Report for Mnist Classification

展子航 计21 2022010745

Zihang Zhan

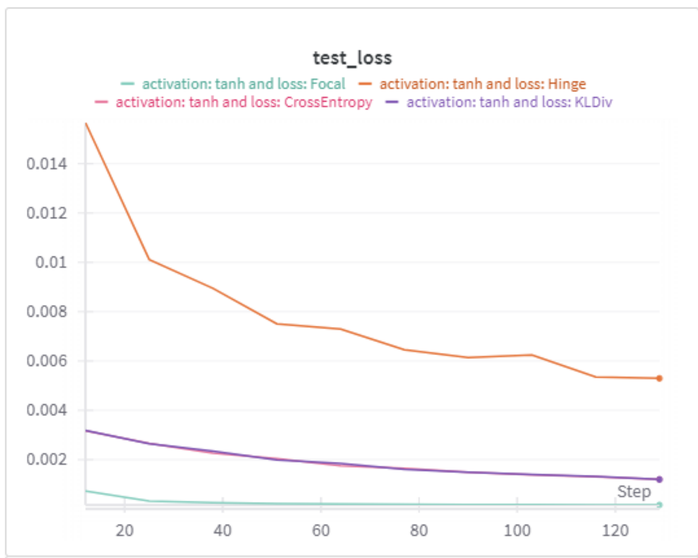
Created on September 25 | Last edited on September 26

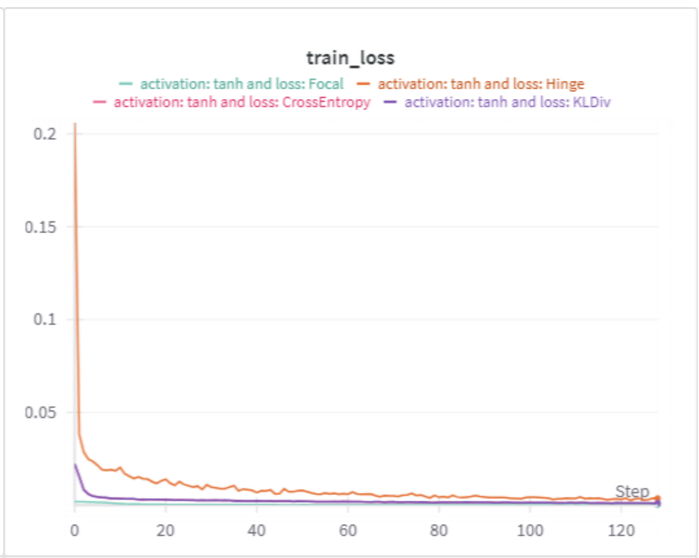
初始参数设定

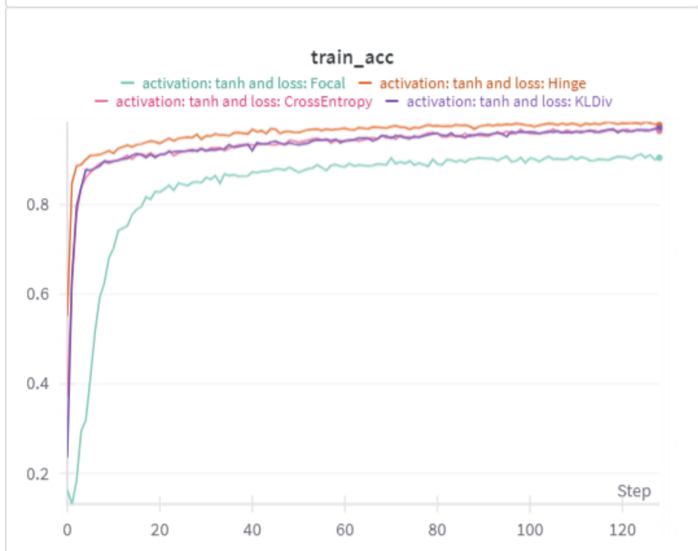
```
default_config = {
    'learning_rate': 1e-4, # 学习率
    'weight_decay': 1e-5, # decay 优化超参
    'momentum': 0.9, # 动量 优化超参 根据课件调成的0.9
    'batch_size': 100, # batch_size 默认
    'max_epoch': 10, # epoch 训练次数
    'disp_freq': 50, # disp_freq 训练情况打印频次
    'test_epoch': 1 # 测试频次
}

# Linear(784:128, 0.01) -> Activation -> Linear(128:10) -> Loss

HingeLoss.margin = 5
```

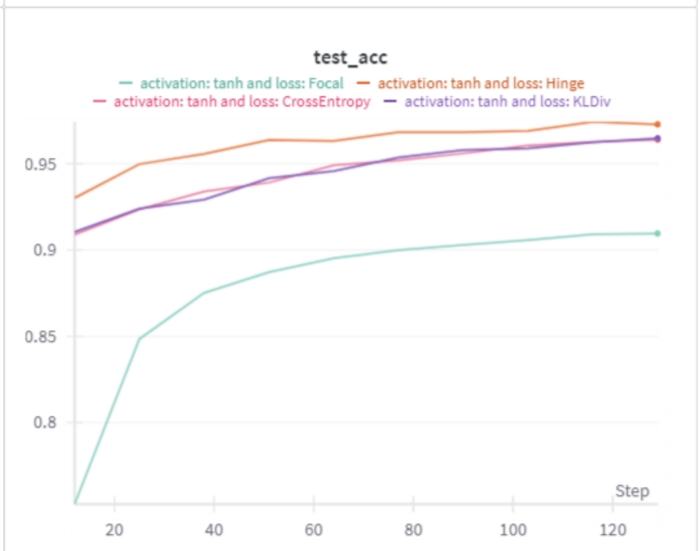
默认状态下运行网络

在初始设定下，使用 tanh 作为激活函数







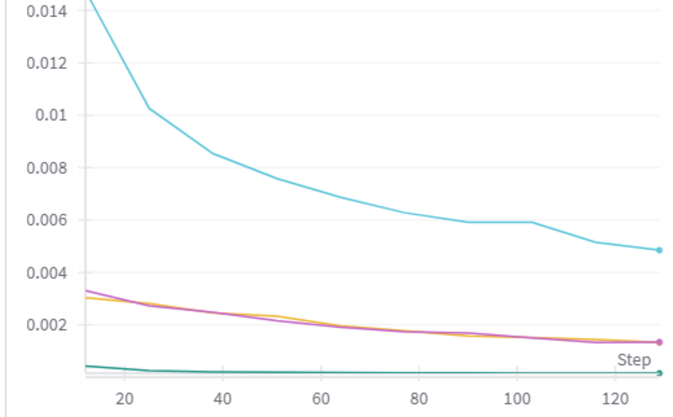


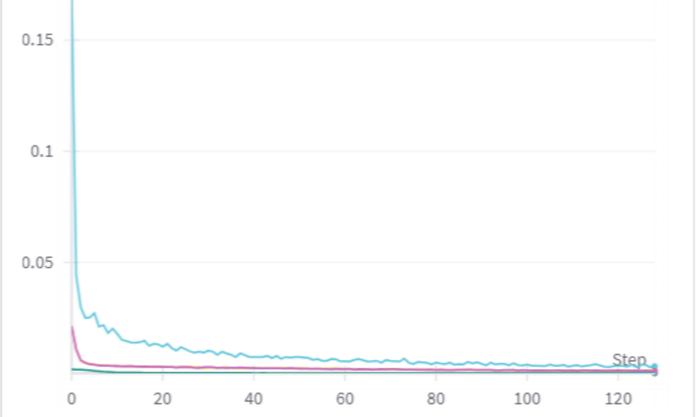
如上数据可见：

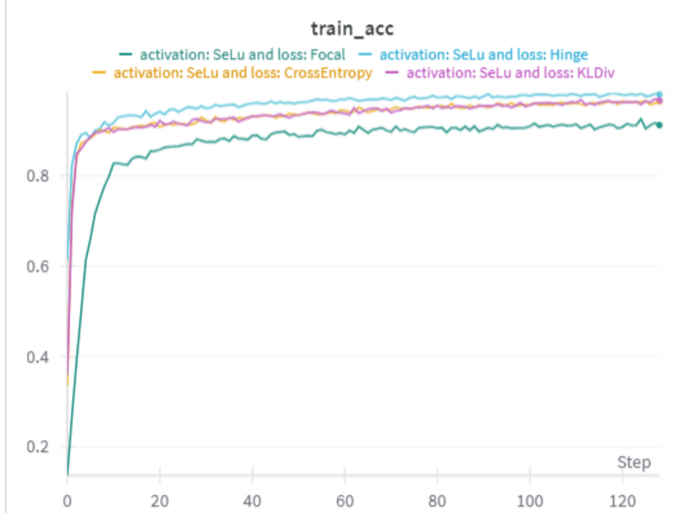
- 在 `tanh` 作为激活函数的情况下，模型的表现都是稳定上升的，且除 `Focal` 以外的损失函数组合最终 `test_acc` 都达到了 95% 以上；
- 损失函数在 `test_acc` 的最终表现为 Hinge (97.28%) > KLDiv (96.47%) > CrossEntropy (96.39%) > Focal (90.96%)；
- 其中`Focal`的表现显然弱于其他三者，考虑调整超参数和变更网络结构。

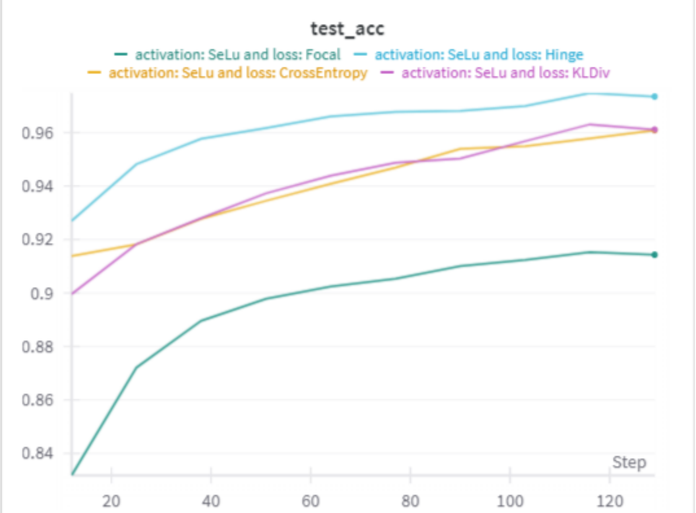
Run set 4

在初始设定下，使用 selu 作为激活函数







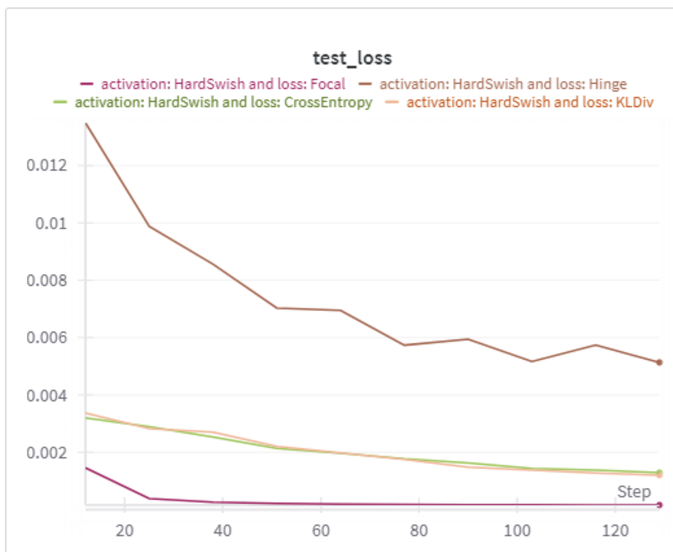


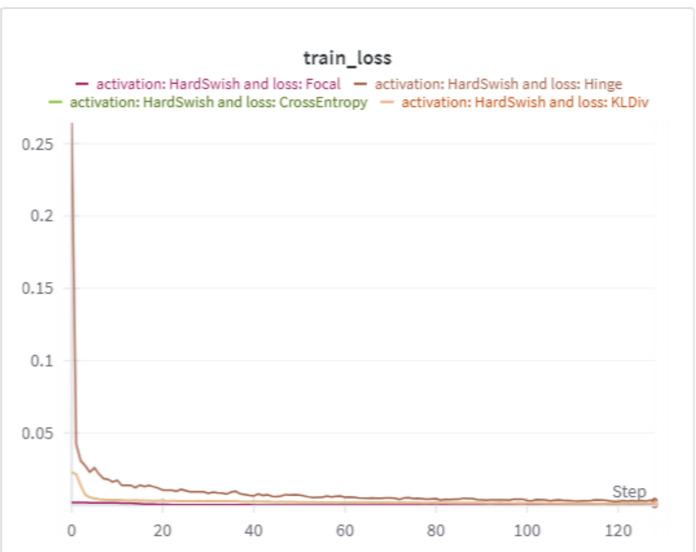
如上数据可见：

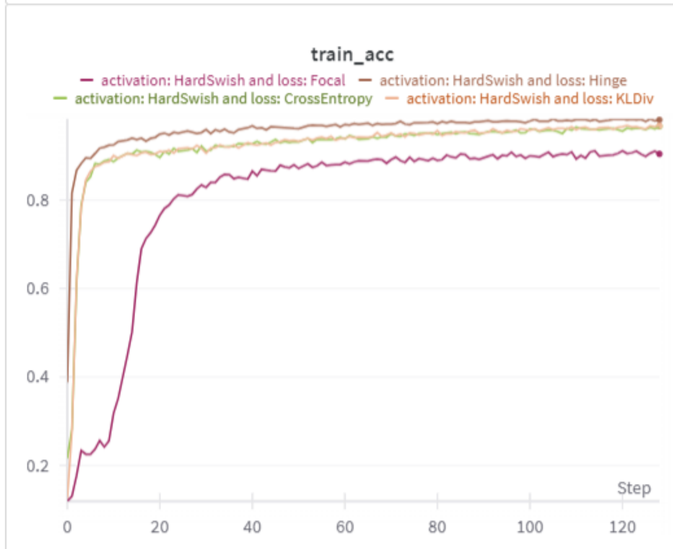
- 在 `SeLu` 作为激活函数的情况下，整体表现与 `tanh` 类似，且除 `Focal` 以外的损失函数组合最终 `test_acc` 都达到了 95% 以上；
- 不同损失函数在 `test_acc` 的最终表现为Hinge (97.35%) > KLDiv (96.11%) > CrossEntropy (96.09%) > Focal (91.43%)；
- 对比 `Tanh` 作为激活函数，会发现前三者有略微的下降，`Focal` 有略微的上升；
- 其中`Focal`的表现显然弱于其他三者，考虑调整超参数和变更网络结构。

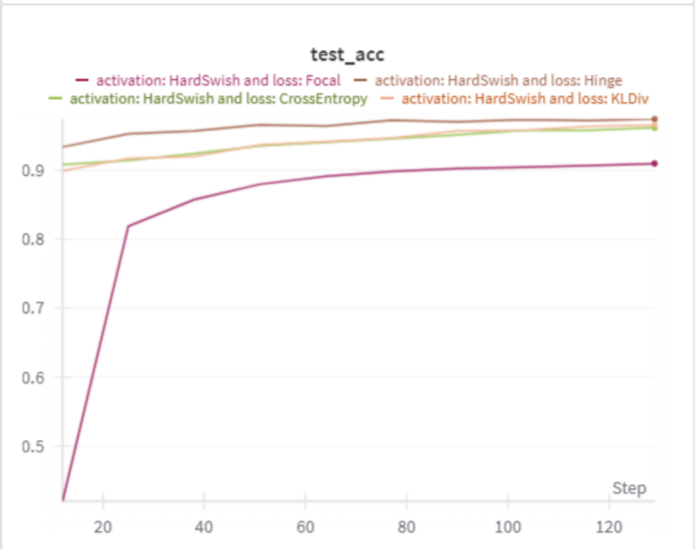
Run set 4

在初始设定下，使用 HardSwish 作为激活函数









如上数据可见：

- 在 `HardSwish` 作为激活函数的情况下，除 `Focal` 以外的损失函数相比其他两个激活函数皆在 `test_acc` 表现稍好；
- 最终表现为 Hinge (97.43%) > KLDiv (96.52%) > CrossEntropy (96.16%) > Focal (90.97%)；

- 而 `Focal` 的表现变得略差。

Run set 4

损失函数总结

经过三种不同激活函数的测试，四者中，`Hinge` 表现最好，而 `KLDiv` 与 `CrossEntropy` 二者的表现类似，`Focal` 表现欠佳，我分析的原因为：

- 我对 `Hinge` 做了一些特殊优化，让正确的 output 拥有了更高的 reward；
- `KLDiv` 与 `CrossEntropy` 二者的正向结构类似，反向传播的导数更是一模一样；
- `Focal` 需要的超参数可能与其他损失函数有区别，需要对其进行额外的调整。这也是我进行超参数调整的动机所在。
- `Focal` 需要的网络结构可能与其他损失函数有区别，需要对其进行额外的调整。这也是我进行网络调整的动机所在。

同时我们从 12 组实验的数据中可以看出 FocalLoss 的特征：**其算出来的数本来就很小**。因此，我们将从 FocalLoss 入手，观察一些参数的调整对其的训练影响。

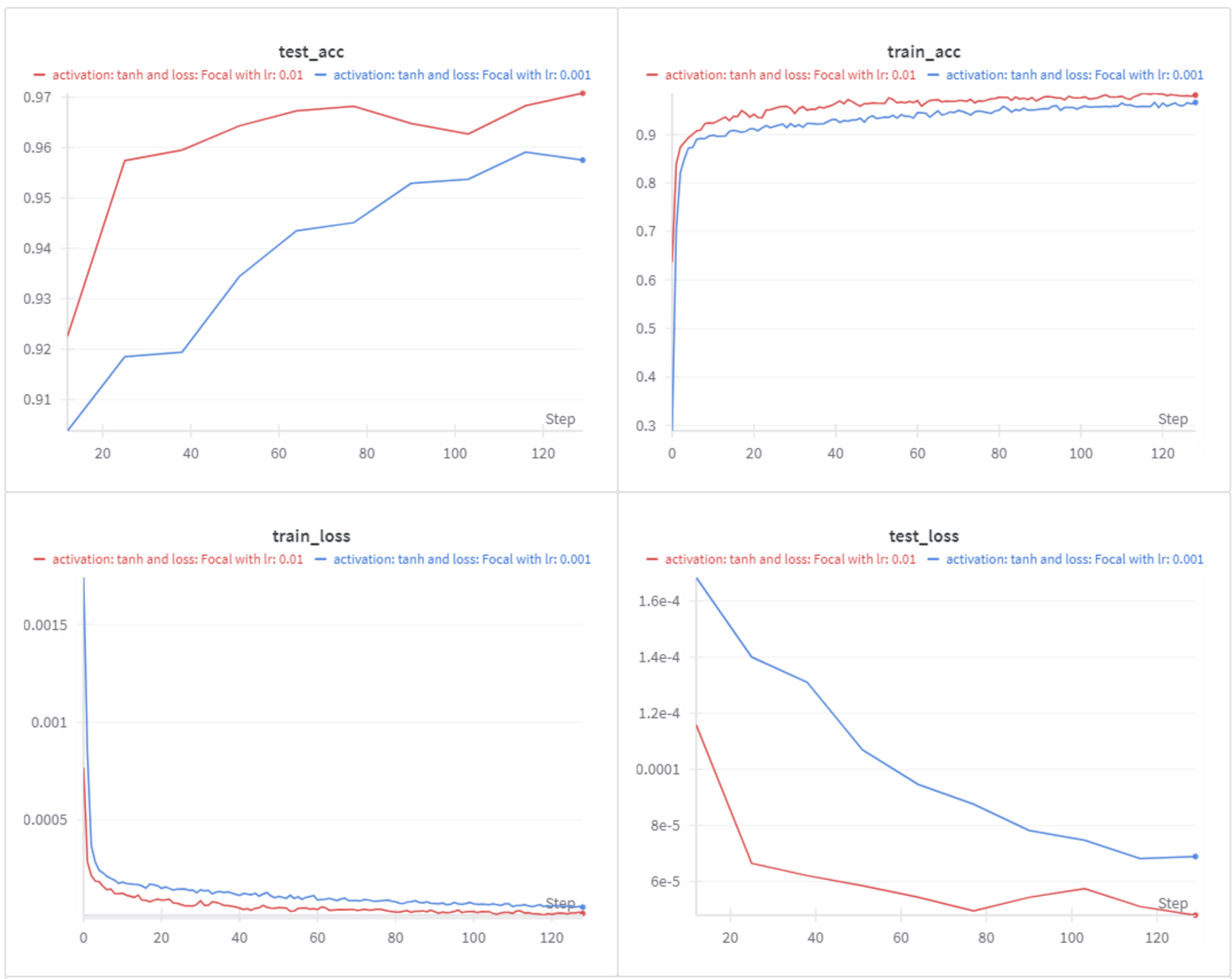
对 FocalLoss 的特定超参数调整

我考虑了以下三种方式对 `FocalLoss` 进行调整：

- 因为 Loss 太小，所以调整 `learning rate`，让其梯度更明显；
- 调整网络结构，一层变两层；
- 调整 `max_epoch`，训练更长时间，曲线救国解决 Loss 问题。

同时，由于在所有激活函数下都有相同的问题，因此我选择固定激活函数为 tanh 。

调整learning rate后的结果

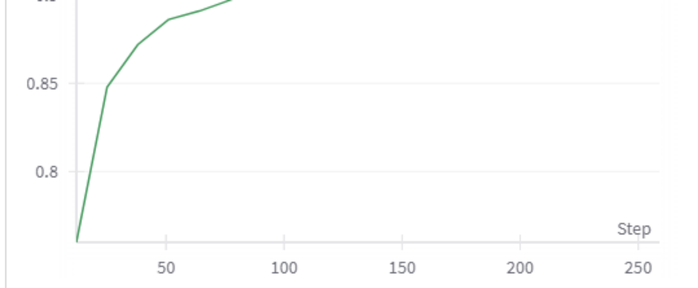


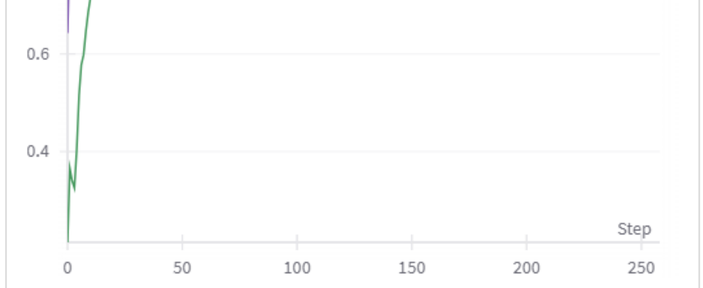
由图可见，显然我对 loss 太小的分析是正确的，整个模型在四个维度上的表现都更好了。

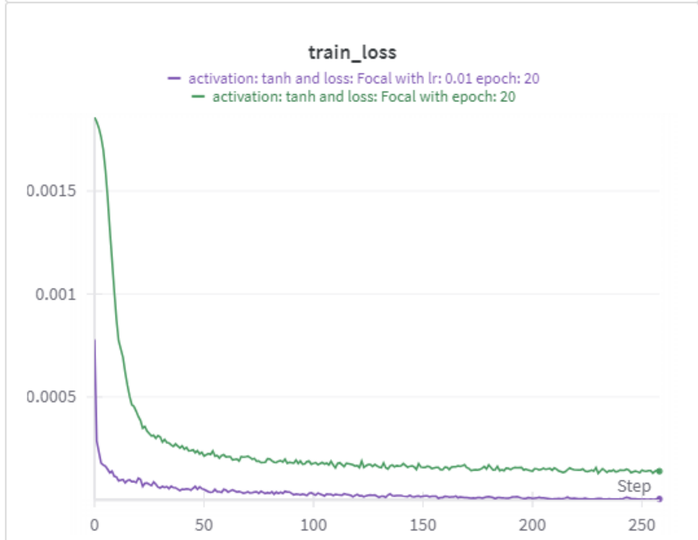
Run set 2

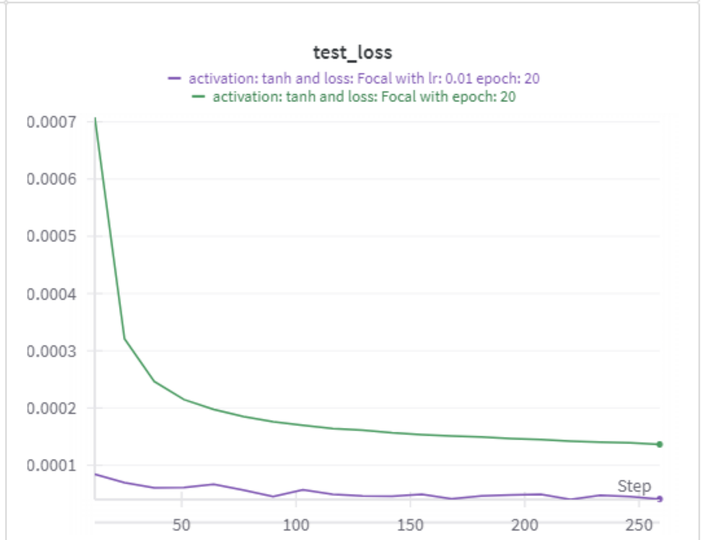
有了以上调整learning_rate的经验，我们将epoch的调整和lr的调整合并起来与单epoch调整对比来看。

只调整epoch和同时调整learning rate后的结果





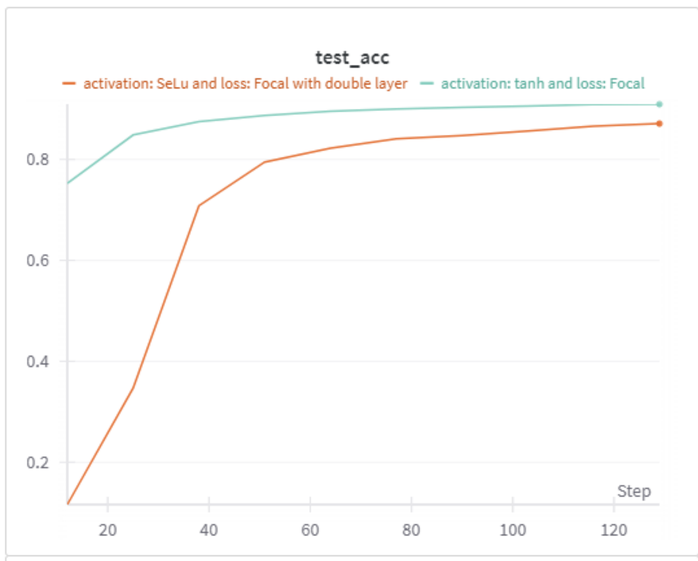


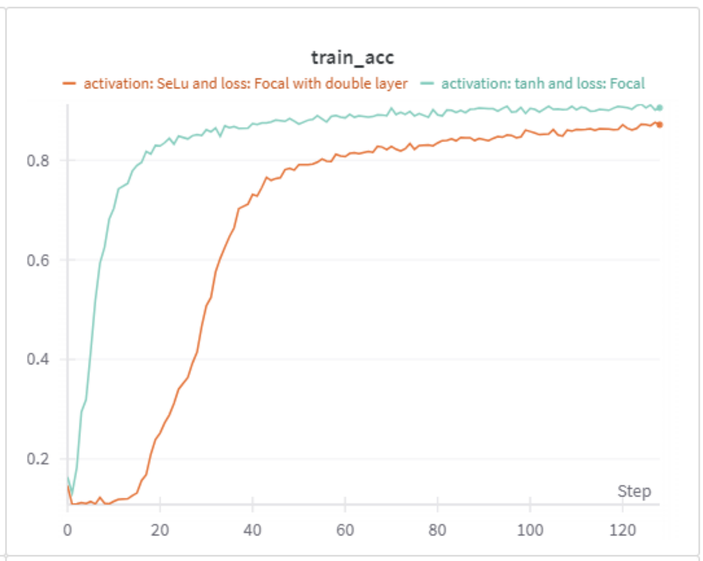


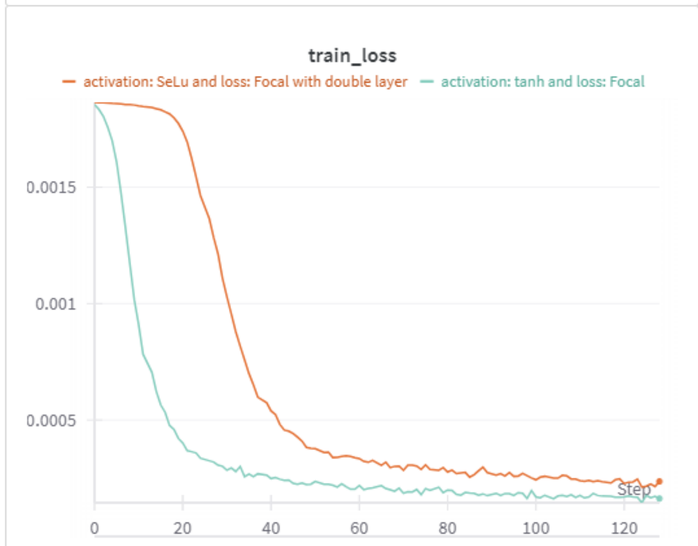
这下更是看出全线优势了，可见对于lr的调整比只拉长训练时间有效率得多，能够在第二维减少训练时间。

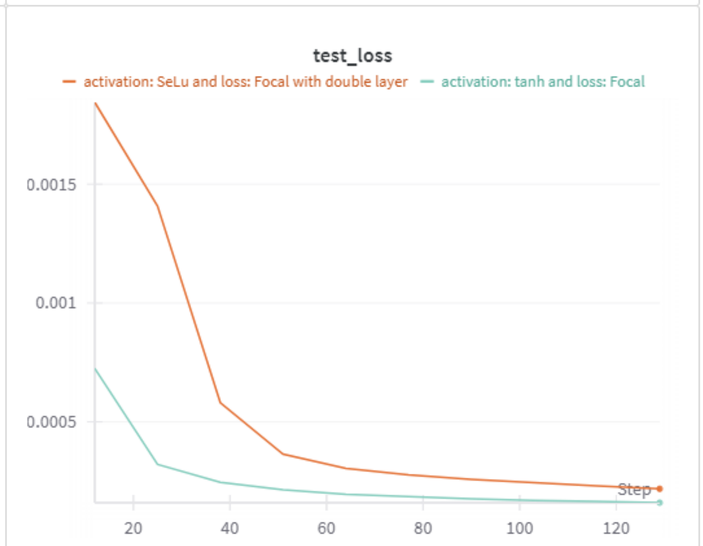
Run set 2

修改隐藏层数的结果





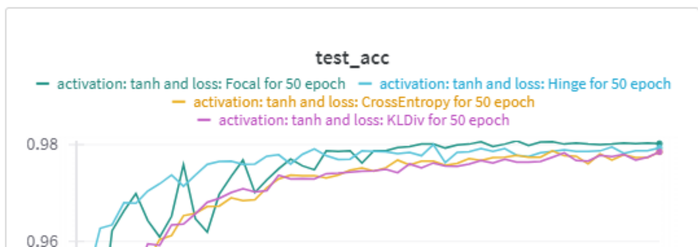


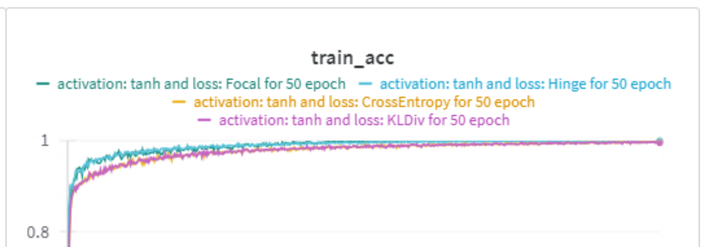


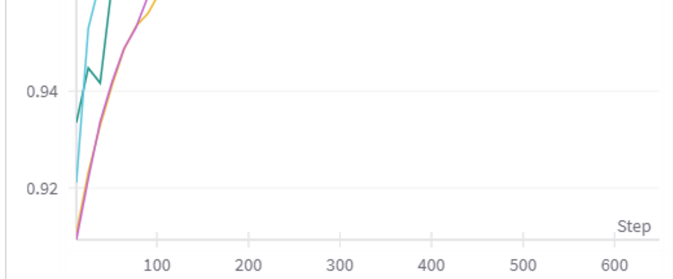
双层神经元反而使效果变差了，推测是双层神经元参数过多导致的收敛慢和过拟合。

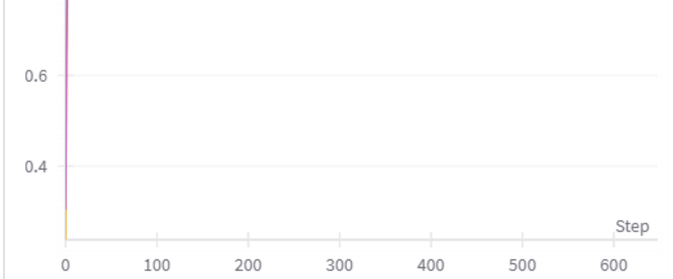
Run set 2

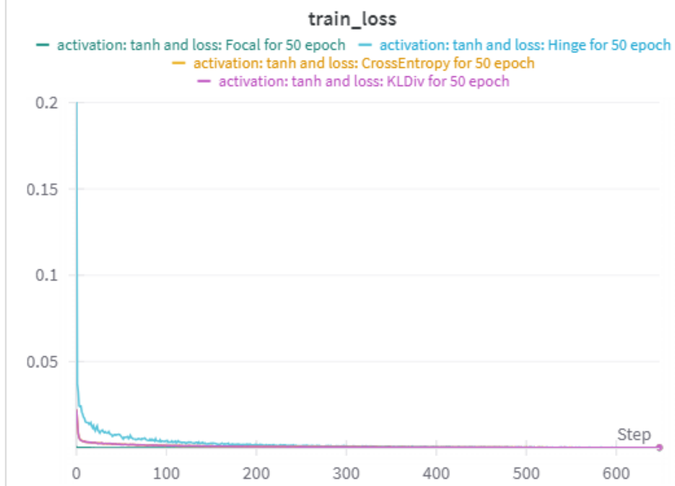
我们将调整 learning rate 为 0.01 后的 FocalLoss 放回一开始的 tanh参数中，并进行50个 epoch 的训练进行对比。

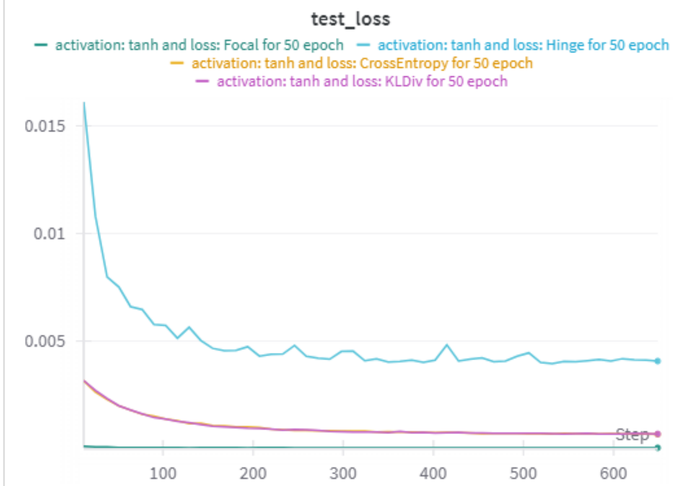












Run set 4

在激活函数给定为 tanh 的情况下，最终的对比结果为：

test acc：Focal（98.02%） > Hinge（97.93%） > CrossEntropy（97.86%） > KLDiv （97.85%）

由于样本量较少，且 learning rate 设置较高，一开始的曲线收敛稍显不稳定，但后续必然是表现较好的一方。

数值与运行稳定性

实验过程中会观察到 Loss 处 Devided by Zero、Inf 的出现，他们会导致模型参数的整个训练直接崩溃，因此我在编写类的过程中采取了一些优化的方式。

Inf

溢出的情况会出现在 Softmax 得出的概率值过大时，因此我依据这篇文章采取了 Normalize 的方式：

```
def softmax(input, axis=1):
    # max_predict = (batch, 10) -> (batch, 1)
    # get max value
    max_predict = np.max(input, axis=axis, keepdims=True)

    # (batch, 10) exp
    input_exp = np.exp(input - max_predict)

    # (batch, 1) sum
    input_exp_sum = np.sum(input_exp, axis=axis, keepdims=True)

    # (batch, 10 = [...h_k])
    return input_exp / input_exp_sum
```

溢出的情况也会出现 KLDiv 算 log(0) 下，因此我做了如下的调整：

```
ln_t_sub_h = np.where(target == 0, 1e-9, np.log(target / input_softmax))
```

Devided by Zero

除数为 0 的情况出现在 KLDiv 中得出 Softmax 概率值过小时，因此我采用了 clip：

```
input_softmax = np.clip(input_softmax, 1e-9, 1.0) # 防止过小
```

其他优化点

- 能直接使用 numpy 的东西尽量直接使用：

```
# tanh forward
ret = np.tanh(input)
return ret.copy()

# Hardswish forward
conditions = [input <= -3, input >= 3, np.logical_and(input > -3, input < 3)]
choices = [0, input, input * (input + 3) / 6]
ret = np.select(conditions, choices)
```

Add a comment