初学者系列：Neural Factorization Machines 神经因子分解机详解

原创 Sha Li 专知 2019-09-09

导读

在预测任务中，特征向量通常是稀疏且高维的。为了解决稀疏预测问题，有效的学习特征交互一直是学者们努力的方向。本文主要通过《Neural Factorization Machines for Sparse Predictive Analytics》一文介绍一种更有效的学习特征交互，解决稀疏预测问题的方法——NCF（Neural Factorization Machines）。

No.1 动机

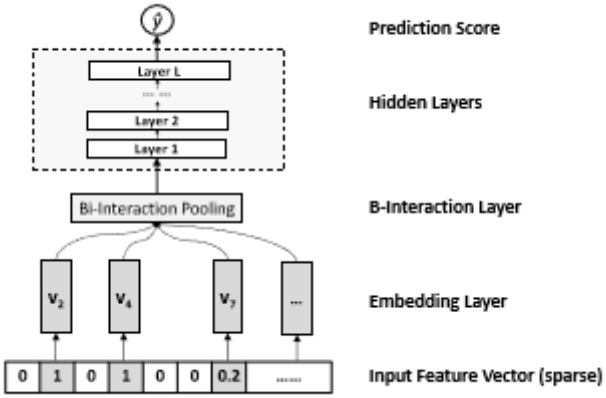
预测分析是推荐系统、搜索排名等信息检索（IR）和数据挖掘（DM）任务的最重要技术之一。预测变量大多是离散的和分类的。为了构建预测模型，通常使用one-hot编码将输入转化为二进制，但是经过转换的特征向量通常是高维且稀疏的。为了更有效的在稀疏数据中学习，更好地学习特征交互是必不可少的。但是之前学习特征交互的方法(例如：人工特征、FM以及DNN)都存在或多或少的局限。

方法	核心思想	缺点
人工特征	通过组合多个预测变量来构建新特征	需要大量的领域知识，无法学习训练集中没有出现的特征组合、代价大，不好推广。
FM（因子分解机）	以线性方式学习二阶特征交互	很难捕获现实数据非线性与复杂的内在结构。
DNN	利用神经网络学习高阶非线性特征	难以训练

为了解决这些方法中的缺陷，更好地学习交互特征，作者提出了NFM（Neural Factorization Machine）模型用于稀疏数据下的预测，**NFM结合了FM在建模二阶特征相互作用中的线性与神经网络（NN）在建模高阶特征相互作用中的非线性**。并且NFM使用更浅的结构 在练习中更容易训练和调整。

No.2 核心思想

NFM结合了FM在建模二阶特征相互作用中的线性与神经网络（NN）在建模高阶特征相互作用中的非线性，实现高阶交互+非线性特征交互。



NCF是将FM中的二阶交叉特征项通过双线性交互层（BI-Interaction）进行池化操作实现，并将得到的交叉特征送入神经网络（NN）中提取高阶与非线性的特征交互。

对于FM的输出为：

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j$$

tips：请点击FM看FM算法详解

NCF的输出为：

$$\hat{y}_{NFM} = w_0 + \sum_{i=1}^{n} w_i x_i + f(x)$$

f（x）是NFM的核心组件，用于建模特征交互。

Embedding层

Embedding层是一个全连接层，是将稀疏输入映射为一个密集向量。可以将FM算法中的隐向量V看作该Embedding层的权重矩阵，经过Embedding后会获得一系列的嵌入向量（

embedding vectors ）V_x：

$$V_x = \{x_1 v_1, \dots, x_n v_n\}$$

由于x为one-hot形式（输入x中只有一个位置不为0），故 V_x 在 x_i 不等于0时可以写为：

$$V_x = \{x_i v_i\}$$

BI-Interaction层

BI-Interaction层是一个池化操作，将Embedding层得到的一系列嵌入向量集合V_x 转化为一个向量（相当于FM模型中的交叉项）。

$$f_{BI}(V_x) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} x_i v_i \odot x_j v_j$$

与FM中的交叉项一样，BI-Interaction层的输出 f_{BI}(V_x)可以化简为如下形式：

$$f_{BI}(V_x) = \frac{1}{2} \left| (\sum_{i=1}^{n} x_i v_i)^2 - \sum_{i=1}^{n} (x_i v_i)^2 \right|$$

其中：

- ⊙ 表示向量的逐元素乘积
- 输出是k维向量，k是embedding的维度

隐藏层

隐藏层学习特征之间的高阶交互，各层的输出如下：

$$z_i = \begin{cases} \sigma_i(W_i z_{i-1} + b_i) & , i = 2 \dots L \\ \sigma_1(W_1 f_{RI}(V_X) + b_1) & , i = 1 \end{cases}$$

- W_i表示第i层的感知机权重矩阵；
- b_i表示第i层的感知机的偏置向量；
- σ_i为第i层的激活函数。

因此，NCF的输出表示为：

$$\begin{aligned} \hat{y}_{NFM} &= w_0 + \sum_{i=1}^{n} w_i x_i + f(x) \\ &= w_0 + \sum_{i=1}^{n} w_i x_i + h^T z_L \\ &= w_0 + \sum_{i=1}^{n} w_i x_i + h^T \sigma_L(W_L(\dots \sigma_1(W_1 f_{RI}(V_X) + b_1) \dots) + b_L) \end{aligned}$$

其中：

- h为输出层神经元权重

> Note:将隐藏层去掉直接连接输出层，NFM就变为了FM

优化

对于不同的优化问题（回归、分类、排名）可以选择不同的损失函数。

优化问题	损失函数
回归	平方损失
分类	交叉熵损失、对数损失
排名	contrastive max-margin loss

我们以回归问题为例，使用平方损失作为目标函数：

$$L_{reg} = \sum_{x \in \chi} (\hat{y}(x) - y(x))^2$$

则参数更新可以表示为：

$$\Theta = \Theta - \eta . 2(\hat{y}(x) - y(x)) \frac{\partial \hat{y}(x)}{\partial \theta}$$

其中：

- η为学习率
- 参数Θ 包括{w_0,{w_i,v_i},h,{W_l,b_l}}

为了更好地训练模型，作者还提出在训练过程中加入Dropout与Batch Normalization.

Dropout

论文中并没有在目标函数中加入L2正则化项，而是分别在Bi-Interaction层、隐藏层进行了Dropout，随机丢弃隐层向量，以防止过拟合。

Batch Normalization

为了避免出现随着网络深度加深，训练越来越困难，收敛越来越慢的问题，作者加入了Batch Normalization对输入进行处理（把每层神经网络任意神经元的输入值的分布强行拉回到均值为0方差为1的标准正态分布）。BN操作层，位于X=W_i *z_{i-1}+b_i激活值获得之后，非线性函数变换之前。

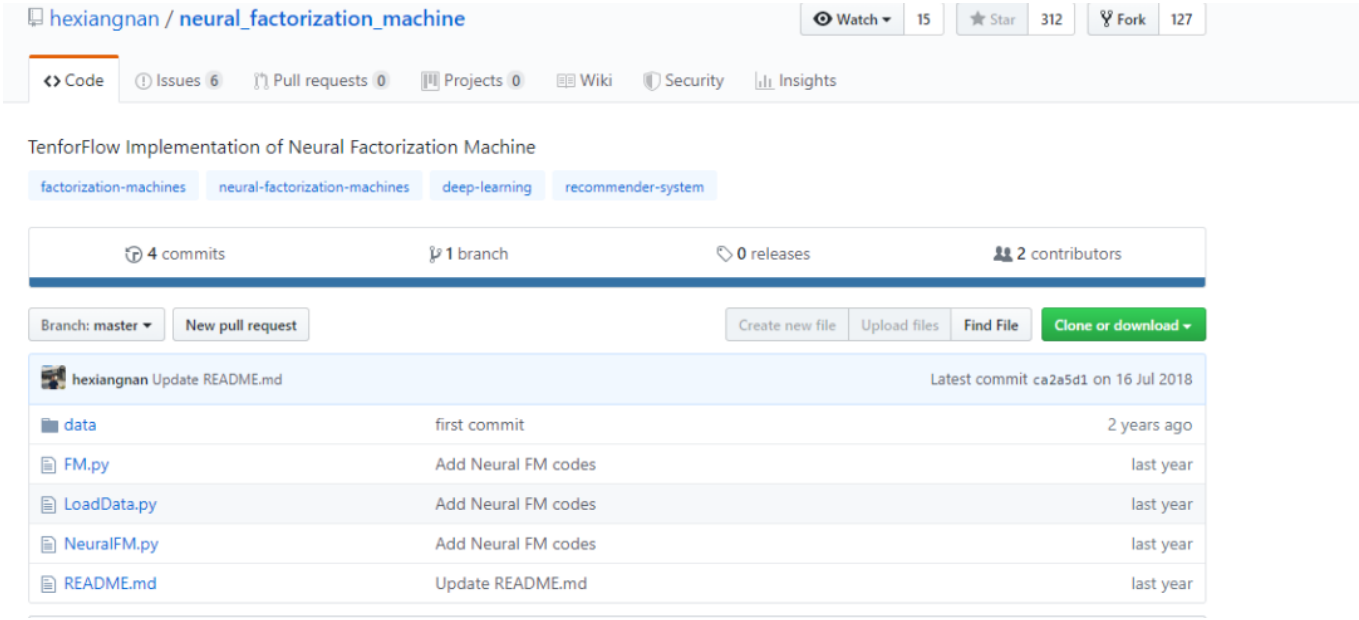
> tips：出现网络收敛越来越慢的原因：深层神经网络在传入激活前的输入（W_i *z_{i-1}+b_i），在训练过程中分布会逐渐往非线性方向偏移，造成低层神经网络梯度消失。

No.3 代码详解

源码地址：

https://github.com/hexiangnan/neural_factorization_machine

主要框架如下：



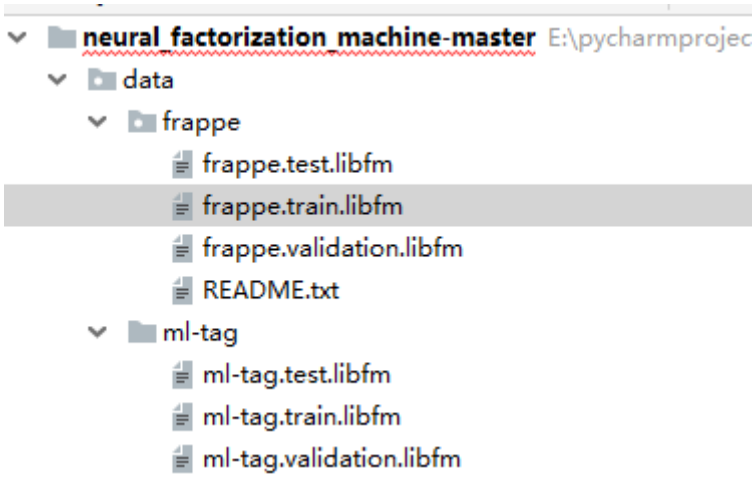
环境：

- Python 2.7
- Tensorflow

> tips：可以将源码中的xrang改为range、sub改为subtract即可使用Python 3

数据

官方给出了经过处理的 MovieLens 1 Million (ml-1m) 和 frappe两个数据集，数据集示例如下：



在所有的数据中，第一位代表标签（1或者-1），其余的都是特征。

```
-1 451:1 4149:1 5041:1 5046:1 5053:1 5055:1 5058:1 5060:1 5069:1 5149:1
-1 91:1 3503:1 5041:1 5047:1 5053:1 5056:1 5058:1 5065:1 5095:1 5149:1
1 168:1 983:1 5040:1 5050:1 5054:1 5055:1 5058:1 5060:1 5069:1 5207:1
-1 620:1 1743:1 5045:1 5051:1 5054:1 5055:1 5058:1 5061:1 5073:1 5149:1
-1 46:1 2692:1 5040:1 5049:1 5054:1 5055:1 5058:1 5060:1 5086:1 5211:1
-1 576:1 4933:1 5041:1 5049:1 5054:1 5056:1 5058:1 5061:1 5075:1 5149:1
1 71:1 966:1 5043:1 5049:1 5054:1 5055:1 5058:1 5061:1 5069:1 5172:1
1 43:1 974:1 5040:1 5048:1 5054:1 5055:1 5058:1 5060:1 5069:1 5252:1
-1 168:1 2928:1 5040:1 5051:1 5054:1 5055:1 5058:1 5062:1 5069:1 5149:1
-1 14:1 2396:1 5039:1 5047:1 5053:1 5055:1 5058:1 5061:1 5076:1 5149:1
-1 107:1 4380:1 5040:1 5046:1 5053:1 5055:1 5058:1 5061:1 5069:1 5149:1
-1 80:1 2662:1 5041:1 5047:1 5053:1 5055:1 5058:1 5061:1 5070:1 5243:1
1 190:1 1093:1 5039:1 5052:1 5054:1 5055:1 5058:1 5061:1 5105:1 5149:1
1 131:1 1432:1 5043:1 5050:1 5054:1 5055:1 5058:1 5061:1 5099:1 5215:1
-1 116:1 986:1 5039:1 5051:1 5054:1 5055:1 5058:1 5062:1 5074:1 5149:1
-1 92:1 4253:1 5041:1 5050:1 5054:1 5056:1 5058:1 5062:1 5069:1 5149:1
-1 16:1 1016:1 5039:1 5046:1 5053:1 5056:1 5059:1 5063:1 5073:1 5156:1
-1 38:1 2047:1 5040:1 5046:1 5053:1 5055:1 5058:1 5060:1 5069:1 5149:1
1 432:1 1060:1 5040:1 5051:1 5054:1 5055:1 5058:1 5061:1 5073:1 5156:1
1 488:1 957:1 5040:1 5052:1 5054:1 5055:1 5058:1 5064:1 5110:1 5149:1
```

构造数据集（LodaData.py）

在构造数据集时主要是通过如下两个函数实现，得到数据集中特征的索引值（**并没有对输入数据直接进行one-hot处理**）与标签，并存放在字典中，组成训练集、验证集、测试集。

```
    ##得到数据中特征的索引、标签
    def read_data(self, file):
        # read a data file. For a row, the first column goes into Y_;
        # the other columns become a row in X_ and entries are maped to indexs in
self.features
        f = open( file )
        X_ = []
        Y_ = []
        Y_for_logloss = []
        line = f.readline()
        while line:
            items = line.strip().split(' ') #将训练集中的数据按空格划分
            Y_.append( 1.0*float(items[0]) )#标签，浮点型[-1.0,-1.0,1.0,......]
            if float(items[0]) > 0:# > 0 as 1; others as 0
                v = 1.0
            else:
                v = 0.0
            Y_for_logloss.append( v )
            X_.append( [ self.features[item] for item in items[1:]] )#x_中存放特征对应的索引
[[第一个数据item索引值]，[第二个数据item索引值]]
            line = f.readline()
        f.close()
        return X_, Y_, Y_for_logloss
    ##获得Data_Dic字典,{'Y': [0,0,1,...,标签], 'X'[[标签对应的item索引], [],[],...[]]}
    def construct_dataset(self, X_, Y_):
        Data_Dic = {}
        X_lens = [ len(line) for line in X_]#X_lens=[10,10,...,每一行数据有多少个索引值]
        indexs = np.argsort(X_lens)#值从小到大排序后,并按照其相对应的索引值输出
        Data_Dic['Y'] = [ Y_[i] for i in indexs]
        Data_Dic['X'] = [ X_[i] for i in indexs]

        return Data_Dic
```

构造得到的数据集形式如下：

```
#从LoadData中可以得到，不同特征值的数量、训练、验证、测试数据集（都是字典）
    ##features_M=5382， 为不同特征值的数量
    ##Train_data、Validation_data、Test_data，都是字典，
    ##形如{'Y': [0,0,1,...,标签], 'X'[[标签对应的item索引], [],[],...[]]}
    ##features={‘451:1’:0, ‘4149:1’: 1, 。。。, ‘特征’: 特征对应的索引值},里面没有重复的特征
```

主函数

源码中除了给出实现NFM的代码以外，还给出了实现FM的代码，下文主要介绍NFM的实现。

在主函数部分主要是通过调用LodaData（）加载数据集、NeuralFM类获得定义的NFM模型以及训练函数train（）训练模型。并且在训练的每一个epoch都会输出训练集、验证集、测试集的损失，并在最后输出效果最好的一次。

```
if __name__ == '__main__':

    # 加载数据
    args = parse_args()
    data = DATA.LoadData(args.path, args.dataset, args.loss_type)
    if args.verbose > 0:
        print("Neural FM: dataset=%s, hidden_factor=%d, dropout_keep=%s, layers=%s,
loss_type=%s, pretrain=%d, #epoch=%d, batch=%d, lr=%.4f, lambda=%.4f, optimizer=%s,
batch_norm=%d, activation=%s, early_stop=%d"
              %(args.dataset, args.hidden_factor, args.keep_prob, args.layers,
args.loss_type, args.pretrain, args.epoch, args.batch_size, args.lr, args.lamda,
args.optimizer, args.batch_norm, args.activation, args.early_stop))
    activation_function = tf.nn.relu
    if args.activation == 'sigmoid':
        activation_function = tf.sigmoid
    elif args.activation == 'tanh':
        activation_function == tf.tanh
    elif args.activation == 'identity':
        activation_function = tf.identity
    t1 = time()
    #加载模型
    model = NeuralFM(data.features_M, args.hidden_factor, eval(args.layers),
args.loss_type, args.pretrain, args.epoch, args.batch_size, args.lr, args.lamda,
eval(args.keep_prob), args.optimizer, args.batch_norm, activation_function, args.verbose,
args.early_stop)
    #开始训练+验证+测试，选择平方差损失时，输出MSE指标
    model.train(data.Train_data, data.Validation_data, data.Test_data)
    # Find the best validation result across iterations
    best_valid_score = 0
    if args.loss_type == 'square_loss':
        best_valid_score = min(model.valid_rmse)#验证集的损失函数中最小的值
    elif args.loss_type == 'log_loss':
        best_valid_score = max(model.valid_rmse)
    best_epoch = model.valid_rmse.index(best_valid_score)#输出效果最好的epoch
    print ("Best Iter(validation)= %d\t train = %.4f, valid = %.4f, test = %.4f [%.1f s]"
           %(best_epoch+1, model.train_rmse[best_epoch], model.valid_rmse[best_epoch],
model.test_rmse[best_epoch], time()-t1))
```

模型

NFM模型的实现主要通过NeuralFM类中的 _init_graph函数。模型主要包括Embedding层、BI层、隐藏层、输出层四部分。并且在 _init_graph（）中定义了损失函数与优化器。

- Embedding层：通过函数 tf.nn.embedding_
  lookup（）根据输入特征的索引号找到对应权重中的一行。（这也是在一开始不需要对数据进行one-hot转换的原因）

- BI层：相当于FM中的二阶交叉项计算。

- 隐藏层：隐藏层通过for循环实现每一层的W_i * z_{i-1} + b_i。
- 输出层：

$$\hat{y}_{NFM} = w_0 + \sum_{i=1}^{n} w_i x_i + f(x)$$

在这里对于FM中的一阶特征w_i * x_i的实现并不是直接相乘，而是通过一维的embedding实现的。

```
# Model
 #-------------------------   Embedding层  ------------------------------------
    ##按照train_features的索引值取出v_i中相应的行
            nonzero_embeddings =
tf.nn.embedding_lookup(self.weights['feature_embeddings'], self.train_features)
 # --------------------    BI层，相当于FM的交叉项计算  -------------------------------
            self.summed_features_emb = tf.reduce_sum(nonzero_embeddings, 1) # None * K
            self.summed_features_emb_square = tf.square(self.summed_features_emb)#None*K

            self.squared_features_emb = tf.square(nonzero_embeddings)
            self.squared_sum_features_emb = tf.reduce_sum(self.squared_features_emb, 1)
            self.FM = 0.5 * tf.subtract(self.summed_features_emb_square,
self.squared_sum_features_emb)  # None * K
       ##-----------batch_norm与dropout，在BI层后以及每一层隐藏层---------------
            if self.batch_norm:
                self.FM = self.batch_norm_layer(self.FM, train_phase=self.train_phase,
scope_bn='bn_fm')
            self.FM = tf.nn.dropout(self.FM, self.dropout_keep[-1])
 #  -------------------------      隐藏层      ----------------------------------
            for i in range(0, len(self.layers)):
                #W*x+b
                self.FM = tf.add(tf.matmul(self.FM, self.weights['layer_%d' %i]),
self.weights['bias_%d'%i]) # None * layer[i] * 1
       ##----------batch_norm与dropout，在BI层后以及每一层隐藏层-------------
                if self.batch_norm:
                    self.FM = self.batch_norm_layer(self.FM,
train_phase=self.train_phase, scope_bn='bn_%d' %i) # None * layer[i] * 1
                self.FM = self.activation_function(self.FM)
                self.FM = tf.nn.dropout(self.FM, self.dropout_keep[i])

 #---------------------------     输出层     ------------------------------------
            self.FM = tf.matmul(self.FM, self.weights['prediction'])     # None * 1
             ##f(x)
            Bilinear = tf.reduce_sum(self.FM, 1, keep_dims=True)  # None * 1
             ##w_i*x_i，相当于进行了一次维度为1的embedding
            self.Feature_bias =
tf.reduce_sum(tf.nn.embedding_lookup(self.weights['feature_bias'], self.train_features) ,
1)  # None * 1
             ##b
            Bias = self.weights['bias'] * tf.ones_like(self.train_labels)  # None * 1
            ##最后的输出
            self.out = tf.add_n([Bilinear, self.Feature_bias, Bias])  # None * 1
```

训练

在函数train()中在每个epoch中都随机的选择batch_size个数据进行训练，获得损失函数。并且调用evaluate（）函数对训练、验证、测试数据进行评价。

```
    def train(self, Train_data, Validation_data, Test_data):  # fit a dataset
        if self.verbose > 0:
            t2 = time()
            init_train = self.evaluate(Train_data)#在平方差损失时 得到均方误差，在对数损失时得到
损失值
            init_valid = self.evaluate(Validation_data)
            init_test = self.evaluate(Test_data)
            print("Init: \t train=%.4f, validation=%.4f, test=%.4f [%.1f s]" %
(init_train, init_valid, init_test, time()-t2))
        for epoch in range(self.epoch):
            t1 = time()
            self.shuffle_in_unison_scary(Train_data['X'], Train_data['Y'])#对数据集进行随机
转换
            total_batch = int(len(Train_data['Y']) / self.batch_size)#总共的批次数
            for i in range(total_batch):
                # 获得一个batch_size的数据
                batch_xs = self.get_random_block_from_data(Train_data, self.batch_size)
                # 得到一个batch_size的损失
                self.partial_fit(batch_xs)
            t2 = time()
            # output validation,

            train_result = self.evaluate(Train_data)
            valid_result = self.evaluate(Validation_data)
            test_result = self.evaluate(Test_data)

            self.train_rmse.append(train_result)
            self.valid_rmse.append(valid_result)
            self.test_rmse.append(test_result)
            if self.verbose > 0 and epoch%self.verbose == 0:
                print("Epoch %d [%.1f s]\ttrain=%.4f, validation=%.4f, test=%.4f [%.1f
s]"
                      %(epoch+1, t2-t1, train_result, valid_result, test_result, time()-
t2))
            if self.early_stop > 0 and self.eva_termination(self.valid_rmse):
                #print "Early stop at %d based on validation result." %(epoch+1)
                break
```

在evaluate（）函数中分了两种情况，当损失函数是平方差损失时，计算了真实数据与预测数据的标准误差（RMSE）；损失函数是对数损失时，计算对数损失。

```
    def eva_termination(self, valid):
        if self.loss_type == 'square_loss':
            if len(valid) > 5:
                if valid[-1] > valid[-2] and valid[-2] > valid[-3] and valid[-3] >
valid[-4] and valid[-4] > valid[-5]:
                    return True
        else:
            if len(valid) > 5:
                if valid[-1] < valid[-2] and valid[-2] < valid[-3] and valid[-3] <
valid[-4] and valid[-4] < valid[-5]:
                    return True
        return False
    #评价指标
    def evaluate(self, data):  # evaluate the results for an input set
        num_example = len(data['Y'])
        feed_dict = {self.train_features: data['X'], self.train_labels: [[y] for y in
data['Y']], self.dropout_keep: self.no_dropout, self.train_phase: False}
        predictions = self.sess.run((self.out), feed_dict=feed_dict)
        print(predictions)
        y_pred = np.reshape(predictions, (num_example,))
        y_true = np.reshape(data['Y'], (num_example,))

        if self.loss_type == 'square_loss':
            #挑出预测之中比-1大比1小的数
            predictions_bounded = np.maximum(y_pred, np.ones(num_example) * min(y_true))
 # 拿预测值跟-1比，输出二者中的最大值
            predictions_bounded = np.minimum(predictions_bounded, np.ones(num_example) *
max(y_true))  # bound the higher values
            RMSE = math.sqrt(mean_squared_error(y_true, predictions_bounded))#均方误差
            return RMSE
        elif self.loss_type == 'log_loss':
            logloss = log_loss(y_true, y_pred) # I haven't checked the log_loss
            return logloss
```

训练结果如下：

```
NeuralFM
Epoch 49 [4.5 s]    train=0.1206, validation=0.3227, test=0.3295 [2.4 s]
Epoch 50 [4.4 s]    train=0.1240, validation=0.3214, test=0.3276 [2.4 s]
Epoch 51 [4.6 s]    train=0.1215, validation=0.3208, test=0.3272 [2.4 s]
Epoch 52 [4.7 s]    train=0.1207, validation=0.3207, test=0.3278 [2.4 s]
Epoch 53 [4.6 s]    train=0.1157, validation=0.3206, test=0.3276 [2.4 s]
Epoch 54 [4.7 s]    train=0.1187, validation=0.3206, test=0.3279 [2.3 s]
Epoch 55 [4.6 s]    train=0.1170, validation=0.3214, test=0.3284 [2.4 s]
Epoch 56 [4.6 s]    train=0.1214, validation=0.3188, test=0.3258 [2.4 s]
Epoch 57 [4.6 s]    train=0.1211, validation=0.3201, test=0.3262 [2.4 s]
Epoch 58 [4.6 s]    train=0.1166, validation=0.3201, test=0.3270 [2.4 s]
Epoch 59 [4.6 s]    train=0.1234, validation=0.3187, test=0.3253 [2.2 s]
Epoch 60 [4.6 s]    train=0.1196, validation=0.3187, test=0.3256 [2.3 s]
Epoch 61 [4.6 s]    train=0.1175, validation=0.3191, test=0.3264 [2.4 s]
Epoch 62 [4.6 s]    train=0.1158, validation=0.3197, test=0.3272 [2.4 s]
Epoch 63 [4.6 s]    train=0.1161, validation=0.3197, test=0.3269 [2.3 s]
Best Iter(validation)= 59    train = 0.1234, valid = 0.3187, test = 0.3253 [450.2 s]
```