Smart Media Journal / Vol.15, No.5 / ISSN:2287-1322
https://dx.doi.org/10.30693/SMJ.2026.15.5.44

SNN 시뮬레이터를 위한 메타모델 기반 하드웨어 인식 자동 코드 생성 프레임워크

(Hardware-Aware Automatic Code Generation Framework Based on Meta-Model for SNN Simulators)

손유현*, 허준영**
(U Hyun Son, Junyoung Heo)

요 약

스파이킹 신경망(SNN) 연구 환경은 snnTorch, SpikingJelly, BindsNET 등 상호 호환되지 않는 여러 시뮬레이터로 분산되어 있어, 동일한 모델을 각 환경에 맞게 다시 구현해야 하는 문제가 있다. 이 문제는 단순한 불편함을 넘어 재현성과 신뢰성에 직접적인 영향을 미치며, 효율성이 떨어진다. 본 논문은 시뮬레이터에서 독립적인 JSON 기반 메타모델과 Jinja2 템플릿 엔진을 활용한 자동 코드 생성 프레임워크를 제안한다. 단일 JSON 명세로부터 snnTorch, SpikingJelly, BindsNET의 실행 가능한 Python 코드를 자동 생성하며, 전력 모드 설정에 따라 LIF와 IF 뉴런 교체, FP16 변환, 타임스텝 축소 등의 하드웨어 인식 최적화를 코드 생성 시점에 자동 적용한다. 다만 본 연구는 완전 연결(FC) 레이어 구조에 한정하며, 컨볼루션 레이어로의 확장은 다루지 않는다. 6가지 구성(3개의 시뮬레이터와 2개의 전력 모드)에 대한 실험을 통해 코드 생성 정확성과 최적화 적용을 검증한다.

■ 중심어 : 스파이킹 신경망 ; 메타모델 ; Jinja2 ; JSON

Abstract

The research environment for Spiking Neural Networks (SNNs) is fragmented across various incompatible simulators, such as snnTorch, SpikingJelly, and BindsNET. This fragmentation necessitates the redundant reimplementation of the same model for each environment, which not only causes significant inconvenience but also undermines reproducibility and reliability. This paper proposes an automated code generation framework utilizing a simulator-independent, JSON-based metamodel and the Jinja2 template engine. From a single JSON specification, the framework automatically generates executable Python code for snnTorch, SpikingJelly, and BindsNET. Furthermore, it applies hardware-aware optimizations during the code generation phase—such as switching between LIF and IF neurons, converting to FP16 precision, and reducing timesteps—based on predefined power mode settings. While this study is limited to Fully Connected (FC) layer structures and does not cover expansion to convolutional layers, we verify the accuracy of the generated code and the effectiveness of the optimizations through experiments across six different configurations (comprising three simulators and two power modes).

■ keywords : Spiking Neural Network ; metamodel ; Jinja2 ; JSON

Ⅰ. 서 론

SNN 시뮬레이터 코드를 작성하다보면 개발 과정에서 공통적으로 불편함을 느끼는 부분이 있는데, snnTorch로 작성한 LIF 기반 분류 모델

* 준회원, 한성대학교 컴퓨터공학과 대학원생
** 정회원, 한성대학교 컴퓨터공학부 교수
본 연구는 한성대학교 교내학술연구비 지원과제임
접수일자 : 2026년 04월 29일
수정일자 : 2026년 05월 20일
게재확정일 : 2026년 05월 20일
교신저자 : 허준영 e-mail : jyheo@hansung.ac.kr

을 SpikingJelly 환경에서 재현하려 했을 때, API 차이가 단순한 함수명 수준이 아니라 구조 전반에 걸쳐 있다는 것을 확인하고, 처음부터 다시 작성하는 것 외에 선택지가 없었다. 이러한 문제를 해결하기 위해 소스 코드 수준 보다 더 추상화된 모델 기술 방법이 필요하였다.

SNN 연구에서 활발하게 사용되는 시뮬레이터 3가지인 snnTorch, SpikingJelly[5], BindsNET[7]은 동일한 LIF 뉴런 모델을 각각 'snn.Leaky', 'neuron.LIFNode', 'LIFNodes'로 구현하며, 시간축을 처리하는 방식도 근본적으로 다르기 때문에 코드 수준의 이식이 불가능한 구조이다. 이와 더불어, snn 모델을 실제 하드웨어에 배포하는 단계에서도 호환성 및 제약 조건에 따른 구조적 한계가 존재한다. 엣지 디바이스나 뉴로모픽 칩에 모델을 올릴 때, 연산 비용이 높은 LIF 뉴런을 IF 뉴런으로 교체하거나 FP16으로 변환하는 최적화를 연구자가 수동으로 처리해야 하는 구조적 제약이 존재한다. 이는 하드웨어 의존적인 최적화 로직이 소스 코드 전반에 파편화되는 구조적 특성 때문이다. 이로 인해 개발자의 실수가 컴파일 시점에 검출되지 않으며, 오직 런타임 시점의 미세한 수치적 정확도 손실로만 나타나므로 디버깅을 통한 오류 추적이 극도로 어렵다는 한계가 있다.

본 연구는 이러한 두 가지 문제를 동시에 해결할 수 있는 접근으로, 시뮬레이터 API와 독립된 SNN 명세를 JSON으로 표현하고 이를 입력으로 시뮬레이터의 실행 코드를 자동 생성하는 프레임워크를 제안한다. JSON으로 표현한 SNN 명세는 추상화 수준이 소스 코드에 비해 현저히 높기 때문에, 다른 시뮬레이터용으로 작성하는 것이 매우 간편하다. 또한, 배포 환경의 제약 조건인 전력 모드나 디바이스 타입은 명세 단계에서 미리 선언하고, 코드 생성 시점에 자동으로 반영된다. 다만 본 연구의 유효성 검증 범위는 SNN의 핵심 연산 구조를 포함하는 완전 연결(FC) 레이어 기반 아키텍처로 제한하여 수행하였다.

본 논문의 구성은 2장에서 기존 연구를 살펴보고, 3장에서 메타모델 설계를, 4장에서 코드 생성 프레임워크 구현을 기술한다. 5장에서 실험 결과를 제시하고 그 뒤에 결론을 맺는다.

Ⅱ. 관련 연구

1. SNN 시뮬레이터 현황과 이식성 문제

본 논문에서 다루고 있는 SNN 시뮬레이터는 snnTorch, SpikingJelly, BindsNET으로 각각은 다른 설계 목표와 사용 패턴을 가진다.

가. snnTorch

snnTorch는 PyTorch 'nn.Module'로 SNN 구성 요소를 제공한다. 뉴런 모델이 스파이크와 막전위를 튜플로 반환하는 구조적 특성 때문에 'nn.Sequential' 직접 사용이 어렵고, 사용자가 'nn.ModuleList'와 수동 루프를 조합해 입력을 직접 보내야 한다.

나. SpikingJelly

SpikingJelly는 'step_mode='m'' 파라미터를 레이어 단위로 지정하면 시간축 처리가 내장되어 'nn.Sequential' 기반의 간결한 구성이 가능하다. CuPy 백엔드와 AMP(자동 혼합 정밀도)를 지원해 고성능 GPU 환경에 적합하다.

다. BindsNET

BindsNET은 중앙 네트워크 객체와 명시적 그래프 구저를 사용하여, 순차 컨테이너와 레이어 적층 방식을 사용하는 앞선 두 프레임워크(snnTorch, SpikingJelly)와 컴포넌트 연결 구조 및 설계 패러다임 측면에서 근본적으로 다르다. 레이어와 연결, 모니터를 모두 'network.add_*' API로 명시적으로 등록하는 그래프 구조를 사용하며, 생물학적 단위 파라미터를 직접 지정한다. STDP 기반 'PostPre' 학습 규칙을 지원해 생물

학적 정확도를 중시하는 연구에 주로 활용된다.

2. ONNX와 SNN의 간극

프레임워크 간 모델 이식 문제는 DNN 분야에서 ONNX(Open Neural Network Exchange)[15]로 일정 부분 해결되었다. 그러나 ONNX의 정적 계산 그래프는 시간 차원이 없어, SNN의 시간적 동작을 표현하기 어렵다. 그러나 ONNX의 정적 계산 그래프는 본질적으로 시간(Time-step) 차원을 내포하지 않아, SNN 특유의 동적 막전위 변화 및 스파이크 발화 매커니즘을 표현하는 데 한계가 있다. 이를 우회하기 위해 커스텀 연산자를 정의하는 방안이 존재하나, 이는 표준 ONNX 런타임과의 호환성을 상실시켜 플랫폼 독립적인 이식성이라는 본연의 이점을 퇴색시킨다. 이러한 런타임 호환성 문제를 해결하기 위해, 본 연구는 계산 그래프 자체를 표준화하려는 방식 대신 프레임워크 독립적인 상위 명세(JSON)를 유지하되 각 시뮬레이터별 시간축 루프를 전용 템플릿 엔진을 통해 맞춤형 코드로 변환(Code Generation)하는 아키텍처를 채택한다. 이를 통해 표준 런타임의 수정 없이도 각 프레임워크의 고유한 시간축 처리 매커니즘을 온전히 활용할 수 있도록 지원한다.

3. 하드웨어 최적화 : post-training 중심의 한계

SNN의 엣지 배포를 위한 최적화 연구는 주로 학습 완료 후(post-training) 단계에서 수행된다. 모델 학습 후 별도 스크립트로 LIF를 IF로 교체하거나 가중치를 FP16으로 변환하는 방식이 일반적이다.[16] 그러나 여러 구성을 실험할 때마다 변환을 수동으로 반복해야 하고, 변환 파라미터를 잘못 설정해도 실수를 발견하기 힘들다는 문제가 있다. 본 연구는 하드웨어 제약을 명세 단계에서 선언하고 코드 생성 시점에는 자동 적용함으로써 이 문제를 해결한다.

4. 템플릿 기반 코드 생성

Model-Driven Engineering(MDE)[17]은 소프트웨어 모델을 일급 객체로 취급하고 모델에서 코드를 자동 생성하는 방법론으로, 임베디드 시스템이나 DSL 개발에서 활용 사례가 보고되어 있다. Jinja2[18]는 Python 생태계의 템플릿 엔진으로, 조건문, 반복문, 필터를 지원해 복잡한 코드 구조를 선언적으로 생성할 수 있다. 본 연구는 이 조합을 SNN 도메인에 적용하였다.

III. 메타모델 설계

1. 설계 방향

본 연구에서 제안하는 SNN 메타모델은 플랫폼 독립적인 경량 명세와 Python 생태계 내에서의 데이터 처리 효율성을 보장하기 위해 JSON(JavaScript Object Notation) 포맷을 기반으로 스키마를 설계하였다.

제안하는 메타모델 스키마는 시뮬레이터 간 이식성과 최적화 자동화를 위해 다음의 세 가지 설계 요구사항과 해결책을 기반으로 구성된다.

첫째, 명세의 최소성(Minimality) 요구이다. 스키마의 복잡도를 낮추기 위해 필수적인 구조적 필드만을 정의하고, 가변적인 세부 파라미터는 대상 시뮬레이터의 기본값(Default)에 위임하도록 설계하였다. 둘째, 플랫폼 확장성(Extensibility) 요구이다. 향후 새로운 백엔드 통합 시 코어 스키마가 변경되는 문제를 방지하고자, 레이어 타입과 시뮬레이터 식별자를 개방형 문자열로 추상화하여 구조적 유연성을 확보하였다. 셋째, 하드웨어 인식(Hardware-Awareness) 요구이다. 배포 환경에 따른 맞춤형 최적화를 보장하기 위해, ‘power_mode’와 ‘hardware_constraints’를 최상위 필드로 배치하여 환경 제약 조건이 코드 생성 엔진의 핵심 메인 입력으로 유기적으로 반영되도록 제어하였다.

2. JSON 스키마 구조

표 1은 메타모델의 최상위 필드를 정리한 것이다.

표 1. 메타모델 JSON 스키마 핵심 필드

필드	타입	설명
‘model_name’	string	모델 고유 식별자
‘target_simulator’	string	대상 시뮬레이터
‘time_steps’	integer	시뮬레이션 타임스텝 수
‘power_mode’	string	전력 모드
‘hardware_constraints.device_type’	string	배포 하드웨어 유형
‘layer’	array	레이어 목록
‘connections’	array	레이어 간 연결 정의
‘backend’	string	연산 백엔드
‘step_mode’	string	실행 모드

‘power_mode’와 ‘device_type’을 별도 필드로 분리한 이유는, 고성능 GPU에서도 전력 제약이 있는 실험 환경이 존재하기 때문이다. 두 필드를 독립적으로 지정하면 하드웨어 유형과 전력 설정을 조합해 더 세밀하게 배포 환경을 표현하는 것이 가능하다.

3. 레이어 표현 및 파라미터 처리

본 연구에서 메타모델이 지원하는 레이어 타입은 세 가지다. linear은 완전 연결 레이어로, 입력과 출력 크기만 필수로 지정하면 된다. LIFNode와 IFNode는 둘 다 필수 파라미터가 없다. 같은 LIF 뉴런이라도 snnTorch는 감쇠 계수를 beta로, SpikingJelly는 시상수 tau로, BindsNET은 thresh와 rest 전압값으로 표현한다. 메타모델에서 이 파라미터들을 하나의 공통 필드로 강제할 경우, 스키마가 특정 시뮬레이터의 표현 방식에 종속되는 문제가 발생한다. 따라서 명세에 포함된 파라미터만 생성 코드에 반영하고, 나머지는 각 시뮬레이터의 기본값을 그대로 따르도록 했다. 만약 사용자가 세밀한 제어가 요구되는 경우, 해당 시뮬레이터 고유의 파라미터 명칭을 메타모델 내 선택적 필드로 정의하여 유연하게 확장할 수 있다.

코드 1은 snnTorch와 SpikingJelly 시뮬레이터를 위해 JSON으로 기술한 SNN 명세로, 실험에서 사용한 MNIST 인식을 위한 모델이다. 대부분의 내용이 같고, 다만 시뮬레이터마다의 특성으로 인한 세부 설정이 일부 다르다.

구체적으로, snnTorch 명세에서는 LIF 뉴런의 감쇠 계수를 ‘beta’: 0.95와 같이 단일 파라미터로 선언하고 서로게이트 함수명을 소문자(‘atan’)로 기술한 반면, SpikingJelly 명세에서는 동일한 뉴런 모델을 시상수 기반의 ‘tau’: 2.0 및 임계 전압 필드들로 세분화하여 정의하고 파스칼 표기법(‘ATan’)을 적용하였다. 아울러 시간축 연산 백엔드 구동을 위한 실행 모드 필드 역시 대상 시뮬레이터의 표준 명명 규칙을 준수하여 각각 ‘multi_step’과 ‘m’으로 다르게 매핑되었음을 확인할 수 있다.

코드 1. snnTorch, SpikingJelly 시뮬레이터용 메타모델

SnnTorchStandardModel

```
{
  "model_name": "SnnTorchStandardModel",
  "target_simulator": "snnTorch",
  "time_steps": 100,
  "power_mode": "normal",
  "hardware_constraints": {
    "device_type": "high_performance_gpu"
  },
  "layers": [
    {
      "type": "linear",
      "in_features": 784,
      "out_features": 256
    },
    {
      "type": "LIFNode",
      "surrogate_function": "atan",
      "beta": 0.95,
      "note": "High precision with ArcTan surrogate"
    },
    {
      "type": "linear",
      "in_features": 256,
      "out_features": 10
    },
    {
      "type": "LIFNode",
      "surrogate_function": "atan",
      "beta": 0.95
    }
  ],
  "connections": [
    { "source": "input", "target": "layer_1" },
    { "source": "layer_1", "target": "layer_2" }
  ],
  "backend": "torch",
```

SpikingJellyStandardModel

```
{
  "model_name": "SpikingJellyStandardModel",
  "target_simulator": "SpikingJelly",
  "time_steps": 100,
  "power_mode": "normal",
  "hardware_constraints": {
    "device_type": "high_performance_gpu"
  },
  "layers": [
    {
      "type": "linear",
      "in_features": 784,
      "out_features": 256
    },
    {
      "type": "LIFNode",
      "tau": 2.0,
      "v_threshold": 1.0,
      "v_reset": 0.0,
      "surrogate_function": "ATan",
      "note": "Standard Leaky Integrate-and-Fire with ATan"
    },
    {
      "type": "linear",
      "in_features": 256,
      "out_features": 10
    },
    {
      "type": "LIFNode",
      "tau": 2.0,
      "v_threshold": 1.0,
      "surrogate_function": "ATan"
    }
  ],
  "connections": [
    { "source": "input", "target":
```

<table>
<tr>
<td>"step_mode": "multi_step"
}</td>
<td>"layer_1" },
{ "source": "layer_1", "target": "layer_2" }
],
"backend": "torch",
"step_mode": "m"
}</td>
</tr>
</table>

Ⅳ. 코드 생성 프레임워크

1. 전체 구조

프레임워크는 parser→codegen→template의 세 단계로 구성된다. parser.py가 JSON을 로드하고 유효성을 검사하면, codegen.py가 하드웨어 인식 최적화를 수행한 뒤, Jinja2 템플릿 엔진이 시뮬레이터별 Python 코드를 렌더링한다.

2. 하드웨어 인식 최적화

표 2는 codegen.py가 수행하는 최적화 전략과 적용 조건을 정리한 것이다.

표 2. 하드웨어 인식 최적화 전략

최적화 전략	적용 조건	효과
LIFNode → IFNode 교체	‘power_mode == “low_energy”’	지수 감쇠 연산 제거, ~30~40% 절감
타임스텝 축소	‘device_type == “edge_ai_processor”’	추론 시간 단축
FP16 변환	‘power_mode == “low_energy”’ + CUDA 환경	메모리 대역폭 ~50% 절감
서로게이트 함수 교체	‘power_mode == “low_energy”’	‘atan’ → ‘fast_sigmoid’

LIF와 IF 교체는 단순한 데이터 타입의 변경이 아니라, 시간 축에 따른 막전위 감쇠 유무라는 내재적 연산 특성의 변환을 의미한다. LIF 뉴런의 감쇠 파라미터(‘beta’, ‘tau’)는 IF 뉴런에서 유효하지 않으므로, codegen.py는 교체 시 해당 파라미터를 제거하고 시뮬레이터별로 IF 뉴런에 유효한 파라미터만 남긴다.

3. 시뮬레이터별 코드 생성 전략

세 시뮬레이터는 네트워크 구조 정의 패러다임이 달라 생성 전략도 각각 다른데, snnTorch는 `nn.ModuleList` 기반으로 레이어를 구성하고, 타임스텝마다 순차적으로 입력을 네트워크에 통과시키는 수동 루프 구조로 생성된다. `nn.Sequential`을 사용하지 않는 이유는 snnTorch의 뉴런 모델이 스파이크와 막전위라는 두 가지 독립된 연산 데이터를 동시에 반환하기 때문이다. 단일 데이터의 일방향 흐름만 제어하도록 설계된 기존의 순차 컨테이너(nn.Sequential)로는 이처럼 다중 출력이 발생하는 뉴런 간의 신호 전달 파이프라인을 온전히 처리할 수 없다. 표준 모드에서는 `snn.Leaky`, 저전력 모드에서는 RC 회로 기반의 `snn.Lapicque`가 선택된다. SpikingJelly는 `nn.Sequential` 기반으로 생성되며, `step_mode='m'`을 레이어 단위로 지정해 시간축 처리를 내장한다. 저전력 구성에서는 `LIFNode`가 `IFNode`로 교체되고, CUDA 환경이면 AMP 기반 FP16 추론 코드가 자동 포함된다. BindsNET은 레이어 등록 → 연결 정의 → 모니터 등록의 세 단계를 순서를 지키며 생성해야 한다. 각 단계에서 레이어 이름을 문자열로 참조하는 구조이기 때문에, 앞선 두 시뮬레이터처럼 레이어 목록을 단순 순회하는 방식으로는 처리할 수 없다. 저전력 구성에서는 `LIFNodes`가 `IFNodes`로 교체되고 `tc_decay` 파라미터가 제거된다.

4. Jinja2 템플릿 구조

각 시뮬레이터 전용 템플릿(`snntorch_template.j2`, `spikingjelly_template.j2`, `bindsnet_template.j2`)은 `templates/` 디렉터리에서 `FileSystemLoader`로 관리된다. 템플릿은 렌더링만 담당하고 최적화 판단은 codegen.py에서 이미 완료된 상태로 전달되므로, 새 시뮬레이터를 추가할 때 기존 최적화 로직을 별도로 수정하거나 변경할 필요가 없다.

V. 실험 및 평가

1. 실험 설정

실험은 표 3의 6가지 구성에 대해 수행하였다. MNIST(입력 차원 784) 기반 2계층 분류 모델을 사용했으며, 프레임워크 검증이 목적이므로 더 복잡한 데이터셋은 사용하지 않았다. 측정 항목은 생성 코드 실행 성공 여부, 메타모델 명세와의 구조적 일치성(정적 분석), 하드웨어 최적화 적용 여부, SpikingJelly 구성에서의 추론 시간 및 메모리 사용량이다. 이러한 지표를 선정하는 이유는 제안하는 프레임워크가 소스 코드를 변환하는 과정에서 유발할 수 있는 문법적, 구조적 오류를 사전에 차단하고(기능적 무결성 검증), 자동 생성된 코드가 실제 하드웨어 환경 및 시뮬레이터 런타임 상에서 실질적인 자원 최적화 효율을 달성하는지 실증적으로 평가하기 위함이다.

표 3. 실험 구성 목록

구성	시뮬레이터	전력 모드	모델명
C1	snnTorch	normal	SnnTorchStandardModel
C2	snnTorch	low_energy	SnntorchLowPowerModel
C3	SpikingJelly	normal	SpikingJellyStandardModel
C4	SpikingJelly	low_energy	SpikingJellyLowPowerModel
C5	BindsNET	normal	BindsNetStandardModel
C6	BindsNET	low_energy	BindsNetLowPowerModel

2. 생성 코드 정확성 검증

6개 구성 모두 오류 없이 실행되었으며, 레이어 수·뉴런 타입·파라미터가 명세와 일치하는 것을 정적 분석으로 확인했다.

3. 성능 비교

SpikingJelly 구성(C3, C4)에 대해 CUDA 환경에서 추론 시간과 메모리 사용량을 측정한 결과를 표 4에 표기하였다.

표 4. spikingJelly Standard & Low-Power 성능 비교 (CUDA, MNIST)

항목	Standard(C3)	Low-Power(C4)	변화율
뉴런 타입	LIFNode	IFNode	-
히든 레이어 크기	256	128	-50%
타임스텝	100	40	-60%
정밀도	FP32	FP16	-
추론 시간(ms)	12.4 ± 0.3	3.1 ± 0.2	-75%
모델 메모리(MB)	4.2	1.1	-73.8%
스파이크 발화율(avg)	0.142	0.138	-2.8%

표 내에서 추론 시간 단축(-75%)은 타임스텝 축소, FP16 변환, 레이어 크기 축소가 복합적으로 작용한 결과다. 흥미로운 점은 스파이크 발화율 변화가 2.8%에 불과하다는 것이다. MNIST처럼 단순한 분류 태스크에서는 LIF와 IF의 동작 차이가 발화율 수준에서 잘 드러나지 않는 것으로 보이며, 더 복잡한 시계열 데이터에서의 확인이 필요하다.

VI. 결 론

본 연구는 구조적 설계 패러다임이 상이한 SNN 시뮬레이터(snnTorch, SpikingJelly, BindsNET)를 대상으로 플랫폼 독립적이고 균일한 코드 생성 파이프라인을 구축하고자 하였다. 특히 순차 컨테이너 적층 방식을 공유하여 비교적 호환성이 높은 프레임워크뿐만 아니라, 중앙 집중형 그래프 아키텍처를 채택하여 템플릿 명세화가 까다로운 BindsNET 구조까지 포괄하는 범용적 통합 메타모델을 제안하였다. 이 과정에서 최적화 로직과 렌더링 로직의 분리가 명확할수록 유지보수의 난이도가 낮아진다는 것을 알 수 있었

다. 단일 JSON 명세로 세 가지의 시뮬레이터의 코드를 자동 생성하고 하드웨어 인식 최적화를 자동 적용한다는 목표는 달성했다. 하지만 시뮬레이터마다 같은 개념에 대한 기본값과 수치 해석이 조금씩 달라, 단일 명세를 메타모델에 넣어도 완전히 동등한 모델을 생성한다고 보장하기는 어려운 부분이 있다. 현재 완전 연결(FC) 레이어에 한정되어 있어, 향후 연구에서는 컨볼루션 레이어 지원을 추가하여 실제 연구 활용성을 높일 예정이다.

REFERENCES

[1] Davison, A. P., et al. (2009). "PyNN: a common interface for neuronal network simulators." *Frontiers in Neuroinformatics*.

[2] Salvador, A., et al. (2019). "NetPyNE, a tool for data-driven multiscale modeling of brain circuits." *eLife*.

[3] Malcolm, k.,&Casco-Rodriguez, J. (2023). "A Comprehensive Review of Spiking Neural Networks: InterPretation, Optimization, Efficiency, and Best Practices." *arXiv*.

[4] Aliyev, I., Svoboda, K., Adegbija, T., & Fellous, J.-M.(2024). "Sparsity-Aware Hardware-Software Co-Design of Spiking Neural Networks: An Overview." *arXiv*.

[5] Fang, W., et al. (2023). "SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence." *arXiv*.

[6] Eshraghian, J. K., et al. (2021). "Training Spiking Neural Networks Using Lessons From Deep Learning." *arXiv*.

[7] Hazan, H., et al. (2018). "BindsNET: A machine learning-oriented spiking neural networks library in python." *arXiv*.

[8] Gleeson, P., et al. (2010). "NeuroML: A Language For Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail." *PloS Computational Biology*.

[9] Raikov, I., et al. (2010). "NineML-a description language for spiking neuron network modeling: the abstraction layer." *PBMC Neuroscience*.

[10] Dai, K., et al. (2020). "The SONATA data format for efficient description of large-scale network models." *PloS Computational Biology*.

[11] Bekolay, T., et al. (2014). "Nengo: a Python tool for building large-scale functional brain models." *Front Neuroinform*.

[12] Rhodes, O., et al. (2018). "sPyNNaker: a Software Package for Running PyNN Simulations on SpiNNaker." *Front Neuroinform*.

[13] Yavuz, E., et al. (2016). "GeNN: a code generation framework for accelerated brain simulations." *Scientific Reports*.

[14] Putra, R. V. W., & Shafique, M. (2024). "HASNAS: A Hardware-Aware Spiking Neural Architecture Search Framework for Neuromorphic Compute-in-Memory Systems." *arXiv*.

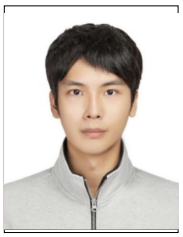
[15] Qin, S. (2025). "ONNX-Net: Towards Universal Representations and Instant Performance Prediction for Neural Architectures." *arXiv*.

[16] Rueckauer, B., & Liu, S. (2018). "Conversion of Analog to Spiking Neural Networks Using Sparse Temporal Coding." *IEEE*.

[17] Di Rocco, J., et al. (2024). "On the use of Large Language Models in Model-Driven Engineering." *arXiv*.

[18] Alidu, A., et al. (2025). "Prompt2DAG: A Modular Methodology for LLM-Based Data Enrichment Pipeline Generation." *arXiv*.

저 자 소 개



손유현(준회원)

2024년 : 한성대학교 컴퓨터공학과 학사 졸업.

2024년 ~ 현재 : 한성대학교 컴퓨터공학과 석사

<주관심분야 : 인공지능, 빅데이터>

허준영(정회원)

1998년 : 서울대학교 컴퓨터공학과 학사 졸업.

2009년 : 서울대학교 컴퓨터공학과 박사 졸업.

2009년 ~현재 : 한성대학교 컴퓨터공학과 교수

<주관심분야 : 운영체제, IoT, 임베디드시스템, 기계학습>