ANN实验报告1

张天祺 2021010719 计12

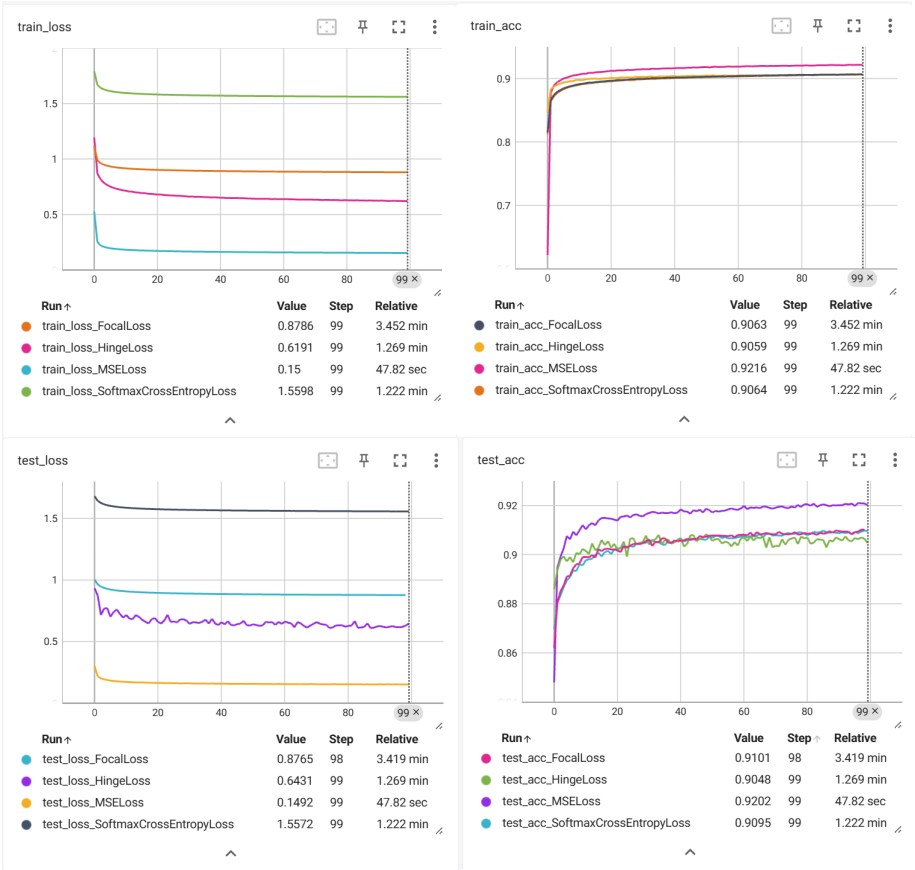
零隐藏层实验

统一使用一层线性层和`sigmoid`激活函数进行训练，使用默认的训练超参数如下，选择4种loss进行实验：

```
default_config = {
    'learning_rate': 0.01,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 100,
    'disp_freq': 50,
    'test_epoch': 1
}
```

由于不同网络架构最适宜的超参数不同，因此这个超参数可能并不能让所有网络都可以得到良好的训练，甚至在一些架构中无法运行。训练pipeline代码在`run_exp_pipeline.py`中的`zero_hidden_layer_test`函数中。

训练得到的结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**：



可以看出，只使用一层线性层的效果并不理想，均没有达到95%的准确率。

单隐藏层实验

添加一层隐藏层，隐藏层的神经元数量设置为100，对5种激活函数和4种损失函数分别进行训练，观察实验结果。

其中`HingeLoss`的margin设置为0.03，`FocalLoss`的alpha设置为0.9，gamma为2.0

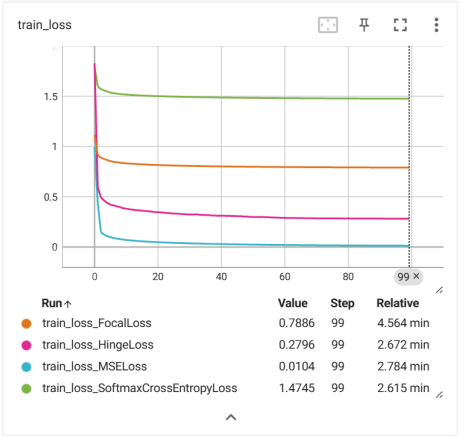
不同损失函数的对比

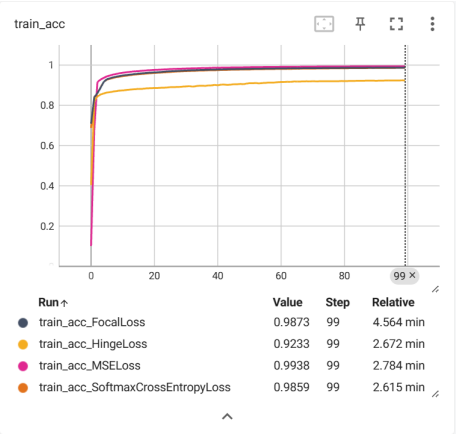
实验使用的统一config如下：

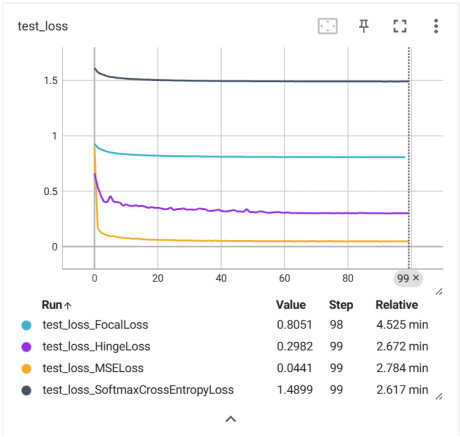
```
default_config = {
    'learning_rate': 0.1,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 100,
    'disp_freq': 50,
    'test_epoch': 1
}
```

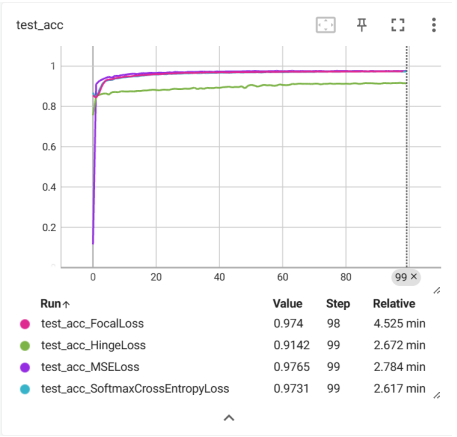
固定激活函数为`sigmoid`，遍历4种损失函数进行训练，得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

训练的pipeline代码位于`run_exp_pipeline.py`中的`one_hidden_layer_loss_test`函数中。



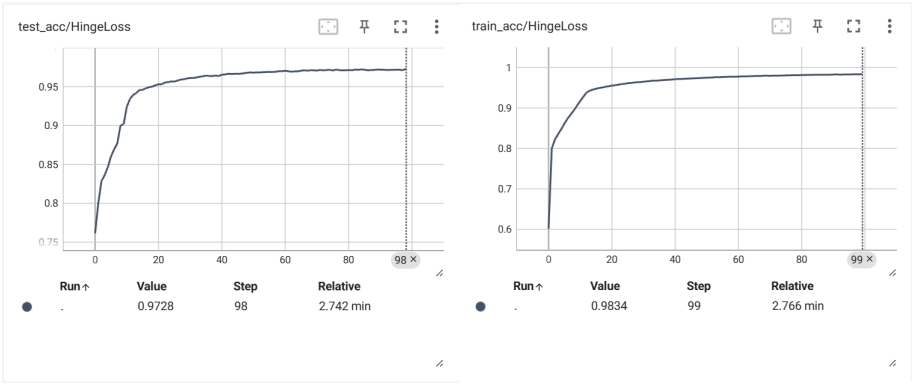


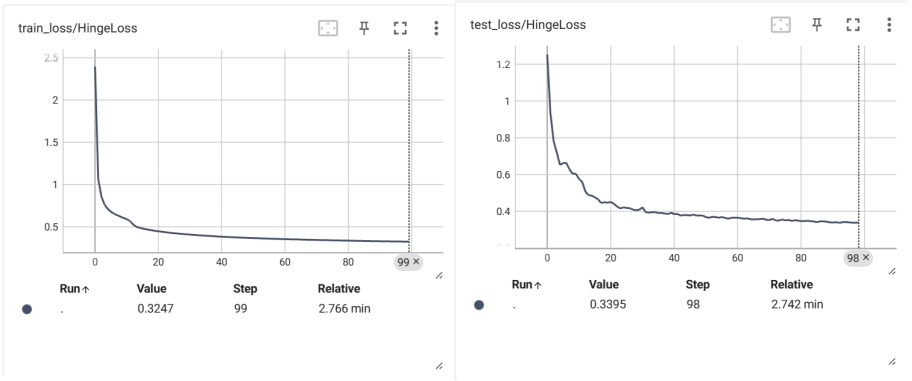




可以发现除了`HingeLoss`效果略差，其余实验的准确率均达到97%以上，训练效果明显。

`HingeLoss`效果不佳的原因是超参数设置不当。修改`learning_rate = 0.01`重新训练`HingeLoss`，准确率也可以轻松达到95%以上，见下图：





可以观察到以下结论：

- 从准确率上看，4种loss的测试集准确率都很高，都达到了97%以上，而训练集的准确率也都达到了98%以上，在训练了100个epoch后并无明显差异，相对而言`HingeLoss`效果较好一点点。但是由于超参数设置的问题，这并不能体现出不同loss的优劣。
- 从收敛速度上看`HingeLoss`的收敛速度最快，不过优势并不显著。
- 从训练时间上看，`FocalLoss`的训练速度显著慢于其他三种loss，其余三种loss的训练速度相差不大。

对此我的看法如下：

- 该分类任务相对比较简单，因此使用一层隐藏层配合这四种损失函数在训练100个epoch之后都能达到很好的拟合效果，所以总体而言训练效果相差不大。
- 就这几种损失函数而言，`FocalLoss`的计算复杂度显著高于剩余几种，这也是使用`FocalLoss`计算速度相对较慢的主要原因。
- `HingeLoss`的计算相对而言更加关注同一样本内不同分类的logits的相对差别，有点类似于对比学习，这样可以加速模型的收敛速度。

不同激活函数的对比

对于隐藏层的激活函数，使用`Sigmoid`，`Selu`，`Swish`，`Gelu`，`Relu`五种激活函数进行实验

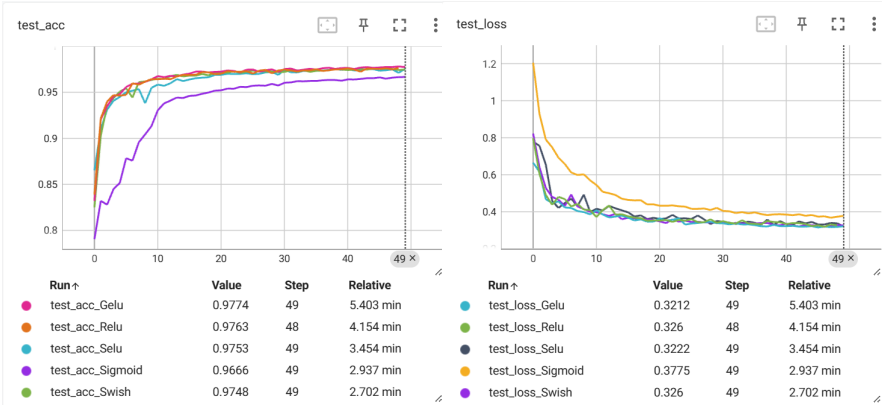
实验使用的统一config如下：

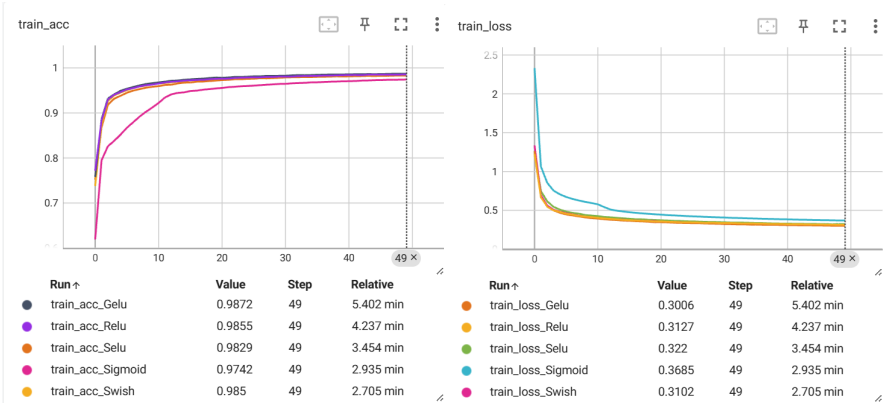
```
default_config = {
    'learning_rate': 0.01,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 50,
    'disp_freq': 50,
    'test_epoch': 1
}
```

由于之前的实验发现100个epoch远大于模型收敛所需要的训练数量且epoch数量过多导致图像特征不够明显，因此缩减`max_epoch`到50

固定损失函数为`HingeLoss`，得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

训练的pipeline代码位于`run_exp_pipeline.py`中的`one_hidden_layer_activate_test`函数中。





可以观察到以下结论：

- 从准确率的角度看，除了`sigmoid`以外，其他几种激活函数的表现没有明显差距，整体上来说`Gelu`的效果最好
- 从收敛速度来看，`Sigmoid`的收敛速度最慢，与其他几种激活函数存在较为明显的差距，而`Gelu`的收敛速度最快。
- 从训练时间来看，`Swish`训练时间最短，而`Gelu`训练所需时间最长

对此我的看法如下：

- `Sigmoid`是全段光滑连续可导的函数，而`Relu`则不会出现梯度消失，因此吸收了`Sigmoid`和`Relu`共同特点的`Gelu`则结合了两者的优点，保留了`Relu`对于梯度消失的免疫性，有着较快的收敛速度。
- 对于`Sigmoid`而言，整体效果较差的原因可能是梯度陡峭变化导致部分神经元梯度消失，因此在损失值较大时，很难做到快速收敛。

- 就训练速度而言，`Gelu`由于表达式较为复杂，需要最多的时间进行计算。

双隐藏层实验

增加一层隐藏层，设置两个隐藏层的神经元分别为256和100，分别进行损失函数和激活函数的实验。

其中`HingeLoss`的margin设置为0.03，`FocalLoss`的alpha设置为0.9，gamma为2.0

不同损失函数的对比

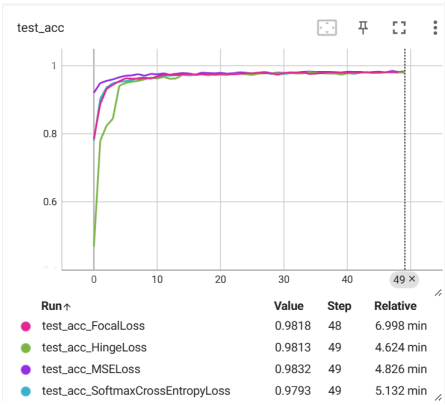
实验使用的统一config如下：

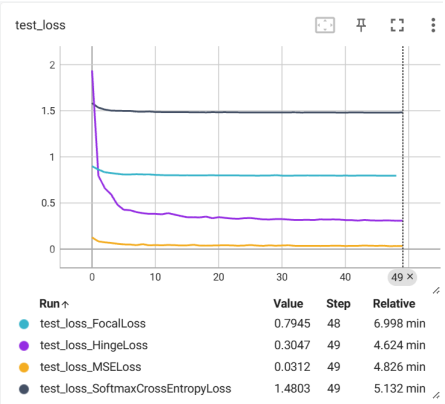
```
default_config = {
    'learning_rate': 0.1,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 50,
    'disp_freq': 50,
    'test_epoch': 1
}
```

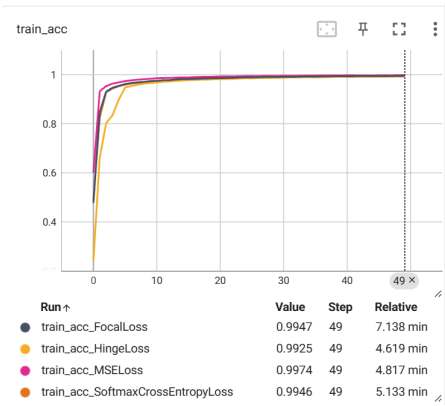
固定激活函数为`Swish`、`Swish`和`Sigmoid`，得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

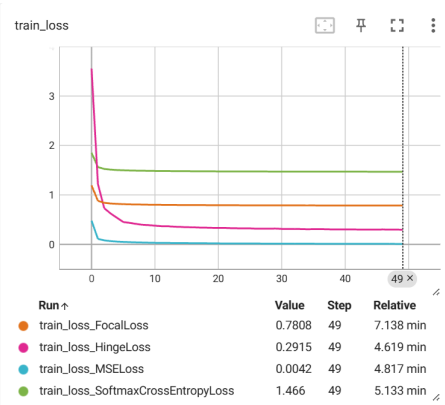
训练的pipeline代码位于`run_exp_pipeline.py`中的`two_hidden_layer_loss_test`函数中。

同时`HingeLoss`实验组学习率依然设置为0.01以完成训练效果（在学习率为0.1时仍然无法有效收敛）。









可以看出较单隐藏层表现有少许提升，但是训练速度显著下降。

原因可能是本次实验的数据较为简单，使用单层隐藏层已经基本足以完成训练任务

不同激活函数的对比

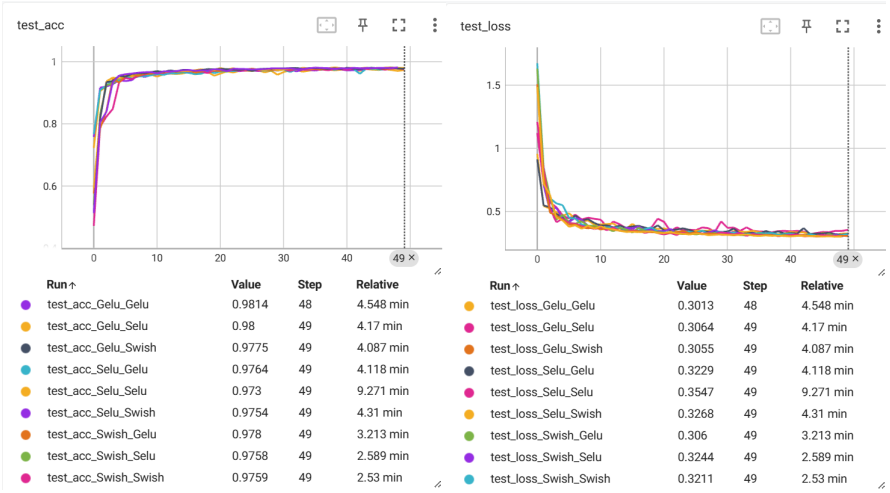
实验使用的统一config如下：

```
default_config = {
    'learning_rate': 0.01,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 50,
    'disp_freq': 50,
    'test_epoch': 1
}
```

固定损失函数为`HingeLoss`，在两层隐藏层中分别使用`Selu`，`Swish`，`Gelu`三种激活函数，两两组合进行九组实验。

得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

训练的pipeline代码位于`run_exp_pipeline.py`中的`two_hidden_layer_activate_test`函数中。



可以看出较单隐藏层表现有极少许提升，但是训练速度显著下降。

原因可能是本次实验的数据较为简单，使用单层隐藏层已经基本足以完成训练任务

Bonus

`FocalLoss`的实现以及分析

`FocalLoss`已经在`loss.py`中按照要求实现。已在上述实验中验证其正确性。

`FocalLoss`针对样本不平衡的问题提出了解决方法，$\gamma > 0$作为可调节因子，对比交叉熵函数，`FocalLoss`多的一个调节因子$(1 - P_t)^\gamma$，对于分类不准确的样本，P_t近似于0，因此调节因子趋近1，因而对于损失没有明显改变，而对于分类准确的因子，这个调节因子会使得损失明显减小，整体而言，相当于增加了分类不准确样本在损失函数中的权重。

对于`FocalLoss`与交叉熵损失的表现对比表现，进入如下实验：

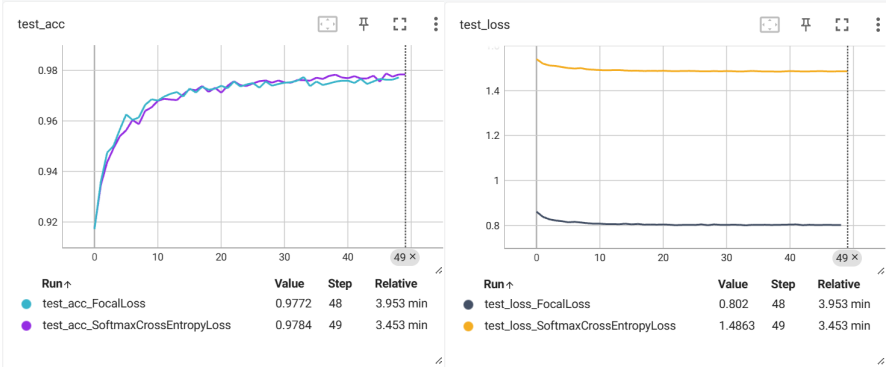
使用一层隐藏层，隐藏神经元为100，使用`Gelu`作为激活函数，对比两种loss的表现

训练的config设置如下：

```
default_config = {
    'learning_rate': 0.1,
    'weight_decay': 0,
    'momentum': 0.9,
    'batch_size': 64,
    'max_epoch': 50,
    'disp_freq': 50,
    'test_epoch': 1
}
```

得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

训练的pipeline代码位于`run_exp_pipeline.py`中的`focal_test`函数中。



Run	Value	Step	Relative
test_acc_FocalLoss	0.9772	48	3.953 min
test_acc_SoftmaxCrossEntropyLoss	0.9784	49	3.453 min
test_loss_FocalLoss	0.802	48	3.953 min
test_loss_SoftmaxCrossEntropyLoss	1.4863	49	3.453 min
train_acc_FocalLoss	0.9897	49	4.049 min
train_acc_SoftmaxCrossEntropyLoss	0.9888	49	3.454 min
train_loss_FocalLoss	0.7837	49	4.049 min
train_loss_SoftmaxCrossEntropyLoss	1.4692	49	3.454 min

得到结论为在这个实验中两种loss的表现没有明显差别，`FocalLoss`的优越性需要在有更复杂实验数据的实验中才能体现。

过拟合的问题

过拟合是指损失函数小但是泛化性能差的问题，具体表现就是在训练集上损失函数很小但是在测试集上损失函数却很大。

对于本实验而言，可以通过调节config中`weight_decay`项来防止过拟合，具体实验如下：

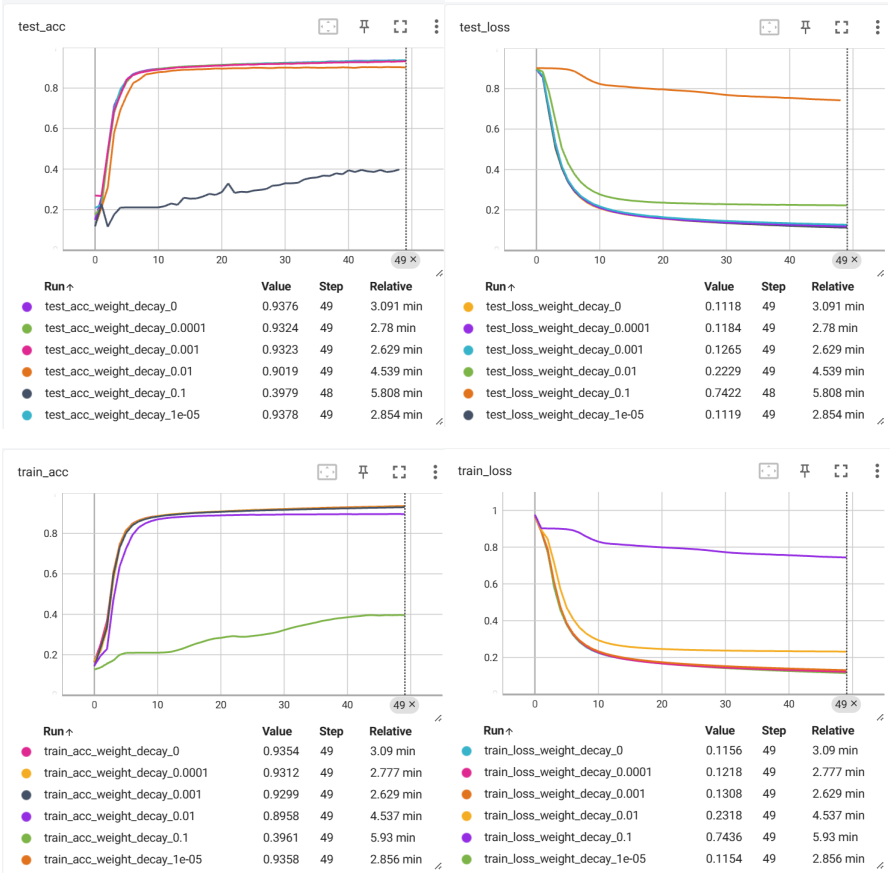
使用一层隐藏层，隐藏神经元为100，使用`Sigmoid`作为激活函数，`MSELoss`作为损失函数，调节`weight_decay`值进行实验：

训练设置的config如下：

```
default_config = {
    'learning_rate': 0.01,
    'momentum': 0,
    'batch_size': 64,
    'max_epoch': 50,
    'disp_freq': 50,
    'test_epoch': 1
}
```

得到结果如下，图中的value值即为train loss, train acc, test loss, test acc的**final value**

训练的pipeline代码位于`run_exp_pipeline.py`中的`overfit_test`函数中。



Run	Value	Step	Relative
test_acc_weight_decay_0	0.9376	49	3.091 min
test_acc_weight_decay_0.0001	0.9324	49	2.78 min
test_acc_weight_decay_0.001	0.9323	49	2.629 min
test_acc_weight_decay_0.01	0.9019	49	4.539 min
test_acc_weight_decay_0.1	0.3979	48	5.808 min
test_acc_weight_decay_1e-05	0.9378	49	2.854 min

Run	Value	Step	Relative
test_loss_weight_decay_0	0.1118	49	3.091 min
test_loss_weight_decay_0.0001	0.1184	49	2.78 min
test_loss_weight_decay_0.001	0.1265	49	2.629 min
test_loss_weight_decay_0.01	0.2229	49	4.539 min
test_loss_weight_decay_0.1	0.7422	48	5.808 min
test_loss_weight_decay_1e-05	0.1119	49	2.854 min

Run	Value	Step	Relative
train_acc_weight_decay_0	0.9354	49	3.09 min
train_acc_weight_decay_0.0001	0.9312	49	2.777 min
train_acc_weight_decay_0.001	0.9299	49	2.629 min
train_acc_weight_decay_0.01	0.8958	49	4.537 min
train_acc_weight_decay_0.1	0.3961	49	5.93 min
train_acc_weight_decay_1e-05	0.9358	49	2.856 min

Run	Value	Step	Relative
train_loss_weight_decay_0	0.1156	49	3.09 min
train_loss_weight_decay_0.0001	0.1218	49	2.777 min
train_loss_weight_decay_0.001	0.1308	49	2.629 min
train_loss_weight_decay_0.01	0.2318	49	4.537 min
train_loss_weight_decay_0.1	0.7436	49	5.93 min
train_loss_weight_decay_1e-05	0.1154	49	2.856 min

可以观察到，`weight_decay`较小时对于结果没有明显影响，但是随着`weight_decay`项的增大，模型的训练效果明显降低。

一般来说，`weight_decay`项可以防止模型过拟合，但是由于本实验较为简单，前面的诸多实验也没有出现明显的过拟合现象，因此增大`weight_decay`项反而导致模型失去表达能力，最终表现为训练的效果变差。

稳定性问题

- 激活函数：
  - `sigmoid`稳定性较好，适合选作最后一个线性层的激活函数，但是表现出来的效果一般。
  - `Gelu`的稳定性一般，在最后一个线性层与某些损失函数在某些条件下可能会出现溢出的问题，但是在合适的实验设置下表现最好。
- 损失函数：
  - `MSELoss`作为最简单的损失函数，计算的稳定性表现最好。
  - `HingeLoss`，`FocalLoss`等较高级的损失函数，往往训练效果好，但是稳定性较差，容易在某些学习率的设置中出现溢出或者无法拟合的问题。