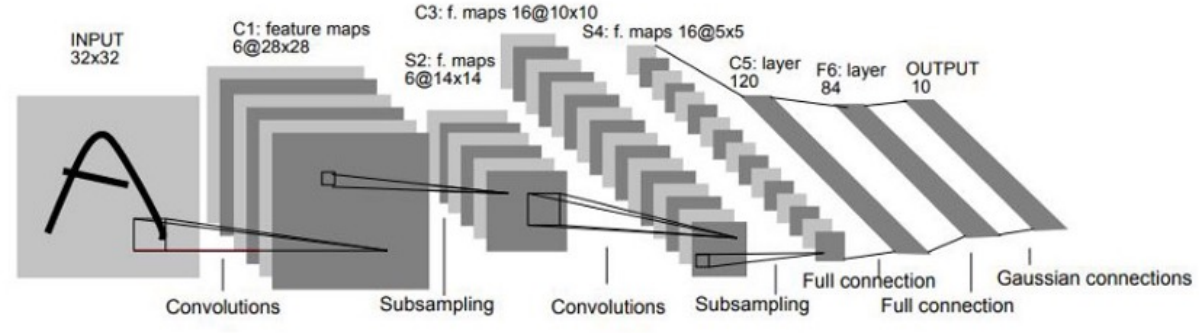
# lab2

## 网络架构

常规的识别手写数字的网络ResNet由两个卷积层和末尾的全连接层构成，如下图所示




我尝试在网络中加入了两个dropout层来减轻网络的过拟合，加强网络的泛化能力

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.softmax(x, dim=1)

network = Net()
```

损失函数使用交叉熵，优化器采用SGD

```
#梯度优化方式采用SGD，带有momentum
optimizer = optim.SGD(network.parameters(), lr=learning_rate, momentum=momentum)
loss_f = nn.CrossEntropyLoss()  #使用交叉熵损失函数
```

# 数据集

采用torch自带的minist数据集合，导入方法如下

```
#设置训练集和测试集
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,
                               transform=torchvision.transforms.ToTensor(),),
    batch_size=batch_size, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=False, download=True,
                               transform=torchvision.transforms.ToTensor(),),
    batch_size=batch_size, shuffle=True)
```

# 训练

采用标准的训练方式，代码如下，测试代码与训练相似

```
#训练模型的函数
def train(epoch):
    loop = tqdm(train_loader, leave=True)
    loss_all = 0
    network.train()  #表示进入了测试模式
    for batch_idx, (data, target) in enumerate(loop):
        optimizer.zero_grad()  #旧的梯度清零
        output = network(data)  #正向传播
        loss = loss_f(output, target)  #计算损失
        loss.backward()    #计算梯度
        optimizer.step()   #梯度下降
        loss_all += loss  #计算dataset的总loss
        loop.set_postfix(loss=loss.item())  #画图相关

        #每隔一段时间记录下当前的loss以便画图
        if batch_idx % log_interval == 0:
            train_losses.append(loss.item())
            train_counter.append((batch_idx * 64) + ((epoch - 1) * len(train_loader.dataset)))

#测试模型的函数
def test(epoch):
    network.eval()   #表明进入测试模式
    test_loss = 0      #测试的总损失
    correct = 0          #测试的总正确数量
    with torch.no_grad():  #在测试的时候需要取消模型的梯度
        for data, target in test_loader:
            output = network(data)
            test_loss += loss_f(output, target)
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()

    test_loss /= (len(test_loader.dataset)/batch_size)
    test_losses.append(test_loss.item())
    print('Epoch: {},Test set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(epoch,
```

```
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

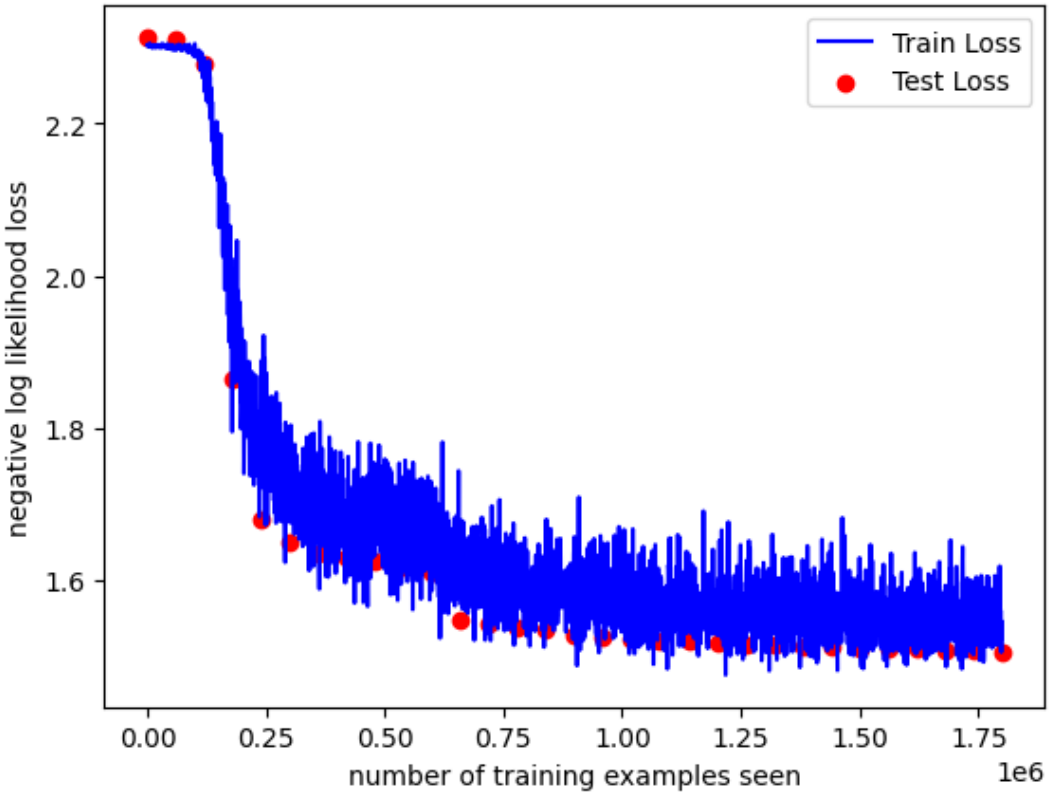
# 实验

实验环境为MacOS Apple Silicon M1 Pro，Python版本3.9，Pytorch版本2.0.0(MacOS版)

相关实验`setting`如下

```
#相关参数
n_epochs = 30   #训练的轮次
batch_size = 64   #batch_size
learning_rate = 0.01   #学习率
momentum = 0.5     #momentum梯度下降时的超参数
log_interval = 10  #画图的记录点
```

根据代码中的绘图函数，我画出了梯度下降过程中的损失和准确率变化。




实验部分采用tqdm做了一个简单的进度条，刚开始的时候模型准确率为10%，跟乱猜没啥区别

大约在23个epoch左右的时候模型收敛，准确率为96%

```
100%|██████████| 938/938 [00:15<00:00, 62.19it/s, loss=1.61]
Epoch: 17,Test set: Avg. loss: 1.5223, Accuracy: 9472/10000 (95%)

100%|██████████| 938/938 [00:15<00:00, 58.92it/s, loss=1.64]
Epoch: 18,Test set: Avg. loss: 1.5187, Accuracy: 9498/10000 (95%)

100%|██████████| 938/938 [00:15<00:00, 62.44it/s, loss=1.54]
Epoch: 19,Test set: Avg. loss: 1.5185, Accuracy: 9510/10000 (95%)

100%|██████████| 938/938 [00:14<00:00, 63.22it/s, loss=1.56]
Epoch: 20,Test set: Avg. loss: 1.5160, Accuracy: 9524/10000 (95%)

100%|██████████| 938/938 [00:14<00:00, 63.39it/s, loss=1.5]
Epoch: 21,Test set: Avg. loss: 1.5142, Accuracy: 9559/10000 (96%)

100%|██████████| 938/938 [00:14<00:00, 63.26it/s, loss=1.55]
Epoch: 22,Test set: Avg. loss: 1.5139, Accuracy: 9555/10000 (96%)

100%|██████████| 938/938 [00:15<00:00, 60.45it/s, loss=1.55]
Epoch: 23,Test set: Avg. loss: 1.5129, Accuracy: 9555/10000 (96%)

100%|██████████| 938/938 [00:15<00:00, 60.51it/s, loss=1.63]
Epoch: 24,Test set: Avg. loss: 1.5112, Accuracy: 9572/10000 (96%)
```

# 探究dropout的作用

在机器学习的模型中，如果模型的参数太多，而训练样本又太少，训练出来的模型很容易产生过拟合的现象。在训练神经网络的时候经常会遇到过拟合的问题，过拟合具体表现在：模型在训练数据上损失函数较小，预测准确率较高；但是在测试数据上损失函数比较大，预测准确率较低。

过拟合是很多机器学习的通病。如果模型过拟合，那么得到的模型几乎不能用。为了解决过拟合问题，一般会采用模型集成的方法，即训练多个模型进行组合。此时，训练模型费时就成为一个很大的问题，不仅训练多个模型费时，测试多个模型也是很费时。

Dropout可以比较有效的缓解过拟合的发生，在一定程度上达到正则化的效果。具体如下

1. **取平均的作用：** 先回到标准的模型即没有dropout，我们用相同的训练数据去训练5个不同的神经网络，一般会得到5个不同的结果，此时我们可以采用 “5个结果取均值”或者“多数取胜的投票策略”去决定最终结果。例如3个网络判断结果为数字9,那么很有可能真正的结果就是数字9，其它两个网络给出了错误结果。这种“综合起来取平均”的策略通常可以有效防止过拟合问题。因为不同的网络可能产生不同的过拟合，取平均则有

可能让一些“相反的”拟合互相抵消。dropout掉不同的隐藏神经元就类似在训练不同的网络，随机删掉一半隐藏神经元导致网络结构已经不同，整个dropout过程就相当于对很多个不同的神经网络取平均。而不同的网络产生不同的过拟合，一些互为“反向”的拟合相互抵消就可以达到整体上减少过拟合。

2. **减少神经元之间复杂的共适应关系：** 因为dropout程序导致两个神经元不一定每次都在一个dropout网络中出现。这样权值的更新不再依赖于有固定关系的隐含节点的共同作用，阻止了某些特征仅仅在其它特定特征下才有效果的情况 。迫使网络去学习更加鲁棒的特征 ，这些特征在其它的神经元的随机子集中也存在。换句话说假如我们的神经网络是在做出某种预测，它不应该对一些特定的线索片段太过敏感，即使丢失特定的线索，它也应该可以从众多其它线索中学习一些共同的特征。从这个角度看dropout就有点像L1，L2正则，减少权重使得网络对丢失特定神经元连接的鲁棒性提高。

我设置了一个没有dropout的网络

```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

network = Net2()
```

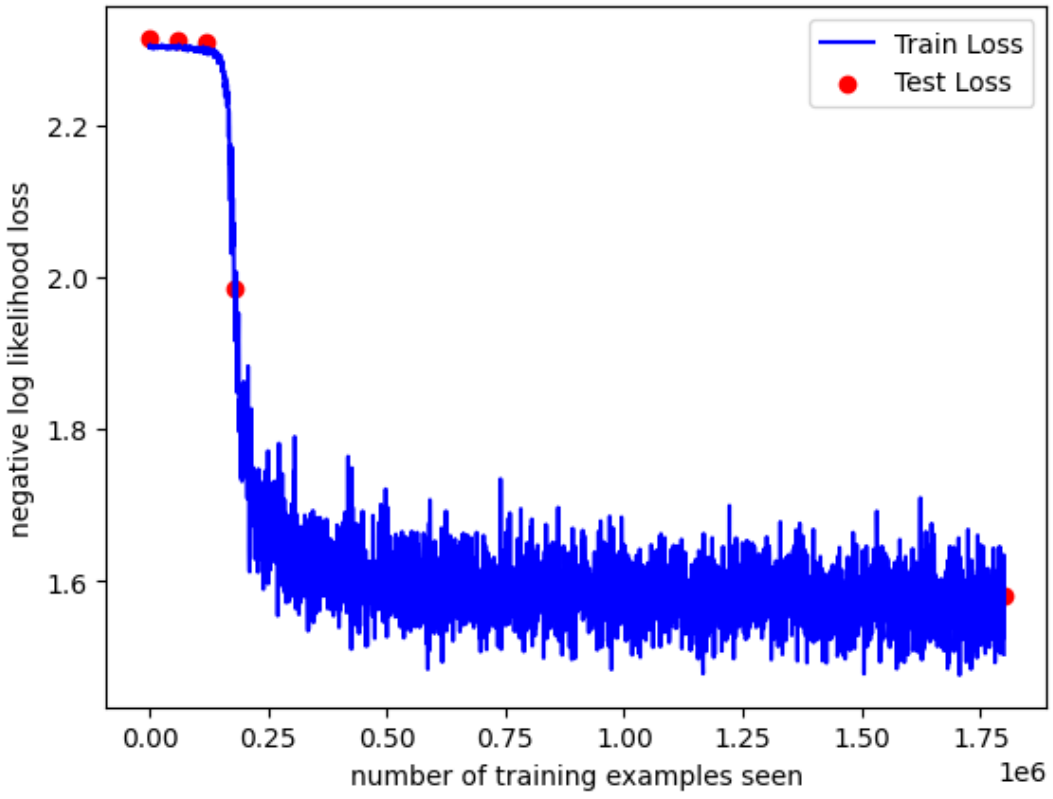
参数与上文一致，训练结果如下

在相同epoch时，（左图为无dropout，右图为有dropout）

```
100%|| 938/938 [00:14<00:00, 66.01it/s, loss=1.66]
Epoch: 17,Test set: Avg. loss: 1.5853, Accuracy: 8838/10000 (88%)

100%|| 938/938 [00:14<00:00, 63.74it/s, loss=1.56]
Epoch: 18,Test set: Avg. loss: 1.5851, Accuracy: 8846/10000 (88%)

100%|| 938/938 [00:14<00:00, 63.11it/s, loss=1.55]
Epoch: 19,Test set: Avg. loss: 1.5837, Accuracy: 8856/10000 (89%)

100%|| 938/938 [00:14<00:00, 63.55it/s, loss=1.6]
Epoch: 20,Test set: Avg. loss: 1.5826, Accuracy: 8862/10000 (89%)

100%|| 938/938 [00:15<00:00, 62.37it/s, loss=1.53]
Epoch: 21,Test set: Avg. loss: 1.5850, Accuracy: 8836/10000 (88%)

100%|| 938/938 [00:15<00:00, 60.94it/s, loss=1.51]
Epoch: 22,Test set: Avg. loss: 1.5820, Accuracy: 8869/10000 (89%)

100%|| 938/938 [00:15<00:00, 62.14it/s, loss=1.56]
Epoch: 23,Test set: Avg. loss: 1.5841, Accuracy: 8839/10000 (88%)

100%|| 938/938 [00:14<00:00, 63.80it/s, loss=1.61]
Epoch: 24,Test set: Avg. loss: 1.5803, Accuracy: 8880/10000 (89%)
```

```
100%|| 938/938 [00:15<00:00, 62.19it/s, loss=1.61]
Epoch: 17,Test set: Avg. loss: 1.5223, Accuracy: 9472/10000 (95%)

100%|| 938/938 [00:15<00:00, 58.92it/s, loss=1.64]
Epoch: 18,Test set: Avg. loss: 1.5187, Accuracy: 9498/10000 (95%)

100%|| 938/938 [00:15<00:00, 62.44it/s, loss=1.54]
Epoch: 19,Test set: Avg. loss: 1.5185, Accuracy: 9510/10000 (95%)

100%|| 938/938 [00:14<00:00, 63.22it/s, loss=1.56]
Epoch: 20,Test set: Avg. loss: 1.5160, Accuracy: 9524/10000 (95%)

100%|| 938/938 [00:14<00:00, 63.39it/s, loss=1.5]
Epoch: 21,Test set: Avg. loss: 1.5142, Accuracy: 9559/10000 (96%)

100%|| 938/938 [00:14<00:00, 63.26it/s, loss=1.55]
Epoch: 22,Test set: Avg. loss: 1.5139, Accuracy: 9555/10000 (96%)

100%|| 938/938 [00:15<00:00, 60.45it/s, loss=1.55]
Epoch: 23,Test set: Avg. loss: 1.5129, Accuracy: 9555/10000 (96%)

100%|| 938/938 [00:15<00:00, 60.51it/s, loss=1.63]
Epoch: 24,Test set: Avg. loss: 1.5112, Accuracy: 9572/10000 (96%)
```

总体loss下降图,最后的准确率为90%




可以看到，训练loss的下降速度和精度都与使用了dropout有较大的区别，在测试集的表现不如加了dropout的版本