实验 2：脉冲神经网络

2024 年 4 月 9 日

实验 2：脉冲神经网络

1 实验简介

本实验主要研究视觉信号的脉冲编码，并通过训练脉冲神经网络实现手写数字识别。

通过本实验，你将学习脉冲神经网络的基本原理及模型搭建，掌握以下技能：

- 以 MNIST 手写数字数据集为例，了解视觉刺激的脉冲编码方法；
- 学习如何使用 Python 及开源工具 PyTorch 和 snnTorch 搭建简单的脉冲神经网络；
- 学习如何进行脉冲神经网络的训练和测试。

2 实验准备

首先，你需要安装本实验所需的工具包。为了加快安装速度，请确保 `pip` 及 `conda` 包管理器已切换到国内源。本实验的依赖包主要包括：

- `PyTorch`: 开源的 Python 机器学习库；
- `snnTorch`: 基于 `PyTorch` 的脉冲神经网络训练框架；
- `ffmpeg`: 音视频的开源库，在本实验中主要用于动画视频编码。

在终端执行以下命令，安装 `ffmpeg`：

```
conda install -c conda-forge ffmpeg
```

[2]:
```
import pkgutil

# 检查 PyTorch 库是否已安装
if pkgutil.find_loader('torch'):
```

```
    print('PyTorch is available.')
else:
    !pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple/

if pkgutil.find_loader('torchvision'):
    print('torchvision is available.')
else:
    !pip install torchvision -i https://pypi.tuna.tsinghua.edu.cn/simple/

# 检查 snnTorch 库是否已安装
if pkgutil.find_loader('snntorch'):
    print('snntorch is available.')
else:
    !pip install snntorch -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

[3]:
```
# 忽略 Warning 输出
import warnings
warnings.filterwarnings("ignore")
```

3 下载和查看数据

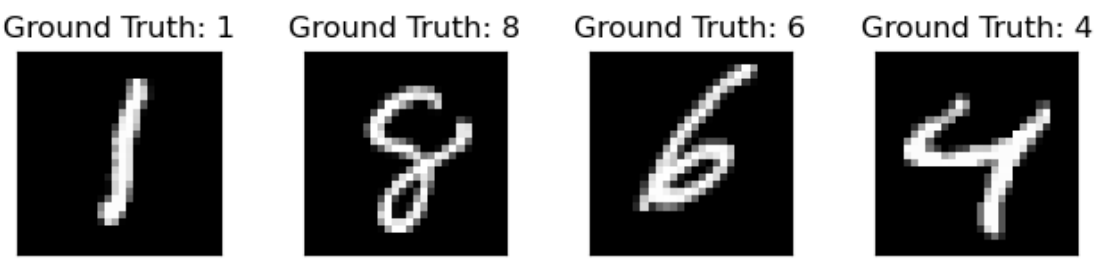
本实验中，我们使用 MNIST（Mixed National Institute of Standards and Technology database）手写数字计算机视觉数据集，该数据集包含 60,000 张用于训练的示例图片和 10,000 张用于测试的示例图片，所有图片已经过尺寸标准化（28×28 像素）且数字已置于图像中心。

首先，我们使用 torchvision 工具包自带的数据集下载功能，获取该数据集，并查看数据样例。

[4]:
```
import os
import torch
import torchvision
import matplotlib.pyplot as plt
from torchvision import datasets
from torchvision import transforms
from torch.utils.data import DataLoader

os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

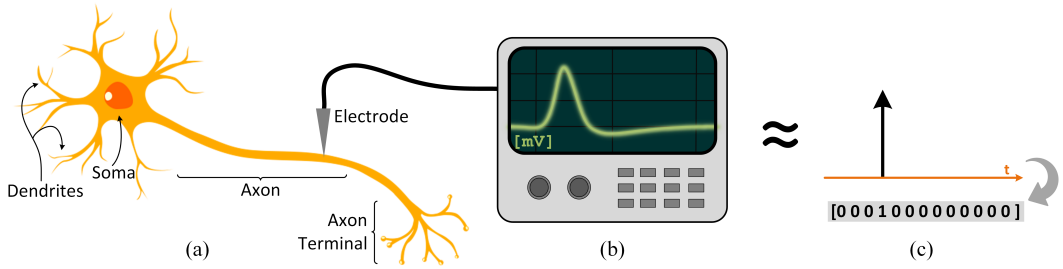
```
# 定义下载数据的目录
data_root = "data"

# 定义 MNIST 测试数据集和要应用的变换
transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize((0.,),(1.,))
])

# 获取数据集，并应用变换
mnist_examples = datasets.MNIST(root=data_root, train=False, download=True,␣
 ↪transform=transform)
```

[5]:
```
# 定义数据加载器
# 由于 shuffle 参数设置为 True，因此每次执行该代码块，都能看到不同的图像示例
mnist_examples_loader = DataLoader(mnist_examples, batch_size=4, shuffle=True)
examples = enumerate(mnist_examples_loader)
batch_idx, (imgs, labels) = next(examples)  # 读取数据，batch_idx 从 0 开始

# 查看数据
fig = plt.figure()
for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.tight_layout()
    plt.imshow(imgs[i][0], cmap='gray', interpolation='none')
    plt.title("Ground Truth: {}".format(labels[i]))
    plt.xticks([])
    plt.yticks([])
```



4 脉冲编码

我们已经了解到，大脑是通过处理电流脉冲的形式来加工信息的。既然我们的目标是模拟大脑构建脉冲神经网络（Spiking neural networks, SNN），那么使用脉冲（Spikes）作为神经网络的输入是直观且合理的。但是，在本实验中我们使用的 MNIST 数据并不是脉冲数据，我们需要首先通过脉冲编码（Spike encoding）的方式，将图像转换成脉冲的形式。



三种常见的脉冲编码方式如下：

- 频率编码（Rate coding）：使用输入特征来确定脉冲的频率；
- 延迟编码（Latency coding）：使用输入特性来确定脉冲发生的时间；
- 增量调制（Delta modulation）：使用输入特征的时间变化来产生脉冲信号。

本实验中，我们重点研究前两种编码方式。

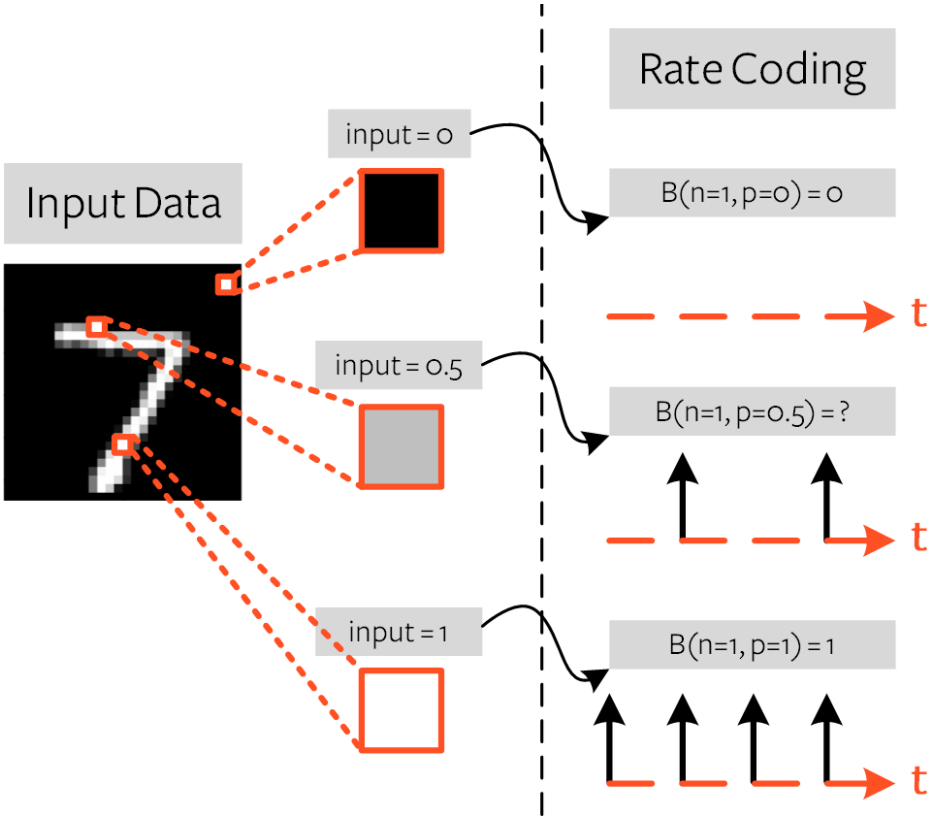
4.1 频率编码（Rate coding）

4.1.1 原理

我们可以将**归一化**的特征 X_{ij} 作为一个事件（或脉冲）在某时间步产生的概率，并返回一个频率编码值 R_{ij}。以上过程可视为一个泊松实验：$R_{ij} \sim B(n,p)$，当 $n=1$ 时，脉冲产生的概率为 $p=X_{ij}$。脉冲产生的概率可形式化地表示为：

$$P(R_{ij}=1)=X_{ij}=1-P(R_{ij}=0).$$

对于 MNIST 图像，脉冲产生的概率与像素的颜色值有关。纯白色的像素，脉冲产生的概率为 100%；纯黑色的像素，脉冲产生的概率为 0；对于灰色的像素，如果灰度值为 0.5，那么脉冲产生的概率也为 0.5。



4.1.2 执行频率编码

snnTorch 提供了 `spikegen.rate` 函数，可方便地进行频率编码。通过制定 `num_steps` 参数可指定时间轴的长度。

[6]:
```
# 指定时间轴长度，即编码的次数
num_steps = 200
```

[7]:
```
from snntorch import spikegen

# 在 minibatch 上迭代
data = iter(mnist_examples_loader)
data_it, targets_it = next(data)

# 获得编码数据
```

```
spike_data = spikegen.rate(data_it, num_steps=num_steps)
```

[8]:
```
# 查看数据的维度
print(spike_data.size())
```

```
torch.Size([200, 4, 1, 28, 28])
```

编码数据是按 $num\_steps \times batc\_size \times input\_dimensions$ 组织的。

4.1.3 可视化

我们使用**动画**和**栅格图**两种方式对编码数据进行可视化。

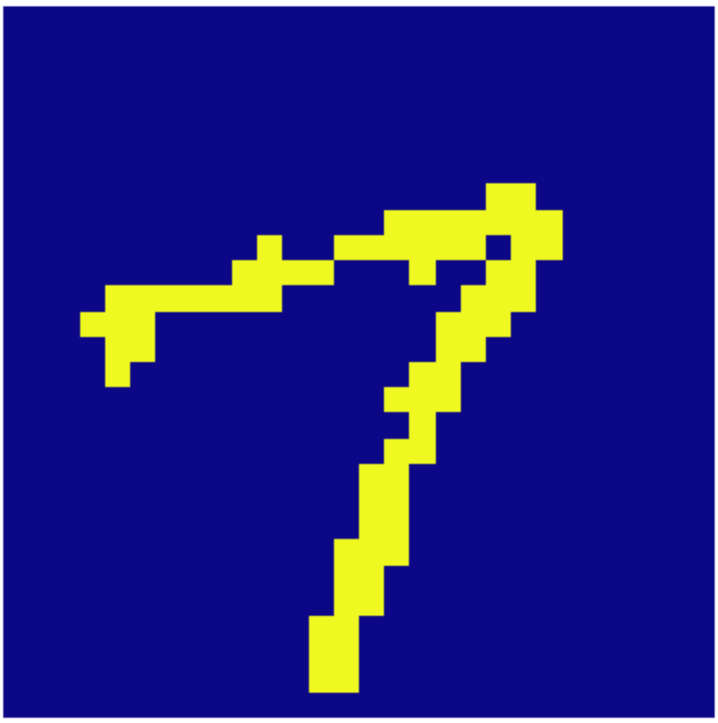
[9]:
```
import ipywidgets as widgets
import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
from IPython.display import HTML

data = iter(mnist_examples_loader)
data_it, targets_it = next(data)
```

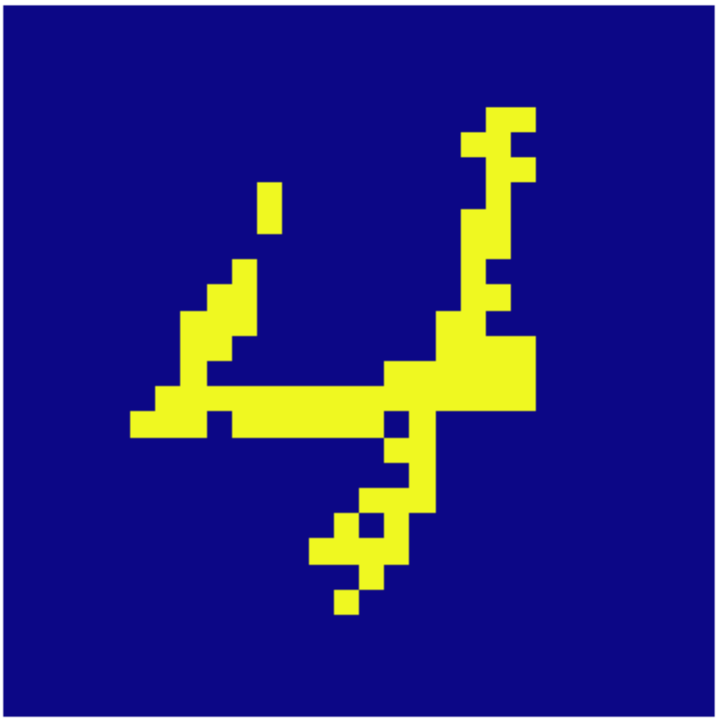
[10]:
```
target_label = {
    0: "Zero",
    1: "One",
    2: "Two",
    3: "Three",
    4: "Four",
    5: "Five",
    6: "Six",
    7: "Seven",
    8: "Eight",
    9: "Nine"
}
```

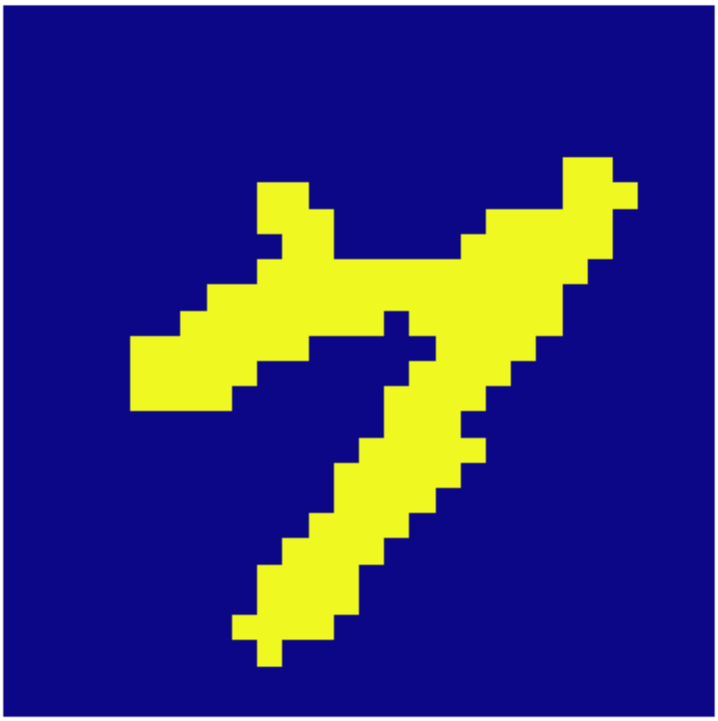
[11]:
```
# 使用 Rate coding 方法，对前 4 张图片进行编码
for i in range(4):
    image = data_it[i]
    print(target_label[targets_it[i].item()])
```

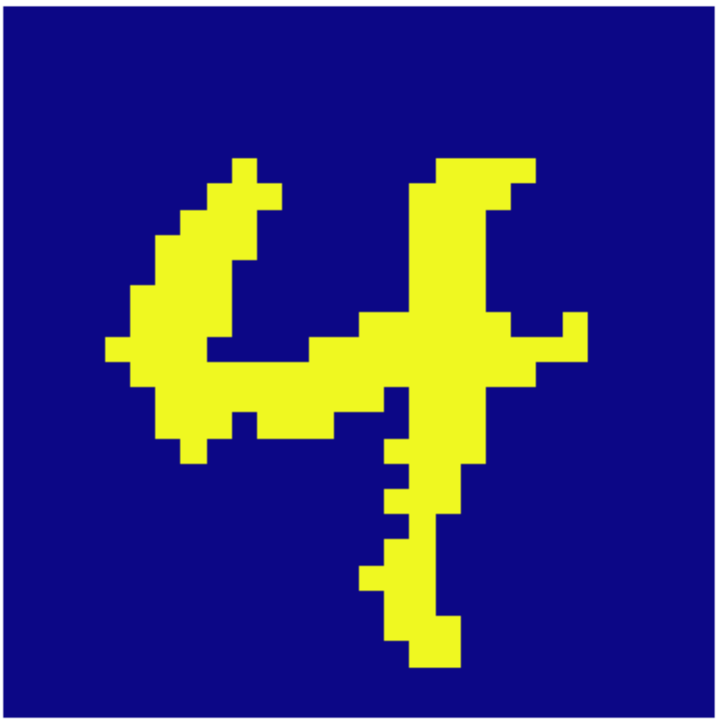
```
    spike_data = spikegen.rate(image, num_steps=200)

    fig, ax = plt.subplots()
    anim = splt.animator(spike_data[:, 0], fig, ax)
    anim.save(
        "videos/rate-coding-video-MNIST-{}.gif".format(
            target_label[targets_it[i].item()].lower().replace(' ', '-')
        )
    )
    display(HTML(anim.to_html5_video()))
```

```
Seven

<IPython.core.display.HTML object>

Four

<IPython.core.display.HTML object>

Seven

<IPython.core.display.HTML object>

Four

<IPython.core.display.HTML object>
```





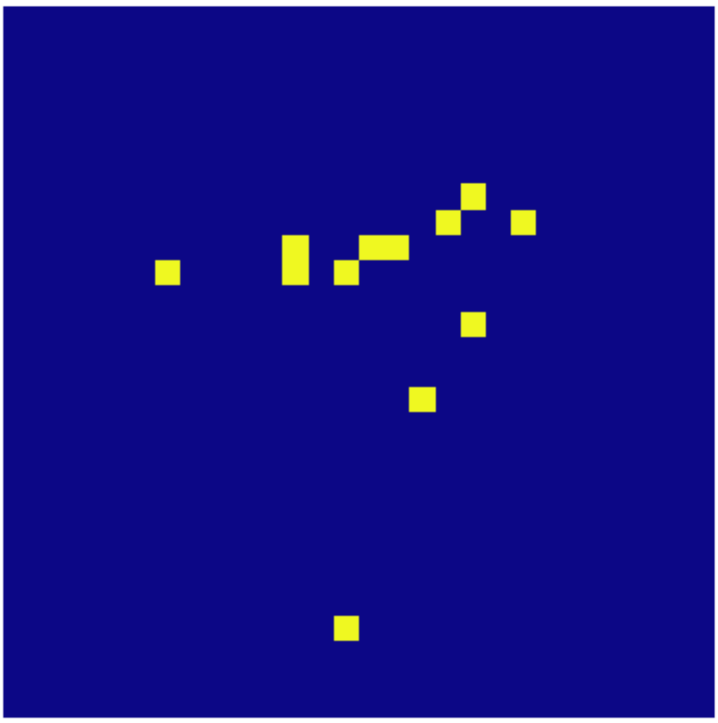


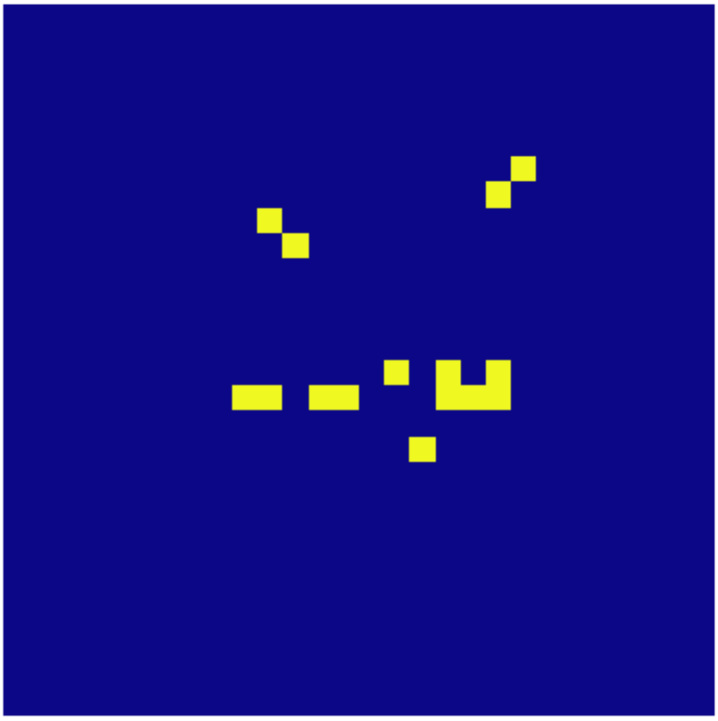


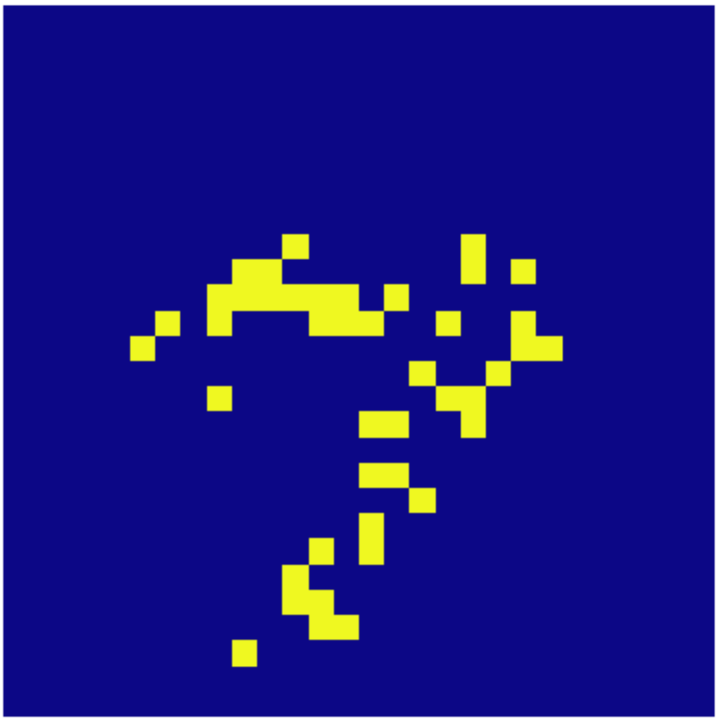
也可以通过指定 gain 参数降低脉冲的频率，例如，以下代码块使用 25% 的脉冲频率进行编码。

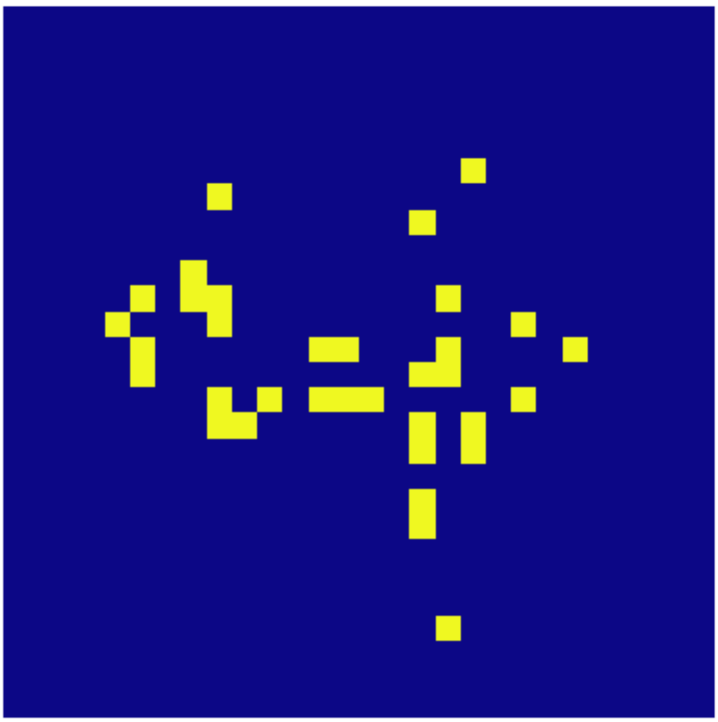
[12]:
```
for i in range(4):
    image = data_it[i]
    print(target_label[targets_it[i].item()])

    spike_data_low_gain = spikegen.rate(image, num_steps=200, gain=0.25)  # 指定
gain 参数

    fig, ax = plt.subplots()
    anim = splt.animator(spike_data_low_gain[:, 0], fig, ax)
    anim.save(
        "videos/rate-coding-video-low-gain-MNIST-{}.gif".format(
            target_label[targets_it[i].item()].lower().replace(' ', '-')
        )
    )
    display(HTML(anim.to_html5_video()))
```

```
Seven

<IPython.core.display.HTML object>

Four

<IPython.core.display.HTML object>

Seven

<IPython.core.display.HTML object>

Four

<IPython.core.display.HTML object>
```

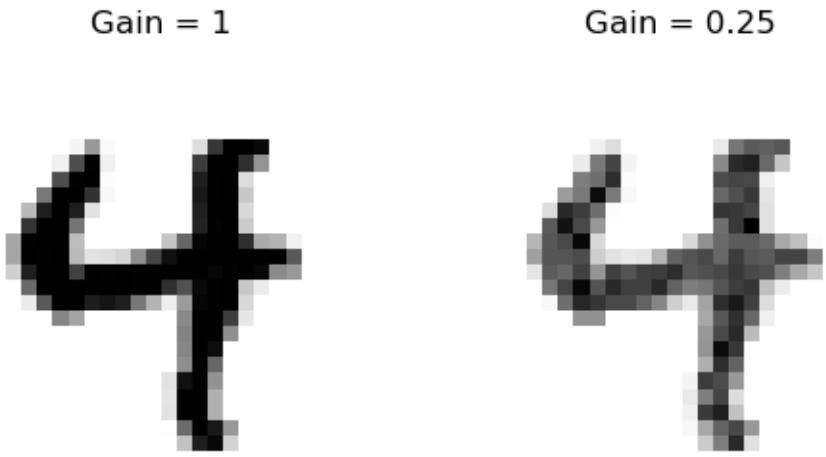








[13]:
```
spike_data = spikegen.rate(image, num_steps=200)
spike_data_low_gain = spikegen.rate(image, num_steps=200, gain=0.25)
spike_data_sample = spike_data[:, 0]
spike_data_sample2 = spike_data_low_gain[:, 0]

plt.figure(facecolor="w")
plt.subplot(1,2,1)
plt.imshow(spike_data_sample.mean(axis=0).reshape((28,-1)).cpu(), cmap='binary')
plt.axis('off')
plt.title('Gain = 1')

plt.subplot(1,2,2)
plt.imshow(spike_data_sample2.mean(axis=0).reshape((28,-1)).cpu(),␣
 ↪cmap='binary')
plt.axis('off')
plt.title('Gain = 0.25')

plt.show()
```

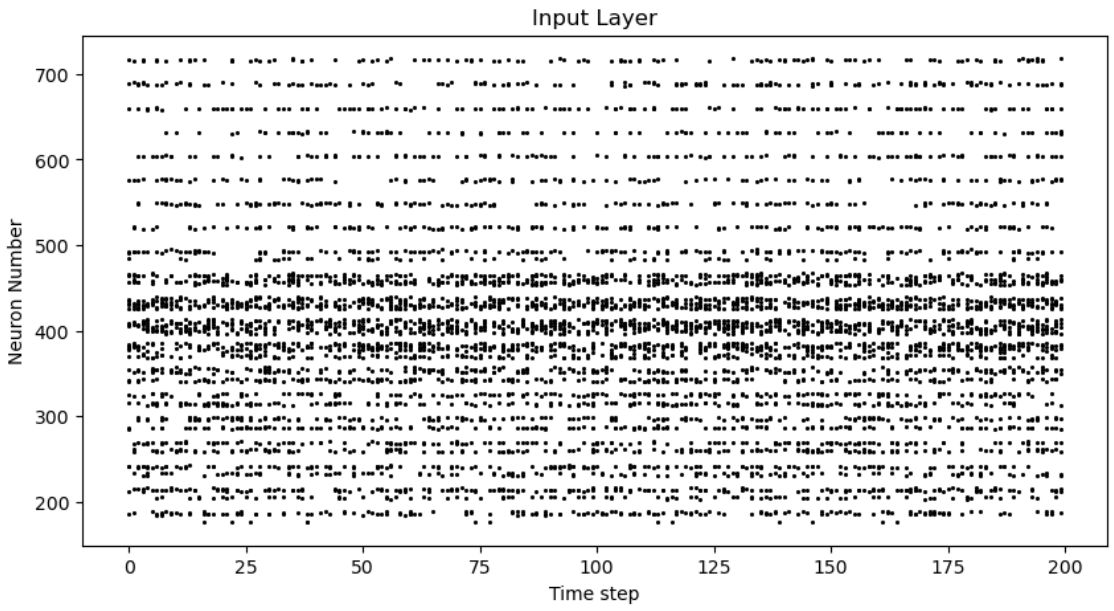


在上面的代码块中，我们重建了不同 `gain` 参数的图像。容易发现，当 `gain=0.25` 时，由于使用

了较低的脉冲频率，图像较 gain=1 时淡了很多。

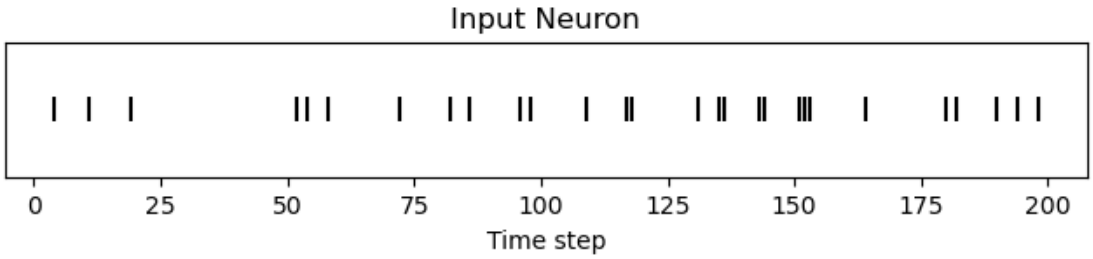
同时，我们也可以采用栅格图的方式可视化编码数据。

[14]:
```
# 将样本数据的维度调整为 2D 张量
spike_data_sample2 = spike_data_sample2.reshape((num_steps, -1))

# 绘制栅格图
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data_sample2, ax, s=1.5, c="black")

plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```



[15]:
```
# 此外，我们也可展示某个单独的神经元的栅格图

idx = 400  # 某些神经元可能一直没有被激活，可适当调整该序号
```

```
fig = plt.figure(facecolor="w", figsize=(8, 1))
ax = fig.add_subplot(111)

splt.raster(spike_data_sample2.reshape(num_steps, -1)[:, idx].unsqueeze(1), ax,␣
↪s=100, c="black", marker="|")

plt.title("Input Neuron")
plt.xlabel("Time step")
plt.yticks([])
plt.show()
```

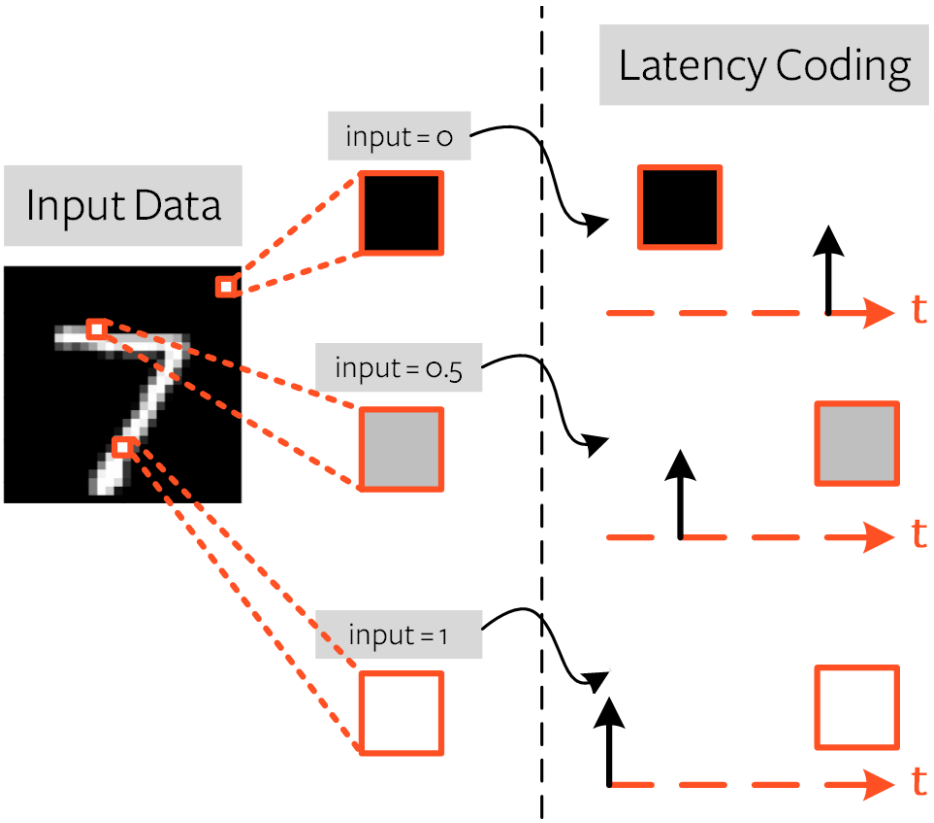


4.2 延迟编码（Latency coding）

4.2.1 原理

采用 Rate coding 的方式是不经济的、耗时的，因为从上面的例子可以看出，表征一个简单的信息就需要大量的脉冲，而每产生一次脉冲都需要消耗能量；同时，人脑中的神经元放电频率无法做到太快，如果所有信息均采用 Rate coding，人类将难以做到对外界刺激的快速反应。因此，有学者提出了延迟编码（Latency coding）的编码方式。

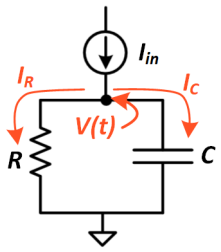
假如我们能够捕获神经元的精确放电时间信息，那么可以采取单个脉冲在不同时刻激发的方式表达信息。虽然这样做增加了对噪声的敏感性，但通过该方式能够大大降低能量消耗。延迟编码的基本原理是：在整个时间周期内最多允许神经元激活 1 次，较大的特征输入会更早触发，较小的特征输入会更晚触发。以 MNIST 图像为例，较亮的像素会更早触发，而较暗的像素会更晚触发。



4.2.2 执行延迟编码

与频率编码类似，我们可以使用 `spikegen.latency` 函数进行延迟编码。

延迟编码可以用 RC 电路模型进行描述。在如下图所示的 RC 电路中，显然有：



$$I_{in} = I_R + R_C \Rightarrow I_{in} = \frac{V(t)}{R} + C\frac{dV(t)}{dt} \Rightarrow I_{in}R = V(t) + RC\frac{dV(t)}{dt}.$$

上式可看作一个一阶常系数线性微分方程。已知对于微分方程 $y' + p(x)y = q(x)$，其通解可写成：

$$y = e^{-\int p(x)dx}[\int q(x)e^{\int p(x)dx}dx + C].$$

那么，解上述微分方程可得：

$$V(t) = I_{in}R + c_1 e^{-\frac{-t}{RC}}.$$

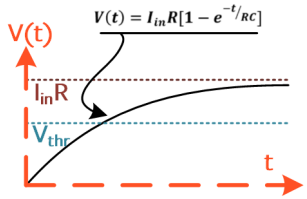
根据边界条件，即当 $t = 0$ 时，$V(t) = 0$，可得：

$$c_1 = -I_{in}R.$$

代入可得：

$$V(t) = I_{in}R[1 - e^{-\frac{t}{RC}}].$$

根据上式及边界条件，$V(t)$ 随时间的变化曲线可表示为：



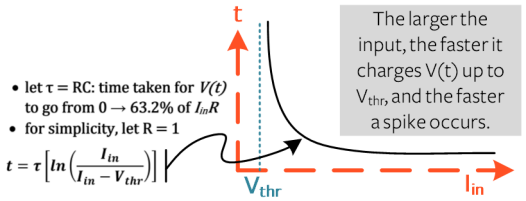
假设脉冲在 $V(t)$ 到达阈值 V_{thr} 时发射，那么这一时刻可以表示为：

$$t = RC[ln(\frac{I_{in}R}{I_{in}R - V_{thr}})].$$

令 $\tau = RC$（$V(t)$ 从 0 达到 $I_{in}R$ 的 63.2% 的时刻），且简洁起见，令 $R = 1$，那么：

$$t = \tau[ln(\frac{I_{in}}{I_{in} - V_{thr}})].$$

因此，时间 t 与输入电流 I_{in} 的关系曲线为对数形式：



在上述原理性描述中，容易发现，有两个关键的参数需要配置：

- τ：RC 时间常数，值越大，V-t 曲线上升越缓慢，也就意味着神经元的激活更慢；
- V_{thr}：膜电位的激活阈值，小于阈值的输入将不会激活神经元，在 SNN 中通常取最后时刻激活。

在下面的例子中，我们将上述两个参数分别设置为 5 和 0.01。

[30]:
```
data = iter(mnist_examples_loader)
data_it, targets_it = next(data)

# 获得编码数据
spike_data = spikegen.latency(data_it, num_steps=200, tau=5, threshold=0.01)
```

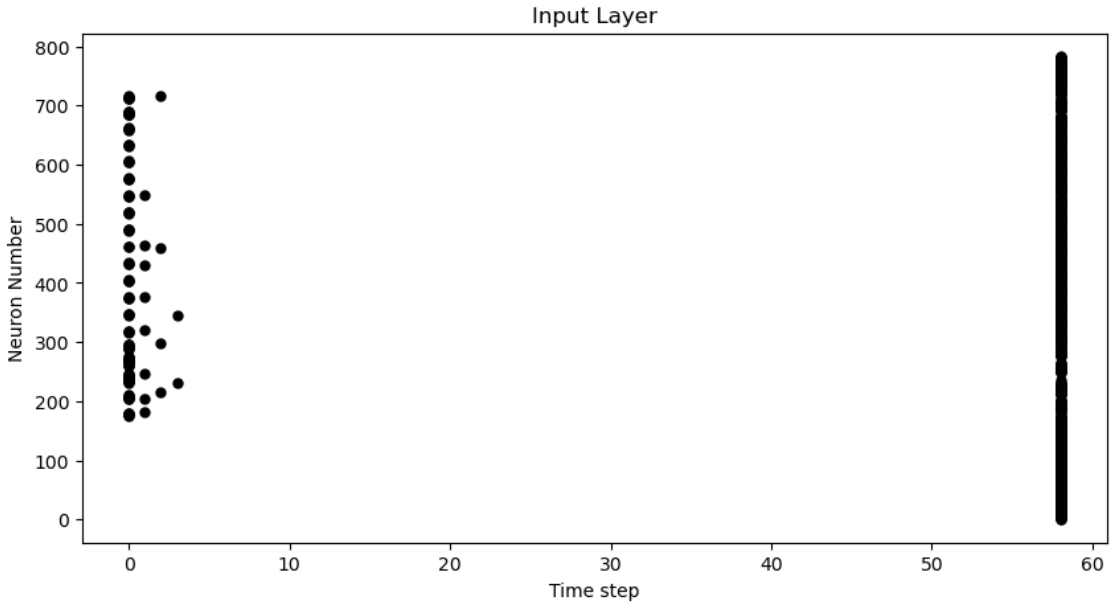
[26]:
```
# 查看数据的维度
print(spike_data.size())
```

```
torch.Size([100, 4, 1, 28, 28])
```

4.2.3 可视化

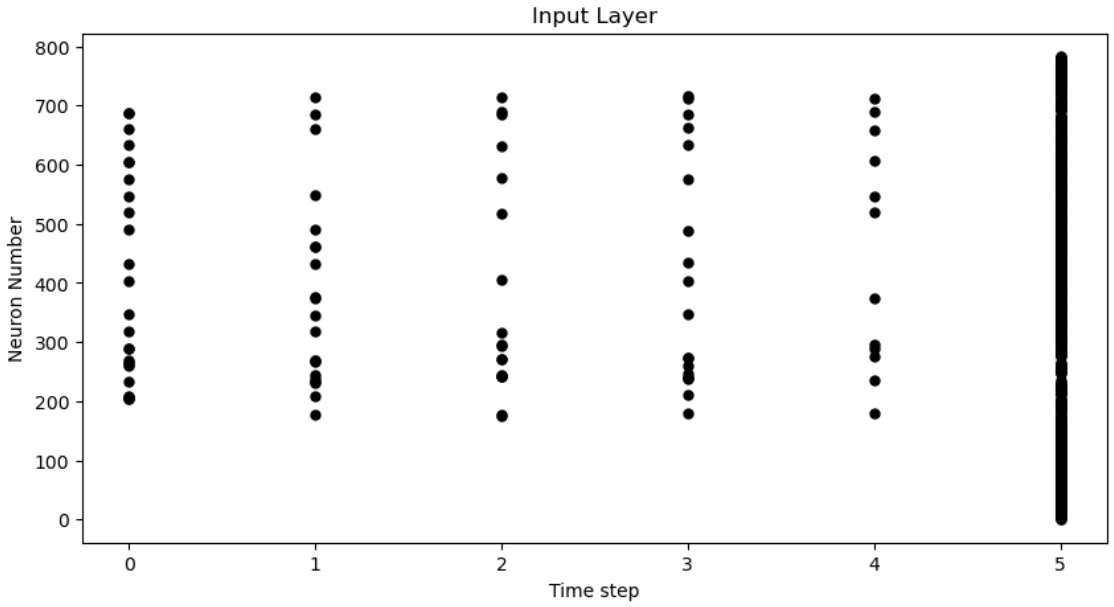
我们首先尝试采用栅格图的方式进行可视化。

[31]:
```
spike_data_sample = spike_data[:, 0]

fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data_sample.reshape(num_steps, -1), ax, s=25, c="black")

plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```



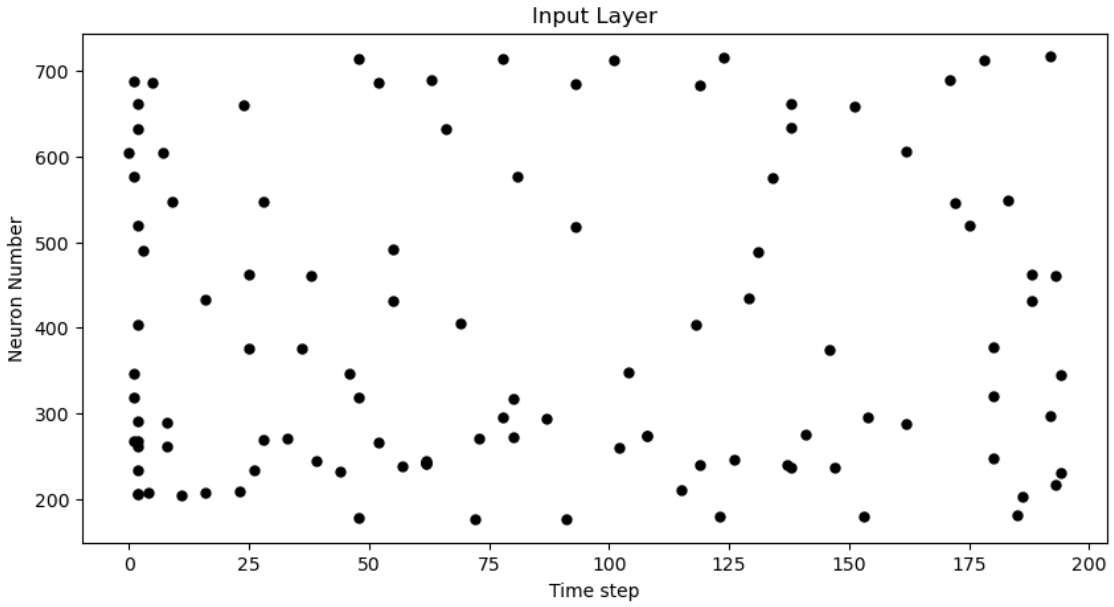
由于输入图片的多数像素都为黑或白，缺乏中间色调，因此激活的时间集中在图的左右两侧。我们可以尝试提高 τ 值延缓激活的时间，并通过 `linear=True` 线性化激活时间，这相当于我们把对数关系改变为线性关系（此时模型没有物理意义，仅表达一种数量关系）。

[32]:
```
spike_data = spikegen.latency(data_it, num_steps=200, tau=5, threshold=0.01,␣
↪linear=True)

fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data[:, 0].reshape(num_steps, -1), ax, s=25, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```



从上图中可以看到，尽管我们将时间周期长度定位 100 个单位，但大多数神经元都是在前几个时间单位激活的。我们可以通过指定 `normalize=True` 使时间分布在整个时间周期。此外，由于在 MNIST 样本图像中，全黑的背景对于数字的识别是没有意义的，我们可以通过 `clip=True` 裁剪掉这些神经元。

[34]:
```
spike_data = spikegen.latency(data_it, num_steps=200, tau=5, threshold=0.01,␣
↪normalize=True, linear=True, clip=True)

fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data[:, 0].reshape(num_steps, -1), ax, s=25, c="black")

plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```

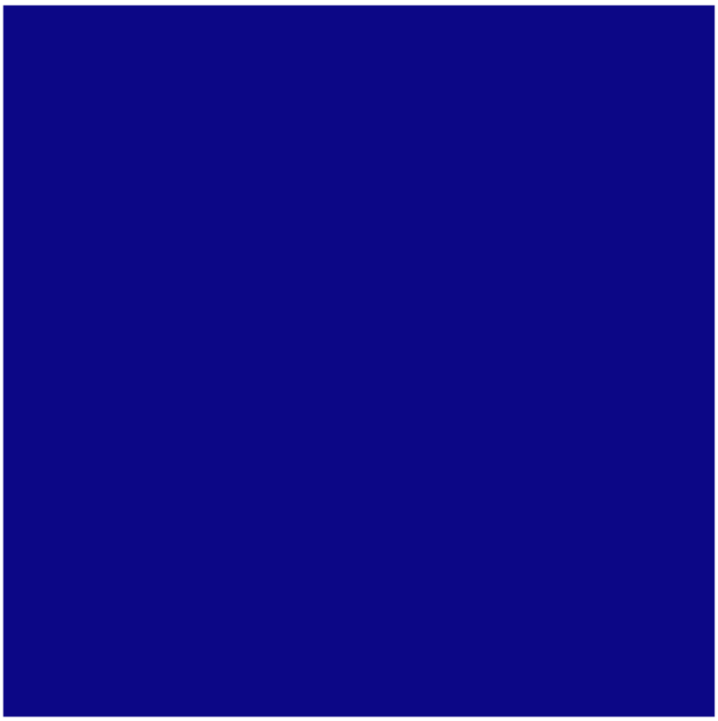


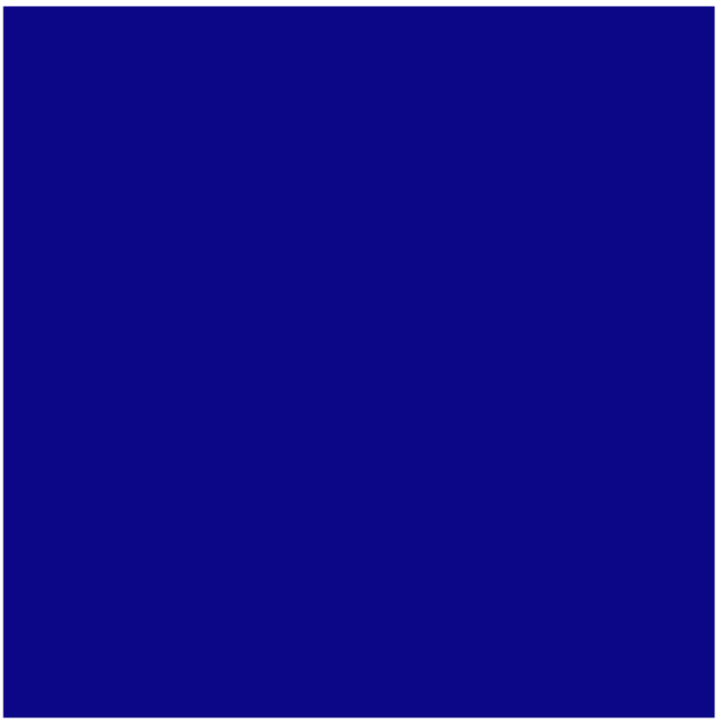
我们也可通过动画的方式进行可视化。

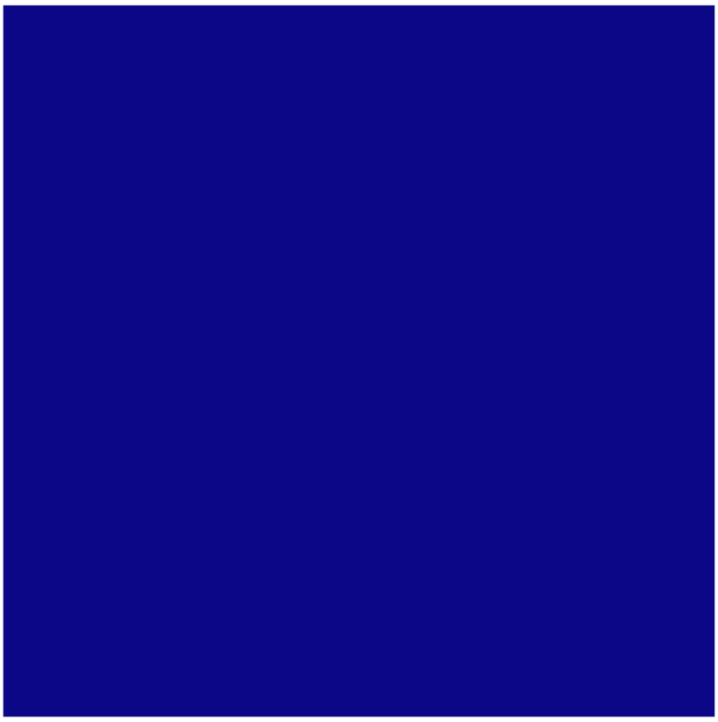
[36]:
```
# 使用 Latency coding 方法，对测试集前 4 张图片进行编码
for i in range(4):
    image = data_it[i]
    print(target_label[targets_it[i].item()])

    spike_data = spikegen.latency(image, num_steps=200, normalize=True,␣
 ↪clip=True, linear=True)

    fig, ax = plt.subplots()
    anim = splt.animator(spike_data[:,0], fig, ax)
    anim.save(
        "videos/latency-coding-video-MNIST-{}.gif".format(
            target_label[targets_it[i].item()].lower().replace(' ','-')
        )
    )
    display(HTML(anim.to_html5_video()))
```

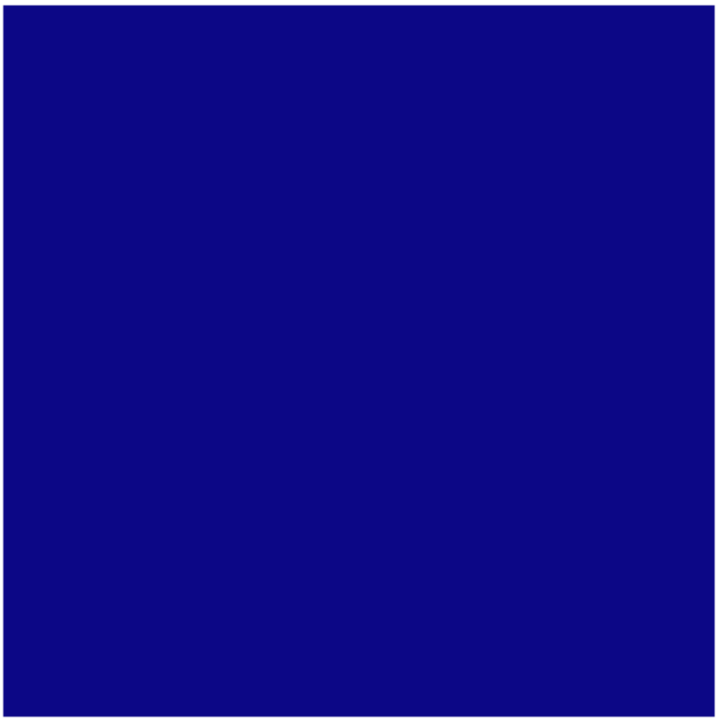
```
Five
```

```
<IPython.core.display.HTML object>

Eight

<IPython.core.display.HTML object>

Three

<IPython.core.display.HTML object>

Five

<IPython.core.display.HTML object>
```









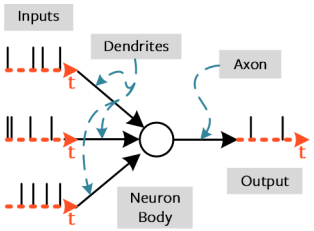
5 脉冲神经网络的搭建

5.1 神经元模型

根据生物神经系统的工作原理，可以构建多种多样的神经元模型。目前常见的神经元模型可分为以下三类：

- Hodgkin-Huxley 神经元模型：准确性强，但比较复杂；
- 人工神经元模型：即感知机、多层感知机等采用的模型，使用简单，但无法真实反映神经元的实际工作原理；
- Leaky Integrate-and-Fire 模型（LIF 神经元模型、整合-发射神经元模型）：在拟真性和易用性上进行了平衡。

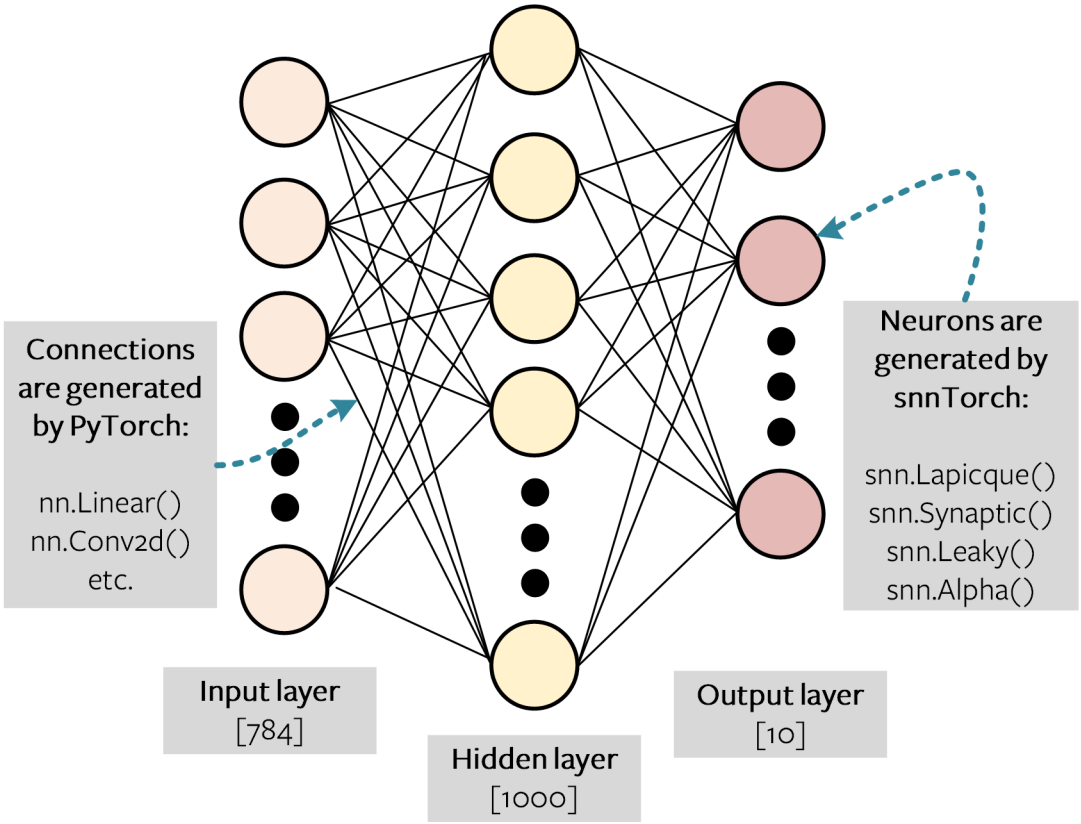
本实验将采用 LIF 神经元模型。在脉冲神经网络中，神经元的输入、输出均为脉冲：



5.2 前馈脉冲神经网络

通过 snnTorch 可以快速构建多层脉冲神经网络。在本实验中，我们将构建一个 3 层全连接脉冲神经网络，各层神经元数量分别为 784、1000、10。

该网络的架构如下：



我们使用 PyTorch 创建神经元的连接，而使用 snnTorch 创建神经元。

首先初始化各层。

[38]:
```
import snntorch as snn
from snntorch import spikeplot as splt
from snntorch import spikegen

import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

[39]:
```
# 网络参数
num_inputs = 784
num_hidden = 1000
num_outputs = 10
beta = 0.99

# 初始化各层
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Leaky(beta=beta)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Leaky(beta=beta)
```

接下来，初始化隐藏层状态和脉冲神经元的输出。

[61]:
```
# 初始化隐藏层状态
mem1 = lif1.init_leaky()
mem2 = lif2.init_leaky()

# 用于记录神经元输出的变量
mem2_rec = []
spk1_rec = []
spk2_rec = []
```

为了进行测试，我们首先创建一个随机脉冲序列。假设序列长度为 200，神经元数量为 784，那么输入的维度为 200×784。在 PyTorch 中，数据是以 Batch 为单位输入的，所以我们增加一个 Batch 维度。这里，我们使用 `unsqueeze(dim=1)`，即 Batch 尺寸为 1。

[41]:
```
spk_in = spikegen.rate_conv(torch.rand((200, 784))).unsqueeze(1)
```

[42]:
```
# 查看维度
print(f"Dimensions of spk_in: {spk_in.size()}")
```

```
Dimensions of spk_in: torch.Size([200, 1, 784])
```

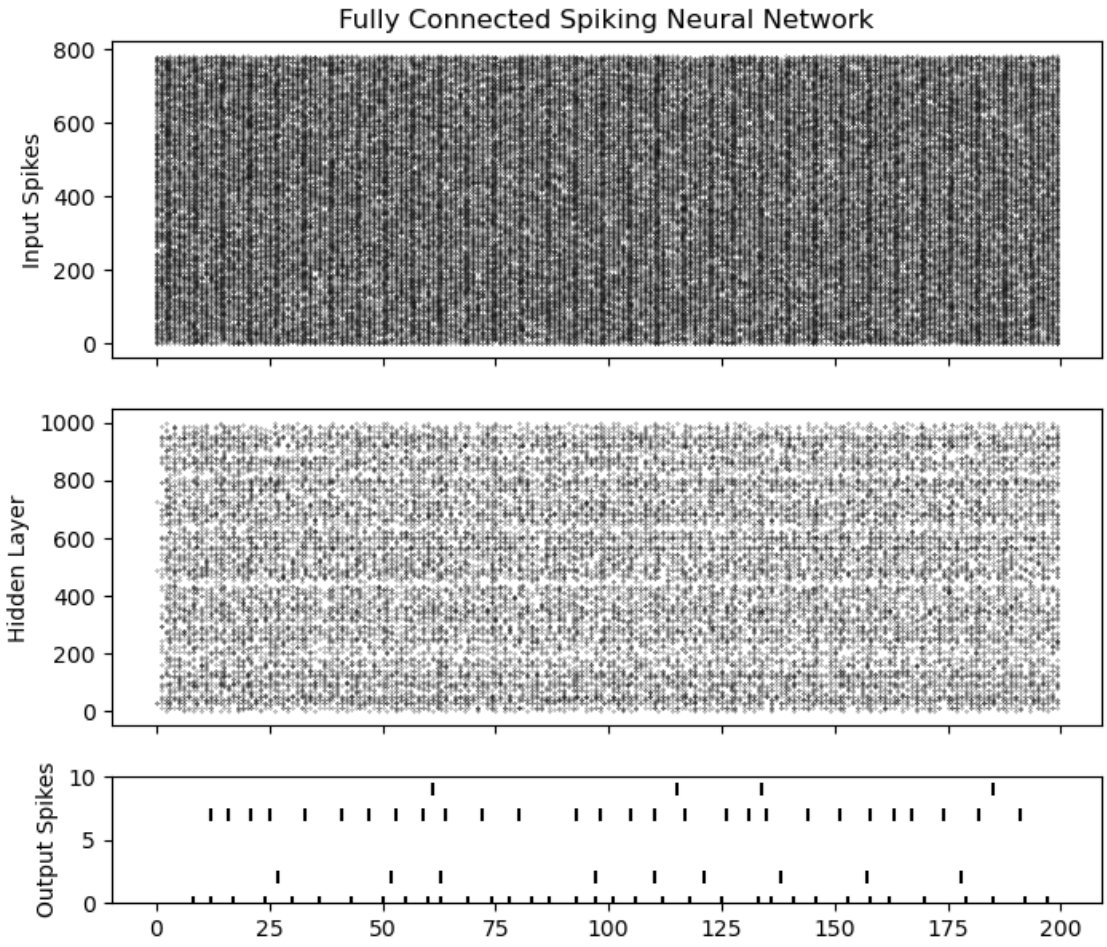
创建一个工具函数，用于进行 SNN 脉冲的可视化。

[59]:
```
def plot_cur_mem_spk(cur, mem, spk, thr_line=False, vline=False, title=False,␣
↪ylim_max1=1.25, ylim_max2=1.25):
    fig, ax = plt.subplots(3, figsize=(8,6), sharex=True,
                           gridspec_kw = {'height_ratios': [1, 1, 0.4]})

    ax[0].plot(cur, c="tab:orange")
    ax[0].set_ylim([0, ylim_max1])
    ax[0].set_xlim([0, 200])
    ax[0].set_ylabel("Input Current ($I_{in}$)")
    if title:
        ax[0].set_title(title)

    ax[1].plot(mem)
    ax[1].set_ylim([0, ylim_max2])
    ax[1].set_ylabel("Membrane Potential ($U_{mem}$)")
    if thr_line:
        ax[1].axhline(y=thr_line, alpha=0.25, linestyle="dashed", c="black",␣
↪linewidth=2)
    plt.xlabel("Time step")

    splt.raster(spk, ax[2], s=400, c="black", marker="|")
    if vline:
        ax[2].axvline(x=vline, ymin=0, ymax=6.75, alpha = 0.15,␣
↪linestyle="dashed", c="black", linewidth=2, zorder=0, clip_on=False)
    plt.ylabel("Output spikes")
    plt.yticks([])

    plt.show()
```

```
def plot_snn_spikes(spk_in, spk1_rec, spk2_rec, title):
    fig, ax = plt.subplots(3, figsize=(8,7), sharex=True,
                           gridspec_kw = {'height_ratios': [1, 1, 0.4]})

    splt.raster(spk_in[:,0], ax[0], s=0.03, c="black")
    ax[0].set_ylabel("Input Spikes")
    ax[0].set_title(title)

    splt.raster(spk1_rec.reshape(num_steps, -1), ax[1], s = 0.05, c="black")
    ax[1].set_ylabel("Hidden Layer")

    splt.raster(spk2_rec.reshape(num_steps, -1), ax[2], c="black", marker="|")
    ax[2].set_ylabel("Output Spikes")
    ax[2].set_ylim([0, 10])

    plt.show()
```

对网络各层的数据进行可视化。

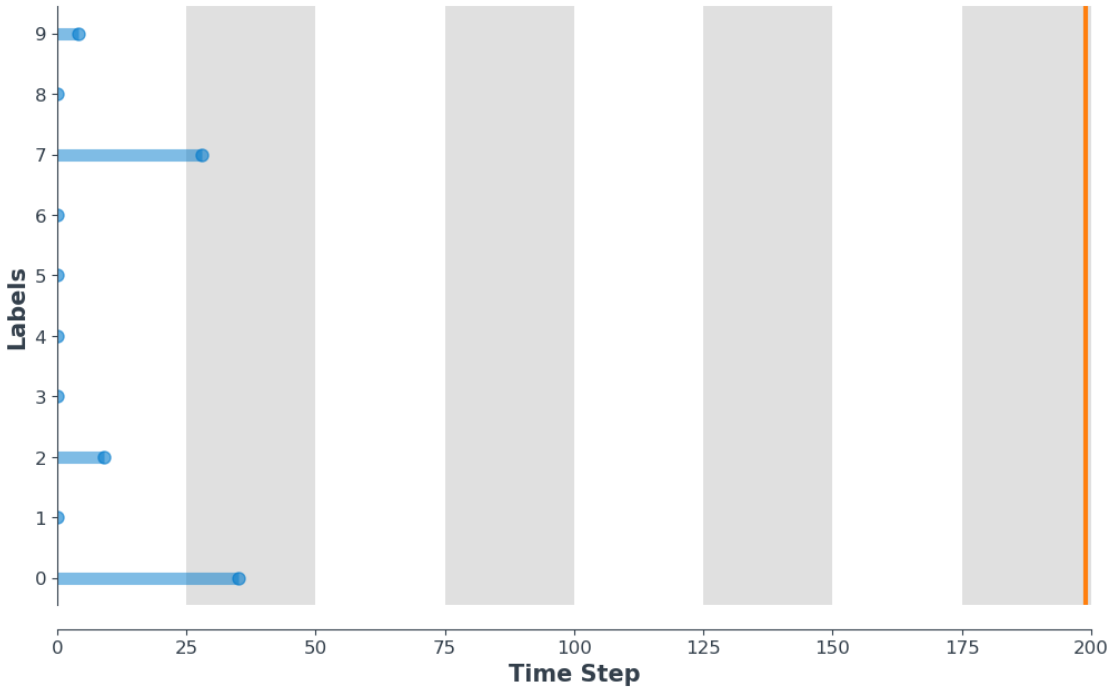
[62]:
```
for step in range(num_steps):
    cur1 = fc1(spk_in[step])
    spk1, mem1 = lif1(cur1, mem1)
    cur2 = fc2(spk1)
    spk2, mem2 = lif2(cur2, mem2)

    mem2_rec.append(mem2)
    spk1_rec.append(spk1)
    spk2_rec.append(spk2)

mem2_rec = torch.stack(mem2_rec)
spk1_rec = torch.stack(spk1_rec)
spk2_rec = torch.stack(spk2_rec)

plot_snn_spikes(spk_in, spk1_rec, spk2_rec, "Fully Connected Spiking Neural␣
 ↪Network")
```



spikeplot.spike_count 能够创建输出层的脉冲计数器。

[64]:
```
from IPython.display import HTML

fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8','9']
spk2_rec = spk2_rec.squeeze(1).detach().cpu()

anim = splt.spike_count(spk2_rec, fig, ax, labels=labels, animate=True)
display(HTML(anim.to_html5_video()))
anim.save("videos/spike_bar.mp4")
```

```
<IPython.core.display.HTML object>
```



6 脉冲神经网络的训练和效果评估

最后，我们训练一个简单的脉冲神经网络，用于完成 MNIST 手写数字识别任务。

6.1 准备工作

[93]:
```
# 数据存放的位置
data_root='data'

# 数据类型
dtype = torch.float

# 判断有无可用的 GPU，如果没有则使用 CPU
device = torch.device("cuda") if torch.cuda.is_available() else torch.
  ↪device("cpu")
```

[94]:
```
# 批尺寸
BATCH_SIZE = 512
```

6.2 数据集及数据加载器

[95]:
```
# 定义一个变换
transform = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize((0,), (1,))
])
```

[96]:
```
# 定义训练集、测试集，应用变换
mnist_train = datasets.MNIST(data_root, train=True, download=True,␣
 ↪transform=transform)
mnist_test = datasets.MNIST(data_root, train=False, download=True,␣
 ↪transform=transform)
```

[97]:
```
# 定义数据加载器
train_loader = DataLoader(mnist_train, batch_size=BATCH_SIZE, shuffle=True,␣
 ↪drop_last=True)
test_loader = DataLoader(mnist_test, batch_size=BATCH_SIZE, shuffle=True,␣
 ↪drop_last=True)
```

6.3 创建工具函数

[98]:
```
def plot_mem_rec(mem_rec, batch_size, targets):
    num_steps = len(mem_rec)
    for i in range(batch_size):
        fig = plt.figure()
        ax = fig.subplots()
        ax.set_xlim((-10,210))
        ax.set_ylim((-2,2))
        ss = mem_rec[:,i,:]
        plt.plot(range(0,num_steps), ss.cpu().detach())
```

```
        plt.title("Output Layer Membrane Output - {}".format(targets[i]))
        plt.xlabel("Time step")
        plt.ylabel("Neuron Number")
        fig.tight_layout()
    plt.show()


def plot_spk_rec(spk_rec, batch_size, targets):
    for i in range(batch_size):
        fig = plt.figure()
        ax = fig.subplots()
        ax.set_xlim((-10,210))
        ax.set_ylim((-1,11))
        ax.set_yticks(range(0,11))
        ss = spk_rec[:,i,:]
        splt.raster(ss, ax, s=1, c="black")

        plt.title("Output Layer - {}".format(targets[i]))
        plt.xlabel("Time step")
        plt.ylabel("Neuron Number")
        fig.tight_layout()
    plt.show()


def print_batch_accuracy(net, data, targets, train=False):
    output, _ = net(data.view(BATCH_SIZE, -1))
    _, idx = output.sum(dim=0).max(1)
    acc = np.mean((targets == idx).detach().cpu().numpy())

    if train:
        print(f"Train set accuracy for a single minibatch: {acc*100:.2f}%")
    else:
        print(f"Test set accuracy for a single minibatch: {acc*100:.2f}%")

    return acc
```

```
def train_printer():
    print(f"Epoch {epoch}, Iteration {iter_counter}")
    print(f"Train Set Loss: {loss_hist[counter]:.2f}")
    print(f"Test Set Loss: {test_loss_hist[counter]:.2f}")
    print_batch_accuracy(data, targets, train=True)
    print_batch_accuracy(test_data, test_targets, train=False)
    print("\n")
```

6.4 定义脉冲神经网络结构

[99]:
```
import torch.nn as nn
import snntorch as snn
import numpy as np
```

[100]:
```
# 网络参数
num_inputs = 28 * 28
num_hidden = 512
num_outputs = 10

# 训练参数
num_steps = 200
beta = 0.95
```

[101]:
```
# 定义网络结构
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # 初始化各层
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta)

    def forward(self, x):
```

```
        # 在 t=0 时刻，初始化隐含层状态
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        # 记录输出层信息
        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)
            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)
```

6.5 网络实例化与模型训练

[102]:
```
# 实例化
net = Net().to(device=device)
```

[103]:
```
# 查看数据维度
data, targets = next(iter(train_loader))
print(data.size())
print(targets.size())
```

```
torch.Size([512, 1, 28, 28])
torch.Size([512])
```

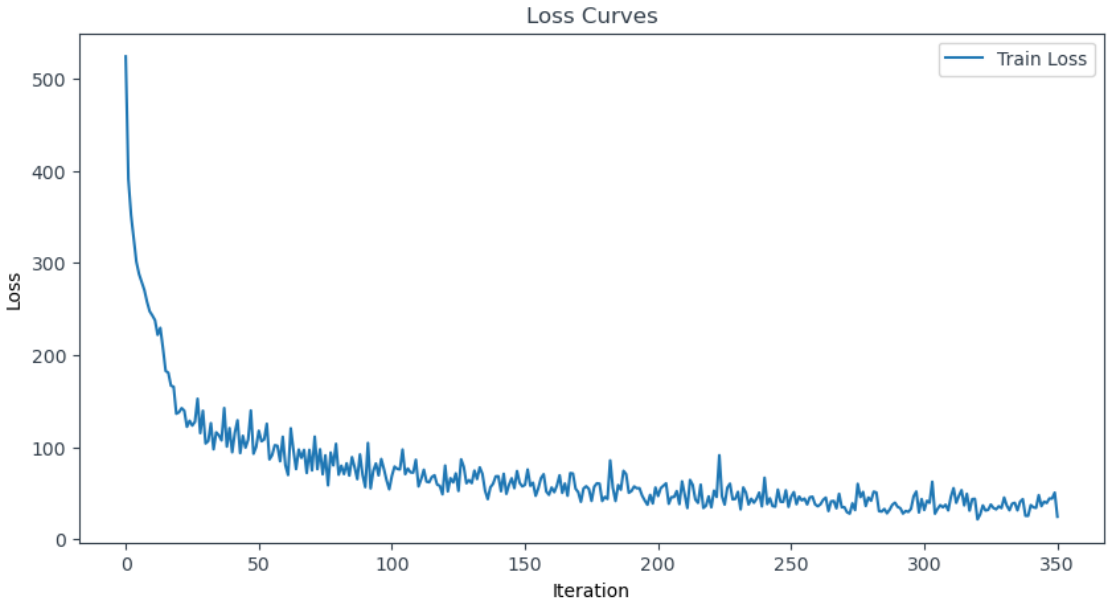
[104]:
```
# 查看数据记录变量的维度
print(data.view(BATCH_SIZE, -1).size())
spk_rec, mem_rec = net(data.to(device).view(BATCH_SIZE,-1))
print(spk_rec.size())
print(mem_rec.size())
```

```
torch.Size([512, 784])
torch.Size([200, 512, 10])
torch.Size([200, 512, 10])
```

[ ]:
```python
# 模型训练
run_train = False
state_dict_file_path = "models/snnTorch-MNIST-training.pt"

try:
    # 如果有模型加载点则载入
    load_state_dict = torch.load(state_dict_file_path, map_location=device)
    net.load_state_dict(load_state_dict)
except FileNotFoundError:
    print("File not found running training")
    run_train = True

if run_train == True:
    num_epochs = 3
    loss_hist = []
    test_loss_hist = []
    counter = 0
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=5e-4, betas=(0.9, 0.999))
    net.train()

    # 外层循环
    for epoch in range(num_epochs):
        # 在每个 mini batch 上训练
        for data, targets in train_loader:
            print( "Epoch ", epoch, " Iteration: ", counter)
            data = data.to(device)
            targets = targets.to(device)

            # 前向传播
            optimizer.zero_grad()
            spk_rec, mem_rec = net(data.view(BATCH_SIZE, -1))
```

```
            # 初始化模型损失，并沿时间维度统计模型损失
            loss_val = torch.zeros((1), dtype=dtype, device=device)
            for step in range(num_steps):
                loss_val += loss(mem_rec[step], targets)
            print(loss_val)

            # 反向传播
            loss_val.backward()
            optimizer.step()

            # 存储模型损失，用于后续的训练过程可视化
            loss_hist.append(loss_val.item())
            counter += 1
            print_batch_accuracy(net, data, targets, train=True)

    torch.save(net.state_dict(), state_dict_file_path)
```

[106]:
```
# 绘制训练过程的模型损失
fig = plt.figure(facecolor="w", figsize=(10, 5))
plt.plot(loss_hist)
plt.title("Loss Curves")
plt.legend(["Train Loss"])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()
```



6.6 模型测试

[107]:
```
with torch.no_grad():
    net.eval()
    acc_rec = np.array([])

    for test_data, test_targets in test_loader:
        test_data = test_data.to(device)
        test_targets = test_targets.to(device)

        # 在测试集上前向传播
        test_spk, test_mem = net(test_data.view(BATCH_SIZE, -1))

        # 测试集损失
        test_loss = torch.zeros((1), dtype=dtype, device=device)
        for step in range(num_steps):
            test_loss += loss(test_mem[step], test_targets)
        test_loss_hist.append(test_loss.item())
```

```
        # 输出测试集上的评估结果
        acc = print_batch_accuracy(net, test_data, test_targets)
        acc_rec = np.append(acc_rec, acc)
        avg_acc = np.mean(acc_rec)

    print(f"Test set average accuracy: {avg_acc*100:.2f}%")
```

```
Test set accuracy for a single minibatch: 95.12%
Test set accuracy for a single minibatch: 94.14%
Test set accuracy for a single minibatch: 95.12%
Test set accuracy for a single minibatch: 96.68%
Test set accuracy for a single minibatch: 96.29%
Test set accuracy for a single minibatch: 94.92%
Test set accuracy for a single minibatch: 94.14%
Test set accuracy for a single minibatch: 94.92%
Test set accuracy for a single minibatch: 94.14%
Test set accuracy for a single minibatch: 97.27%
Test set accuracy for a single minibatch: 95.51%
Test set accuracy for a single minibatch: 96.09%
Test set accuracy for a single minibatch: 96.48%
Test set accuracy for a single minibatch: 96.68%
Test set accuracy for a single minibatch: 97.07%
Test set accuracy for a single minibatch: 95.12%
Test set accuracy for a single minibatch: 95.31%
Test set accuracy for a single minibatch: 95.12%
Test set accuracy for a single minibatch: 95.12%
Test set average accuracy: 95.54%
```