DL用过的一些方法的整理

写在前面：这是为了高通的AI\_IoT的intern面试做的准备工作。想了一下，还是把这些方法整理一下，方便以后查阅。以及，也许我并非学不懂ML AI，也许和网络，并行计算一样，我只是需要再次系统的学习一遍，或者很多遍。

CNN

**传统机器学习方法的问题**

1. 传统的方法无法处理图片这种数据量很大的问题，一个图片1000 \* 1000带上3的RGB，就会产生3000000的参数。CNN能够将参数压缩，并且做到三百万参数的图片，压缩成几百个像素也不影响我们的识别任务。
2. 传统的方法无法保留图片特征。图片中对象位置稍微变化一下，传统把图片转换成数字化的方法就会产生巨大差异，非常影响我们保留真正的图片特征。
3. CNN比较传统方法还有一个特点，就是需要最少的预处理。简单使用场景下矩阵数据并不需要数据转换或者预处理或者特征提取而是直接塞入。

**Fun Fact**

CNN的方法也是符合视觉原理的研究的。人类的视觉识别最底层是一些瞳孔摄入的像素，然后初步处理边缘和方向，然后逐层分级抽象。

**架构**

CNN包括卷积层（Convolution Layer），池化层（Pooling Layer），全连接层（Fully Connected Layer）。

卷积的过程就是用Kernel对矩阵进行滑动窗口，每次对同样大小的sub matrix element-wise相乘（也就是每个对应的kernel element \* 矩阵的element，然后将结果相加）得到一个数值，填入output矩阵中。也就是说，output矩阵的长宽就是滑动窗口的次数。比如原矩阵5 \* 5，kernel 3 \* 3的情况下，output会是3 \* 3因为滑动窗口一共九次。

注意Kernel里面是有值的（也就是weights），具体实现中可能用很多的kernel，每一个代表一种图片的pattern，生成多个feature maps。

卷积的参数中，out channels正是生成feature maps的数量，也就是filter/kernel的数量。In channels则是前面给的out channels，而对于输入层后面的第一层卷积来说，这通常是图片的input channels也就是3。Padding类似CUDA计算中的ghost row，在边缘包一层人为数据，作用是增加边缘部分数据在卷积和池化中的计算，从而保留边缘部分的特征（如果不使用填充，网络层数越深，所丢失的边缘数据就越多，到最后可能无特征可用）。Stride则是滑动窗口的步长。

池化是为了下采样数据降维（由于kernel一般小，那么卷积后数据仍然大），避免过拟合。这里是不滑动窗口的，传入矩阵20 \* 20，采样窗口10 \* 10，则output矩阵就是2 \* 2。池化核（也就是采样窗口）可以取最大值，平均值，最小值等等。

全连接层就是正儿八经的全连接神经网络层。经过前两层的特征提取和维度压缩，全连接层才能跑的动。

应用中，CNN是多层结构的，比如池化之后继续卷积。

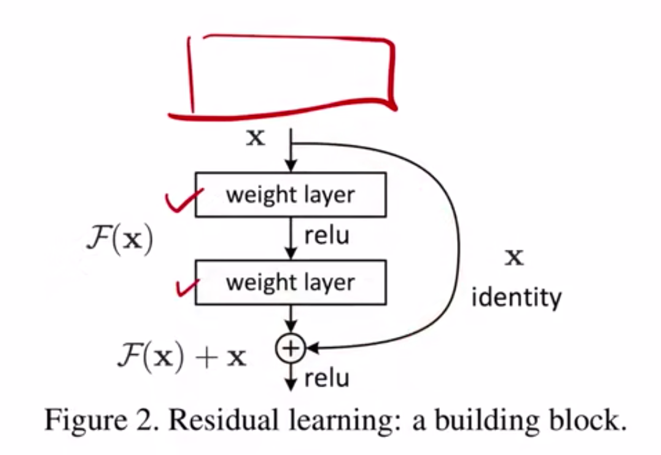
**输出层大小的计算**

高通的第一轮面试考了2D卷积的代码实现，和如何计算卷积输出的大小。我大致答对了后者，没有2 \* padding，当时以为padding是包括了两边的，没有问清楚。我的推理是给定Win, K (kernel size), S (stride), P (padding), 首先基本肯定是Win / S，因为滑动窗口嘛，然后要考虑padding，所以是(Win + 2 \* padding) / S；然后是kernel size的考虑，我们最后一个窗口的开始不可能超过Win - K，我们第一个窗口是0，第二个是1S，假设最后一个是xS，xS <= (Win + 2 \* padding) - K，所以x <= ((Win + 2 \* padding) - K ) / S。还有最后一个考虑，就是x是从0开始的，要把第一个算进去，所以是((Win + 2 \* padding) - K ) / S + 1。  
2D卷积的C++代码我写在了2d\_conv.cpp里，同时还有池化层和全连接层的代码。

ResNet

ResNet出现之前，深度学习的模型一般都是20-30层。而ResNet则能够达到100甚至1000层。ResNet论文的第一句话便是Deeper Neural Networks are more difficult to train。 不仅仅是计算成本，而是训练误差都会随着网络更深加大。深层网络面临的梯度爆炸和梯度消失可以用在中间加入Batch Normalization Layer以及normalized initialization让网络收敛，但是精度仍然会变差。这并不是overfitting因为甚至训练精度也降低了。Residual learning framework正是解决深度问题，同时反而降低了计算复杂度，因此开启了新的篇章。

这个层数问题实际上是能够有一种解法的，最坏的情况就是后面的layers全部变成identity mapping（比如输出是输入的成比例的一一对应）那精度就起码不会变差。但是SGD无法找到这个解法。ResNet提出的解法就是后面的layer不学习H(x) （也就是实际任务，比如输入图片输出标签），而是学习F(x) = H(x) - x，也就是前面学习的输出x和真实输出（标签）H(x)的残差。 而这些层的输出，就是F(x) + x，把x再加回来。 这里这个加x回来是纯粹的identity mapping，不会造成额外的计算复杂度。



在实现上，如何解决残差输入和输出的size不同呢。作者提出了两个方法，第一是在identity x中补0使得形状一样，第二是Projection时使用1 \* 1的卷积层，也就是不做任何卷积，但是可以改变output channel，使得输出通道是输入通道的两倍，从而匹配identity的形状。两种方法的stride都要调成2（这部分还没理解）。

在实际使用中，ResNet作为骨干的变种非常常见。林杉在Audio Synthesis的检测时，将声音先转成了histogram，然后把histogram变成图片，塞到了ResNet-50里。。。就这样获得了98%的准确度。我自己则是把声音通过傅立叶变化转到了frequency domain然后生成图片格式，塞到了CNN里，有87%左右的准确度。当时甚至还不明白ResNet之于普通的CNN的优势，只是觉得是一种优化，但是现在明白根本不是一个量级的。

DL的基础方法（训练，模型大小，etc.）

**K-fold cross validation**

一般来说，训练的时候会留出15%-20%的evaluation data用来test。这部分的test训练的时候是不能看到的。但是当数据量比较小的时候，预留test数据就非常的奢侈。所以可以把数据集分成很多块，每一块轮流当evaluation data，剩下的用于训练然后轮回，最后的结果应该是平均。在Capstone中我使用了K-fold cross validation，这是因为数据是自己跳出来的，很累的，一共也就1000左右的windows。

**Data Normalization与Batch/Layer Normalization Layers**

在数据预处理的时候，我们通常会对数据进行Normalization，这样才能学习到数据的规律，避免受到数据规律能学习但是值相差很大导致学不到规律的情况。比如，在Capstone时，各个sensor收上来的数据是差异很大的，我使用了Stand Scaling（也就是Z-score scaling）将数据缩放到了std调整到1(表面上看是缩放到0-1左右的range)，保持了数据的形状。Z-score是统计中当前值x与数据平均值除以alpha得到的新值。这是最标准的normalization方法。

图表

描述已自动生成形状

描述已自动生成

在Capstone中我还尝试了MinMaxScaling，也就是计算出所有数据中的min和max之后，把数据缩放到那个范围内。效果差别不大，思考了一下计算上也没有差别，同时新的数据在min max range外的情况，其实同样会对Z- score产生影响形成一个不在range里的值。

在前向传播forward propagation的时候，输出y = x \* weight + bias，这个输出经过比如tanh的激活函数（上图右）之后，会有无论多大的数据都挤在波动很小的边界上的问题。比如tanh(0.1) = 0.1，tanh(2) = 0.96，相差非常的近，把这个输出传给后面的layers就学不到东西了。如果我们能在传入activation function之前对输出数据做normalization，那么这将能够把输出数据拉回到tanh波动很大的那一段，有利于保留数据的规律/特征。这不仅是输入层而是隐藏层的问题，所以我们要在网络中加入BN，对forward propagation的结果归一化。Batch则是指对整个数据分批次的Normalization。注意还有一步scale & shift，这是为了让网络能够学习调整BN的优化作用，如果没有效果，用aX \* b的a b抵消掉BN的效果。

Layer Normalization则是Batch Normalization的另一种打开方法, 用于Transformer中. 在一个二维数组中, 假设 columns是feature, rows是一行行的数据, 这个二维数组是一个batch, 那么batch normalization就是计算的一列的variance和均值(一个feature的所有数据), 而layer normalization则是一行的, 一个数据的所有features的variance和均值. 在3D的数据中,我们假设一个batch rows还是不同的数据, 但是每一个都是一个sequence序列, 序列上的一个数据有许多features(下图). 这里就是Transformer提出的时候用来解决的是文本翻译问题, 也就是每一个序列的长度是不一样的. 那么batch normalization(下图蓝色框)切出来, 计算的variance和均值就会收到各个样本不一样长度的影响, 而且如果在预测的时候, 我们要计算全局的平均和方差, 数据是一个很长的序列之前没有见过, 那么之前训练用的variance和均值就并不适用. 因此Layer normalization(下图黄色框)就会更加适用, 每个样本自己来算均值和方差, 受全局均值方差的抖动影响没有那么大.

图片包含 形状

描述已自动生成

**Stochastic Gradient Descent 随机梯度下降（SGD）**

SGD是对模型进行求解的，我的理解是用来学习parameters（在卷积中就是Kernel上的weight）的方法。W是model parameter，B是batch size（这一段我记录的是mini- batch的SGD），l是learning rate，我们在训练直到收敛（converge）的过程中如何去学习新的parameters呢？mini batch SGD提出的是每次随机的从训练数据集中取出B大小的batch，对这个batch的input数据求出输出，然后求出损失函数相对于W的导数，导数乘上l，旧的parameters减去这个数值就是新的weight parameters。High level的理解的话，Learning rate相当于每次的新尝试，导数相当于给一个尝试的方向。

如果重新整理一下ML的思路的话，我们有一个目标函数，对于一个input X，计算output Y’，并且比较和真实标签Y的区别。假设我们的loss函数是MSE，L = 0.5 \* (Y’ - Y) \*\* 2 = 0.5 \* (f(X) - Y) \*\* 2。我们计算该模型，优化parameter的方法就是想办法把loss降到最低，也就是找到一种f(X)的计算方法（也就是里面的weights）使得L最低。如果L是一条线，要找到最低点就要对0.5 \* (f(X) - Y) \*\* 2进行求导，当导数是0的时候我们就到了线的谷底。但是事实上，最复杂函数求导成本很高而且有时不能求出答案。所以SGD的方法就是对损失函数进行求导（对于weights）然后\*-1使其反过来朝向谷底，我们按照这个方向就能够接近谷底（最低的导数）。

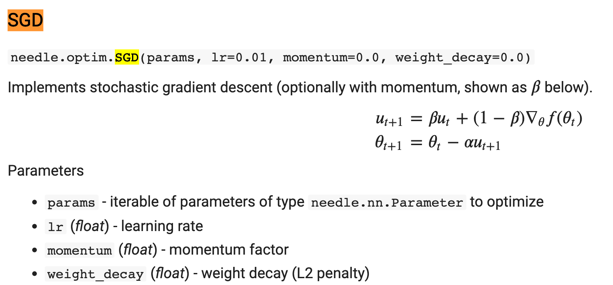
文本

描述已自动生成白板上的文字

描述已自动生成

优点在于几乎所有的线性模型和神经网络都能用SGD求解（除了决策树），缺点在于对于B和l的参数选择非常敏感。

在PyTorch的实现中，对导数先进行了加权平均，然后在正常乘上learning rate。公式：



**Back Propagation**

所以事实上，基于上面的理解我们基本上就理解了反向传播是如何对网络中的weights进行更新的。Weights的更新方式正是旧的weights减去（learning rate乘上损失函数的导数）。我们先看最后一个神经元，也就是算出来Y’的这个，如何计算导数：

图示

描述已自动生成图示

描述已自动生成

上图左拆分成两部，对MSE损失函数关于Y的求导得到Y’ - Y，对于Y’的计算（也就是目标函数）关于weight的求导得到input h1（也就是前一层的output），两者相乘得到导数。

对于在前面的神经元的导数，也可以拆分成三份，有两份和之前的计算是一样的，中间的那一份现在是对于H1求导，所以会得到W5。结果是三者相乘。注意这里W5不是更新后的值。我猜测这说明其实没有dependency，这也是为什么石知一说前向也可以更新weight。

**Adam**

Adam是一种对标SGD的优化方法。我认为掩盖掉具体的数学complexity之后，就是有两组的加权平均过的导数，并在最后用这两个导数计算出最终的导数。它的效果应该优于SGD，因为更加的平稳/避免震荡。但是需要更多的内存/计算量，因为要计算多一组导数，在图片数据中成本过高。

文本

描述已自动生成

**Pruning剪枝**

剪枝可以显著的降低卷积网络的size。基本的原理是如果一些weight已经非常小了（小于某个threshold）那么就可以被裁掉，反正它在卷积的时候不会对结果产生任何的影响了。常见的剪枝方法比如整个filter/channel裁掉，或者weight pruning（只裁掉距离0很近或者小于阈值的weight），或者更加精细，由filter channel的组合决定（regular pruning）。还一个重要的技术是threshold是可以动态学习的。在A\*STAR做research的时候，研究的是pruning filter in filter，是裁特定的group的。下图中，N是filter数量也即output channels数量。

手机屏幕截图

中度可信度描述已自动生成

**损失函数**

损失函数是用于weight update的, 根据前向计算的output, 我们可以评估其的正确性, 通过和某个我们期待的label进行比较. 损失函数的loss也会在SGD中使用.

MSE是一种很简单的损失函数, 对于每一个output, 计算和真实label的距离, 然后平方, 所有的样本的计算加起来取平均. 这个损失函数的缺点是在配合sigmoid/softmax计算的时候, 在sigmoid的两端0/1计算的时候, derivation非常小几乎接近0, 这就导致模型一开始的weight更新很小, 学习非常困难. 一般这个时候, 会用上交叉熵损失函数.

图表

描述已自动生成

I3D

**论文的初衷**

I3D的提出是用于视频是时序分析的。作者认为，光凭一张照片是没有办法得知即将发生或者已经发生什么的。所以文章提供了一个新的大型的视频数据库Kinetics。至于作者新提出的I3D方法，则是将现有的已经训练好的2D网络，直接做迁移学习变成3D网络（3 \* 3 2D kernel直接变成3 \* 3 \* 3 3D kernel），并且加入Two Stream双流。

对于视频这种有时间维度的数据来说，有几种方法可以做：第一种是2D的卷积网络配上一些LSTM blocks。如下图所示，简单的对每一个时间戳上的图片做2D卷积抽取特征，然后交给LSTM，LSTM能够做时序建模，然后在最后一个时间戳上加上全连接层，输出做分类任务。第二种就是直接3D卷积神经网络，暴力的3D卷积核，直接学习时间维度的特征。3D卷积网络的限制在于参数量实在是太大，导致网络不能够很深。我用过的C3D，当时不知道谁更好，实际上C3D只有8层，效果也超不过双流方法。第三种是双流方法，光流是说视频中从第一张图片到最后一张图片动作的轨迹，我们提前把视频中的光流提取出来。而这是一个2D的数据可以用普通的2D卷积网络，再结合第一章起始图片的2D卷积，方式是late fusion（两个logist加权平均）给出结果。第四种无非就是嫌弃late fusion简单，将所有时间维度上双流提取的特征，塞给一个小型的3D卷积网络。

图示

描述已自动生成

作者方法的机理在于之前的3D卷积是受到数据量太小的影响的。在有了足够的数据之后，3D卷积的效果要好于2D，配合光流的2D卷积特征提取，作者用了两个3D卷积，然后late fusion合在一起。作者的3D卷积网络用的是inflation，非常暴力的每个kernel扩张成3D。但是要利用预训练好的2D网络参数，怎么样才能在3D卷积网络里用呢？

作者用了非常巧妙的bootstrapping方法。要想达到两个网络用同样的参数后效果一样，作者给的输入是一个一张图复制生成的视频，然后把2D filter同样复制生成3D filter，这样卷积的结果一定是和原来的2D卷积网络一致的。

在具体网络的实现中，注意作者除了inception module之外单独加入的池化层并没有从3 \* 3 inflate成3 \* 3 \* 3，理由是在时间维度上最好不要做下采样，64帧的数据其实非常短，不适合再做下采样。故此作者用的池化层是1 \* 3 \* 3，stride也是1 \* 2 \* 2.

在我的使用中，I3D用于视频的DeepFake检测，效果非常好，记录中达到了100%或者接近的精度。可能是NTU的视频数据集太简单了。

双流方法 Two Stream

是视频领域的开山之作。在双流方法前，不管是对单独frame进行2D卷积然后进入LSTM模型，还是直接给视频数据整上3D卷积，效果都甚至不如手工设计特征。双流方法的提出开启了深度学习在视频识别领域的新思路，从双流方法开始，深度学习也逐渐占据主流。

卷积网络学习局部特征的能力很强，但作者认为对motion information这种动态的信息的学习能力不够。所以作者直接把动态的motion信息抽取出来让卷积网络学习。模型分为两个stream，作者把单帧的2D卷积称作Spatial Stream，把抽取出来的光流信息的2D卷积叫做Temporal Stream，最后对两个预测结果做加权平均（late fusion）。

光流是什么呢？Optical Flow是说比如连续的两帧，人在移动，但是背景完全没有变化，那么光流预测的算法得到的结果，背景就全部是0（光没有流动，背景没有变），但是运动的人物部分有值，而且运动幅度越大值越大。连续的两帧可以得到一个光流，因为每一个像素都有可能运动，得到的光流和图片的长宽是一样的。又运动可以拆分成水平和纵深两个方向，所以如果图片是X \* Y \* 3，那么光流的大小就是X \* Y \* 2。

图片包含 图示

描述已自动生成

一张光流的大小是X \* Y \* 2，L+1张图就会有L个光流。那么最后输入就是X \* Y \* 2L，2L可以看作是2D卷积的input channel。作者用10帧做一次叠加，那么2L就是20（这里可能应该是19？）。

Autoencoder

通过encode前半部分网络对输入input进行特征提取，通过decoder对提取出来的特征复原出原来的input。训练的时候把复原出来的输入和真实的输入进行比较计算loss并反向传播。使用的时候可以只用前半部分的网络对数据进行降维或者特征提取。如果和原输入比较的话，Autoencoder相当于一个无监督网络。当然如果在clear-blur图片对上训练，也可以完成模糊原输入图片的操作。神经网络简单的使用了全连接层（PyTorch中我使用的是nn.Linear）。

在SP Digital实习的时候，组里对time-series data用了autoencoder做anomaly detection。Autoencoder部分的定义代码如下：

表格

低可信度描述已自动生成

非常简单的一个模型。使用的时候应该是用了前半部分输出标签。真正的核心可能在于数据预处理。我们对数据进行了Period finding，也就是从原数据中找出窗口，还有Drop Extrema，Align Data，Normalize，Fill Gap等预处理。Find Period是调用了statsmodel的包，进行了autocorrelation计算。

ISP （Identity- Guided Human Semantic Parsing）

这是我在做person re-Id的时候参考的最核心的论文。我们觉得在occluded环境下面需要一些额外的方法才能保证ReID的准确性。这篇论文的作者提出的就是训练原来组图（同ID）经过Backbone网络（我们用的是ResNet50）卷积特征提取，然后进行Cascade Clustering，这一步就是human semantic parsing，会对割出来的各个part（比如头，书包）给出pseudo-labels，然后用该部分的pseudo- labels，配合backbone该部分的输出，得到这一part的结果。最终整张图片的结果是很多part结果做加权池化得到标签。

图示

描述已自动生成

Transformer

RNN对于存在时序的数据的处理方式, 是计算一个hidden state隐藏状态. 对于一个时间戳t, ht的计算取决于ht-1和当前t的input. 这就使得RNN能够有效处理时序信息的原因, 但是这种方法带来了dependency, 很难并行. 而且一旦时序比较长, RNN如果不抛弃掉早期的hidden state的话, 就必须额外存储大量的hidden states;一旦抛弃掉早期的hidden state, 就无法利用前面的信息. 解决这个时序神经网络的本质问题, 作者提出了完全基于自注意力机制的网络.

基于卷积做时序分析, 则是有潜在的限制的. 卷积是和周围邻居卷积, 这就意味着时间隔得比较远的数据, 要到网络的后面才能合, 这使得distant position之间的dependency比较难学习. 当然卷积也有好处, 卷积网络是有多个input channel的, !!!

**模型结构**

Transformer是基于encoder-decoder结构的. 在这里, encoder的工作是对于一个从x1到xn的input sequence, 生成等长的z1到zn的序列. 这里x可能是一个句子, x1是一个单词, 而z1则是机器学习可以理解的一些向量. Decoder的工作是给定z1到zn的序列, 逐个生成y1到ym的output. 这里要注意, m与n可能不一样, 比如中译英句子单词数是不一样, 同时要注意这一部分是ym的计算使用previous outputs的, 这就是auto- regressive自回归.

图示

描述已自动生成

Transformer基于这个整体的encoder decoder结构. 可以看到左边是encoder, 右边是decoder. 两者都使用了stack self-attention(也就是多头注意力机制) + point-wise全连接层 (图中的feed forward). 注意multi-head attention layer和feed forward layer都有residual connect残差连接, 然后加入layer normalization (见上normalization section). 也就是说, 对于input x, output应该是LayerNorm(x + Sublayer(x)). 这里为了保证残差连接计算时的 size统一, embedding(也就是最初文本->机器向量)的output和layer的output都被设置成了同样的值, d = 512. 后续的延伸网络调整的参数, 一般就是这个output参数和transformer blocks的数量Nx.

这里我们要理解一下简单的注意力机制. 注意力是mapping a query with key-value set. 假设我们有一组key-value set, 从1-3. 如果通过比对, 发现query1和key1比较像, 那么value1的weight就会比较高, 而最后的output就是weighted sum of all values, 可想而知, value1的占比就比较大. 所以即便key-value完全不变, query不一样就会导致output计算时候的权重变化, 造成output的变化. Transformer使用的基础attention方法叫做scaled dot-product attention, 这是说对于一个query, 会和所有的keys dot product, 然后除以一个系数(dk是keys的dimension, 这也是为什么方法名字里有一个scaled). 最后通过softmax函数, 把刚才的结果转换成非负权重. 这样再乘上values就得到output了. 对于一个query矩阵, size n \* dk, 注意这里columns要和keys相同, keys则有size m \* dk, 这样两者scale dot (keys要先transpose) + softmax, 得到n \* m矩阵, 然后再和values m \* dv, 注意这里rows自然和keys一样, 不过columns不一定一样, 得到n \* dv, 这就是我们最后的输出. 这就是比RNN更加parallel的地方, 是通过两次矩阵相乘直接求 output, 没有dependency; 另一方面, 这一步本身是不管任何时序的. 算法复杂度是n2 \* d.

这里scale得原理是, 如果不scale, 点乘+softmax的结果在dk比较大的时候, 非常向01两极靠拢. 这样的话, 梯度就会非常的小, 就有可能学习不动, 毕竟数据已经在01两端里. 所以除以了一个系数解决这个问题.

文本

描述已自动生成

那么, decoder里的mask是什么意思呢. 我们假设query高度是n, query是带有时序的, 比如Q3是Q2之后的词; keys同样长度, 同样带有时序. 那么decoder的时候, Q2运算的时候, 不应该用到后面的keys. 而我们在做正常的scale dot product的时候, 是全部keys点乘的. 那我们就需要一个mask, 把后面的计算结果换成很大的负数, 进入soft Max之后这些值就会清零.

Multi-head attention: 对于n \* dk queries, 我们会得到n \* dv outputs. 模型的output dmodel是512, 我们如果用单个attention function, dv就得是512了. 多头注意力机制就有一点类似卷积中的output channel的意思. 作者提出head数量是8, 这样我们就对QKV做projection, dv = 512 / 8 = 64, 然后并行计算, 最后再concate起来. 注意这里最后的结果不是add, 而是concate, 拼接成最后的output. 这里论文里的Q和论文前面说的scale dot-product attention里的Q不是一个概念. 这里可以将Q理解为X input, XW的结果是我们需要的Q. 所以一个head里的Q的维度就是n \* dk, 一个head的output是n \* dv, 在最后一个维度concate得到n \* hdv, 和最后一个linear WO举证相乘的到n \* dmodel, 完美的得到了我们最后想要的维度.

文本

描述已自动生成

白板上的文字

描述已自动生成

Transformer使用的是自注意力机制. 可以看到结构图里的encoder的attention layer, input即是Query, 又是Keys, 又是Values. 那么实际上, keys和input是一个东西, 意味着整个output计算就是对input一个数据对于其他input的相似度的weighted sum. 当然如果有多头机制的话, 就会有一点不一样了. 而decoder的第一个attention layer是masked, 后面第二个attention layer, 则是要依赖encoder的 output作为key和value, query是上一层的输出.

Position-wise Feed forward layer是一个MLP, 是对input x进行linear 计算 xW1 + b1, 然后通过relu, max(0, xW1 + b1), 然后再进行第二次linear计算, 就是完整的公式, max(0, xW1 + b1)W2 + b2.

到这里我们可以发现, attention layer是不会对时序信息做任何处理的, 只是单纯的点乘的加权和. 那么怎么才能考虑数据中的时序呢? 和RNN的做法不一样, transformer用了position encoding. 假设input维度512, 我们讲position, 通过周期不同的sin/cos运算(值大概在1和-1之间抖动), 变成等长的数字, 直接加入到input里面, 让input也能囊括position的信息.

为什么自注意力机制比较好呢? 下图展现了自注意力机制和别的深度学习层的方法的比较, 可以看到sequential operation, 也就是并行中的serial执行限制, 是O(1)的; maximum path length则是指distant position之间的dependency能够学习到的时间, RNN就要等待前面的output算完, 所以是O(N). 这部分CNN则是log级别的, 因为每次和邻居是O(1), 然后一层一层上去.

表格

描述已自动生成