1 SNN 模型描述

本次实验 SNN 网络的实现基于 snnTorch.

1.1 神经元

所使用的脉冲神经元是 snnTorch 中的 LIF 神经元，即 Leaky Integrate-and-Fire 神经元。

Leaky 用于模拟离子渗透的过程，由于细胞膜内外的电位是靠不同的离子浓度维持的，细胞膜不断进行膜内外离子的交换，神经元受到的刺激会改变细胞内外离子的分布，而当只有一次输入时，leaky 就能够模拟电荷泄漏，神经元回落到静息状态的过程。Integrate 的概念和实现则类似于传统的 ANN 结构，即每个脉冲神经元都会接受到直接与其相连的其他神经元的脉冲，这也是较为简化而便于实现大规模集成运算的一种方式，同时也贴合人脑的神经元连接方式。Fire 则模拟了神经元的脉冲输出，即当神经元的电位达到阈值时，神经元会发出脉冲，并且电位会回落到静息电位。

1.2 拓扑结构

为了便于比较 CNN 和 SNN 的性能，本次实验所实现的两种网络有着一样的宏观拓扑结构，而差别仅仅在于微观上神经元的差异和神经元之间的连接方式、激活方式等。SNN 和 CNN 均内嵌了两层卷积层，两层池化层，一个全连接层。SNN 不同的地方在于，在每个池化层或者全连接层后，SNN 还内嵌了一个 Leaky 层，以实现上述对 LIF 神经元前馈计算和反向传播的模拟。

```
    SNN(
  (0): Conv2d(1, 12, kernel_size=(5, 5), stride=(1, 1))
  (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (2): Leaky()
  (3): Conv2d(12, 64, kernel_size=(5, 5), stride=(1, 1))
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Leaky()
  (6): Linear(in_features=1024, out_features=10, bias=True)
  (7): Leaky()
)
    ANN(
  (conv1): Conv2d(1, 12, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(12, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc): Linear(in_features=1024, out_features=10, bias=True)
)
```

可以看到，SNN 和 CNN 的各层网络参数是完全相同的，唯一的区别是 SNN 内嵌了 Leaky 层。

1.3 输入输出

由于 SNN 和 CNN 的入口是 channel, kernel_size 完全相同的卷积层，其输入是一致的，两个网络可以复用一个 dataset, 即 torch 内置的 MNIST 数据集。神经网络每次接受形如 `batchsize*channel*height*width`

的输入，其中 batchsize 我设定为 128，而 channel、height、width 分别为 1、28、28，是 MNIST 数据集本身的参数。

SNN 和 CNN 在输出上略有一些差别。CNN 部分我们已经很熟悉，在上面的网络结构中也可以看到，对于 MNIST 这样的分类任务，CNN 的输出接口是 out_features=10 的全连接层，即一个能够表征各个方向上的概率的 10 维向量。而 SNN 的输出接口是 Leaky 层，故其输出是一个形如 num_step*batchsize*feature 的张量，其中 num_step 是模拟的时间步数，我设定为 50，batchsize 仍为 128，feature 为分类的所有可能类别数，也为 10.

1.4 学习方法

两种神经网络使用的优化器均为 torch 内置的 Adam 优化器，参数选择上，我将学习率设定为 0.01，betas 设定为 (0.9, 0.999)，其余采用默认设置。

两种网络的学习方法均为前馈计算和反向传播，其中 CNN 的方法我们已经较为熟悉了，SNN 的方法则集成在了 snnTorch.backprop 的 BPTT 函数中。BPTT 函数实现的是依时间反向传播的方法，具体来说，在损失累积的同时，对每个时间步应用前向传播，而仅在每个时间步序列的末尾应用反向传播和参数更新。我们需要为其传入一个损失函数，我按照教程的建议采取了 snnTorch 库中内置的 ce_rate_loss 函数，即 CrossEntropySpikeRateLoss.

此外，由于 MNIST 数据集本身是时间无关数据集，在调用 BPTT 函数时，需要将 time_var 参数设定为 False，以避免在每个时间步都进行数据集的采样。

2 实验结果与对比

2.1 效率对比

2.1.1 单轮训练时间开销

单纯从每一轮的计算开销上来讲，SNN 是比 CNN 大的。在本次作业我的实验代码环境下，SNN 的每个 epoch 的计算时间为 CNN 的 5 倍左右，SNN 的计算开销可以通过 num_step 调整，由于我设置的 num_step 为 50，是一个相对而言比较高的数值，实际上为了达到本任务的精度，num_step 可以设置得更小，以让 SNN 用更短的时间完成任务。

2.1.2 相同精度时间开销

虽然 SNN 每个 epoch 训练所需要的时间比 CNN 长，但是如果我们希望达到相同的精度，二者需要的训练时间大致相当。在本实验的环境下，SNN 仅需 5 个 epoch 就能达到比较高的精度（95%），而 CNN 则需要约 30 个 epoch。从这个角度来讲，SNN 并没有产生过分“昂贵”的训练开销，甚至训练速度会较 CNN 快一点。

2.2 精度对比

2.2.1 收敛速度和稳定性

由于 SNN 与 CNN 两者定义的 loss 函数不同，因此无法直接比较两者的 loss 值，但是可以从收敛速度和稳定性上来比较两者的效果。在本实验的环境下，正如上文提到的，SNN 的 loss 的收敛速度快于 CNN。在

两者的 loss 各自均达到稳定后，其波动程度并没有表现出明显的差别，均稳定在一定范围内。

2.2.2 准确率

在我使用的 `BPTT` 训练方式下，SNN 能够很快地达到收敛。具体表现为在训练的前 5 个 epoch 中，SNN 的准确率就已经达到了 95% 以上，并且在随后的训练中，准确率没有发生明显的变化，最终准确率会稳定在 95% 左右。而 CNN 在前 20 个 epoch 的准确率都在 90% 以下，而在后续的训练中，CNN 的准确率稳定提升，最终也能达到 95% 左右。这说明，单纯就 MNIST 手写数字识别这一任务而言，两者虽然消耗了不同的计算资源，需要不同的计算时间，但是最后达到的准确率是相当的。

2.3 总结

仅就 MNIST 数据集分类任务而言，SNN 与 CNN 均能达到比较高的精度，且为了达到同等精度所需要的训练时间大致相当。为了探寻 SNN 相比于以 CNN 为代表的传统神经网络的优劣性，应当需要以更适合 SNN 发挥的任务和场景作为实验对象，在手写数字识别这一任务下，SNN 并没有表现出明显的优势。（不过也没有明显的劣势）