Data prepare：数据准备部分，说说怎么在kaggle上找数据集的吧。

Data Preprocess：

1. 数据清洗：

数据去重：

self.data.drop\_duplicates()

过滤数据，去除不需要的特征数据：

self.data = self.data.drop('Extracurricular Activities', axis=1)

1. 归一化处理：

由于数据集中四个特征数据的范围和尺度之间存在一定的差异，所以我们对数据进行了归一化处理：

归一化算法：对于每一个特征x有：

# 归一化算法采用x'=(x-mean(x))/(max-min)

我们找到每个特征数据中的mean值，min值和max值，使用上述公式进行处理。

（3）训练集和测试集的划分：

self.data\_train, self.data\_test = train\_test\_split(self.new\_data, test\_size=0.1)

Model construct：

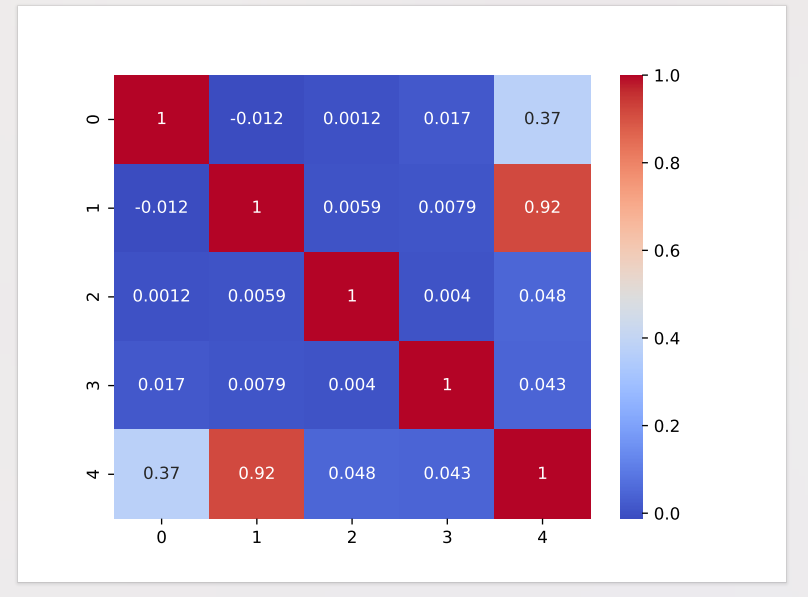
1. 确定训练集的X值和Y值
2. 初始化权重W 和偏差B
3. 完成SGD（随机梯度下降算法）的实现
4. 对每个批次数据计算损失函数值，并更新参数

Train & Test：

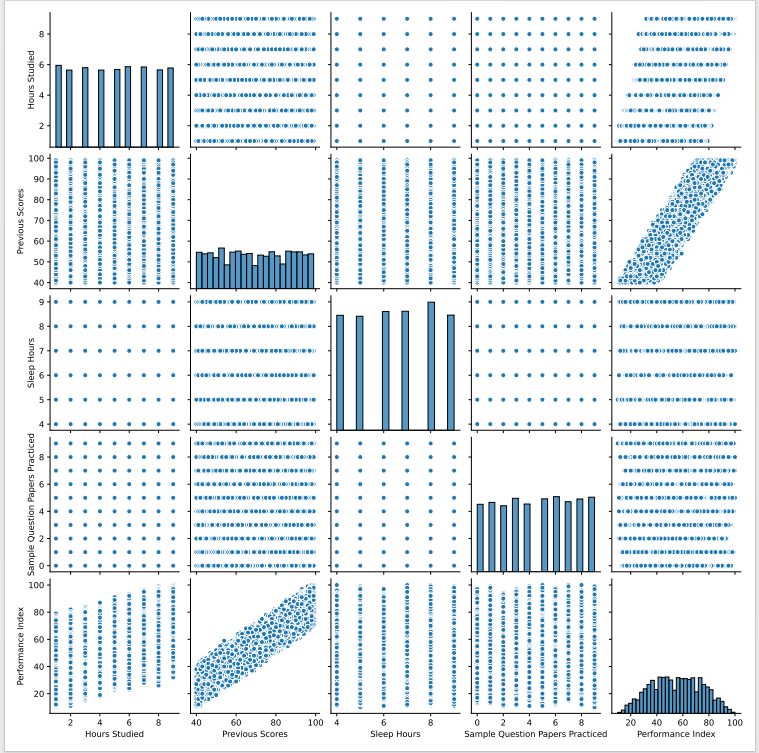
1. 将具体的训练数据导入，进行训练和测试

Plot result：

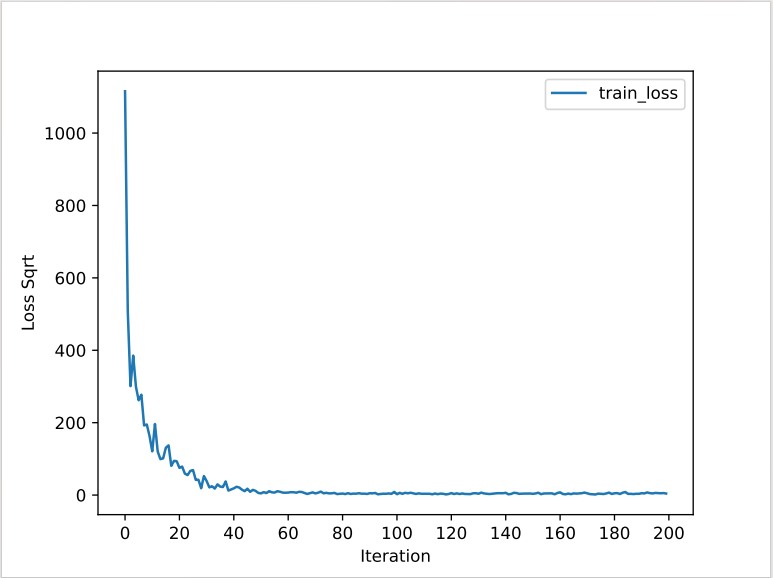
热力图：



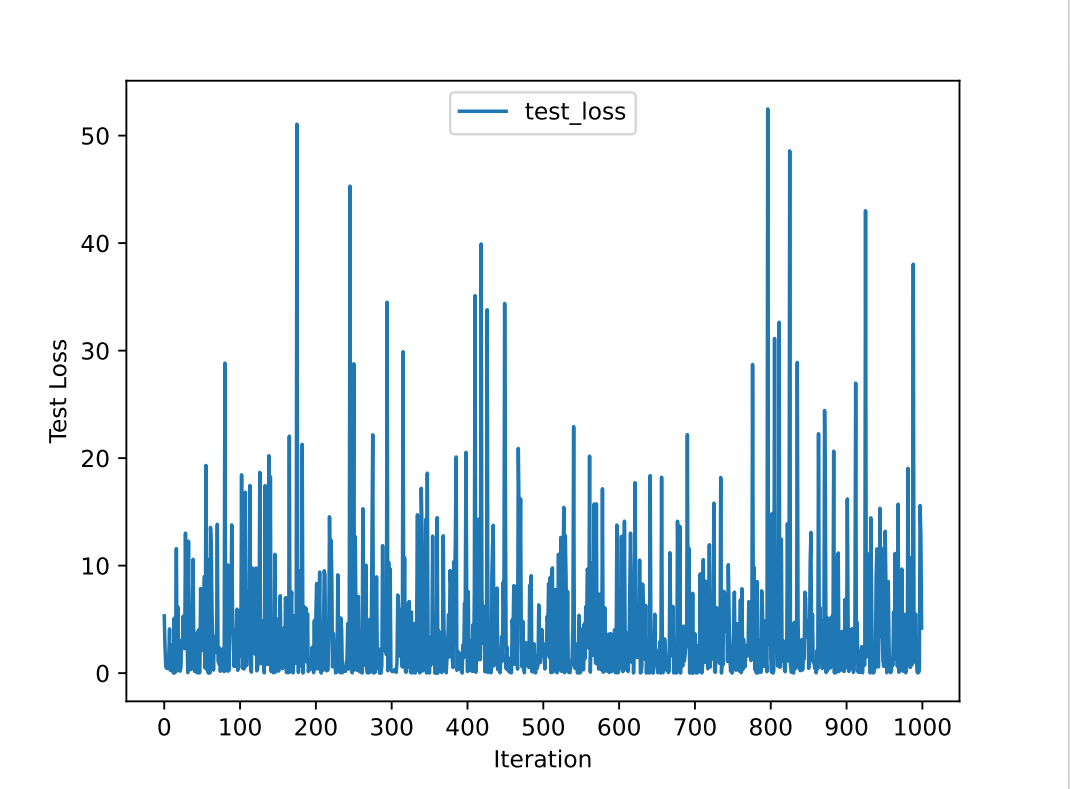
散点矩阵图：



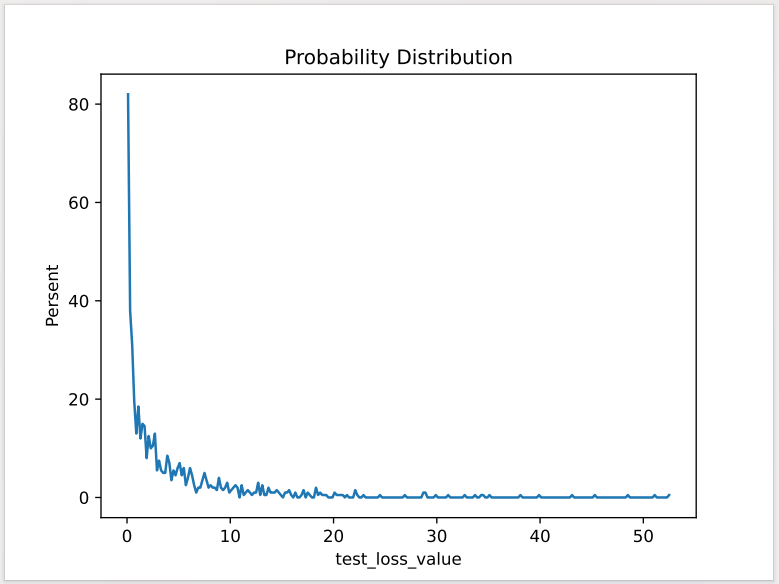
训练集上的loss随轮次变化曲线：



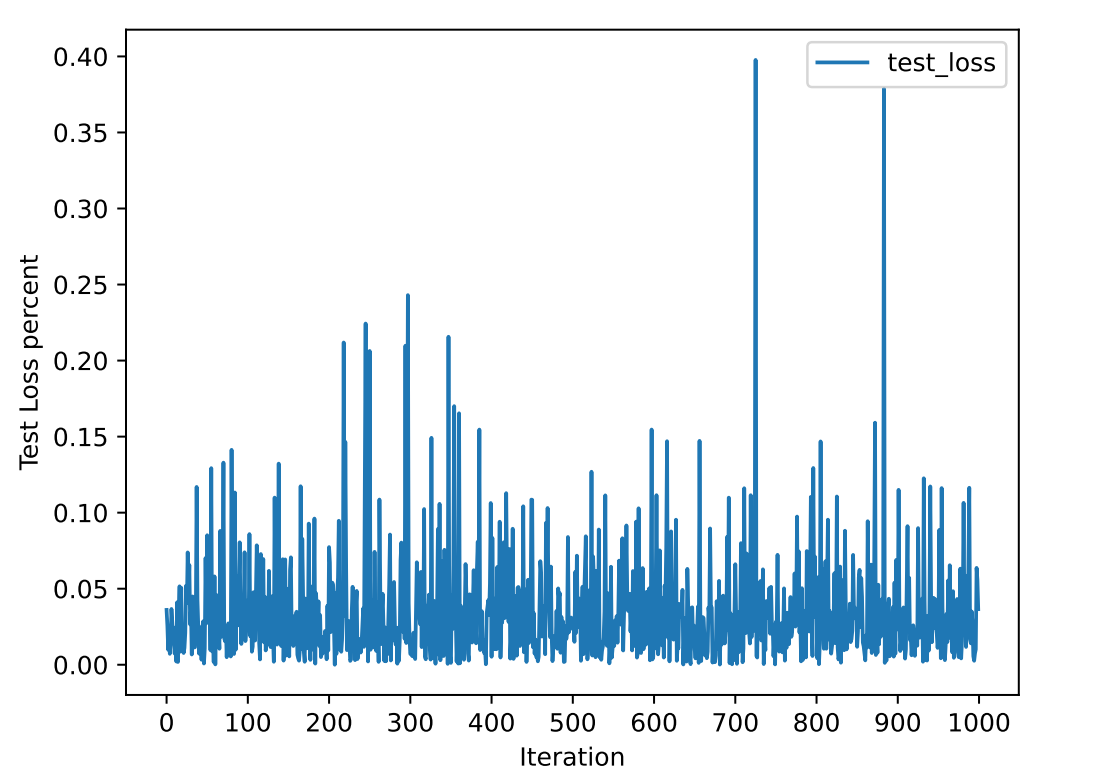
测试集上每个数据的loss函数值：



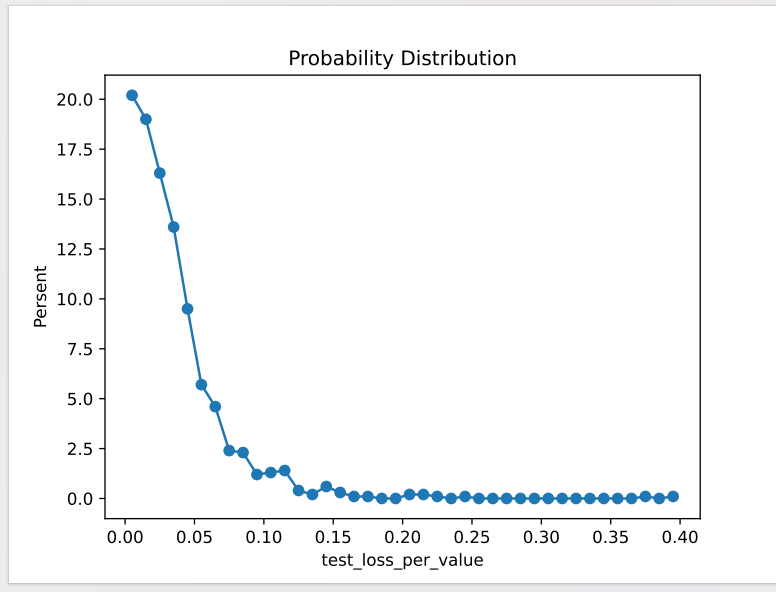
测试集上每个数据的loss函数值的密度分布情况：



测试集上每个数据预测值与真实值偏差占比 abs(y\_hat-y)/y：



测试集上每个数据预测值与真实值偏差占比的密度分布情况 abs(y\_hat-y)/y：



Optimize & Review：

Extra Credit：

1. 手写回归算法：原码

2. 学习率与正则化参数问题：

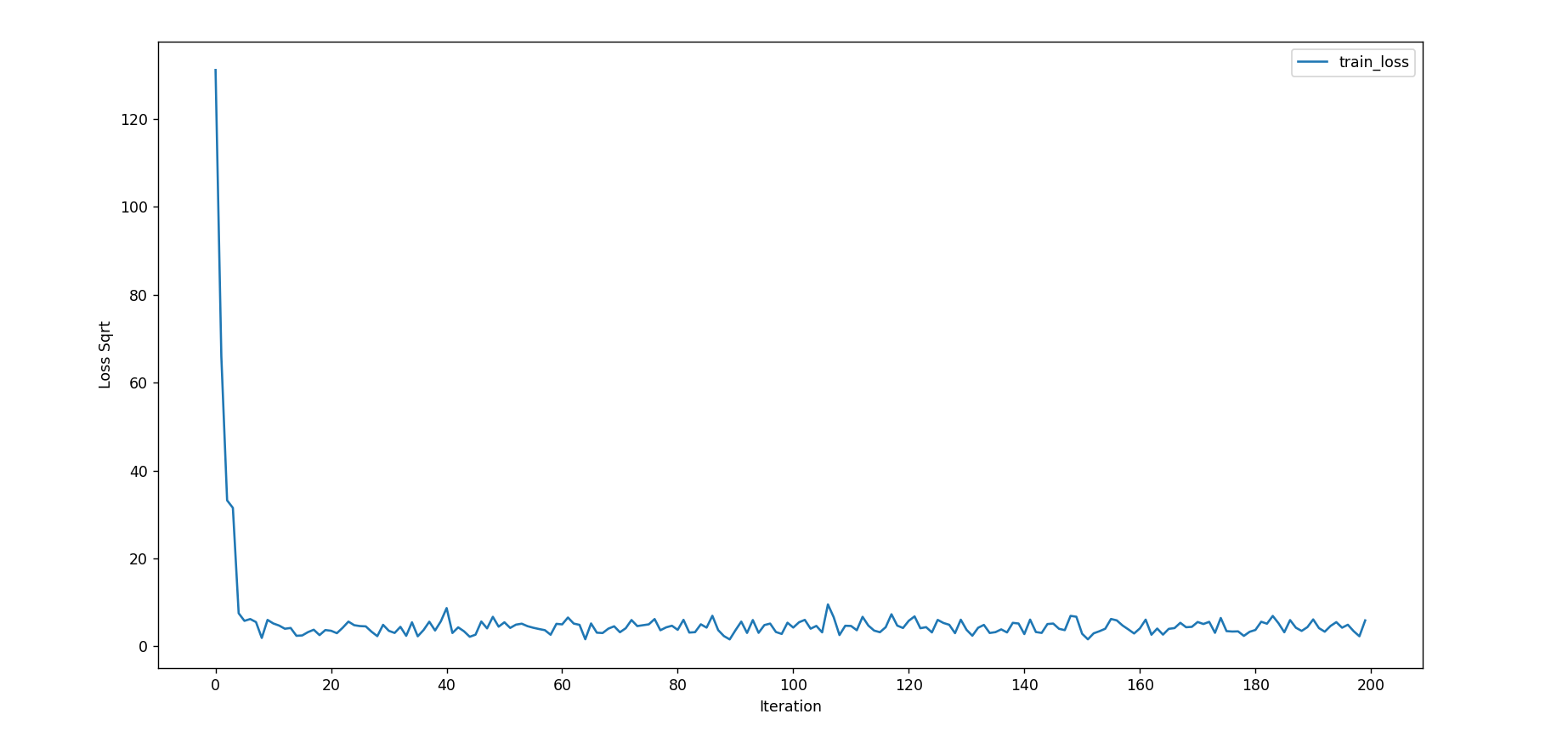
学习率问题：

1. 采用固定学习率进行训练时：

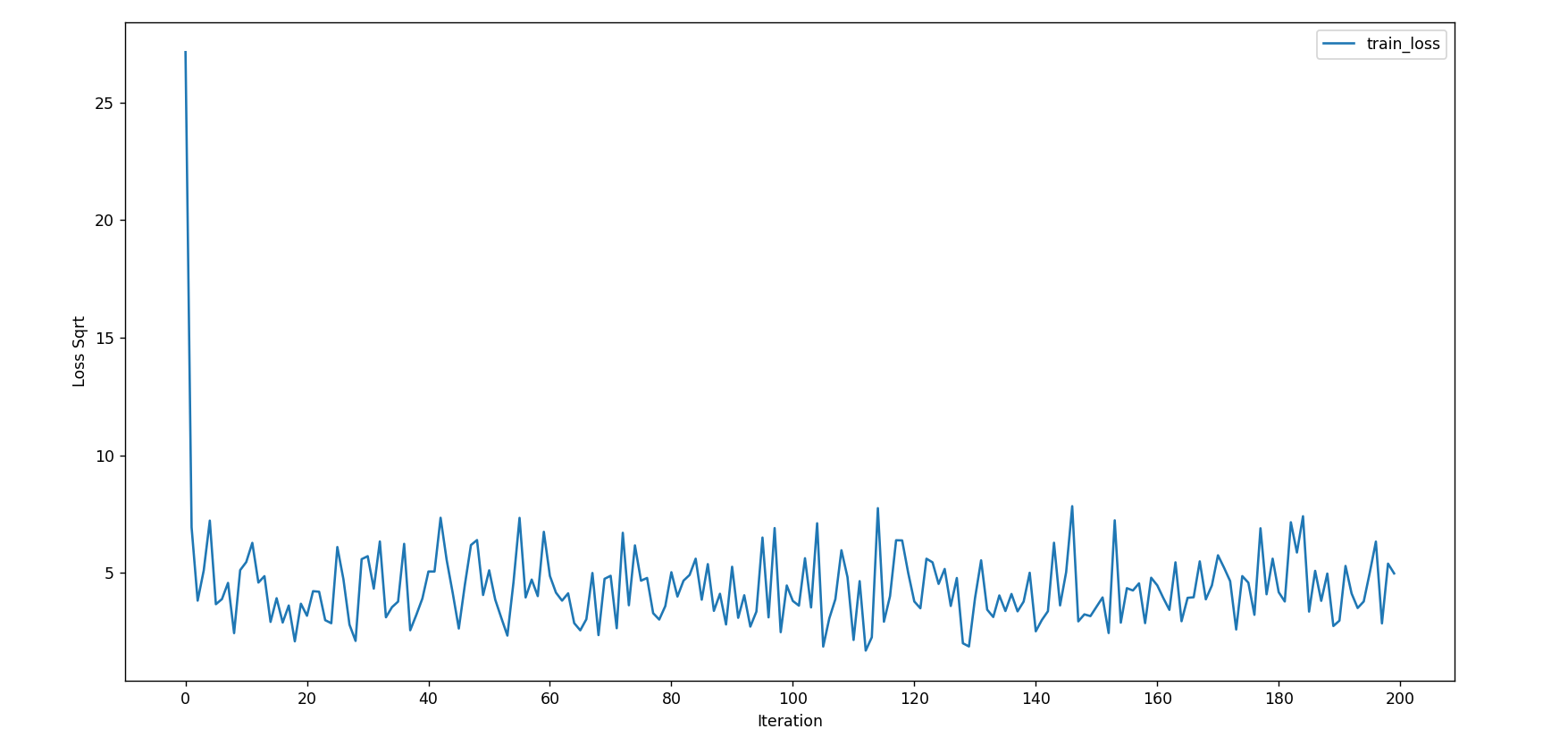
当我们以200轮次进行训练的时候，固定学习率的取值就会有一定的要求：

当学习率比较大的时候，比如当学习率大于0.01的时候，每次更新参数所迈的步伐比较大，在刚开始的时候能够很快的逼近最优模型，loss的函数值能够快速下降，但是当靠近最优解的时候，弊端就产生了，每次步伐都很大，导致迟迟不能逼近最优解，一直在最优解的附近来回波动，随着训练轮次的增多，依旧呈现这样的情况，loss函数值在不停的波动。

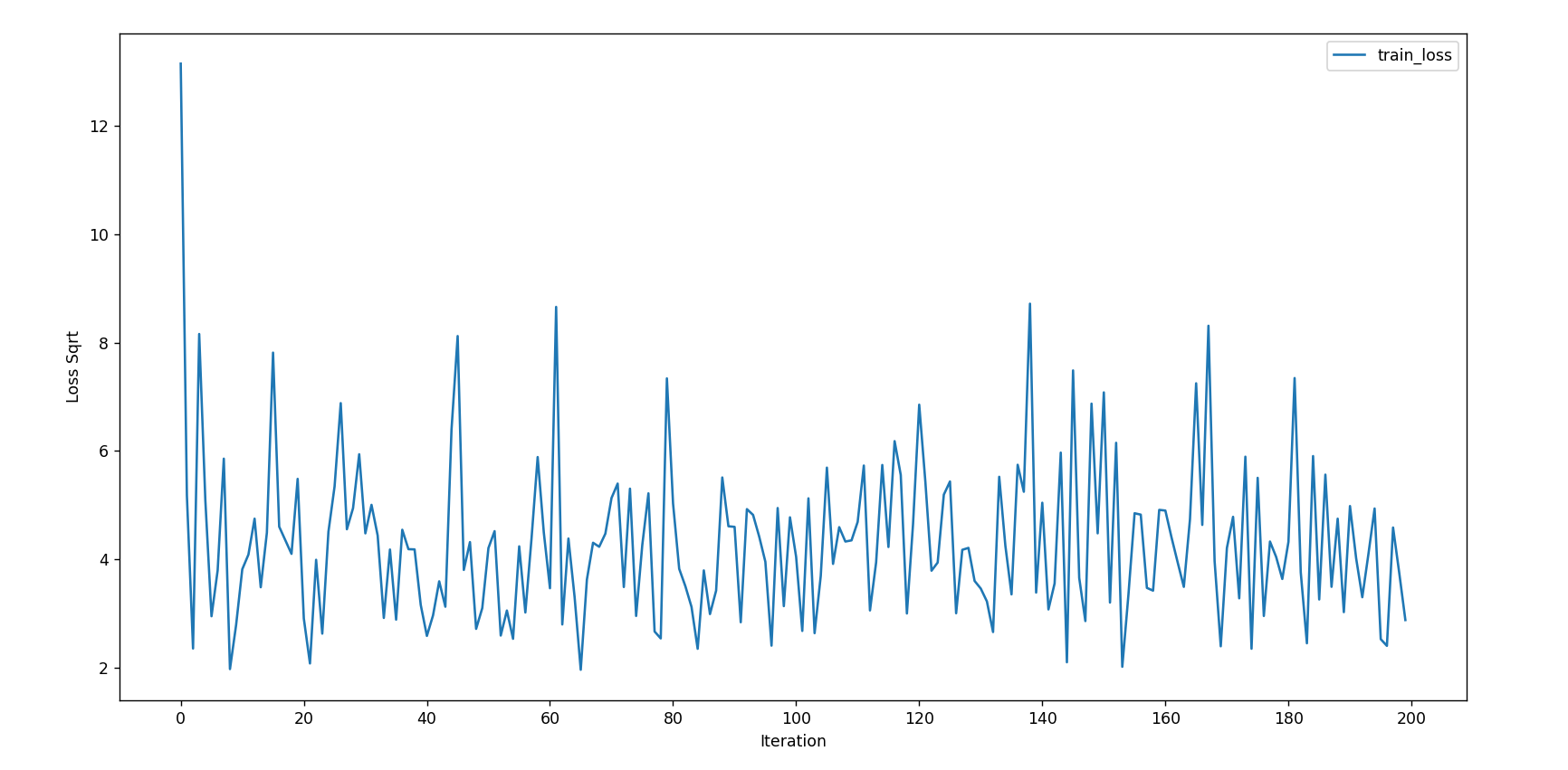
比如当学习率Alpha = 0.01的时候：



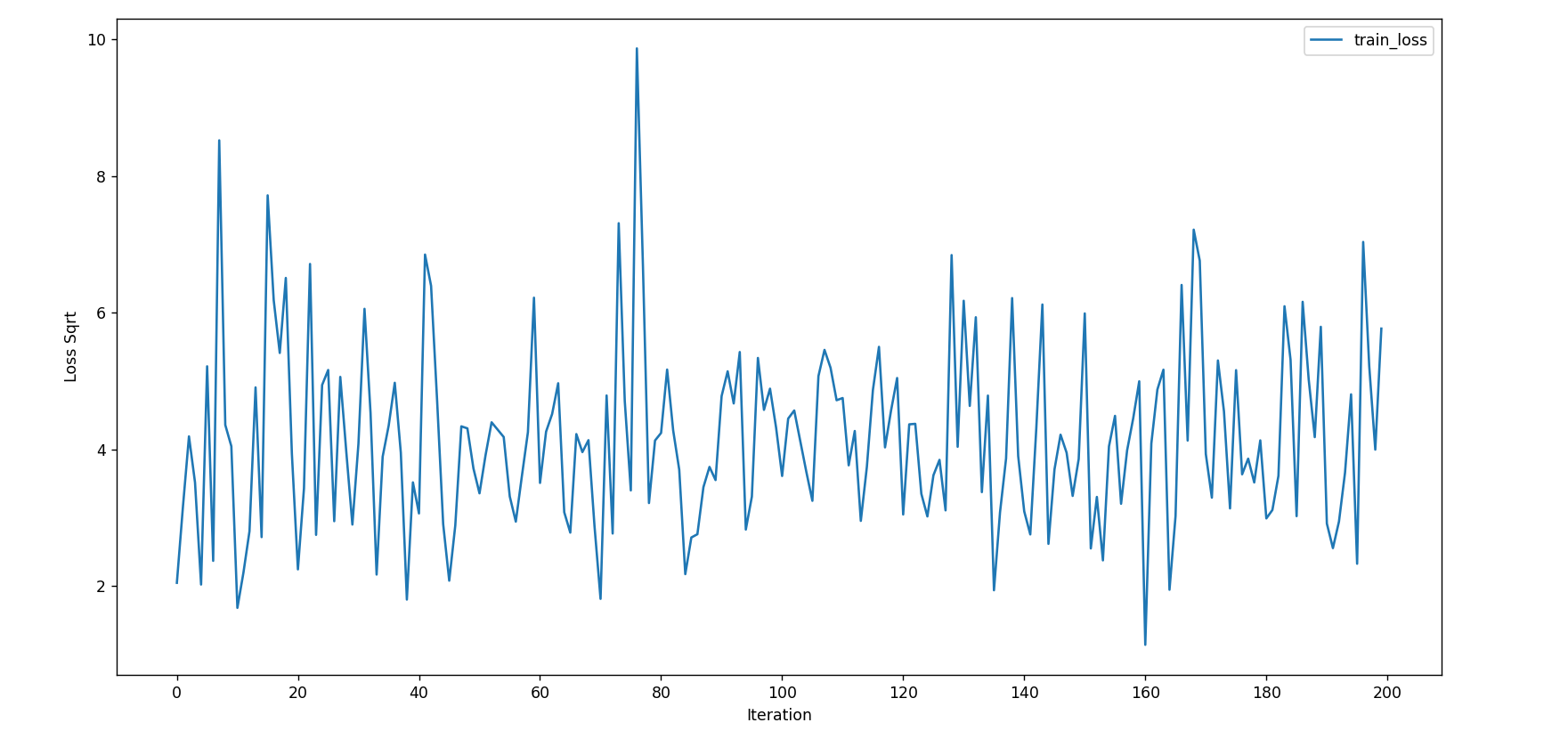
当学习率Alpha = 0.03的时候：



当学习率Alpha = 0.05的时候：



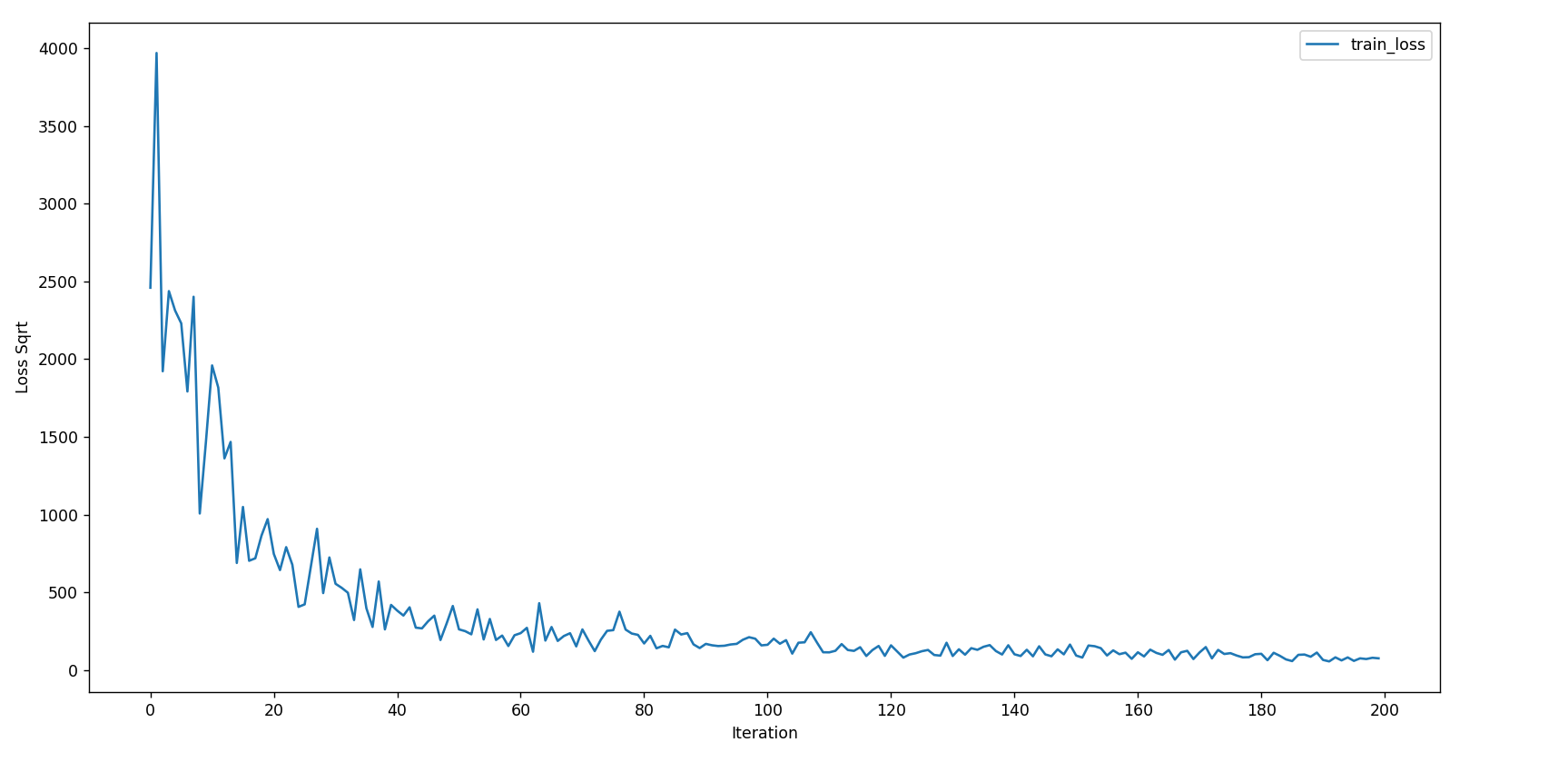
当学习率Alpha = 0.1的时候：



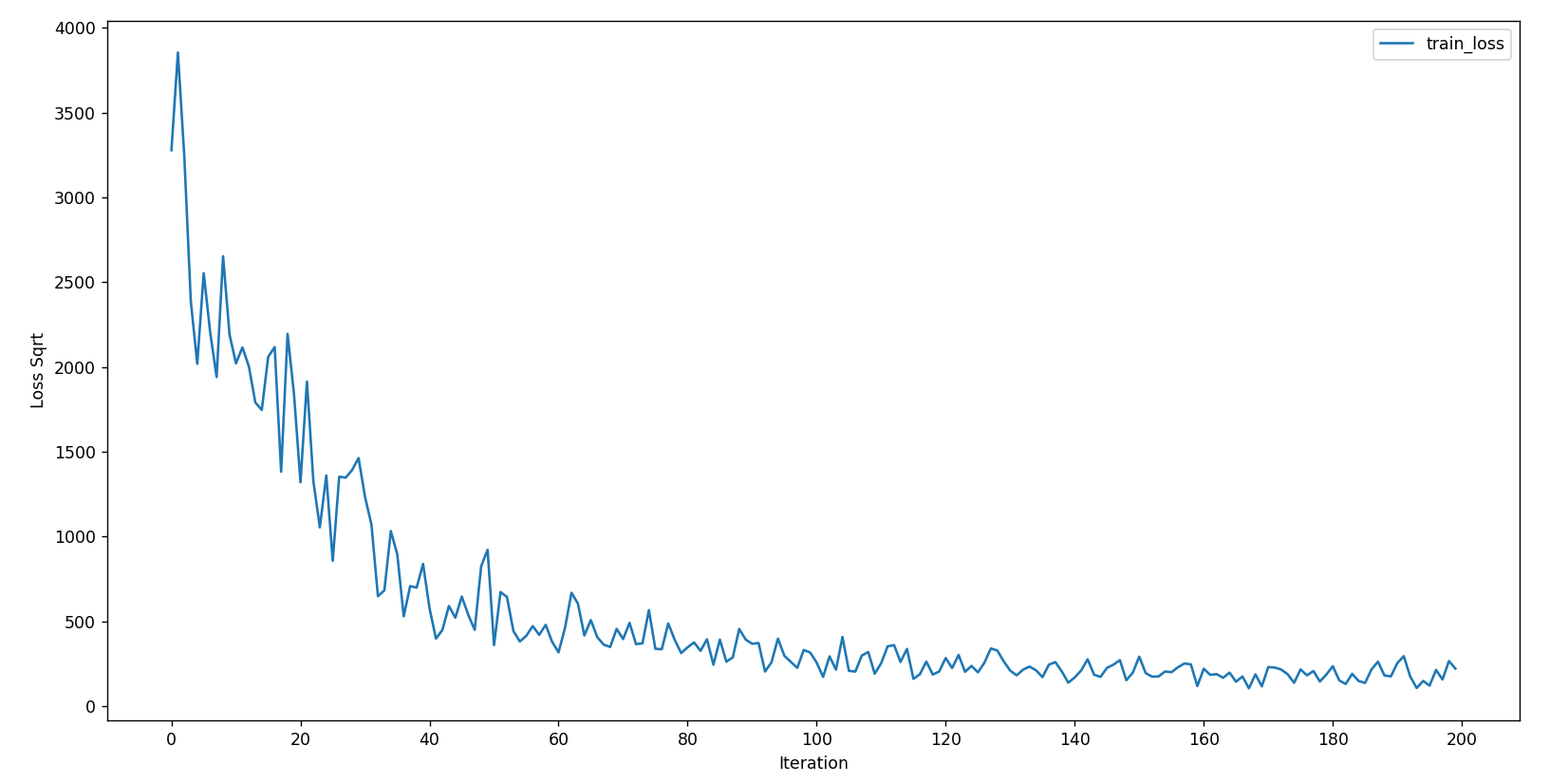
由上我们看到，当学习率大于一定的值的时候，在本次实验中大概是0.01的时候，可能在训练10轮就已经达到一个不错的结果了，之后loss损失值函数在不断地波动，并没有出现趋向于稳定，相反后面轮次的训练在一定程度上属于浪费计算资源。

当学习率比较小的时候，比如当学习率小于0.00001的时候，每次更新参数所迈的步伐比较小，在刚开始的时候需要很多轮次才能逼近最优模型，loss的函数值下降十分缓慢，有时候甚至所有轮次都完成了仍然达不到预期的目标。就算loss函数值经过一段时间的训练下降很多，也还需要很长的时间去逼近最优解。

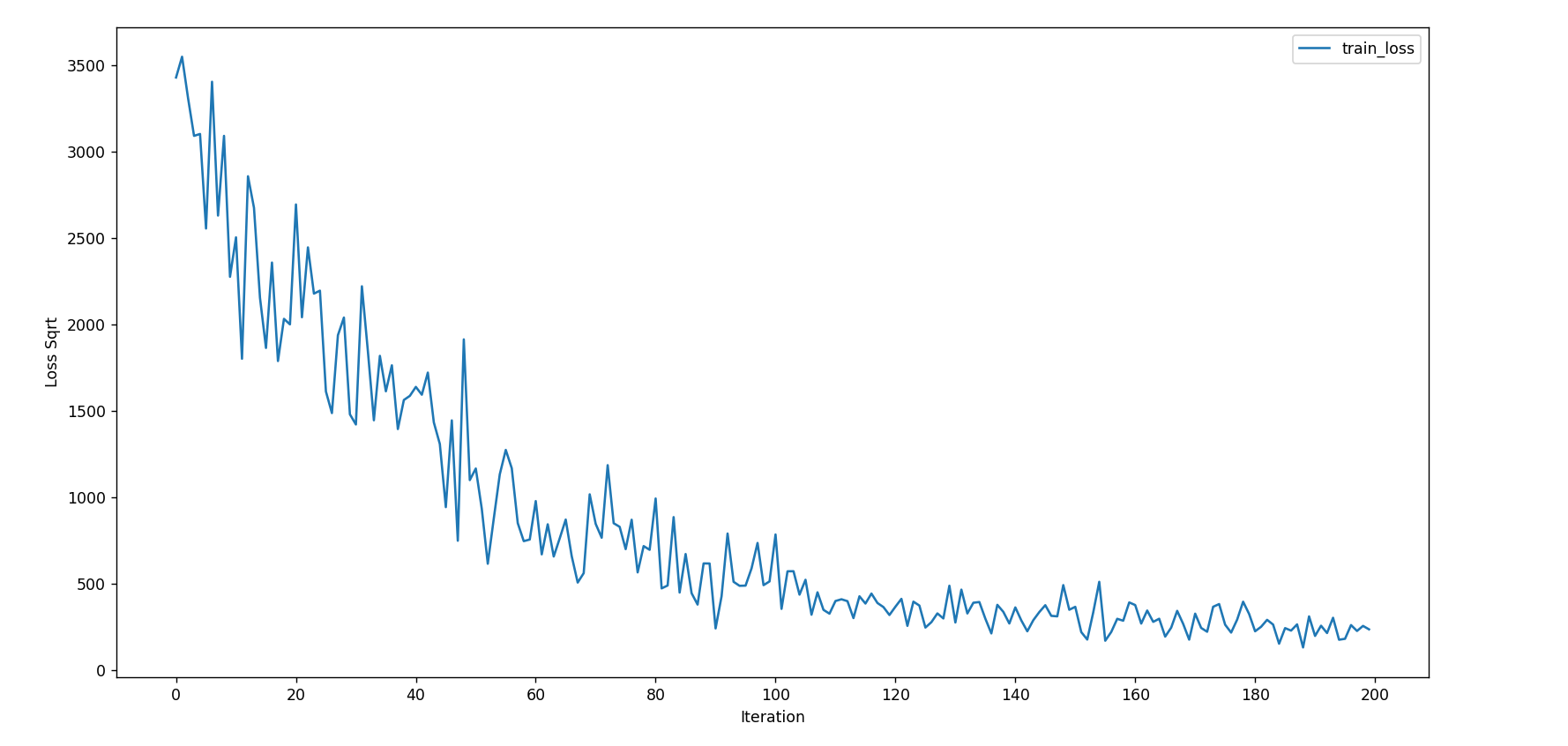
当学习率Alpha = 0.0001的时候：



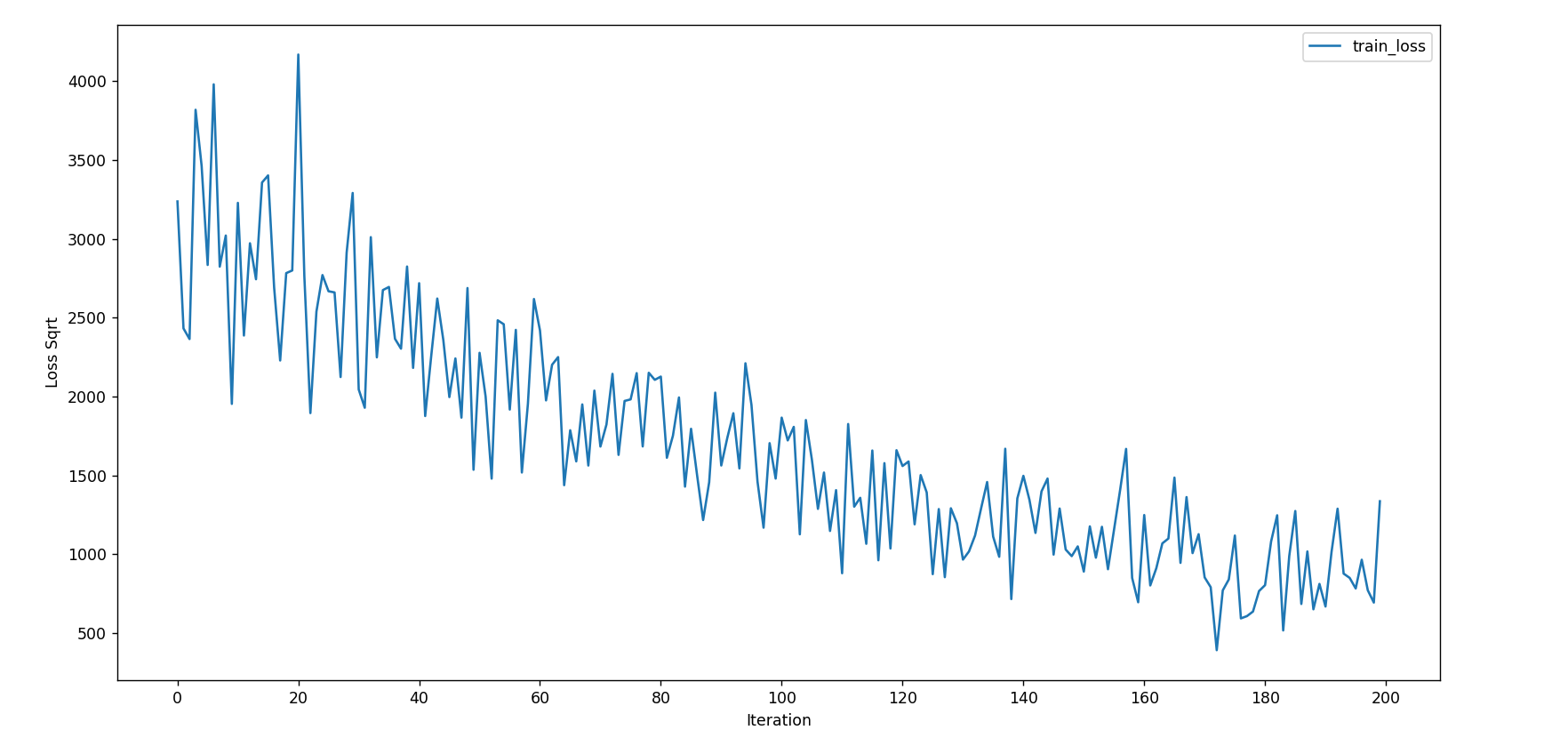
当学习率Alpha = 0.00005的时候：



当学习率Alpha = 0.00003的时候：



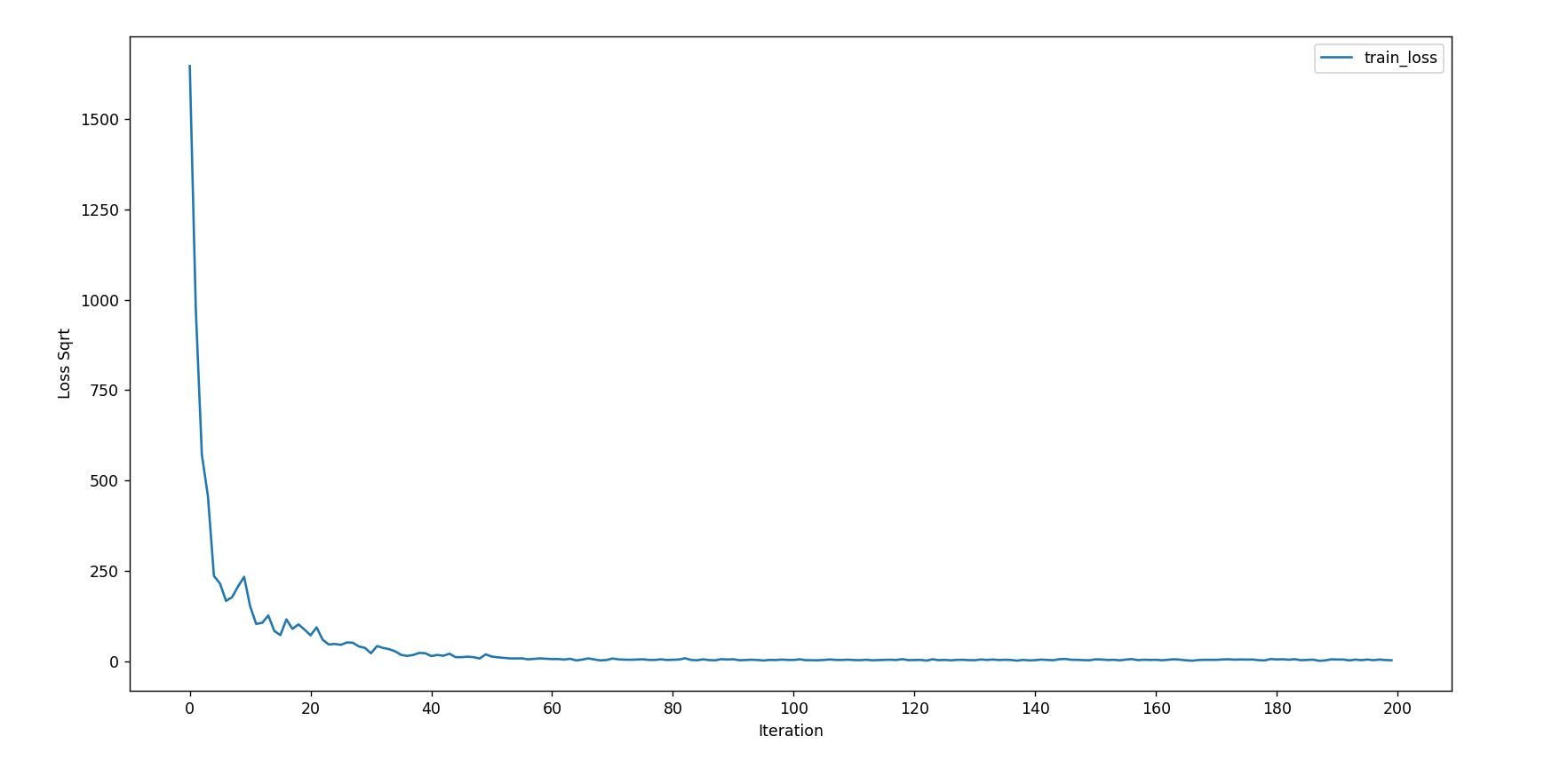
当学习率Alpha = 0.00001的时候：



由上我们看到，当学习率小于一定的值的时候，在本次实验中大概是0.0001的时候，可能训练了150轮还没有出现预期的结果，比如当Alpha = 0.0001时，训练到最后loss函数值才下降到70左右，误差还是很大。

综上我们可以大致知晓本次实验，固定学习率的取值范围大概在0.0001-0.01的一个取值范围之内，我选择的是0.001进行实验。

损失值函数随训练轮次的变换曲线：



（2）采用学习率衰减的方法：

学习率的选择过大会导致迈的步伐较大，在最优解附近不停波动，学习率选择过小又会导致迈的步伐较小，需要花费很长的时间才能到达最优解的附近。因此我们将学习率的取值进行一定的修改，采用学习率随着训练轮次的增加而不断衰减的方法。

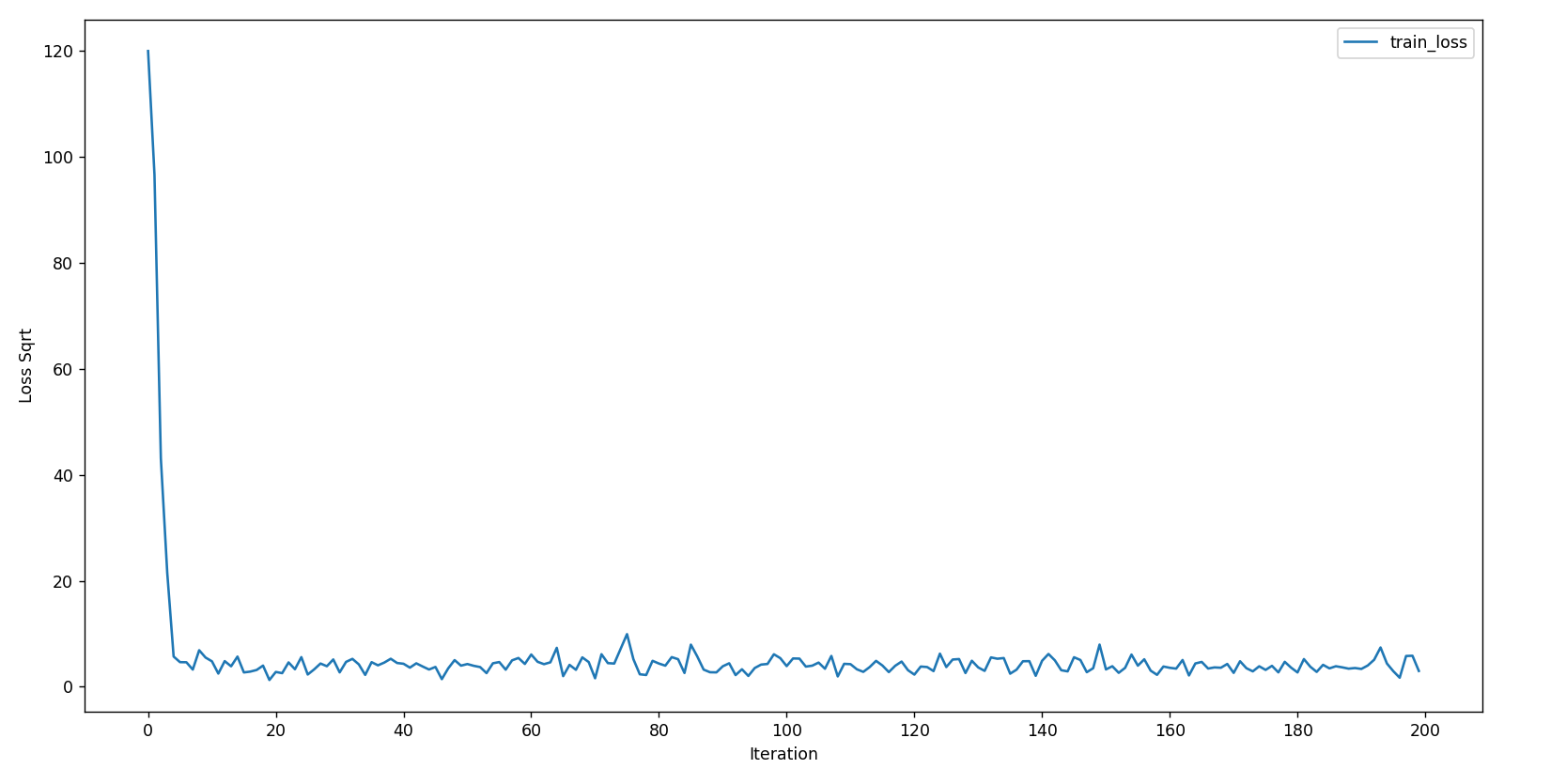
我们修改代码，学习率初始值采用0.01

model.SGD(200, 20, 0.01)

学习率随着轮次的增加，每20轮变为原来的二分之一

if (epoch+1) % 20 == 0:  
 Alpha \*= 0.5

训练效果：



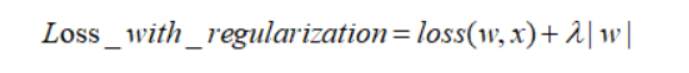
能够很快的逼近最优解，到最后学习率比较小，参数几乎不再变化，随验证样本的变化loss在2-4之间波动，变化较小。

正则化参数问题：

比较常用的两个正则化方法是L1正则化和L2正则化。

（1）L1正则化方法：

公式：



然后我们的梯度下降算法就要变为：

# 如果你 使用 L1 正则化来修改损失函数  
# Lambda\_L1 = 0.1  
# delta\_w += Alpha \* ((y\_hat - y) \* x.reshape(-1, 1) + Lambda\_L1 \* np.sign(self.W))

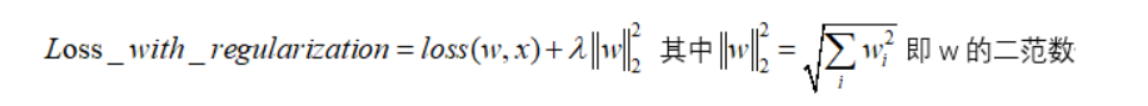
在更新W的参数时，需要加一个正则化参数Lambda\_L1乘以W中对应值的符号。

正则化项的用处是通过控制模型的复杂程度，来防止过拟合，比如L1正则方法就会让W中的参数向0的方向去靠近，使得最终得到的W参数向稀疏矩阵上靠拢，以降低模型的复杂性。

但是本次实验过于简单，特征值也比较少，最终加入正则化项对结果的影响并不大，loss函数的变化程度也不大。

1. L2正则化方法：

公式：



然后我们的梯度下降算法就要变为：

# 如果你 使用 L2 正则化来修改损失函数  
# Lambda\_L2 = 0.1  
# if (math.sqrt(np.sum(self.W\*\*2)) ) == 0:  
# delta\_w += Alpha \* ((y\_hat - y) \* x.reshape(-1, 1))  
# else :  
# delta\_w += Alpha \* ((y\_hat - y) \* x.reshape(-1, 1) + Lambda\_L2 \* self.W / math.sqrt(np.sum(self.W\*\*2)))

在更新W的参数时，需要加一个正则化参数Lambda\_L1乘以W中对应值除以W的二范数。

正则化项的用处是通过控制模型的复杂程度，来防止过拟合，而L2正则方法就会让W中的参数变小，使得权重更加稳定平滑，以降低模型的复杂性。

但是本次实验过于简单，特征值也比较少，最终加入正则化项对结果的影响并不大，loss函数的变化程度也不大。

输入变量参数之间的关联性问题：

我们在确定了X和Y值的时候，通过研究变量之间的关联性，我们能够筛选出对预测Y值用处不大的特征，将它们去掉可能会使得最终的模型准确率提高。

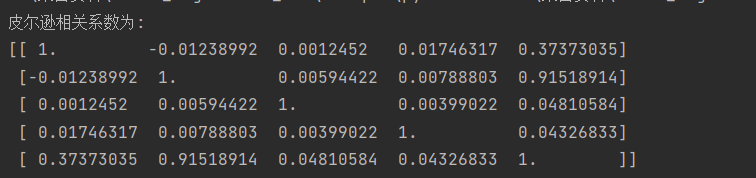
1. 皮尔逊相关系数：

我们在数据处理阶段对数据计算皮尔逊相关系数，查看各个变量之间的相关关系：

查找资料我们可以知道皮尔逊相关系数的绝对值越接近1，两个变量之间相关性越强。当在0.7-1的时候呈现强相关关系，0.3-0.7时呈现中等相关关系，0-0.3呈现弱相关关系

我们修改代码，计算皮尔逊相关系数：

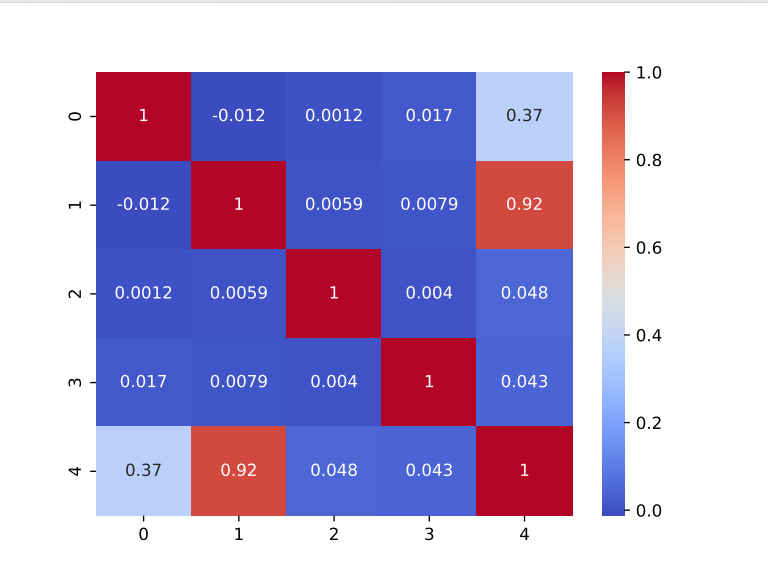
#计算线性相关系数 皮尔逊相关系数 观察4个特征与目标值之间的关系  
data = np.array(self.data)  
data\_R = np.corrcoef(data.T)  
print(f'皮尔逊相关系数为: \n{data\_R}')



我们看到特征二与Y相关性最强达到0.91，其他特征处于中等相关关系，在训练的时候都可以使用，并不需要筛选数据。

（2）热力图：

我们将上面的皮尔逊相关系数打印出来，也就变成了热力图：



（3）散点矩阵图：

还有一个衡量相关性的图是散点矩阵图，它将任意两个变量的取值分别作为坐标，通过点的分布情况观察各个变量之间的相关关系，与上面是类似的。

