SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence

SpikingJelly：面向脉冲智能的开源机器学习基础设施平台

方维,[1,2] 陈彦骐,[1,2] 丁健豪,[1] 余肇飞,[4] Timothée Masquelier,[5]
陈鼎,[6,2] 黄力炜,[1,2] 周晖晖,[2] 李国齐,[7,8,2*] 田永鸿 [1,2,3*]

[1] 北京大学计算机学院，中国

[2] 鹏城实验室，中国

[3] 北京大学深圳研究生院信息工程学院，中国

[4] 北京大学人工智能研究院，中国

[5] 脑与认知研究中心（CERCO），法国国家科学研究中心 UMR5549 - 图卢兹第三大学，法国

[6] 上海交通大学计算机科学与工程系，中国

[7] 中国科学院自动化研究所，中国

[8] 中国科学院大学人工智能学院，中国

*E-mail: guoqi.li@ia.ac.cn, yhtian@pku.edu.cn

脉冲神经网络（SNN）通过引入神经元动力学和脉冲机制，为在高能效神经形态芯片上实现类脑智能提供了可能。随着脉冲深度学习这一新兴范式受到越来越多关注，传统编程框架已难以满足自动微分、并行计算加速、神经形态数据集处理以及部署等高度集成的需求。为此，我们提出了 Spiking-

Jelly 框架。它提供了一套全栈工具，用于预处理神经形态数据集、构建深度 SNN、优化网络参数，并将 SNN 部署到神经形态芯片上。与现有方法相比，SpikingJelly 可将深度 SNN 的训练加速至最高 11×；同时，其优异的可扩展性与灵活性也使用户能够通过多层继承和半自动代码生成，以较低成本高效实现自定义模型。SpikingJelly 为构建真正节能的基于 SNN 的机器智能系统铺平了道路，并将进一步丰富神经形态计算的研究生态。

介绍

近年来，人工神经网络（ANN），例如卷积神经网络（CNN）(*1*)、循环神经网络（RNN）(*2*) 和 Transformer (*3*)，在图像分类 (*1,4,5*)、目标检测 (*6–8*)、机器翻译 (*3,9–11*)、语音识别 (*12,13*) 和博弈 (*14,15*) 等多个领域击败了大多数其他方法，甚至在部分任务上超过了普通人类水平。这些进展主要得益于基于梯度的数值优化方法 (*16,17*)、大规模数据 (*18,19*) 以及借助图形处理器（GPU）实现的大规模并行计算 (*20,21*)。尽管神经科学对 ANN 的直接影响在一定程度上有所减弱 (*22*)，但神经科学所提供的启发对于构建人类水平的通用人工智能（AI）系统仍然至关重要 (*23,24*)。人脑是已知最智能的系统之一，在迁移学习、持续学习等认知与学习任务上，相比现有人工系统仍具有显著优势 (*24*)。因此，神经科学界一直在探索更加符合生物机制的计算范式，以理解、模仿并利用人脑卓越的信息处理能力。与此相应，面向神经科学的脉冲神经网络（SNN）被提出，并常被视为下一代神经网络 (*25*)。SNN 使用短暂电脉冲（即脉冲）来处理和传递信息，因此与生物神经系统更加相似。SNN 中的神经元是对生物神经元更低层次的抽象：它们从树突接收信号，并通过非线性的神经元动力学处理刺激，因此 SNN 在处理时空数据方面具有竞争力 (*26,27*)。基于脉冲的生物神经系统极其节能，例如人脑的功耗仅约为 20 W (*28*)。受益于事件驱动的计算方式，SNN 在定制神经形态芯片上运行时，包括 TrueNorth (*29*)、Loihi (*30*) 和天机 (*31*)，其能效最高可比 ANN 提升 1000× (*31*)。近

年来，SNN 的生物学合理性、时空信息处理能力和事件驱动的计算范式吸引了越来越多的研究兴趣。

新兴的脉冲深度学习方法由于其不可微的脉冲触发机制和复杂的时空传播过程，为 SNN 设计高性能学习算法具有挑战性。传统的学习算法主要结合了生物学上合理的无监督学习规则和原始的基于梯度的监督学习方法。受生物神经系统启发的无监督学习算法已被用于 SNN，包括 Hebbian 学习 (*32*)、脉冲时序依赖可塑性（STDP）(*33*) 及其变体 (*34–38*)。早期的监督学习方法，例如 SpikeProp (*39*)、Tempotron (*40*)、ReSume (*41*) 和 SPAN (*42*)，虽然性能优于生物学上合理的无监督方法，但仍存在明显局限。大多数基于 SpikeProp 的方法只允许脉冲神经元发放不超过一个脉冲，而 Tempotron、ReSume 和 SPAN 则无法训练超过一层的 SNN。因此，这些早期监督学习方法只能处理难度不超过 MNIST 分类的任务 (*43*)。

导致人工神经网络快速进步的关键技术之一是深度学习 (*44*)，它通过反向传播优化多层人工神经网络的参数，并学习具有多个抽象级别的数据的高维表示。为了克服 SNN 训练中的挑战，研究者开始探索将深度学习方法引入 SNN，并取得了显著的性能提升。当前 SNN 中最常用的两类深度学习方法是代理梯度法 (*45–47*) 和从 ANN 到 SNN 的转换（ANN2SNN）(*48–53*)。采用代理梯度训练的 SNN 在 CIFAR (*54*)、DVS Gesture (*55*) 以及具有挑战性的 ImageNet (*19*) 等数据集上，仅用少量仿真时间步便取得了很高性能 (*56–61*)；而由 ANN 转换得到的 SNN，则能够在使用数十个仿真时间步的情况下，在 ImageNet 上达到与原始 ANN 几乎相同的精度 (*51, 62, 63*)。随着深度学习方法快速发展，SNN 的应用已从分类拓展到目标检测 (*64–66*)、目标分割 (*67, 68*)、深度估计 (*69*) 和光流估计 (*70*) 等任务。研究的蓬勃发展表明，脉冲深度学习已经成为一个极具潜力的研究方向。

对框架的需求人工神经网络开发的经验表明，软件框架在深度学习中发挥着至关重要的作用。现代框架，包括 TensorFlow (*71*)、Keras (*72*) 和 PyTorch (*73*)，提供了由 Python 实现的用户友好前端接口，以及由 C++ 库加速的高性能后端，例如 Intel MKL

和 Nvidia CUDA。统计数据显示，现代框架发布后，新采用者和新项目的数量呈指数级增长 (*74*)。这表明它们显著降低了构建和训练 ANN 的工作量，帮助用户更快实现想法，也极大推动了深度学习研究的发展。机器学习框架的迅速演进本身也在加速研究界的进步，这进一步说明了框架之于深度学习的重要性。

然而，目前仍缺少一个真正成熟的脉冲深度学习框架。多数现有 SNN 框架，例如 NEURON (*75*)、NEST (*76*)、Brian1/2 (*77, 78*) 和 GENESIS (*79*)，能够构建具有复杂神经元动力学的精细脉冲神经元，通过数值方法近似常微分方程（ODE），并高精度模拟生物神经系统，但它们并不集成自动微分，而这正是基于梯度的深度学习所需的核心组件。这些框架构建的 SNN 具有很强的生物可信度，适合研究真实神经系统的功能，但并非为机器学习任务而设计。Nengo (*80*)、SpykeTorch (*81*) 和 BindsNET (*82*) 采用了更简化的神经元模型，因而计算复杂度更低，可以支持部分机器学习和强化学习算法，但仍缺乏现代 SNN 深度学习能力。

由于缺乏可用的框架，想要将先进的深度学习方法与 SNN 相结合的研究人员必须从头开始构建基本的脉冲神经元和突触，从而导致重复和不协调的工作。深度 SNN 涉及数据空间和时间维度上的大量矩阵运算，这需要研究人员改进代码以创建由 GPU 加速的高性能程序。这样的工作量增加了研究人员的负担。在神经形态领域，SNN 经常被用来处理来自神经形态传感器的数据并部署在神经形态计算芯片上，但数据处理和部署也需要大量的时间和精力。即便经验丰富的研究人员各自完成了项目，不同作者之间在编程语言、编码风格和模型定义上的不一致，也会使代码难以复用，并进一步割裂社区。如果存在一个至少具有以下三个特征的现代脉冲深度学习框架，则可以大大提高科学研究的效率：利用并加速基于脉冲的操作；支持 CPU/GPU 上的模拟和神经形态芯片上的部署；并提供用于构建、训练和分析深度 SNN 的全栈工具包。

SpikingJelly：脉冲深度学习的现代框架为了解决上述问题并促进脉冲深度学习的研究，我们提出了 SpikingJelly，一个开源深度学习框架，用于连接深度学习和 SNN。图 **1a** 给出了 SpikingJelly 架构的分层概览。SpikingJelly 基于最常用的机器学习框架

之一 PyTorch，支持在具备自动求导能力的 CPU 和 GPU 上模拟 SNN。除了 PyTorch 提供的加速之外，还采用了额外的 CUDA 内核来实现 GPU 级加速。为了在易用性、灵活扩展性和高性能之间取得平衡，SpikingJelly 将深度学习相关功能组织为四个部分：*Components*、*Functions*、*Acceleration* 和 *Networks*（参见补充材料中的 *Subpackages of Deep Learning*）。*Components* 提供构建深度 SNN 所需的基础模块，例如脉冲神经元和突触。*Functions* 则包含训练、仿真、分析、转换、量化和部署 SNN 的实用函数。其中一部分模块是 *Components* 中相应模块的函数式实现。通过这种设计，SpikingJelly 同时支持面向对象编程和过程式编程，以满足用户的不同需求。*Acceleration* 通过额外的半自动生成 CUDA 内核来加速 SNN 仿真，从而兼顾底层编程的高效率与代码生成带来的低开发成本。在此基础上，*Networks* 提供了诸如 Spiking ResNet 等经典的大规模网络结构，便于模型快速复用，也为初学者提供了开箱即用的 SNN 应用示例，因此获得了社区的广泛认可。考虑到由 ATIS、DAVIS、DVS 等神经形态传感器产生的数据集 (*83–85*) 在 SNN 中被广泛使用，SpikingJelly 集成了神经形态数据集处理功能，包括下载、统一数据布局以及与通用 NumPy (*86*) 格式兼容的读取接口。借助 *Functions* 中提供的 *quantize* 和 *exchange* 等包，SpikingJelly 还通过网络权重低比特量化器和用于部署 SNN 的转换函数，有效实现了与神经形态芯片的兼容。

作为一个全栈框架，SpikingJelly 使研究人员能够借助灵活便捷的 API 构建 SNN，以极高效率进行仿真，并将 SNN 部署到边缘 AI 设备上。SpikingJelly 为实现真正基于脉冲的节能机器智能范式提供了有效途径，也进一步丰富了该领域的研究生态。

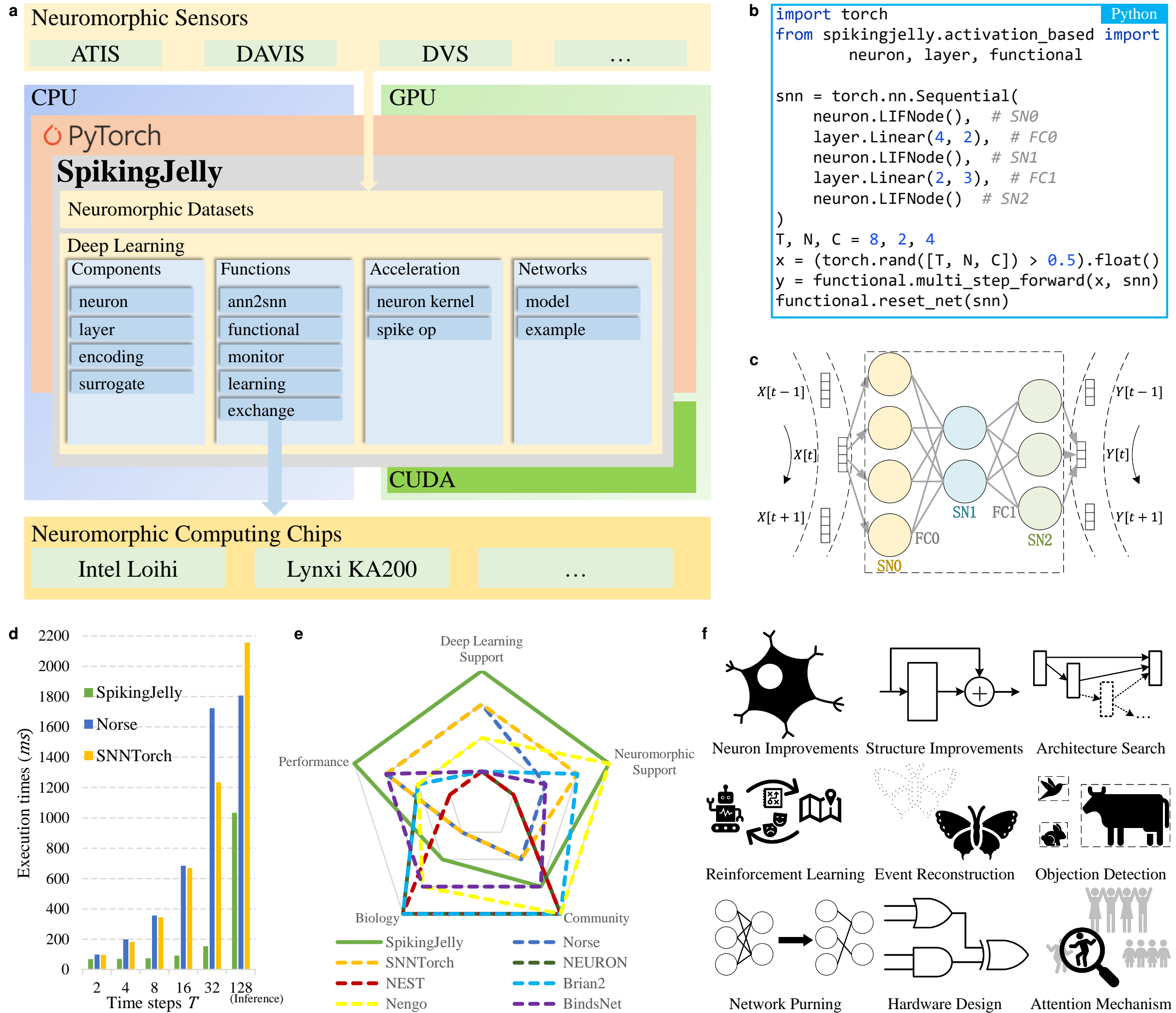

图 1: **SpikingJelly 概述。** **a.** 整个框架的架构。**b.** 构建和运行 SNN 的代码示例，其架构如 **c** 所示。**d.** 使用 SpikingJelly、Norse 和 SNNTorch 构建的 Spiking ResNet-18 进行单次训练迭代（$T = 2, 4, 8, 16, 32$）和推理（$T = 128$）所需的执行时间。**e.** SpikingJelly 与其他框架的生态位比较。**f.** 基于 SpikingJelly 的研究中的典型采用。

结果

便捷灵活的 SNN 构建

SpikingJelly 为用户提供了便捷灵活的 API，只需几个步骤即可构建 SNN。这些 API 采用广受好评的 PyTorch 风格设计。因此，精通深度学习的研究人员也能快速上手 SpikingJelly。图 1**b** 是构建和运行 SNN 的示例，其结构如图 1**c** 所示。注意，我们输入的是形状为 (T, N, C) 的三维张量 X，其中 T 是时间步数，N 是批量大小，C 是特征数。然后，我们使用 *multi_step_forward* 函数将 X 输入 SNN，该函数实现了对时间步的循环，将输入 $X[t]$ 发送到网络并将输出 $Y[t]$ 拼接为 Y。在 SpikingJelly 中，所有隐藏状态都存储在模块内部，这使得构建具有顺序层的网络和访问特定模块的属性变得非常方便。需要注意的是，在发送新样本之前需要重置隐藏状态，本例中我们调用 *reset_net* 函数来重置 SNN。

高性能仿真

SpikingJelly 利用 GPU 的大规模并行计算能力来实现极高的 SNN 仿真性能。为了进行定量评估，我们评估了现代脉冲深度学习框架在 Spiking ResNet 上的训练和推理性能，Spiking ResNet 是经典 ResNet ANN 结构 (*87*) 的脉冲版本，被 SNN 研究人员广泛使用 (*60, 88–91*)。比较的框架是 Norse (*92*) 和 SNNTorch (*93*)，它们与 SpikingJelly 几乎同时或稍后提出。Norse 和 SNNTorch 的版本分别为 1.0.0 和 0.6.0，均为最新版本。网络中使用了泄漏积分放电（LIF）神经元 (*94*)，因为它们是深度 SNN 中最常用的脉冲神经元之一。实验在 Ubuntu 18.04 服务器上进行，配备 Intel Xeon Silver 4210R CPU、NVIDIA A100-SXM-80GB GPU 和 256 GB 内存。

研究人员通常以两种方式模拟 SNN。第一种方法是通过代理梯度方法直接训练 SNN，使用时间反向传播（BPTT）。由于内存消耗与时间步数 T 近似成正比，直接训练通常使用较少的时间步；例如，对于高分辨率数据集 $T \leq 32$，对于小规模数据集

$T \leq 64$。第二种方法是使用通过 ANN2SNN 获得的权重运行 SNN。ANN2SNN 的评估仅限于推理，因为此过程不需要训练。大多数 ANN2SNN 方法基于速率编码，需要比代理梯度方法多得多的时间步，例如 $T \geq 128$。也有基于延迟编码而非速率编码的转换方法 (*95*)。值得注意的是，最近的研究已经大幅降低了 T，例如使用 $T = 5$ 的代理学习方法 (*96*) 和 $T \leq 64$ 的 ANN2SNN 方法 (*51, 97*)。我们的评估涉及最常见用法的两个步骤：1）使用小时间步进行代理学习训练；2）使用大时间步进行 ANN2SNN 推理。

首先，我们使用代理梯度方法训练 $T = 2, 4, 8, 16, 32$ 的 SNN。图 **1d** 显示了三个框架在 Spiking ResNet-18 上进行单次训练迭代所需的执行时间。可以发现，SpikingJelly 在训练速度上具有显著优势，当 T 较大时速度提升更大，例如当 $T = 32$ 时最高可达 11×。其次，我们测试了 SNN 在时间步 $T = 128, 256, 512, 1024$ 下的推理性能，以模拟 ANN2SNN 的用法。实验结果表明，三个框架的推理时间均与时间步数 T 成正比。因此，为简便起见，我们仅报告 $T = 128$ 的结果。如图 **1d** 所示，SpikingJelly 同样实现了比其他框架高达 2× 的加速。实验的源代码和结果在补充材料中提供。

神经形态设备支持

神经形态设备，包括传感器和计算芯片，是神经形态工程的关键组件，也常用于脉冲深度学习。SpikingJelly 提供了高效的接口，将 SNN 与神经形态设备集成。

大多数神经形态数据集是从神经形态传感器采集的（例如，N-MNIST (*98*) 从 ATIS 传感器采集，DVS Gesture 从 DVS128 相机采集），而其他数据集如 ES-ImageNet (*99*) 则通过软件算法从静态图像转换而来。通过传感器采集的原始数据以特定格式存储，例如 AEDAT，需要特定的二进制解码方法。为降低使用成本，SpikingJelly 实现了不同格式的解码方法。SpikingJelly 提供了基于事件和基于下采样帧的数据集表示。事件表示与地址事件表示（AER）格式相同，这也是默认的神经形态芯片间通信协议。第 i 个事件为 $\mathbf{E}[i] = (x_i, y_i, t_i, p_i)$，其中 (x_i, y_i) 是坐标，t_i 是时间戳，p_i 是极性。帧表示被广泛使用 (*45, 56, 57, 60, 100, 101*)，通常从事件表示进行时间下采样得到。帧 $\mathbf{F}$ 是一个包含

事件计数的四维张量，形状为 (T, C, H, W)，其中 T 是帧数，C 是通道数，H 和 W 分别是帧的高度和宽度。SpikingJelly 还提供了常用的事件到帧的下采样方法。

SNN 的优势在很大程度上取决于事件驱动的方式，而这只有在使用神经形态计算芯片时才能实现。为了实现这一目标，研究人员首先在强大的 CPU/GPU 上训练 SNN 以获得优化的权重，然后将预训练的 SNN 部署到神经形态计算芯片上进行推理。SpikingJelly 也支持这样的流程。SpikingJelly 提供必要的转换功能，将原生 SpikingJelly 模块转换为神经形态计算芯片支持的格式。这使得原生 SpikingJelly SNN 可以在神经形态芯片上运行（参见补充材料中的 *Exchange Modules*）。目前，SpikingJelly 支持两种最常用的神经形态计算芯片：来自 Intel 的 Loihi 和来自天机 (*31*) 团队的 Lynxi KA200。通过开发特定的交换模块，也可以在其他神经形态计算芯片上实现部署。

生态位

各种具有独特特征和特定优势的 SNN 框架已经可用，我们将这些特征和优势称为框架的生态位。为了说明 SpikingJelly 在其中的独特生态位，我们总结了常用 SNN 框架的特征。总体而言，框架可以分为三类。

第一类是经典生物学框架，包括 NEURON、NEST 和 Brian2，它们使用生物神经元模型的最低层抽象，并集成（或易于实现）生物学上合理的学习规则，如 STDP。为了支持 GPU，一些经典框架提供子框架，例如 NEURON 的 CoreNEURON 和 Brian2 的 Brian2GENN，以加速从父框架派生的部分模块。Brian2 还提供了 Brian2Loihi 来支持 Loihi 芯片。随着近几十年的快速发展，经典框架已被神经科学家广泛采用，并发展成为一个庞大而活跃的研究社区。

第二类可以被视为神经科学和计算机科学的交叉领域，包括 Nengo 和 BindsNet。这些框架采用 NumPy 风格设计，使用中等复杂度的神经元模型，这使得其计算成本低于经典框架。这些框架与 GPU 更兼容，因为它们提供 GPU 支持的后端，如 OpenCL，或直接基于完全支持 GPU 的现代机器学习框架实现。更具体地说，Nengo 使用基于 OpenCL

的 NengoOCL 或基于 TensorFlow 的 NengoDL 来使用 GPU，而 BindsNet 基于 PyTorch。此外，Nengo 通过 NengoLoihi 子框架支持 Loihi。生物学上合理的规则是这些框架中的主要训练算法。值得注意的是，Nengo 也通过 NengoDL 子框架支持 ANN2SNN。Nengo 丰富的生态系统使其成为最常用的框架之一。此外，新兴的 BindsNet 在 GitHub 社区中吸引了数千名关注者。

第三类考虑神经科学和深度学习的交叉领域，包括 Norse、SNNTorch 和 SpikingJelly。所有这些框架都基于 PyTorch，支持 GPU 和自动微分。采用高级神经元模型抽象，使得这些框架可以轻松地与反向传播配合使用。所有这些框架都至少支持一种深度学习方法。由于近年来对脉冲深度学习的日益增长的兴趣，这些框架受到了研究界的广泛关注。与其他框架相比，SpikingJelly 的优势在于全栈集成，支持神经形态数据集和芯片、ANN2SNN 和代理梯度以及生物学上合理的学习规则，并通过针对脉冲操作的特定优化技术最大化其仿真效率。

为了清晰说明，我们从五个方面比较了现有的 SNN 框架：仿真性能（*Performance*）、对神经形态传感器和计算芯片的支持（*Neuromorphic Support*）、社区规模（*Community*）、生物抽象级别（*Biology*）以及对代理学习和 ANN2SNN 的支持（*Deep Learning Support*）。结果如图 1e 所示，清楚地展示了 SpikingJelly 的生态位。有关框架差异的更多详细信息，请参阅补充材料中的 *Comparison of SNN Frameworks*。

社区采用

社区采用数量的不断增长标志着一个框架的成功。自 2019 年 12 月开源以来，SpikingJelly 已被广泛用于许多脉冲深度学习研究，包括对抗攻击 (*102, 103*)、ANN2SNN (*97, 104–108*)、注意力机制 (*109, 110*)、DVS 数据深度估计 (*69, 111*)、创新材料开发 (*112*)、情感识别 (*113*)、能量估计 (*114*)、基于事件的视频重建 (*115*)、故障诊断 (*116*)、硬件设计 (*117–119*)、网络结构改进 (*60, 61, 120–123*)、脉冲神经元改进 (*56, 124–129*)、训练方法改进 (*130–140*)、医学诊断 (*141, 142*)、网络剪枝 (*143–147*)、神经架构搜索 (*148, 149*)、神

经形态数据增强 (*150*)、自然语言处理 (*151*)、DVS/帧数据的目标检测/跟踪 (*65,66,152*)、气味识别 (*153*)、DVS 数据光流估计 (*154*)、强化学习控制 (*155–157*) 和语义通信 (*158*)。图 **1f** 展示了部分采用案例。社区的广泛采用标志着 SpikingJelly 成为最常用的脉冲深度学习框架之一。

SpikingJelly 核心模块

使 SpikingJelly 能够实现灵活性、效率和完整性结合的关键因素是其模块的设计理念。一些核心组件和功能模块分别如图 2 和图 3 所示。

图 2**a-e** 展示了 SpikingJelly 中的脉冲神经元及其组件。如图 2**a** 所示，SpikingJelly 使用三个离散时间方程（充电、放电和重置方程）来描述脉冲神经元的行为。$X[t], H[t], S[t]$ 和 $V[t]$ 分别是时间步 t 的输入、充电后的膜电位、脉冲和重置后的膜电位。图 2**b** 展示了 SpikingJelly 中 LIF 神经元对给定刺激的响应。图 2**c** 展示了 LIF 神经元类的继承关系。通过继承父类，LIF 神经元只需几行代码即可轻松实现。图 2**d** 展示了 Heaviside 函数 $\Theta(x)$、sigmoid 代理函数 $\sigma(x) = \frac{1}{1+\exp(-\alpha x)}$（其中 α 是用于控制形状的超参数）和代理梯度 $\sigma'(x)$，它们是图 2**c** 所示脉冲神经元的关键属性。图 2**e** 说明了代理学习在 SpikingJelly 中的实现方式。在前向传播期间应用 $\Theta(x)$ 生成脉冲，在反向传播期间应用 $\sigma'(x)$ 计算梯度。图 2**f**, **g** 可视化了 SpikingJelly 中两种典型的脉冲编码器。图 2**f** 所示的延迟编码器将较大的值（较深的颜色）编码为较早的脉冲。图 2**g** 所示的泊松编码器将输入值编码为放电概率服从泊松分布的脉冲。图 2**h** 展示了具有两种极性的合成事件，SpikingJelly 下采样的帧如图 2**i** 所示。SpikingJelly 中用于处理数据集的工作流程如图 2**j** 所示。

SpikingJelly 中的模块具有 *step_mode* 属性，用于决定是应用单步还是多步前向传播。单步模式下的模块接收 $X[t]$ 并输出 $Y[t]$，它们是单个时间步的数据。相比之下，多步模式下的模块接收 $X = \{X[0], X[1], ..., X[T-1]\}$ 并输出 $Y = \{Y[0], Y[1], ..., Y[T-1]\}$，它们是包含多个时间步数据的序列。相应地，所有模块处于单步或多步模式的 SNN 分

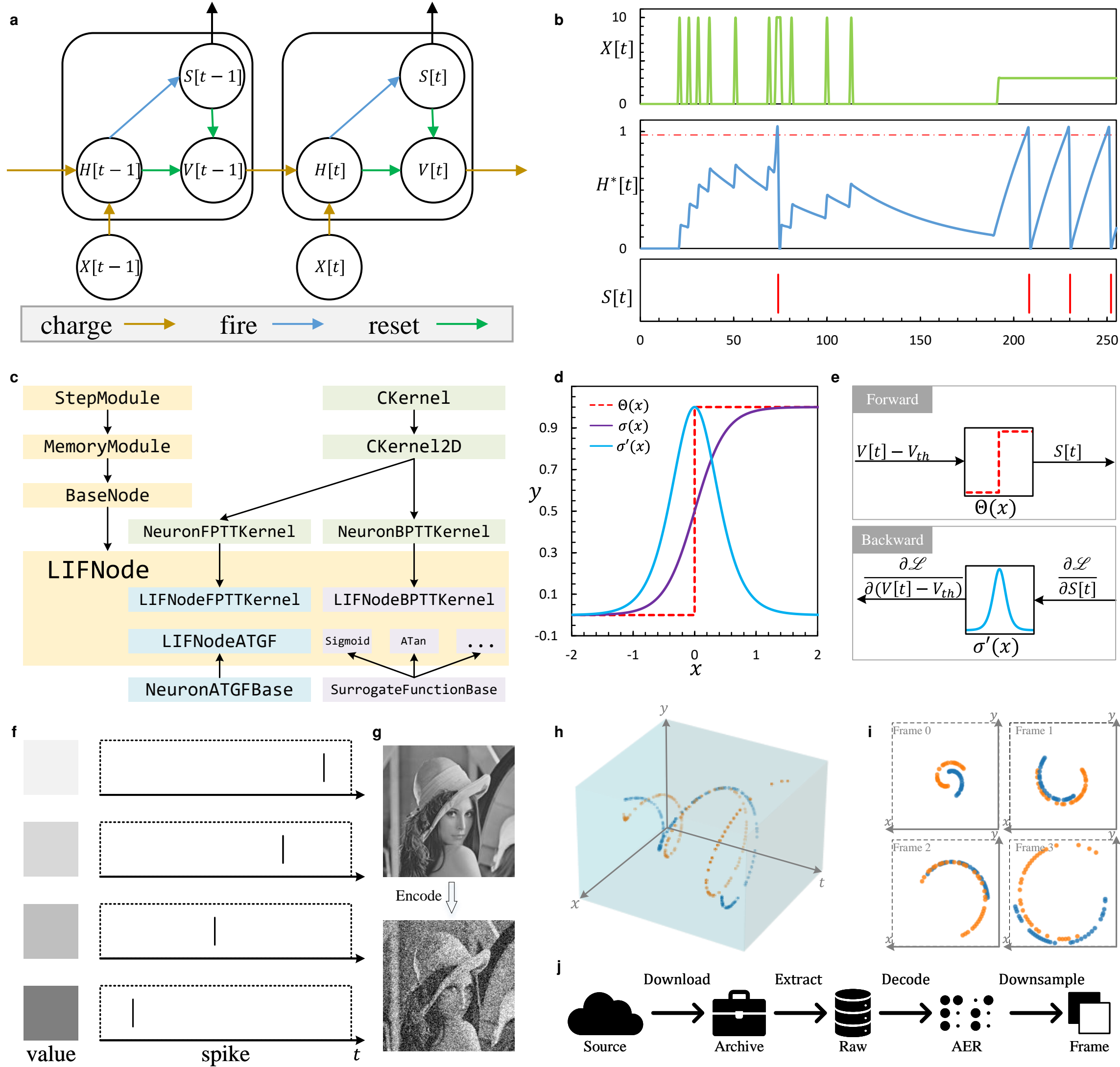

图 2: **SpikingJelly 中包含的组件模块。** **a.** 由方程 (5, 6, 7) 描述的通用离散时间脉冲神经元模型：神经元充电、放电和重置方程。**b.** LIF 神经元的模拟。输入 $X[t]$ 在 $0 \leq t < 128$ 时为 0 或 10，在 $128 \leq t < 196$ 时为 0，在 $196 \leq t < 256$ 时为 3。这里 $\{H^*[t]\}$ 记录了所有时间步的 $H[t]$，并包含 $S[t] = 1$ 时的 $V[t]$。**c.** LIF 神经元的继承关系。**d.** sigmoid 代理函数 $\sigma(x) = \frac{1}{1+\exp(-\alpha x)}$ 的示例，它是 Heaviside 函数 $\Theta(x)$ 的近似器。**e.** 代理梯度方法的原理：在前向传播期间使用 $\Theta(x)$ 生成离散二值脉冲，在反向传播期间使用 $\sigma'(x)$ 获得连续浮点梯度。**f.** 延迟编码器将较大的值（用较深的方块表示）编码为较早放电时间的脉冲。**g.** 输入图像和泊松编码器输出脉冲的图示。**h.** 具有两种极性的合成事件的可视化以及 **i.** 从这些事件下采样得到的帧。**j.** 数据集处理的工作流程。

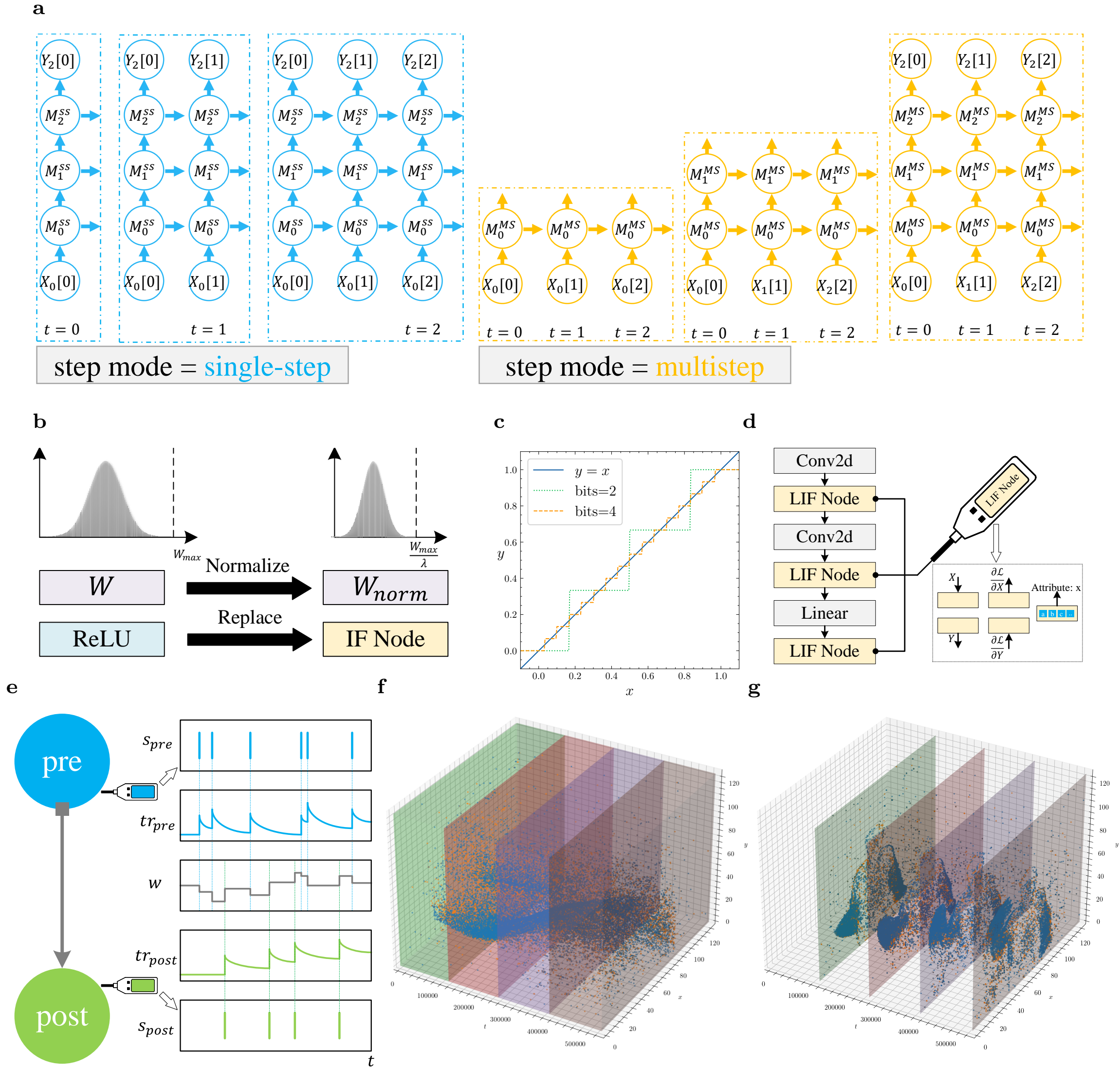

图 3: **SpikingJelly 中的功能模块。** **a.** 示意图展示了使用逐步或逐层传播模式模拟具有 3 层和 3 个时间步的网络。$X_i[t]$ 是第 i 层在时间步 t 的输入，$Y_i[t]$ 是第 i 层在时间步 t 的输出。M_i^{ss} 表示单步模式下的第 i 层，而 M_i^{ms} 表示多步模式下的第 i 层。两种传播模式的完整计算图相同，但以不同顺序构建。**b.** ANN2SNN 的功能：归一化权重并用积分放电（IF）神经元层替换整流线性单元（ReLU）。**c.** 量化器将范围 $[0,1]$ 内的输入映射到最近的定点数，定点数的数量由位数控制。**d.** 监控器像探针一样工作，记录指定模块的输入、输出、输入梯度、输出梯度和属性。**e.** STDP 学习器使用监控器记录突触前脉冲和突触后脉冲，计算迹，并更新权重。**f.** 示意图展示了将 DVS Gesture 数据集的事件切片为 4 个 bin 并整合为 4 帧的过程（如 **g** 所示）。

别遵循逐步或逐层传播模式，如图 3**a** 所示。这些传播模式的区别在于计算图的构建顺序。更具体地说，逐步传播模式是深度优先搜索（DFS），而逐层模式是广度优先搜索（BFS）。SpikingJelly 中的传播模式旨在满足用户的特定意图（参见补充材料中的 *Distinction of Propagation Patterns*）。逐步方法在构建循环连接方面更加灵活，适用于 ANN2SNN，因为其推理期间的内存消耗与时间步数 T 不成正比。逐层模式具有效率优势，通过在 SpikingJelly 中为无状态模块将时间步合并到批次维度、为有状态模块融合内核来优化。图 3**b** 展示了 ANN2SNN 转换功能，它归一化权重并用积分放电（IF）神经元层替换整流线性单元（ReLU）。图 3**c** 展示了 $k = 2, 4$ 的 k 位量化器，它将输入 $x \in (0, 1)$ 量化为最近的定点值 y。量化器支持量化感知训练，在训练期间量化权重。注意量化器的梯度几乎处处为零，因此采用代理梯度方法。图 3**d** 展示了使用监控器记录数据的示例。监控器像探针一样工作，从指定模块收集数据。可以记录输入、输出、输入梯度、输出梯度和属性，以满足主要的数据收集需求。图 3**e** 展示了基于迹方法 (*159*) 的 STDP 学习器，它使用两个监控器记录突触前/后脉冲。然后更新迹，并修改突触的权重。图 3**f**, **g** 可视化了事件切片和下采样操作。在图 3**f** 中，DVS Gesture 数据集的一个样本被切片为四个 bin，用四个不同颜色的长方体表示。然后累积每个长方体中的事件，在图 3**g** 中生成四帧。有关 SpikingJelly 中模块的更多详细信息，请参阅 材料与方法部分。

典型应用

作为一套面向脉冲深度学习的全栈工具包，SpikingJelly 几乎覆盖了 SNN 的主要应用场景。为便于说明，我们在图 4 中给出了三个典型示例。

图 4**a** 展示了使用深度 SNN 对神经形态 DVS Gesture 数据集进行分类的完整流程。首先，通过 SpikingJelly 的 *dataset* 子包将 DVS 相机采集的原始事件预处理为张量；随后，利用 *layer*、*neuron* 和 *exchange* 子包构建并训练深度卷积 SNN；训练完成后，再借助 *exchange* 子包将该 SNN 部署到神经形态 Loihi 芯片上执行推理。

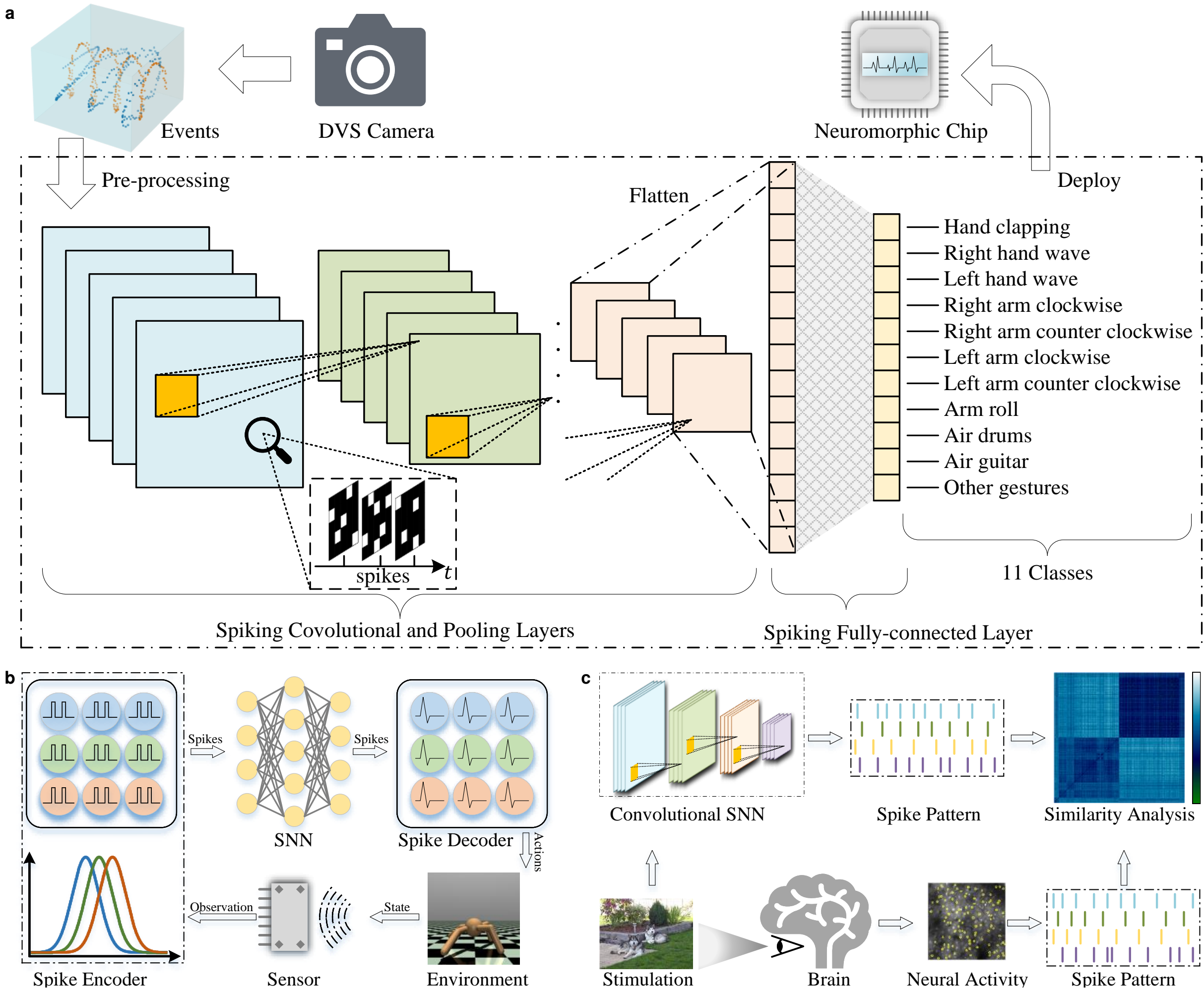

图 4: **SpikingJelly 的典型应用。** **a**. 一个全栈流程示例：将 DVS Gesture 数据集中的原始事件预处理为张量，构建并训练用于分类的深度 SNN，再将该 SNN 部署到神经形态 Loihi 芯片上进行推理。**b**. 强化学习与控制的流程。脉冲编码器将 OpenAI Gym 环境状态对应的观测值转换为脉冲序列，SNN 处理后，脉冲解码器再将其变换为表示动作的连续值。**c**. 衡量生物视觉系统与 SNN 之间神经相似性的流程。将相同的视觉刺激输入生物视觉系统和预训练 SNN，得到两个系统的神经表征，再通过表征相似性来评估模型的类脑程度。

图 4**b** 展示了使用深度 SNN 处理 OpenAI Gym (*160*) 连续控制任务的示例。环境状态对应的观测值首先通过由群体神经元组成的不同高斯感受野进行编码 (*39*)，从而将浮点值有效转换为脉冲序列；经过 SNN 处理后，再将离散脉冲转换为浮点值，本质上对应于后续非脉冲神经元层的膜电位，用来表示动作。网络中的各个模块均可通过 *layer* 和 *neuron* 子包实现。

图 4**c** 展示了利用深度 SNN 建模生物视觉皮层的示例。我们首先通过 *layer* 和 *neuron* 子包构建并训练用于 ImageNet 分类任务的深度 SNN；随后，为获得视觉皮层与模型的神经表征，将相同的视觉刺激同时输入生物视觉系统和预训练 SNN；最后，通过计算得到的表征相似性，对 SNN 建模视觉皮层的效果进行定量分析。上述三个应用的源代码与实验结果均见补充材料。

讨论

尽管 SNN 在生物合理性和功耗效率方面优于 ANN，但由于缺乏可用的学习方法，其应用长期局限于神经科学而非计算科学。随着深度学习方法的引入，SNN 的性能得到了极大提升，使脉冲深度学习成为新的研究热点。然而，这一新兴研究方向也面临两难：传统软件框架偏重神经科学而非深度学习，而面向这一新范式的框架又尚不充分。SpikingJelly 正是为满足脉冲深度学习快速增长的研究需求而提出的。

脉冲深度学习是一个新兴的交叉领域，研究者往往需要同时理解神经科学与深度学习。然而，专精于其中一个方向的研究人员，未必熟悉另一个方向。这一点也与我们在 GitHub 上回答用户问题、参与社区讨论时的经验一致。为降低学习和使用成本，SpikingJelly 提供了简洁易用的 API，并内置了经典模型和常用训练脚本。借助少量代码，用户即可快速构建多种类型的 SNN 并运行模型，即使并不熟悉底层实现，也能高效开展研究。

复杂而多样的脉冲神经元与突触是 SNN 的核心组件。无论是借鉴生物神经系统

(*56, 100, 143*)，还是吸收深度 ANN 的经验 (*60, 91*)，通过修改神经元和突触来改进脉冲深度学习，都是切实可行的路径。研究人员通常希望只通过修改少量函数或属性，就能定义新的脉冲模块，并显著改变模型的行为与性能。SpikingJelly 的灵活 API 正是为支持这种研究范式而设计。框架中的大多数模块都可通过继承父类、覆盖函数以及增删属性来构建，这也为研究人员开发新模块提供了直接而清晰的参考。

深度学习通常依赖大规模数据集与大模型 (*22*)，并常通过增加训练轮次进一步提升性能 (*161*)。这些特点同样适用于脉冲深度学习，而且由于额外的时间维度，深度 SNN 的计算复杂度往往高于深度 ANN。因此，深度 SNN 的仿真效率至关重要。尤其是近期在包含 128 万张图像的 ImageNet 数据集上评估 50 层以上深度 SNN 的研究，已成为广泛采用的性能基准 (*60, 88–91*)。SpikingJelly 在设计上高度重视计算效率：其仿真过程得益于 PyTorch 提供的 OpenMP/MKL CPU 加速以及 cuBLAS/CUDNN GPU 加速，同时还通过维度合并、半自动生成的 CUDA 内核以及即时编译（JIT）等方式实现了更高层次的融合加速，从而带来极高的训练与推理效率。借助这些加速手段，SpikingJelly 达到了当前领先的仿真速度，显著减少了研究人员在深度 SNN 长时间训练中的等待成本。

深度学习方法提升了 SNN 的性能，使其在实际任务中的应用变得切实可行 (*26*)。借助 SpikingJelly 提供的构建、训练与部署 SNN 的全栈方案，深度 SNN 的应用边界已从玩具数据集分类扩展到具有现实价值的任务，包括达到人类水平的分类、网络部署以及事件数据处理。除了经典机器学习任务外，深度 SNN 的一些前沿应用也已陆续出现，例如由可校准人工感觉神经元构成的基于脉冲的神经形态感知系统 (*162*)、运行在忆阻器上的神经形态计算模型 (*163*)，以及事件驱动 SNN 硬件加速器的设计 (*117*)。这些成果共同表明，SpikingJelly 的出现将进一步推动脉冲深度学习社区的发展。

材料与方法

在本节中，我们首先介绍 SpikingJelly 的关键模块及其设计理念。为便于说明，我们将模块分为两类：组件模块和功能模块。组件模块是 SNN 的基本构件，常用于定义网络；功能模块则用于模拟 SNN、加速其仿真过程、记录特定数据或修改变量。最后，我们讨论 SpikingJelly 中采用的加速方法。

组件模块

神经元模型。脉冲神经元是 SNN 的关键组件。神经科学家更倾向于使用具有复杂神经元动力学的脉冲神经元，例如 Hodgkin-Huxley 模型 (*164*) 和 Izhikevich 模型 (*165*)，以获得更高的生物合理性。在将深度学习应用于 SNN 时，简化的脉冲神经元模型（包括 IF 和 LIF 模型）更为常用，因为它们的动力学更直观、超参数更紧凑，从而降低了计算复杂度，也减少了研究人员调节训练过程的工作量。

一般而言，脉冲神经元的行为可以概括为三个过程：神经元充电、神经元放电和神经元重置。神经元充电也称为阈下动力学，可以由一个或多个常微分方程（ODE）描述。例如，LIF 神经元的充电方程可写为

$$\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{rest}) + X(t), \tag{1}$$

其中 τ_m 是膜时间常数，$V(t)$ 是时刻 t 的膜电位，V_{rest} 是静息电位，$X(t)$ 是输入电流。当脉冲神经元的膜电位 $V(t)$ 超过阈值 V_{th} 时，它会发放一个脉冲 $S(t) = 1$；若 $V(t) < V_{th}$，则保持静默（$S(t) = 0$），其形式可表示为

$$S(t) = \Theta(V(t) - V_{th}), \tag{2}$$

其中 $\Theta(x)$ 是 Heaviside 阶跃函数，定义为：当 $x \geq 0$ 时 $\Theta(x) = 1$，当 $x < 0$ 时 $\Theta(x) = 0$。如果神经元发放脉冲，它将被重置。重置通常分为两类：硬重置和软重置。硬重置将 $V(t)$ 直接重置为 V_{reset}，由于实验表现更好 (*166*)，通常用于直接训练的 SNN；软重置

则在神经元发放时减去 V_{th}，由于其对 ReLU 激活的理论拟合误差更低 (*50*)，常用于 ANN2SNN。神经元重置可写为

$$\lim_{\Delta t \to 0^+} V(t+\Delta t) = \begin{cases} V_{reset}，\text{硬重置} \\ V(t) - V_{th}，\text{软重置} \end{cases}. \tag{3}$$

为了模拟脉冲神经元，连续时间微分方程通常用离散时间差分方程来近似。在实践中，由于计算代价较低，最简单的一阶欧拉方法被广泛采用。例如，方程 (1) 的离散时间形式为

$$\tau_m(V[t] - V[t-1]) = -(V[t-1] - V_{rest}) + X[t]. \tag{4}$$

不同神经元的充电方程各不相同，而放电和重置方程通常可以共享。基于我们此前的研究 (*56*)，SpikingJelly 使用以下三个离散时间方程来描述脉冲神经元：

$$H[t] = f(V[t-1], X[t]), \tag{5}$$

$$S[t] = \Theta(H[t] - V_{th}), \tag{6}$$

$$V[t] = \begin{cases} H[t]\cdot(1 - S[t]) + V_{reset}\cdot S[t],\ \text{硬重置} \\ H[t] - V_{th}\cdot S[t],\ \text{软重置} \end{cases}. \tag{7}$$

为避免混淆，我们用 $H[t]$ 表示充电后但放电前的膜电位，用 $V[t]$ 表示重置后的膜电位。方程 (5) 是神经元充电方程，其中 f 随神经元类型而变化；方程 (6) 和方程 (7) 分别是神经元放电与重置方程。图 **2a** 展示了由这三类方程描述的通用离散时间神经元模型。

图 **2b** 展示了由 SpikingJelly 构建的 LIF 神经元模拟，参数为 $\tau_m = 50, V_{th} = 1, V_{reset} = 0$。输入 $X[t]$ 在 $0 \le t < 128$ 时为 0 或 10，在 $128 \le t < 196$ 时为 0，在 $196 \le t < 256$ 时为 3。这里 $\{H^*[t]\}$ 不仅记录了所有时间步的 $H[t]$，还包含神经元发放脉冲时（$S[t'] = 1$）的 $V[t']$，因此相比单独可视化 $H[t]$ 或 $V[t]$，它能更清楚地展示放电和重置行为。

图 **2c** 以 LIF 神经元为例，说明了 SpikingJelly 如何通过多层继承简化模块构建过程。*StepModule* 是一个可在单步/多步模式下工作的模块，它接收单个张量或序列输入，

是 SpikingJelly 中大多数模块的基类（参见补充材料中的 *Design of the Step Module*）。基于 *StepModule*，*MemoryModule* 通过添加隐藏状态这一额外属性，成为有状态模块（包括神经元与突触）的基类。进一步地，通用离散时间神经元 *BaseNode* 可以通过扩展 *MemoryModule* 轻松构建：它加入了分别服从方程 (6) 和方程 (7) 的放电与重置操作，并保留一个空的神经元充电函数，由子类补全以满足方程 (5)。最终，我们通过补全该充电方程，并加入额外膜电位参数 τ_m，构建出继承自 *BaseNode* 的 *LIFNode*。除了朴素的 PyTorch 实现外，SpikingJelly 还提供了可选的 CUDA 内核来加速脉冲神经元。LIF 神经元的前向 CUDA 内核 *LIFNodeFPTTKernel*、后向 CUDA 内核 *LIFNodeBPTTKernel*、自动求导函数 *LIFNodeATGF* 以及代理函数，同样借助多层继承扩展，仅需少量代码改动。这种模块设计范式增强了 SpikingJelly 的可扩展性，也降低了研究人员开发新模型的工作量。

代理梯度。深度学习模型之所以能取得巨大成功，很大程度上依赖于反向传播和梯度下降驱动的多层训练过程 (*44*)。然而，方程 (6) 中的 Heaviside 函数 $\Theta(x)$ 在 $x = 0$ 处导数为 $+\infty$，在 $x \neq 0$ 处导数为 0，这会阻断梯度传播。为了在 SNN 中应用基于梯度的深度学习方法，研究者提出了代理梯度方法 (*45, 46, 167, 168*)：在反向传播阶段，用代理函数 $\sigma(x)$ 的导数来替代 $\Theta(x)$ 的导数。常用代理函数通常是 $\Theta(x)$ 的平滑近似，一个典型例子是图 **2d** 所示的 sigmoid 代理函数 $\sigma(x) = \frac{1}{1+\exp(-\alpha x)}$，其中 α 是控制形状的超参数。

由于代理函数直接服务于脉冲触发机制，SpikingJelly 将其作为脉冲神经元的组成部分进行封装。如图 **2e** 所示，在前向传播期间，神经元使用 $S[t] = \Theta(V[t] - V_{th})$ 生成离散二值脉冲 $S[t] \in \{0, 1\}$；在反向传播期间，梯度通过 $\frac{\partial \mathcal{L}}{\partial (V[t]-V_{th})} = \frac{\partial \mathcal{L}}{\partial S[t]} \cdot \sigma'(V[t] - V_{th})$ 计算，其中 $\mathcal{L}$ 为损失函数。

编码器。脉冲编码器是 SNN 中一类独特模块，用于将非二值输入数据编码为脉冲。图 **2f**

展示了 SpikingJelly 中的延迟编码器，其形式可表示为

$$S[t] = \begin{cases} 1, t = t_f, \\ 0, t \neq t_f, \end{cases} \tag{8}$$

$$t_f = [(T_{max} - 1) \cdot (1 - x)], \tag{9}$$

其中 $S[t]$ 是时间步 t 的输出脉冲，t_f 是发放时间步，$[\cdot]$ 表示取整，T_{max} 是最晚发放时间步，x 是输入值。延迟编码器会将较大的 x 编码为更早的发放时间步 t_f，这与视网膜通路中“刺激越强、响应越快”的现象一致 (*169*)。

泊松编码器是另一种常用编码器，它将输入 $x \in [0, 1]$ 编码为泊松事件序列 $\{s[t]\}$，满足

$$P(\sum_{t=0}^{T} s[t] = k) = \Pr(X = k) = \frac{x^k \exp(-x)}{k!}. \tag{10}$$

为了简化实现，SpikingJelly 中采用了如下近似形式：

$$P(S[t] = 1) = x. \tag{11}$$

图 **2g** 给出了输入图像与泊松编码器输出脉冲的示例。可以看到，输出脉冲保留了原始图像的大部分信息。除泊松编码器外，SpikingJelly 还提供了加权相位编码器 (*170*) 和高斯调谐编码器 (*39*) 等常用编码器。

神经形态数据集。神经形态数据集常被用作评估 SNN 时序信息处理能力的基准 (*171*)。SpikingJelly 提供了多种常用数据集，包括 ASL-DVS (*172*)、CIFAR10-DVS (*173*)、DVS Gesture (*55*)、ES-ImageNet (*99*)、HARDVS (*174*)、N-Caltech101 (*98*)、N-MNIST (*98*)、Nav Gesture (*175*) 和 Spiking Heidelberg Digits（SHD）(*176*)（参见补充材料中的 *Neuromorphic Datasets*）。通过继承数据集基类并实现少量处理函数，也可以轻松集成新的数据集。

图 **2h** 可视化了一组具有两种极性的事件，分别以蓝色和绿色表示。从传感器采集的原始数据通常存储为特定格式，例如 AEDAT，因此需要专门的二进制解码方法。为降低使用门槛，SpikingJelly 实现了多种数据格式的解码器。相应地，SpikingJelly 中数据集的输出采用 NumPy 格式，与各类 Python 应用程序兼容。单个样本中的事件数量往往可达数百万，网络难以直接处理，因此常见做法是将事件整合为帧。图 **2i** 给出了从图 **2h** 所示事件整合得到的 4 帧。SpikingJelly 也内置了常用的整合方法。

图 **2j** 展示了 SpikingJelly 中数据集处理的整体流程，包括从源头下载、提取压缩包、将原始数据解码为 NumPy 格式的 AER Python 字典，并进一步将事件下采样为帧。若要添加新数据集，用户只需继承基类并实现 'download'、'extract' 和 'decode' 函数，其余处理工作由基类配合多线程机制完成，这也体现了 SpikingJelly 的良好可扩展性。

功能模块

步进模式与传播模式。在 SpikingJelly 中，每个模块都有一个可变属性 *step_mode*，用于控制该模块使用单步模式还是多步模式（参见补充材料中的 *Design of the Step Module*）。在单步模式下，模块接收 $X[t]$ 并输出 $Y[t]$，它们表示单个时间步的数据。第 0 层的输入序列记为 $X_0 = \{X_0[0], X_0[1], ..., X_0[T-1]\}$。当 SNN 由 L 个单步模块 $\{M_0^{ss}, M_1^{ss}, ..., M_{L-1}^{ss}\}$ 堆叠而成时，它以逐步传播模式运行，如算法 **1a** 所示。

当将 *step_mode* 切换到多步模式时，模块接收序列 $X = \{X[0], X[1], ..., X[T-1]\}$ 并输出序列 $Y = \{Y[0], Y[1], ..., Y[T-1]\}$。相应地，由 L 个多步模块 $\{M_0^{ms}, M_1^{ms}, ..., M_{L-1}^{ms}\}$ 构成的 SNN 可以采用逐层方式模拟，如算法 **1b** 所示。

图 **3a** 展示了逐步传播与逐层传播如何以不同顺序构建同一个计算图。考虑到 SNN 的计算图同时具有时间步和网络深度两个维度，可以将逐步传播看作对计算图的深度优先搜索（DFS），而逐层传播则对应广度优先搜索（BFS）。在 SpikingJelly 中，用户可通过修改各层的 *step_mode* 属性，在两种传播模式之间灵活切换。具体使用哪种模式，

Algorithm 1: 传播模式

(a) 传播模式：逐步

输入：由 L 个单步模块 $\{M_0^{ss}, M_1^{ss}, ..., M_{L-1}^{ss}\}$ 堆叠的 SNN，第 0 层输入序列 $X_0 = \{X_0[0], X_0[1], ..., X_0[T-1]\}$

创建空列表 $Y_{L-1} = \{\}$

for $t \leftarrow 0, 1, ...T-1$

  for $l \leftarrow 0, 1, ...L-1$

    $X_{l+1}[t] = Y_l[t] = M_l^{ss}(X_l[t])$

  将 $Y_{L-1}[t]$ 追加到 $Y_{L-1} = \{Y_{L-1}[0], Y_{L-1}[1], ..., Y_{L-1}[t-1]\}$

输出 $Y_{L-1} = \{Y_{L-1}[0], Y_{L-1}[1], ..., Y_{L-1}[T-1]\}$

(b) 传播模式：逐层

输入：由 L 个多步模块 $\{M_0^{ms}, M_1^{ms}, ..., M_{L-1}^{ms}\}$ 堆叠的 SNN，第 0 层输入序列 $X_0 = \{X_0[0], X_0[1], ..., X_0[T-1]\}$

for $l \leftarrow 0, 1, ...L-1$

  $X_{l+1} = Y_l = M_l^{ms}(X_l)$

输出 $Y_{L-1} = \{Y_{L-1}[0], Y_{L-1}[1], ..., Y_{L-1}[T-1]\}$

取决于用户需求。逐层模式在效率上更具优势，因为它允许无状态层在时间维度上并行计算，并允许有状态层执行融合操作；而当 $X[t+1]$ 依赖于 $X[t]$（例如层间存在循环连接）时，逐层模式就不再适用，此时逐步模式更灵活也更合适。

转换方法。除了使用代理梯度直接训练外，获得高性能 SNN 的另一条路径是 ANN 到 SNN 的转换（ANN2SNN）。该转换基于如下事实：当 ANN 使用 ReLU 激活和平均池化时，速率编码 SNN 中的脉冲发放率与 ANN 的激活值之间存在等价关系 (*49, 97*)。据此，可以将训练好的 ANN 转换为对应的 SNN。

为便于解决 ANN2SNN 问题，SpikingJelly 提供了 *Converter* 模块。它对 ANN 权重进行归一化，并通过用 IF 神经元层替换 ReLU，将 ANN 转换为 SNN，如图 3**b** 所示。IF 神经元的充电函数为

$$H[t] = V[t-1] + X[t], \tag{12}$$

该模型不含衰减项，因此可通过发放率近似 ANN 中的 ReLU。SpikingJelly 的实现基于突触后电位建模 (*51, 104, 105*)，从而使平均突触后电位与原 ANN 激活值相对应。我们提供了 *VoltageHook* 模块，用于记录隐藏特征的最大值（或某个百分位数），并将其作为转换后 SNN 的突触后电流尺度。随后，只需将激活层替换为 IF 神经元层即可完成转换。此外，为实现少时间步下的快速推理，SpikingJelly 还提供了即插即用的优化膜电位初始化方法 (*104*)。

量化器。在 FPGA、神经形态芯片、手机等资源受限硬件上执行 SNN 推理时，网络量化是降低存储和计算成本的关键技术。例如，将网络量化为 8 bit 后，相比 32 bit 模型，模型体积与内存带宽需求可下降 4×，吞吐量最高可提升 4×。SpikingJelly 提供了量化感知训练量化器，可在训练期间对权重和神经元动力学进行量化。图 3**c** 展示了 SpikingJelly 中典型的 k bit 量化器：它将范围 $[0,1]$ 内的输入 x 映射到最近的定点数

$y = q(x) = [\frac{(2^k-1)\cdot x}{2^k-1}]$，其中 $[\cdot]$ 表示取整。由于 $q'(x)$ 几乎处处为零，SpikingJelly 同样使用代理方法重新定义其梯度。

监控器。研究人员常常需要记录中间数据，例如脉冲神经元层的发放率。为满足这类需求，SpikingJelly 提供了 5 种通用监控器，用于监控前向/后向传播过程中的输入、输出以及特定层属性。如图 **3d** 所示，监控器就像一个探针，插入到用户指定类型的层中，并通过自定义变换 λ 记录数据。例如，若我们希望统计 SNN 中所有脉冲神经元层的发放率，可以将 *OutputMonitor* 的层类型设为脉冲神经元层，并将变换设为 $\lambda(S) = \frac{1}{T}\sum_{t=0}^{T-1} S[t]$，其中 T 表示时间步数，$S[t]$ 表示第 t 个时间步的脉冲。借助该监控器，我们便可轻松记录每一层脉冲神经元的发放率。

STDP 学习器。将局部无监督的 STDP 学习过程与全局监督的代理梯度结合，已经成为深度 SNN 中一种很有前景的学习方式 (*177*)，而这一过程可以通过 SpikingJelly 中的 STDP 学习器实现。在图 **3e** 所示的两个神经元最小示例中，SpikingJelly 的 STDP 学习器采用了基于迹方法的“全对全”STDP (*159*)，即考虑所有突触前与突触后脉冲对。它使用监控器分别记录突触前脉冲 s_{pre} 和突触后脉冲 s_{post}。随后，迹 $tr_{pre}[t], tr_{post}[t]$ 更新为

$$tr_{pre}[t] = tr_{pre}[t-1] - \frac{tr_{pre}[t-1]}{\tau_{pre}} + s_{pre}[t], \tag{13}$$

$$tr_{post}[t] = tr_{post}[t-1] - \frac{tr_{post}[t-1]}{\tau_{post}} + s_{post}[t], \tag{14}$$

其中 τ_{pre}, τ_{post} 分别表示突触前与突触后迹的时间常数。接着，权重 w 按照

$$\Delta w[t] = F_{post}(w[t]) \cdot tr_{pre}[t] \cdot s_{post}[t] - F_{pre}(w[t]) \cdot tr_{post}[t] \cdot s_{pre}[t], \tag{15}$$

进行更新。其中 F_{pre} 和 F_{post} 为用户自定义函数，用于控制突触变化幅度。在 SpikingJelly 中，Δw 可直接加到梯度 $\frac{\partial \mathcal{L}}{\partial w}$ 上，这意味着 STDP 学习器能够与梯度下降方法及各类优化器（包括 SGD 和 Adam (*17*)）协同工作。

事件下采样方法。事件到帧的下采样方法在深度 SNN 中被广泛使用，且具有多种切片与整合形式。为适应这种多样性，SpikingJelly 同时支持预定义与用户自定义的切片和整合方法。其中，一种常见方法是按固定时间长度 ΔT 对事件进行整合，其形式为

$$\mathbf{F}[j,p,y,x] = \sum_{t_0 + j\cdot\Delta T \le t_i < t_0 + (j+1)\cdot\Delta T} \mathcal{I}_{p,x,y}(p_i, y_i, x_i), \tag{16}$$

其中 $\mathcal{I}_{p,y,x}(p_i, y_i, x_i)$ 是指示函数，仅当 $(p,y,x) = (p_i, y_i, x_i)$ 时取 1。图 **3f** 可视化了 DVS Gesture 样本按固定时间长度切片得到的四个事件 bin，而对应的四个整合帧则如图 **3g** 所示。

然而，基于固定时间长度的整合会产生帧数不一致的问题，因为神经形态数据集中不同样本的持续时间并不相同。SpikingJelly 提供的另一种常用方法是基于固定帧数的整合 (*56*)，其形式为

$$\mathbf{F}[j,p,y,x] = \sum_{i=j_l}^{j_r - 1} \mathcal{I}_{p,x,y}(p_i, y_i, x_i), \tag{17}$$

其中 j_l 和 j_r 由具体分割方式生成。例如，若按事件数量分割，则有

$$j_l = \lfloor \frac{N}{T} \rfloor \cdot j, \tag{18}$$

$$j_r = \begin{cases} \lfloor \frac{N}{T} \rfloor \cdot (j+1), & j < T-1, \\ N, & j = T-1, \end{cases} \tag{19}$$

其中 $\lfloor\cdot\rfloor$ 表示向下取整，N 为事件总数。若按事件持续时间分割，则有

$$\Delta T = \left\lceil \frac{t_{N-1} - t_0}{T} \right\rceil, \tag{20}$$

$$j_l = \arg\min_k \{t_k \mid t_k \ge t_0 + \Delta T \cdot j\}, \tag{21}$$

$$j_r = \begin{cases} \arg\max_k \{t_k \mid t_k < t_0 + \Delta T \cdot (j+1)\} + 1, & j < T-1 \\ N, & j = T-1 \end{cases}. \tag{22}$$

需要注意的是，基于固定帧数的整合只能在所有事件都到达后进行，因此并不适用于事件持续时间不可预测的实时系统。

SpikingJelly 同样支持用户自定义的切片与整合方法；它们以可调用函数的形式实现，并作为数据集类初始化参数的一部分使用。

加速模块

加速原理。 SpikingJelly 通过两种方式加速 SNN 仿真：其一是对无状态层顺序计算进行并行化，其二是对有状态层执行 CUDA 内核融合。

无状态层（如卷积层与线性层）在 SNN 中被广泛使用。与有状态层类似，这些层同样接收形状为 $(T, N, ...)$ 的序列 $X = \{X[0], X[1], ..., X[T-1]\}$ 作为输入。它们与有状态层的区别在于：无状态层中 $Y[t]$ 的计算仅依赖于 $X[t]$，因此不存在时间依赖，沿时间步的 for 循环也就不是必须的。SpikingJelly 提供了 *SeqToANNContainer* 及其函数式形式 *seq_to_ann_forward*，用于包装无状态层及其前向传播。*SeqToANNContainer* 会在将输入序列送入无状态层之前，把时间步维度合并到批次维度中，将输入从 $(T, N, ...)$ 重塑为 $(T \cdot N, ...)$，从而实现所有时间步上的并行计算，最后再将输出拆分回长度不变的序列 $Y = \{Y[0], Y[1], ..., Y[T-1]\}$。图 5**a** 给出了用 *SeqToANNContainer* 包装一维卷积层 *Conv1d* 的示例，其中输入形状为 (N, C, L)，C 为通道数，L 为输入长度。为了避免沿时间步执行 for 循环，另一种可行方案是采用 $(n+1)$ 维卷积实现 n 维卷积，这一方法也已在部分框架中使用。该方法将时间步维度视为最后一个维度，因此输入序列形状为 $(N, C, ..., T)$；随后，对 $(n+1)$ 维输入施加的 n 维卷积，可通过时间维度上权重和步长均为 1 的 $(n+1)$ 维卷积实现。图 5**b** 比较了不同实现方式在形状为 $(T, N, C, H, W) = (T, 16, 128, 64, 64)$ 的输入上施加一个具有 128 个通道、卷积核大小为 3、步长为 1 的二维卷积时的执行时间。在图 5**b** 中，“SJ” 表示 SpikingJelly 的“合并批次和时间步维度”方法，“RAW” 表示沿时间步执行的朴素 for 循环，“3D” 表示将时间步维度作为 3D 卷积深度维度的实现。结果显示，“SJ” 的执行时间随时间步增长得最

慢，因此整体性能最佳。

对于有状态层（例如脉冲神经元），$Y[t]$ 不仅依赖于 $X[t]$，还依赖于隐藏状态 $H[t-1]$，而 $H[t-1]$ 又递归依赖于 $X[t-1], X[t-2], ...X[0]$。因此，沿时间步的 for 循环无法避免。然而，当使用 PyTorch 实现朴素 for 循环时，每个时间步都会触发一个或一系列 CUDA 内核调用。频繁启动大量小型 CUDA 内核会带来显著的调用开销（包括内存访问时间和内核启动时间），从而降低网络计算效率。为加速此类有状态层，SpikingJelly 将许多小内核融合为少量大内核，从而减少调用开销。图 **5c** 对比了采用 CUDA 内核融合实现的 LIF 神经元在多步模式下训练和推理时（“SJ”）与 PyTorch 朴素 for 循环实现（“RAW”）的执行时间。结果表明，“SJ” 明显快于 “RAW”。进一步可以看到，随着 T 增大，两种实现的推理时间都近似线性增长；但在训练阶段，“RAW” 的执行时间呈二次增长，而 “SJ” 的执行时间仍仅缓慢增长。有关各类加速方法效果的更多细节，请参见补充材料中的 *Ablation Study of Acceleration Methods*。

基于上述两种加速方式，我们建议在训练深度 SNN 时尽量使用逐层传播模式以获得更高效率。如算法 **1b** 所示，每个模块接收序列数据后，无状态层可以执行顺序计算并行化，有状态层则可以执行 CUDA 内核融合。然而，逐层传播模式的空间复杂度为 $\mathcal{O}(T \cdot N)$，即使在推理阶段也是如此。当 T 很大时，例如在 ANN2SNN 推理中需要数百甚至数千个时间步时，逐层传播模式往往会因内存受限而无法使用。此时，由于逐步传播模式的空间复杂度为 $\mathcal{O}(N)$，不随 T 成正比增长，因此更适合作为推理模式。

半自动 CUDA 代码生成。将多个操作融合到单个 CUDA 内核中，往往能获得最高计算效率，但同时也会带来巨大的开发工作量。标准做法通常包括编写 CUDA 代码、在 C++ 接口中封装 CUDA 函数，并通过 pybind 将 C++ 绑定到 Python。为简化这一过程，SpikingJelly 使用 CuPy (*178*) 来实现 CUDA 内核。开发者只需在 Python 字符串中定义原始 CUDA 内核，CuPy 就会对其进行封装与编译，生成可缓存并在后续运行中复用的 CUDA 二进制库。引入 CuPy 后，开发者可以省去 CUDA、C++ 与 Python 之

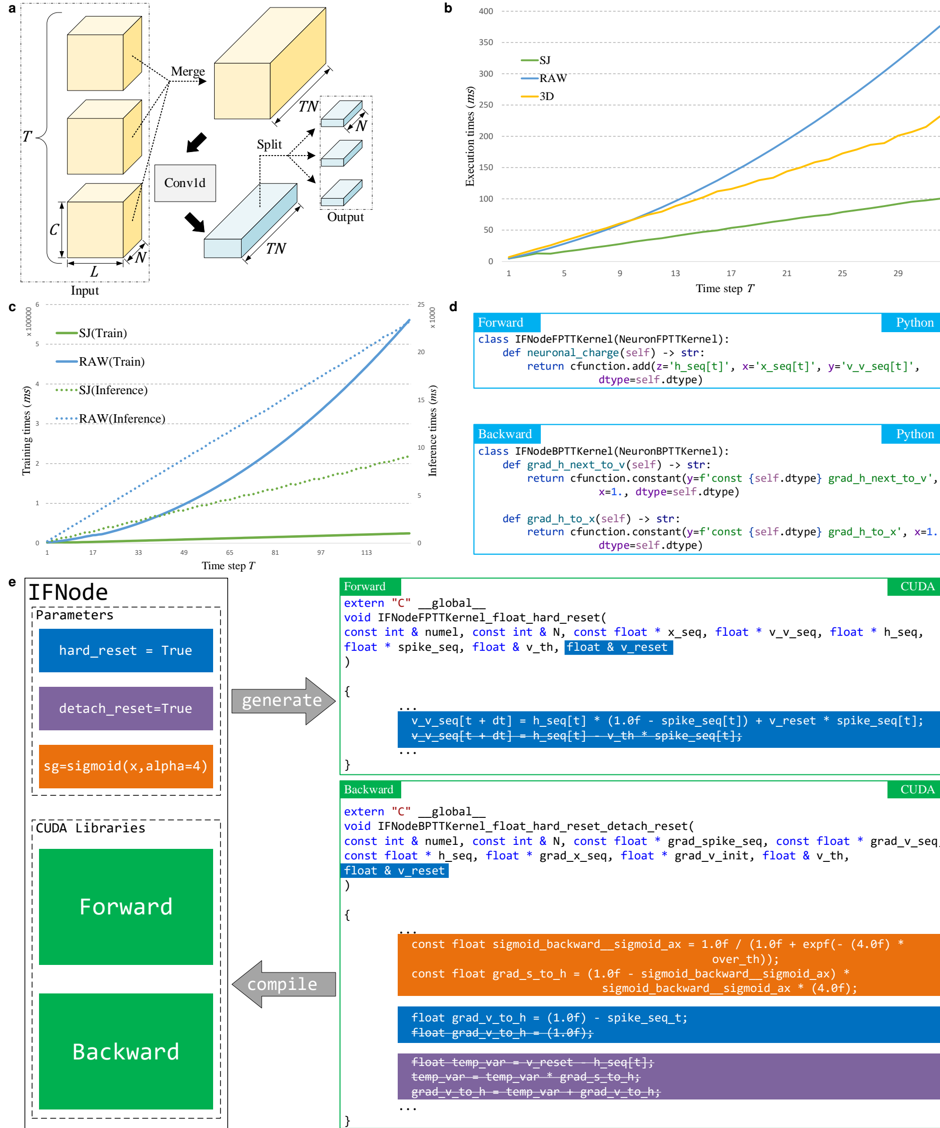

图 5: **SpikingJelly 中的加速。** **a**. 使用 *seq_to_ann_forward* 包装无状态层，并将时间步维度并入批次维度，从而在时间维度上并行计算。**b**. 对顺序数据应用二维卷积时不同实现方式的执行时间比较。“SJ” 表示 SpikingJelly 合并批次与时间步维度的方法，“RAW” 表示沿时间步执行的朴素 for 循环，“3D” 表示将时间步维度视为 3D 卷积深度维度的方法。**c**. 采用融合操作加速的 LIF 神经元实现（“SJ”）与 PyTorch 朴素 for 循环实现（“RAW”）的执行时间比较。**d**. 通过继承轻松实现 IF 神经元前向与后向内核的示例。**e**. IF 神经元示例，展示了 SpikingJelly 中 CUDA 神经元的完整工作流程。神经元中的参数以不同颜色标记，相应受这些参数控制的 CUDA 代码也使用相同颜色高亮显示。删除线如 ~~*abc*~~ 表示删除代码 *abc*。

间繁琐的绑定工作，更专注于纯 Python 开发环境。

使用 CuPy 的一个显著优势在于：CUDA 代码本身以 Python 字符串形式编写，因此我们可以在 Python 环境中灵活控制 CUDA 代码。结合 SpikingJelly 优秀的可扩展性，这一特性显著简化了 CUDA 内核的创建过程。如图 **2c** 所述，某个具体神经元类的前向内核和后向内核，都可以在神经元内核基类的基础上扩展得到，所需改动相对较少（参见补充材料中的 *Details of the Neuron Kernel*）。根据方程 (5)、方程 (6) 和方程 (7) 所描述的前向传播，反向传播可写为

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t]} = \frac{\partial \mathcal{L}}{\partial S[t]}\frac{\mathrm{d}S[t]}{\mathrm{d}H[t]} + \left(\frac{\partial \mathcal{L}}{\partial V[t]} + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t+1]}\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}\right)\frac{\mathrm{d}V[t]}{\mathrm{d}H[t]}, \tag{23}$$

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}X[t]} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t]}\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}, \tag{24}$$

其中 $\frac{\mathrm{d}S[t]}{\mathrm{d}H[t]}$ 由代理函数定义，$\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}$ 与 $\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}$ 由方程 (5) 给出，其余部分则对不同类型神经元具有通用性。因此，SpikingJelly 中的神经元内核基类本质上是一个“不完整”的 CUDA 内核，其中预留了供子类补全的空函数，包括用于前向传播的方程 (5)，以及用于反向传播的 $\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}$ 和 $\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}$。例如，当我们使用 IF 神经元时，其充电函数及对应的反向传播形式为

$$H[t] = V[t-1] + X[t], \tag{25}$$

$$\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]} = 1, \tag{26}$$

$$\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]} = 1, \tag{27}$$

于是，前向传播时间（FPTT）内核 *IFNodeFPTTKernel* 和反向传播时间（BPTT）内核 *IFNodeBPTTKernel* 的定义如图 **5d** 所示。FPTT 内核在单个 CUDA 内核中实现 IF 神经元跨 T 个时间步的动力学，包括方程 (12)、(6) 和 (7)；相应地，BPTT 内核则在单个 CUDA 内核中实现所有时间步上的反向传播。我们发现，补全这些空函数通常只需要几行代码。

图 5**e** 以 IF 神经元为例，展示了 SpikingJelly 中 CUDA 神经元的完整工作流程。神经元中的参数以不同颜色标记，由这些参数控制的 CUDA 代码也使用相同颜色高亮显示。例如，当 *hard_reset* 为真时，Python 类会执行以下操作，如图 5**e** 中蓝色高亮所示：

- 在前向与后向内核的参数中加入 *float & v_reset*；
- 在前向内核中保留硬重置命令 *v_v_seq[t + dt] = h_seq[t] * (1.0f - spike_seq[t]) + v_reset * spike_seq[t]*，并删除软重置命令 *v_v_seq[t + dt] = h_seq[t] - v_th * spike_seq[t]*；
- 在后向内核中保留后向硬重置命令 *float grad_v_to_h = (1.0f) - spike_seq_t*，并删除后向软重置命令 *float grad_v_to_h = (1.0f)*。

当 *detach_reset* 为真时，方程 (7) 中的 $S[t]$ 会从反向计算中分离 (*179*)，随后对应的 CUDA 代码也会被删除，如图 5**e** 中紫色高亮所示。用于定义 $\frac{\mathrm{d}S[t]}{\mathrm{d}H[t]}$ 的代理学习 CUDA 代码，则由代理函数类自动生成，如图 5**e** 中橙色高亮所示。完成 CUDA 代码生成后，CuPy 会将这些代码编译为 CUDA 库，于是前向和反向计算便能够通过大 CUDA 内核中的融合操作获得加速。

JIT 加速。 JIT 是另一种网络加速方法，它可以在运行时将 Python 代码编译成高性能的 C++/CUDA 程序。一些简单的逐元素操作，例如神经元放电（方程 (6)）和神经元重置（方程 (7)），可以借助 JIT 封装进单个 CUDA 内核。作为一种常见优化手段，JIT 的性能通常低于专门设计的优化方法，但它的开发成本也更低。因此，SpikingJelly 会对复杂操作（如带梯度的脉冲神经元跨时间步前向/后向传播内核）采用半自动生成的 CUDA 内核，而对其他简单函数（如单个时间步的神经元动力学）采用 JIT 加速。图 5**c** 中就给出了一个典型例子：其中 “SJ” 的推理过程就是通过 JIT 获得加速的。

参考文献

1. A. Krizhevsky, I. Sutskever, G. E. Hinton, *Advances in Neural Information Processing Systems (NeurIPS)* (2012), pp. 1097–1105.

2. S. Hochreiter, J. Schmidhuber, Long short-term memory. *Neural Computation* **9**, 1735–1780 (1997).

3. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)* **30** (2017).

4. K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

5. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 1–9.

6. R. Girshick, J. Donahue, T. Darrell, J. Malik, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2014), pp. 580–587.

7. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A. C. Berg, *European Conference on Computer Vision (ECCV)* (Springer, 2016), pp. 21–37.

8. J. Redmon, S. Divvala, R. Girshick, A. Farhadi, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 779–788.

9. I. Sutskever, O. Vinyals, Q. V. Le, Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* **27** (2014).

10. D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

11. R. Sennrich, B. Haddow, A. Birch, Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

12. A. Graves, A.-r. Mohamed, G. Hinton, *IEEE International Conference on Acoustics, Speech and Signal Processing* (Ieee, 2013), pp. 6645–6649.

13. A. Graves, N. Jaitly, A.-r. Mohamed, *IEEE workshop on Automatic Speech Recognition and Understanding* (IEEE, 2013), pp. 273–278.

14. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning. *Nature* **518**, 529-533 (2015).

15. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).

16. D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors. *Nature* **323**, 533–536 (1986).

17. D. P. Kingma, J. Ba, Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

18. X.-W. Chen, X. Lin, Big data deep learning: challenges and perspectives. *IEEE Access* **2**, 514–525 (2014).

19. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei, Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* **115**, 211–252 (2015).

20. R. Raina, A. Madhavan, A. Y. Ng, *International Conference on Machine Learning (ICML)* (2009), pp. 873–880.

21. A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks. *Communications of the ACM* **60**, 84–90 (2017).

22. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016). `http://www.deeplearningbook.org`.

23. B. Goertzel, Artificial general intelligence: concept, state of the art, and future prospects. *Journal of Artificial General Intelligence* **5**, 1 (2014).

24. D. Hassabis, D. Kumaran, C. Summerfield, M. Botvinick, Neuroscience-inspired artificial intelligence. *Neuron* **95**, 245–258 (2017).

25. W. Maass, Networks of spiking neurons: the third generation of neural network models. *Neural Networks* **10**, 1659–1671 (1997).

26. A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, A. Maida, Deep learning in spiking neural networks. *Neural Networks* **111**, 47–63 (2019).

27. D. Salaj, A. Subramoney, C. Kraisnikovic, G. Bellec, R. Legenstein, W. Maass, Spike frequency adaptation supports network computations on temporally dispersed information. *eLife* **10**, e65459 (2021).

28. D. D. Cox, T. Dean, Neural networks and neuroscience-inspired computer vision. *Current Biology* **24**, R921–R929 (2014).

29. P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, D. S. Modha, A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* **345**, 668–673 (2014).

30. M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, H. Wang, Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* **38**, 82–99 (2018).

31. J. Pei, L. Deng, S. Song, M. Zhao, Y. Zhang, S. Wu, G. Wang, Z. Zou, Z. Wu, W. He, F. Chen, N. Deng, S. Wu, Y. Wang, Y. Wu, Z. Yang, C. Ma, G. Li, W. Han, H. Li, H. Wu, R. Zhao, Y. Xie, L. Shi, Towards artificial general intelligence with hybrid tianjic chip architecture. *Nature* **572**, 106–111 (2019).

32. D. O. Hebb, *The organization of behavior: A neuropsychological theory* (Psychology Press, 2005).

33. G.-q. Bi, M.-m. Poo, Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience* **18**, 10464–10472 (1998).

34. T. Masquelier, S. J. Thorpe, Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS Comput Biol* **3**, e31 (2007).

35. S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, T. Masquelier, STDP-based spiking deep convolutional neural networks for object recognition. *Neural Networks* **99**, 56–67 (2018).

36. Q. Liu, H. Ruan, D. Xing, H. Tang, G. Pan, *Proceedings of the AAAI conference on artificial intelligence* (2020), vol. 34, pp. 1308–1315.

37. R. Xiao, H. Tang, Y. Ma, R. Yan, G. Orchard, An event-driven categorization model for aer image sensors using multispike encoding and learning. *IEEE Transactions on Neural Networks and Learning Systems* **31**, 3649-3657 (2020).

38. A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. P. Maguire, T. M. McGinnity, A review of learning in biologically plausible spiking neural networks. *Neural Networks* **122**, 253–272 (2020).

39. S. M. Bohte, J. N. Kok, H. La Poutre, Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing* **48**, 17–37 (2002).

40. R. Gütig, H. Sompolinsky, The tempotron: a neuron that learns spike timing–based decisions. *Nature Neuroscience* **9**, 420–428 (2006).

41. F. Ponulak, A. Kasiński, Supervised learning in spiking neural networks with ReSuMe: sequence learning, classification, and spike shifting. *Neural Computation* **22**, 467–510 (2010).

42. A. Mohemmed, S. Schliebs, S. Matsuda, N. Kasabov, Span: Spike pattern association neuron for learning spatio-temporal spike patterns. *International Journal of Neural Systems* **22**, 1250012 (2012).

43. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**, 2278–2324 (1998).

44. Y. LeCun, Y. Bengio, G. Hinton, Deep learning. *Nature* **521**, 436–444 (2015).

45. Y. Wu, L. Deng, G. Li, J. Zhu, L. Shi, Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience* **12** (2018).

46. S. B. Shrestha, G. Orchard, *Advances in Neural Information Processing Systems (NeurIPS)* (2018), pp. 1419–1428.

47. E. O. Neftci, H. Mostafa, F. Zenke, Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine* **36**, 51–63 (2019).

48. E. Hunsberger, C. Eliasmith, Spiking deep networks with lif neurons. *arXiv preprint arXiv:1510.08829* (2015).

49. Y. Cao, Y. Chen, D. Khosla, Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision* **113**, 54–66 (2015).

50. B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, S.-C. Liu, Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience* **11**, 682 (2017).

51. S. Deng, S. Gu, *International Conference on Learning Representations (ICLR)* (2021).

52. Y. Li, S. Deng, X. Dong, R. Gong, S. Gu, *International Conference on Machine Learning (ICML)* (PMLR, 2021), pp. 6316–6325.

53. C. Stöckl, W. Maass, Optimized spiking neurons can classify images with high accuracy through temporal coding with two spikes. *Nature Machine Intelligence* **3**, 230–238 (2021).

54. A. Krizhevsky, G. Hinton, Learning multiple layers of features from tiny images, *Tech. rep.* (2009).

55. A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, D. Modha, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

56. W. Fang, Z. Yu, Y. Chen, T. Masquelier, T. Huang, Y. Tian, *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 2661–2671.

57. Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, L. Shi, Direct training for spiking neural networks: Faster, larger, better. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* **33**, 1311–1318 (2019).

58. A. Sengupta, Y. Ye, R. Wang, C. Liu, K. Roy, Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience* **13**, 95 (2019).

59. H. Zheng, Y. Wu, L. Deng, Y. Hu, G. Li, *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (2021), vol. 35, pp. 11062–11070.

60. W. Fang, Z. Yu, Y. Chen, T. Huang, T. Masquelier, Y. Tian, Deep residual learning in spiking neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* **34** (2021).

61. Z. Zhou, Y. Zhu, C. He, Y. Wang, S. YAN, Y. Tian, L. Yuan, *International Conference on Learning Representations (ICLR)* (2023).

62. B. Han, G. Srinivasan, K. Roy, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 13558–13567.

63. B. Han, K. Roy, *European Conference on Computer Vision (ECCV)* (2020), pp. 388–404.

64. S. Kim, S. Park, B. Na, S. Yoon, *Proceedings of the AAAI conference on artificial intelligenc (AAAI)* (2020), vol. 34, pp. 11270–11277.

65. L. Cordone, B. Miramond, P. Thierion, Object detection with spiking neural networks on automotive event data. *arXiv preprint arXiv:2205.04339* (2022).

66. S. Barchid, J. Mennesson, J. Eshraghian, C. Djéraba, M. Bennamoun, Spiking neural networks for frame-based and event-based single object localization. *arXiv preprint arXiv:2206.06506* (2022).

67. K. Patel, E. Hunsberger, S. Batir, C. Eliasmith, A spiking neural network for image segmentation. *arXiv preprint arXiv:2106.08921* (2021).

68. C. M. Parameshwara, S. Li, C. Fermüller, N. J. Sanket, M. S. Evanusa, Y. Aloimonos, *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (IEEE, 2021), pp. 3414–3420.

69. U. Rançon, J. Cuadrado-Anibarro, B. R. Cottereau, T. Masquelier, Stereospike: Depth learning with a spiking neural network. *arXiv preprint arXiv:2109.13751* (2021).

70. C. Lee, A. K. Kosta, A. Z. Zhu, K. Chaney, K. Daniilidis, K. Roy, *European Conference on Computer Vision (ECCV)* (Springer, 2020), pp. 366–382.

71. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

72. F. Chollet, *et al.*, Keras, `https://keras.io` (2015).

73. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, *Advances in Neural Information Processing Systems (NeurIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett, eds. (Curran Associates, Inc., 2019), pp. 8024–8035.

74. H. Ben Braiek, F. Khomh, B. Adams, *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (2018), pp. 353–363.

75. N. T. Carnevale, M. L. Hines, *The NEURON book* (Cambridge University Press, 2006).

76. M.-O. Gewaltig, M. Diesmann, Nest (neural simulation tool). *Scholarpedia* **2**, 1430 (2007).

77. D. Goodman, R. Brette, Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics* **2** (2008).

78. M. Stimberg, R. Brette, D. F. Goodman, Brian 2, an intuitive and efficient neural simulator. *eLife* **8**, e47314 (2019).

79. H. Cornelis, A. L. Rodriguez, A. D. Coop, J. M. Bower, Python as a federation tool for genesis 3.0. *PLOS One* **7**, e29018 (2012).

80. T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. Voelker, C. Eliasmith, Nengo: a python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics* **7**, 48 (2014).

81. M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, T. Masquelier, Spyketorch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron. *Frontiers in Neuroscience* **13** (2019).

82. H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, R. Kozma, Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics* **12** (2018).

83. P. Lichtsteiner, C. Posch, T. Delbruck, A 128×128 120 db 15μs latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits* **43**, 566–576 (2008).

84. C. Posch, D. Matolin, R. Wohlgenannt, A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds. *IEEE Journal of Solid-State Circuits* **46**, 259–275 (2010).

85. C. Brandli, R. Berner, M. Yang, S.-C. Liu, T. Delbruck, A 240× 180 130 db 3 μs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits* **49**, 2333–2341 (2014).

86. C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with numpy. *Nature* **585**, 357–362 (2020).

87. K. He, X. Zhang, S. Ren, J. Sun, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

88. A. Sengupta, Y. Ye, R. Wang, C. Liu, K. Roy, Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience* **13** (2019).

89. B. Han, G. Srinivasan, K. Roy, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 13558–13567.

90. Y. Li, S. Deng, X. Dong, R. Gong, S. Gu, *International Conference on Machine Learning (ICML)* (2021), vol. 139, pp. 6316–6325.

91. Y. Hu, H. Tang, Y. Wang, G. Pan, Spiking deep residual network. *arXiv preprint arXiv:1805.01352* (2018).

92. C. Pehle, J. E. Pedersen, Norse - A deep learning library for spiking neural networks (2021). Documentation: https://norse.ai/docs/.

93. J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, W. D. Lu, Training spiking neural networks using lessons from deep learning. *arXiv preprint arXiv:2109.12894* (2021).

94. W. Gerstner, W. M. Kistler, R. Naud, L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition* (Cambridge University Press, 2014).

95. D. Lew, K. Lee, J. Park, *Proceedings of the 59th ACM/IEEE Design Automation Conference* (2022), pp. 265–270.

96. N. Rathi, K. Roy, Diet-snn: A low-latency spiking neural network with direct input encoding and leakage and threshold optimization. *IEEE Transactions on Neural Networks and Learning Systems* pp. 1–9 (2021).

97. J. Ding, Z. Yu, Y. Tian, T. Huang, Optimal ann-snn conversion for fast and accurate inference in deep spiking neural networks. *arXiv preprint arXiv:2105.11654* (2021).

98. G. Orchard, A. Jayawant, G. K. Cohen, N. Thakor, Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in Neuroscience* **9** (2015).

99. Y. Lin, W. Ding, S. Qiang, L. Deng, G. Li, Es-imagenet: A million event-stream classification dataset for spiking neural networks. *Frontiers in Neuroscience* **15** (2021).

100. H. Fang, A. Shrestha, Z. Zhao, Q. Qiu, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, C. Bessiere, ed. (International

Joint Conferences on Artificial Intelligence Organization, 2020), pp. 2799–2806. Main track.

101. J. Kaiser, H. Mostafa, E. Neftci, Synaptic plasticity dynamics for deep continuous local learning (decolle). *Frontiers in Neuroscience* **14** (2020).

102. G. Abad, O. Ersoy, S. Picek, V. J. Ramírez-Durán, A. Urbieta, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 3315–3317.

103. G. Abad, O. Ersoy, S. Picek, A. Urbieta, Sneaky spikes: Uncovering stealthy backdoor attacks in spiking neural networks with neuromorphic data. *arXiv preprint arXiv:2302.06279* (2023).

104. T. Bu, W. Fang, J. Ding, P. Dai, Z. Yu, T. Huang, *International Conference on Learning Representations (ICLR)* (2021).

105. T. Bu, J. Ding, Z. Yu, T. Huang, Optimized potential initialization for low-latency spiking neural networks. *arXiv preprint arXiv:2202.01440* (2022).

106. J. Tang, J. Lai, X. Xie, L. Yang, W.-S. Zheng, Snn2ann: A fast and memory-efficient training framework for spiking neural networks. *arXiv preprint arXiv:2206.09449* (2022).

107. Z. Hao, T. Bu, J. Ding, T. Huang, Z. Yu, Reducing ann-snn conversion error through residual membrane potential. *arXiv preprint arXiv:2302.02091* (2023).

108. Z. Hao, J. Ding, T. Bu, T. Huang, Z. Yu, *International Conference on Learning Representations (ICLR)* (2023).

109. R.-J. Zhu, Q. Zhao, T. Zhang, H. Deng, Y. Duan, M. Zhang, L.-J. Deng, Tcja-snn: Temporal-channel joint attention for spiking neural networks. *arXiv preprint arXiv:2206.10177* (2022).

110. M. Yao, G. Zhao, H. Zhang, Y. Hu, L. Deng, Y. Tian, B. Xu, G. Li, Attention spiking neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* pp. 1–18 (2023).

111. X. Wu, W. He, M. Yao, Z. Zhang, Y. Wang, G. Li, Mss-depthnet: Depth prediction with multi-step spiking neural network. *arXiv preprint arXiv:2211.12156* (2022).

112. B. Gong, C. Wei, H. Yang, Z. Yu, L. Wang, L. Xiong, R. Xiong, Z. Lu, Y. Zhang, Q. Liu, Control and regulation of skyrmionic topological charge in a novel synthetic antiferromagnetic nanostructure. *Nanoscale* (2023).

113. B. Wang, G. Dong, Y. Zhao, R. Li, H. Yang, W. Yin, L. Liang, Spiking emotions: Dynamic vision emotion recognition using spiking neural networks. *International Conference on Algorithms, High Performance Computing and Artificial Intelligence* (2022).

114. E. Lemaire, L. Cordone, A. Castagnetti, P.-E. Novac, J. Courtois, B. Miramond, An analytical estimation of spiking neural networks energy efficiency. *arXiv preprint arXiv:2210.13107* (2022).

115. L. Zhu, X. Wang, Y. Chang, J. Li, T. Huang, Y. Tian, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 3594–3604.

116. Z. Xu, Y. Ma, Z. Pan, X. Zheng, Deep spiking residual shrinkage network for bearing fault diagnosis. *IEEE Transactions on Cybernetics* pp. 1–6 (2022).

117. S. Li, L. Gong, T. Wang, C. Wang, X. Zhou, *12th International Symposium on Parallel Architectures, Algorithms and Programming* (IEEE, 2021), pp. 14–18.

118. M. A.-A. Kaiser, G. Datta, Z. Wang, A. P. Jacob, P. A. Beerel, A. R. Jaiswal, Neuromorphic-p2m: Processing-in-pixel-in-memory paradigm for neuromorphic image sensors. *arXiv preprint arXiv:2301.09111* (2023).

119. J. Fu, S. Gou, Z. Guo, *Intelligence Science IV*, Z. Shi, Y. Jin, X. Zhang, eds. (Springer International Publishing, Cham, 2022), pp. 37–44.

120. Y. Han, T. Yu, S. Cheng, J. Xu, Cascade spiking neuron network for event-based image classification in noisy environment. *TechRxiv* (2021).

121. A. Vicente-Sola, D. L. Manna, P. Kirkland, G. Di Caterina, T. Bihl, Keys to accurate feature extraction using residual spiking neural networks. *arXiv preprint arXiv:2111.05955* (2021).

122. Y. Li, Y. Lei, X. Yang, Spikeformer: A novel architecture for training high-performance low-latency spiking neural network. *arXiv preprint arXiv:2211.10686* (2022).

123. C. Yu, Z. Gu, D. Li, G. Wang, A. Wang, E. Li, Stsc-snn: Spatio-temporal synaptic connection with temporal convolution and attention for spiking neural networks. *arXiv preprint arXiv:2210.05241* (2022).

124. C. Jin, R.-J. Zhu, X. Wu, L.-J. Deng, Sit: A bionic and non-linear neuron for spiking neural network. *arXiv preprint arXiv:2203.16117* (2022).

125. J. Tang, J.-H. Lai, W.-S. Zheng, L. Yang, X. Xie, Relaxation lif: A gradient-based spiking neuron for direct training deep spiking neural networks. *Neurocomputing* **501**, 499–513 (2022).

126. X. Wu, Y. Zhao, Y. Song, Y. Jiang, Y. Bai, X. Li, Y. Zhou, X. Yang, Q. Hao, Dynamic threshold integrate and fire neuron model for low latency spiking neural networks. *Available at SSRN 4179879* .

127. H. Wang, Y.-F. Li, Bioinspired membrane learnable spiking neural network for autonomous vehicle sensors fault diagnosis under open environments. *Reliability Engineering & System Safety* **233**, 109102 (2023).

128. I. Hammouamri, T. Masquelier, D. G. Wilson, Mitigating catastrophic forgetting in spiking neural networks through threshold modulation. *Transactions on Machine Learning Research* (2022).

129. X. Yao, F. Li, Z. Mo, J. Cheng, *Advances in Neural Information Processing Systems (NeurIPS)*, A. H. Oh, A. Agarwal, D. Belgrave, K. Cho, eds. (2022).

130. S. R. Kheradpisheh, M. Mirsadeghi, T. Masquelier, Spiking neural networks trained via proxy. *IEEE Access* (2022).

131. Q. Meng, M. Xiao, S. Yan, Y. Wang, Z. Lin, Z.-Q. Luo, *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 12444–12453.

132. Y. Zhou, Y. Jin, Y. Sun, J. Ding, Surrogate-assisted cooperative co-evolutionary reservoir architecture search for liquid state machines. *IEEE Transactions on Emerging Topics in Computational Intelligence* pp. 1–15 (2023).

133. X. Lin, Z. Zhang, D. Zheng, Supervised learning algorithm based on spike train inner product for deep spiking neural networks. *Brain Sciences* **13** (2023).

134. Q. Zhan, G. Liu, X. Xie, M. Zhang, G. Sun, Bio-inspired active learning method in spiking neural network. *Knowledge-Based Systems* **261**, 110193 (2023).

135. S. Lucas, E. Portillo, A. Zubizarreta, I. Cabanes, *XLIII Jornadas de Automática* (Universidade da Coruña. Servizo de Publicacións, 2022), pp. 216–223.

136. G. Chen, P. Peng, G. Li, Y. Tian, Training full spike neural networks via auxiliary accumulation pathway. *arXiv preprint arXiv:2301.11929* (2023).

137. C. Duan, J. Ding, S. Chen, Z. Yu, T. Huang, *Advances in Neural Information Processing Systems (NeurIPS)*, A. H. Oh, A. Agarwal, D. Belgrave, K. Cho, eds. (2022).

138. Q. Yang, J. Wu, M. Zhang, Y. Chua, X. Wang, H. Li, *Advances in Neural Information Processing Systems (NeurIPS)*, A. H. Oh, A. Agarwal, D. Belgrave, K. Cho, eds. (2022).

139. M. Xiao, Q. Meng, Z. Zhang, D. He, Z. Lin, *Advances in Neural Information Processing Systems (NeurIPS)*, A. H. Oh, A. Agarwal, D. Belgrave, K. Cho, eds. (2022).

140. Z. Zheng, X. Jia, Label distribution learning via implicit distribution representation. *arXiv preprint arXiv:2209.13824* (2022).

141. Y. Feng, S. Geng, J. Chu, Z. Fu, S. Hong, Building and training a deep spiking neural network for ecg classification. *Biomedical Signal Processing and Control* **77**, 103749 (2022).

142. Y. Xing, L. Zhang, Z. Hou, X. Li, Y. Shi, Y. Yuan, F. Zhang, S. Liang, Z. Li, L. Yan, Accurate ecg classification based on spiking neural network and attentional mechanism for real-time implementation on personal portable devices. *Electronics* **11** (2022).

143. Y. Chen, Z. Yu, W. Fang, T. Huang, Y. Tian, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI)*, Z.-H. Zhou, ed. (International Joint Conferences on Artificial Intelligence Organization, 2021), pp. 1713–1721. Main Track.

144. F. Liu, W. Zhao, Y. Chen, Z. Wang, F. Dai, *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (IEEE, 2022), pp. 2130–2134.

145. Y. Chen, Z. Yu, W. Fang, Z. Ma, T. Huang, Y. Tian, *International Conference on Machine Learning (ICML)* (PMLR, 2022), pp. 3701–3715.

146. Y. Kim, Y. Li, H. Park, Y. Venkatesha, R. Yin, P. Panda, Lottery ticket hypothesis for spiking neural networks. *arXiv preprint arXiv:2207.01382* (2022).

147. Y. Chen, Z. Ma, W. Fang, X. Zheng, Z. Yu, Y. Tian, *International Conference on Learning Representations (ICLR)* (2023).

148. B. Na, J. Mok, S. Park, D. Lee, H. Choe, S. Yoon, Autosnn: Towards energy-efficient spiking neural networks. *arXiv preprint arXiv:2201.12738* (2022).

149. Y. Kim, Y. Li, H. Park, Y. Venkatesha, P. Panda, Neural architecture search for spiking neural networks. *arXiv preprint arXiv:2201.10355* (2022).

150. Y. Li, Y. Kim, H. Park, T. Geller, P. Panda, Neuromorphic data augmentation for training spiking neural networks. *arXiv preprint arXiv:2203.06145* (2022).

151. R.-J. Zhu, Q. Zhao, J. K. Eshraghian, Spikegpt: Generative pre-trained language model with spiking neural networks. *arXiv preprint arXiv:2302.13939* (2023).

152. S. Xiang, T. Zhang, S. Jiang, Y. Han, Y. Zhang, C. Du, X. Guo, L. Yu, Y. Shi, Y. Hao, Spiking siamfc++: Deep spiking neural network for object tracking. *arXiv preprint arXiv:2209.12010* (2022).

153. Y. Xiong, Y. Chen, C. Chen, X. Wei, Y. Xue, H. Wan, P. Wang, An odor recognition algorithm of electronic noses based on convolutional spiking neural network for spoiled food identification. *Journal of The Electrochemical Society* **168**, 077519 (2021).

154. J. Cuadrado, U. Rançon, B. Cottereau, F. Barranco, T. Masquelier, Optical flow estimation with event-based cameras and spiking neural networks. *arXiv preprint arXiv:2302.06492* (2023).

155. G. Liu, W. Deng, X. Xie, L. Huang, H. Tang, Human-level control through directly-trained deep spiking q-networks. *arXiv preprint arXiv:2201.07211* (2021).

156. D. Chen, P. Peng, T. Huang, Y. Tian, Deep reinforcement learning with spiking q-learning. *arXiv preprint arXiv:2201.09754* (2022).

157. L. Qin, R. Yan, H. Tang, A low latency adaptive coding spiking framework for deep reinforcement learning. *arXiv preprint arXiv:2211.11760* (2022).

158. M. Wang, J. Li, M. Ma, X. Fan, Snn-sc: A spiking semantic communication framework for classification. *arXiv preprint arXiv:2210.06836* (2022).

159. A. Morrison, M. Diesmann, W. Gerstner, Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics* **98**, 459–478 (2008).

160. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym (2016).

161. R. Wightman, H. Touvron, H. Jégou, Resnet strikes back: An improved training procedure in timm (2021).

162. R. Yuan, Q. Duan, P. J. Tiw, G. Li, Z. Xiao, Z. Jing, K. Yang, C. Liu, C. Ge, R. Huang, Y. Yang, A calibratable sensory neuron based on epitaxial vo2 for spike-based neuromorphic multisensory system. *Nature Communications* **13**, 1–12 (2022).

163. X. Li, Z. Feng, J. Zou, X. Wang, G. Hu, F. Wang, C. Ding, Y. Zhu, F. Yang, Z. Wu, Y. Dai, A model of taox threshold switching memristor for neuromorphic computing. *Journal of Applied Physics* **132**, 064904 (2022).

164. A. L. Hodgkin, A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology* **117**, 500–544 (1952).

165. E. M. Izhikevich, Simple model of spiking neurons. *IEEE Transactions on Neural Networks* **14**, 1569–1572 (2003).

166. E. Ledinauskas, J. Ruseckas, A. Juršėnas, G. Buračas, Training Deep Spiking Neural Networks. *arXiv preprint arXiv:2006.04436* (2020).

167. J. H. Lee, T. Delbruck, M. Pfeiffer, Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience* **10**, 508 (2016).

168. C. Lee, S. S. Sarwar, P. Panda, G. Srinivasan, K. Roy, Enabling spike-based backpropagation for training deep neural network architectures. *Frontiers in Neuroscience* **14** (2020).

169. T. Gollisch, M. Meister, Rapid neural coding in the retina with relative spike latencies. *Science* **319**, 1108–1111 (2008).

170. J. Kim, H. Kim, S. Huh, J. Lee, K. Choi, Deep neural networks with weighted spikes. *Neurocomputing* **311**, 373–386 (2018).

171. L. R. Iyer, Y. Chua, H. Li, Is neuromorphic mnist neuromorphic? analyzing the discriminative power of neuromorphic datasets in the time domain. *Frontiers in Neuroscience* **15** (2021).

172. Y. Bi, A. Chadha, A. Abbas, E. Bourtsoulatze, Y. Andreopoulos, *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (2019).

173. H. Li, H. Liu, X. Ji, G. Li, L. Shi, Cifar10-dvs: An event-stream dataset for object classification. *Frontiers in Neuroscience* **11** (2017).

174. X. Wang, Z. Wu, B. Jiang, Z. Bao, L. Zhu, G. Li, Y. Wang, Y. Tian, Hardvs: Revisiting human activity recognition with dynamic vision sensors. *arXiv preprint arXiv:2211.09648* (2022).

175. J.-M. Maro, S.-H. Ieng, R. Benosman, Event-based gesture recognition with dynamic background suppression using smartphone computational capabilities. *Frontiers in Neuroscience* **14** (2020).

176. B. Cramer, Y. Stradmann, J. Schemmel, F. Zenke, The heidelberg spiking data sets for the systematic evaluation of spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems* **33**, 2744–2757 (2022).

177. Y. Wu, R. Zhao, J. Zhu, F. Chen, M. Xu, G. Li, S. Song, L. Deng, G. Wang, H. Zheng, S. Ma, J. Pei, Y. Zhang, M. Zhao, L. Shi, Brain-inspired global-local

learning incorporated with neuromorphic computing. *Nature Communications* **13**, 1–14 (2022).

178. R. Nishino, S. H. C. Loomis, Cupy: A numpy-compatible library for nvidia gpu calculations. *Advances in Neural Information Processing Systems (NeurIPS)* **151** (2017).

179. F. Zenke, T. P. Vogels, The remarkable robustness of surrogate gradient learning for instilling complex function in spiking neural networks. *BioRxiv* (2020).

180. B. Yin, F. Corradi, S. M. Bohté, Accurate and efficient time-domain classification with adaptive spiking recurrent neural networks. *Nature Machine Intelligence* **3**, 905–913 (2021).

181. R. V. Florian, Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation* **19**, 1468–1502 (2007).

182. S. Park, S. Kim, B. Na, S. Yoon, *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20 (IEEE Press, 2020).

183. Y. Zhu, Z. Yu, W. Fang, X. Xie, T. Huang, T. Masquelier, *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, K. Cho, eds. (2022).

184. A. Rao, P. Plank, A. Wild, W. Maass, A long short-term memory for ai applications in spike-based neuromorphic hardware. *Nature Machine Intelligence* **4**, 467–479 (2022).

185. S. Kim, S. Kim, S. Hong, S. Kim, D. Han, H.-J. Yoo, *2023 IEEE International Solid-State Circuits Conference (ISSCC)* (IEEE, 2023), pp. 334–336.

186. M. Chang, A. S. Lele, S. D. Spetalnick, B. Crafton, S. Konno, Z. Wan, A. Bhat, W.-S. Khwa, Y.-D. Chih, M.-F. Chang, A. Raychowdhury, *2023 IEEE International Solid-State Circuits Conference (ISSCC)* (IEEE, 2023), pp. 426–428.

187. G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, W. Maass, Long short-term memory and learning-to-learn in networks of spiking neurons. *Advances in neural information processing systems* **31** (2018).

188. W. Ponghiran, K. Roy, *Proceedings of the AAAI Conference on Artificial Intelligence* (2022), vol. 36, pp. 8001–8008.

189. Y. Bengio, N. Léonard, A. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

190. I. Loshchilov, F. Hutter, Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

191. A. Shymyrbay, M. E. Fouda, A. Eltawil, Low precision quantization-aware training in spiking neural networks with differentiable quantization function. *arXiv preprint arXiv:2305.19295* (2023).

192. G. Tang, N. Kumar, R. Yoo, K. Michmizos, *Conference on Robot Learning* (PMLR, 2021), pp. 2016–2029.

193. S. Fujimoto, H. Hoof, D. Meger, *International conference on machine learning* (PMLR, 2018), pp. 1587–1596.

194. L. Huang, Z. Ma, L. Yu, H. Zhou, Y. Tian, *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (2023), vol. 37, pp. 31–39.

195. C. Choy, J. Gwak, S. Savarese, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 3075–3084.

196. D. Zhang, T. Zhang, S. Jia, B. Xu, *Proceedings of the AAAI Conference on Artificial Intelligence* (2022), vol. 36, pp. 59–67.

致谢

作者在此诚挚感谢以下支持者与贡献者：来自 GitHub 和 OpenIntelligence（OpenI）社区的 Yifan Huang、Liuzhenghao LYU、Liutao Yu、Ruijie Zhu、Quanyu Pu、Yumin Ye、Haonan Qiu、Dongyang Ma、Ismail Khalfaoui-Hassani、Yichen Pan、Mingqing Xiao、Chen Sun、Yaoyu Zhu、Huanhuan Gao、Hengyu Zhang 和 Man Yao，感谢他们提交代码；感谢 OpenI 提供奖金和服务器支持；感谢马征宇及鹏城云脑提供计算资源；感谢 Intel Neuromorphic Research Community 提供远程访问 Intel Loihi 的机会；感谢鹏城实验室的 Yanyu Lin、Xueke Zhu、Wenjie Lin 以及 Lynxi 的工程师协助完成 Lynxi KA200 上的部署；感谢 Tong Bu、Xinhao Luo 和 Siyu Ding 协助完成 Intel Loihi 上的部署；感谢 Donghyun Lee 和 Zidong Zhou 对量化数学运算的 CUDA 加速进行了探索。

资金

该工作得到国家自然科学基金（62027804、61825101、62088102、62236009 和 62176003）、鹏城实验室重大攻关项目（PCL2021A13）和北京市自然科学基金（JQ21015）的资助。

作者贡献

田永鸿主导了论文撰写，提出了构建 SpikingJelly 的最初设想，并监督了整个项目。李国齐参与了本文结构设计，修订了论文内容，贡献了交换模块、数据集和 CUDA 加速相关工作，并在社区中推广了 SpikingJelly。方维主导了 SpikingJelly 的开发并撰写了本文。陈彦骐参与了 SpikingJelly 的结构设计，并贡献了复杂神经元相关代码。

丁健豪贡献了 ANN2SNN 相关代码，并参与了深度 SNN 模型架构的开发。余肇飞和 Timothée Masquelier 参与了数据集、有状态模块和传播模式的设计，并在社区中推广了 SpikingJelly。陈鼎贡献了强化学习及强化学习控制应用相关代码。黄力炜贡献了编码器以及神经相似性应用相关代码。周晖晖参与了 SpikingJelly 的结构设计，并申请了资金和计算资源。所有作者均审阅了手稿。

竞争利益

作者声明没有竞争利益。

数据和材料可用性

SpikingJelly 的源代码可在其 GitHub 页面获取:`https://github.com/fangwei123456/spikingjelly`。图 1**d** 中实验以及图 4 中典型应用的源代码可在 `https://doi.org/10.5281/zenodo.8310901` 获取。

附录

深度学习子包

为简化 SpikingJelly 的使用、协调不同的编程风格、提升可扩展性并降低代码复用成本，SpikingJelly 将面向深度学习的部分进一步划分为四个部分：*Components*、*Functions*、*Acceleration* 和 *Networks*。表 1 展示了这些子包及其典型模块或函数。

Components 包含脉冲神经元、层、编码器和代理函数等基础模块，它们都是 SNN 的组成部分。例如，带有可学习 τ_m 的 Parametric LIF（PLIF）神经元 (*56*) 被实现于

neuron 子包中，其神经元充电函数为

$$H[t] = V[t-1] + \tau_m(a) \cdot (X[t] - (V[t-1] - V_{reset})), \tag{28}$$

$$\tau_m(a) = \frac{1}{1+\exp(-a)}, \tag{29}$$

其中 a 是控制 $\tau_m(a)$ 的可学习参数。

Functions 提供用于训练、仿真、分析、转换、量化和部署 SNN 的函数。例如，面向 SNN 的 FPTT 在线训练算法 (*180*) 就实现于 *functional* 子包中，如算法 2 所示。*Functions* 中的一些模块还是 *Components* 中对应模块的函数式表述，例如 *Functions* 中的 *multi_step_forward* 与 *Components* 中的 *MultiStepContainer*。借助这些函数式表述，用户可以在保留模块定义的同时，通过重写 *forward* 函数来修改前向传播过程。这一特性尤其适合模型复用，因为重写 *forward* 不会影响权重加载。例如，一个已经训练好的 ANN 可以通过重写激活层的 *forward* 函数，方便地改写为 SNN。

Algorithm 2: FPTT 在线训练算法

输入：

带参数 $\mathbf{W}$ 和隐藏状态 $\mathbf{H}[-1]$ 的网络 $f_{\mathbf{W}}$

用于计算梯度、学习率为 η 的优化器

所有时间步上的输入序列 $\{X[t]\}, t = 0, 1, ..., T-1$

所有时间步上的目标序列 $\{O[t]\}, t = 0, 1, ..., T-1$

用于计算每个时间步损失的损失函数 l

梯度缩放系数 α

参数的滑动平均 $\bar{\mathbf{W}}$

初始化 $\frac{\partial L[-1]}{\mathbf{W}[0]} = 0, \mathbf{W}[0] = \bar{\mathbf{W}}[0] = \mathbf{W}$

for $t \leftarrow 0, 1, ..., T-1$

  分离所有隐藏状态 $\mathbf{H}[t-1]$，以避免 BPTT

  $Y[t] = f_{\mathbf{W}[t]}(X[t], \mathbf{H}[t-1])$

  $L[t] = l(Y[t], O[t]) + \frac{\alpha}{2}||\mathbf{W}[t] - \bar{\mathbf{W}}[t] - \frac{1}{2\alpha} \cdot \frac{\partial L[t-1]}{\partial \mathbf{W}[t]}||^2$

  使用优化器和梯度 $\frac{\partial L[t]}{\partial \mathbf{W}[t]}$ 更新 $\mathbf{W}[t]$

  $\bar{\mathbf{W}}[t+1] = \frac{1}{2}(\bar{\mathbf{W}}[t] + \mathbf{W}[t+1]) - \frac{1}{2\alpha}\mathbf{W}[t+1]$

Acceleration 通过半自动生成的 CUDA 内核为 SNN 仿真提供额外的可选加速功能，

从而兼顾底层编程语言的高效率以及代码生成技术带来的低开发成本。一些利用二值特性降低内存消耗的优化层也被集成到 *Acceleration* 中。

在以上三个部分的基础上，*Networks* 提供经典和大规模网络结构。常见结构如常规 SNN 会配合教程一起提供，以帮助新用户以较低成本快速上手 SpikingJelly。大规模网络结构则包括 PLIF Net (*56*)、Spiking VGG、Spiking ResNet 和 SEW ResNet (*60*)，并附带训练脚本，可直接用于模型复用。

使用示例

借助 SpikingJelly，构建和训练 SNN 都可以较为轻松地完成。下面我们给出两个示例。若需更多示例，建议参考 SpikingJelly 首页中的教程。

构建循环 SNN

图 6 展示了构建带循环连接 SNN 的示例。图 6**a** 为该示例的源代码。在 **第 5–11 行**中，循环结构由全连接层 FC 和 LIF 神经元层 LIF 构成，并通过逐元素循环容器进行封装，其中逐元素函数取加法。记该循环结构在时间步 t 的外部输入和输出分别为 $X[t]$ 和 $Y[t]$，则其前向传播可写为

$$Y[t] = LIF(FC(X[t] + Y[t-1])). \tag{30}$$

需要注意的是，逐元素循环容器中的逐元素函数可以由用户自由定义。在 **第 13–19 行**中，我们分别在循环结构前后各加入两个前馈层，从而构建完整的 SNN，其结构如图 6**b** 所示。

需要注意，循环结构只能工作在单步模式下，因为在 $t = 0$ 时无法预先给出所有时间步上的真实输入 $X[t] + Y[t-1]$。在图 6**b** 的 **第 21 行**中，我们使用 set_step_mode 函数将 SNN 中所有模块都设置为多步模式，但这一设置不会作用于循环结构中的 FC 和 LIF，它们仍保持单步模式。其他模块，包括循环容器本身，则会工作在多步模式

部分	子包	说明	典型模块/函数
Components	neuron	脉冲神经元	LIF 神经元 带可学习 τ_m 的 Parametric LIF 神经元 (*56*)
	layer	突触与容器	有状态突触 多维注意力 (*110*)
	encoding	脉冲编码器	泊松编码器 加权相位编码器 (*170*)
	surrogate	代理梯度函数	sigmoid 代理函数 arctan 代理函数
Functions	ann2snn	ANN2SNN 转换函数	conv/bn 融合函数 最大激活归一化
	functional	前向、损失和设置函数	重置整个网络状态 FPTT 在线训练 (*180*)
	monitor	记录特定层数据的监视器	记录层属性的 attribute monitor 记录层输出的 output monitor
	learning	生物合理学习规则	STDP learner 带资格迹的调制式 STDP learner (*181*)
	quantize	量化器	带代理梯度的舍入函数 k bit 量化器
	exchange	面向神经形态计算芯片的转换模块	简化 IF 神经元 将突触和神经元打包为 Lava block
Acceleration	auto cuda	用于加速的 CUDA 内核	sigmoid 代理梯度函数 LIF 神经元 BPTT 的 CUDA 内核
	spike op	面向脉冲操作的存储优化层	float-to-char 脉冲 CUDA 转换函数 脉冲卷积层
Networks	model	预定义经典 SNN	Spiking ResNet SEW ResNet (*60*)
	example	便于上手的教程示例	训练脉冲 CNN 的教程 训练大规模深度 SNN 的教程

表 1: **SpikingJelly 子包介绍。**

a

```
1.   import torch
2.   import torch.nn as nn
3.   from spikingjelly.activation_based import neuron, layer, functional
4.
5.   recurrent_layers = layer.ElementWiseRecurrentContainer(
6.           sub_module=nn.Sequential(
7.               layer.Linear(8, 8),
8.               neuron.LIFNode(),
9.           ),
10.      element_wise_function=torch.add
11.  )
12.
13.  net = nn.Sequential(
14.      layer.Linear(8, 8),
15.      neuron.LIFNode(),
16.      recurrent_layers,
17.      layer.Linear(8, 4),
18.      neuron.LIFNode(),
19.  )
20.
21.  functional.set_step_mode(net, step_mode='m')
```

b

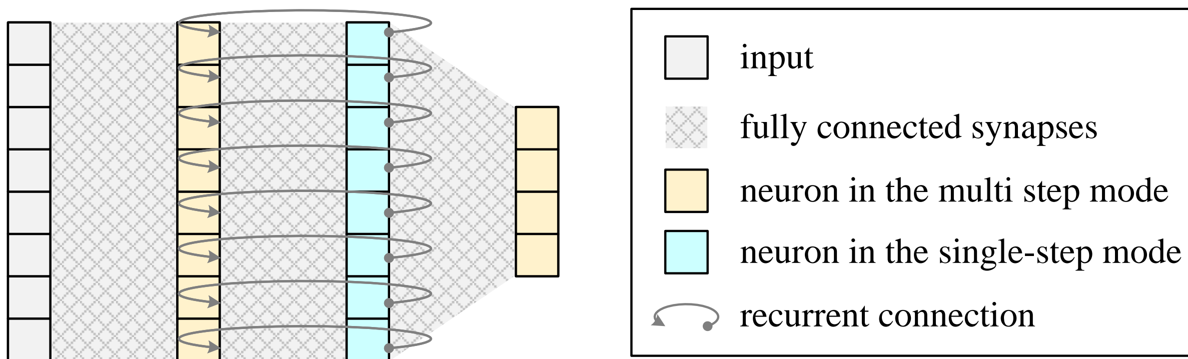

图 6: 构建带循环连接 SNN 的示例。**a**. 源代码。**b**. 网络结构。

下。这样一来，该 SNN 就形成了一种有趣的混合情况：仿真时同时使用两种传播模式。循环部分（即循环容器内部模块）采用逐步传播模式，而前馈部分则采用逐层传播模式，因此基于多步模式的加速方法，例如时间步与批次维度融合以及融合 CUDA 内核，仍然可以被利用。

训练用于分类任务的 SNN

图 7 展示了如何使用 SpikingJelly 构建并训练一个用于 MNIST 分类的 SNN。对于熟悉 PyTorch 的读者来说，这段代码会比较直观。在 **第 8–13 行**中，我们定义了时

```
1.  import torch
2.  import torch.nn as nn
3.  import torch.nn.functional as F
4.  import torch.utils.data as data
5.  import torchvision
6.  from spikingjelly.activation_based import neuron, encoding, functional, layer
7.  
8.  T = 100
9.  device = 'cuda:0'
10. batch_size = 64
11. epochs = 100
12. data_dir = 'D:/datasets/MNIST'
13. lr = 1e-3
14. 
15. net = nn.Sequential(
16.         encoding.PoissonEncoder(),
17.         layer.Flatten(),
18.         layer.Linear(28 * 28, 10, bias=False),
19.         neuron.LIFNode(),
20.     )
21. functional.set_step_mode(net, step_mode='m')
22. net.to(device)
23. optimizer = torch.optim.Adam(net.parameters(), lr=lr)
24. 
25. to_tensor = torchvision.transforms.ToTensor()
26. train_dataset = torchvision.datasets.MNIST(root=data_dir, train=True, transform=to_tensor, download=True)
27. test_dataset = torchvision.datasets.MNIST(root=data_dir, train=False, transform=to_tensor, download=True)
28. train_data_loader = data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)
29. test_data_loader = data.DataLoader(dataset=test_dataset, batch_size=batch_size)
30. 
31. for epoch in range(epochs):
32.     net.train()
33.     train_loss = 0
34.     train_acc = 0
35.     train_samples = 0
36.     for img, label in train_data_loader:
37.         optimizer.zero_grad()
38.         img = img.to(device).unsqueeze(0).repeat(T, 1, 1, 1, 1)  # [N, C, H, W] -> [1, N, C, H, W] -> [T, N, C, H, W]
39.         label = label.to(device)
40.         label_onehot = F.one_hot(label, 10).float()
41.         out_fr = net(img).mean(0)
42.         loss = F.mse_loss(out_fr, label_onehot)
43.         loss.backward()
44.         optimizer.step()
45. 
46.         train_samples += label.numel()
47.         train_loss += loss.item() * label.numel()
48.         train_acc += (out_fr.argmax(1) == label).float().sum().item()
49.         functional.reset_net(net)
50.     train_loss /= train_samples
51.     train_acc /= train_samples
52. 
53.     net.eval()
54.     test_loss = 0
55.     test_acc = 0
56.     test_samples = 0
57.     with torch.no_grad():
58.         for img, label in test_data_loader:
59.             img = img.to(device).unsqueeze(0).repeat(T, 1, 1, 1, 1)
60.             label = label.to(device)
61.             label_onehot = F.one_hot(label, 10).float()
62.             out_fr = net(img).mean(0)
63.             loss = F.mse_loss(out_fr, label_onehot)
64. 
65.             test_samples += label.numel()
66.             test_loss += loss.item() * label.numel()
67.             test_acc += (out_fr.argmax(1) == label).float().sum().item()
68.             functional.reset_net(net)
69.     test_loss /= test_samples
70.     test_acc /= test_samples
71.     print(f'epoch = {epoch}, train loss = {train_loss}, train acc = {train_acc}, test loss = {test_loss}, test acc = {test_acc}')
```

图 7: 构建并训练用于 MNIST 分类的 SNN 示例。

间步数 T、设备、批量大小、训练轮数、数据集路径和学习率等超参数。在 **第 15–20 行**中，我们将 SNN 定义为由泊松编码器、展平层、全连接层和 LIF 神经元层构成。泊松编码器用于将浮点输入值转换为脉冲；展平层则将形状为 $(..., C, H, W)$ 的输入重塑为 $(..., C \cdot H \cdot W)$，其中 C、H 和 W 分别表示通道数、高度和宽度。

在 **第 21 行**中，我们将 SNN 中的所有模块都设置为多步模式；在 **第 22 行**中，将 SNN 移动到指定设备。需要注意的是，SpikingJelly 与 CPU 和 GPU 都完全兼容，因此设备也可以设为 *cpu*。在这个示例中，我们将设备设为 *cuda:0*，即训练环境中的第 0 块 GPU。在 **第 23–29 行**中，定义了 Adam 优化器、训练集加载器和测试集加载器，这也是训练网络时的标准做法。

在 **第 31–71 行**中，定义了训练循环。**第 32 行**将 SNN 置于训练模式，这是 PyTorch 中的概念，SpikingJelly 也遵循这一约定。诸如 batch normalization 和 dropout 之类的模块在训练和推理阶段具有不同的行为，因此需要显式区分。**第 32 行**和 **第 53 行**分别对应训练和推理模式设置。在 **第 33–35 行**中，我们初始化了用于记录训练损失、训练精度和样本数的变量。在 **第 36–49 行**中，实现了对训练集的小批量梯度下降。**第 38 行**中，原始图像的形状为 (N, C, H, W)，并不包含时间步维度，因此我们先将其扩展为 $(1, N, C, H, W)$，再沿时间步维度重复 T 次，得到形状为 (T, N, C, H, W) 的输入序列。**第 41 行**中，输入序列送入 SNN 后，输出序列的形状为 $(T, N, 10)$。我们利用输出序列的发放率来计算损失并得到分类结果；这里的发放率指的是在时间步维度上进行统计。**第 42 行**中，损失被定义为发放率与 one-hot 标签之间的均方误差。在 **第 43–44 行**中，我们执行反向传播计算梯度，并用优化器更新参数。需要注意，这一过程中使用了代理学习方法，而它已经在脉冲神经元层内部完成实现。**第 48 行**中，将发放率最高的类别索引视为分类结果，也就是代码中的 *argmax*。在 **第 53–70 行**中定义的推理流程与训练流程基本相同。

这个简单的 SNN 在测试集上达到了 92.9% 的准确率。关于训练细节和实验结果，可进一步参考 SpikingJelly 首页中的“Single Fully Connected Layer SNN to Classify

MNIST”教程；这里的示例是该教程的一个简化版本。

StepModule 的设计

StepModule 是 SpikingJelly 中的核心基类，其属性 *step_mode* 用于控制步进模式。图 8 展示了 *StepModule* 的继承关系图。当 *step_mode = 's'* 时，模块工作在单步模式下，并处理形状为 $(N, ...)$ 的数据；当 *step_mode = 'm'* 时，模块工作在多步模式下，并处理形状为 $(T, N, ...)$ 的数据。继承自 *StepModule* 后，无状态模块（如 *Conv2d*）能够自然支持两种步进模式，而多步前向过程则可通过将时间步维度并入批次维度来加速。需要注意的是，如果一个 SNN 中的所有模块都基于 *StepModule* 构建，那么该网络就可以方便地在逐步传播和逐层传播两种模式之间切换。这种设计满足了灵活仿真的需求，例如在训练阶段使用较小的 T 和逐层传播以获得更快速度与更低内存开销，而在推理阶段使用较大的 T 和逐步传播。

MultiStepModule 通过固定 *step_mode = 'm'* 从 *StepModule* 继承而来，以强制自身运行于多步模式。像 *SeqToANNContainer* 这样的模块，就是建立在 *MultiStepModule* 基础之上的。

脉冲神经元等有状态模块在 SNN 中被广泛使用。与 PyTorch 将隐藏状态视为输入和输出一部分的风格不同，SpikingJelly 选择将隐藏状态存储在模块内部，这一机制由 *MemoryModule* 实现。在 *StepModule* 的基础上，*MemoryModule* 增加了一个 Python 字典 *memory*，用于存储隐藏状态。例如，神经元基类 *BaseNode* 的 *memory* 中存储了表示脉冲神经元膜电位 $V[t]$ 的 v。再比如，SpikingJelly 中由 *Dropout* 实现的 SNN dropout 层使用时间不变掩码 (*168*)，该掩码同样被视为 *memory* 的一项属性。

步进模式与传播模式的区别

模块的步进模式控制其传播行为。将所有模块的 *step_mode* 设置为 *"s"*（单步）或 *"m"*（多步）后，SNN 将分别以逐步传播模式或逐层传播模式运行。因此，两种步进模

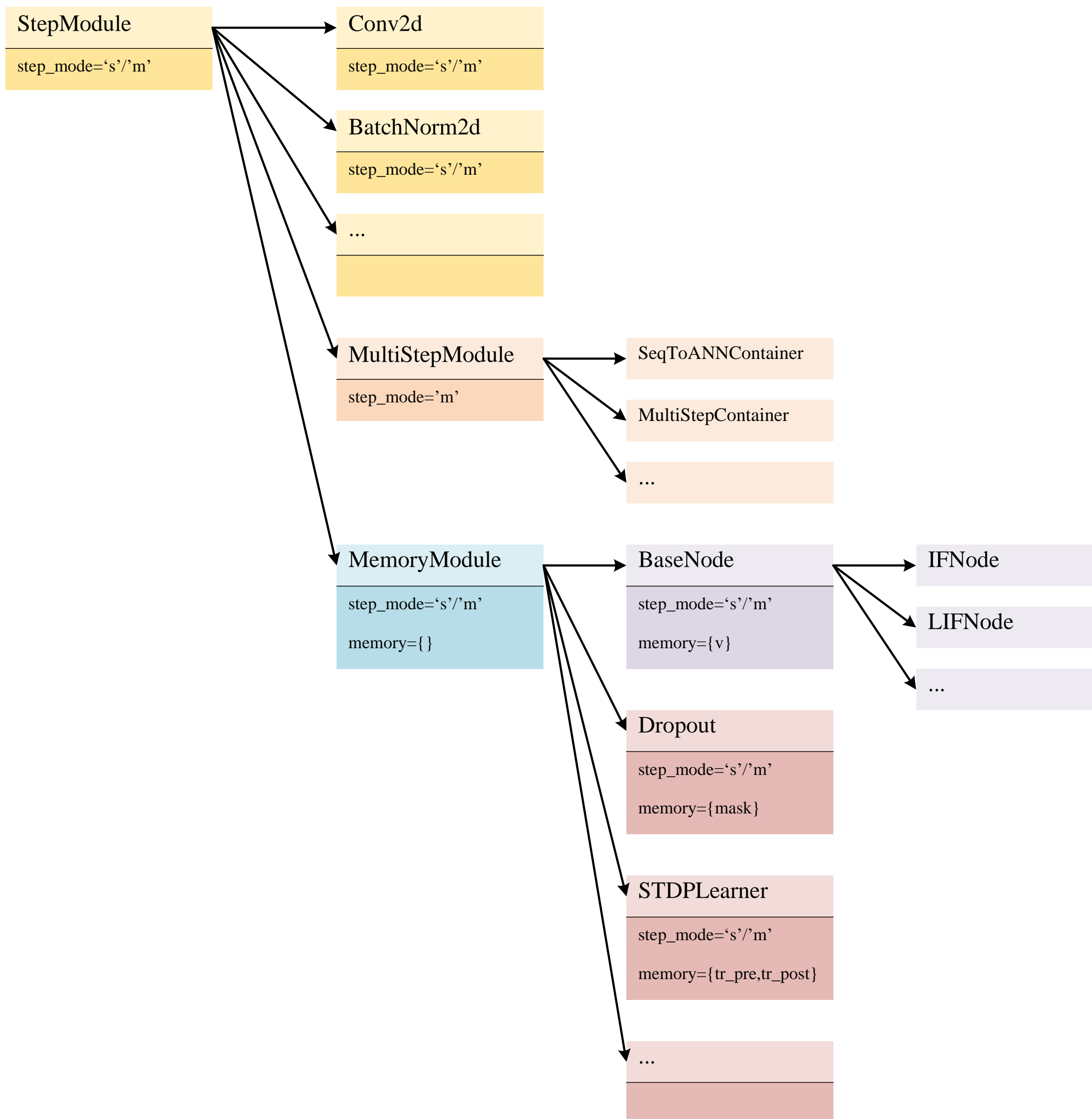

图 8: ***StepModule* 的继承关系图。** *StepModule* 是核心基类，其 *step_mode* 属性控制模块是对形状为 $(N, ...)$ 的数据使用单步模式，还是对形状为 $(T, N, ...)$ 的数据使用多步模式。继承自 *StepModule* 后，无状态模块（如 *Conv2d*）可以很方便地同时支持两种步进模式。通过固定 *step_mode='s'*，子类 *MultiStepModule* 被用作那些只能在多步模式下工作的模块基类。进一步加入用于存储膜电位等隐藏状态的 *memory* 属性后，就得到作为有状态模块基类的 *MemoryModule*。

式之间的差异，主要体现在两种传播模式的差异上。设一个 SNN 的层数为 L，时间步数为 T，批量大小为 N；设其中第 i 层在单个时间步上处理数据所需时间为 $d_s[i]$，在所有时间步上处理数据所需时间为 $d_m[i]$。$d_s[i]$ 和 $d_m[i]$ 也分别对应第 i 层在两种步进模式下的时延。记 $X[t]$ 为时间步 t 的输入，$Y_{L-1}[t]$ 为 SNN 在该时间步的输出。

内存消耗

当使用 BPTT 训练 SNN 时，由于两种传播模式对应的完整计算图是相同的，如图 3 所示，因此二者的空间复杂度均为 $\mathcal{O}(N \cdot T \cdot L)$。

但在使用 SNN 进行推理时，由于数据与计算的局部性，空间复杂度会有所不同。如图 **11a** 所示，当采用逐步推理并在时间步 t 上传播时，计算局限于单个时间步，因此时间步 $t-1$ 上所有层的输出和隐藏状态都可以被释放。于是，逐步传播模式在推理时的空间复杂度为 $\mathcal{O}(N \cdot L)$，不随 T 成比例增长。因此，这种模式适合于时间步很多的推理任务，例如 ANN2SNN。

当采用逐层推理并传播到第 i 层时，如图 **11b** 所示，计算局限于单层，因此前面各层的数据都可以被释放。于是，逐层传播模式在推理时的空间复杂度为 $\mathcal{O}(N \cdot T)$，更适合用于极深 SNN 的推理。

时延

当第 i 层工作在单步模式下时，它需要用时 $d_s[i]$ 来处理输入 $Y_{i-1}[t]$ 并输出 $Y_i[t]$。若所有层都采用单步模式，并且 SNN 以逐步传播模式运行，则从输入 $X[0]$ 到输出 $Y_{L-1}[T-1]$ 的时延为

$$D_s = T \cdot \sum_{i=0}^{L-1} d_s[i]. \tag{31}$$

当第 i 层工作在多步模式下时，它需要用时 $d_m[i]$ 来处理输入 $Y_{i-1} = \{Y_{i-1}[0], Y_{i-1}[1], ..., Y_{i-1}[T-1]\}$，并输出 $Y_i = \{Y_i[0], Y_i[1], ..., Y_i[T-1]\}$。如果设备仅支持串行计算，则 $d_m[i] = T \cdot d_s[i]$。

在 GPU 上，$d_m[i]$ 可以进一步优化。若第 i 层是无状态层，则在 SpikingJelly 中所有时间步的计算可以并行执行；在理想情况下，如果内存读写尚未达到瓶颈、并发线程数也未超过 GPU 的最大并行处理能力，那么 $d_m[i] \approx d_s[i]$。若第 i 层是有状态层，则多步模式下跨时间步的计算可以融合为一个大型 CUDA 内核。与单步模式下每个时间步都调用小型 CUDA 内核相比，多步模式具有更低的调用开销，因此运行速度更快；当 T 较大时，通常有 $d_m[i] << T \cdot d_s[i]$。若所有层都使用多步模式，并且 SNN 以逐层传播模式运行，则从输入 $X[0]$ 到输出 $Y_{L-1}[T-1]$ 的时延为

$$D_m = \sum_{i=0}^{L-1} d_m[i]. \tag{32}$$

由于几乎总有 $d_m[i] \leq T \cdot d_s[i]$ 成立，因此可以推出 $D_m \leq D_s$。

需要说明的是，上述时延分析尚未考虑输入序列本身的到达时延。逐层传播模式要求在传播前获得所有时间步上的输入，因此在这类任务中，总时延应写为

$$D'_m = D_m + t_{X[T-1]} - t_{X[0]}, \tag{33}$$

其中 $t_{X[0]}$ 和 $t_{X[T-1]}$ 分别表示 $X[0]$ 和 $X[T-1]$ 到达的时刻。

为验证上述分析，我们构建了一个常用结构的深度 SNN：$\{\{Conv2d - IF\} * 2 - MaxPool\} * 2 - Flatten - \{FC - IF\} * 2 - Loss$，其中 $Conv2d$ 为卷积层，IF 为 IF 神经元层，$MaxPool$ 为最大池化层，$*2$ 表示重复 2 次，$Flatten$ 为展平层，FC 为全连接层，$Loss$ 为计算损失的模块。我们在 GPU 上使用 $T = 8$ 个时间步训练该网络，并测量了训练过程中各层前向和反向传播的耗时。基于这些数据，绘制得到图 9 和图 10。两张图都清楚地表明：无论是否启用 CuPy，多步模式下的时延都显著小于单步模式。比较 “MS” 与 “SS (sum)” 可以发现，多步模式下无状态层的时延明显更低，这正是时间步维度与批次维度合并所带来的效果。比较 “MS + CuPy” 与 “MS” 则可以看到，CuPy 后端通过使用大型 CUDA 内核融合操作，显著降低了脉冲神经元的时延。尽管 “MS” 与 “SS” 中不使用 CuPy 后端的脉冲神经元，以及 “MS + CuPy” 与 “MS” 中的无状态

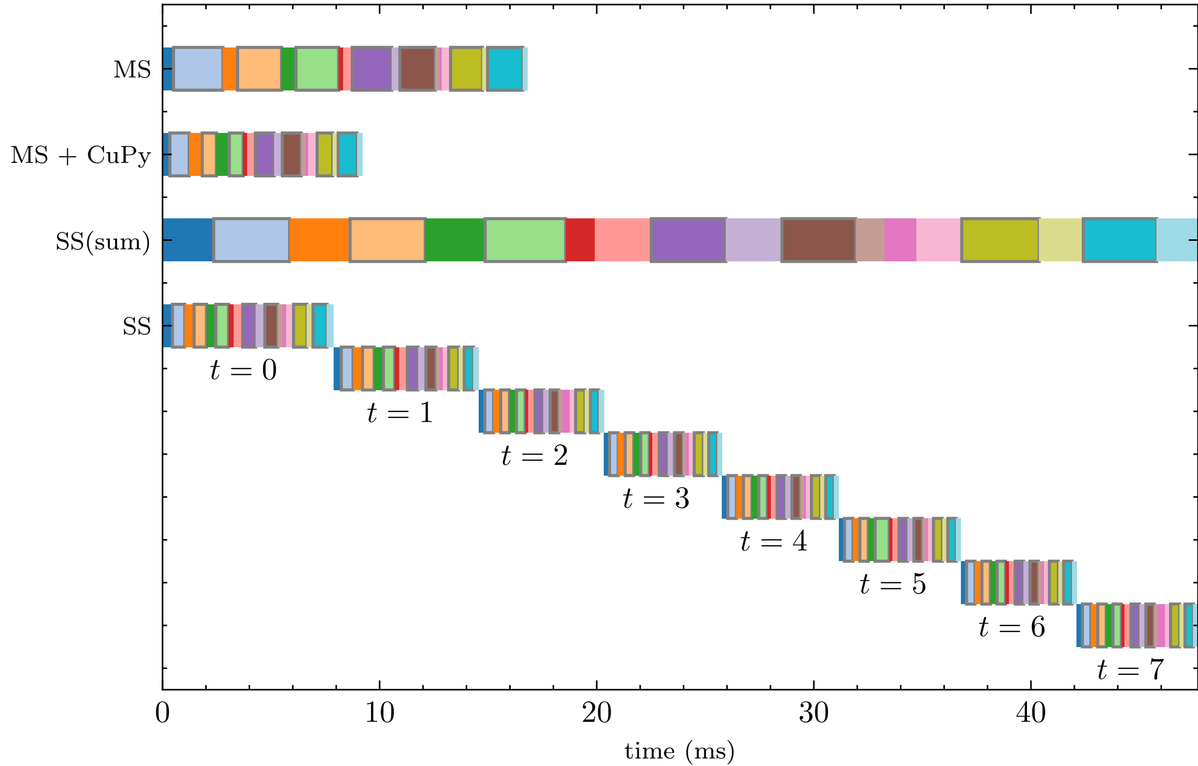

图 9: **SNN 前向传播中各层的时延。**每一层用不同颜色表示。脉冲神经元层以灰色边框标出，无灰边的其他层则为无状态层。“MS” 表示使用多步模式，“CuPy” 表示启用 CuPy 后端，“SS” 表示使用单步模式。“SS (sum)” 是由 “SS” 合并得到的结果，表示将所有时间步上同一层的时延相加。

层，在行为上是相同的，但它们的时延仍存在差异。这是因为当任务 A 和 B 顺序运行时，若任务 A 的效率较低，GPU 就需要花费更多时间进行诸如内存分配、缓存和流重置等后处理操作，从而拖慢任务 B 的执行速度。因此，即便某些层本身的行为一致，当 SNN 中其他层工作在更高效模式下时，这些层的实际运行速度也会更快。总体而言，该实验验证了 $d_m[i] \leq T \cdot d_s[i]$，从而进一步说明 $D_m \leq D_s$。

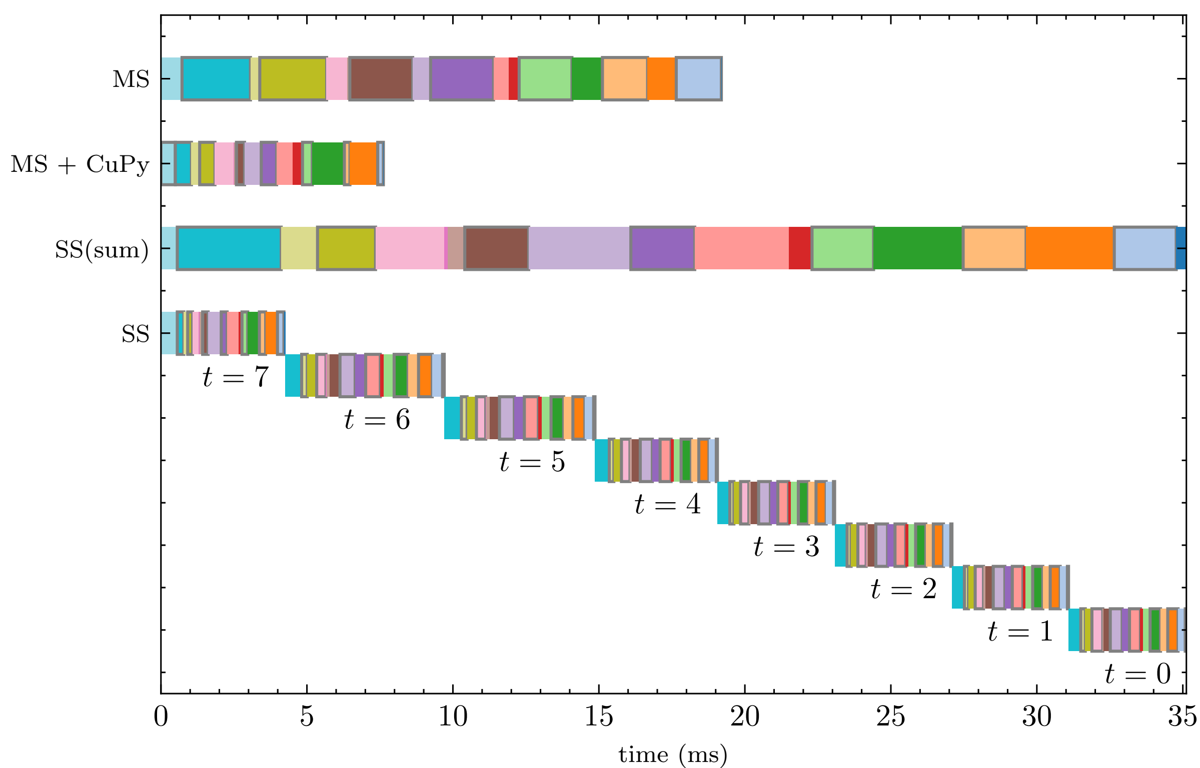

图 10: **SNN 反向传播中各层的时延。**图例说明参见图 9。

步进模式与传播模式的结论

SNN 中各模块的步进模式决定了网络采用哪一种传播模式。两种传播模式仅在构建计算图的顺序上不同，而不会改变 SNN 的输出结果；但这种顺序差异会影响速度、内存消耗和时延。

在 SpikingJelly 中切换传播模式非常容易，只需统一修改所有模块的步进模式即可。用户可以根据具体目标来选择合适的步进模式。在大多数情况下，为了获得更快速度，通常更推荐多步模式。在进行 SNN 推理且内存有限时，当 L 很大可以优先采用多步模式，而当 T 很大则更适合采用单步模式。

某些特定网络、任务和硬件并不同时兼容两种传播模式。例如，一些 SNN 只能在逐层传播模式下运行，如使用首脉冲时间编码 (*95, 182*)、事件驱动反向传播 (*183*) 和注意力 SNN (*110*) 的网络。这类网络需要对所有时间步上的数据进行联合操作，因此只能以逐层模式仿真。在真实任务中，输入 $X[t]$ 往往是间隔到达的，而在某些在线任务中 T 甚至可能趋于无穷，此时网络需要在接收到 $X[t]$ 后尽快输出 $Y[t]$。另外，带循环连接的 SNN (*180, 184*) 会将 $Y[t-1]$ 作为时间步 t 输入的一部分，这也使得网络不可能事先获得所有时间步上的输入数据。

在上述情况下，只能使用逐步传播模式。据我们所知，大多数事件驱动神经形态芯片的工作方式更接近逐步传播模式，即每个事件被处理后再在核心之间路由；但也有一些异构芯片 (*185, 186*) 采用逐层传播模式。总体而言，这两种传播模式都在脉冲深度学习社区中被广泛使用，而 SpikingJelly 对它们都提供了良好支持。

复杂神经元的实现示例

得益于多层继承机制，用户可以在基础神经元类之上以较低成本定义复杂神经元。下面我们用两个示例来说明这一点。

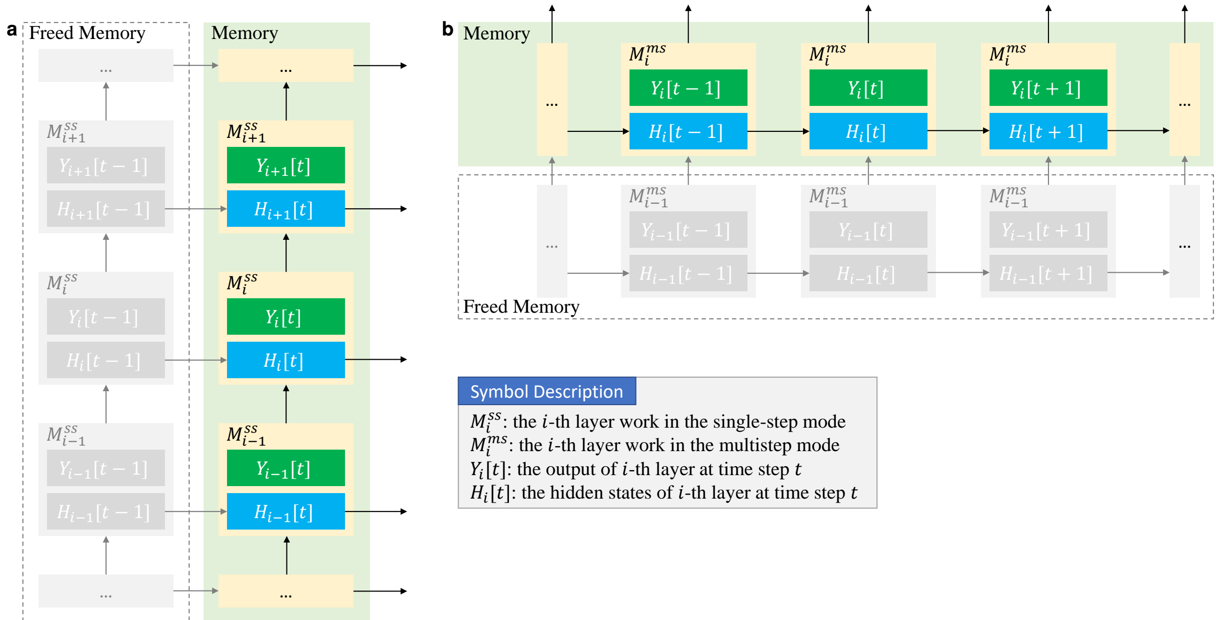

图 11: **两种传播模式在推理时的空间复杂度。**设 SNN 的层数为 L，时间步数为 T。**a.** 当采用逐步推理并在时间步 t 传播时，内存中时间步 $t-1$ 的数据可以被释放，因此逐步传播模式的空间复杂度为 $\mathcal{O}(N \cdot L)$。**b.** 当采用逐层推理并传播到第 i 层时，内存中第 $(i-1)$ 层的数据可以被释放，因此逐层传播模式的空间复杂度为 $\mathcal{O}(N \cdot T)$。

自适应泄漏积分发放神经元的实现

第一个示例是实现自适应泄漏积分发放（Adaptive-Leaky Integrate-and-Fire, ALIF）神经元 (*187*)。它在 LIF 神经元基础上引入了阈值动力学：

$$V_{th}[t] = B^0 + \beta \cdot B[t], \tag{34}$$

$$B[t+1] = \rho \cdot B[t] + (1-\rho) \cdot S[t], \tag{35}$$

其中，$V_{th}[t]$ 表示时间步 t 的阈值，$B[t]$ 表示动态偏移量。B^0 是基础阈值，β 和 ρ 是控制阈值动力学的参数。

如图 12**a** 所示，ALIF 神经元的实现过程如下。**第 5 行**首先继承基础神经元类。**第 6 行**定义构造函数，将新增参数 β、ρ 放在前面，而基类中的参数，如 V_{th}、V_{reset}、代理函数以及步进模式，则通过 **args, **kwargs* 传入。**第 7 行**调用基类构造函数，此后 V_{th} 和 V_{reset} 会被初始化。因此在 **第 8 行**中，我们将初始阈值作为 B^0。在 **第 9–12 行**中，我们将 $B[t]$、β、ρ 和 τ_m 加入成员变量。由于 $B[t]$ 是状态变量，我们将其注册为 memory 变量，这一点在 *StepModule* 的设计一节中已经介绍过。

β、ρ 和 τ_m 不随时间步变化，因此被注册为 buffer。buffer 是 PyTorch 中的一个概念，表示模块中不可学习的参数。需要注意的是，β、ρ 和 τ_m 都是张量而非 float 标量，因此只要形状设置满足 PyTorch 的广播机制，它们就可以按元素、按通道或按层进行设置。比如，当该神经元层位于一个输出形状为 (N, C, H, W) 的卷积层之后时，我们可以把 β、ρ 和 τ_m 的初始形状设置为 (C, H, W)、$(C, 1, 1)$ 或 (1)，分别对应按元素、按通道和按层的神经元动力学。这也是为什么这里没有直接继承 SpikingJelly 中的 LIF 神经元，因为后者将 τ_m 固定定义为按层形式。**第 9–12 行**也很好地说明了：SpikingJelly 原生的 LIF 神经元只需少量修改，就可以变为按元素或按通道版本。

第 14–15 行按照方程 (4) 定义神经元充电函数。**第 17–21 行**定义神经元发放函数。阈值按照方程 (34) 生成，随后可直接调用基类继承来的发放函数完成发放。发放完成后，动态偏移量 $B[t]$ 也会依据方程 (35) 按照 $S[t]$ 更新。最终，只用 21 行代码就完成

a

```
1.  import torch
2.  from spikingjelly.activation_based import neuron, surrogate
3.  from typing import Callable
4.
5.  class ALIFNode(neuron.BaseNode):
6.      def __init__(self, beta: torch.Tensor, rho: torch.Tensor, tau_m: torch.Tensor, *args, **kwargs):
7.          super().__init__(*args, **kwargs)
8.          self.b0 = self.v_threshold
9.          self.register_memory('b', 0.)
10.         self.register_buffer('beta', beta)
11.         self.register_buffer('rho', rho)
12.         self.register_buffer('tau_m', tau_m)
13.
14.     def neuronal_charge(self, x: torch.Tensor):
15.         self.v = self.v + (x - (self.v - self.v_reset)) / self.tau_m
16.
17.     def neuronal_fire(self):
18.         self.v_threshold = self.b0 + self.beta * self.b
19.         spike = super().neuronal_fire()
20.         self.b = self.rho * self.b + (1. - self.rho) * spike
21.         return spike
```

b

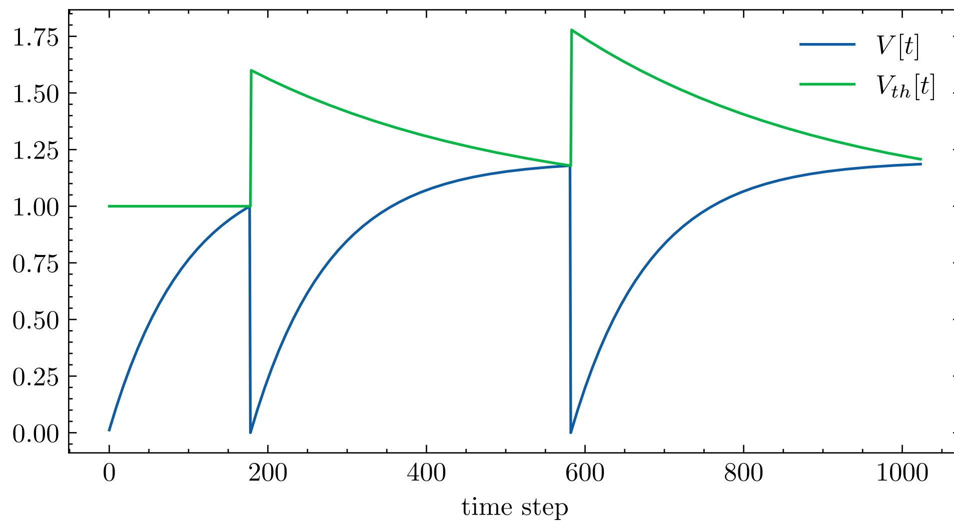

图 12: ALIF 神经元的实现示例。**a**. 源代码。**b**. 在恒定输入下 ALIF 神经元的 $V[t]$ 与 $V_{th}[t]$。

了实现。

图 12**b** 展示了在恒定输入下 ALIF 神经元的 $V[t]$ 与 $V_{th}[t]$ 曲线。可以看到，$V_{th}[t]$ 会在每次发放后瞬时升高，并随后呈指数衰减。这说明 ALIF 神经元具有自适应阈值动力学，能够通过调节阈值来避免发放率过高或过低。

Izhikevich 神经元的实现

第二个示例是 Izhikevich 神经元，其充电函数为

$$H[t] = V[t-1] + \frac{1}{\tau_m} \cdot (X[t] + a_0 \cdot (V[t-1] - V_{rest}) \cdot (V[t-1] - V_c) - W[t-1]), \quad (36)$$

其中，τ_m 是膜时间常数，V_{rest} 是复位电位，a_0 是兴奋性参数，V_c 是临界电位，$W[t]$ 是恢复变量。Izhikevich 神经元在充电后还会更新 $W[t]$：

$$W_{pre}[t] = W[t-1] + \frac{1}{\tau_w} \cdot (a \cdot (V[t-1] - V_{rest}) - W[t-1]), \quad (37)$$

其中，τ_w 是恢复时间常数，a 是恢复更新参数。这里用 $W_{pre}[t]$ 表示更新后、复位前的恢复变量。与膜电位类似，恢复变量在神经元发放后也会被重置：

$$W[t] = W_{pre}[t] + b \cdot S[t], \quad (38)$$

其中，b 是恢复变量的复位参数。

图 13 展示了 Izhikevich 神经元的代码实现示例。在 **第 5–33 行**中，我们先将基础神经元扩展为一个带适应性的基础神经元。更具体地说，**第 5–9 行**继承基础神经元，并在构造函数中加入 τ_w、W_{rest}、a、b 等附加参数。**第 13 行**将 $W[t]$ 设为 memory 状态，因为它是一个状态变量，其默认值为 W_{rest}。**第 15–19 行**将这些附加参数加入成员变量。**第 21–22 行**按照方程 (37) 添加神经元适应函数。**第 24–26 行**在神经元复位函数中加入 $W[t]$ 的更新。**第 28–33 行**重写 forward 函数，并在 **第 30 行**中于神经元充电之后插入适应函数。

```
1.    import torch
2.    from spikingjelly.activation_based import neuron, surrogate
3.    from typing import Callable
4.
5.    class AdaptBaseNode(neuron.BaseNode):
6.        def __init__(self, v_threshold: float = 1., v_reset: float = 0.,
7.                     v_rest: float = 0., w_rest: float = 0., tau_w: float = 2., a: float = 0., b: float = 0.,
8.                     surrogate_function: Callable = surrogate.Sigmoid(), detach_reset: bool = False, step_mode='s',
9.                     backend='torch', store_v_seq: bool = False):
10.
11.           super().__init__(v_threshold, v_reset, surrogate_function, detach_reset, step_mode, backend, store_v_seq)
12.
13.           self.register_memory('w', w_rest)
14.
15.           self.w_rest = w_rest
16.           self.v_rest = v_rest
17.           self.tau_w = tau_w
18.           self.a = a
19.           self.b = b
20.
21.       def neuronal_adaptation(self):
22.           self.w = self.w + 1. / self.tau_w * (self.a * (self.v - self.v_rest) - self.w)
23.
24.       def neuronal_reset(self, spike):
25.          super().neuronal_reset(spike)
26.          self.w = self.w + self.b * spike
27.
28.       def single_step_forward(self, x: torch.Tensor):
29.           self.neuronal_charge(x)
30.           self.neuronal_adaptation()
31.           spike = self.neuronal_fire()
32.           self.neuronal_reset(spike)
33.           return spike
34.
35.
36.   class IzhikevichNode(AdaptBaseNode):
37.       def __init__(self, tau_m: float = 2., v_c: float = 0.8, a0: float = 1., v_threshold: float = 1.,
38.                    v_reset: float = 0., v_rest: float = -0.1, w_rest: float = 0., tau_w: float = 2., a: float = 0.,
39.                    b: float = 0.,
40.                    surrogate_function: Callable = surrogate.Sigmoid(), detach_reset: bool = False, step_mode='s',
41.                    backend='torch', store_v_seq: bool = False):
42.           super().__init__(v_threshold, v_reset, v_rest, w_rest, tau_w, a, b, surrogate_function, detach_reset, step_mode,
43.                            backend, store_v_seq)
44.           self.tau_m = tau_m
45.           self.v_c = v_c
46.           self.a0 = a0
47.
48.       def neuronal_charge(self, x: torch.Tensor):
49.           delta_v = (x + self.a0 * (self.v - self.v_rest) * (self.v - self.v_c) - self.w) / self.tau_m
50.           self.v = self.v + delta_v
```

图 13: Izhikevich 神经元的代码实现示例。

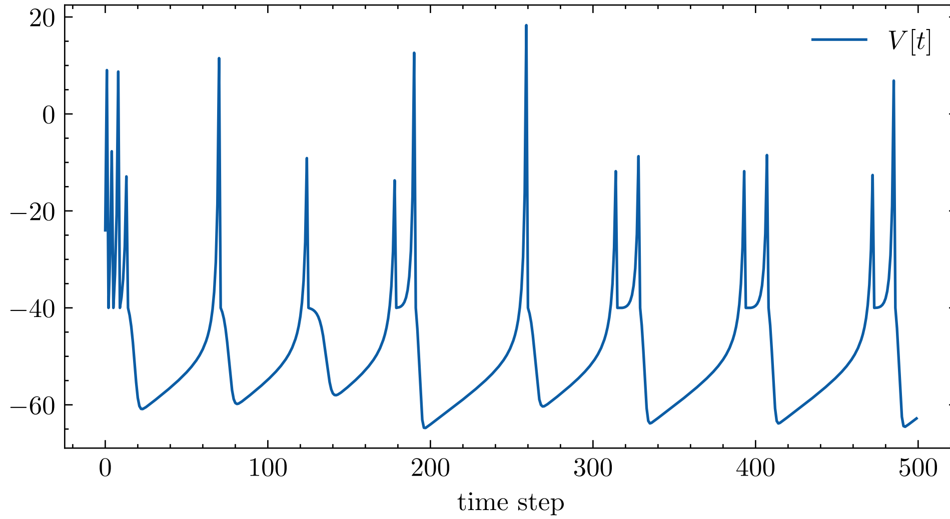

图 14: Izhikevich 神经元所建模的颤鸣型动力学。

在完成对基础神经元的适应性扩展后，实现 Izhikevich 神经元本身并不复杂。**第 36–46 行**中，我们继承这个带适应性的基础神经元，并加入 Izhikevich 神经元特有的参数，包括 τ_m、V_c 和 a_0。**第 48–50 行**按照方程 (38) 定义神经元充电函数。最终，仅用 50 行代码就完成了实现。

图 14 展示了由 Izhikevich 神经元建模得到的颤鸣型动力学，它与猫视觉皮层锥体神经元的体内记录结果相似。颤鸣型神经元也被称为快速节律性爆发（fast rhythmic bursting, FRB）神经元，在注入去极化电流时会产生高频重复爆发。所使用的超参数为 $\tau_m = 50, V_c = -40, a_0 = 1.5, V_{th} = 25, V_{reset} = -40, W_{rest} = 0, \tau_w = 100, a = \frac{100}{3}, b = 150$，输入为恒定的 $X[t] = 800$。

新学习规则的实现示例

本节将展示如何使用 SpikingJelly 实现新的学习规则，这也进一步体现了 SpikingJelly 出色的可扩展性与灵活性。

神经元动力学

在脉冲深度学习研究中，引入可学习神经元动力学以提升深度 SNN 性能的方法已被频繁报道 (*56, 129, 188*)。在 SpikingJelly 中，实现带有可学习参数的脉冲神经元是很直接的。假设我们希望实现一个广义线性神经元，其充电函数为

$$H[t] = \alpha \cdot (V[t-1] - V_{reset}) + \beta \cdot X[t], \tag{39}$$

其中，α 和 β 是可学习参数。我们还希望阈值 V_{th} 也是可学习的。该神经元的实现代码如图 15 所示。

在 **第 6–9 行**中，我们首先继承基础神经元类，并增加参数 *alpha_init*、*beta_init* 作为 α 和 β 的初始值。**第 10–11 行**调用基础神经元的构造函数，然后删除属性 *v_threshold*，因为其默认数据类型是 float，无法直接作为可学习参数。**第 13–15 行**中，我们设置三个可学习参数，即 V_{th}、α 和 β。这些参数被包装为 *torch.nn.Parameter*，因此可以自动通过梯度下降进行优化。**第 17–19 行**按照方程 (39) 定义神经元充电函数。为了确保神经元不会在没有输入时自行持续充电，我们将 α 限制在 $[0, 1]$ 范围内。**第 22–31 行**用于测试该实现：给定一个随机输入序列，执行一次前向和反向传播，然后检查这些参数的梯度。**第 32–38 行**给出了梯度结果，表明这个带有可学习 α、β 和 V_{th} 的神经元可以正常工作。

代理函数

图 16 展示了直通估计器（Straight Through Estimator, STE）(*189*) 的实现方式。STE 是量化神经网络中最常用的代理梯度函数之一。

在 **第 5–17 行**中，我们定义了 STE 的前向与反向函数：

$$y = \Theta(x), \tag{40}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \Theta(w - |x|), \tag{41}$$

```python
1.  import torch
2.  import torch.nn as nn
3.  from typing import Callable
4.  from spikingjelly.activation_based import neuron, surrogate, functional
5.  
6.  class GeneralLinearNeuron(neuron.BaseNode):
7.      def __init__(self, alpha_init: torch.Tensor, beta_init: torch.Tensor, v_threshold: torch.Tensor, v_reset: float = 0.,
8.                   surrogate_function: Callable = surrogate.Sigmoid(), detach_reset: bool = False,
9.                   step_mode='s', backend='torch', store_v_seq: bool = False):
10.         super().__init__(1., v_reset, surrogate_function, detach_reset, step_mode, backend, store_v_seq)
11.         del self.v_threshold
12. 
13.         self.v_threshold = nn.Parameter(torch.as_tensor(v_threshold))
14.         self.alpha = nn.Parameter(alpha_init)
15.         self.beta = nn.Parameter(beta_init)
16. 
17.     def neuronal_charge(self, x: torch.Tensor):
18.         torch.clamp_(self.alpha.data, 0., 1.)
19.         self.v = self.alpha * (self.v - self.v_reset) + self.beta * x
20. 
21. 
22. # example
23. alpha = torch.as_tensor(0.5)
24. beta = torch.as_tensor(0.5)
25. v_threshold = torch.as_tensor(1.)
26. net = GeneralLinearNeuron(alpha, beta, v_threshold)
27. x_seq = torch.rand([8, 4])
28. 
29. functional.multi_step_forward(x_seq, net).sum().backward()
30. for name, p in net.named_parameters():
31.     print(name, 'grad =', p.grad)
32. '''
33. outputs:
34. 
35. v_threshold grad = tensor(-9.7882)
36. alpha grad = tensor(6.2809)
37. beta grad = tensor(9.4236)
38. '''
```

图 15: 带可学习神经元动力学的广义线性神经元实现示例。

```
1.  import torch
2.  import torch.nn as nn
3.  from spikingjelly.activation_based import surrogate, neuron
4.
5.  class ste(torch.autograd.Function):
6.      @staticmethod
7.      def forward(ctx, x, width):
8.          if x.requires_grad:
9.              ctx.save_for_backward(x)
10.             ctx.width = width
11.         return surrogate.heaviside(x)
12.
13.     @staticmethod
14.     def backward(ctx, grad_output):
15.         x = ctx.saved_tensors[0]
16.         width = ctx.width
17.         return grad_output * surrogate.heaviside(width - x.abs()), None
18.
19. class STE(nn.Module):
20.     def __init__(self, width: float):
21.         super().__init__()
22.         self.width = width
23.
24.     def forward(self, x):
25.         return ste.apply(x, self.width)
26.
27. # example
28. sn = neuron.IFNode(surrogate_function=STE(width=0.5))
29. x_seq = torch.arange(9) / 4
30. x_seq.requires_grad = True
31. sn(x_seq).sum().backward()
32. print('x_seq =\n', x_seq)
33. print('x_seq.grad =\n', x_seq.grad)
34. '''
35. outputs:
36.
37. x_seq =
38.  tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000, 1.2500, 1.5000, 1.7500, 2.0000],
39.        requires_grad=True)
40. x_seq.grad =
41.  tensor([0., 0., 1., 1., 1., 1., 1., 0., 0.])
42. '''
```

图 16: 自定义直通估计器（STE）代理函数的实现示例。

其中，x 为输入，y 为输出，$\mathcal{L}$ 为损失，w 为宽度参数。**第 19–25 行**将 STE 代理函数封装为一个模块。**第 28 行**中，我们创建了一个以 STE 作为代理函数的 IF 神经元层。**第 29–33 行**中，我们向该 IF 神经元层输入 9 个元素，执行反向传播，并检查输入的梯度。需要注意的是，这里只运行了一个时间步，因此梯度可以直接计算为 $\frac{\partial \mathcal{L}}{\partial X_i} = \Theta(w - |X_i - V_{th}|)$，其中 $w = 0.5$、$V_{th} = 1$、$X_i = \frac{i-1}{4}$。**第 34–42 行**展示了梯度结果，可以看到它与公式计算完全一致。

```
1.    import torch
2.    from torch.optim.optimizer import Optimizer
3.    import math
4.
5.
6.    class GradRewiring(Optimizer):
7.        """
8.        Based on the implementation of Adam optimizer in PyTorch.
9.        """
10.
11.       def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
12.                    weight_decay=0, amsgrad=False, alpha=1e-5, s=None):
13.           if not 0.0 <= lr:
14.               raise ValueError("Invalid learning rate: {}".format(lr))
15.           if not 0.0 <= eps:
16.               raise ValueError("Invalid epsilon value: {}".format(eps))
17.           if not 0.0 <= betas[0] < 1.0:
18.               raise ValueError("Invalid beta parameter at index 0: {}".format(betas[0]))
19.           if not 0.0 <= betas[1] < 1.0:
20.               raise ValueError("Invalid beta parameter at index 1: {}".format(betas[1]))
21.           if not 0.0 <= weight_decay:
22.               raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
23.           if 0.5 > s:
24.               raise ValueError("Invalid target sparsity: {}, must larger than or equal 0.5".format(s))
25.
26.           defaults = dict(lr=lr, betas=betas, eps=eps,
27.                           weight_decay=weight_decay, amsgrad=amsgrad, alpha=alpha, s=s)
28.           super(GradRewiring, self).__init__(params, defaults)
29.
30.       def __setstate__(self, state):
31.           super(GradRewiring, self).__setstate__(state)
32.           for group in self.param_groups:
33.               group.setdefault('amsgrad', False)
```

图 17: Grad R 优化器实现示例中关于参数定义的代码部分。

基于梯度的优化器

我们以 Gradient Rewiring（Grad R）算法 (*143*) 的代码为例，说明如何在 SpikingJelly 中实现基于梯度的优化器。

Grad R 是一种剪枝算法，它维持了一个无需预训练的流程，从而尽量减小额外计算负担。作为 SpikingJelly 中可即插即用的 PyTorch 优化器，Grad R 基于 PyTorch 官方 Adam 优化器实现，仅做了少量修改。由于 SpikingJelly 中所有模块都遵循 PyTorch 风格，研究者只需简单替换原有优化器，就可以在训练过程中直接对 SNN 执行剪枝。

Grad R 优化的代码如图 17 和图 18 所示。除学习率、权重衰减等常规训练超参数外，*GradRewiring* 类还引入了目标稀疏度 s 以及稀疏先验惩罚项 α，如图 17 的 **第 11–12 行**所示。这些超参数会在图 17 的 **第 26–27 行**与其他训练超参数一起加入状态字典。

图 18 展示了该剪枝算法的核心部分。**第 37 行**开始遍历网络参数。**第 46 行**提取

所有超参数以及 Adam 的部分状态。由于 Grad R 需要保持每个权重分量的符号，初始符号 s 会在优化器初始化时的 **第 60 行**（即 **第 49–66 行**）被记录下来，并在每次训练迭代的 **第 68 行**保持固定。强度参数 θ 是一个可学习变量，用于描述训练过程中对应隐式权重的幅值。实际权重 W、初始符号 s 与推理时隐式权重 θ 之间的关系，通过 **第 108 行**的 ReLU 映射给出：

$$s = \text{Sign}(W[0]), \tag{42}$$

$$W[k] = s \cdot \max(0, \theta[k]), \tag{43}$$

其中，$W[k]$ 和 $\theta[k]$ 分别表示第 k 次训练迭代时的实际权重和隐式权重。通常，上式意味着损失函数会先对 $W[k]$ 反向传播，再传递到 $\theta[k]$。Grad R 将这一过程拆解为五个步骤：

1. 在 **第 71–94 行**中，计算典型 Adam 更新在稀疏权重情形下的步长。
2. 在 **第 96 行**中，利用初始符号 s 与隐式权重 θ 生成稠密权重。
3. 在 **第 102–103 行**中，对 θ 施加稀疏先验。
4. 在 **第 105 行**中，生成更新后的隐式权重。
5. 在 **第 108 行**中，通过 ReLU 映射得到稀疏权重。

上述 θ 的稀疏先验形式为

$$p(\theta) = \frac{\alpha}{2}\exp(-\alpha|\theta - \mu|), \tag{44}$$

其中，μ 由目标稀疏度 s 和 α 计算得到：

$$\mu = \frac{1}{\alpha}\ln(2 - 2s). \tag{45}$$

局部学习规则

像 STDP 这样的局部学习规则，可以借助 SpikingJelly 中的 monitor 高效实现。为简洁起见，这里以全连接层的 Hebbian 学习规则实现为例，其代码如图 19 所示。

```
35.     @torch.no_grad()
36.     def step(self):
37.         for group in self.param_groups:
38.             for p in group['params']:
39.                 if p.grad is None:
40.                     continue
41.                 grad = p.grad
42.                 if grad.is_sparse:
43.                     raise RuntimeError('Adam does not support sparse gradients, please consider SparseAdam instead')
44.                 amsgrad = group['amsgrad']
45.
46.                 state = self.state[p]
47.
48.                 # State initialization
49.                 if len(state) == 0:
50.                     state['step'] = 0
51.                     # Exponential moving average of gradient values
52.                     state['exp_avg'] = torch.zeros_like(p, memory_format=torch.preserve_format)
53.                     # Exponential moving average of squared gradient values
54.                     state['exp_avg_sq'] = torch.zeros_like(p, memory_format=torch.preserve_format)
55.                     if amsgrad:
56.                         # Maintains max of all exp. moving avg. of sq. grad. values
57.                         state['max_exp_avg_sq'] = torch.zeros_like(p, memory_format=torch.preserve_format)
58.
59.                     # Record initial sign
60.                     state['sign'] = torch.sign(p)
61.
62.                     # Hidden parameter theta
63.                     state['strength'] = torch.abs(p)
64.
65.                     if group['alpha'] > group['eps']:
66.                         state['center_distr'] = math.log(2. * (1. - group['s'])) / group['alpha']
67.
68.                 sgn = state['sign']
69.                 strength = state['strength']
70.
71.                 exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
72.                 if amsgrad:
73.                     max_exp_avg_sq = state['max_exp_avg_sq']
74.                 beta1, beta2 = group['betas']
75.
76.                 state['step'] += 1
77.                 bias_correction1 = 1 - beta1 ** state['step']
78.                 bias_correction2 = 1 - beta2 ** state['step']
79.
80.                 if group['weight_decay'] != 0:
81.                     grad = grad.add(p, alpha=group['weight_decay'])
82.
83.                 # Decay the first and second moment running average coefficient
84.                 exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
85.                 exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)
86.                 if amsgrad:
87.                     # Maintains the maximum of all 2nd moment running avg. till now
88.                     torch.max(max_exp_avg_sq, exp_avg_sq, out=max_exp_avg_sq)
89.                     # Use the max. for normalizing running avg. of gradient
90.                     denom = (max_exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(group['eps'])
91.                 else:
92.                     denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(group['eps'])
93.
94.                 step_size = group['lr'] / bias_correction1
95.
96.                 p_hidden = sgn * strength
97.
98.                 # Gradient term
99.                 p_hidden.addcdiv_(exp_avg, denom, value=-step_size)
100.
101.                # Prior term
102.                if group['alpha'] > group['eps']:
103.                    p_hidden.addcmul_((strength - state['center_distr']).sign(), sgn, value=-group['alpha'] * step_size)
104.
105.                state['strength'] = p_hidden * sgn
106.
107.                # Prune those connection changed their signs
108.                p.data = state['strength'].clamp(min=0.0).mul(sgn)
```

图 18: Grad R 优化器实现示例中的核心算法部分。

在 **第 5–7 行**中，我们定义构造函数，其参数包括步进模式、突触层以及脉冲神经元。**第 8 行**检查突触层，确保其为全连接层。**第 10–13 行**设置属性以及 monitor，用于记录输入脉冲和输出脉冲。**第 15–17 行**定义 reset 函数，用于清空 monitor 中记录的数据。**第 19 行**定义 *step* 函数，用以计算权重 W 的 ΔW。在 **第 20–36 行**中，我们计算单步模式下输入脉冲与输出脉冲的发放率；在 **第 38–54 行**中，则计算多步模式下的发放率。**第 56–58 行**中，按照

$$\Delta W[i][j] = scale \cdot fr_{pre}[i] \cdot fr_{post}[j], \tag{46}$$

生成 ΔW，其中 $fr_{pre}[i]$ 是第 i 个输入脉冲的发放率，$fr_{post}[j]$ 是第 j 个输出脉冲的发放率，$scale$ 是缩放因子。这里借助广播机制高效完成计算。**第 59–65 行**中，当 on_grad 为 true 时，将 ΔW 加到全连接层权重的梯度上；否则，该函数直接返回 ΔW。

图 20 展示了图 19 中 Hebbian learner 的使用方式。为了让输出神经元更容易发放，我们在 **第 74 行**将 V_{th} 设为 0.1。由于 Hebbian learner 不依赖 autograd 机制，因此在**第 77 行**中关闭 autograd。**第 78–79 行**中，我们向该 SNN 输入随机脉冲；需要注意的是，Hebbian learner 已在 SNN 前向传播过程中同步记录了所需数据。**第 81 行**调用 *step* 函数以获得并打印 ΔW，对应的结果显示在 **第 85–88 行**。

加速方法的消融研究

为了说明 SpikingJelly 中各类加速方法的作用，我们在 Spiking ResNet-18 上设计了消融实验。代理学习训练阶段采用 $T = 2, 4, 8, 16, 32$，推理阶段采用 $T = 128$，这一设置也与图 **1d** 中的实验选项保持一致。图 21 的实验结果可概括如下：

- 对于逐步传播模式，JIT 在推理阶段带来的加速非常明显；但在训练阶段，其效果会随具体情况而变化。对逐步传播模式而言，JIT 的加速比例较小；在逐层传播模式下，它甚至可能略微拖慢速度。不过，当逐层传播模式对无状态层启用了“时间步维度并入批次维度”后，JIT 仍可带来约 10% 的加速。

```
1.  import torch
2.  import torch.nn as nn
3.  from spikingjelly.activation_based import base, monitor, neuron, layer
4.
5.  class FCLayerHebbianLearner(base.StepModule):
6.      def __init__(self, step_mode: str, synapse: nn.Linear, sn: neuron.BaseNode):
7.          super().__init__()
8.          assert isinstance(synapse, nn.Linear)
9.
10.         self.step_mode = step_mode
11.         self.synapse = synapse
12.         self.in_spike_monitor = monitor.InputMonitor(synapse)
13.         self.out_spike_monitor = monitor.OutputMonitor(sn)
14.
15.     def reset(self):
16.         self.in_spike_monitor.clear_recorded_data()
17.         self.out_spike_monitor.clear_recorded_data()
18.
19.     def step(self, on_grad: bool = True, scale: float = 1.):
20.         if self.step_mode == 's':
21.             T = self.in_spike_monitor.records.__len__()
22.             N = self.in_spike_monitor.records[0].shape[0]
23.
24.             # data in recodes have a shape of [N, C_in]
25.             spikes_in = 0
26.             for item in self.in_spike_monitor.records:
27.                 spikes_in += item.sum(0)
28.
29.             fr_in = spikes_in / T / N
30.
31.             # data in recodes have a shape of [N, C_out]
32.             spikes_out = 0
33.             for item in self.out_spike_monitor.records:
34.                 spikes_out += item.sum(0)
35.
36.             fr_out = spikes_out / T / N
37.
38.         elif self.step_mode == 'm':
39.             T = 0
40.             N = self.in_spike_monitor.records[0].shape[0]
41.             # data in recodes have a shape of [t, N, C_in]
42.             spikes_in = 0
43.             for item in self.in_spike_monitor.records:
44.                 spikes_in += item.sum([0, 1])
45.                 T += item.shape[0]
46.
47.             fr_in = spikes_in / T / N
48.
49.             # data in recodes have a shape of [t, N, C_out]
50.             spikes_out = 0
51.             for item in self.out_spike_monitor.records:
52.                 spikes_out += item.sum([0, 1])
53.
54.             fr_out = spikes_out / T / N
55.
56.         # fr_in.shape = [C_in], fr_out.shape = [C_out]
57.         # delta_w.shape = [C_out, C_in]
58.         delta_w = - fr_in.view(-1, 1) * fr_out.view(1, -1) * scale
59.         if on_grad:
60.             if self.synapse.weight.grad is None:
61.                 self.synapse.weight.grad = delta_w
62.             else:
63.                 self.synapse.weight.grad -= delta_w
64.         else:
65.             return delta_w
```

图 19: Hebbian learner 的实现示例。

```
67.   # example
68.   C_in = 3
69.   C_out = 3
70.   T = 64
71.   N = 32
72.   net = nn.Sequential(
73.       layer.Linear(C_in, C_out),
74.       neuron.IFNode(v_threshold=0.1)
75.   )
76.   hb_learner = FCLayerHebbianLearner(step_mode='m', synapse=net[0], sn=net[1])
77.   with torch.no_grad():
78.       x_seq = (torch.rand([T, N, C_in]) > 0.2).float()
79.       y_seq = net(x_seq)
80.
81.       print('delta_w =\n', hb_learner.step(on_grad=False))
82.   '''
83.   outputs:
84.
85.   delta_w =
86.    tensor([[-0.2004, -0.2004, -0.1576],
87.           [-0.2015, -0.2015, -0.1584],
88.           [-0.1965, -0.1965, -0.1545]])
89.   '''
```

图 20: 图 19 中 Hebbian learner 的使用示例。

- “MergeTB” 和 “CuPy” 对逐层传播模式至关重要。它们分别加速无状态层和有状态层。仅当 T 很小、小型 CUDA 内核调用开销还不明显时，使用 “CuPy” 才会比只使用 “JIT” 更慢。
- 在逐层传播模式下，同时启用 “MergeTB” 与 “CuPy”，在大多数情况下都能获得最快的训练仿真速度。

在 SpikingJelly 中，只需将所有模块设置为多步模式，并为脉冲神经元启用 CuPy 后端，就能开启 “LBL + MergeTB + CuPy”，从而最大化训练效率。

SNN 框架比较

框架概览

表 2 给出了一个较为清晰的 SNN 框架比较，可视为图 1e 的补充说明。

模块复杂度在 NEURON、NEST 和 Brian2 中较高，因为这些框架的目标是面向神经科学，对真实生物神经系统进行模拟。例如，NEURON 中的神经元通常包含大量参数，如胞体属性、形态属性和离子通道，神经元动力学中也涉及多个常微分方程。Nengo 和 BindsNet 在神经科学与计算机科学之间取得了一种折中，采用了复杂度适中的模型。

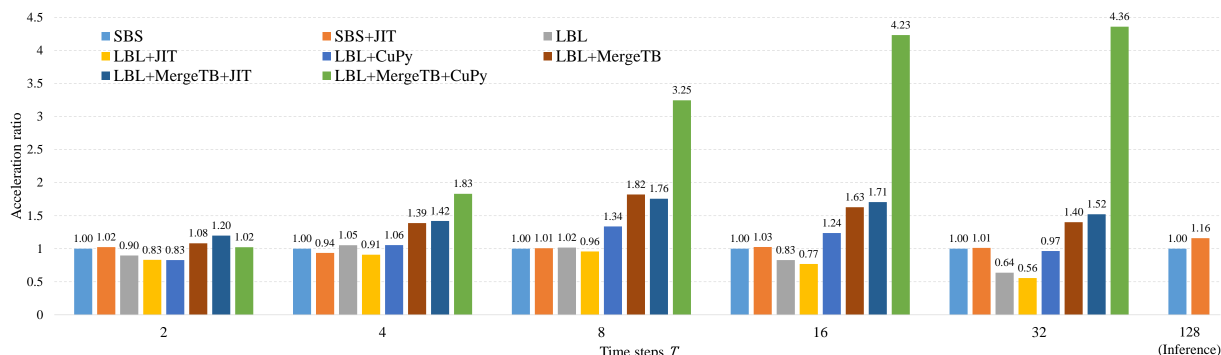

图 21: **以朴素实现（“SBS”）为基线时，不同方法带来的加速比。**“SBS” 表示使用单步模式，对应逐步传播模式；“LBL” 表示使用多步模式，对应逐层传播模式；“CuPy” 表示脉冲神经元启用 CuPy 后端；“JIT” 表示脉冲神经元启用 JIT；“MergeTB” 表示对无状态层将时间步维度并入批次维度。需要注意，大时间步推理（$T = 128$）只使用单步模式，这也是 ANN2SNN 的典型用法。

相比之下，SpikingJelly、Norse 和 SNNTorch 使用的神经元与突触复杂度都较低。举例来说，Nengo 中会考虑 LIF 神经元的不应期，而 SpikingJelly 为了追求更快的仿真速度和更高的训练精度，则省略了这一机制。

GPU 支持主要取决于其 **GPU 后端**。传统框架如 NEURON、NEST、Brian2 和 Nengo 被提出时，GPU 还不是高性能计算的核心角色，因此这些框架本身并不原生支持 GPU。后续虽然发展出了表 2 中所示的若干 GPU 子包，用于增强母框架能力，但这些子包无法覆盖母框架的全部功能，因此 GPU 支持并不完善。BindsNet、SpikingJelly、Norse 和 SNNTorch 都基于 PyTorch，而 PyTorch 中几乎所有模块和函数都兼容 GPU，并且良好支持多 GPU 分布式训练。因此，这些框架可以充分利用 GPU 来加速仿真。除此之外，SpikingJelly 还引入了 CuPy 后端，通过 CUDA 级别的加速获得了明显高于朴素 PyTorch 模块的效率。

神经形态支持并不是某些框架的主要目标。SpikingJelly 集成了一套全栈的神经形态数据集处理方案，并提供了 9 个数据集。Norse 只集成了 1 个数据集，而 SNNTorch 提供了 3 个。至于神经形态芯片支持，只有 Brian2、Nengo 和 SpikingJelly 提供了相应

子包。其中，NengoLoihi 是目前较为成熟的 Loihi 支持包，因此 Nengo 对 Loihi 的兼容性最好。

突触可塑性指的是利用 STDP 等生物学上合理的学习规则来修改突触权重，这方面 Brian2、Nengo 和 BindsNet 具有明显优势。需要说明的是，尽管 NEURON 没有内建可塑性机制，但根据 NEURON 论坛维护者的建议，用户仍然可以较容易地在 NEURON 中自行实现这类规则。而在深度 SNN 中，修改突触权重的主要方式通常是基于梯度的优化或基于转换的方法，因此许多脉冲深度学习框架都会忽略突触可塑性。在这些框架中，只有 SpikingJelly 集成了生物合理学习规则。

深度学习支持是 SpikingJelly、Norse 和 SNNTorch 最关心的方向。它们都支持代理学习方法，而 SpikingJelly 还进一步集成了 ANN2SNN 方法。值得一提的是，Nengo 也通过其子包 NengoDL 支持 ANN2SNN；不过它是使用 LIF 神经元响应来近似 ANN 中的 ReLU，这与许多采用 IF 神经元的 ANN2SNN 工作并不完全兼容。

论坛包括论坛帖子数（若该框架有论坛）以及 GitHub 上的 issue 与 pull request 数量。**论文数**指相关学术论文数量。**论坛**和 **论文数**分别反映了该框架的用户规模与学术研究者规模，也从侧面体现了社区活跃度。正如 **发布时间**所示，NEURON、NEST、Brian2 和 Nengo 等经典框架已经发展多年，拥有很高声誉和大规模社区。BindsNet、SpikingJelly、Norse 和 SNNTorch 则属于较年轻的一代，当前社区规模仍小于经典框架。不过，随着脉冲机器学习研究兴趣的快速增长，它们的社区也在迅速壮大。比如，SpikingJelly 尽管在 2019 年才发布，但三年内就已经积累了 94+ 篇论文，已经接近一些经典框架的规模。

SpikingJelly、Norse 与 SNNTorch 的差异

除表 2 中给出的整体比较外，这三个深度学习框架之间还存在一些更细微的差异。

状态管理 SpikingJelly 将膜电位 $V[t]$ 等状态保存在模块内部。如果一个模块含有状态，则它是有状态模块；否则就是无状态模块。例如，在 SpikingJelly 中，LIF 神经元

	模块复杂度	GPU 支持	GPU 后端	神经形态支持	突触可塑性	深度学习支持	社区	论文数	发布时间
NEURON	★★★★★	★★	CoreNEURON				18000+ 帖子	2672+	1984
NEST	★★★	★	NEST GPU		★★★		1200+ issues/prs	668+	1994
Brian2	★★★★	★★	Brian2GENN	Brian2Loihi	★★★		3000+ 帖子 800+ issues/prs	100+	2013
Nengo	★★	★★★	NengoOCL TensorFlow	NengoLoihi	★★	ANN2SNN	7200+ 帖子 800+ issues/prs	100+	2003
BindsNet	★★	★★★★	PyTorch		★★		630+ issues/prs	90+	2018
SpikingJelly	★	★★★★★	PyTorch CuPy	9 个数据集 LavaExchange LynxiExchange	★	代理学习 ANN2SNN	390+ issues/prs	94+	2019
Norse	★	★★★★	PyTorch	1 个数据集		代理学习	360+ issues/prs	3+	2019
SNNTorch	★	★★★★	PyTorch	3 个数据集		代理学习	170+ issues/prs	45+	2020

表 2: SNN 框架比较

是有状态模块，而线性层是无状态模块。相比之下，Norse 将状态视为模块输入/输出的一部分；SNNTorch 则提供选项来控制是否存储状态。将状态存放在模块内部，会使构建和训练 SNN 更加方便，因为状态由模块自行管理。从外部视角看，这些有状态模块只需要接收前一层输入并向后一层输出，因此可以很容易地与无状态模块协同工作，例如可以直接与 PyTorch 中的 *torch.nn.Sequential* 一起封装构建深层 SNN。而在 Norse 中，状态不保存在模块内部时，通常需要额外使用 *norse.torch.SequentialState* 这样的包装器。对于内部存储状态的有状态模块，额外需要做的唯一操作，就是在处理下一组新输入前重置状态，而这可以通过对整个网络施加 reset 函数轻松完成。

步进模式 SpikingJelly 中的模块可以在单步模式和多步模式之间切换，分别处理单个时间步或多个时间步的数据。基于步进模式，又可以进一步派生出逐步传播和逐层传播两种模式，从而灵活适应不同场景。Norse 和 SNNTorch 中则没有显式定义步进模式和传播模式这两个概念。Norse 中大多数模块采用多步模式，少数模块采用单步模式。它提供了包装器 *Lift*，用于将来自 PyTorch 的单步模块（如无状态卷积层）包装后工作在多步模式下。Norse 中的 *Lift* 与 SpikingJelly 中用于包装单步有状态模块以支持多步模式的 *MultiStepContainer* 行为相似。而 SpikingJelly 对无状态层使用的是 *SeqToANNContainer*，它通过合并时间步维度与批次维度来执行计算，速度明显快于 *Lift* 和 *MultiStepContainer*。Norse 默认采用逐层传播模式，但在使用单步模块时也可以表现为逐步传播模式。SNNTorch 中所有模块都工作在单步模式，因此其传播模式是

逐步传播。

神经元内核细节

Algorithm 3: IF 神经元的 BPTT

Require:

批量大小 N // `N`

时间步数 T // `numel` = $N \cdot T$

所有时间步输出脉冲的梯度 $\{\frac{\partial \mathcal{L}}{\partial S[t]}\}, t = 0, 1, ..., T-1$ // `grad_spike_seq`

最后一个时间步复位后膜电位的梯度 $\frac{\partial \mathcal{L}}{\partial V[T-1]}$ // `grad_v_seq`

所有时间步充电后膜电位 $\{H[t]\}, t = 0, 1, ..., T-1$ // `h_seq`

阈值电位 V_{th} // `v_th`

复位电位 V_{reset} // `v_reset`

Outputs:

所有时间步输入的梯度 $\{\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}X[t]}\}, t = 0, 1, ..., T-1$ // `grad_x_seq`

初始时刻复位后膜电位的梯度 $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}V[-1]}$ // `grad_v_init`

$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[T]} = 0$ // `grad_h = 0`

for $t \leftarrow T-1, T-2, ..., 0$

$\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]} = 1$ // `grad_h_next_to_v`

$\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]} = 1$ // `grad_h_to_x`

$\frac{\mathrm{d}S[t]}{\mathrm{d}H[t]} = \sigma'(H[t] - V_{th})$ // `surrogate gradient`

$\frac{\mathrm{d}V[t]}{\mathrm{d}H[t]} = 1 - S[t] + (-H[t] + V_{reset}) \cdot \frac{\mathrm{d}S[t]}{\mathrm{d}H[t]}$

$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t]} = \frac{\partial \mathcal{L}}{\partial S[t]} \cdot \frac{\mathrm{d}S[t]}{\mathrm{d}H[t]} + (\frac{\partial \mathcal{L}}{\partial V[t]} + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t+1]} \cdot \frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}) \cdot \frac{\mathrm{d}V[t]}{\mathrm{d}H[t]}$

$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}X[t]} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[t]} \cdot \frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}$

$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}V[-1]} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}H[0]} \cdot \frac{\mathrm{d}H[0]}{\mathrm{d}V[-1]}$

半自动 CUDA 代码生成方法是 SpikingJelly 中一项非常实用的技术。该技术建立在 CuPy 之上。CuPy 是一个兼容 NumPy/SciPy 并支持 GPU 加速的数组库，同时提供了对底层 CUDA 特性的访问能力。CuPy 能够对由 Python 字符串生成的 CUDA 代码进行编译和运行。SpikingJelly 充分利用了这一特性，通过 Python 中的 if-else 语句生成不同的 CUDA 代码，从而将 CUDA 编写过程模块化，并显著降低开发成本。由于数组库之间具有兼容性，PyTorch 等 NumPy 风格库中的张量可以直接调用由 CuPy 编译出的 CUDA 内核，而无需在 PyTorch、CuPy 与 CUDA 数组之间复制或搬移数据。在

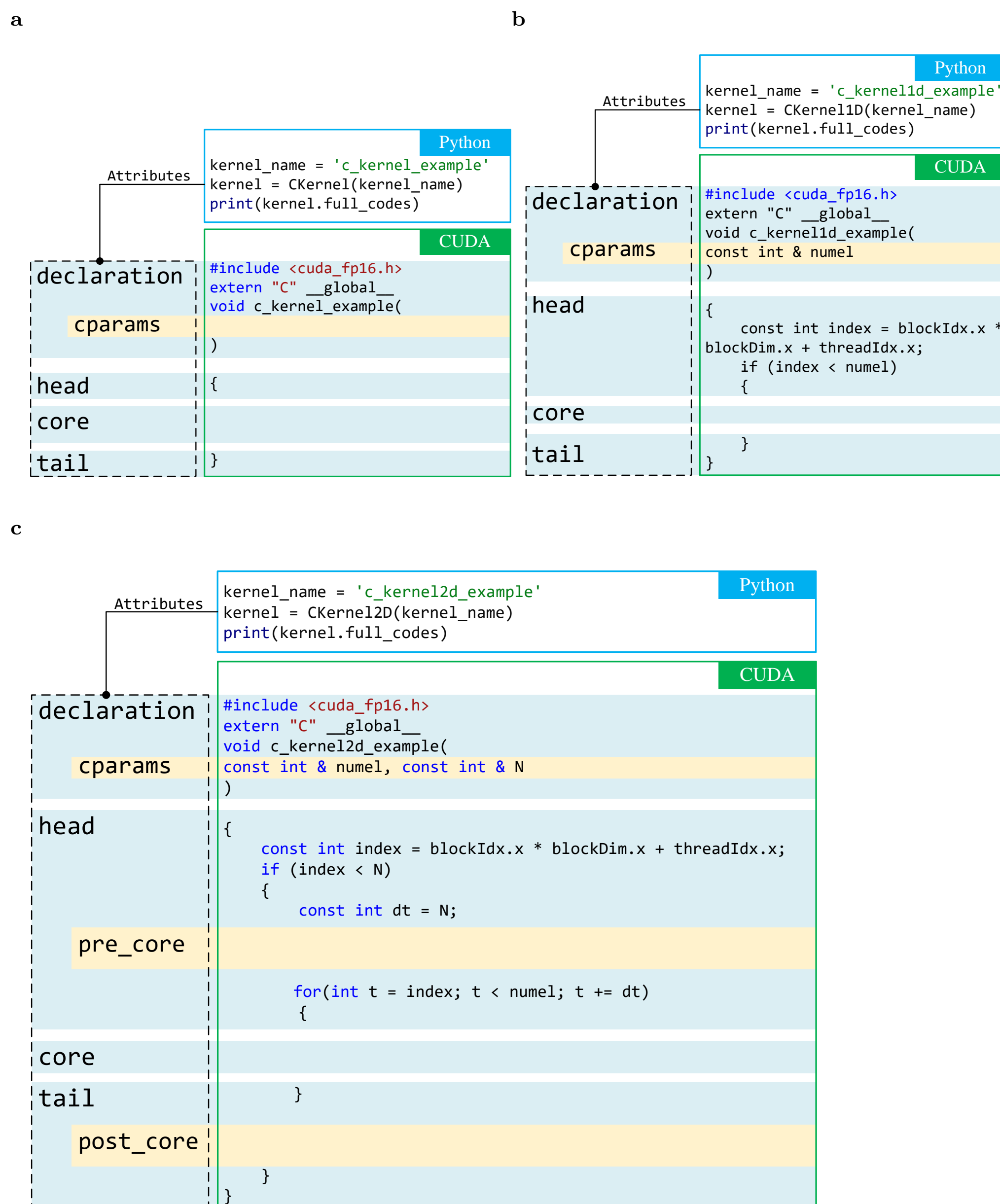

图 22: **由 Python 类 *CKernel*、*CKernel1D* 和 *CKernel2D* 生成的 CUDA 内核。** **a**. *CKernel* 具有 *declaration*、*head*、*core* 和 *tail* 等属性。*declaration* 由属性 *cparams* 生成，而在 *CKernel* 中，*cparams* 默认是一个空的 Python 字典。**b**. *CKernel1D* 从 *CKernel* 扩展而来，用于处理 1 维张量上的逐元素运算。*cparams* 中的默认项为 *numel*，表示元素个数。**c**. 在 *CKernel* 的基础上，*CKernel2D* 通过增加一个额外的 for 循环来处理 2 维张量。*cparams* 中的默认项包括 *numel*（所有元素总数）以及 *N*（单个时间步上的元素数）。为了实现更细粒度的代码控制，又额外引入了 *pre_core* 和 *post_core* 属性，分别用于在 *core* 前后插入 CUDA 代码。

a

Python

```python
from spikingjelly.activation_based.auto_cuda import cfunction, base
import torch
dtype = 'float'
kernel = base.CKernel1D(kernel_name='heaviside')
kernel.add_param(ctype=f'const {dtype} *', cname='x')
kernel.add_param(ctype=f'{dtype} *', cname='y')
kernel.core = cfunction.heaviside(y='y[index]', x='x[index]', dtype=dtype)
print(kernel.full_codes)
```

CUDA

```cuda
#include <cuda_fp16.h>
extern "C" __global__
void heaviside(
const int & numel, const float * x, float * y
)

{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < numel)
    {
        y[index] = x[index] >= 0.0f ? 1.0f: 0.0f;
    }
}
```

b

Python

```python
x = torch.rand([4], device='cuda:0') - 0.5
y = torch.zeros_like(x)
kernel.simple_call(x=x, y=y)
print(f'x={x}')
print(f'y={y}')
```

Output

```
x=tensor([ 0.2790, -0.3608,  0.2219,  0.1362],
         device='cuda:0')
y=tensor([1., 0., 1., 1.], device='cuda:0')
```

c

Python

```python
cfunction.heaviside(y='y', x='x', dtype='float')
```

CUDA

```cuda
y = x >= 0.0f ? 1.0f: 0.0f;
```

PyThon

```python
cfunction.heaviside(y='y', x='x', dtype='half2')
```

CUDA

```cuda
y = __hgeu2(x, __float2half2_rn(0.0f));
```

图 23: **使用 *CKernel1D* 的示例。** **a**. 实现 Heaviside 函数的 Python 代码及其生成的 CUDA 代码。用户只需通过 *add_function* 定义函数参数，并通过设置 *core* 属性给出核心代码。这些 Python 代码以不同颜色标出，并控制相同颜色对应的 CUDA 代码。**b**. 在 Python 环境中执行 **a** 中 CUDA 内核的示例。**c**. *cfunction.heaviside* 在不同数据类型下的示例及其生成的 CUDA 代码。*cfunction* 提供用于生成 CUDA 代码的 Python 函数，同时支持 *float* 和 *half2* 两种数据类型。通过切换 *dtype*，用户可以很方便地获得不同数据类型对应的 CUDA 代码。

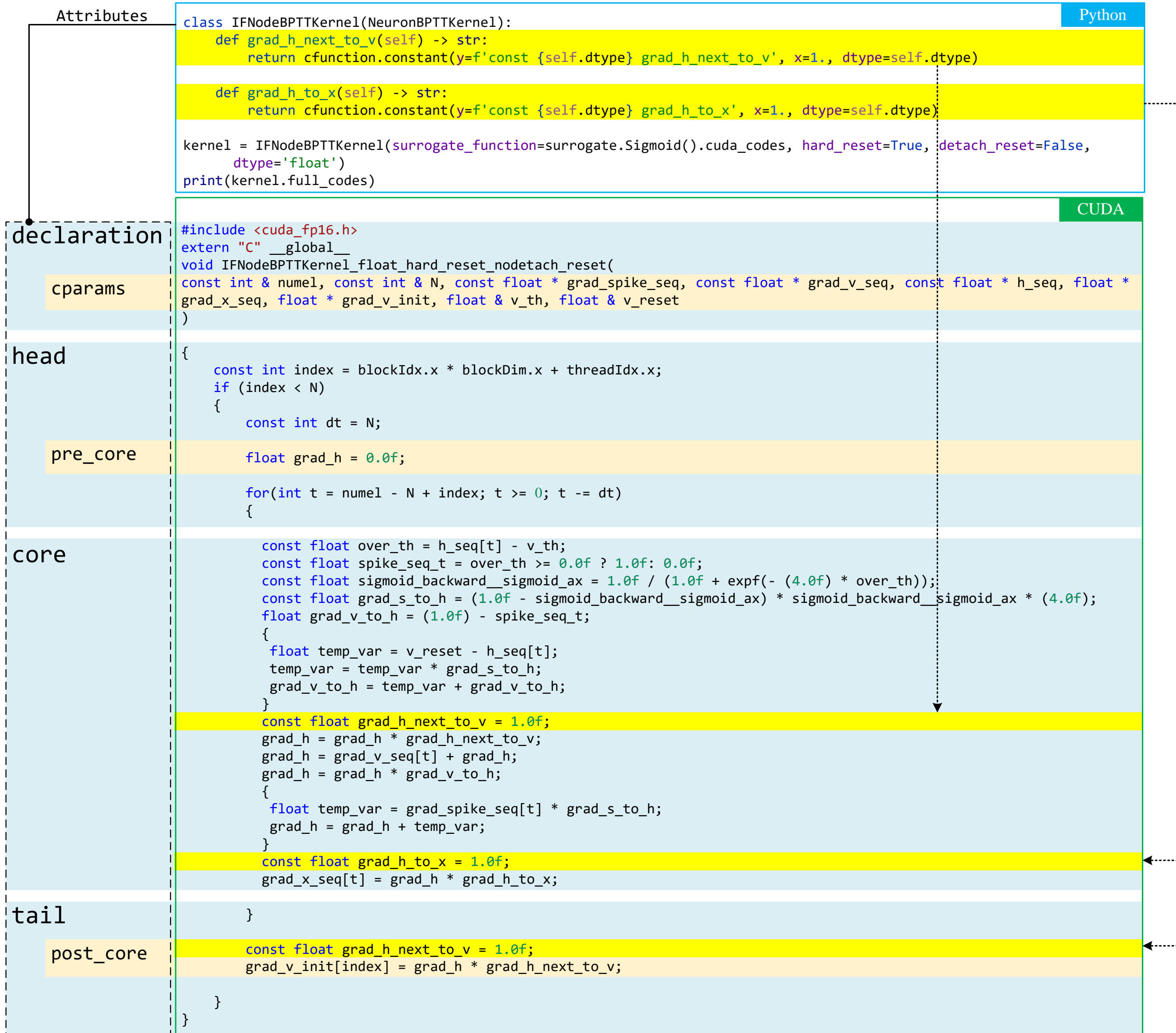

图 24: **IF 神经元反向 CUDA 内核类 *IFNodeBPTTKernel* 的 Python 代码及其生成的 CUDA 代码。**这些代码实现了算法 3 所描述的 IF 神经元 BPTT。基于 *NeuronBPTTKernel*，*IFNodeBPTTKernel* 只需分别通过 $grad_h_next_to_v$ 和 $grad_h_to_v$ 函数补全 $\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}$ 与 $\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}$。这些部分在 Python 代码中以黄色高亮标出，而由其控制生成的对应 CUDA 代码也同样以黄色高亮显示。

SpikingJelly 中，GPU 上的张量会先被整理为连续内存布局，然后其首元素地址被视为指针传入 CUDA 内核。最终，这些张量就可以被当作 CUDA 中的数组直接执行对应内核。相比之下，标准 PyTorch CUDA 扩展的工作流通常需要配置 CUDA 开发环境、编写 CUDA 代码、编写 C++ 包装层、设置并加载 C++/CUDA 文件。而在 SpikingJelly 中，由于采用了 CuPy，只需编写 CUDA 代码，其他繁琐步骤都可以省去；再结合半自动 CUDA 代码生成技术，CUDA 扩展的开发与使用成本都被大幅降低。

SpikingJelly 中的内核基类 *CKernel* 具有 *declaration*、*head*、*core* 和 *tail* 等属性，它们都以字符串形式存在，并用于以拆分方式拼接出完整的 CUDA 代码。*declaration* 由 *cparams* 属性生成，而在 *CKernel* 中，*cparams* 默认是一个空的 Python 字典。图 22**a** 展示了 *CKernel* 生成的 CUDA 代码。*CKernel1D* 在 *CKernel* 基础上扩展而来，用于处理 1 维张量上的逐元素操作；其 *cparams* 中的默认参数为 *numel*，表示元素总数。图 22**b** 给出了 *CKernel1D* 及其生成 CUDA 代码的示例。为了处理形状为 (T, N) 的序列数据，*CKernel2D* 在设计上增加了一个额外的 for 循环，如图 22**c** 所示。此时 *cparams* 中默认包含 *numel*（全部元素个数）和 N（单个时间步内的元素数）。为了实现更细致的代码控制，又额外引入了 *pre_core* 和 *post_core* 属性，分别用于在 *core* 前后添加 CUDA 代码。例如，在 for 循环中使用的临时变量，就可以在 *pre_core* 中完成初始化。

图 23 展示了使用 *CKernel1D* 的一个例子。若要实现一个逐元素 CUDA 内核，用户只需通过调用 Python 函数 *add_param* 添加函数参数，例如输入和输出张量，再通过设置属性 *core* 给出核心 CUDA 代码。在 *CKernel1D* 中，CUDA 线程索引与元素索引是相同的，也就是 CUDA 代码中的 *index*。因此，用户可以直接利用 *[index]* 来实现逐元素操作。图 23**a** 展示了实现 Heaviside 函数所需的 Python 代码及其对应生成的 CUDA 代码。图中为添加函数参数和设置核心 CUDA 代码所写的 Python 代码都用颜色标出，并控制了相同颜色对应的 CUDA 代码。图 23**b** 展示了如何执行该内核。可以看到，整个执行流程完全在 Python 环境中完成，从而减少了将 C++/CUDA 绑定到 Python 所需的额外工作量。图 23**c** 则给出了 *cfunction.heaviside* 在不同数据类型下的示例及其对

应生成的 CUDA 代码。*cfunction* 是 SpikingJelly 的一个子包，提供用于生成 CUDA 代码的函数，同时支持 *float* 和 *half2* 数据类型。通过切换 *dtype*，用户可以方便地获得不同数据类型下的 CUDA 代码，而无需分别为两种数据类型手写内核。

与 *CKernel1D* 相比，*CKernel2D* 主要增加了一个与时间步相关的 for 循环，其余部分基本相同。在 *CKernel2D* 基础上，可以很容易地实现脉冲神经元 FPTT/BPTT 所需的内核。图 24 以 IF 神经元为例，展示了如何通过反向内核 *IFNodeBPTTKernel* 实现算法 3 中给出的 BPTT。其基类 *NeuronBPTTKernel* 继承自 *CKernel2D*，并已经定义好了 *declaration*、*head*、*pre_core*、*core*、*post_core* 和 *tail* 等 CUDA 代码部分。正如图 24 中黄色高亮的 CUDA 代码所示，*core* 与 *post_core* 中预留了三个占位位置，用于定义 $\frac{\mathrm{d}H[t+1]}{\mathrm{d}V[t]}$ 和 $\frac{\mathrm{d}H[t]}{\mathrm{d}X[t]}$。这些方程由子类中的 *grad_h_next_to_v* 和 *grad_h_to_v* 函数来补全，这一点也在图 24 中以黄色高亮的 Python 代码展示了出来。

交换模块

在神经形态芯片上运行 SNN 是一种资源受限任务。例如，Lynxi KA200 的典型精度为 16 bit，而 Loihi 的典型精度为 8 bit。多数情况下，神经形态芯片都配有各自的部署工具链，例如 Lynxi 芯片使用 lyngor，Loihi 使用 Lava 与 NxSDK。为了兼容这些工具链，SpikingJelly 定义了交换模块（exchange modules），即既支持在 SpikingJelly 中运行、又支持转换到目标工具链的模块。针对每种芯片工具链，都会定义一个对应的 exchange 子包，例如面向 Lynxi KA200 的 *lynxi_exchange*，以及面向 Loihi 的 *lava_exchange*。交换模块在行为上与目标工具链中的对应模块保持一致。例如，*lava_exchange* 中的 *CubaLIFNode* 对应 Lava-DL 框架中的 *slayer.neuron.cuba.Neuron*。这种行为一致性保证了：使用 SpikingJelly 中交换模块构建的 SNN，可以无损转换到目标工具链中，进而部署到目标芯片上。

图 25**a** 展示了面向 Intel Loihi 的工作流程。Loihi 的官方软件框架是 Lava，它能够将 SNN 编译到芯片上。Lava-DL 是 Lava 的一个子包，基于 PyTorch 构建，只要 SNN

由 *Block* 模块组成，就能够在 CPU/GPU 和 Loihi 上保持一致行为并支持量化 SNN 的构建。因此，SpikingJelly 提供了对应的 *BlockContainer* 模块，其中包含量化后的突触与神经元，并与 Lava-DL 中的 *Block* 具有相同行为。由 *BlockContainer* 构建的 SNN 可以被转换为 Lava-DL 格式的 SNN，并在相同输入下产生一致输出。

图 25**b** 展示了面向 Lynxi KA200 的工作流程。为了让由 SpikingJelly 构建的 SNN 能在 KA200 上运行，需要先将突触和神经元转换为交换模块中定义的简化版本，仅保留其核心功能。例如，由于 CuPy 仅支持 GPU，因此相关 CuPy 功能会被移除。随后，使用这些简化突触和神经元构建的 SNN 就能够与 KA200 兼容，并被编译为模型文件，其中包括用于定义网络结构的 JSON 文件以及用于恢复网络权重的 BIN 文件。最后，KA200 通过加载这些模型文件重建 SNN，并执行推理过程。

神经形态数据集

最常用的神经形态数据集已集成到 SpikingJelly 中，包括 ASL-DVS (*172*), CIFAR10-DVS (*173*), DVS 手势 (*55*), ES-ImageNet (*99*), HARDVS (*174*), N-Caltech101 (*98*), N-MNIST (*98*), Nav Gesture (*175*) 和 Spiking Heidelberg Digits（SHD）(*176*)。图 26 展示了这些神经形态视觉数据集的可视化结果。

典型应用详情

基于固定帧数的积分方法 (*56*) 可以表示为：

$$\mathbf{F}[j,p,y,x] = \sum_{i=j_l}^{j_r-1} \mathcal{I}_{p,x,y}(p_i,y_i,x_i), \tag{47}$$

其中 j_l 和 j_r 是由特定分割方法创建的索引。

应用 a 在该应用中，我们首先使用 SpikingJelly 中的 *DVS128Gesture* 数据集类处理 IBM DVS128 Gesture 数据集的原始事件数据。原始事件按照方程 (17) 被切片并积分为 $T = 64$ 帧。目前，Lava 中的模块仅支持二值脉冲作为输入，因此我们进一步将这些

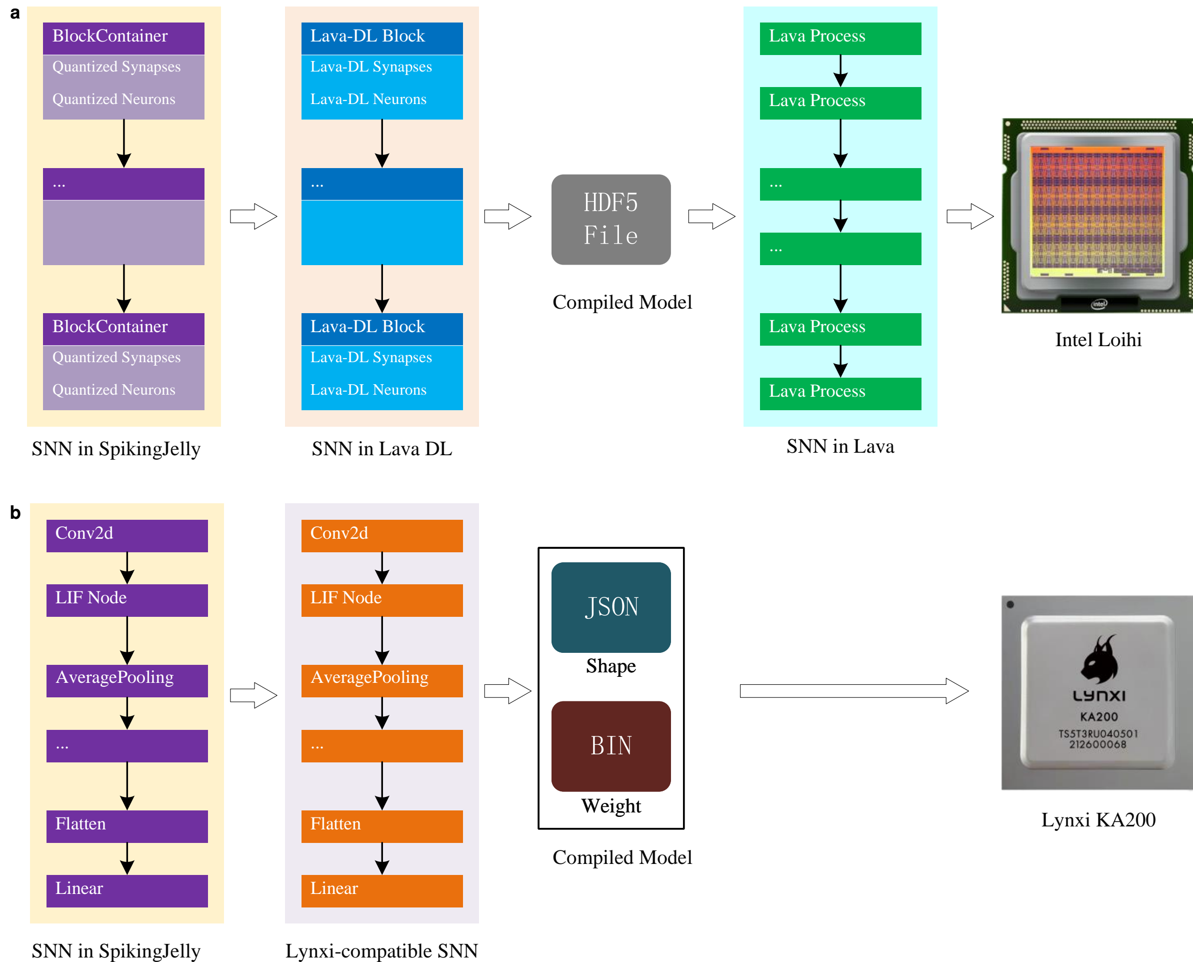

图 25: **借助 SpikingJelly 将 SNN 部署到神经形态芯片上的工作流程。** **a**. 面向 Intel Loihi 的流程。由 SpikingJelly 的 *BlockContainer* 构建的 SNN 可以转换为 Lava-DL 框架格式的 SNN，Lava-DL 是 Loihi 芯片的官方框架。之后，该 SNN 即可进入标准部署流程，包括从 Lava-DL 导出为 HDF5 文件、在 Lava 中从 HDF5 重建 SNN，以及将 SNN 编译到 Loihi。**b**. 面向 Lynxi KA200 的流程。由 SpikingJelly 构建的 SNN 可以通过将突触与神经元替换为它们的简化版本，转换为兼容 Lynxi 的 SNN。该兼容版本随后可被编译为模型文件，之后 KA200 即可通过加载这些模型文件重建 SNN 并执行推理。

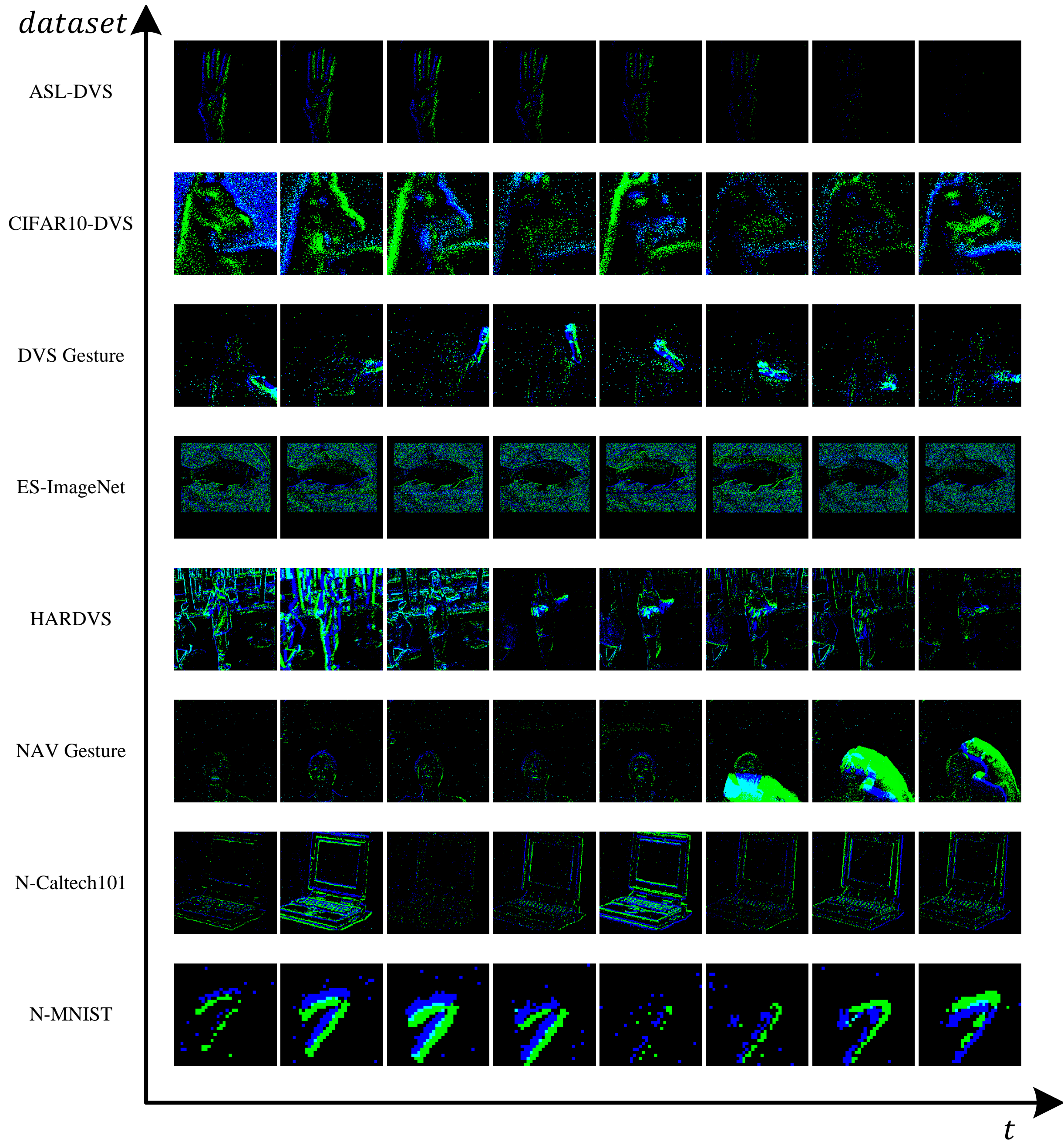

图 26: **集成到 SpikingJelly 中的神经形态视觉数据集可视化。**我们从每个数据集中选择一个样本，并将事件下采样为 8 帧。

帧二值化，即把所有非零值置为 1。同时，我们将帧大小从 128×128 下采样到 64×64；实验观察表明，这样能够获得更高的训练精度。批量大小设置为 16。最终，SNN 的输入张量形状为 $(T, N, C, H, W) = (64, 16, 2, 64, 64)$。

网络结构为 $\{C32k3s1 - LIF - C32k2s2 - LIF\} * 3 - FC11 - LIF$。其中，$C32k3s1$ 是具有 32 个通道、卷积核大小为 3、步长为 1 的卷积层；$C32k2s2$ 是具有 32 个通道、卷积核大小为 2、步长为 2 的卷积层；LIF 表示 LIF 神经元层；$*3$ 表示该结构重复 3 次；$FC11$ 表示具有 11 个输出特征的全连接层。需要说明的是，我们采用了量化感知训练，所有突触和神经元动力学都被量化为 8 bit。

我们使用 Adam 优化器，学习率为 0.005，训练轮数为 128，并采用余弦退火学习率调度 (*190*)，其中 $T_{max} = 128$。该 SNN 在测试集上达到了 85.42% 的准确率。

该 SNN 的所有层都由 SpikingJelly 子包 *lava_exchange* 中的模块构成，因此可以被转换为 Lava 格式。按照补充材料中 *Exchange Modules* 的流程，我们首先将该 SNN 转换为 Lava-DL 格式，再导出为 Lava HDF5 格式的编译网络。生成的 HDF5 模型文件可以被 Lava 框架读取，并在 Loihi/Loihi2 上运行，也可以借助 Loihi/Loihi2 模拟器在 CPU 上运行。我们使用 Lava 提供的 *Loihi1SimCfg* 在 CPU 上通过 Loihi 模拟器运行该 SNN，并获得了与在 SpikingJelly 中运行原始 SNN 相同的精度。对于能够访问 Intel vLab 虚拟机提供的 Loihi 云平台的用户，还可以通过 *Loihi1HwCfg* 或 *Loihi2HwCfg* 在 Loihi 或 Loihi2 上运行该 SNN。

需要说明的是，该应用旨在展示 SpikingJelly 在处理神经形态数据集、构建、训练与部署 SNN 方面的全栈能力，而非追求最先进精度。我们没有采用大规模网络或复杂训练技巧，因此该示例仅达到 85.42% 的准确率。值得注意的是，文献 (*191*) 基于 SpikingJelly 在 DVS Gesture 数据集上实现了 95.45% 的准确率，且 SNN 仅量化到 1 bit。

应用 b 在该应用中，我们首先获取 OpenAI Gym 环境状态对应的观测值。不同任务的状态维度和动作维度差异显著，因此网络结构也会随之变化。以图中的 Ant-v3 为例，其状态维度 N_S 为 111，动作维度 N_A 为 8。由于批量大小设为 100，因此观测值是

形状为 $(N, N_S) = (100, 111)$ 的浮点张量。

类似于 PopSAN 中提出的人口编码器 (*192*)，脉冲编码器使用具有不同可学习高斯感受野的神经元群体，将每个状态维度编码为脉冲序列。最终，SNN 的输入是形状为 $(T, N, N_S \cdot P_{in}) = (5, 100, 1110)$ 的二值脉冲，其中 P_{in} 表示每个状态维度对应的输入群体规模（$P_{in} = 10$）。主要区别在于，我们这里使用反正切函数作为代理函数。SNN 位于脉冲编码器与脉冲解码器之间，其输入和输出均为脉冲序列。网络结构为 $FC256 - LIF - FC256 - LIF - FC80 - LIF$。其中，$LIF$ 表示 LIF 神经元层，$FC80$ 表示具有 80 个输出特征的全连接层，也即 $N_A \cdot P_{out}$。我们将最后一层的脉冲神经元平均划分为 N_A 个输出群体，每个群体的规模为 $P_{out} = 10$。每个输出群体对应一个脉冲解码器。每个脉冲解码器均包含一个可学习层（$FC1$）和一个积分神经元。积分神经元持续累积可学习层的输出到其膜电位上，但自身不发放脉冲。在每次经过 T 个仿真时间步后，脉冲解码器将其输入解码为积分神经元最终的膜电位，该值对应相应动作维度的输出。我们的脉冲 Actor 网络（SAN）在功能上等价于深度 Actor 网络（DAN），并可与深度 Critic 网络结合，使用 TD3 算法进行训练 (*193*)。训练过程中，SAN 学习从状态到动作的映射，以表示智能体策略；深度 Critic 网络则估计对应的 Q 值，用于指导 SAN 学习更优策略。在评估阶段，训练好的 SAN 能够预测使训练好的 Critic 网络给出最大 Q 值的动作。为确保结果可复现，每个模型都使用 10 个随机种子分别训练 10 次。每次训练持续 100 万步，并每隔 10K 步进行一次评估；每次评估都采用确定性策略，并报告 10 个回合的平均奖励。每个 episode 最多持续 1000 个环境步。其余超参数均与 PopSAN 的开源实现保持一致 (*192*)。我们实现的 SAN 中各个模块都可以通过 SpikingJelly 的 layer 和 neuron 子包构建。

我们将 SAN 与 PopSAN 进行比较，并以不同 SAN 相对于对应 DAN 在所有任务上的平均性能比（APR）作为评价指标。我们的 SAN 在 OpenAI Gym 四个常用任务（Ant-v3、HalfCheetah-v3、Hopper-v3 和 Walker2d-v3）上达到了 88.45% 的 APR。相比之下，基于 PopSAN 的结果不仅 APR 更低（71.94%），训练速度也更慢。该应用展

示了 SpikingJelly 作为实时机器人控制任务中节能替代方案的潜力。

应用 c 在该应用中，我们首先使用 SpikingJelly 的 *activation_based* 模块，在 ImageNet 数据集上预训练 SEW ResNet (*60*)。网络深度分别为 18、34、50、101 和 152。对于所有网络，我们采用相同的训练设置，并将仿真时间步设为 $T = 4$。每个网络均训练 320 个 epoch，小批量大小为 32；优化器为动量 0.9 的 SGD。初始学习率为 0.1，并采用余弦退火学习率调度，其中 $T_{max} = 320$，与总 epoch 数一致。所有这些网络在 ImageNet 上的准确率都达到或超过了 63%。

在完成这些网络的预训练后，我们通过比较其神经表征与生物视觉系统的神经表征来评估它们的类脑相似性。对于生物视觉系统，我们以动物观看若干静态图像时的神经响应作为神经表征；对于网络，我们输入与生物视觉系统相同的静态图像，并使用与训练阶段相同的仿真时间步长。随后，我们从网络所有层中提取特征作为神经表征。最后，采用三个指标 (*194*) 计算网络神经表征与生物视觉系统神经表征之间的相似性。

我们将 SEW ResNet 与其对应的 CNN（ResNet）的表征相似性进行比较，发现前者平均高出 6.6%。这一应用表明，SpikingJelly 可以作为构建更多类脑计算模型的有效工具。

循环结构的性能比较

图 1**d** 展示了前馈 SNN 在 SpikingJelly、Norse 和 SNNTorch 上的性能比较。与广泛用于静态数据集和神经形态数据集分类的前馈 SNN 相比，循环 SNN 具有学习长期依赖的潜力，并已被用于时序任务 (*180, 184, 194*)。为了比较不同框架在模拟循环 SNN 时的性能，我们对文献 (*194*) 中使用的循环 SEW ResNet-18 进行了基准测试。实验设置与图 1**d** 相同，结果如图 27 所示。由于循环 SEW ResNet 在四个 stage 中都包含循环连接，并且只能采用单步模式，因此无法使用 SpikingJelly 基于多步模式的加速方法。在这种情况下，JIT 是 SpikingJelly 唯一可用的加速方式。训练结果表明，当 T 较小（如 $T = 2, 4, 8$）时，速度排序为 Norse > SpikingJelly > SNNTorch。当 T 较大（如

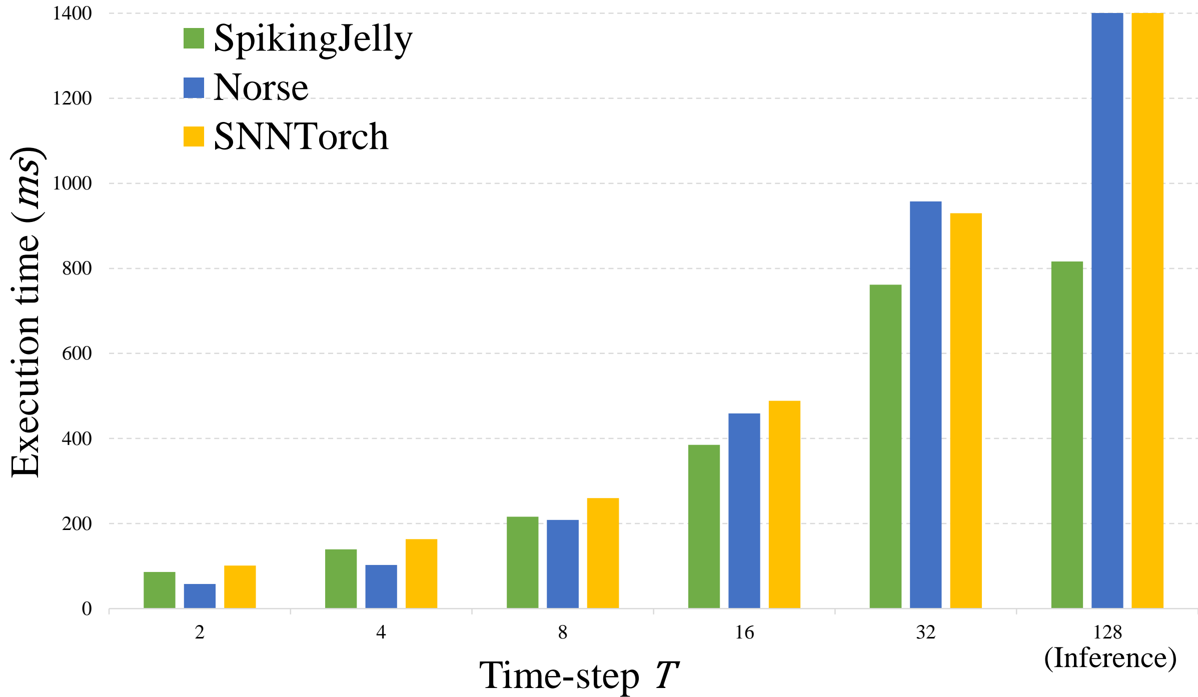

图 27: 由 SpikingJelly、Norse 和 SNNTorch 构建的循环 SEW ResNet-18 (*194*)，在单次训练迭代（$T=2,4,8,16,32$）和推理（$T=128$）时的执行时间。

$T=16,32$）时，SpikingJelly 比 Norse 更快，而 SNNTorch 仍然最慢。推理结果与图 **1d** 一致；SpikingJelly 的执行时间仅为 Norse 和 SNNTorch 的 0.57×，而后两者的推理时间彼此接近。相关实验的源代码和数据同样在补充材料中给出。

总的来说，SpikingJelly、Norse 和 SNNTorch 在训练完全循环结构时表现接近：当 T 较小时，Norse 更快；当 T 较大时，SpikingJelly 更具优势。而在推理阶段，SpikingJelly 仍然保持速度优势。需要指出的是，对于只包含部分循环连接的 SNN，可以采用混合传播模式，因此用户仍可在网络的前馈部分使用基于多步模式的加速方法。

限制和未来发展计划

尽管经过三年的发展，SpikingJelly 已经成为一个较为成熟的框架，但它仍然存在一些局限。结合开发者自身经验以及与用户的交流，我们总结出当前的主要局限性和未

来可能的发展方向如下。

基于时序的 SNN 工具包近年来，大多数深度 SNN 采用的是基于激活的表示，即使用二值张量 S 表示脉冲。较早的工作，例如 SpikeProp (*39*) 和 Tempotron (*40*)，则采用基于时序的表示，用发放时刻 t_f 来表示脉冲。例如，若时间单位为 1，则在基于激活的表示中，$S = [0, 1, 0, 1]$；而在基于时序的表示中，则对应为 $t_f = [1, 3]$。可以看出，基于时序的表示更加稀疏、更加节省内存，并且有望在连续时间而不是离散时间步上工作。然而，由于卷积和矩阵乘法等复杂运算更难实现，基于时序的 SNN 发展速度明显慢于基于激活的 SNN。相应地，虽然 SpikingJelly 中已经包含 *timing_based* 子包，但该部分目前仍较为初步。考虑到基于时序表示的潜在优势，设计更完善的时序型 SNN 工具包，并补全 *timing_based* 子包，是一个值得推进的发展方向。

稀疏加速虽然 SNN 层间传递的脉冲通常是稀疏的，但这些脉冲往往仍以稠密张量形式存储。这样虽然便于兼容深度学习中的常用操作，却没有充分利用其稀疏性。当脉冲发放率足够低时，使用稀疏计算库（如用于基础线性代数的 cuSPARSE 和用于稀疏卷积/池化的 Minkowski Engine）来训练 SNN (*195*)，可以减少内存消耗并提升仿真速度。在 SpikingJelly 早期的 0.0.0.0.4 版本中，曾提供 *AutoSparseLinear* 层，它会在首次运行时执行基准测试，以估计临界稀疏度。所谓临界稀疏度，是指稀疏矩阵乘法与稠密矩阵乘法速度相同时对应的稀疏程度。当输入脉冲的稀疏度高于该阈值时，使用稀疏矩阵乘法；否则使用稠密矩阵乘法。因此，*AutoSparseLinear* 层能够在训练过程中自动选择更快的矩阵乘法实现。不过，由于开发时间和精力有限，对稀疏加速的探索在 0.0.0.0.4 之后暂时搁置，未来我们仍会考虑重新启动这一方向的开发。

复杂神经元的自动 CUDA 代码生成在没有 CuPy 后端时，实现复杂脉冲神经元本身并不需要太高的开发成本。然而，复杂神经元动力学往往意味着 PyTorch 需要频繁调用大量细粒度 CUDA 内核，这会明显拖慢 SNN 的训练速度，尤其是在 T 很大时。如图 **5c** 所示，即使是动力学最简单的 IF 神经元，也会遭遇这一问题。训练速度缓慢，可能正是复杂脉冲神经元在深度 SNN 中较少被采用的原因之一。因此，只有少量工作在深度 SNN

中使用了与 Izhikevich 神经元复杂度相近的神经元 (*124*) 或更高阶的脉冲神经元 (*196*)。通过为复杂神经元构建 CuPy 后端，并用大型 CUDA 内核融合其全部神经元动力学，可以有效缓解仿真速度慢的问题。然而，为前向和反向传播分别编写 CUDA 代码，需要开发者投入大量精力。SpikingJelly 中的半自动代码生成技术在一定程度上缓解了这一问题，但仍不是最终理想方案。理想的 CUDA 代码生成方式应当是全自动的，即直接从定义神经元动力学的 Python 代码生成 CUDA 代码。在 SpikingJelly 0.0.0.0.14 版本中，我们引入了原型性质的生成器子包 *spikingjelly.activation_based.auto_cuda.generator*。尽管该子包尚未完全成熟，但其发展路线已经明确：生成器将根据 Python 代码中定义的神经元动力学构建计算图，分析图中变量及其数学关系，并据此自动生成 CUDA 代码。

统计趋势

我们统计了 AI 顶级会议中与脉冲深度学习相关的录用论文数量，如图 28 所示。所选会议包括 AAAI 人工智能会议（AAAI）、神经信息处理系统大会（NeurIPS）、IEEE 计算机视觉与模式识别会议（CVPR）、国际计算机视觉会议（ICCV）、欧洲计算机视觉会议（ECCV）、国际人工智能联合会议（IJCAI）、国际机器学习会议（ICML）以及国际学习表示会议（ICLR）。这些会议普遍被认为是人工智能研究领域最主要的顶级会议。自 2019 年以来，相关论文的录用数量显著增加，表明研究兴趣也从那时开始快速上升。SpikingJelly 恰好于 2019 年 12 月开源。我们还统计了自 2020 年起每年使用 SpikingJelly 的论文数量：2020 年 1 篇、2021 年 10 篇、2022 年 42 篇，以及 2023 年 17 篇（截至 2023 年 3 月 1 日）。这些统计结果表明，SpikingJelly 满足了研究人员对脉冲深度学习工具包的需求，并推动了社区的发展。

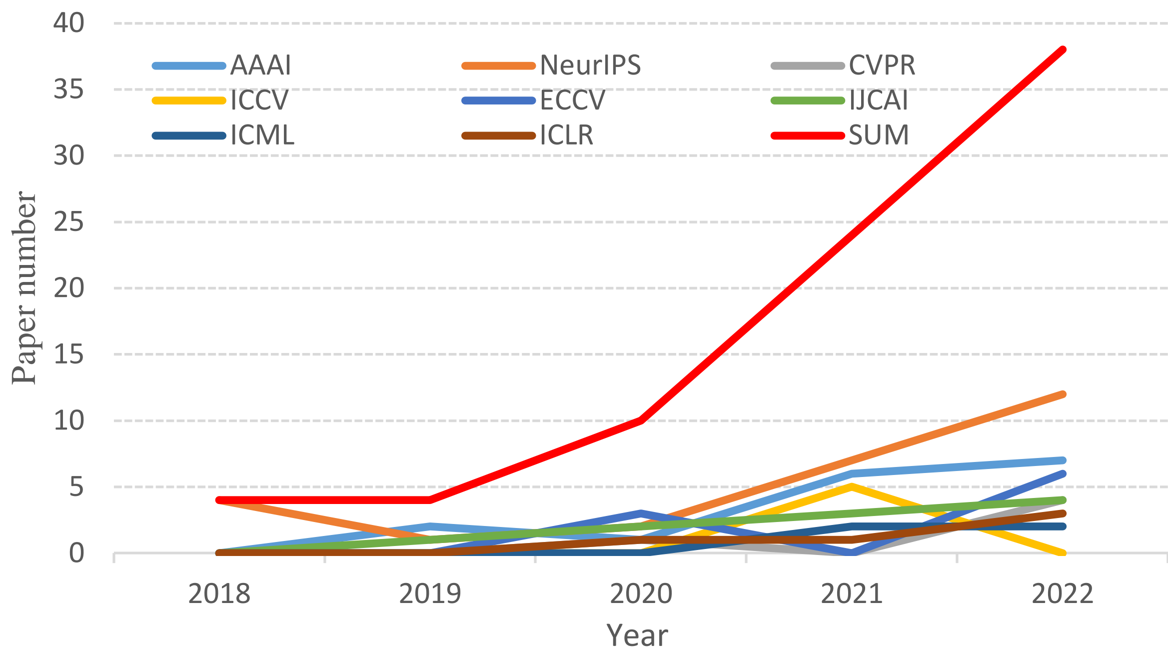

图 28: **顶级人工智能会议上与脉冲深度学习相关的录用论文数量增长。**请注意，“SUM”表示所有会议统计值的总和。可以看出，自 2019 年起，相关研究兴趣显著上升。