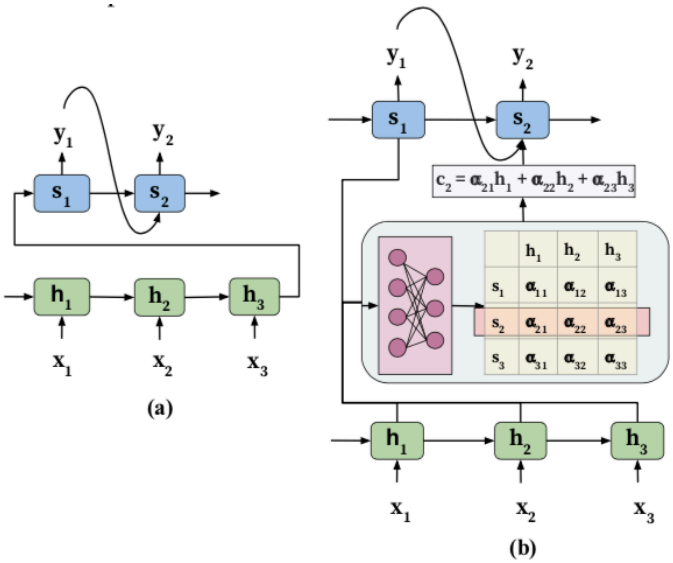
本文分为两个部分，第一部分介绍 attention 机制和 transformer，第二部分介绍预训练模型在近几年的发展。

一．Attention、self-attention、transformer

Attention 机制是一种仿生的思想。在观看一幅图片时，我们的视觉处理系统会自动聚焦在图片的某一部分，而忽略不重要的部分。而 learning machine 也可以引入 attention 机制，比如说在机器翻译或者文本摘要任务中，input sequence 中，只有有限的几个词与 output sequence 中当前要生成的词有关，image caption 任务中，只有图像的有限区域与当前 caption 要生成的词有关。Attention 机制除了在诸多 task 上产生了 SOTA 的模型，还有两点额外的好处，一是增加了模型的可解释性，二是克服了 RNN 的一些缺点，比如在处理过长的序列，会产生“遗忘”的问题，以及单向的 rnn 学到的顺序是固定的（这个问题在机翻中很明显）。



上图是原本的 auto-encoder 和加了 attention 机制的 AE，可以看到 attention 机制在产生每个词的时候，会去计算一个 attention weight，代表着对 input 不同位置的关注程度，这个 weight 通过计算当前的 hidden state 与 encoder 产生的 hidden state 之间的相关程度得到，然后再利用这个权重对 encoder hidden state 加权平均得到一个 context vector，这个向量可以作为 decoder 的输入的一部分，也可以与输出的 hidden state 拼接起来去预测下一个 token（tf 中 BahdanauAttention 与 LuongAttention 的区别）。

输入和输出分别为：

$$x = (x_1, \ldots, x_{T_x})$$

$$y = (y_1, \ldots, y_{T_y})$$

整个过程为：

$$h_t = RNN_{enc}(x_t, h_{t-1})$$

$$s_t = RNN_{dec}(\hat{y_{t-1}}, s_{t-1})$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \qquad \alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})}$$

$$e_{ij} = score(s_i, h_j)$$

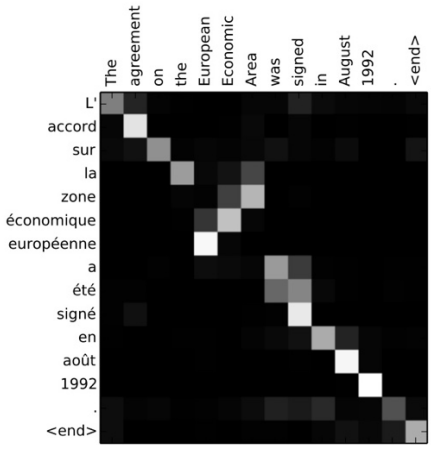
$$\hat{s}_t = tanh(W_c[c_t; s_t])$$

$$p(y_t|y_{<t}, x) = softmax(W_s\hat{s}_t)$$

其中计算 score 有三种方式，点积、双线性、加性模型：

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

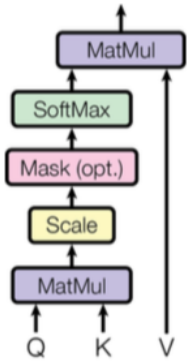
attention weight 可以可视化，如下图：



到目前为止，我们的模型仍然要依赖于 RNN 或者 CNN 结构，RNN 需要逐步递归获得序列的全部信息，CNN 需要层叠的卷积操作逐步增大感受野，google 在 17 的 Attention Is All You Need 提出了一种纯 attention 的模型--transformer，直接获得全局的信息。

Google 将 attention 机制形式化的定义为一个 query 从 key-value 的集合中去查询到一个 output 的过程，output 是 value 的加权和，而权重则是利用 query 和 key 的相似度来计算。

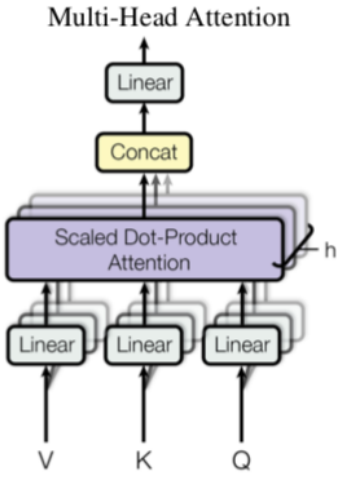
Scaled Dot-Product Attention



上图是 scaled Dot-Product Attention 的图示，矩阵 Q 是一组 query 的集合，K 和 V 分别是 key 和 value 的集合，output 的矩阵为：

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

其中 d_k 是每个 query 向量的维度，之所以要缩放它的开方是因为，这个值越大 q 和 k 的内积可能会越大，这会导致 softmax 之后概率值只有 1 和 0，无法关注到更多的信息。



Multi-head attention 机制如上图所示，将 V、K、Q 经过可训练的 h 个线性映射层，用映射之后的 h 组矩阵各自计算 scaled Dot-Product Attention 的输出，这些输出拼接起来经过一个线性层得到最终的 output：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

self attention 就是 Attention(X,X,X)，也就是说在序列内部做 attention，而不是 decoder 产生 query，encoder 产生 key 和 value，这时序列每个位置产生 output 时都能获得所有位置的信息。了解了这些之后，我们可以来看一下 transformer 的结构：

transformer的encoder部分由6个独立的layer组成，每个层包含两个sub-layer，分别为一个multi-head self-attention层和一个前向的全连接层，借鉴了Resnet，每个sub-layer都有一个residual connection：LayerNorm(x + Sublayer(x))。Decoder部分一个layer包含三个sub-layer，第一层为Masked Multi-head attention layer，和常见的decoder一样，信息是从左向右自回归的产生，这可以通过将目前还没有token的位置上不加attention实现。第二层为encoder-decoder attention layer，该层query来自于decoder的上一层，key和value来自encoder的output，这模仿了经典的基于AE的attention机制。Encoder和decoder的每一层都包含一个position-wise的个前向网络，包括两个线性层和一个relu激活层：

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

最后，由于self-attention模型完全无法捕捉到序列的位置信息，（可以想象序列a b c和序列a c b最后产生的output除了后两个position的位置颠倒以外，是一模一样的），而序列生成的任务又非常依赖于token的顺序信息，google的做法

是在每个position除了输入一个token的emb以外，再附加上一个position emb，这个emb的计算公式为：

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

其中pos是当前token的位置比如第一个词就是1，i是emb的第i维，分奇偶数由这两个公式产生。

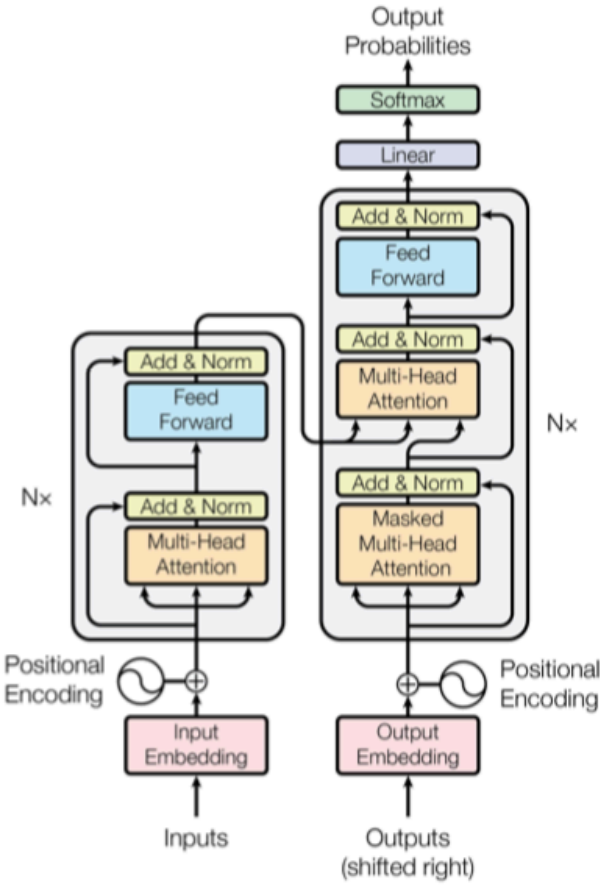
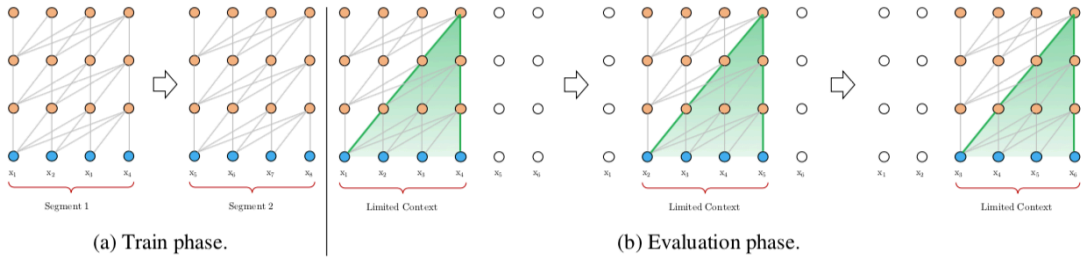


Figure 1: The Transformer - model architecture.

Transformer-xl

这是目前 SOTA 的语言模型，由于时间和设备的限制，我们不能对无限长的上下文去做 attention，一般只能对固定长度的上下文做 attention，如果长度超过限制，我们会把文本分成一些段，对每个段并行的进行训练，而在生成的时候在逐字符的进行生成，但这很难捕捉文本中长距离的依赖。



(a) Train phase. (b) Evaluation phase.

transformer-xl 的想法是建立段之间 recurrent connection，对于两个连续的段，第 n 层的 hidden state 如下计算：

$$\widetilde{\mathbf{h}}_{\tau+1}^{n-1} = \left[\mathrm{SG}(\mathbf{h}_{\tau}^{n-1}) \circ \mathbf{h}_{\tau+1}^{n-1}\right],$$
$$\mathbf{q}_{\tau+1}^{n}, \mathbf{k}_{\tau+1}^{n}, \mathbf{v}_{\tau+1}^{n} = \mathbf{h}_{\tau+1}^{n-1}\mathbf{W}_{q}^{\top}, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_{k}^{\top}, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_{v}^{\top},$$
$$\mathbf{h}_{\tau+1}^{n} = \text{Transformer-Layer}\left(\mathbf{q}_{\tau+1}^{n}, \mathbf{k}_{\tau+1}^{n}, \mathbf{v}_{\tau+1}^{n}\right).$$

梯度只在一个 segment 之间传播.

二．预训练模型

nlp 中的很多任务比如文档分类、问答、语义相似度检测，都需要在有 label 的文本集上有监督的训练，而这些文本时有限的。预训练模型的存在，是为了充分利用大量的没有 label 的文本集，来帮助在某个具体的有监督任务上提升模型的效果。

1. ELMO

ELMO 全称是 Embedding from Language Model，从名称上来看是一种学习 word representation 的模型。本文认为好的 word representation 应该具备两个能力，一是能捕捉单词在语义和语法上的用法，这一点常见的 word2vec、glove 都可以做到，二是对于一词多义的语言现象，每个词的 representation 应该随着上下文不同有所不同，这就是本文想做的工作---学习上下文感知 word representation。

本文先训练了一个 Bidirectional LM，前向和后向的 LM 都使用了多层 LSTM 来实现，优化目标为（其中 word representation θx 和 softmax 层的参数 θs 是两个方向共享的）：

$$\sum_{k=1}^{N} \Big(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big)$$

在每个位置 k，在前向的第 j 层会得到一个表示 $\overrightarrow{\mathbf{h}}_{k,j}^{LM}$，利用前向和后向这些中间层的表示，我们可以针对具体的任务有监督的学习这些表示的组合方式，来得到最终的 word representation。对于每个 token t_k，L 层的双向 LSTM 有 2L+1 个表示，将每层前向后向得到的向量拼接起来，我们可以得到：

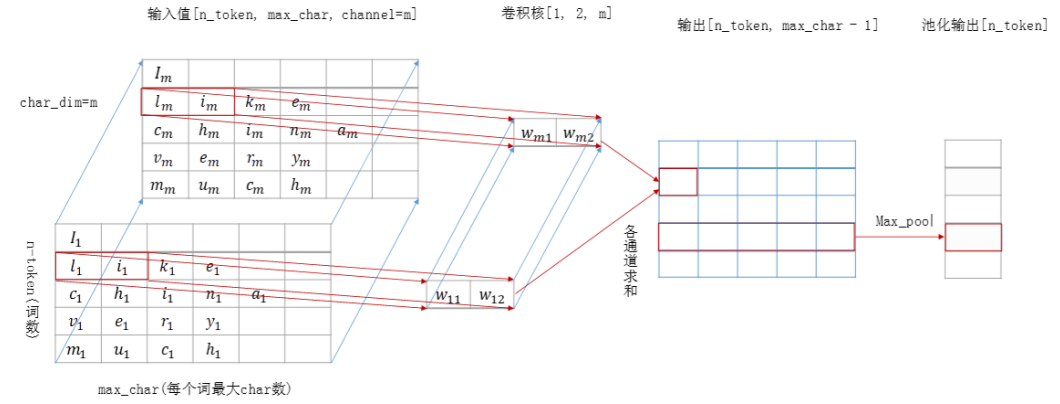
$$\begin{aligned} R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\ &= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\}, \end{aligned}$$

对不同的任务我们可以学习一组权重：

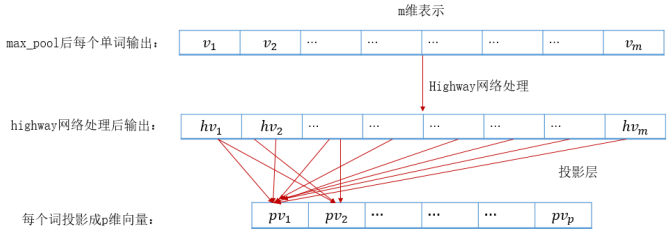
$$\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}$$

s^{task} 是一个概率向量，γ 是用来对整体所有的表示进行大小缩放的。在有监督训练阶段，已经训练好的 biLM 的参数（包括上下文无关的 embeddings x_k）被固定住（这些参数也可以在具体的任务上 fine-tune），将 x_k 与每个词的 ELMO 向量拼接起来只作为目标 RNN 的输入，有时将目标 RNN 的输出 hk 与 ELMO 向量（有一组新的权重）拼接起来代替 RNN 的输出的 hidden state 也会有更好的提升。

值得一提的是，文本使用了一种对字符训练 embedding 的方法，这样的好处是对于训练集之外的单词也可以得到一个 embedding 表示。下图展示了对于一个句子，我们可以写成词数*词长（字符数）*字符的 emb 维度的三维向量，然后通过一个(1,2,m,m)的卷积核，再经过一个 max pool，得到对每个词的 m 维向量，再线性映射到指定维度，就得到了想要的对于词的 emb。



char_cnn网络图



char_cnn网络图

最后，本文分析了为什么 biLSTM 的中间层的融合会取得很好的效果，用两层学到的表示直接去做 word sense disambiguation 任务和 POS tagging 任务，结论是第一层表示捕捉到了更多的语法信息，第二层表示捕捉到了更多的单词语义的信息。

2. GPT 1.0

GPT 全称是 generative pre-training，也就先对文本学习了一个无监督生成式模型，再迁移到有监督判别式的任务中。

GPT 预训练了一个 LM，loss 是标准的 k-gram 最大似然：

$$L_1(\mathcal{U}) = \sum_i \log P(u_i|u_{i-k}, \ldots, u_{i-1}; \Theta)$$

使用了 multi-layer Transformer decoder 来建模这个 LM，具体结构如下：

$$h_0 = UW_e + W_p$$
$$h_l = \texttt{transformer\_block}(h_{l-1}) \forall i \in [1, n]$$
$$P(u) = \texttt{softmax}(h_n W_e^T)$$

其中 $U = (u_{-k}, \ldots, u_{-1})$，是上下文的 token id，$W_e$ 是 embedding 矩阵，W_p 是 position embedding 矩阵。

LM 训练完成后，我们需要针对具体的 task 有监督的微调，对简单的文档分类任务我们在最上层 transformer 的输出上加上一个线性层，并且优化对应的目标即可：

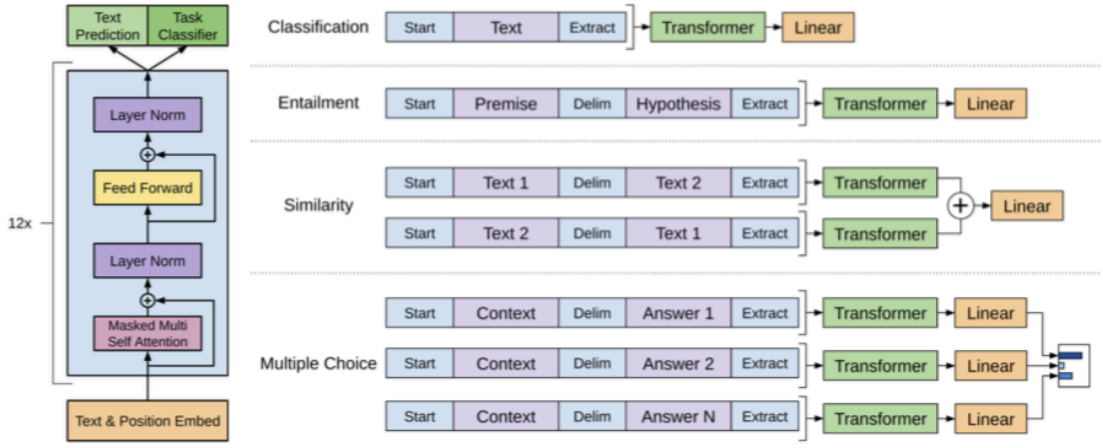
$$P(y|x^1, \ldots, x^m) = \texttt{softmax}(h_l^m W_y)$$

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y|x^1, \ldots, x^m)$$

除了该目标之外，在这个新的训练集上训练 LM，同时训练两个目标：

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C})$$

但是，除了简单的分类任务之外，还有一些其他的任务，会提供一些结构化的输入（比如一对 sequence），ELMO 等方法是单独的设计每一个的架构重新训练，而只将之前学到的表示进行迁移。而 GPT 的做法是将所有的结构化的输入合并成一个序列，从而可以交给 transformer。



textual entailment：

这个任务中我们有一对文本序列：text 和 hypothesis。我们认为一个 text 是真实的，如果可以推断出假设 b 是真实的，则称 a 蕴含 b，如果可以判断出 b 是不真实的，则称 a 推翻了假设 b。

positive TE:

text: If you help the needy, God will reward you.

hypothesis: Giving money to a poor man has good consequences.

negative TE:

text: If you help the needy, God will reward you.
hypothesis: Giving money to a poor man has no consequences.
我们可以直接将 text 和 hypothesis 拼接起来（用一个定界符分隔）作为输入最后得到一个 0~1 的概率值。

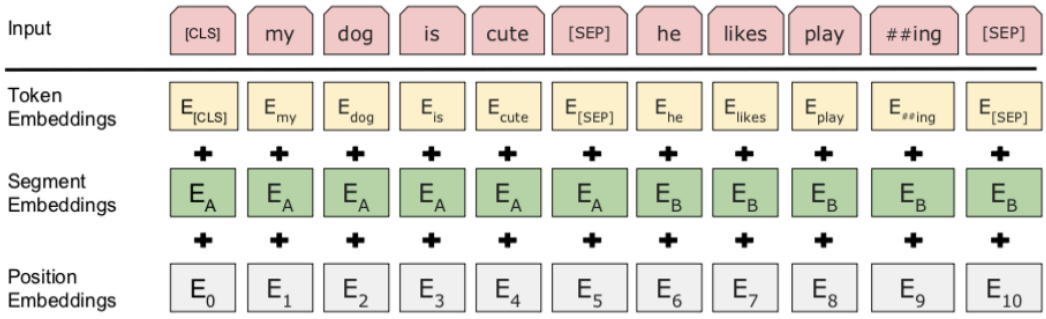
Similarity：
这个任务需要判断两个句子是否相似，因为两个句子没有固定的顺序关系，所以分别正向和反向拼接输入 transformer。

Question Answering and Commonsense Reasoning:
这个 task 中会给定一个 context document z，一个问题 q 和一个可能的回答的集合，将 z q 分别和每一个回答拼接起来，输入 transformer，经过线性层后计算 softmax 之后的概率值。

3. BERT

BERT 的全称是 Bidirectional Encoder Representations from Transformers。Google 认为 GPT 的一个主要缺陷是使用了标准的单向语言模型，信息的流动是从左向右的，在 self-attention 层中每个 token 只能看到它之前的 token。BERT 使用了多层双向 Transformer，GPT 使用了 Transformer 结构的 decoder，而 BERT 使用 Transformer 结构的 encoder，这两者的区别在于 1. encoder 可以看到两个方向的上下文，而 decoder 的某个 position 做 self-attention 对于右边的 position 会 mask 掉，2.Transform 的 decoder 部分是对于前面有的 tokens 计算下一个 token 的条件概率，而 encoder 在本文中是计算当前 token 本身的概率，该 Transformer 结构类似于一个 AE（在原始的 Transformer 中是是提供信息给 decoder 的）。

类似 GPT，为了让一个 LM 架构可以迁移到多种不同任务中，（有点任务输入是一个单独的句子，有的任务输入是一对句子），本文将两个句子拼接成一个 sequence 作为输入，为了区分该 seq 中两句，BERT 用了一个特殊字符隔开，同时增加了一个 emb 对应不同的句子，如下图：

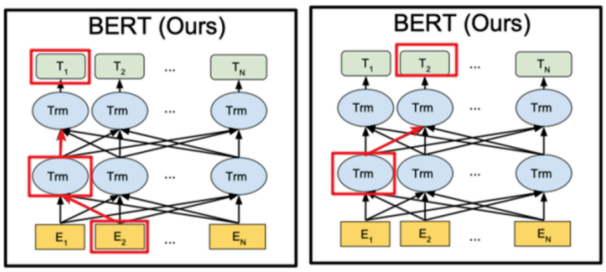


BERT 最大的创新点在于它的训练目标不再是常见的单向 LM，而是下面这两个任务：

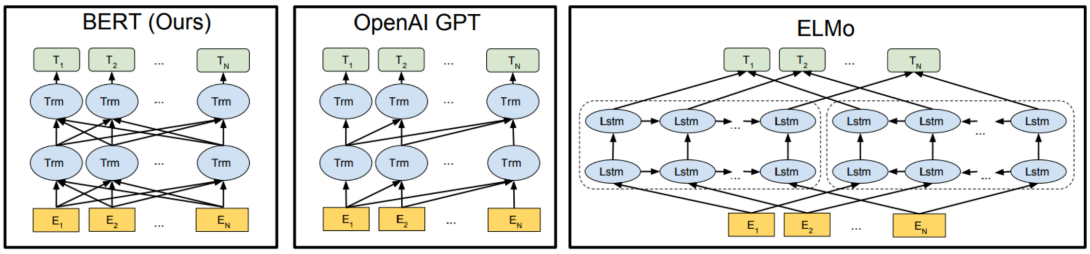
1. Masked LM

BERT 想要训练一个双向语言模型，但是直接对每一个 position 去对对应 token 最大似然是难以让模型真正学到上下文信息的。这是因为 transformer 结构自带了 self-attention，而多层的 self-attention 会让该 position 间接的得到待预测的 token 的信息，如下图，在产生 e2 位置的概率向量 t2 时，在第一个位置中间

层的向量会观察到 e2，而这个向量又会被 t2 观察到。这样就会导致模型的退化，并没有真的学到通过上下文预测。



而 ELMO 也是通过双向 LSTM 为什么没有这个问题呢，这是因为 ELMO 只是分别训练了两个方向的 LM，两套参数互不干涉，LSTM 又是个串行训练的模型。



BERT 通过训练 masked LM 来解决这个问题，将所有文本中的 15%mask 掉，并对其进行预测，这非常像 denoising autoencoder，区别在 DAE 是要解码出全部的输出。但加了 mask 又引出了一个新的问题，pre-train 阶段和 fine tune 阶段会出现不一致，因为第二阶段没有[mask]这一 token，所以最终方案是对于这 15%待预测的 token，80%替换为 mask，10%随机替换，10%保存不变。

2. Next sentence prediction

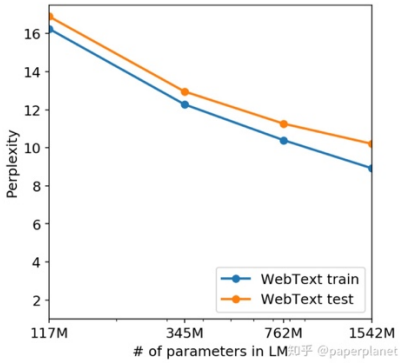
为了让模型理解一个 sequence 中两个句子之间的联系，BERT 设计了另外一个任务，将第一句 A 以 50%的概率与真实的下一句 B 拼接，以 50%的概率与随机的一个句子拼接，利用 C 来进行二分类的预测。在 fine tune 过程中，同样将 sentence pair 拼接起来，如果是 token level 的任务，就在每个 position 的 output 上进行预测，否则在 CLS 对应的 output 位置训练一个分类器。

最后 BERT 的缺点有两个，一个是 mask 在 fine-tune 的时候不存在，所以如果在 pre-train 阶段使用太大比例的 mask 会影响效果，而只对 15%的 token 进行预测，比其他模型收敛更慢。

4. GPT 2.0

从 GPT 到 BERT 的发展历程很像是 word embedding 的发展历程，从最一开始的作为训练 LM 额外得到的 emb，到后面的 Word2vec 专门为了得到这样的 emb 用来迁移到下游任务，而已经丧失的语言模型的功能------对一个句子输出它是自然语言的概率，并且可以生成更多的自然语言。因此 bert 会将所有任务转换为一个有监督分类任务。而 GPT 则可以做生成文本的任务。

GPT2.0 继续保留了这个优点，模型上除了去掉了 fine tune 阶段没有什么区别，但是将参数规模和数据规模、质量进行了提升。并且证实在 15 亿参数的情况下，模型仍然是欠拟合的，意味着还有提升的空间。在 GPT2.0 中 Transformer 充分展现了它的能力，在深度为 48 层的时候仍然不会梯度消失。



5. XLNet

Google 在提出 BERT 之后，又提出了 XLNet，刷新了 BERT 在各个任务上的记录。

以 GPT 为代表的自回归模型和以 BERT 为代表的 AE 模型各有其优劣，BERT 的目标是重建每个 masked token：

$$\max_{\theta} \quad \log p_\theta(\bar{\mathbf{x}} \mid \hat{\mathbf{x}}) \approx \sum_{t=1}^{T} m_t \log p_\theta(x_t \mid \hat{\mathbf{x}}) = \sum_{t=1}^{T} m_t \log \frac{\exp\left(H_\theta(\hat{\mathbf{x}})_t^\top e(x_t)\right)}{\sum_{x'} \exp\left(H_\theta(\hat{\mathbf{x}})_t^\top e(x')\right)}$$

上式中 m_t 只对 masked token 是 1。可以看到 BERT 存在一个独立性假设，所有 masked tokens 是相互独立的被重建，而 GPT 等基于 AR 模型则是根据乘积法则展开的条件概率。BERT 的缺点在于不能对一个句子产生一个联合概率，此外 mask 只在预训练阶段存在造成了 pretrain-finetune discrepancy。而 GPT 的缺点在于只能看到一侧的上下文。

为了同时克服这两种模型的问题，XLNet 提出新的训练目标：Permutation Language Modeling。对于一个句子的联合分布，我们除了按前后顺序分解如下的条件概率：

$$\max_{\theta} \quad \log p_\theta(\mathbf{x}) = \sum_{t=1}^{T} \log p_\theta(x_t \mid \mathbf{x}_{<t}) = \sum_{t=1}^{T} \log \frac{\exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x_t)\right)}{\sum_{x'} \exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x')\right)}$$

也可以按任意顺序分解。简单来说就是对于输入序列进行全排列，然后再按新的排列训练自回归语言模型，比如输入序列长为 T，我们有 T 的阶乘的排列，用同一个模型训练这些排列，则模型会学会整合上下文的信息。训练目标如下式：

$$\max_{\theta} \quad \mathbb{E}_{\mathbf{z} \sim \mathcal{Z}_T} \left[\sum_{t=1}^{T} \log p_\theta(x_{z_t} \mid \mathbf{x}_{\mathbf{z}_{<t}}) \right].$$

采样一个排列 z，利用 z 中前 t-1 个 token 预测第 t 个。但是，实际操作中并不会真的打乱输入的 sequence order，而只用排列 factorization order（这里的分解指的是对整个句子的联合概率分解为一系列自回归的条件概率），作者通过设计 self-attention 中的 mask 来实现。

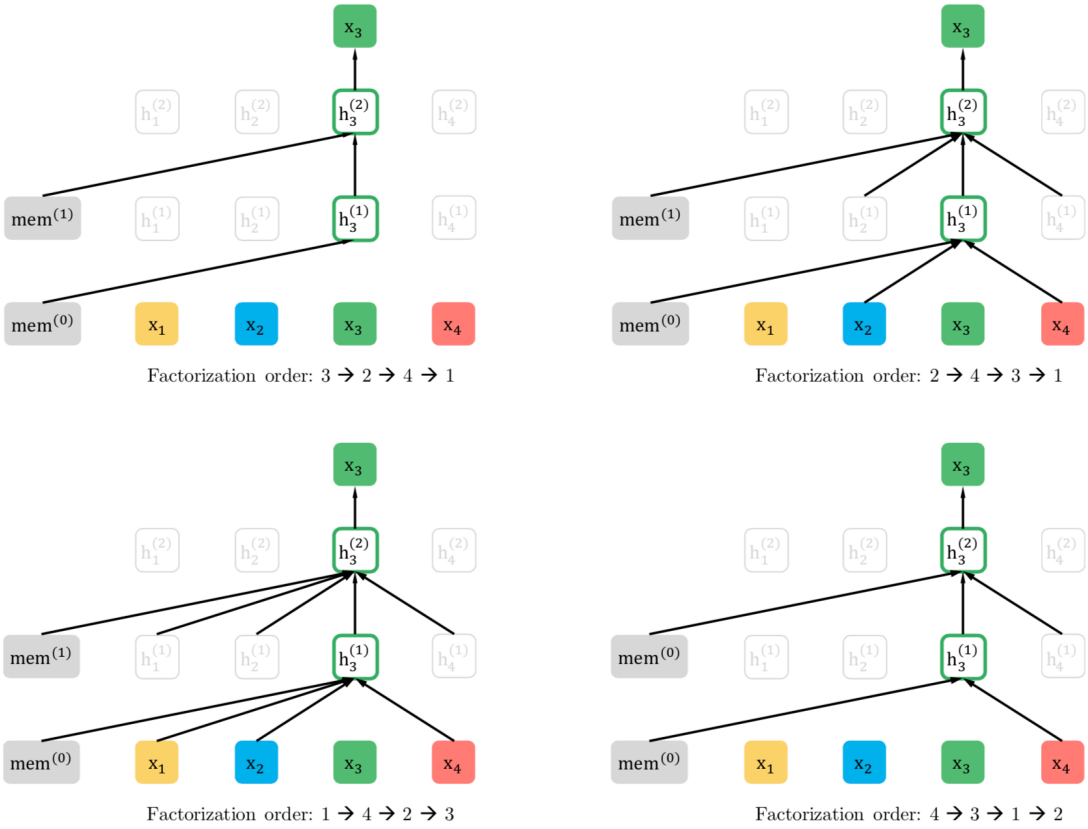


Figure 1: Illustration of the permutation language modeling objective for predicting x_3 given the same input sequence $\mathbf{x}$ but with different factorization orders.

我们来看一下具体用基于 self-attention 的 transformer 实现有什么问题。因为我们要实现一个自回归模型，优先考虑用 transformer 的 decoder，即用前面的序列进行 attention 操作，预测下一个词

$$p_\theta(X_{z_t} = x \mid \mathbf{x}_{\mathbf{z}<t}) = \frac{\exp\left(e(x)^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\right)}{\sum_{x'} \exp\left(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\right)}$$

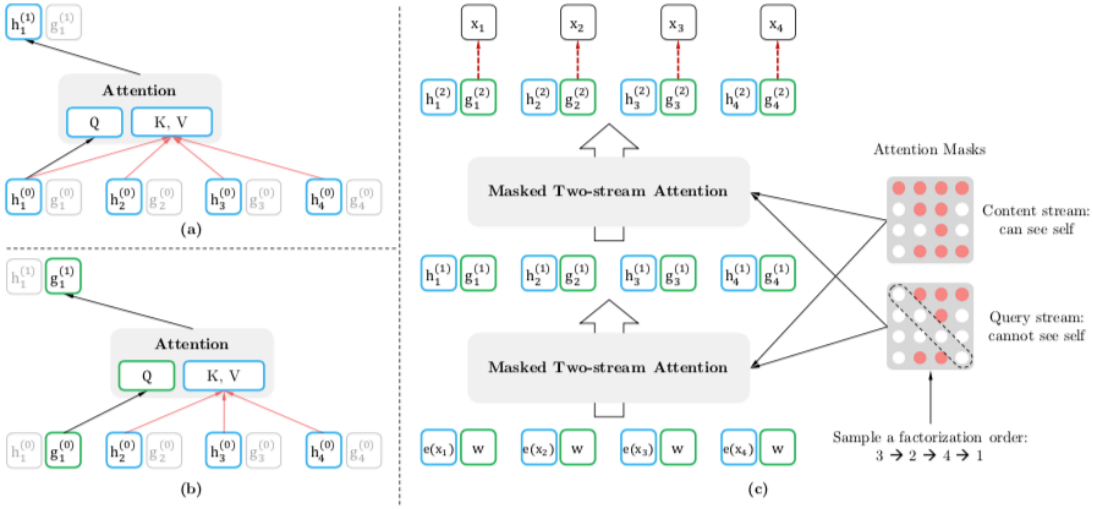
但这样会出现的问题是，模型并不知道当前要预测的 position 是哪一个，比如对于 2134 和 2135，对于不同的位置，前缀 213 会得到同样的概率分布。

我们不能在 3 这个位置上对下一个位置产生不同的预测，所以要分别在 4 和 5 的位置上做预测，这就更像是 BERT 用一个 mask 作为 query，但是我们还需要让 query 知道它的目标位置（4/5）：

$$p_\theta(X_{z_t} = x \mid \mathbf{x}_{z<t}) = \frac{\exp\left(e(x)^\top g_\theta(\mathbf{x}_{\mathbf{z}<t}, z_t)\right)}{\sum_{x'} \exp\left(e(x')^\top g_\theta(\mathbf{x}_{\mathbf{z}<t}, z_t)\right)}$$

但我们只能让 query 知道位置，不能知道具体的 token，不然就会出现类似于 BERT 中间接的看到待预测的词的问题。但对于排列 z 中，t 之后的 position 做预测时需要知道第 t 个位置的词，这个矛盾使我们需要使用 two-steam self-attention，对两种 query 分别表示。

$$g_{z_t}^{(m)} \leftarrow \text{Attention}(\mathrm{Q} = g_{z_t}^{(m-1)}, \mathrm{KV} = \mathbf{h}_{\mathbf{z}<t}^{(m-1)}; \theta), \quad \text{(query stream: use } z_t \text{ but cannot see } x_{z_t})$$
$$h_{z_t}^{(m)} \leftarrow \text{Attention}(\mathrm{Q} = h_{z_t}^{(m-1)}, \mathrm{KV} = \mathbf{h}_{\mathbf{z}\le t}^{(m-1)}; \theta), \quad \text{(content stream: use both } z_t \text{ and } x_{z_t}).$$



在训练阶段，我们算出最上层 query steam 的值，从而可以优化上式。在 finetune 阶段，我们抛弃 query stream，使用 content stream（就是一个常规的 transformer）来有监督的训练。