基于不同方式的多任务推荐系统模型MMOE的性能验证

吴锦明 202421090316

一MMOE的基本介绍

1.1摘要

MMOE全称[Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts](https://link.zhihu.com/?target=https%3A//www.kdd.org/kdd2018/accepted-papers/view/modeling-task-relationships-in-multi-task-learning-with-multi-gate-mixture-)，于2018年8月发表在数据挖掘领域的顶会KDD，是一篇多任务学习领域的论文。

本文将基于MMOE这个多任务模型，在UCI census income人口数据集上，针对每周工作时长以及工资这两个特征分别设置回归任务和二分类任务，利用K折、留出、自助三种方式划分数据集，最后通过tensorboard来可视化AUC，MAE，MSE，Loss等性能评估指标。本文将在第一章中介绍MMOE的基本原理和网络架构，第二章中从代码的角度，给出MMOE的实现和调用，第三章中使用K折等方式划分数据集以及训练，并对比不同划分方式带来测试结果的差异。

1.2多任务学习的早期动机

MMOE是应用在多任务推荐系统的模型。多任务学习的早期动机，来源于统计学中的stein悖论——即使在高斯随机变量是独立的情况下，使用所有高斯随机变量的样本来估计三个或更多高斯随机变量的均值比单独估计要好。此外，在现实世界当中，许多看上去不相关的事件很强的依赖性。例如，尽管自动驾驶和物体操纵似乎无关，但底层数据受相同的光学、材料特性和动力学定律的支配。况且，在真正的多任务推荐系统，任务的设置也不可能是完全对立的。基于上述种种原因，许多多任务学习相关的工作就此展开了。

1.3MMOE的基本原理

多任务学习最核心的思想就是利用不同任务的数据，来协同训练各个子任务的模型。对于单任务推荐系统模型，它们往往在提取特征之间的交互特征，或者是关联性上下功夫，如基于embedding&mlp模式的FFM，PNN，DeepFM，它们往往采用内积，外积，DNN等方式。而对于基于注意力机制的DIN，DIEN，它们则重点挖掘用户行为和广告内容的相关性。而对于多任务模型，它们的重心放在如何权衡共有特征与专有特征的关系，并由最初的share bottom DNN，演化出了软参数共享，硬参数共享，门控系统等一系列技术。

本文的MMOE模型，是基于MOE模型的改进。在MOE模型中，考虑到对于不同的子任务，它们在专家子网络中的的最佳特征表示是不同的，即不同的子任务模型应该有不同的输入，于是MOE中引入了门控系统，通过门控，去选择那些适合子任务的特征。而MMOE在MOE的基础上，进一步为每个子任务都设置了一套门控系统，以便更好的学习子任务的特征表示。下面将结合MMOE网络架构，对原理进一步说明。

1.4MMOE网络架构

在早期的share bottom DNN中，可以看到所有子任务共享一个特征提取网络，这么做的好处是相较于单任务为每个任务设置一个底部，共享底部能够节省运算开销，但这样学习出的特征明显是单一，尤其是当任务之间的关联性相对较小，甚至部分数据集上冲突，为了权衡各子任务，特征的学习就会折中。于是在MOE中，设置了多个专家expert子网络，去学习不同的特征规律，并且增加了门控系统，在几乎不增加计算开销的情况下，更好的为子任务选择expert子网络的输出。而MMOE则是进一步，为不同子任务都分配了门控系统。

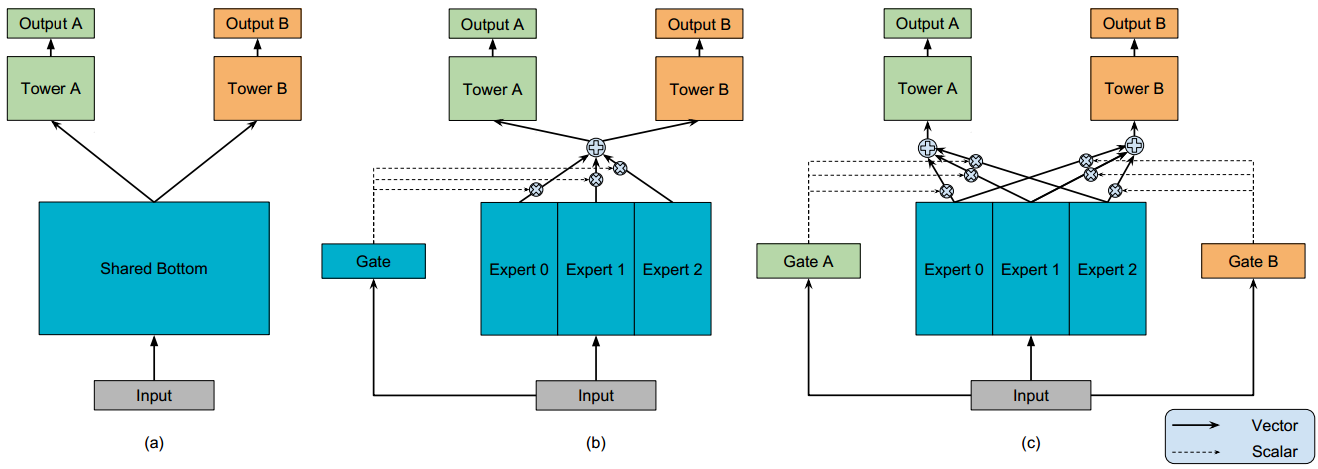


图1-1 share bottom DNN模型，MOE模型以及MMOE模型架构

本文在UCI census-income datase上对不同模型的性能进行了测试，可以看到，MMOE在所有模型中表现最好。

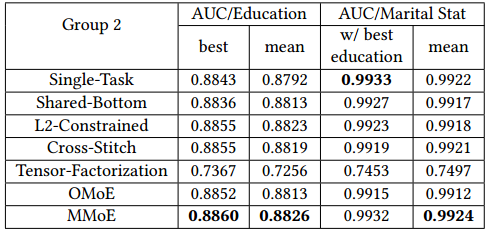


图1-2 MMOE和其他模型的性能对比

二模型的基本设计与实现

接下来将根据论文的原理和实验内容，对MMOE进行实现以及调用，以及给出一些超参数的选择过程的考虑。

2.1模型的设计与实现

MMOE模型通过mmoe.py文件实现，其中定义了继承了nn.module的类MMOE。主要包含了init和forward两个方法，还涉及到一些数据处理，嵌入的方法。下面重点介绍init和forward两个方法的主要实现逻辑。

首先在init中，需要完成一系列的初始化，如experts子网络以及towers网络参数的有关设置，以及嵌入矩阵的创建，然后根据初始化的信息，定义experts子网络以及towers网络的结构，还有最终的预测层。

在初始化部分，需要设置experts中expert的数目，输入层的维度，中间隐藏层的神经元数目，输出层的维度，这里通过nn.ModuleList（）即可实现。门控系统和towers网络的设计也是类似，在此不过多赘述。然后就是定义嵌入矩阵，这需要根据dnn\_feature\_columns实现，其中包含了稀疏特征密集特征的feature\_name，vocabulary\_size，embedding\_size等信息，最后调用create\_embedding\_matrix（）即可实现嵌入矩阵的定义。

接着是forward部分。首先要完成词嵌入的过程，将嵌入矩阵，标签编码后的特征向量，以及dnn\_feature\_columns作为nput\_from\_feature\_columns（）的输入，得到稀疏特征和密集特征的嵌入向量，然后将两种嵌入向量展平并且concat，就得到了experts网络的输入dnn\_input了。后续的实现就很简单，for循环遍历各个expert，得到中间输出，然后将中间输出和dnn\_input送入门控系统，得到对应的权重系数，最后根据系数，不同的tower得到不同的输入。最后遍历各个tower，得到子任务的输出，再将其送入到最终的预测层中，完成预测。至此，MMOE的主要设计和实现部分完成了。

2.2主调函数的实现

主调函数在run\_mmoe.py中实现。其中主要涉及到数据集的加载和预处理，模型的加载编译，以及训练和预测环节。

首先是数据集部分。通过read\_csv（）读取census income数据集的内容，其中有些行的特征值空缺，因此需要剔除这些部分。接着定义稀疏特征和密集特征的特征项，然后完成编码。为了简化操作过程，这里使用了标签编码，并对稀疏特征进行了归一化，以消除取值范围差异带来的影响。然后是对数据集进行划分，由于涉及到不同的划分方法，这一部分将在第三节中展开说明。

接着就是模型的的加载，complie（），以及fit（）和predict（），最后在测试集上验证效果。

代码2-1 主调函数的实现

|  |
| --- |
| sparse\_features = ["workclass", "education", "occupation",  "relationship", "race", "sex", "native-country","marital-status"]  dense\_features = ["age", "fnlwgt", "education-num", "capital-gain","capital-loss"]  target = ['hours-per-week', 'income']  # 1.Label Encoding  lbe = LabelEncoder()  for feat in sparse\_features:  data[feat] = lbe.fit\_transform(data[feat])  mms = MinMaxScaler(feature\_range=(0, 1))  data[dense\_features] = mms.fit\_transform(data[dense\_features])  # 2.count #unique features for each sparse field,and record dense feature field name  data["income"] = lbe.fit\_transform(data["income"])  data["hours-per-week"] = mms.fit\_transform(data[["hours-per-week"]])  fixlen\_feature\_columns = [SparseFeat(feat, vocabulary\_size=data[feat].max() + 1, embedding\_dim=1 if feat=="sex" else 4)  for feat in sparse\_features] + [DenseFeat(feat, 1, )  for feat in dense\_features] |

代码2-1 续 主调函数的实现

|  |
| --- |
| model = MMOE(dnn\_feature\_columns, task\_types=['regression','binary'],  l2\_reg\_embedding=1e-5, task\_names=target, device=device)  model.compile("adagrad", loss=["mse", "binary\_crossentropy"],  metrics=['mse','binary\_crossentropy','mae','auc'],)  history = model.fit(x=train\_model\_input, y=train[target].values, batch\_size=256, epochs=6, verbose=2,  validation\_data=[valid\_model\_input,valid[target].values],  writer=writer\_metrics)  pred\_ans = model.predict(test\_model\_input, 256) |

2.3超参数的选择

这里超参数主要涉及到学习率，epochs，batch\_size。在超参数的选择过程中，简单的将训练集和验证集按照4-1进行划分。划分过后，训练集中有19303条数据，验证集中有4826条数据。学习率的初始值设置为0.005，epochs为8，batch\_size为64。

2.3.1batch\_size的设置

首先对batch\_size进行调整。当batch\_size为64，其余为默认值时。可以看到，各项性能评估指标都抖动严重，勉强能看出训练朝优化方向发展，初步分析是当前批次设置过小，批次数据间差异较大因此导致剧烈的抖动。

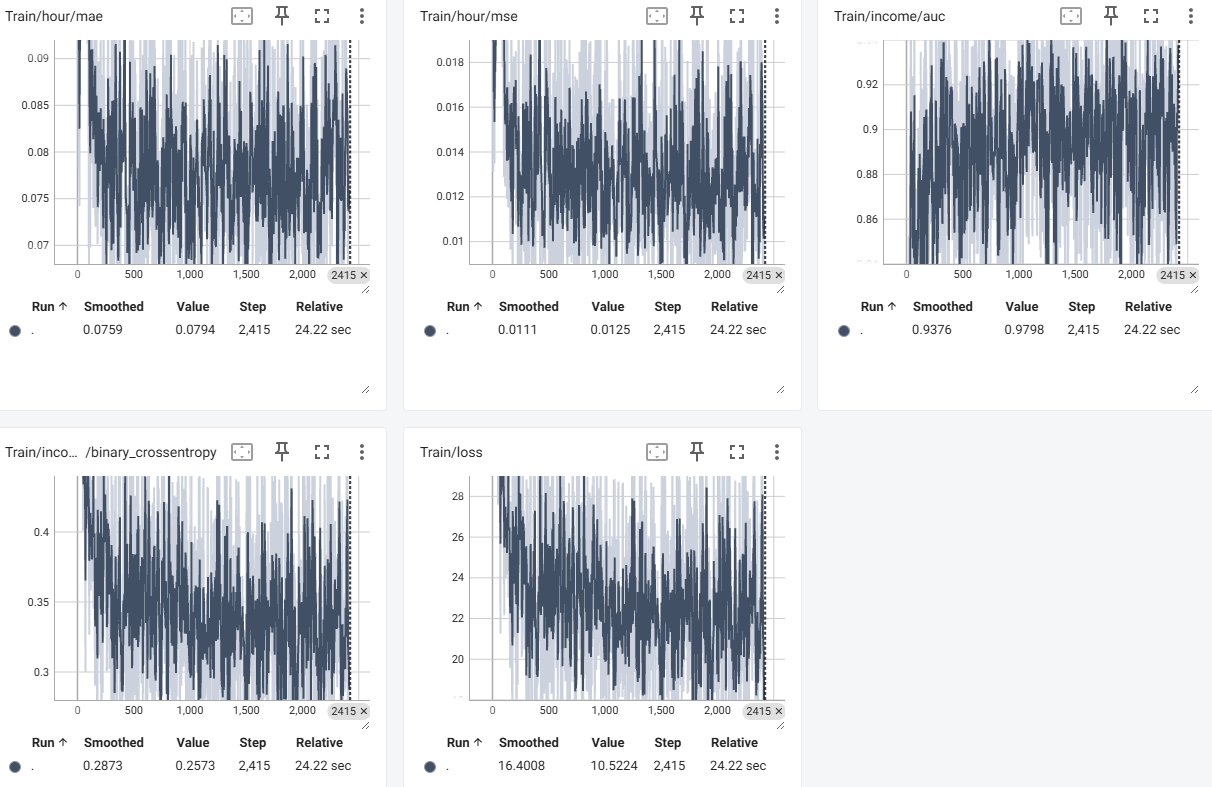


图2-1 学习率0.005，epochs 8，batch\_size 64

接着将batch\_size为128，其余为默认值时。效果明显得到了改善，能够明显看出，训练朝优化的方向进行，各项性能指标都呈变好的方向。但依然抖动严重，考虑到训练集中数据有19000多条，且内存开销较小，因此再次增加批次的大小，看是否继续得到优化。

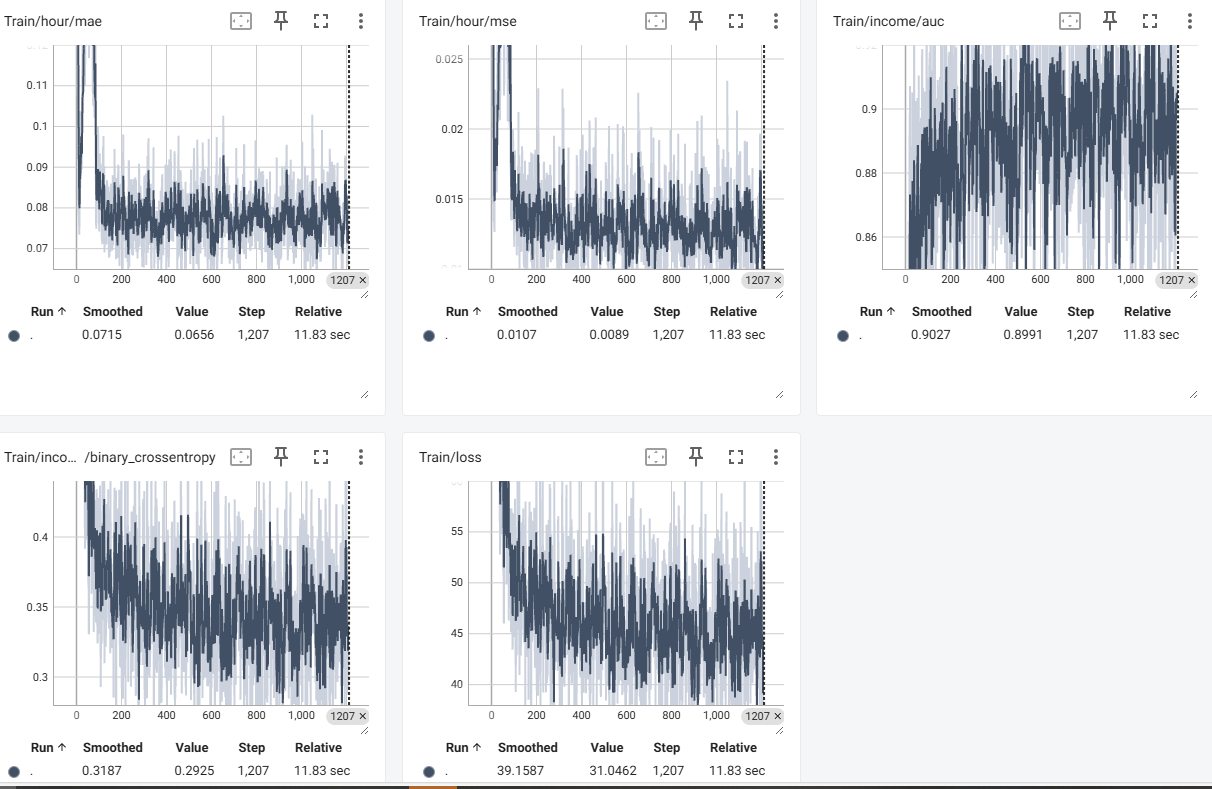


图2-2 学习率0.005，epochs 8，batch\_size 128

接着将batch\_size为256，其余为默认值时。可以看到，效果进一步得到了改善。评估hour的mae和mse指标几乎收敛了，整体的损失也下降的更加稳定，整体呈现的效果较好。不过评估income的auc和交叉熵loss仍较为抖动。

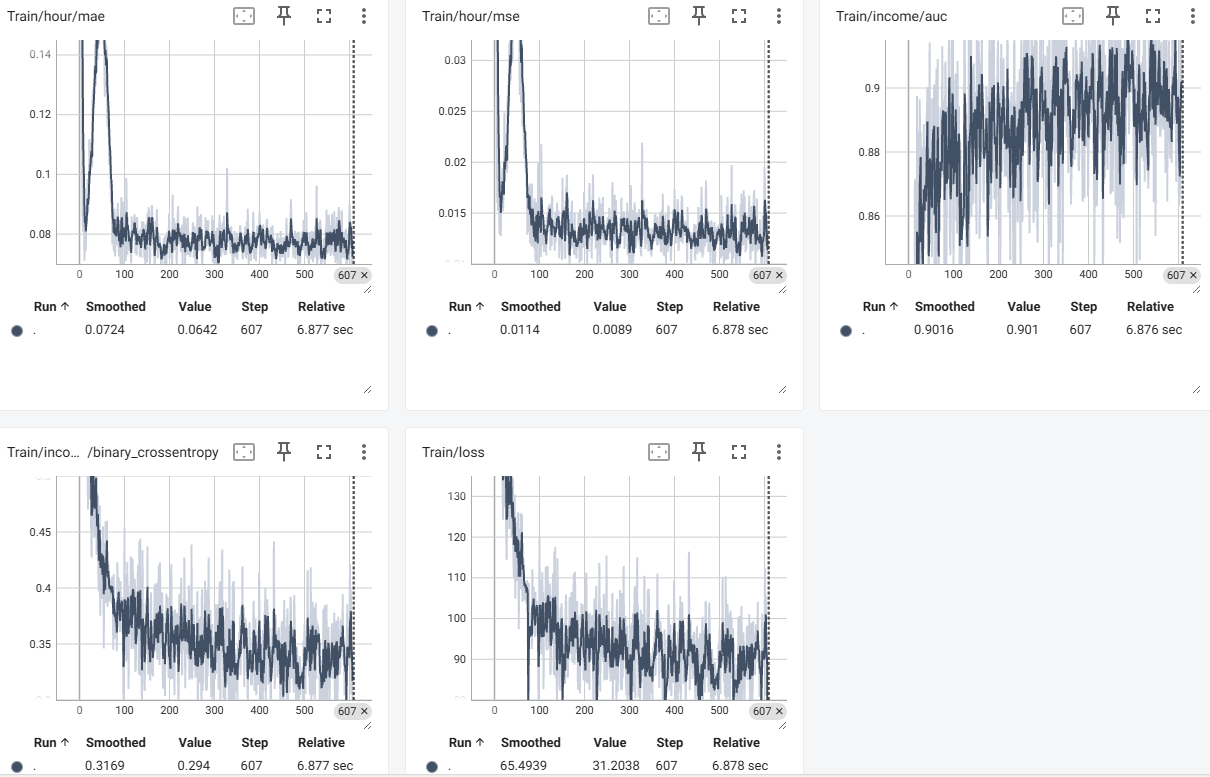


图2-3 学习率0.005，epochs 8，batch\_size 256

随着批次的增加，各项任务的性能评估也表现的越来越好，于是进一步增加批次的大小。如下是批次大小为1024，其余为默认值时。单从下降趋势和稳定性而已，无疑本次的效果最好，但是仔细观察，却会发现，相较于批次为512，本次实验的loss变大了，但测试集验证集性能指标却几乎不变，这其实是因为，在计算损失时，是计算一个批次的损失 ，批次从256变成了512，自然loss也会相应的增大，性能几乎没变化即可。

综上，epoch设置为512时的表现看来最好。

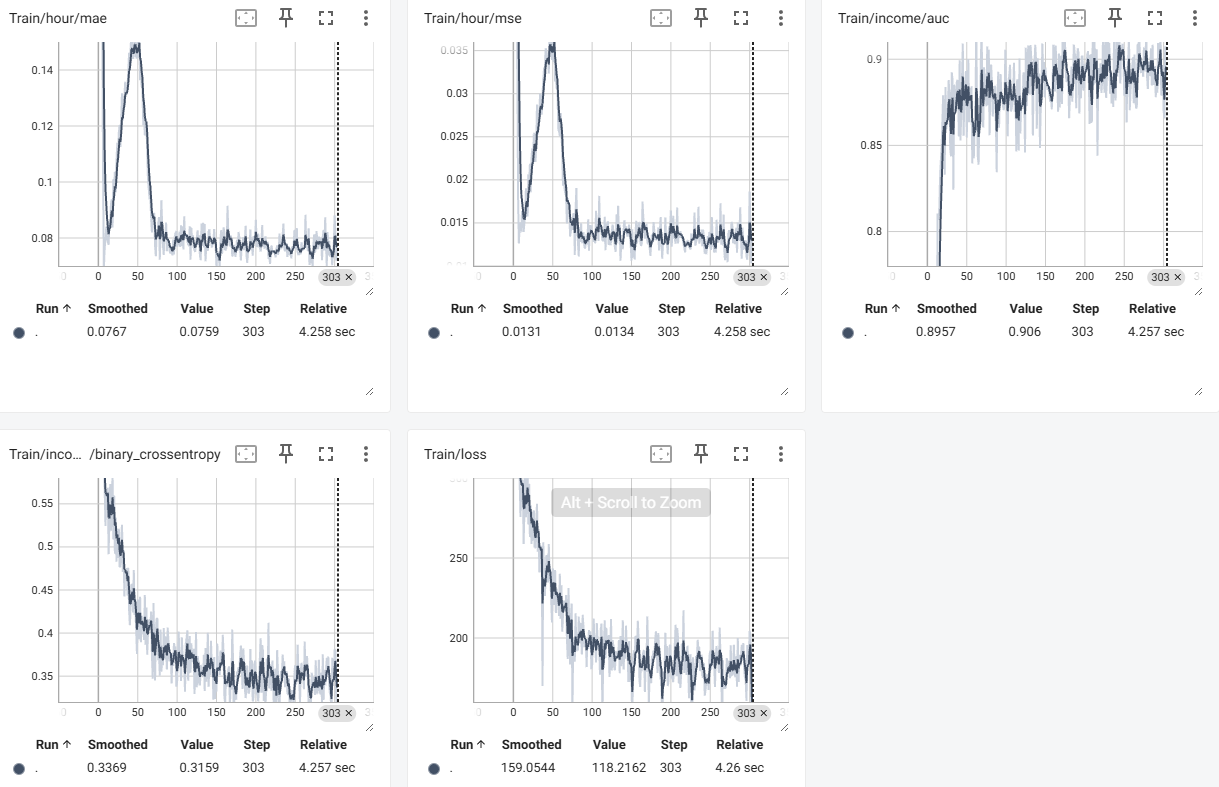


图2-4 学习率0.005，epochs 8，batch\_size 512

2.3.2学习率和Epochs的设置

在2.3.1节的各次实验中，都会发现训练集上，hour的评估指标mae和mse一开始都有一个抖动的现象，如下图所示。并且在训练的末期，各项指标没有趋于一条直线，这有可能是学习率过大导致的。

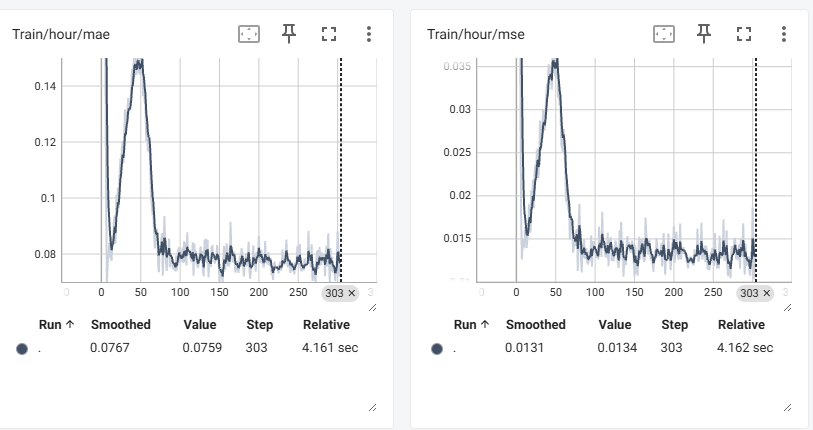


图2-5批次为512时，hour评估指标一开始抖动的情况实例

于是本节决定对学习率进行调整，设置的小一点，但学习率和epochs的相关性又较强，因此在本节协同调整。首先批次固定为512，学习率从0.005调整为0.001，epochs也相应设置大点，为80，也便观察最终的趋势。可以看到，经过调整以后，训练集上的hour消除了一开始的抖动问题，但在step达到90步时，又反弹了，于此同时income表现的越来越好，这是多任务学习中常见的“跷跷板”现象，即任务之间关联性可能较强，也可能较弱，较弱时，可能在优化的过程中存在一定的冲突，即没法让两个子任务同时往好的方向发展。但到step达到800多时，又开始下降了。

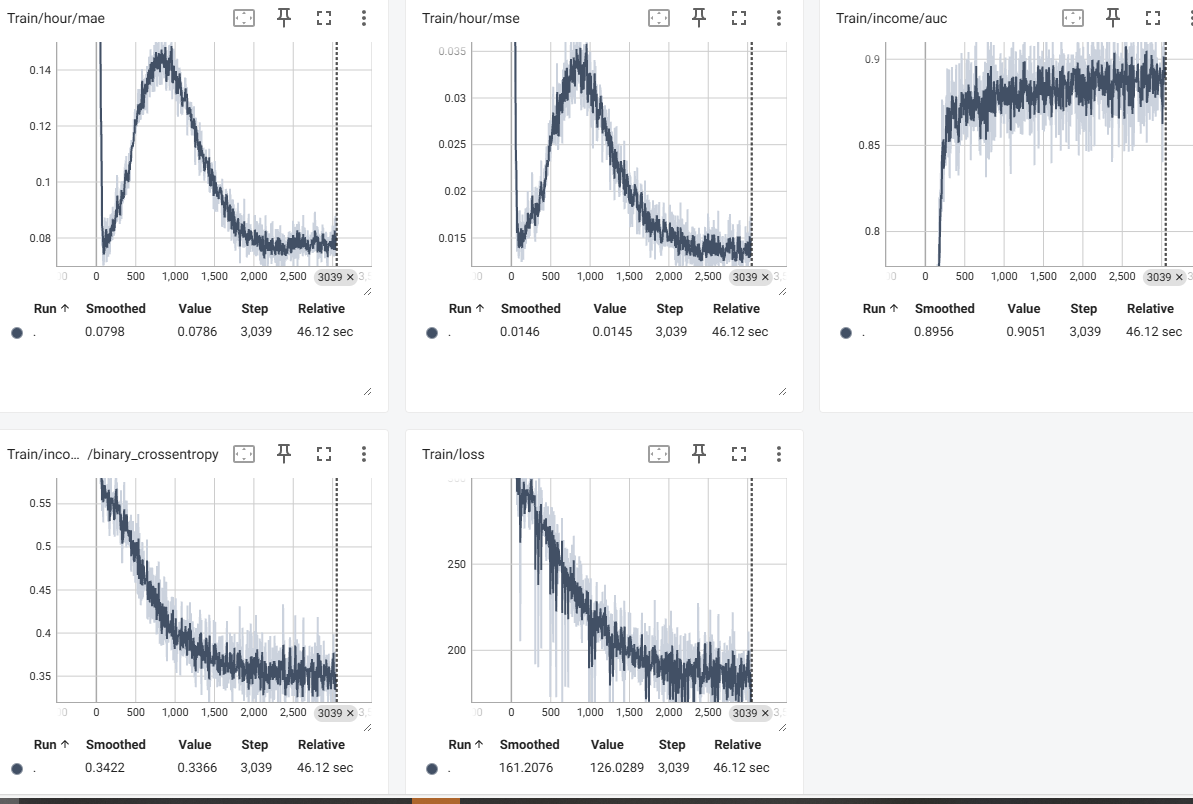


图2-6学习率0.001，epochs 80，batch\_size 512（训练集）

为了防止这是在训练集上过拟合，这里查看验证集上的情况。可以看到，验证集上也呈和训练集上一样的趋势，说明表现确实在变好，而没有过拟合。

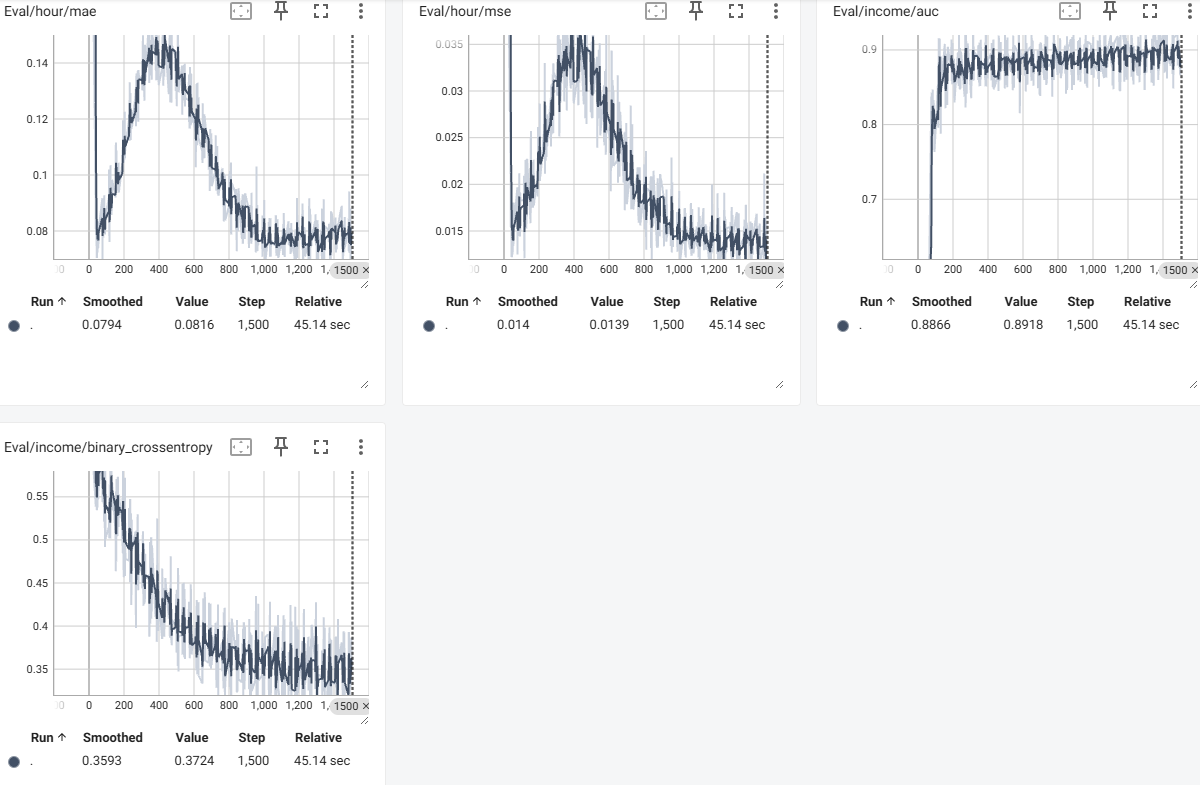


图2-6学习率0.001，epochs 80，batch\_size 512（验证集）

接下来在对学习率设置小一点，从0.001变为0.0005，批次数不变，epochs增大为250。总的来说，差别不大。

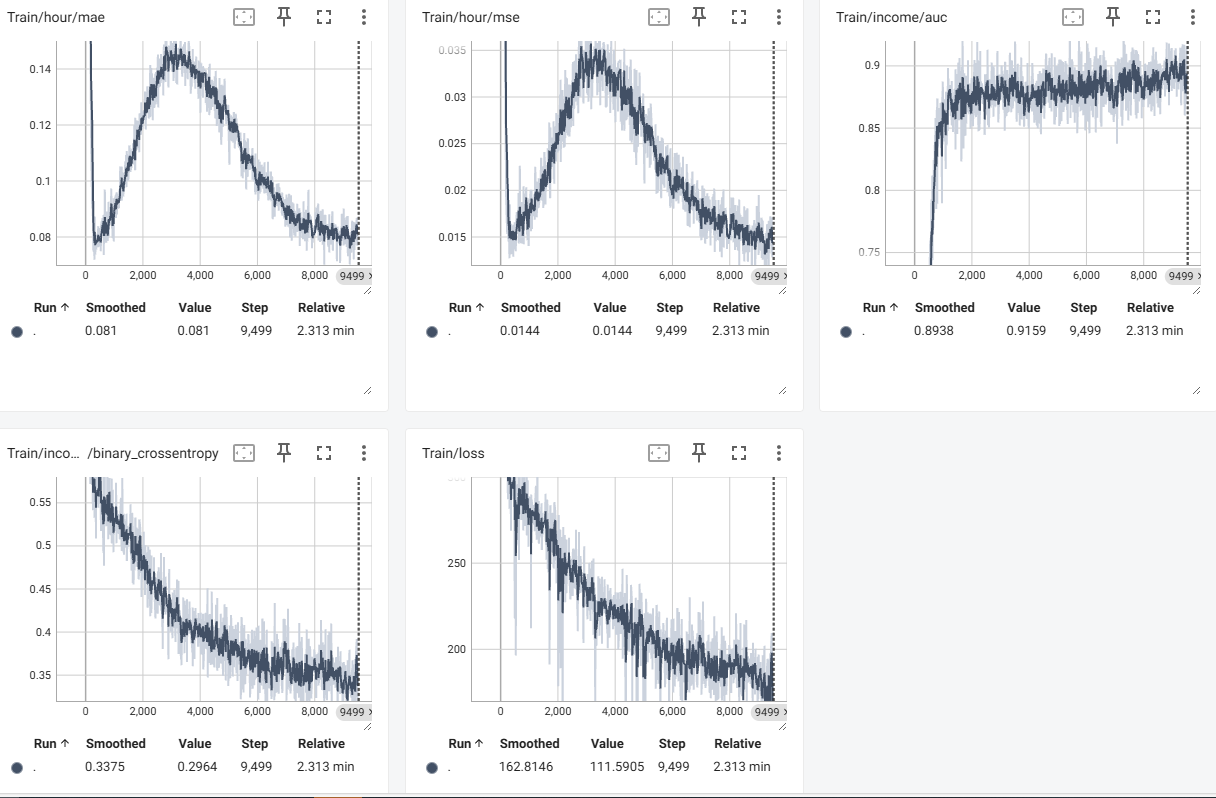


图2-6学习率0.005，epochs 250，batch\_size 512

三基于不同方法的性能验证

3.1任务设置与性能指标

census income数据集共15个特征，包括年龄，性别，工作情况，婚姻情况等内容。在本实验中，基于census income数据集设置了两个任务，一个是预测用户的每周工作时长，另一个是预测用户的收入是否大于50K。这两个任务一个回归任务，一个是二分类任务。这里分别使用MAE和MSE，以及AUC和Logloss作为性能指标，来衡量模型的表现情况。

可视化方面使用tensorboard，将测试集上的性能评估指标添加到日志文件中，来观察不同的划分方法，划分结果，对结果的影响。值得一提的是，因为测试集上的性能验证是在模型训练完了以后进行测试的，因此tensorboard中呈现的效果将是点的形式。此外，控制台中还会输出每次测评的结果以及所有测试结果的平均值，以更好的可视化不同划分方式和结果对性能的影响。测试的日志结果存放在test目录下的各个方法名文件夹中。

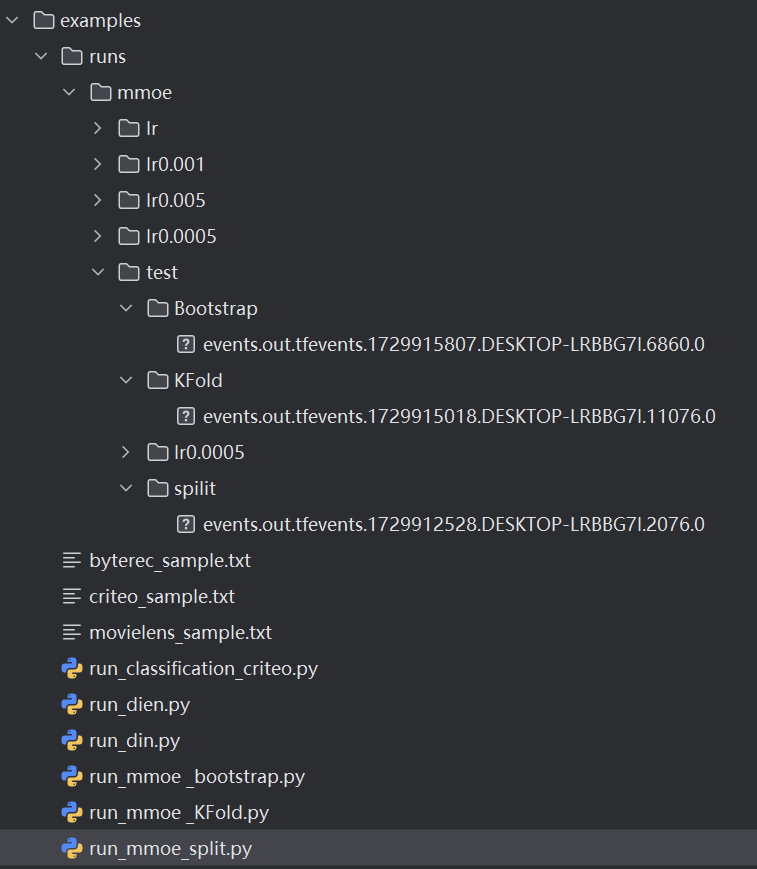


图3-1不同方式的主调程序以及日志文件

3.2基于留出法

首先使用留出法对模型进行测试。在sklearn的model\_selection中，提供了留法法的库函数train\_test\_split（）。因为留出法只有一次测试，因此这里使用for循环，为每次分配指定不同的random\_state，从而确保数据集划分不同。这里具体是设置是，进行8次重复实验，然后考虑到时间开销，这里学习率设置为0.0005，批次为512，epochs为50。

代码3-1 留出法划分数据集

|  |
| --- |
| for i in range(n\_splits):  model = MMOE(dnn\_feature\_columns, task\_types=['regression', 'binary'],  l2\_reg\_embedding=1e-5, task\_names=target, device=device)  model.compile("adagrad", loss=["mse", "binary\_crossentropy"],  metrics=['mse', 'binary\_crossentropy', 'mae', 'auc'], )  train, valid = train\_test\_split(data, test\_size=0.2, random\_state=i)  train\_model\_input = {name: train[name] for name in feature\_names}  valid\_model\_input = {name: valid[name] for name in feature\_names} |

运行run\_mmoe\_split.py，输出不同划分结果中测试集的性能指标情况以及最终的平均结果。



图3-2不同划分结果中测试集的性能指标情况

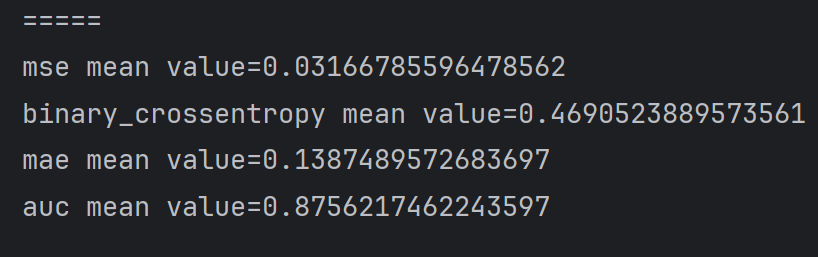


图3-3基于留出法的性能验证

最后查看tensorboard中的动态过程。可以看到，由于输出是点，Tensorboad以为是曲线，因此去线性拟合了，因为只需要关注点的信息，后续也是如此。可以看到，尽管数据集存在差异，但是各次测试的效果几乎一致，性能表现可能只有0.1%的差异，可以忽略不记。

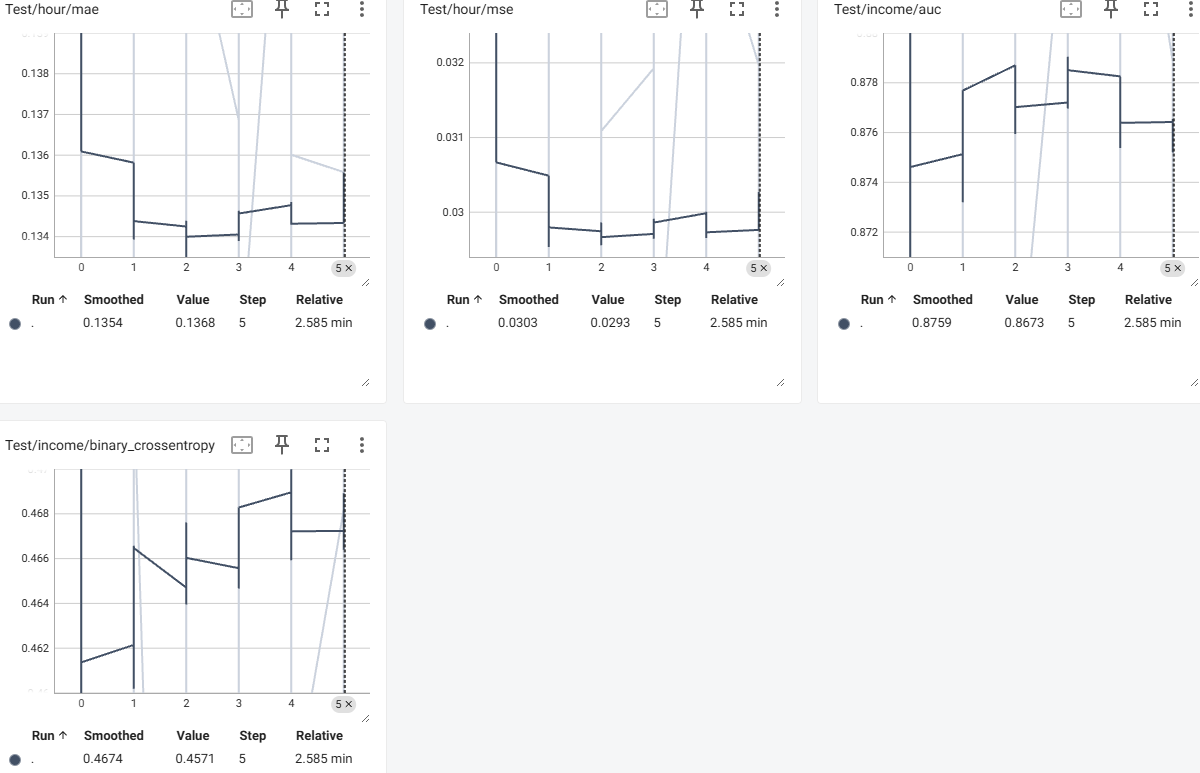


图3-4基于留出法的性能验证

3.3基于K折交叉验证法

接下来使用K折交叉验证法。在sklearn.model\_selection中提供了KFold方法用于构建K折交叉验证。这里使用了10折10次划分，即最终进行10次测试。这里超参数的具体是设置是，学习率设置为0.0005，批次为512，epochs为50。核心代码如下所示：

代码3-2 K折交叉验证法划分数据集

|  |
| --- |
| n\_splits=10 kf = KFold(n\_splits=n\_splits, shuffle=True, random\_state=42) i = 0 for train\_index, val\_index in kf.split(data):  train, valid = data.iloc[train\_index],data.iloc[val\_index]  i=i+1  model = MMOE(dnn\_feature\_columns, task\_types=['regression', 'binary'],  l2\_reg\_embedding=1e-5, task\_names=target, |

代码3-2续 K折交叉验证法划分数据集

|  |
| --- |
| device=device)  model.compile("adagrad", loss=["mse", "binary\_crossentropy"],  metrics=['mse', 'binary\_crossentropy', 'mae', 'auc'], )  train\_model\_input = {name: train[name] for name in feature\_names}  valid\_model\_input = {name: valid[name] for name in feature\_names} |

运行run\_mmoe\_KFold.py，输出结果。

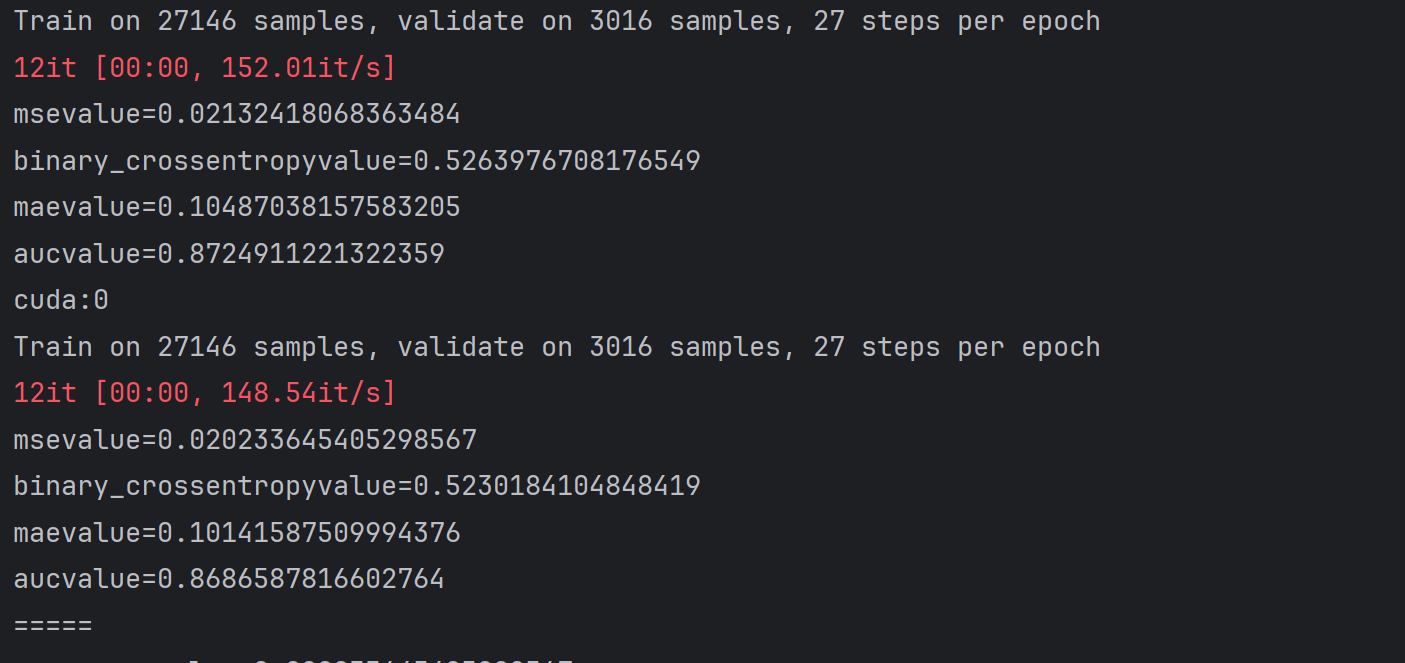


图3-5基于留出法的性能验证

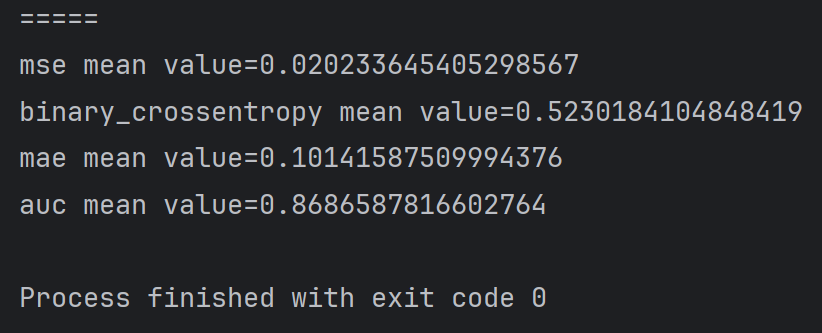


图3-6基于留出法的性能验证

同样的，在tensorboard中查看过程。和留出法的情况类似，不同划分方式带来的差异几乎可以忽略不计。但值得一提是的，在回归任务中，K折的划分方式表现更好，二分类任务中，则略差于留出法。

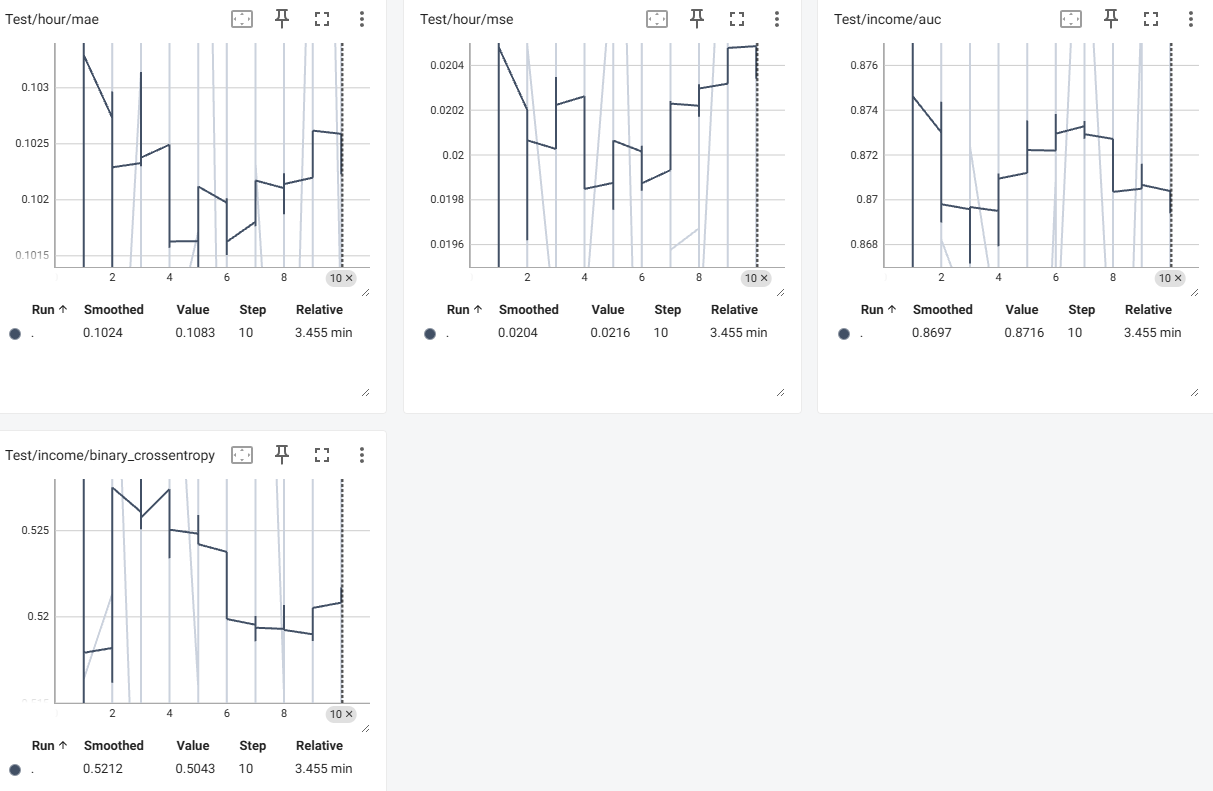


图3-7基于留出法的性能验证

3.4基于自助法

最后是自助法。在numpy中提供了random.choice（）方法用于构建训练集，再在训练集的基础上，使用setdiff1d（）构建测试集。这里超参数的具体是设置是，学习率设置为0.0005，批次为512，epochs为50。核心代码如下所示。

代码3-3 自助法划分数据集

|  |
| --- |
| n\_splits = 6 n\_samples = len(data) train\_indices = np.random.choice(np.arange(n\_samples), size=n\_samples, replace=True) train = data.iloc[train\_indices] # 验证集是未被采样到的数据 validation\_indices = np.setdiff1d(np.arange(n\_samples), train\_indices) valid = data.iloc[validation\_indices] model = MMOE(dnn\_feature\_columns, task\_types=['regression', 'binary'],  l2\_reg\_embedding=1e-5, task\_names=target, device=device) model.compile("adagrad", loss=["mse", "binary\_crossentropy"],  metrics=['mse', 'binary\_crossentropy', 'mae', 'auc'], ) |

运行run\_mmoe\_Bootstrap.py，输出结果。因为留出法只进行一次划分，故只输出了一个结果，这里直接在控制台输出结果。在自助法中，表现和留出法类似。

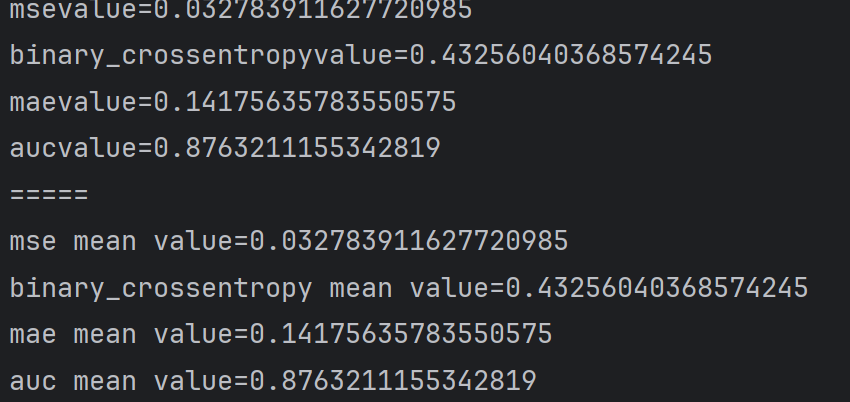


图3-8基于自助法的性能验证