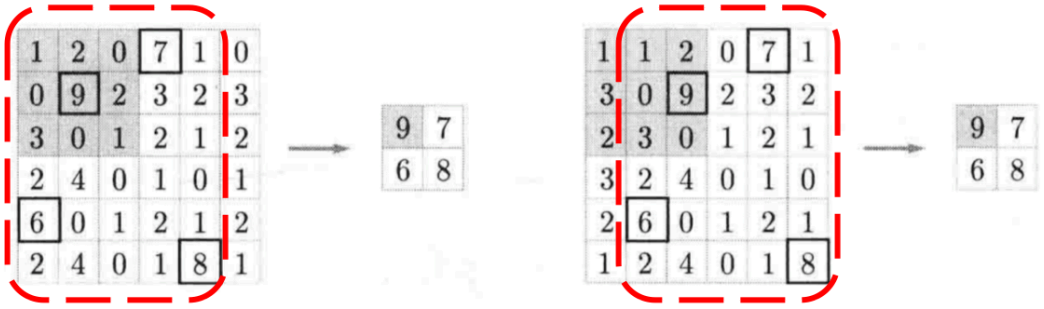
- **最大池化：** 从输入特征图的某个区域子块中选择值最大的像素点作为最大池化结果。如上图右侧，对池化窗口覆盖区域内的像素取最大值，得到输出特征图的像素值。当池化窗口在图片上滑动时，会得到整张输出特征图。



- **K-max池化：** 对输入特征图的区域子块中像素点取前K个最大值，常用于自然语言处理中的文本特征提取。如上图，从包含了4个取值的每一列中选取前2个最大值就得到了K最大池化结果。

5.1 特点

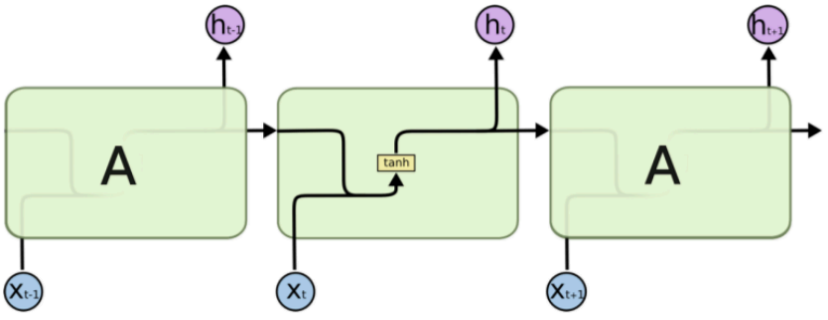
当输入数据做出少量平移时，经过池化后的大多数输出还能保持不变，因此，池化**对微小的位置变化具有鲁棒性**。例如下图中，输入矩阵向右平移一个像素值，使用最大池化后，结果与平移前依旧能保持不变。



由于池化之后特征图会变小，如果后面连接的是全连接层，能有效的**减小神经元的个数，节省存储空间并提高计算效率**。

循环神经网络

1 RNN



$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

其中h_t是第t步的隐藏状态，x_t是第t步的输入，h_{t-1}是第t-1步的隐藏状态或初始隐藏状态。tanh也可以换成relu。

RNN 是一种死板的逻辑，越晚的输入影响越大，越早的输入影响越小，且无法改变这个逻辑。

1.1 缺点

- RNN 只有短期记忆，无法处理很长的输入序列
- 训练 RNN 需要投入极大的成本。

1.2 RNNs 训练和传统 ANN 训练异同点？

- 相同点：都使用 BP 误差反向传播算法。
- 不同点：
  - RNNs 网络参数 W,U,V 是共享的，而传统神经网络各层参数间没有直接联系。
  - 对于 RNNs，在使用梯度下降算法中，每一步的输出不仅依赖当前步的网络，还依赖于之前若干步的网络状态。

1.3 为什么 RNN 训练的时候 Loss 波动很大？

由于 RNN 特有的 memory 会影响后期其他的 RNN 的特点，梯度时大时小，lr 没法个性化调整，导致 RNN 在训练过程中，Loss 会震荡起伏，为了解决 RNN 的这个问题，在训练的时候，可以设置临界值，当梯度大于某个临界值，直接截断，用这个临界值作为梯度的大小，防止大幅震荡。

1.4 RNN 中为什么会出现梯度消失？

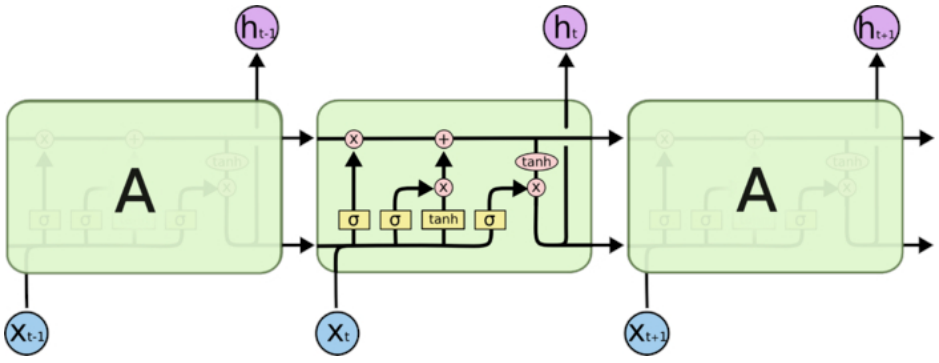
rnn结构的循环累乘会导致激活函数导数的累乘，如果取 tanh 或 sigmoid 函数作为激活函数的话，那么必然是一堆小数在做乘法，结果就是越乘越小。随着时间序列的不断深入，小数的累乘就会导致梯度越来越小直到接近于 0。

实际使用中，会优先选择 tanh 函数，原因是 tanh 函数相对于 sigmoid 函数来说梯度较大，收敛速度更快且引起梯度消失更慢。

1.5 如何解决 RNN 中的梯度消失问题？

1. 选取更好的激活函数，如 Relu 激活函数。ReLU 函数的左侧导数为 0，右侧导数恒为 1，这就避免了“梯度消失“的发生。但恒为 1 的导数容易导致“梯度爆炸“，但设定合适的阈值可以解决这个问题。
2. 加入 BN 层，加速收敛，控制过拟合，可以少用或不用 Dropout 和正则。
3. 改变传播结构，LSTM 结构可以有效解决这个问题。

2 LSTM



$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

其中h_t是第t步的hidden state，c_t是第t步的cell state，x_t是第t步的输入，h_{t-1}是第t-1步的hidden state或者初始hidden state，i_t、f_t、g_t、o_t是输入门、遗忘门、候选记忆元和输出门

2.1 为什么 LSTM 模型中既存在 sigmoid 又存在 tanh 两种激活函数，而不是选择统一一种 sigmoid 或者 tanh？

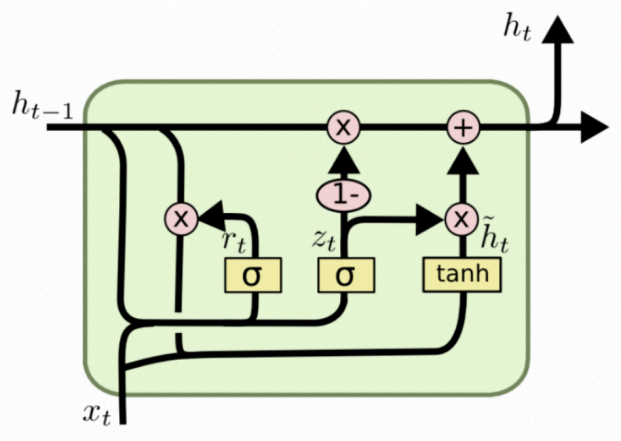
- sigmoid 用在了各种 gate 上，产生 0~1 之间的值，一般只有 sigmoid 最直接了；
- tanh 用在了状态和输出上，是对数据的处理，这个用其他激活函数或许也可以。

2.2 LSTM 中为什么经常是两层双向 LSTM？

有些时候预测需要由前面若干输入和后面若干输入共同决定，这样会更加准确。

（譬如在bert等模型流行之前，ner任务常用BiLSTM-CRF模型）

3 GRU



（2014 年提出）是对于LSTM结构复杂性的优化，其主要是从以下两个方面进行改进。

1. GRU只有两个门。GRU将LSTM中的输入门和遗忘门合二为一，称为更新门（update gate），上图中的z_t，控制前边记忆信息能够继续保留到当前时刻的数据量，或者说决定有多少前一时间步的信息和当前时间步的信息要被继续传递到未来；GRU的另一个门称为重置门（reset gate），上图中的r_t，控制要遗忘多少过去的信息。
2. 取消进行线性自更新的记忆单元（memory cell），而是直接在隐藏单元中利用门控直接进行线性自更新。GRU的逻辑图如上图所示。

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
\widetilde{h} &= \tanh(W \cdot [r_t \odot h_{t-1}, x_t]) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h}_{t-1}
\end{aligned}
$$

Transformer

经典老图，搭配后面的介绍食用

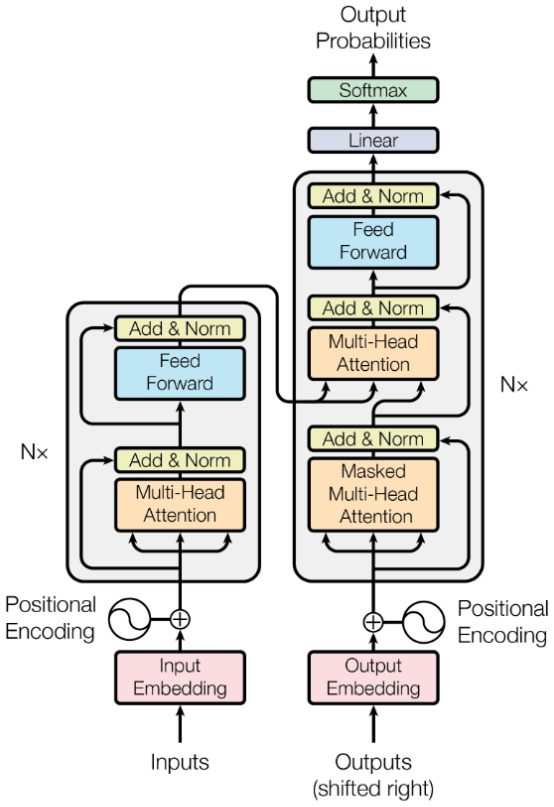


Figure 1: The Transformer - model architecture.

1 transformer结构与原理

- 输入处理:
  输入序列通过词嵌入（Word Embedding）层转化为高维向量表示。加入位置编码（Positional Encoding），使得模型能感知序列中各个词的位置。

补充:

实际处理文本内容时，对于输入的一段句子(eg. “ I'm studying deep learning. ”)，我们会将其拆分为若干个token（称之为分词，tokenize）再进行后面的运算。

token是比单个单词更小的文本单位（token包含各种单词/汉字等，但不仅仅只有单词/汉字，也会有数字，符号等），例如模型可能会将 I'm 拆分为 I 和 ‘m、将 studying 拆分为 study 和 -ing等。

每个语言模型都会有自己的词表，词表中会记录成千上万个token以及其对应的索引。


```
"model": {
    "vocab": {
        "##ki": 8614,
        "103": 8615,
        "comments": 8616,
        "name": 8617,
        "##の て": 8618,
        "##pe": 8619,
        "##ine": 8620,
        "max": 8621,
        "1987": 8622,
```

对于给到的例句，分词后的输出可能为：[151, 112, 155, 12685, 8221, 12592, 12315, 119]，对应词表中的[i, ', m, study, ##ing, deep, learning, .]。分词器（tokenizer）也可以对纯数字进行分词，例如它可能会将114514分词为[8866, 9216, 8159]，对应词表中的[114, ##51, ##4]。

```
1 print([tokenizer.decode(token_ids=i) for i in tokenizer('I\'m studying deep learning.')["input_ids"]])
✓ 0.0s
['[CLS]', 'i', "'", 'm', 'study', '##ing', 'deep', 'learning', '.', '[SEP]']

1 print([tokenizer.decode(token_ids=i) for i in tokenizer('114514')["input_ids"]])
✓ 0.0s
['[CLS]', '114', '##51', '##4', '[SEP]']
```

- 编码器（Encoder）：
  由 N 层相同的编码器层组成，每层包括两个子层：多头自注意力机制（Multi-Head Self-Attention）和前馈神经网络（Feed-Forward Neural Network）。每个子层后都有残差连接（Residual Connection）和层归一化（Layer Normalization）。
- 多头自注意力机制：将输入映射到查询（**Q**uery）、键（**K**ey）、值（**V**alue）矩阵，通过自注意力计算输出表示。
- 前馈神经网络：对每个位置的表示独立地进行两次线性变换和激活函数操作。(Linear->relu->Linear)
- 解码器（Decoder）：
  也由 N 层相同的解码器层组成，每层有三个子层：多头自注意力机制（对解码器自身输入）、多头注意力机制（对编码器输出）和前馈神经网络。每个子层后也有残差连接和层归一化。解码器的多头自注意力机制在训练时使用掩码（Masking），以确保当前位置只能关注之前的位置。
- 输出处理：
  解码器的最终输出经过线性变换和 Softmax 层，得到每个位置上可能的词汇分布。

2 注意力机制（Attention）

通过引入注意力机制，神经网络能够自动地学习并选择性地关注输入中的重要信息，提高模型的性能和泛化能力。

（个人理解就是对一个文本序列中的每个token赋予一个可学习的权重，表示该token对于序列的重要程度）

2.1 计算流程

2.1.1 计算Q、K、V

对于序列q, k, v

$$Q = W^Q q, K = W^K k, V = W^V v$$

其中W^Q, W^K, W^V为可学习的权重矩阵

2.1.2 计算注意力权重

注意力权重计算公式（x_i基于x_j的权重）：

$$\text{Attention}(i, j) = \frac{Q_i K_j^T}{\sqrt{d_k}}$$

其中Q_i和K_j分别是第i个和第j个元素的Query和Key向量，d_k是Key向量的维度。

除以$\sqrt{d_k}$是为了稳定训练过程（为了归一化q*k的结果，防止输入softmax的值过大，导致偏导数趋近于0）

2.1.3 计算最终输出

利用上一步得到的注意力权重对Value进行加权求和，得到最终的输出O_i

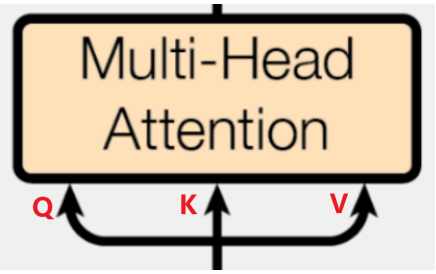
$$O_i = \sum_{j=1}^{L} \text{Attention}(i,j)V_j$$

2.1.4 总结

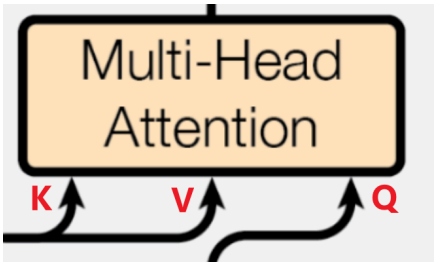
实际是通过矩阵运算实现：

$$O = \text{Attention}(Q,K,V) = \text{softmax}(\frac{Q_i K_j^T}{\sqrt{d_k}})V$$

2.2 自注意力/交叉注意力



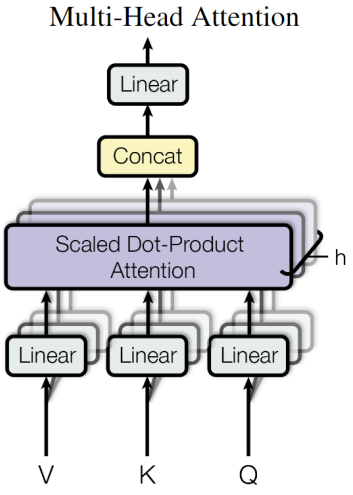
自注意力就是Q、K、V来自同一个序列，例如上图transformer**编码器**中的多头注意力



交叉注意力就是Q、K、V来自不同序列，例如上图transformer**解码器**的多头注意力，这里query源自解码器的输入，key和value源自编码器的输出

2.3 多头注意力

大概就是牺牲了W映射后的维度大小，换取多个独立的小权重矩阵，以获取更丰富的文本表示



对于每个头h_i，其都是一个完整的注意力运算（回顾【计算流程】一节），即：

$$h_i = f(W_i^Q q, W_i^K k, W_i^V v) \in \mathbb{R}^{p_v}$$

其中f是注意力操作，譬如【计算流程】一节所述的原始注意力。

最终的输出是将所有的h_i拼接起来，并线性映射为想要的维度：

$$O = W_o \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix} \in \mathbb{R}^{p_o}$$

实际中为了避免计算代价和参数代价的大幅增长，通常设定$p_q = p_k = p_v = p_o/h$，其中p_q是W^Q对Query线性映射后的维度，其他同理，h为多头注意力的头数。

为什么要多头？模型能学习到多维度的特征信息，这使得模型可以从多个维度更好的理解数据。

> 在实践中，当给定相同的查询、键和值的集合时，我们希望模型可以基于相同的注意力机制学习到不同的行为，然后将不同的行为作为知识组合起来，捕获序列内各种范围的依赖关系（例如，短距离依赖和长距离依赖关系）。因此，允许注意力机制组合使用查询、键和值的不同*子空间表示*（representation subspaces）可能是有益的。——动手学深度学习

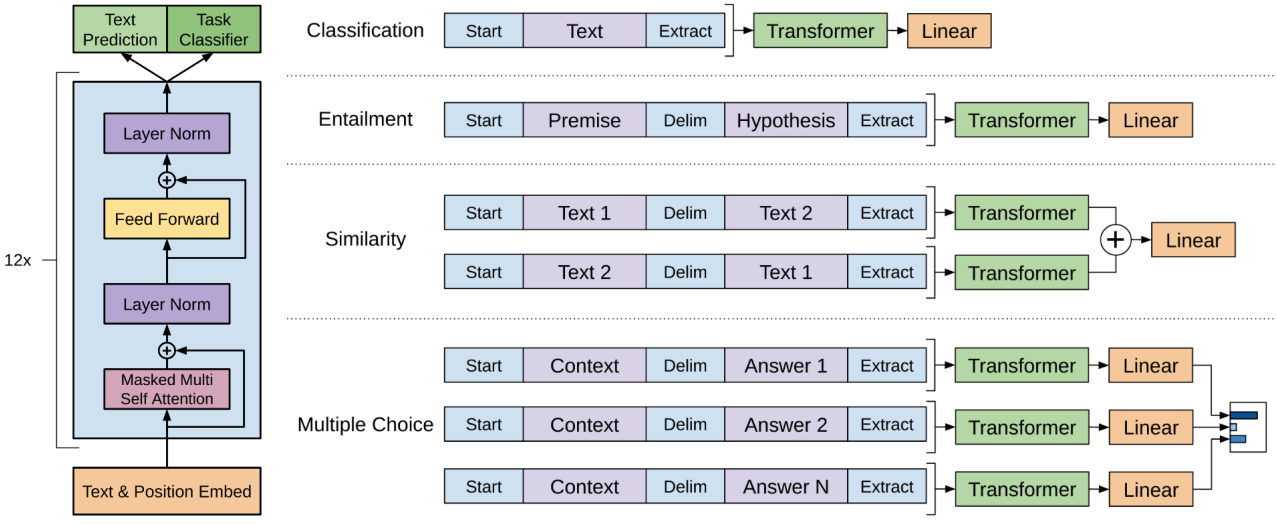
---

3 衍生模型

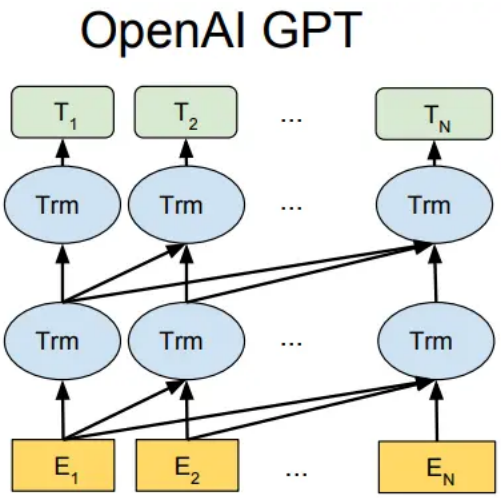
注：为了便于理解，笔者对以下内容进行了精简，如果想深入了解可以自行找博客或者看原论文。

3.1 gpt

GPT(Generative Pre-Training)，是OpenAI在2018年提出的模型，利用Transformer模型来解决各种自然语言问题，例如分类、推理、问答、相似度等应用的模型。GPT采用了Pre-training + Fine-tuning的训练模式，使得大量无标记的数据得以利用，大大提高了这些问题的效果。



GPT采用了单向Transformer结构，具体而言，OpenAI将Transformer Encoder中的Self-Attention替换成了Masked Self-Attention，句子中的每个词都只能对包括自己在内的前面所有词进行Attention。



前面的官方描述有点晦涩难懂，简单来说：

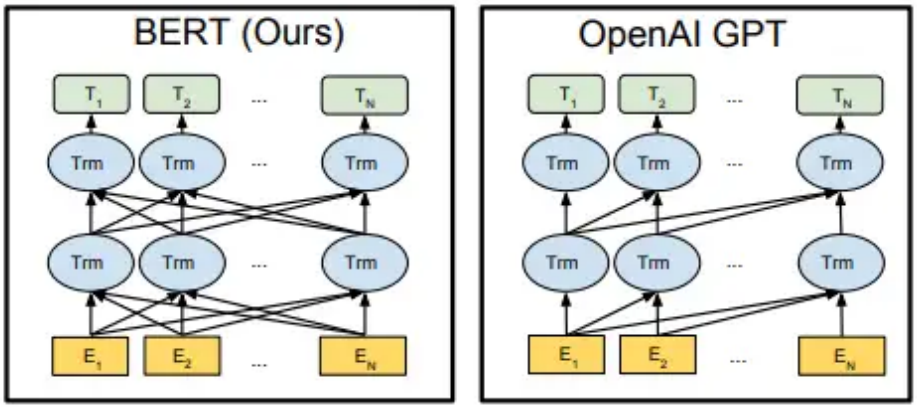
GPT注重于文本生成。因此它的预训练任务是给定一段话，预测下一个词是什么，例如：

基于训练数据`I am a student majoring in artificial intelligence, and I'm currently studying deep learning.`，gpt需要在给定`I am a student majoring in artificial intelligence, and I'm currently studying`的前提下，预测出`deep`，再根据`I am a student majoring in artificial intelligence, and I'm currently studying deep`，预测出`learning`，以此类推，进行文本生成。

3.2 bert

BERT(Bidirectional Encoder Representation from Transformer)，是Google Brain在2018年提出的基于Transformer的自然语言表示框架。BERT与GPT一样，采取了Pre-training + Fine-tuning的训练方式，在分类、标注等任务下都获得了更好的效果。

BERT与GPT非常的相似，都是基于Transformer的二阶段训练模型，都分为Pre-Training与Fine-Tuning两个阶段，都在Pre-Training阶段无监督地训练出一个可通用的Transformer模型，然后在Fine-Tuning阶段对这个模型中的参数进行微调，使之能够适应不同的下游任务。



但是**BERT更注重对文本的理解**，希望能够同时得到上下文的信息，故而采用了双向的Transformer，这种结构会导致每个Transformer的输出都是基于整个上下文的全部信息，所以它虽然也能勉强进行文本生成，但文本理解才是它的强项。

bert提出了一种新的预训练方法：Masked Language Model (MLM)。

简单来说就是做完形填空，给定一段话，我们随机盖住一些词让bert猜是啥词，待选词是整个词表。例如：

基于训练数据`I am a student majoring in artificial intelligence, and I'm currently studying deep learning.`，bert需要在给定`I am a <MASK> majoring in artificial <MASK>, and I'm currently studying deep <MASK>.`的前提下，预测出被mask的词分别是`student`、`intelligence`和`learning`。而想要在海量词表中正确地挑选出这些词，就必须理解文本内容，进而训练了bert的文本理解能力。

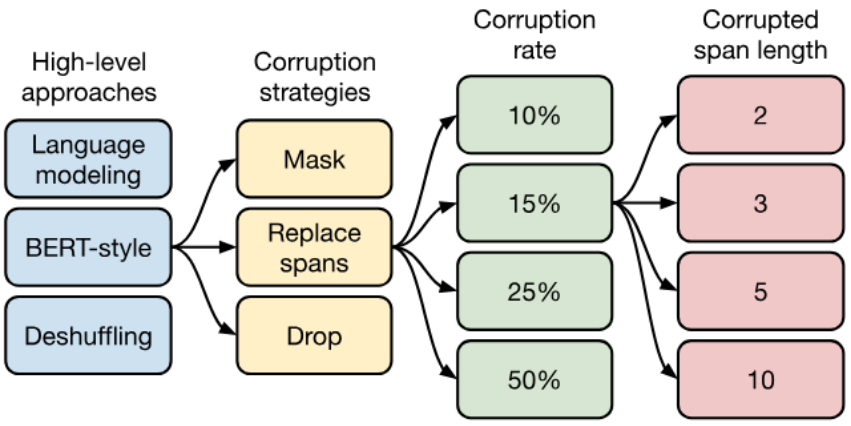
3.3 t5

transformer的其中一部分被gpt和bert拿出来用作了自回归模型和自编码模型，t5是用整个transformer做工作。从结构上看，t5就是一个完整的transformer模型，它的贡献在于 ~~让别人看到google有多少圈子~~ 提出了一个统一框架，靠着大力出奇迹，**将所有 NLP 任务都转化成 Text-to-Text （文本到文本）任务**。

比如英德翻译，只需将训练数据集的输入部分前加上“translate English to German（给我从英语翻译成德语）”就行。假设需要翻译"That is good"，那么先转换成 "translate English to German: That is good." 输入模型，之后就可以直接输出德语翻译 “Das ist gut.”

再比如情感分类任务，输入"sentiment: This movie is terrible!"，前面直接加上 “sentiment: ”，然后就能输出结果 “negative（负面）”。

通过这样的方式就能将 NLP 任务都转换成 Text-to-Text 形式，也就可以用同样的模型，同样的损失函数，同样的训练过程，同样的解码过程来完成所有 NLP 任务。



选取原始的transformer结构的原因很简单，谷歌财大气粗地逐层遍历所有可能的模型参数，最后发现原始的encoder-decoder结构表现最好~~（流下了羡慕的口水）~~。

> t5这种早在2019年就发表的、将所有 NLP 任务都转化成 Text2Text 任务的思想，与当今大火的大模型+prompt的思想如出一辙，连同gpt的文本生成、bert的文本理解，一瞬间有一股承前启后感扑面而来，令人感慨。

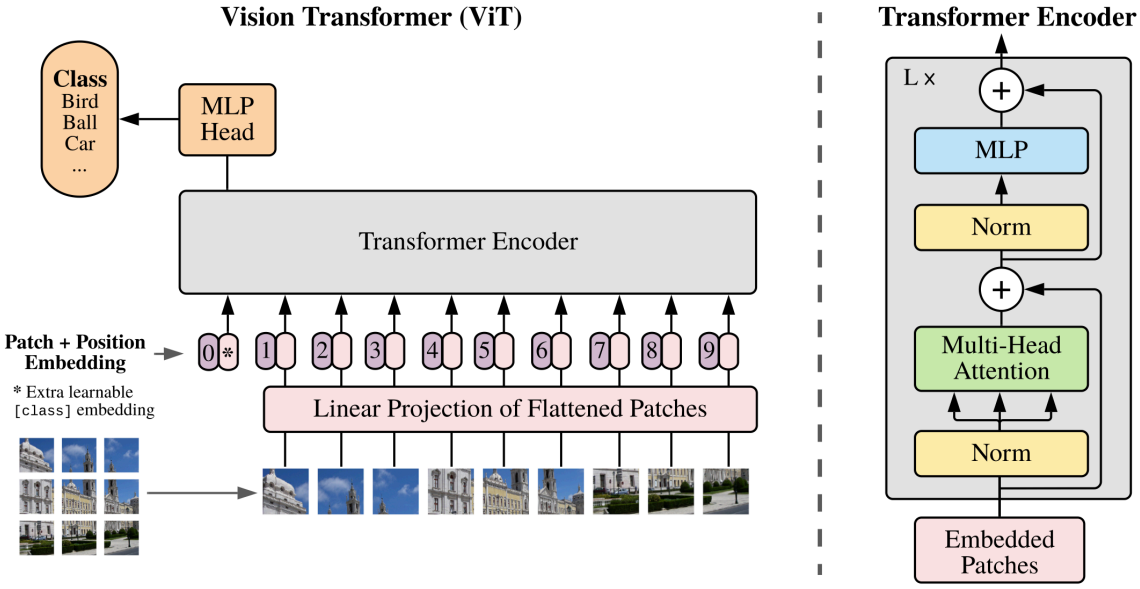
3.4 ViT

在计算机视觉领域中，多数算法都是保持CNN整体结构不变，在CNN中增加attention模块或者使用attention模块替换CNN中的某些部分。有研究者提出，没有必要总是依赖于CNN。借由transformer在nlp领域的出色表现，研究者提出了ViT算法，仅仅使用Transformer结构也能够在图像分类任务中表现很好。

受到NLP领域中Transformer成功应用的启发，ViT算法中尝试将标准的Transformer结构直接应用于图像，并对整个图像分类流程进行最少的修改。具体来讲，ViT算法中，会将整幅图像拆分成小图像块，然后把这些小图像块的线性嵌入序列作为Transformer的输入送入网络，然后使用监督学习的方式进行图像分类的训练。

该算法在中等规模（例如ImageNet）以及大规模（例如ImageNet-21K、JFT-300M）数据集上进行了实验验证，发现：

- Transformer相较于CNN结构，缺少一定的平移不变性和局部感知性，因此在数据量不充分时，很难达到同等的效果。具体表现为使用中等规模的ImageNet训练的Transformer会比ResNet在精度上低几个百分点。
- 当有大量的训练样本时，结果则会发生改变。使用大规模数据集进行预训练后，再使用迁移学习的方式应用到其他数据集上，可以达到或超越当前的SOTA水平。



4 transformer 和 cnn 的区别

（当然最无脑的回答是模型结构不同）

CNN	Transformer
依赖于局部连接和权重共享，这意味着每个卷积核只关注输入数据的一个局部区域，适合处理具有强烈局部相关性的数据，如图像。	通过自注意力机制处理序列数据，能够捕捉长距离依赖关系，适合处理序列数据，如文本或时间序列。
卷积核的参数在整个输入上共享，这减少了模型的参数数量，但限制了模型捕获全局依赖的能力。	虽然也有参数共享（如自注意力和前馈网络中的权重），但自注意力机制允许模型在序列的每个位置捕获全局信息。
在计算机视觉任务中非常流行，如图像分类、目标检测和图像分割。	在自然语言处理（NLP）任务中表现出色，如机器翻译、文本摘要、问答系统等。
卷积层可以被解释为提取图像中的局部特征，如边缘和纹理。	自注意力机制提供了一种直观的方式来理解模型是如何在不同位置之间分配注意力权重的。