Summary Of MTL

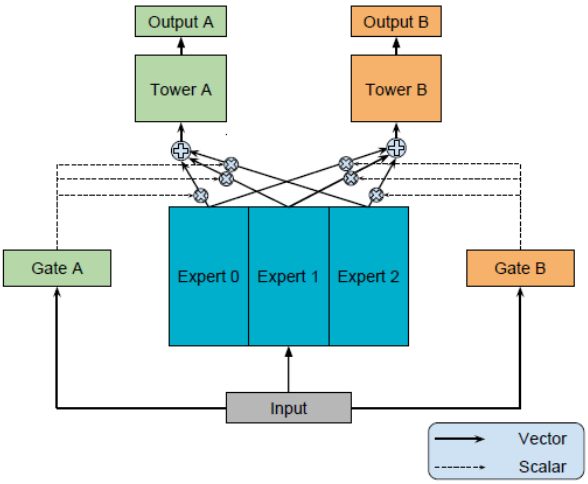
2020.12 Elssky

Contents

Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts

MMoE 是 Google 于 2018 年提出的多任务学习模型。在多任务学习中，任务之间的相关性是影响模型预测质量的关键因素。任务间差异带来的内部冲突实际上会损害某些任务的预测，尤其是当模型参数在所有任务间广泛共享时。因此，研究特定任务目标和任务间关系二者的权衡非常重要。在这篇论文中，作者提出了一种新颖的多任务学习方法，即 MMoE。该模型在 MoE 结构的基础上添加多个经过训练的门控网络以优化每个任务。MMoE 模型在任务相关性低的多任务学习中显著地展现了其结构优势带来的实际效果。

1.模型结构



MMoE 模型的思想是：所有任务共享一组底层网络，每个底层网络被称为一个 expert。每一个底层网络都是前馈网络，然后为每一个任务都引入一个门控网络（softmax）。门控网络以不同的权重将 experts 进行组合，从而允许不同的任务以不同的方式利用 experts。最后，将聚集 experts 的结果传递到特定任务的 Tower 层网络。这样，针对不同任务的门控网络就可以学习 experts 聚集的不同混合方式。

2.建模方法

门控网络g^k可表示为：

$$g^k(x) = \text{softmax}\left(W_{gk}x\right)$$

其中，$W_{gk} \in \mathbb{R}^{n\times d}$是可训练的矩阵，$n$是 experts 的个数，$d$是特征维数，$x$为输入向量 input。

门控网络g^k的输出为：

$$f^k(x) = \sum_{i=1}^{n} g^k(x)_i f_i(x)$$

$f_i(x)$是第i 个 expert 的输出，$g^k(x)_i$是门控网络$g^k(x)$对应的第i维向量

最后将门控网络g^k的结果传递到 tower 层然后输出：

$$y_k = h^k(f^k(x))$$

h^k为对于任务k的 tower 层网络

3.实验测试

3.1 Sythetic Data

因为多任务学习模型的性能高度依赖于数据中固有的任务相关性。但是，在实际应用场景中，很难直接更改任务之间的相关性并观察效果。因此，作者首先使用了人工合成的方式生成数据集来进行验证研究，这样就可以简单地对任务间相关性进行测量和控制。

思想：生成两个回归任务，并使用这两个任务的回归函数间的 pearson 相关性作为定量指标。

数据集生成方式如下：

(1) Given the input feature dimension d, we generate two orthogonal unit vectors $u_1, u_2 \in \mathbb{R}^d$, i.e.,

$$u_1^T u_2 = 0, \|u_1\|_2 = 1, \|u_2\|_2 = 1.$$

(2) Given a scale constant c and a correlation score $-1 \le p \le 1$, generate two weight vectors w_1, w_2 such that

$$w_1 = cu_1, \ w_2 = c\left(pu_1 + \sqrt{(1-p^2)}u_2\right). \tag{2}$$

(3) Randomly sample an input data point $x \in \mathbb{R}^d$ with each of its element from $\mathcal{N}(0,1)$.

(4) Generate two labels y_1, y_2 for two regression tasks as follows,

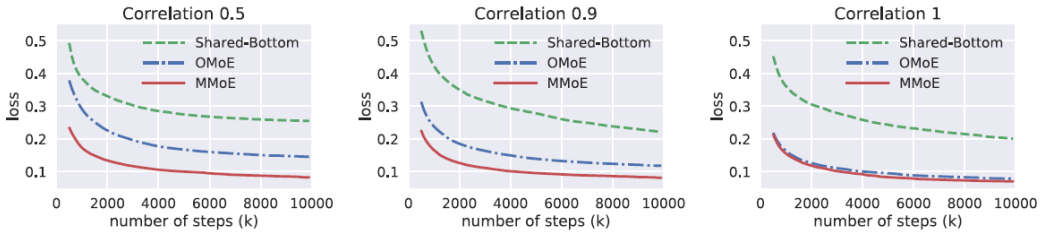
$$y_1 = w_1^T x + \sum_{i=1}^{m} \sin\left(\alpha_i w_1^T x + \beta_i\right) + \epsilon_1 \tag{3}$$

$$y_2 = w_2^T x + \sum_{i=1}^{m} \sin\left(\alpha_i w_2^T x + \beta_i\right) + \epsilon_2 \tag{4}$$

where $\alpha_i, \beta_i, i = 1, 2, \ldots, m$ are given parameters that control the shape of the sinusoidal functions and $\epsilon_1, \epsilon_2 \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, 0.01)$.

(5) Repeat (3) and (4) until enough data are generated.

实验结果



在相关性较低时，MMoE 模型 loss 最低，较大程度解决了由于低任务相关性带来的内部参数冲突,；而在相关性较高时，MMoE 模型与 MoE 模型 loss 相近，表明其结构不会对任务之间的信息共享产生阻碍。

3.2 Census-income Data

Census-income Data 是从 1994 年人口普查数据库中提取的。它包含了 299285 个美国成年人的人口统计信息实例，总共有 40 个特征。在此基础上建立了两个多任务学习问题，并将其中的一些特征作为预测目标。

(1) Task 1: Predict whether the income exceeds $50K;
Task 2: Predict whether this person's marital status is never married.
Absolute Pearson correlation: 0:1768.

(2) Task 1: Predict whether the education level is at least college;
Task 2: Predict whether this person's marital status is never married.
Absolute Pearson correlation: 0:2373.

实验结果

Group 1	AUC/Income		AUC/Marital Stat	
	best	mean	w/ best income	mean
Single-Task	0.9398	0.9337	**0.9933**	0.9922
Shared-Bottom	0.9361	0.9295	0.9915	0.9921
L2-Constrained	0.9389	0.9359	0.9922	0.9918
Cross-Stitch	0.9406	**0.9361**	0.9917	0.9922
Tensor-Factorization	0.7460	0.6765	0.8175	0.8412
OMoE	0.9387	0.9319	0.9928	0.9923
MMoE	**0.9410**	0.9359	0.9926	**0.9927**

Group 2	AUC/Education		AUC/Marital Stat	
	best	mean	w/ best education	mean
Single-Task	0.8843	0.8792	**0.9933**	0.9922
Shared-Bottom	0.8836	0.8813	0.9927	0.9917
L2-Constrained	0.8855	0.8823	0.9923	0.9918
Cross-Stitch	0.8855	0.8819	0.9919	0.9921
Tensor-Factorization	0.7367	0.7256	0.7453	0.7497
OMoE	0.8852	0.8813	0.9915	0.9912
MMoE	**0.8860**	**0.8826**	0.9932	**0.9924**

鉴于这两个组中的任务相关性（大致由 Pearson 相关性度量）都不很强，因此 Shared-Bottom 模型在多任务模型中几乎总是最差的。L2-Constrained 和 Cross-Stitch 都为每个任务提供了单独的模型参数，并增加了如何学习这些参数的约束，因此，其性能要比 Shared-Bottom 更好。但是，对模型参数学习的约束在很大程度上依赖于任务关系，MMoE 使用的调参机制灵活性最好。

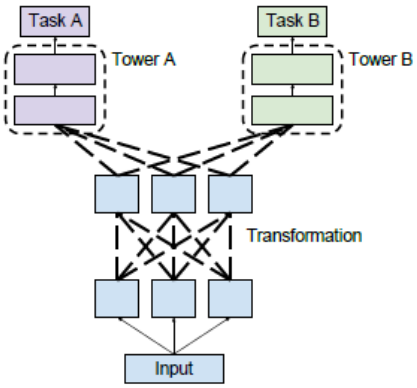
SNR: Sub-Network Routing for Flexible Parameter Sharing in Multi-Task Learning

SNR 模型是 Google 继 MMoE 模型之后于 2019 年提出的又一个多任务学习模型，型同样是主要用来解决多任务之间相关性较低带来的预测质量下降的问题。SNR 主要针对模型内部任务间的参数共享方式进行优化，提出了子网路由（SNR）的新颖框架，以实现更灵活的参数共享，同时保持经典的多任务神经网络模型的计算优势。不同于 MMoE 模型的门控网络，SNR 模型将共享底层隐藏层模块化为多个子网，并通过可学习的潜在变量（latent variables）控制子网连接。该方法提高了多任务模型的准确性，同时又保证了计算效率。

1.模型结构

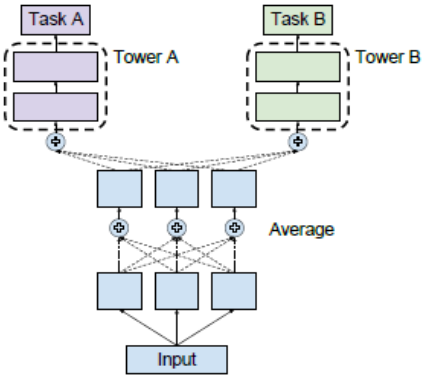
论文中提出了两种 SNR 的结构，分别为 SNR-Trans 和 SNR-Aver。

Sub-Network Routing with Transformation (SNR-Trans)，共享层划分为多个子网，并且子网之间的连接以转换矩阵乘以潜在变量(标量)作为权重。



SNR-Trans

Sub-Network Routing with Average (SNR-Aver)，共享层划分为多个子网，并且子网之间的连接以加权平均的潜在变量(标量)作为权重。



SNR-Aver

在上述结构中，两个子网之间的连接性由二进制变量控制（编码变量），利用子网的多个层和不同的编码变量，实现大量不同的共享架构。

2.建模方法

2.1 Sub-Network Modularization and Routing

假设有两层子网，较低层有 3 个子网，较高层有 2 个子网。设u_1, u_2, u_3为低层子网络的输出，v_1, v_2为高层子网络的输入。那么 SNR-Trans 可以表示为

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} z_{11}W_{11} & z_{12}W_{12} & z_{13}W_{13} \\ z_{21}W_{21} & z_{22}W_{22} & z_{23}W_{23} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

其中W_{ij}为从第j个低层子网到第i个高层子网的变换矩阵，z表示编码变量。

类似地，SNR-Aver 可以表示为

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} z_{11}I_{11} & z_{12}I_{12} & z_{13}I_{13} \\ z_{21}I_{21} & z_{22}I_{22} & z_{23}I_{23} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

其中I_{ij}是所有i, j的单位矩阵。

2.2 Connecting to Manual Tuning and NAS

在 SNR 模型中，如果将z的所有元素都设置为 1，则对应的模型会退化为经典的共享底模型；如果将$z_{11} = z_{22} = 1$并将z的所有其他元素设置为 0，则该模型会退化为两个小的单任务模型。当多任务模型有多个任务和大量隐藏层时，编码变量z的搜索空间将成指数成长：$2^{|z|}$，其中$|z|$表示z中元素的数量。所以在缺乏任务相关性知识时，手动调参效率会非常低。作者受到 NAS 方法的启发，采用灵活的 SNR 框架自动学习子网的连接路由。将z作为参数化分布中的潜在变量，并学习分布参数和模型参数。

2.3 Learning the Architecture with Latent Variables

令$f(\cdot; \boldsymbol{W}, \boldsymbol{z})$是一个由权重$W$和编码变量$z$参数化的神经网络模型，其中编码变量应从参数化的潜在分布$p(z; \pi)$中得出,π为分布参数。给定数据集$\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$，其中$\boldsymbol{x}_i$是样本$i$的特征向量，而$\boldsymbol{y}_i$是包含多个任务标签的矢量。学习编码变量的问题可以将模型参数公式化为优化问题，如下所示：

$$\min_{\boldsymbol{W}, \boldsymbol{\pi}} \boldsymbol{E}_{\boldsymbol{z} \sim p(\boldsymbol{z}; \boldsymbol{\pi})} \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{z}), \boldsymbol{y}_i)$$

L为损失函数。

目标函数关于分布参数π不可微分，文中采用一种松弛方法来平滑目标函数，以便于计算分布参数的梯度。首先找到一个连续随机变量$s \sim q(s; \phi)$，并将编码变量z作为s的 hard-sigmoid 值，即

$$z = g(s) = min(1, max(0, s))$$

则目标函数变为

$$\min_{\boldsymbol{W},\boldsymbol{\pi}} \boldsymbol{E}_{\boldsymbol{s}\sim q(\boldsymbol{s};\boldsymbol{\phi})} \frac{1}{N}\sum_{i=1}^{N} L(f(\boldsymbol{x}_i;\boldsymbol{W},g(\boldsymbol{s})),\boldsymbol{y}_i)$$

然后利用重参数化技巧重构目标函数，以解决反向传播问题。

$$\min_{\boldsymbol{W},\boldsymbol{\pi}} \boldsymbol{E}_{\boldsymbol{\epsilon}\sim r(\boldsymbol{\epsilon})} \frac{1}{N}\sum_{i=1}^{N} L(f(\boldsymbol{x}_i;\boldsymbol{W},g(h(\boldsymbol{\phi},\boldsymbol{\epsilon}))),\boldsymbol{y}_i)$$

$\boldsymbol{\epsilon}$是随机噪声变量，$r(\boldsymbol{\epsilon})$是无参数的噪声分布。

实际应用中，s采用 hard concrete 分布：

$$u \sim U(0,1), s = \text{sigmoid}\left((\log(u) - \log(1-u) + \frac{\log(\alpha)}{\beta}\right)$$
$$\bar{s} = s(\zeta - \gamma) + \gamma, z = min(1, max(\bar{s}, 0))$$

u是均匀随机变量，$\log(\alpha)$是可学习的分布参数，β，γ，ζ是超参数。

2.4 Applying L0 Regularization on Latent Variables

潜在变量z的 L0 约束可表示为：

$$\boldsymbol{E}_{\boldsymbol{z}\sim p(\boldsymbol{z};\boldsymbol{\pi})} \parallel \boldsymbol{z} \parallel_0 = \sum_{i=1}^{|\boldsymbol{z}|} p(z_i = 1;\pi_i)$$

并且z可以松弛为连续随机变量s，则有

$$p(z_i = 1;\pi_i) = 1 - Q(s_i < 0;\phi_i)$$

$Q(\cdot;\phi_i)$是s_i的累积分布函数。

L0 正则化的完整目标函数为

$$E_{\epsilon\sim r(\epsilon)} \frac{1}{N}\sum_{i=1}^{N} L(f(x_i;W,g(h(\phi,\epsilon))),\boldsymbol{y}_i) + \lambda \sum_{j=1}^{|z|} 1 - Q(s_j < 0;\phi_j)$$

3.实验测试

YouTube8M Dataset

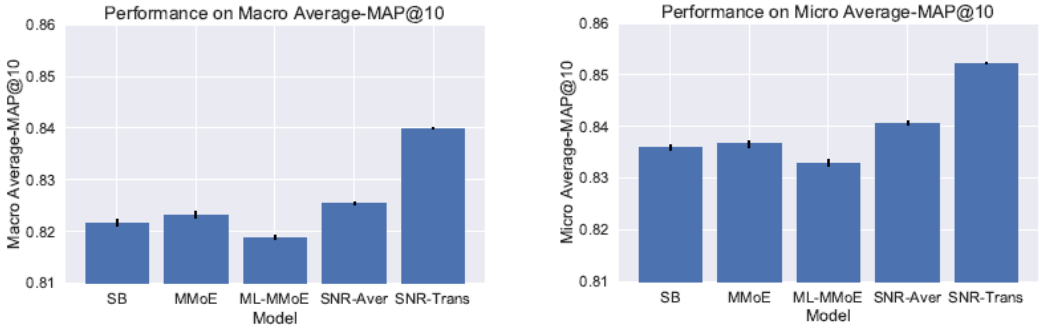
YouTube8M 数据集包含 610 万个 YouTube 视频，每个视频带有多个标签，来自 3000 多个主题实体的词汇表中。主题实体可以进一步分为 24 个顶级主题类别。为了从数据集中创建多任务学习问题，每个顶级主题类别都视为一个单独的预测任务，因此每个任务都是一个多任务标签分类问题。为了确保每个任务的数据量，测试使用了数据量中排名前 16 位的类别。

评估指标

由于YouTube8M数据集中的每个任务都是多标签分类问题，因此使用平均精度（MAP）作为每个任务的预测准确性的度量。由于不同的任务具有不同的样本量，因此计算两种类型的 Average-MAP @ 10 指标：第一种类型直接计算所有任务的 MAP @ 10 得分的平均值，称其为 Macro Average-MAP @ 10；二种类型采用 MAP @ 10 分数的加权平均值，该分数由每个任务中数据示例的数量加权，称之为 Micro Average-MAP @ 10。

实验结果

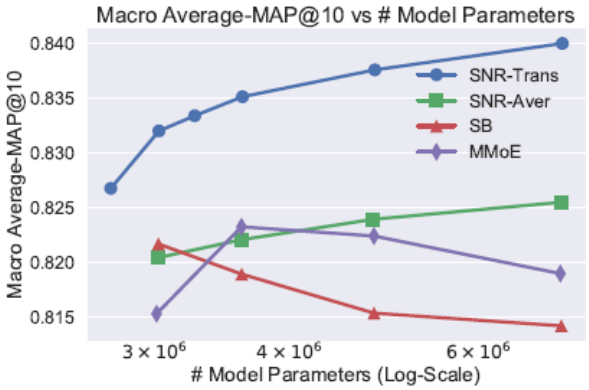
Accuracy of Best-Tuned Models



(a) Macro Average-MAP@10　　(b) Micro Average-MAP@10

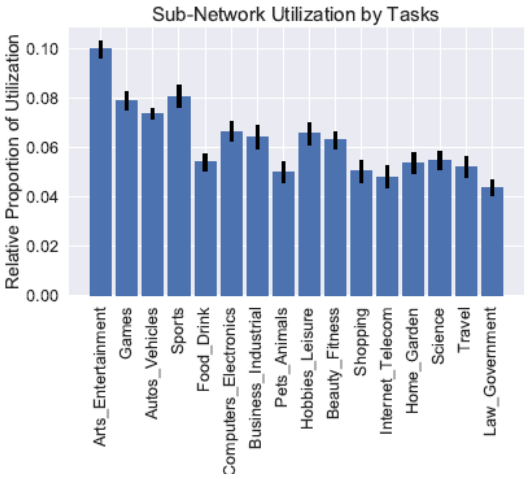
如图所示，SNR-Trans 和 SNR-Aver 模型由于其他模型。ML-MMoE 模型表现最差，可能是由于堆叠结构会导致优化困难。选择两层 experts 之间的门控网络也存在设计困难。

Accuracy vs Model Size



随着模型尺寸增加，基准模型的精度已经达到饱和，而 SNR-Trans 和 SNR-Aver 模型在边界处仍缓慢增长，此结果表明 SNR 能够更好地训练参数量大的模型。

Analysis of Sub-Network Utilization

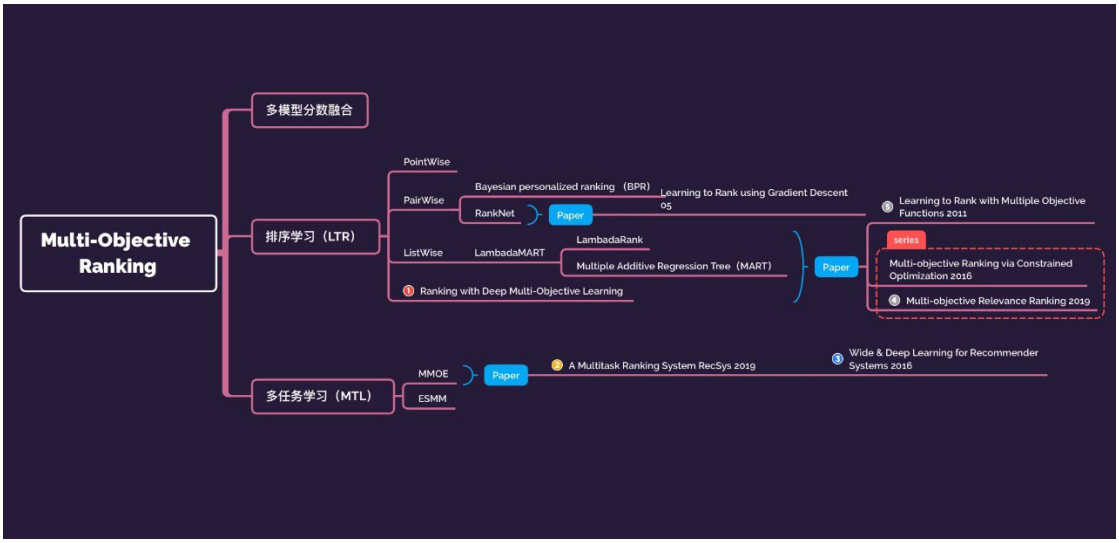


结果显示子网的利用率与任务的样本量成正比。

Survey on MOR & Recommending What Video to Watch Next: A Multitask Ranking System

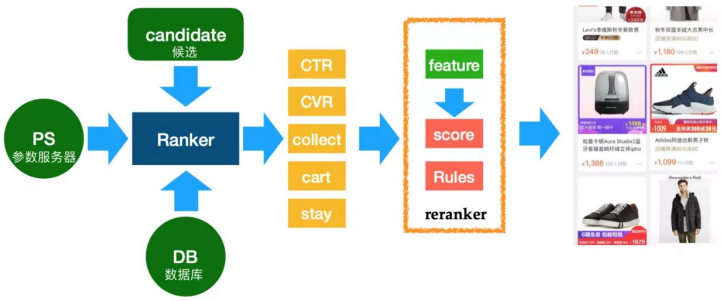
多目标排序（MOR）通常是指有两个或两个以上的目标函数，目的是寻求一种排序使得所有的目标函数都达到最优或满意。 在工业界推荐系统中，大多是基于隐式反馈来进行推荐的，用户对推荐结果的满意度通常依赖很多指标（比如，淘宝基于点击，浏览深度（停留时间），加购，收藏，购买，重复购买，好评等行为的相关指标），在不同的推荐系统、不同时期、不同的产品形态下，这些指标的重要程度或者所代表的意义也会有所不同，如何优化最终推荐列表的顺序来使得众多指标在不同的场景下近可能达到最优或者满意，这就是一个多目标排序问题。

近几年的 MOR 方法总结：



一.多模型分数融合

多模型融合的方式是比较经典传统的做法，每个目标训练一个模型，每个模型算出一个分数，然后根据自身业务的特点，通过某种方式将这些分数综合起来，计算出一个总的分数再进行排序，综合分数的计算通常会根据不同目标的重要性设定相应的参数来调节。



上图流程图展示的是一个工业界很常用的推荐流程，包括召回（Match）、排序（Rank）、重排序（Rerank）、推荐（Recommend）过程。

重排序模块有两个主要作用，一个是**融合**，也就是将排序阶段传递来的多个目标分数进

行融合，形成一个分数，另外一个是**约束**，也就是规制过滤。
分数融合的一种思路是利用一个带参数公式来实现，如下：

$$\text{score} = \text{CTR} * (\alpha + \text{CVR}) * (\beta + \text{price}) * \text{stay}^{a} * \text{cart}^{b} * \text{collect}^{c} * \text{rule\_wight} \ ...$$

二.排序学习（LTR）

排序学习是一个典型的有监督机器学习过程。
排序学习的模型通常分为**单点法**（Pointwise Approach）、**配对法**（Pairwise Approach）和**列表法**（Listwise Approach）三大类，三种方法并不是特定的算法，而是排序学习模型的设计思路，主要区别体现在损失函数（Loss Function）、以及相应的标签标注方式和优化方法的不同。

1、Pointwise

pointwise把排序问题当成一个二分类问题，训练的样本被组织成为一个三元组 $(q_i, c_{i,j}, y_{i,j})$ 。 $y_{i,j}$ 为一个二进制值，表明 $c_{i,j}$ 是否为 q_i 正确回答。我们就可以训练一个二分类网络：$h_\theta(q_i, c_{i,j}) \rightarrow y_{i,j}$ ，其中 $0 \leq y_{i,j} \leq 1$ 。训练的目标就为最小化数据集中所有问题和候选句子对的交叉熵。

在预测阶段，二分类模型 h_θ 被用来排序每一个候选句子，选取top-k的句子作为正确回答。

2、Pairwise

在pairwise方法中排序模型 h_θ 让正确的回答的得分明显高于错误的候选回答。给一个提问，pairwise给定一对候选回答学习并预测哪一个句子才是提问的最佳回答。训练的样例为 (q_i, c_i^+, c_i^-) ,其中 q_i 为提问, c_i^+ 为正确的回答， c_i^- 为候选答案中一个错误的回答。

损失函数为合页损失函数：

$$L = max\left\{0,\ \ m - h_\theta(q_i, c_i^+) + h_\theta(q_i, c_i^-)\right\}$$

其中 m 为边界阀值。如果 $h_\theta(q_i, c_i^+) - h_\theta(q_i, c_i^-) < m$ 损失函数 L 大于0，当满足这个不等式的时候，意味着模型把非正确的回答排在正确答案的上面；如果 L 等于0，模型把正确的回答排在非正确的回答之上。用另一种方式解释就是，如果正确的答案的得分比错误句子的得分之差大于 m （ $h_\theta(q_i, c_i^+) - h_\theta(q_i, c_i^-) \geq m$ ）,总之合页损失函数的目的就是促使正确答案的得分比错误答案的得分大于 m 。和pairwise类似，在预测阶段得分最高的候选答案被当作正确的答案。

3、Listwise

pariwise和pointwise忽视了一个事实就是答案选择就是从一系列候选句子中的预测问题。在listwise中单一训练样本就：query和它的所有候选回答句子。在训练过程中给定提问数据 q_i 和它的一系列候选句子 $C\left(c_{i1}, c_{i2}, \ldots, c_{im}\right)$ 和标签 $Y\left(y_{i1}, y_{i2}, \ldots, y_{im}\right)$,归一化的得分向量 S 通过如下公式计算：

$$Score_j = h_\theta(q_i, c_{ij})$$

$$S = softmax([Score_1, Score_2, \ldots, Score_m])$$

标签的归一化方法为：

$$Y = \frac{Y}{\sum_m^{j=1} y_{ij}}$$

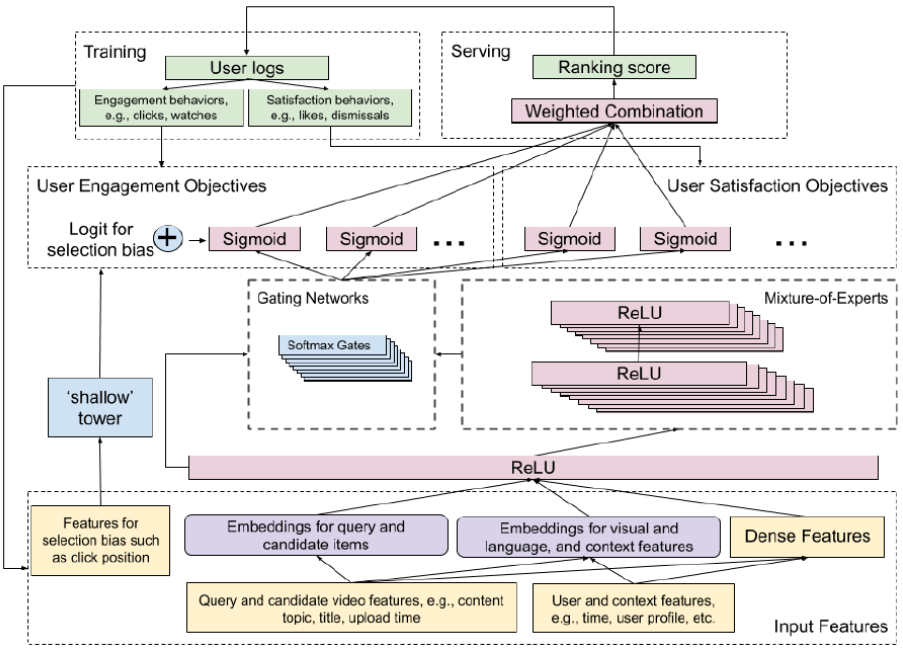
训练的目标可以为最小化 S 和 Y 的KL散度。

三.多任务学习（MTL）

Recommending What Video to Watch Next: A Multitask Ranking System

这篇论文同样是 Google 于 2019 年发表的多任务学习模型，但是更注重于在推荐系统中的实用性。文章介绍了一个大型多目标排名系统，其结构上既采用 MMoE 模型来处理多目标排名问题，又引入 Wide&Deep 模型解决由于位置等因素带来的隐式偏差。总而言之，本文是一篇综述性较强，偏向于推荐系统领域的应用性文章。

3.1 模型结构



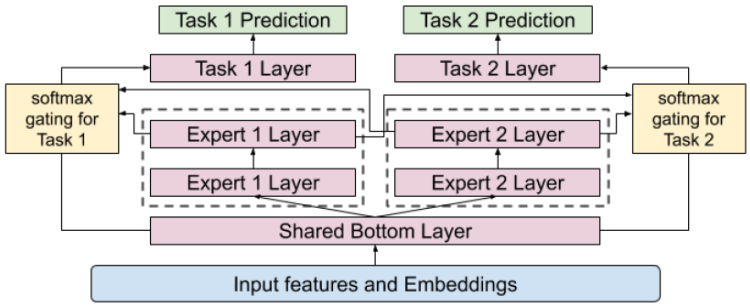
设计与开发现实世界中的大型视频推荐系统又两个主要问题：

- 我们经常需要优化多个可能会彼此冲突的目标。例如，除了观看以外，还有评价、分享等多项指标。
- 系统中经常存在隐式偏差。例如，用户可能只是因为视频在推荐列表的位置靠前才点击观看，而不是因为它是用户最喜欢的视频。

为了解决这些问题，模型通过采用 MMoE 扩展了 Wide&Deep 模型用于多任务学习。另外，它引入了'shallow tower'用来建模和消除选择偏差。

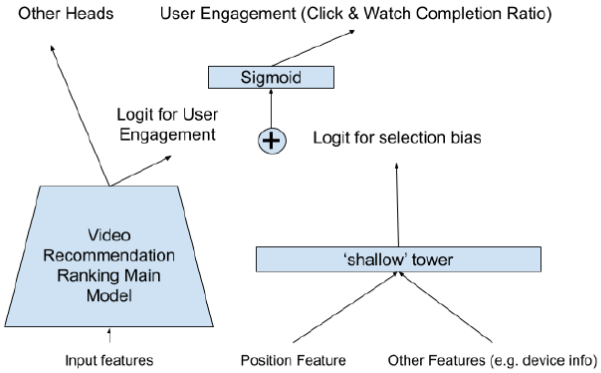
作者将多个目标分为两类：1）参与度目标，例如用户点击次数和推荐视频的参与度；2）满意度目标，例如用户的点赞和评分。为了学习和估计多种类型的用户行为，作者使用 MMoE 自动学习参数以在可能冲突的目标之间共享信息。为建模和减少来自有偏见的训练数据的选择偏见（例如，位置偏见），在主模型左侧引入'shallow tower'用于接受与选择偏差相关的输入（例如，由当前系统确定的排名顺序），并输出标量作为偏差项，用于最终预测主模型。

参与度目标捕获用户的点击和观看等行为。作者将这些行为的预测公式化为两种类型的任务：针对诸如点击之类行为的二分类任务，以及针对观看时长的回归任务。同样，对于满意度目标，作者将与用户满意度有关的行为的预测公式化为二分类任务或回归任务。例如，诸如对视频的点赞之类的行为被表述为二分类任务，诸如评级之类的行为被表述为回归任务。对于二分类任务，计算交叉熵损失。对于回归任务，计算平方损失。一旦确定了多个排名目标及其问题类型，就会为这些预测任务训练一个多任务排名模型。



Multi-gate Mixture-of-Expert Model with one shared bottom layer and separate hidden layers for two tasks.

MMoE 层旨在捕获任务差异，而与共享底模型相比，不需要太多的模型参数。关键思想是用 MoE 层替换共享的 ReLu 层，并为每个任务添加一个单独的门控网络。



Adding a shallow side tower to learn selection bias

训练'shallow tower'，使其有助于选择偏差，例如位置偏差的位置特征，然后将其添加到主模型的最终 logit 中。

3.2 实验测试

在全球最大的视频共享平台之一 YouTube 上进行在线实验。

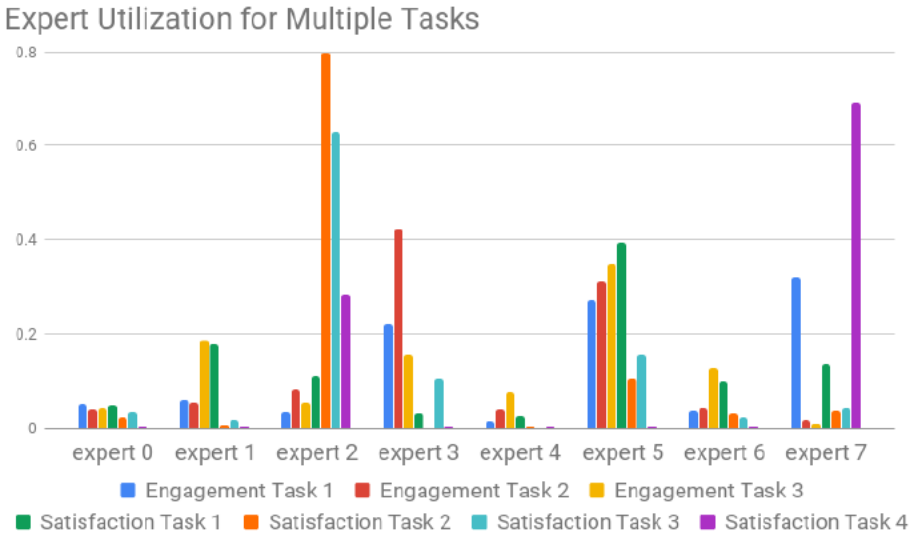
Live Experiment Results.

Model Architecture	Number of Multiplications	Engagement Metric	Satisfaction Metric
Shared-Bottom	3.7M	/	/
Shared-Bottom	6.1M	+0.1%	+ 1.89%
MMoE (4 experts)	3.7M	+0.20%	+ 1.22%
MMoE (8 Experts)	6.1M	+0.45%	+ 3.07%

Table 1: YouTube live experiment results for MMoE.

从表中可以看出，使用相同的模型复杂度，MMoE 显著提高了参与度与满意度指标。

Gating Network Distribution



参与度任务相互之间共享多个 experts，分布更为均匀；满意度任务则倾向于共享一小部分高利用率的 experts。

Analysis of User Implicit Feedback.

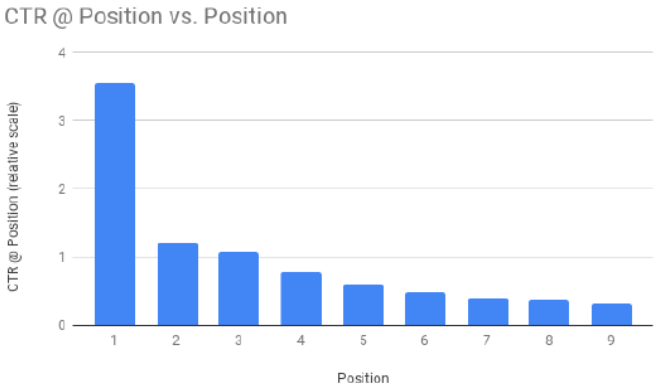


Figure 6: CTR for position 1 to 9.

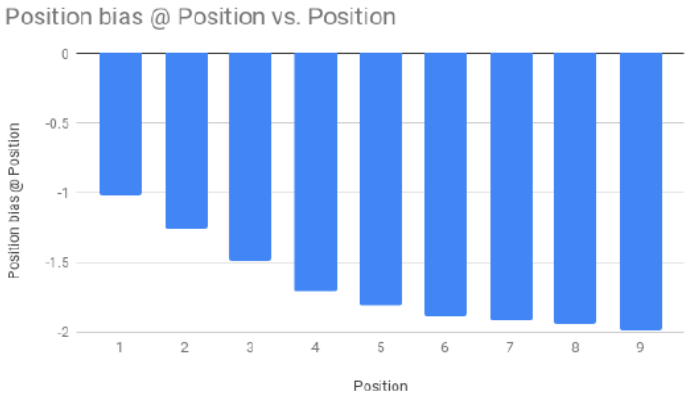


Figure 7: Learned position bias per position.

正如预期的那样，随着位置的降低，CTR 显著降低。靠前位置的点击率较高是由于推荐了更多相关项目和位置偏差的综合效果。

Method	Engagement Metric
Input Feature	-0.07%
Adversarial Loss	+0.01%
Shallow Tower	+0.24%

Table 2: YouTube live experiment results for modeling position bias.

文中提出的方法通过建模和减少位置偏差显着改善了参与度指标。

Progressive Layered Extraction (PLE): A Novel Multi-Task Learning (MTL) Model for Personalized Recommendations

腾讯在2020年提出了PLE模型，是MTL领域最新的模型。现实推荐系统中复杂而竞争的任务相关性，MTL模型经常遭受负传递而导致性能下降的问题。同时，MTL模型存在一个有趣的跷跷板现象，一项任务的执行通常会因伤害其他任务的执行而得到改善。为了解决这些问题，PLE模型明确分离共享组件和特定于任务的组件，并采用渐进式路由机制逐步提取和分离更深层的语义知识，从而提高了通用表示中联合表示学习和跨任务的信息路由的效率。PLE是由多层CGC模型叠加形成。

1.模型结构

1.1 Customized Gate Control(CGC)

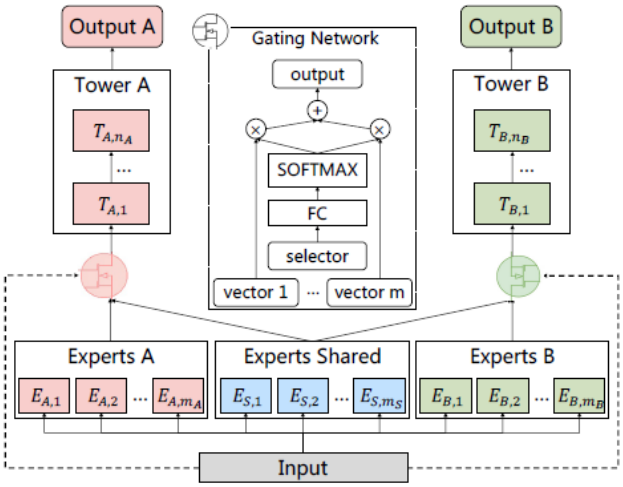


Figure 4: Customized Gate Control (CGC) Model

在CGC模型底部有一些experts模块，在顶部有一些特定于任务的tower网络。每个experts模块由多个子网组成，每个模块中的expert数量是需要调整的超参数。类似地，tower网络也是具有宽度和深度作为超参数的多层网络。具体来说，CGC中的shared experts负责学习共享模式，而特定任务的模式则由task-specific experts提取。每个tower网络都吸收共享专家及其特定任务专家的知识，这意味着shared experts的参数受所有任务的影响，而task-specific experts的参数仅受相应特定任务的影响。在CGC中，shared experts和task-specific experts通过门控网络进行组合，以进行选择性融合。如图所示，门控网络的结构基于单层前馈网络，其中SoftMax作为激活函数。

任务k的门控网络输出可以表示为：

$$g^k(x) = w^k(x)S^k(x)$$

x是输入向量，$w^k(x)$是加权函数，用于计算任务k的权重向量。$w^k(x)$通过线性变换和Softmax层来计算：

$$w^k(x) = \text{Softmax}\left(W_g^k x\right)$$

$W_g^k \in R^{(m_k+m_s)\times d}$是参数矩阵，$m_s$和$m_k$分别是shared experts的数量和任务$k$的specific experts数量，$d$是输入向量的维数。$S^k(x)$是由所有被选择的向量组成的选择矩阵，包括shared experts和task-specific experts：

$$S^k(x) = \left[E_{(k,1)}^T, E_{(k,2)}^T, \ldots, E_{(k,m_k)}^T, E_{(s,1)}^T, E_{(s,2)}^T, \ldots, E_{(s,m_s)}^T\right]^T$$

最终，任务k的预测值为：

$$y^k(x) = t^k(g^k(x))$$

t^k是任务k的tower网络。

与MMOE相比，CGC消除了任务tower网络与其他任务task-specific experts之间的连接，使不同类型的experts可以集中精力有效地学习不同的知识，而不会受到干扰。结合选通网络的优势，基于输入动态融合表示，CGC可以在任务之间实现更灵活的平衡，并更好地处理任务冲突和样本相关性。

1.2 Progressive Layered Extraction

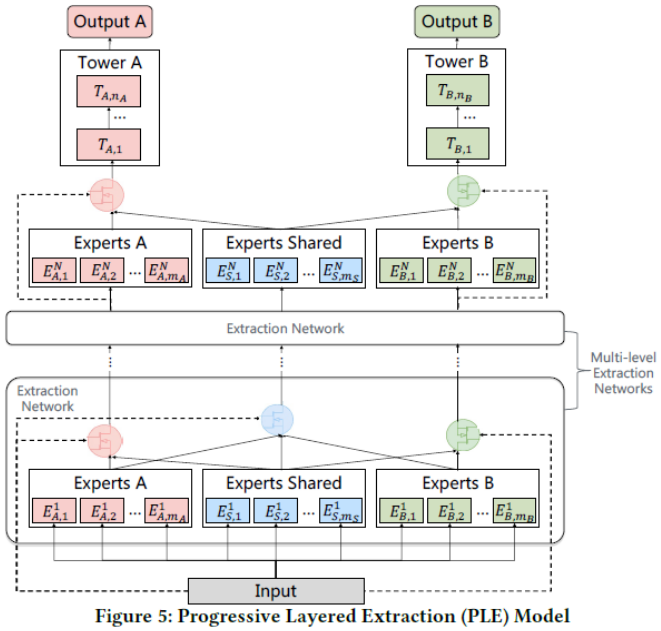


Figure 5: Progressive Layered Extraction (PLE) Model

如图所示，PLE中有多级提取网络以提取更高级别的共享信息。除了针对task-specific experts的gate外，提取网络还为shared experts采用gate，以合并该层中所有experts的知识。因此，PLE中不同任务的参数没有像CGC那样在早期层中完全分离，而是在上层中逐渐分离。高级别提取网络中的门控网络将低级别提取网络中的门的融合结果作为输入，而不是原始输入，因为它可以为选择高级experts中提取的抽象知识提供更好的信息。 PLE中的加权函数，选择矩阵和选通网络的计算与CGC中的相同。

具体地，PLE的第j层extraction网络中的任务k的门控网络的公式为：

$$g^{k,j}(x) = w^{k,j}\left(g^{k,j-1}(x)\right)S^{k,j}(x)$$

$w^{k,j}$是以$g^{k,j-1}$作为输入，关于任务k的加权函数。$S^{k,j}$是第j层extraction网络关于任务k的选择矩阵。值得注意的是，PLE中shared模块的选择矩阵与task-specific模块略有不同，因为它由该层中的所有shared experts和task-specific experts组成。计算所有选通网络和experts后，我们可以获得PLE中任务k的最终预测：

$$y^k(x) = t^k(g^{k,N}(x))$$

PLE采用渐进式分离路由，以吸收所有下层experts的信息，提取更高级别的共享知识，并逐步分离特定于任务的参数。逐步分离的过程类似于从化合物中提取所需化学产物的过程。在PLE的知识提取和转换过程中，较低层的表示被联合提取/聚合并路由到较高层的shared experts处，获得共享的知识并逐步分布到特定的tower层，从而实现更高效，更灵活联合代表学习和分享。

1.3 Joint Loss Optimization for MTL

在本文中，作者优化了联合损失函数，以解决实际推荐系统中遇到的两个关键问题。第一个问题是由于顺序的用户操作而导致的异构样本空间。例如，用户只能在单击某个项目后共享或评论该项目，这将导致不同任务的不同样本空间，如图所示。

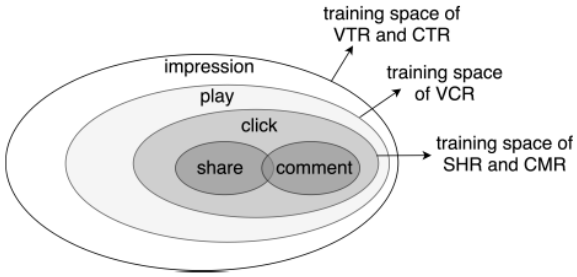


Figure 6: Training Space of Different tasks

要共同训练这些任务，考虑所有任务的样本空间的整体训练集，并在计算每个任务的损失时忽略其自身样本空间之外的样本：

$$L_k(\theta_k, \theta_s) = \frac{1}{\sum_i \delta_k^i} \sum_i \delta_k^i \text{loss}_k\left(\hat{y}_k^i(\theta_k, \theta_s), y_k^i\right)$$

loss_k是任务k根据预测值$\hat{y}_k^i$和真实值y_k^i计算的样本损失，$\delta_k^i \in \{0,1\}$表示样本i是否存在于任务k的样本空间中。

第二个问题是MTL模型的性能对训练过程中损失权重的选择很敏感，因为它确定了每个任务对联合损失的相对重要性。在实践中，观察到每个任务在不同的训练阶段可能具有不同的重要性。因此，将每个任务的损失权重视为动态权重，而不是静态权重。首先，为任务k设置初始损失权重$\omega_{k,0}$，然后根据更新率γ_k对其每一步进行更新：

$$\omega_k^{(t)} = \omega_{k,0} \times \gamma_k^t$$

t是训练的轮次，$\omega_{k,0}$和γ_k是模型的超参数。

2.实验测试

2.1 Evaluation on the Video Recommender System in Tencent

通过连续8天从提供腾讯新闻的视频推荐系统中对用户日志进行采样来收集行业数据集。有469.26万用户，268.2万个视频和9.95亿个样本。如前所述，VCR，CTR，VTR，SHR（共享率）和CMR（评论率）是在数据集中模拟用户偏好的任务。

在实验中，VCR预测是通过MSE损失进行训练和评估的回归任务，对其他动作进行建模的任务都是通过交叉熵损失进行训练并通过AUC进行评估的二分类任务。前7天的样本用于训练，其余样本用作测试集。

2.1.1 Evaluation on Tasks with Complex Correlation

VCR预测是一项经过MSE损失训练的回归任务，可以预测每次观看的完成率。VTR预测是一种经过交叉熵损失训练的二分类任务，用于预测有效观看的概率，该有效观看的概率定义为超过观看时间特定阈值的播放动作。VCR和VTR之间的相关模式很复杂，VTR的标签是播放动作和VCR的组合，因为只有观看时间超过阈值的播放动作才被视为有效观看。

除了AUC和MSE等通用评估指标外，文中还定义了**MTL Gain**指标，以定量评估针对某项任务的多任务学习优于单任务模型的优势。如下公式所示，对于给定的任务组和MTL模型q，将任务A上q的MTL增益定义为任务A在具有相同网络结构和训练样本的情况下相对于单任务模型的MTL模型q的性能提升。

$$\text{MTL gain} = \begin{cases} M_{MTL} - M_{\text{single}}, & M \text{ is a positive metric} \\ M_{\text{single}} - M_{MTL}, & M \text{ is a negative metric} \end{cases}$$

Result

Table 1: Performance on VTR/VCR Task Group

Models	AUC VTR	MSE VCR	MTL Gain VTR	MTL Gain VCR
Single-Task	0.6787	0.1321	-	-
Hard Parameter Sharing	0.6740	0.1320	-0.0047	+1.8E-5
Asymmetric Sharing	0.6823	0.1346	+0.0036	-0.0025
Cross-Stitch	0.6740	0.1320	-0.0047	+1.6E-5
Sluice Network	0.6825	0.1329	+0.0038	-0.0008
Customized Sharing	0.6780	0.1318	-0.0007	+0.0002
MMOE	0.6803	0.1319	+0.0016	+0.0001
ML-MMOE	0.6815	0.1329	+0.0028	-0.0009
CGC	**0.6832**	0.1320	**+0.0045**	+3.5E-5
PLE	**0.6831**	**0.1307**	**+0.0044**	**+0.0013**

结果表明，CGC和PLE在VTR中明显优于所有基准模型。由于VTR和VCR之间存在复杂的相关性，我们可以用锯齿形的灰色分布清楚地观察到跷跷板现象，有些模型改善了VCR但损害了VTR，而有些模型改善了VTR但损害了VCR。具体地说，MMOE比单任务改进了两个任务，但改进并不显着，而ML-MMOE改进了VTR但损害了VCR。与MMOE和ML-MMOE相比，CGC可以大大改善VTR，同时也可以稍微改善VCR。最后，PLE取得最佳的VCR MSE和最佳的VTR AUC之一，以类似的速度收敛，并在上述模型上取得了显著改进。

2.1.2 Evaluation on Tasks with Normal Correlation

Table 2: Performance on CTR/VCR Task Group

Models	AUC CTR	MSE VCR	MTL Gain CTR	MTL Gain VCR
Single-Task	0.7379	0.1179	-	-
Cross-Stitch	0.7220	0.1158	-0.0158	+0.0021
Sluice Network	0.7382	0.1157	+0.0004	+0.0021
MMOE	0.7382	0.1175	+0.0003	+0.0004
ML-MMOE	0.7378	0.1169	-0.0001	+0.0010
CGC	0.7398	0.1155	+0.0020	+0.0023
PLE	**0.7406**	**0.1150**	**+0.0027**	**+0.0029**

尽管CGC和PLE在具有非常复杂的相关性的任务上表现良好，但是作者在具有正常相关性模式的CTR / VCR的一般任务组上进一步验证了它们的普遍性。在这种情况下，CGC和PLE在两个任务上的表现都显着优于所有SOTA模型，并具有出色的MTL增益，这证明CGC和PLE的好处是普遍的，实现了更好的共享学习效率，并在整个相关性范围内始终提供性能改进增益，不仅适用于难以协作的复杂相关性任务，还适用于一般相关性的任务。

2.1.3 Evaluation with Multiple Tasks

Table 4: MTL gain of CGC and PLE on Multiple Tasks

Task Group	Models	MTL Gain VTR	MTL Gain VCR	MTL Gain SHR	MTL Gain CMR
VTR+VCR +SHR	CGC	**+0.0131**	**+0.0019**	-0.0001	-
	PLE	**+0.0132**	**+0.0036**	**+0.0013**	-
VTR+VCR +CMR	CGC	**+0.0180**	**+0.0012**	-	+0.0000
	PLE	**+0.0197**	**+0.0033**	-	+0.0001
VTR+VCR +SHR+CMR	CGC	**+0.0097**	**+0.0016**	+0.0008	**+0.0012**
	PLE	**+0.0128**	**+0.0017**	**+0.0058**	**+0.0080**

最后，作者探索了CGC和PLE在具有更多任务的更具挑战性的方案中的可扩展性。如下表所示，CGC和PLE几乎在所有任务组的所有任务上都比单任务模型显着改善。这表明CGC和PLE可以防止带有两个以上任务的一般情况下的负迁移和跷跷板现象。在所有情况下，PLE均明显优于CGC。因此，PLE在提高不同规模任务组的共享学习效率方面显示出更大的

2.2 Evaluation on Public Datasets

2.2.1 Synthetic Data

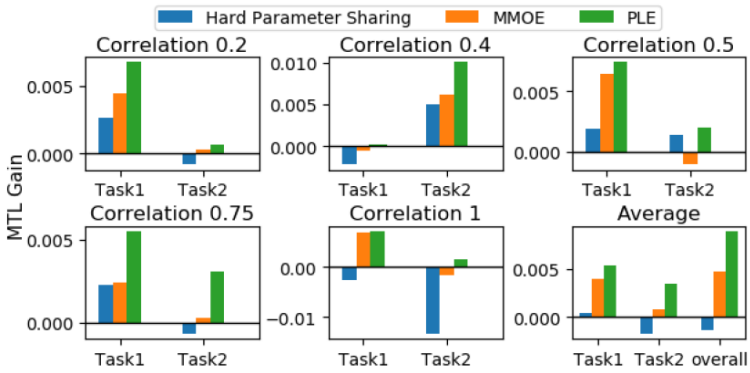


Figure 7: MTL gain on Synthetic Data

如图所示，硬参数共享和MMOE有时会出现跷跷板现象，并且在两个任务之间失去平衡。相反，PLE始终在不同相关性的两项任务中均表现最佳，并且平均MTL收益比MMOE增长87.2%。

2.2.2 Census-income Dataset & Ali-CCP Dataset

Models	**Census-income** Task1		**Census-income** Task2		**Ali-CCP** CTR		**Ali-CCP** CVR	
	AUC	MTL Gain	AUC	MTL Gain	AUC	MTL Gain	AUC	MTL Gain
Single-Task	0.9445	-	0.9923	-	0.6088	-	0.6040	-
MMOE	0.9393	+0.0048	0.9928	+0.0005	0.6094	+0.0006	0.5738	-0.0302
PLE	**0.9522**	**+0.0078**	**0.9945**	**+0.0022**	**0.6112**	**+0.0024**	**0.6097**	**+0.0057**

如表所示，Ali-CCP和Census-income Dataset的结果，PLE消除了跷跷板现象，并且在两个任务上均始终胜过单任务模型和MMOE。

2.3 Expert Utilization Analysis

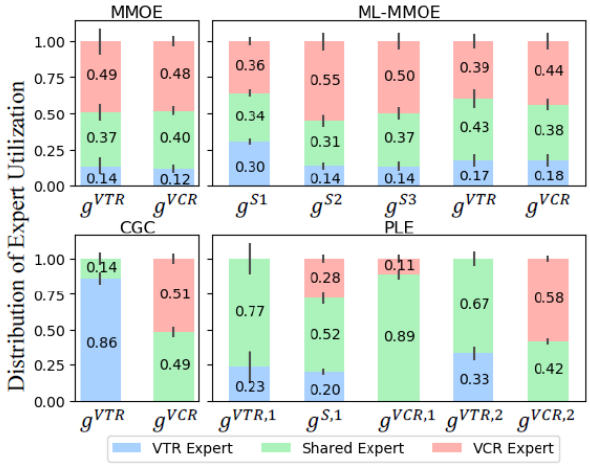


Figure 8: Expert Utilization in Gate-Based Models

结果表明，VTR和VCR组合了CGC中权重明显不同的experts，而MMOE中的权重非常相似，这表明CGC精心设计的结构有助于更好地区分不同experts。与CGC相比，PLE中的shared experts对tower网络的输入有更大的影响，尤其是对于VTR任务。PLE比CGC更好的事实表明了共享的更高级的更深层表示的价值。

Reproduce ML-MMoE

本文中 ML-MMoE 的复现基于 Alvin Deng 实现的 MMoE 模型。具体源码见 https://github.com/Ritchiegit/Reproduction-of-PLE-CGC/blob/main/bottom_model/ml_mmoe.py
为了方便解释 ML-MMoE 模型的结构和运行方式，以下面给定参数为例，进行说明。

模型参数

num_features = 100,	特征维数
units=16,	每个 expert 的单元数
num_experts=8,	experts 个数
num_tasks=2,	任务数
num_layers=3,	层数

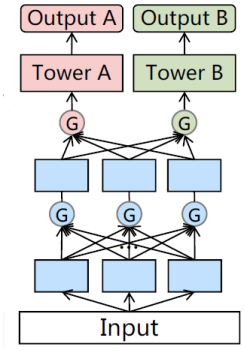
外部参数

inputs = [None, 100]	输入向量形状，（None 表示批次，这里不固定）

本文使用 keras 的自定义层实现 ML-MMoE，根据官方文档，需要完成`__init__()`，用于初始化层；`build()`，用于定义权重；`call()`编写层的功能逻辑。`compute_output_shape()`定义形状变化的逻辑。重点说明`build()`与`call()`部分。

1）build()

在 ML-MMoE 模型中，除第一层 experts 以外，其余 experts 的输入向量均为上一层 gate 的输出，即输入向量的第一维均为 units=16（以文章开头给定参数为例，下文所有参数同理）。第一层 experts 的输入则为原始输入向量，第一维为 num_feature=100。



ML-MMOE

所以，第一层 expert kernels 与其他层要分开定义：

```
"""first expert """
first_expert_kernels
= [num_features, units, num_experts]
= [100, 16, 8]
```

"""expert """
expert_kernels
= [units, units, num_experts, num_layers - 1]
= [16, 16, 8, 2]

而对 gate 而言，除第一层外为原始输入向量外，每一层 gate 的输入向量均为上一层 gate 的输出，所以，第一层 gate kernels 第一维为 num_features=100，而其余层 gate kernels 第一维为 units=16；而且，除了最后一层 gate 的第二维为 num_tasks=2，其余层 gate 的第二维均为 num_experts=8。综上，gate 的定义要将第一层与最后一层分开定义：

"""first gate"""
first_gate_kernels
= [num_features, num_experts]
= [100, 8]
"""gate"""
gate_kernels
= [units, num_experts, num_layers - 2]
= [16, 8, 1]

"""last gate"""
last_gate_kernels
= [units, num_tasks]
= [16, 2]

2）call()

在 ML-MMoE 模型中，按照之前在 build()中定义的权重，我们在 call()函数中也需要据此将模型划分为第一层、中间层和最后一层。（本文仅注重于向量形状的变换，所以省略了激活函数部分，具体实现请见源码）

for j in range(num_layers):

"""first layer"""
expert_outputs #expert 的输出
= K.tf.tensordot(a=inputs,b= first_expert_kernels, axes=1)
= [None, 100] * [100, 16, 8] = [None, 16, 8]

gate_output = K.dot(x=inputs, y=first_gate_kernel) = [None, 100] * [100, 8] = [None, 8]
#一个 gate 的输出

gate_outputs = list(gate_output)*8 =list([None, 8] * 8) #共计 8 个 gate，以列表存储其输出

for gate_output in gate_outputs: #总共有 8 个 gate，每个 gate 产生 1 个输出

expanded_gate_output = K.expand_dims(gate_output, axis=1) #变换形状
= [None, 1, 8]

```
        weighted_expert_output
        =expert_outputs * K.repeat_elements(expanded_gate_output, units, axis=1)
        =[None, 16, 8]   .*   [None, 16, 8] (对应元素乘)
        =[None, 16, 8]                  #根据 gate 的输出对 experts 的输出进行加权
        inner_outputs.append(K.sum(weighted_expert_output, axis=2))
        =list([None, 16] * 8)               #将每个 gate 的输出在 axis=2 求和，即把加权后的 experts 输出相加，最后以列表形式将 8 个 gate 的输出存储

"""mid layer"""
inputs = inner_outputs = [None, 16]
for k in range(num_experts):
    expert_outputs_list.append(K.tf.tensordot(a=inputs[k], b=expert_kernels[:,:,k,j], axes=1))
#这里是因为在中间层部分，每个 expert 把与其直接相连的 gate 的输出作为输入，即一个 expert 对应一个 gate，所以需要利用切片操作进行划分

expert_outputs
= K.stack(expert_outputs_list,axis=2)
= [None, 16 , 8]
for index, gate_kernel in enumerate(self.gate_kernels):
    gate_output = K.dot(x=inputs[index], y=gate_kernel[:,:,j - 1])
                = [None, 16] * [16, 8] = [None, 8]
gate_outputs.append(gate_output)
gate_outputs = list(gate_output)*8 =([None, 8] * 8)

weighted_expert_output
=expert_outputs * K.repeat_elements(expanded_gate_output, units, axis=1)
=[None, 16, 8]   .*   [None, 16, 8] (对应元素乘)
=[None, 16, 8]

inner_outputs.append(K.sum(weighted_expert_output, axis=2))
=list([None, 16] * 8)

"""last layer"""
inputs = inner_outputs = [None, 16]
for k in range(num_experts):
    expert_outputs_list.append(K.tf.tensordot(a=inputs[k], b=expert_kernels[:,:,k,j], axes=1))
#与 mid layer 蓝色部分相同
expert_outputs
= K.stack(expert_outputs_list,axis=2)
= [None, 16 , 8]

for index, last_gate_kernel in enumerate(self.last_gate_kernels):
    last_gate_output = K.dot(x=inputs[index],y=last_gate_kernel)
```

= [None, 16] * [16, 2] = [None, 2]

#这里是因为最后一层的输出将传递给 task 的 tower 层，所以第二维=num_tasks

for gate_output in gate_outputs:

expanded_gate_output = [None, 1, 8]

weighted_expert_output

=expert_outputs * K.repeat_elements(expanded_gate_output, units, axis=1)

=[None, 16, 8] .* [None, 16, 8] (对应元素乘)

=[None, 16, 8]

final_outputs.append(K.sum(weighted_expert_output, axis=2))

=list([None, 16] * 2)

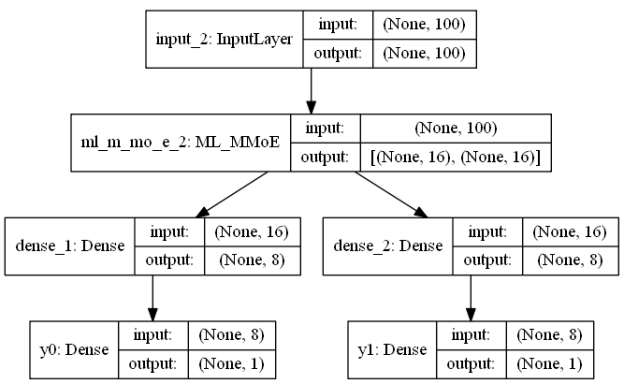
将实现的 ML-MMoE 层融入整个模型

具体源码见 Line 388~420

利用 model.summary()查看模型结构

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
==================================================================================================
input_2 (InputLayer)         (None, 100)               0
__________________________________________________________________________________________________
ml_m_mo_e_2 (ML_MMoE)        [(None, 16), (None,  24576
input_2[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)              (None, 8)                 136
ml_m_mo_e_2[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)              (None, 8)                 136
ml_m_mo_e_2[0][1]
__________________________________________________________________________________________________
y0 (Dense)                   (None, 1)                 9
dense_1[0][0]
__________________________________________________________________________________________________
y1 (Dense)                   (None, 1)                 9
dense_2[0][0]
==================================================================================================
Total params: 24,866
Trainable params: 24,866
Non-trainable params: 0
```

利用 plot_model ()将模型可视化



Reproduce PLE

PLE模型与ML-MMoE在实现上是十分相似的，主要区别在于如何划分shared experts和task-specific experts。本文利用掩码，对gate的输出进行屏蔽，除shared和其专属expert外都为0。

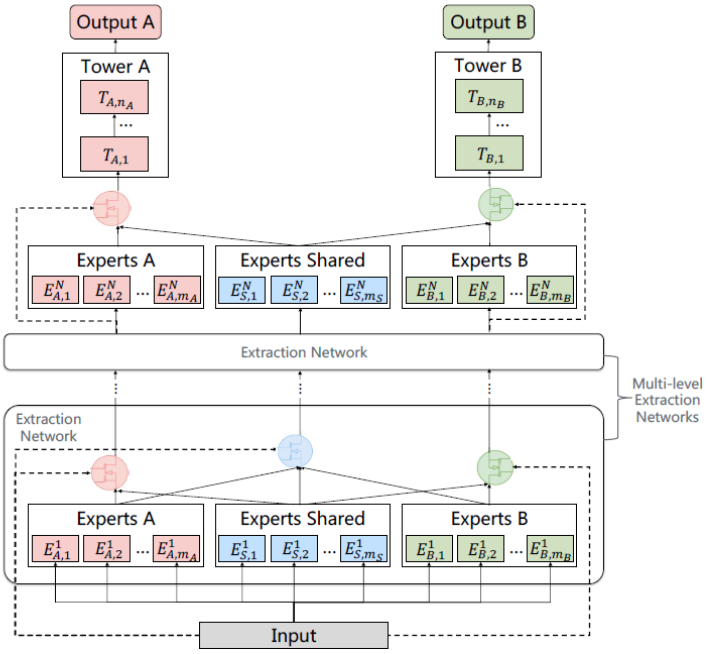
模型参数

num_features = 100,	特征维数
units=16,	每个 expert 的单元数
num_shared_experts=4,	shared experts 个数
num_task_experts_list=[2,2]	task specific experts 个数
num_tasks=2,	任务数
num_layers=3,	层数

外部参数

inputs = [None, 100]	输入向量形状，（None 表示批次，这里不固定）

由于PLE与ML-MMoE很相似，这里仅介绍二者区别之处。



1）build()

基本与ML-MMoE一致，主要区别在于加入掩码部分，总共有8个experts，若gate接受某个expert的输出，则将其掩码对应位置置1，否则置0；最后返回[1, 8]的二进制向量。除最后一层外，一个gate同样对应一个expert。

```
for i in range(self.num_tasks):
            each_mask_of_gate_np = np.zeros((1, self.num_experts))
            each_mask_of_gate_np[0, 0:self.num_shared_experts] = 1  # remain the shared task related experts
            # print("expert_start_idx", expert_start_idx)
```

```
            expert_end_idx = expert_start_idx + self.num_task_experts_list[i]

            # print("expert_end_idx", expert_end_idx)
            each_mask_of_gate_np[0, expert_start_idx: expert_end_idx] = 1  # remain the task related experts
            expert_start_idx = expert_end_idx

            each_mask_of_gate = K.constant(each_mask_of_gate_np)
            # print("each_mask_of_gate", each_mask_of_gate)
            for j in range(self.num_task_experts_list[i]):
                self.mask_of_gates.append(each_mask_of_gate)
                self.mask_of_gates_np.append(each_mask_of_gate_np)
                print(each_mask_of_gate_np)
        for j in range(self.num_shared_experts):
            self.mask_of_gates.append(K.constant(np.ones((1, self.num_experts)))) #shared related experts
            self.mask_of_gates_np.append(np.ones((1, self.num_experts)))
            print((np.ones((1, self.num_experts))))
        print("self.mask_of_gates", self.mask_of_gates)
```

下图为mask矩阵：

```
[[1. 1. 1. 1. 1. 1. 0. 0.]]
[[1. 1. 1. 1. 1. 1. 0. 0.]]
[[1. 1. 1. 1. 0. 0. 1. 1.]]
[[1. 1. 1. 1. 0. 0. 1. 1.]]
[[1. 1. 1. 1. 1. 1. 1. 1.]]
[[1. 1. 1. 1. 1. 1. 1. 1.]]
[[1. 1. 1. 1. 1. 1. 1. 1.]]
[[1. 1. 1. 1. 1. 1. 1. 1.]]
```

因为除最后一层外，每层都有8个gate，所以共有8个[1, 8]的选择向量。横向看，每个选择向量的前4列为shared experts，所以每一个gate的mask前4个均为1，表示连通；纵向看，对于前两行，是针对任务1 task specific experts的gate，所以只有shared experts和任务1的task specific experts为1，剩余为0；之后两行是针对任务2 task specific experts的gate，同理；最后四行为shared experts的gate，所以连通全部的experts。

2）call()

大体上同样与 ML-MMoE 相似，主要是在每个 gate 的输出之后，还要与对应掩码进行乘积运算。

```
for index, gate_kernel in enumerate(self.gate_kernels):
    gate_output = K.dot(x=inputs[index], y=gate_kernel[:,:,j - 1])
    # Add the bias term to the gate weights if necessary
    if self.use_gate_bias:
        gate_output = K.bias_add(x=gate_output, bias=self.gate_bias[index][:,j-1])
    gate_output = self.gate_activation(gate_output)

```

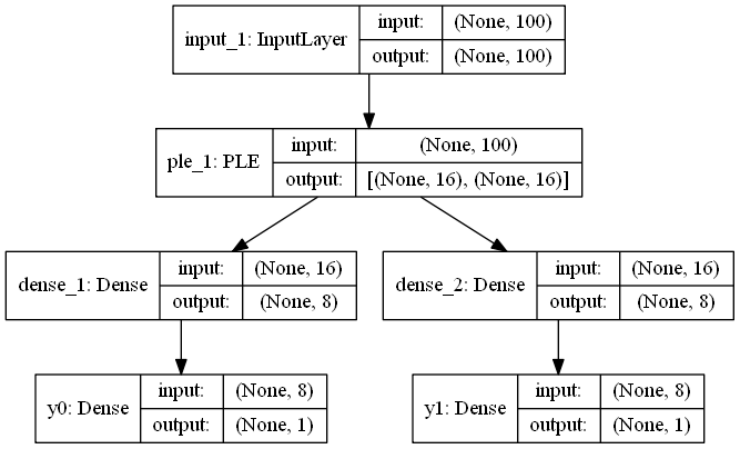
```
        ######################### use gate ############################
        # gate_output = gate_output * mask
        # 这里可以添加掩码 mask 除 shared 和其专属 expert 外都为 0
        gate_output = gate_output * self.mask_of_gates[index]
        ######################### end ################################
        gate_outputs.append(gate_output)
```

将实现的 PLE 层融入整个模型

利用 model.summary()查看模型结构

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
==================================================================================================
input_1 (InputLayer)         (None, 100)               0
_________________________________________________________________
ple_1 (PLE)                  [(None, 16), (None,  25104
input_1[0][0]
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 136
ple_1[0][0]
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 136
ple_1[0][1]
_________________________________________________________________
y0 (Dense)                   (None, 1)                 9
dense_1[0][0]
_________________________________________________________________
y1 (Dense)                   (None, 1)                 9
dense_2[0][0]
==================================================================================================
Total params: 25,394
Trainable params: 25,394
Non-trainable params: 0
_________________________________________________________________
```

利用plot_model ()将模型可视化

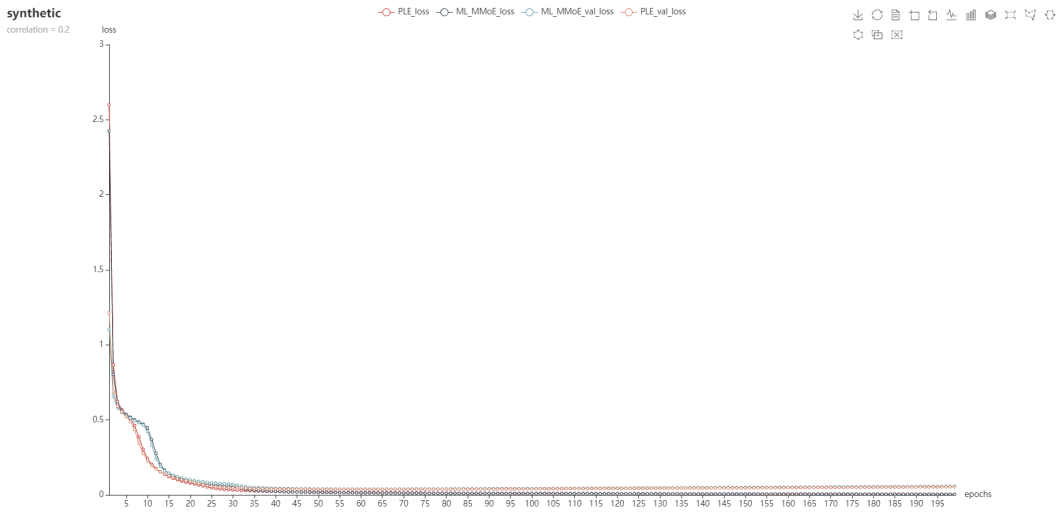


Experiment On Synthetic Data

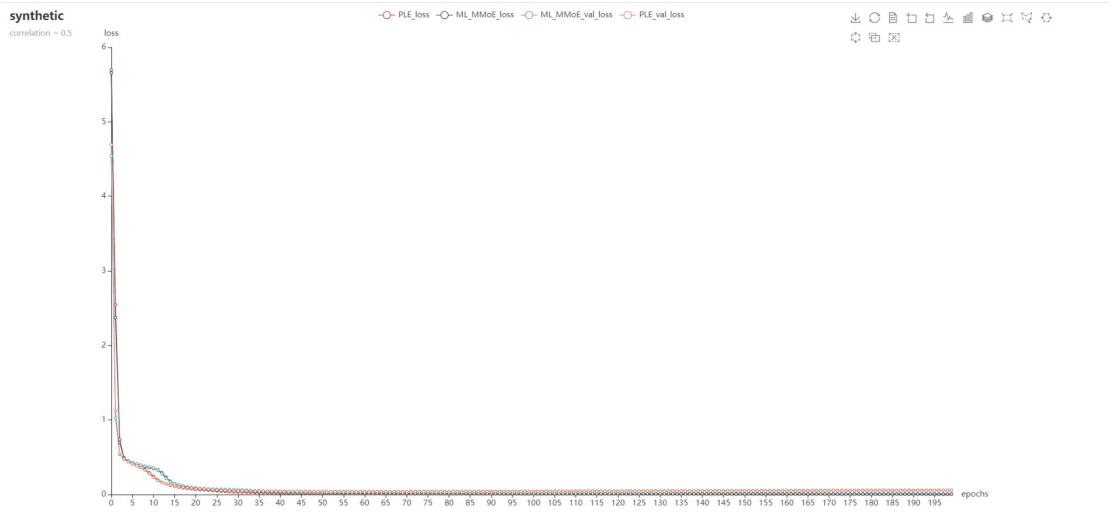
ML-MMoE 与 PLE 模型的比较

在 synthetic 数据集上，epoch 设置为 200，共生成 10 组随机数据。
c=0.3, m=5, correlation 分别取 0.2 和 0.5。

实验结果



详见loss_line_on_synthetic_correlation=0.2



详见loss_line_on_synthetic_correlation=0.5

结论：

当correlation=0.5,epoch=199时， PLE_loss = 0.004756,MMoE_loss=0.004470, PLE_val_loss =0.05511, MMoE_val_loss=0.05808;

当correlation=0.2,epoch=199时,PLE_loss = 0.004910, MMoE_loss=0.004461,
PLE_val_loss =0.05417, MMoE_val_loss=0.05730;
横向比较，correlation较低时，val_loss较小，PLE_loss变化较大，可能是由于knowledge未被充分共享;
纵向比较，PLE模型比ML-MMoE模型泛化效果更好。