creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

석사학위논문

SNN 시뮬레이터를 위한 메타모델 기반 프레임워크

HANSUNG UNIVERSITY

2026년

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

손 유 현

HANSUNG
UNIVERSITY

석 사 학 위 논 문
지도교수 허준영

SNN 시뮬레이터를 위한 메타모델 기반 프레임워크

Metamodel-based framework for SNN simulators

HANSUNG UNIVERSITY

2026년 6월 일

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

손 유 현

석 사 학 위 논 문
지도교수 허준영

SNN 시뮬레이터를 위한 메타모델 기반 프레임워크

Metamodel-based framework for SNN simulators

위 논문을 공학 석사학위 논문으로 제출함

2026년 6월 일

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

손 유 현

손유현의 공학 석사학위 논문을 인준함

2026년 06월 일

심사위원장 정진만 (인)

심 사 위 원 허준영 (인)

심 사 위 원 민 홍 (인)

HANSUNG UNIVERSITY

국 문 초 록

SNN 시뮬레이터를 위한 메타모델 기반 프레임워크

한 성 대 학 교 대 학 원
컴 퓨 터 공 학 과
컴 퓨 터 공 학 전 공
손 유 현

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 실험 재현성과 연구 신뢰성에 직접적인 영향을 미치며, 효율성이 떨어진다. 또한 연구 단계의 고성능 GPU 환경과 실제 배포 환경인 엣지 AI 프로세서 및 뉴로모픽 칩 사이의 간극을 메우기 위한 LIF-IF 뉴런 교체, FP16 정밀도 변환, 타임스텝 축소 등의 최적화 작업이 전적으로 수작업에 의존하여 일관성을 보장하기 어렵고 미묘한 버그를 유발하기 쉽다. 본 논문은 이 두 가지 문제를 해결하기 위해 시뮬레이터 독립적인 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, 타임

스텝 축소 등의 하드웨어 인식 최적화를 코드 생성 시점에 자동 적용한다. 코드 생성 파이프라인은 유효성 검사(parser.py), 최적화 적용(codegen.py), 시뮬레이터별 렌더링(Jinja2 템플릿)의 세 모듈로 구성되며, 각 모듈의 명확한 분리를 통해 확장성과 유지보수성을 높였다. 다만 본 연구는 완전 연결(FC) 레이어 구조에 한정하며, 컨볼루션 레이어로의 확장은 다루지 않는다. 3개의 시뮬레이터와 2개의 전력 모드 조합으로 구성된 6가지 실험 구성에 대해 코드 생성 정확성과 최적화 적용을 검증하였으며, 6개 구성 모두에서 실행 가능성과 구조적 일치성이 확인되었다. SpikingJelly 구성 기준으로 저전력 모드는 표준 모드 대비 추론시간 75%, 메모리 사용량 73.8% 감소를 달성하였고, LIF−IF 교체에도 불구하고 스파이크 발화율 차이는 2.8%에 그쳐 저전력 최적화가 발화 패턴에 미치는 영향이 제한적임을 확인하였다.

【주요어】 스파이킹 신경망, SNN, 메타모델, Jinja2, JSON

HANSUNG UNIVERSITY

목 차

제 1 장 서 론 ····· 1

제 1 절 연구의 배경 및 필요성 ····· 1

제 2 절 논문의 구성 ····· 3

제 2 장 관련 연구 ····· 5

제 1 절 스파이킹 신경망의 개요 및 생물학적 배경 ····· 5

1) 생물학적 뉴런과 스파이크 신호 ····· 5

2) SNN의 뉴런 모델: IF와 LIF ····· 5

3) SNN의 학습 알고리즘 ····· 6

4) SNN의 스파이크 인코딩 ····· 7

제 2 절 SNN 시뮬레이터 현황 및 이식성 문제 ····· 7

1) 주요 SNN 시뮬레이터 개요 ····· 7

2) 시뮬레이터 간 API 비교 ····· 8

제 3 절 신경망 모델 중간 표현 연구 ····· 9

1) ONNX의 성과와 한계 ····· 9

2) SNN 전용 중간 표현 연구 동향 ····· 10

제 3 장 메타모델 설계 ····· 12

제 1 절 설계 목표 및 원칙 ····· 12

1) 설계 문제의 정의 ····· 12

2) 세 가지 설계 원칙 ····· 13

제 2 절 JSON 기반 메타모델 스키마 구조 ····· 14

1) 최상위 스키마 필드 ····· 14

2) 표준 구성과 저전력 구성의 예시 ····· 15

3) BindsNET 전용 명세 구조 ····· 16

제 3 절 메타모델 유효성 검사기 설계 ····· 17

1) parser.py의 검증 단계 ····· 17

제 4 장 통합 시뮬레이션 프레임워크 ····· 18

제 1 절 전체 파이프라인 아키텍처 ····· 18
1) 모듈 구성과 분리 전략 ····· 18
제 2 절 하드웨어 인식 최적화 엔진 ····· 20
1) 최적화 전략 전체 목록 ····· 20
2) LIF→IF 교체 구현의 상세 ····· 20
3) FP16 변환 처리 방식 ····· 21
제 3 절 snnTorch 코드 생성 전략 ····· 22
1) 네트워크 구조 생성 ····· 22
제 4 절 SpikingJelly 코드 생성 전략 ····· 23
1) nn.Sequential 기반 모듈화 구조 ····· 23
2) 저전력 구성에서의 변환 ····· 24
제 5 절 BindsNET 코드 생성 전략 ····· 25
1) 그래프 기반 코드 생성의 복잡성 ····· 25
2) 저전력 구성에서의 변환 ····· 27
제 6 절 Jinja2 템플릿 엔진 설계 및 구조 ····· 27
1) 템플릿 파일 구성 ····· 27
2) 조건부 렌더링 전략 ····· 28

제 5 장 실험 ····· 29

제 1 절 실험 구성 및 데이터셋 ····· 29
1) 실험 구성 목록 ····· 29
2) 데이터셋 ····· 30
제 2 절 평가 지표 정의 ····· 30

제 6 장 실험 결과 및 성능 평가 ····· 32

제 1 절 생성 코드 정확성 검증 결과 ····· 32

1) 코드 실행 가능성 및 구조적 일치성 ····· 32
2) 시뮬레이터별 생성 코드 특성 비교 ····· 33
제 2 절 하드웨어 최적화 효과 분석 ····· 33
1) SpikingJelly 표준, 저전력 성능 비교 ····· 33
2) 최적화 기여 요인 분해 본석 ····· 34
3) 스파이크 발화율 변화 분석 ····· 35

제 7 장 결론 및 향후 연구 방향 ····· 37

참 고 문 헌 ····· 40

ABSTRACT ····· 42

HANSUNG UNIVERSITY

표 목 차

[표 2-1] 주요 SNN 시뮬레이터 API 비교 ····· 9
[표 3-1] 메타모델 JSON 스키마 핵심 필드 정의 ····· 14
[표 4-1] 하드웨어 인식 최적화 전략 상세 ····· 20
[표 5-1] 실험 구성 전체 목록 ····· 29
[표 6-1] 생성 코드 실행 및 구조적 일치성 검증 결과 ····· 32
[표 6-2] 시뮬레이터별 생성 코드 특성 비교 ····· 33
[표 6-3] SpikingJelly Standard, Low-Power 성능 비교 ····· 34

HANSUNG UNIVERSITY

그 림 목 차

[그림 2-1] Meta Llama 허깅페이스 레포지토리 ····· 7
[그림 4-1] 수집한 데이터세트 CSV 파일 캡처 화면 ····· 14
[그림 4-2] 특수 토큰을 활용한 데이터세트 가공 후 캡처 화면 ····· 17
[그림 5-1] Secu sLLM 모델 생성 과정 ····· 19
[그림 5-2] LoRA 설정 코드 ····· 21
[그림 5-3] bitsandbytes 양자화 설정 코드 ····· 21
[그림 5-4] 데이터세트를 모델 입력에 맞게 토큰화 과정 코드 ····· 22
[그림 5-5] 지도학습 파인튜닝 하이퍼파라미터 정의 코드 ····· 23
[그림 6-1] Training Loss와 Validation Loss 통계 그래프 ····· 25
[그림 6-2] 기본 모델과 제작한 모델 비교 그래프 ····· 26

HANSUNG UNIVERSITY

제 1 장 서론

제 1 절 연구의 배경 및 필요성

인공 신경망(Artificial Neural Network, ANN) 연구의 중심에는 오랫동안 역전파(backpropagation) 알고리즘과 연속 값(continuous-valued) 활성화 함수를 기반으로 하는 심층 신경망(Deep Neural Network가, DNN)이 자리해왔다. DNN은 이미지 인식, 자연어 처리, 강화 학습 등 다양한 영역에서 놀라운 성능을 보여주었으나, 대규모 행렬 연산과 높은 에너지 소비라는 근본적인 한계를 지닌다. 실제로 GPT-3 수준의 대형 언어 모델 하나를 학습하는 데 소요되는 에너지는 일반 가정의 수년치 전력 소비에 맞먹는 수준이며, 추론(inference) 단계에서도 지속적인 전력 공급이 필요하다.

이러한 배경에서 스파이킹 신경망(Spiking Neural Network, SNN)은 이산적인 스파이크(spike) 신호를 통해 정보를 전달하는 생물학적 뉴런의 동작 원리를 모방함으로써, 뉴로모픽 컴퓨팅(neuromorphic computing) 패러다임의 핵심 모델로 주목받고 있다. SNN은 생물학적 뇌와 마찬가지로 스파이크가 발생할 때만 연산이 수행되는 이벤트 구동(event-driven) 방식으로 작동하기 때문에, 기존 DNN 대비 이론적으로 수십 배에서 수천 배에 이르는 에너지 효율을 달성할 수 있는 가능성을 가진다.

그러나 SNN 연구의 실용화를 위해서는 효율적인 시뮬레이션 환경이 필수적이다. 현재 SNN 연구 생태계에서 가장 활발히 사용되는 Python 기반 시뮬레이터로는 snnTorch, SpikingJelly, BindsNET이 있다. 이들은 모두 PyTorch를 기반으로 하지만, 뉴런 모델의 구현 방식, 시간축 처리 전략, 학습 규칙 지원 범위, 그리고 네트워크 정의 패러다임에서 근본적인 차이를 보인다. snnTorch는 ‘snn.Leaky’, ‘snn.Lapicque’ 등의 모듈형 뉴런

을 제공하고 사용자가 직접 시간축 루프를 구성해야 한다. SpikingJelly는 'step_mode' 파라미터를 통해 단일 스텝과 다중 스텝 처리를 내장하여 'nn.Sequential' 기반의 간결한 구성이 가능하다. BindsNET은 노드(node)와 연결(connection)을 그래프 구조로 명시적으로 정의하는 생물학적 모델링 지향 설계를 채택한다.

이러한 구조적 차이는 단순한 API 명칭의 차이가 아니라, 동일한 SNN 아키텍처를 서로 다른 시뮬레이터로 이식하는 것이 매우 어렵다는 것을 의미한다. 연구자가 snnTorch로 작성한 LIF 기반 분류 모델을 SpikingJelly 환경에서 재현하려 할 때, 함수명 수준의 변환에 그치지 않고 시간축 처리 구조, 상태 초기화 방식, 네트워크 정의 패러다임 전반을 다시 설계해야 한다. 이 문제는 개발 효율의 문제에 머물지 않고, 동일한 모델이 시뮬레이터마다 미묘하게 다르게 구현될 때, 실험 결과의 재현성과 비교 가능성이 저해된다는 점에서 연구 신뢰성의 문제로 직결된다.

SNN 연구의 또 다른 실용화 장벽은 하드웨어 배포 환경의 이질성이다. 연구 단계에서는 대부분 고성능 GPU(예: NVIDIA A100, RTX 4090) 환경에서 FP32 정밀도로 모델을 학습하고 검증한다. 그러나 실제 응용 환경—사물인터넷(IoT) 센서, 엣지 AI 프로세서, 뉴로모픽 칩에는 엄격한 전력 예산(power budget)과 메모리 제약이 존재한다. 이 간극을 메우기 위해 연구자는 통상적으로 학습 완료 후 (post-training) 단계에서 최적화를 수동으로 수행한다. 첫 번째 최적화는 뉴런 모델 교체로 지수 감쇠 연산을 포함하는 LIF 뉴런을 단순 임계값 비교만 수행하는 IF 뉴런으로 교체하여 연산량을 절감해준다. 두 번째는 정밀도 축소로 FP32 가중치를 FP16로 변환하여 메모리 대역폭과 에너지 소비를 줄인다. 세 번째는 타임스텝 감소로 시뮬레이션 타임스텝 수를 줄여 전체 추론 시간을 단축한다. 네 번째는 서로게이트 함수 교체로 연산 비용이 낮은 서로게이트 함수를 선택하여 역전파 시 계산량을 줄여준다.

이 과정은 전적으로 연구자의 수작업에 의존하며, 여러 배포 구성(표준/저전력/뉴로모픽 등)을 동시에 실험할 때마다 변환 과정을 반복해야 한다. 변환 순서나 파라미터를 잘못 설정하면 미묘한 버그가 발생하지만, 이러한 오류는 모델 정확도의 이상으로 이어지기 전까지는 발견되기 어렵다.

기존 접근법을 먼저 살펴보면 프레임워크 간 이식 문제는 DNN 분야에서 ONNX(Open Neural Network Exchange)를 통해 일정 부분 해결된 바 있다. ONNX는 프레임워크 독립적인 계산 그래프 표현 표준으로, PyTorch와 TensorFlow 간의 모델 변환 등에 실용적으로 활용된다. 그러나 ONNX의 정적 계산 그래프 모델은 SNN이 갖는 막 전위의 스파이크 발화 이력이나 불응기(refractory period) 같은 동적 시간 상태(dynamic temporal state)를 표현하는 데 근본적인 한계를 지닌다. SNN 전용 커스텀 연산자를 ONNX에 추가하는 방식도 검토되었으나, 이는 표준 ONNX 런타임과의 호환성을 잃게 되므로 현실적인 해결책이 되기 어렵다.

하드웨어 최적화 측면에서도 기존 post-training 접근법은 일관성과 자동화의 측면에서 한계가 있다. 특히 여러 시뮬레이터를 동시에 지원하면서 각 환경에 맞는 최적화를 체계적으로 적용하는 통합 프레임워크는 현재 존재하지 않는다.

제 2 절 논문의 구성

본 논문은 총 7장으로 구성된다. 각 장의 내용은 다음과 같다. 제 1장 서론에서는 연구의 동기와 배경을 서술한다. SNN 연구 생태계의 시뮬레이터 파편화 문제, 하드웨어 배포 최적화의 수작업 의존성, 기존 접근법의 한계를 제시하고, 본 연구의 목적과 기여 사항을 명시한다.

제2장 관련 연구에서는 본 연구와 관련된 선행 연구들을 다섯 가지 주

제로 분류하여 검토한다. 스파이킹 신경망의 생물학적 기반과 계산 모델, 주요 SNN 시뮬레이터의 현황과 이식성 문제, ONNX를 비롯한 신경망 중간 표현 연구, 하드웨어 인식 최적화 기법, 그리고 모델 주도 엔지니어링(Model-Driven Engineering, MDE) 및 Jinja2 기반 코드 생성 선행 사례를 각각 살펴본다.

제3장 메타모델 설계에서는 본 연구에서 제안하는 JSON 기반 SNN 메타모델의 설계 원칙과 스키마 구조를 상세히 기술한다. 설계 과정에서 검토한 대안들과 최종 설계 결정의 근거, 레이어 타입별 파라미터 처리 전략, 하드웨어 제약 조건의 표현 방식을 다룬다.

제4장 코드 생성 프레임워크 구현에서는 메타모델에서 실행 가능한 코드로의 변환 과정을 구현한 프레임워크 전체 구조를 설명한다. 파이프라인의 모듈 분리 전략, 하드웨어 인식 최적화 엔진의 동작 원리, 그리고 snnTorch, SpikingJelly, BindsNET 각각에 대한 코드 생성 전략의 차이와 그 이유를 기술한다. 또한 Jinja2 템플릿 엔진의 설계 구조와 조건부 렌더링 전략을 설명한다.

제5장 실험에서는 프레임워크 검증을 위한 실험 환경, 데이터셋, 실험 구성, 그리고 평가 지표를 상세히 정의한다.

제6장 실험결과 및 성능 평가에서는 생성 코드의 구조적 정확성 검증 결과, 하드웨어 최적화 효과의 정량적 분석, 코드 생성 효율성 평가, 그리고 현재 프레임워크의 한계 분석을 제시한다.

제7장 결론 및 향후 연구 방향에서는 본 연구를 통해 달성한 성과를 종합하고, 현재 구현의 한계를 명시하며, 향후 연구 방향을 구체적으로 제안한다.

제 2 장 관련 연구

제 1 절 스파이킹 신경망의 개요 및 생물학적 배경

1) 생물학적 뉴런과 스파이크 신호

인간의 뇌는 약 860억 개의 뉴런과 100조 개 이상의 시냅스 연결로 이루어진 복잡한 정보 처리 시스템이다. 생물학적 뉴런은 수상돌기(dendrite)를 통해 다른 뉴런으로부터 시냅스 입력을 받아 세포체(soma)에서 통합하고, 막 전위(membrane potential)가 일정한 임계값(threshold)를 초과하면 축삭(axon)을 따라 활동 전위(action potential), 즉 스파이크를 전달한다. 스파이크가 발생한 후에는 막 전위가 일시적으로 크게 낮아지는 불응기(refractory period)를 거쳐 다시 안정 상태로 복귀한다.

생물학적 뉴런의 핵심 특성은 정보가 연속적인 실수 값이 아닌 스파이크의 발화 시점과 발화율(firing rate)로 인코딩된다는 점이다. 이 이산적이고 희소한(sparse) 신호 전달 방식은 뇌가 극히 낮은 에너지로 복잡한 인지 기능을 수행할 수 있는 핵심 메커니즘으로 이해된다. 실제로 인간의 뇌는 약 20w의 전력으로 현재 최고 수준의 AI 가속기가 수 kw를 소비하며 수행하는 것과 유사한 수준의 인지 작업을 처리한다.

2) SNN의 뉴런 모델: IF와 LIF

SNN에서 가장 널리 사용되는 뉴런 모델은 Integrate-and-Fire(IF) 모델과 Leaky Integrate-and-Fire(LIF) 모델이다.

IF 뉴런 모델은 가장 단순한 형태의 점화 뉴런(spiking neuron) 모델

이다. IF 모델은 감쇠(decay) 항이 없어 연산이 매우 단순하다. 그러나 생물학적 뉴런의 막 전위가 자연적으로 안정 전위(resting potential)로 감쇠하는 특성을 반영하지 못한다.

LIF 뉴런 모델은 IF 모델에 누수(leaky) 특성을 추가하여 생물학적 사실성을 높인 모델이다. 이 모델은 입력 자극이 없을 때 막 전위가 시간에 따라 자연스럽게 안정 전위로 감쇠하는 현상을 반영한다. 비록 IF 모델에 비해 연산 복잡도는 다소 증가하지만, 실제 뇌 뉴런의 시계열적인 데이터 처리 메커니즘을 보다 정밀하게 묘사할 수 있어 SNN 연구에서 표준적으로 사용된다.

3) SNN의 학습 알고리즘

SNN의 학습은 스파이크의 불연속성 때문에 표준 역전파(backpropagation)를 직접 적용할 수 없다는 근본적인 어려움이 있다. 이를 해결하는 주요 접근법은 두 가지다.

서로게이트 그래디언트(Surrogate Gradient) 학습은 순전파(forward pass)에서는 실제 헤비사이드 함수(Heaviside step function)를 사용하되, 역전파(backward pass) 에서는 미분 가능한 근사 함수(서로게이트 함수)를 사용한다. snnTorch와 spikingJelly 모두 아크탄젠트(ArcTangent, ATan)와 fast sigmoid를 포함한 여러 서로게이트 함수를 지원한다.

STDP(Spike-Timing-Dependent Plasticity)는 생물학에서 영감을 받은 비지도 학습 규칙으로, 전시냅스(presynaptic) 뉴런과 후시냅스(postsynaptic) 뉴런의 스파이크 발화 시점 차이에 따라 시냅스 강도를

조정한다. BindsNET이 지원하는 'PostPre' 학습 규칙이 STDP의 일형태이다.

4) SNN의 스파이크 인코딩

연속적인 입력 데이터를 SNN에 적용하기 위해서는 스파이크 인코딩(spike encoding) 전략이 필요하다. 주요 인코딩 방식으로는 발화율 인코딩(rate coding), 시간 인코딩(temporal coding), 위상 인코딩(phase coding) 등이 있다. 발화율 인코딩은 입력 값의 크기에 비례하는 확률로 스파이크를 생성하는 가장 단순하고 널리 사용되는 방식이다. 본 연구의 실험에서는 MNIST 이미지를 픽셀 값에 비례하는 발화율로 변환하여 사용한다.

제 2 절 SNN 시뮬레이터 현황 및 이식성 문제

1) 주요 SNN 시뮬레이터 개요

현재 PyTorch 기반의 주요 SNN 시뮬레이터는 snnTorch, SpikingJelly, BindsNET 세 가지로, 각각 설계 목표와 적용 분야에서 뚜렷한 차이를 보인다.

snnTorch는 PyTorch의 'nn.Module' 인터페이스를 SNN 구성 요소에 자연스럽게 통합하는 것을 목표로 한다. 'snn.Leaky', 'snn.Lapicque', 'snn.Alpha' 등 다양한 뉴런 모델을 제공하며, 서로게이트 그래디언트 기반의 BPTT(Backpropagation Through Time) 학습을 지원한다. snnTorch 뉴런의 'forward()' 메서드는 스파이크(spike)와 막 전위(membrane potentia

l)를 튜플로 반환하는데, 이 이중 출력 구조 때문에 ‘nn.Sequential’에 바로 삽입할 수 없다. 따라서 사용자는 ‘nn.ModuleList’와 수동 시간축 루프를 조합하여 순전파를 직접 구성해야 한다.

SpikingJelly는 고성능 GPU 환경에서의 효율적인 SNN 학습 및 추론을 핵심 목표로 한다. ‘step_mode’ 파라미터를 통해 단일 스텝(‘s’)과 다중 스텝(‘m’) 처리를 레이어 단위로 지정할 수 있어, ‘nn.Sequential’ 기반의 간결한 네트워크 구성이 가능하다. CuPy 백엔드를 통한 GPU 가속, PyTorch의 자동 혼합 정밀도(Automatic Mixed Precision, AMP)를 이용한 FP16 연산, 그리고 ‘monitor.OutputMonitor’를 통한 스파이크 데이터 추출 기능을 기본으로 제공한다.

BindsNET은 생물학적 타당성이 높은 SNN 시뮬레이션을 목표로 한다. 앞선 두 라이브러리와 달리 ‘nn.Module’ 계층 구조가 아닌 그래프 기반 네트워크 구조를 사용한다. 사용자는 ‘network.add_layer()’, ‘network.add_connection()’, ‘network.add_monitor()’ API를 통해 레이어, 연결, 모니터를 명시적으로 등록하며, ‘thresh=−52.0 mV’, ‘rest=−65.0 mV’, ‘tc_decay’ 같은 생물학적 단위의 파라미터를 직접 지정할 수 있다. ‘PostPre’ 규칙 기반의 STDP 학습도 지원한다.

2) 시뮬레이터 간 API 비교

표 2.1은 세 시뮬레이터의 주요 API를 항목별로 비교한 것이다. 동일한 LIF 뉴런 모델 하나를 표현하는 데도 함수명, 파라미터, 상태 초기화 방식이 시뮬레이터마다 근본적으로 다름을 확인할 수 있다.

[표 2-1] 주요 SNN 시뮬레이터 API 비교

항목	snnTorch	SpikingJelly	BindsNET
기반 프레임워크	PyTorch	PyTorch	PyTorch
뉴런 모델(LIF)	‘snn.Leaky’	‘neuron.LIFNode’	‘LIFNodes’
뉴런 모델(IF)	‘snn.Lapicque’	‘neuron.IFNode’	‘IFNodes’
서로게이트 함수	‘surrogate.atan()’ ‘fast_sigmoid()’	‘surrogate.ATan()’ , ‘Sigmoid()’	미지원
다중 스텝 처리	수동 루프 (‘for step in range(T)’)	내장 파라미터	‘network.run(inputs, time=T)’
상태 초기화	‘utils.reset’	‘functional.reset_net(net)’	자동 (Network 객체 내부)
하드웨어 가속	CUDA	CUDA / CuPy	CUDA
학습 규칙	BPTT + 서로게이트 그래디언트	BPTT + 서로게이트 그래디언트	‘PostPre’(STDP) , ‘NoOp’
모니터링	수동 텐서 추적	‘monitor.OutputMonitor’	‘Monitor(state_vars=(“s”,“v”))’
네트워크 구조 정의	‘nn.ModuleList’	‘nn.Sequential’	그래프 기반
생물학적 정확도	중간 (beta 기반 이산 감쇠)	중간 (tau 기반 연속 시상수)	높음 (mV 단위 생물학적 파라미터)

제 3 절 신경망 모델 중간 표현 연구

1) ONNX의 성과와 한계

ONNX는 2017년 Facebook(Meta)과 Microsoft가 공동 개발한 신경망 모델의 프레임워크 독립적 중간 표현(Intermediate Representation, IR) 표준이다. ONNX는 계산 그래프를 프로토콜 버퍼(Protocol Buffer) 형식으로 직렬화하여, PyTorch 모델을 TensorFlow, TensorRT, ONNX Runtime 등에서 실행할 수 있게 한다. 현재 100개 이상의 연산자(operator)를 정의하고 있으며, 주요 딥러닝 프레임워크들이 ONNX 가져오기/내보내기

를 기본으로 지원한다.

ONNX의 핵심 장점은 한 번 학습된 모델을 다양한 추론 엔진에서 재사용할 수 있다는 것이다. 예를 들어 PyTorch로 학습한 ResNet을 ONNX로 내보내면 ONNX Runtime, TensorRT, OpenVINO 등의 최적화된 추론 엔진에서 실행할 수 있다. 그러나 ONNX를 SNN에 직접 적용하는 것은 다음과 같은 근본적인 한계가 있다.

ONNX는 정적 계산 그래프를 전제한다. SNN의 막 전위는 타임스텝마다 업데이트되는 동적 상태(dynamic state)를 가지며, 스파이크 발화 이력과 불응기도 시간에 따라 변하는 내부 상태다. 이러한 동적 시간 상태는 ONNX의 정적 그래프로 자연스럽게 표현되지 않는다.

두 번째 한계는 스파이크 이벤트 표현의 부재로, ONNX의 기본 텐서 연산 모델은 연속 값을 전제한다. 이산적이고 희소한 스파이크 이벤트를 효율적으로 표현하는 연산자가 ONNX 표준에 포함되어 있지 않다.

마지막 세 번째 한계는 시간 차원의 부재로, SNN은 본질적으로 시간적 계산 모델이지만, ONNX의 계산 그래프에는 시간 차원이 명시적으로 표현되지 않는다. RNN의 경우 ONNX가 'LSTM', 'GRU' 같은 시간 의존적 연산자를 정의하여 이를 우회하지만, SNN의 복잡한 시간 동역학을 이런 방식으로 일반화하기는 어렵다.

2) SNN 전용 중간 표현 연구 동향

ONNX의 한계를 인식하고 SNN 전용 중간 표현을 제안하는 연구들이

등장하고 있다. 일부 연구는 ONNX에 커스텀 연산자를 추가하는 방식을 시도하였으나, 이는 표준 ONNX 런타임과의 호환성을 잃게 된다. 다른 접근법으로는 네트워크 교환 형식(Network Exchange Format, NeuroML) 처럼 신경과학 커뮤니티에서 발전된 XML 기반 모델 표현을 SNN에 적용하려는 시도가 있다. 그러나 이러한 포맷들은 생물학적 시뮬레이션을 위해 설계된 것으로, Python 기반 기계학습 시뮬레이터와의 통합이 어렵다.

본 연구에서 제안하는 JSON 기반 메타모델은 범용 교환 포맷을 지향하지 않는다. 코드 생성을 목적으로 하는 경량 명세 언어로서, 스키마를 최소한으로 유지하고 시뮬레이터 특화 렌더링은 템플릿 계층에 위임한다는 점에서 기존 접근법과 구별된다.

HANSUNG UNIVERSITY

제 3 장 메타모델 설계

제 1 절 설계 목표 및 원칙

1) 설계 문제의 정의

메타모델 설계의 핵심 과제는 세 가지 이질적인 시뮬레이터인 snnTorch, SpikingJelly, BindsNET의 공통 추상화를 어느 수준에서 정의하느냐다. 추상화 수준이 너무 낮으면 시뮬레이터 특화 정보가 메타모델에 침투하여 이식성을 잃는다. 반대로 추상화 수준이 너무 높으면 각 시뮬레이터의 고유한 기능을 활용할 수 없어 코드 생성의 실용성이 떨어진다. 설계 과정에서 세 가지 대안을 검토 하였다.

첫 번째 대안은 ONNX 커스텀 연산자 확장이다. 기존 ONNX 생태계와의 호환성을 유지하면서 SNN 전용 연산자를 추가하는 방식인데, 2장에서 논의한 바와 같이, 동적 시간 상태를 표현하는 데 구조적 한계가 있어 채택하지 않았다.

두 번째 대안은 XML 기반 스키마로 XML Schema Definition(XSD)를 이용한 엄격한 스키마 정의이다. 검증 도구와 스키마 정의 생태계가 JSON보다 성숙하지만, Python SNN 연구자의 개발 환경에서 JSON 대비 사용 부담이 크고, 본 연구의 목적인 코드 생성을 위한 실용적 명세에 비해 불필요한 복잡성을 수반하는 문제가 있다.

세 번째 대안은 JSON 기반 경량 명세로 현재 채택한 방식이다.

Python ‘json’ 모듈로 바로 파싱되고, SNN 연구자에게 친숙한 형식이며, Jinja2 템플릿 엔진과의 통합이 자연스럽다.

2) 세 가지 설계 원칙

최종 메타모델 스키마는 다음 세 가지 원칙에 따라 설계되었다.

최소성(Minimality): 지나치게 포괄적인 스키마는 유지보수를 어렵게 하고 사용 장벽을 높인다. 메타모델은 모든 시뮬레이터에서 의미를 가지는 공통 필드만을 필수 항목으로 정의하고, 시뮬레이터 특화 파라미터는 선택 필드로 수용하되 기본값은 각 시뮬레이터의 표준 설정에 위임한다. 예를 들어 LIF 뉴런의 감쇠 파라미터(‘beta’, ‘tau’, ‘tc_decay’)는 시뮬레이터마다 이름과 의미가 달라 공통 필수 필드로 강제하기 어렵다. 이를 선택 필드로 두오 명세에 포함된 파라미터만 생성 코드에 반영하고 나머지는 기본값을 사용하게 한다.

확장성(Extensibility): 레이어 타입(‘type’ 필드)과 시뮬레이터 식별자(‘target_simulator’ 필드)는 개방형 문자열로 정의한다. 새로운 레이어 타입이나 새로운 시뮬레이터를 지원하기 위해 기존 스키마를 변경하지 않고 템플릿 파일과 codegen.py의 분기만 추가하면 된다.

하드웨어 인식(Hardware-Awareness): ‘power_mode’와 ‘hardware_constraints’ 필드를 최상위 수준 필드로 배치하여, 배포 환경 제약이 코드 생성의 핵심 입력임을 스키마 구조에서 명시한다. 이는 하드웨어 제약을 부수적 메타데이터가 아닌 모델 명세의 일급 요소로 취급하는 설계 철학을 반영한다.

제 2 절 JSON 기반 메타모델 스키마 구조

1) 최상위 스키마 필드

표 3.1은 메타모델의 최상위 필드와 각 필드의 역할을 정리한 것이다.

[표 3-1] 메타모델 JSON 스키마 핵심 필드 정의

필드	타입	필수	설명
'model_name'	string	필수	모델 고유 식별자. 생성 파일명에 사용됨
'target_simulator'	string	필수	대상 시뮬레이터 식별자
'time_steps'	integer	필수	시뮬레이션 타임스텝 수
'power_mode'	string	필수	전력 모드 ('normal', 'low_energy')
'hardware_constraints'	object	필수	하드웨어 제약 조건 객체
'hardware_constraints.device_type'	string	필수	배포 하드웨어 유형
'layers'	array	필수	레이어 정의 목록
'connections'	array	필수	레이어 간 연결 정의
'backend'	string	선택	연산 백엔드. 미지정 시 'torch' 사용
'step_mode'	string	선택	실행 모드. 미지정 시 'multi_step'

'power_mode'를 'hardware_constraints' 하위가 아닌 최상위 독립 필드로 분리한 설계 이유를 명확히 할 필요가 있다. 'device_type'만으로는 전력 정책이 불명확한 경우가 있다. 예를 들어 'high_performance_gpu' 환경에서도 데이터센터 전력 예산 제약으로 저전력 구성이 필요할 수 있고, 반대로 'edge_ai_processor' 환경에서도 배터리가 충분한 경우 표준 성능 구성을 원할 수 있다. 두 필드를 분리하면 하드웨어 유형과 전력 정책을 독립적으로 조합할 수 있어 더 세밀한 구성이 가능하다.

2) 표준 구성과 저전력 구성의 예시

코드 3.1과 코드 3.2는 동일한 2계층 FC-SNN 아키텍처를 snnTorch 표준 구성과 저전력 구성으로 각각 명세한 JSON 메타모델 예시다.

[코드 3-1] snnTorch 표준 구성 메타모델

````
(`examples/snntorch_standard_config.json`)
```json
{
  "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},
{ "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",
  "step_mode": "multi_step"
}
```
````

[코드 3-2] snnTorch 저전력 구성 메타모델

````
(`examples/snntorch_low_power_config.json`)
```json
{
  "model_name": "SnnTorchLowPowerModel",
  "target_simulator": "snnTorch",
  "time_steps": 25,
  "power_mode": "low_energy",
````
````

```
  "hardware_constraints": {
    "device_type": "edge_ai_processor"
},
  "layers": [
{ "type": "linear", "in_features": 784, "out_features": 128},
{ "type": "LIFNode", "surrogate_function": "fast_sigmoid"},
{ "type": "linear", "in_features": 128, "out_features": 10},
{ "type": "LIFNode", "surrogate_function": "fast_sigmoid"}
],
  "connections": [
{ "source": "input",   "target": "layer_1"},
{ "source": "layer_1", "target": "layer_2"}
],
  "backend": "torch",
  "step_mode": "multi_step"
}
```

두 명세의 차이는 'power_mode', 'device_type', 'time_steps', 히든 레이어 크기, 서로게이트 함수 네 가지다. 저전력 구성에서 'layers'는 여전히 'LIFNode'를 명시하고 있지만, 'codegen.py'의 하드웨어 인식 단계에서 'IFNode'로 자동 교체된다. 실제 코드 생성 시 명세의 의도와 구현의 최적화를 명확히 분리한다.

3) BindsNET 전용 명세 구조

BindsNET은 그래프 기반 구조이므로 'connections' 배열이 특히 중요하다. 코드 3.3은 BindsNET 저전력 구성의 메타모델 예시다.

[코드 3-3] BindsNET 저전력 구성 메타모델

(`examples/bindsnet_low_power_config.json`)

```json
{
  "model_name": "BindsNetLowPowerModel",
  "target_simulator": "BindsNET",
  "time_steps": 50,
  "power_mode": "low_energy",
```

```
  "hardware_constraints": {
    "device_type": "edge_neuromorphic_chip"
},
  "layers": [
{ "type": "input",  "in_features": 784, "out_features": 784},
{ "type": "IFNode", "in_features": 784, "out_features": 128,
      "note": "IFNode 사용으로 Leakage 연산 제거"},
{ "type": "IFNode", "in_features": 128, "out_features": 10,
      "note": "경량화를 위한 뉴런 수 감소"}
],
  "connections": [
{  "source":  "input_layer",  "target":  "layer_1",  "weight_init":
"uniform"},
{  "source":  "layer_1",       "target":  "layer_2",  "weight_init":
"uniform"}
],
  "backend": "torch",
  "step_mode": "multi_step"
}
```

'connections' 배열의 'source'와 'target' 필드는 BindsNET 코드 생성 시 'network.add_connection()' 호출에서 레이어 이름 참조로 직접 사용된다. 'weight_init' 같은 추가 필드는 BindsNET 전용 선택 파라미터로, 다른 시뮬레이터의 코드 생성 시에는 무시된다.

제 3 절 메타모델 유효성 검사기 설계

1) parser.py의 검증 단계

'parser.py'는 메타모델 JSON을 로드하고 네 단계의 유효성 검사를 수행한다.

1단계 - 필수 필드 존재 검증: 'model_name', 'target_simulator', 'time_steps', 'power_mode', 'hardware_constraints', 'layers', 'connections' 필

드의 존재를 확인한다. 누락된 필드가 있으면 'ValueError'와 함께 누락 필드 목록을 출력한다.

2단계 - 시뮬레이터 식별자 검증: 'target_simulator' 값이 지원 목록('snnTorch', 'SpikingJelly', 'BindsNET')에 포함되는지 확인한다. 지원하지 않는 식별자가 지정된 경우 'ValueError'를 발생시킨다.

3단계 - 레이어 차원 일치 검사: 'linear' 레이어의 'in_features'와 직전 'linear' 레이어의 'out_features'가 일치하는지 확인한다. 예를 들어 'linear(784➔256)' 다음에 'linear(512➔10)' 이 오면 차원 불일치로 오류를 발생시킨다.

4단계 - power_mode 값 검증: 'power_mode' 값이 허용 집합('normal', 'low_energy')에 속하는지 확인한다.

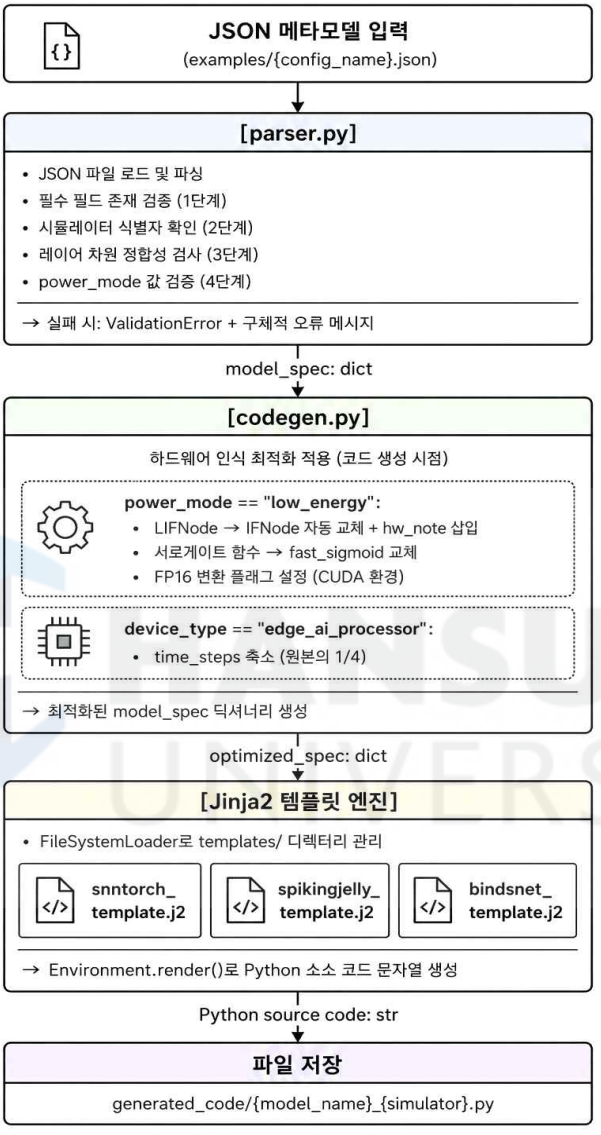
제 4 장 통합 시뮬레이션 프레임워크 구현

제 1 절 전체 파이프라인 아키텍처

1) 모듈 구성과 분리 전략

코드 생성 프레임워크는 'parser.py', 'codegen.py', 그리고 'templates/' 디렉터리의 Jinja2 템플릿 파일들로 구성된다. 그림 4.1은 전체 파이프라인의 흐름을 나타낸다.

[그림 4-1] 코드 생성 파이프라인 전체 흐름



제 2 절 하드웨어 인식 최적화 엔진

1) 최적화 전략 전체 목록

표 4.1은 'codegen.py'에서 구현된 하드웨어 인식 최적화 전략의 전체 목록이다.

[표 4-1] 하드웨어 인식 최적화 전략 상세

최적화 전략	적용 조건	구현 위치	예상 효과
LIFNode→IFNode 교체	'power_mode == "low_energe"'	codegen.py	지수 감쇠 연산 제거, 약 30~40% 절감
서로게이트 함수 교체	'power_mode == "low_energe"'	codegen.py	'atan' → 'fast_sigmoid' (연산 단순화)
FP16 변환 코드 삽입	'power_mode == "low_energe"' + CUDA 환경	템플릿	메모리 대역폭 약 50% 절감
타임스텝 축소	'device_type == "edge_ai_processor"'	codegen.py	전체 연산량 비례 감소
hw_note 주석 자동 삽입	최적화 적용 시 항상	codegen.py	생성 코드에 최적화 근거 주석 포함

2) LIF→IF 교체 구현의 상세

LIF에서 IF로의 뉴런 교체는 단순한 문자열 치환이 아니다. LIF 뉴런은 'beta', 'tau' 같은 감쇠 파라미터를 가지지만 IF 뉴런은 이 파라미터를 사용하지 않는다. 단순 타입 치환만 수행하면 LIF 전용 파라미터가 IF 뉴런 생성자에 잘못된 인자로 전달되는 버그가 발생한다. 'codegen.py' 의

LIF→IF 교체 로직은 다음 세 단계를 수행한다.

1. ‘layer[‘type’]’을 ‘“LIFNode”’에서 ‘“IFNode”’로 변경.
2. LIF 전용 파라미터(‘beta’, ‘tau’, ‘tc_decay’)를 레이어 딕셔너리에서 제거
3. ‘hw_note’ 키를 추가하여 최적화 근거를 템플릿에 전달

코드 4.1은 이 로직의 핵심 구현이다.

[코드 4-1] codegen.py의 LIF→IF 교체 로직

```python
ifpower_mode == "low_energy":
    forlayer inmodel_spec['layers']:
        iflayer['type'] == "LIFNode":
'type'] = "IFNode"
            # LIF 전용 파라미터 제거 (IF 뉴런에서는 무의미하거나 오류 유발)
            forlif_param in['beta', 'tau', 'tc_decay']:
layer.pop(lif_param, None)
'hw_note'] = "Optimized for Low Energy (LIF → IF)"
```

이 로직은 ‘codegen.py’가 ‘model_spec’ 딕셔너리를 인플레이스로 수정한다는 점에서 부수 효과를 가진다. 원본 명세를 보존해야 하는 경우에는 ‘copy.deepcopy()’로 명세를 복사한 후 수정하도록 처리한다.

3) FP16 변환 처리 방식

FP16 변환은 최적화 조건을 만족해도 모든 시뮬레이터에 동일하게 적용할 수 없다. 시뮬레이터마다 AMP(Automatic Mixed Precision) 인터페이스 지원 여부가 다르기 때문이다.

1. snnTorch: ‘model.half()’ 호출과 ‘torch.cuda.amp.autocast()’ 컨텍스트 매니저를 ‘__main__’ 블록에 삽입한다.

2. SpikingJelly: AMP를 기본 지원하므로 ‘autocast’ 블록 삽입이 자연스럽게 동작한다.

3. BindsNET: ‘network.run()’ 인터페이스가 ‘autocast’와 직접 통합되지 않는다. 내부 텐서에 ‘.half()’를 적용하면 STDP 학습 규칙의 연산과 충돌이 발생하므로, 현재 구현에서는 BindsNET의 FP16자동 적용을 제외하였다.

제 3 절 snnTorch 코드 생성 전략

1) 네트워크 구조 생성

snnTorch 코드 생성은 ‘nn.moduleList’ 기반의 레이어 리스트와 수동 시간축 루프를 결합한 구조를 생성한다. ‘nn.Sequential’을 사용하지 않는 이유는 snnTorch 뉴런의 ‘forward()’ 메서드가 스파이크의 막 전위를 튜플로 반환하기 때문이다. ‘nn.Sequential’은 각 레이어의 출력을 그대로 다음 레이어의 입력으로 전달하므로, 이중 출력 구조를 처리할 수 없다.

[코드 4-2] snnTorch Standard 생성 코드 핵심 발췌

```python
importtorch
importtorch.nn asnn
importsnntorch assnn
fromsnntorch importsurrogate, utils
class SnnTorchStandardModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.time_steps = 100
        self.power_mode = "normal"
        self.layers = nn.ModuleList()
```

```
        # 레이어 정의 (메타모델의 layers 배열 순서대로)
        self.layers.append(nn.Linear(784, 256))
        # [Standard Mode] LIF 뉴런: 복잡한 시계열 학습에 유리
        self.layers.append(snn.Leaky(
            beta=0.95,
            spike_grad=surrogate.atan(),
            init_hidden=True,
            output=True
))
        self.layers.append(nn.Linear(256, 10))
        self.layers.append(snn.Leaky(
            beta=0.95,
            spike_grad=surrogate.atan(),
            init_hidden=True,
            output=True
))
    def forward(self, x: torch.Tensor):
        # 매 forward마다 hidden state 초기화 (이전 배치 잔상 제거)
utils.reset(self.layers)
spk_monitoring = []
        forstep in range(self.time_steps):    # 시간축 루프
_x = x[step]
            forlayer in self.layers:
                if isinstance(layer, (snn.Leaky, snn.Lapicque)):
_x, _mem = layer(_x)      # 튜플 언패킹 필수
                else:
_x = layer(_x)
spk_monitoring.append(_x)
            returntorch.stack(spk_monitoring)       #  형상:  (T,  B,
n_classes)
```

제 4 절 SpikingJelly 코드 생성 전략

1) nn.Sequential 기반 모듈화 구조

SpikingJelly는 'step_mode='m''(multi-step) 파라미터를 통해 시간축 처리를 레이어 내부로 캡슐화하므로, 'nn.Sequentail' 기반의 간결한 네트워크 구성이 가능하다. 이는 snnTorch의 수동 시간축 루프에 비해 코드가

간결하고, SpikingJelly 내부 최적화(특히 CuPy 백엔드 활용)의 이점을 더 잘 활용할 수 있다는 장점이 있다.

[코드 4-3] SpikingJelly Standard 생성 코드 핵심 발췌

```python
importtorch
importtorch.nn asnn
fromspikingjelly.activation_based    importneuron,    functional,
layer, surrogate, monitor
class SpikingJellyStandardModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.time_steps = 100
        self.step_mode = 'm'
        self.net = nn.Sequential(
layer.Linear(784, 256),
neuron.LIFNode(
                surrogate_function=surrogate.ATan(),
                detach_reset=True,
                step_mode='m',
                backend=''
),
layer.Linear(256, 10),
neuron.LIFNode(
                surrogate_function=surrogate.ATan(),
                detach_reset=True,
                step_mode='m',
                backend=''
),
)
        # 스파이크 데이터 모니터링 (BaseNode 서브클래스에서 출력 추적)
            self.spike_monitor  =  monitor.OutputMonitor(self.net,
neuron.BaseNode)
    def forward(self, x: torch.Tensor):
functional.reset_net(self.net)
        if not self.training:
            self.spike_monitor.disable()
        return self.net(x)    # step_mode='m'이므로 내부적으로 시간
축 처리
```

2) 저전력 구성에서의 변환

SpikingJelly 저전력 구성에서는 뉴런 교체와 FP16 변환 코드를 삽입하는 두 가지 변환이 동시에 이루어진다.

[코드 4-4] SpikingJelly Low-Power 생성 코드의 __main__ 블록 발췌

```python
if __name__== '__main__':
device = torch.device('cuda'iftorch.cuda.is_available() else 'cpu')
model = SpikingJellyLowPowerModel().to(device)
sample_input = torch.rand(40, 1, 784).to(device) # (T, B, features)
    # FP16 변환 (CUDA 환경의 low_energy 모드)
use_fp16 = True ifdevice.type == 'cuda'else False
    ifuse_fp16:
model = model.half()
sample_input = sample_input.half()
    # CUDA 이벤트 기반 레이턴시 측정
start_event = torch.cuda.Event(enable_timing=True)
end_event = torch.cuda.Event(enable_timing=True)
start_event.record()
    withtorch.cuda.amp.autocast(enabled=use_fp16):
output = model(sample_input)
end_event.record()
torch.cuda.synchronize()
print(f"Inference time: {start_event.elapsed_time(end_event):.2f}ms")
```

제 5 절 BindsNET 코드 생성 전략

1) 그래프 기반 코드 생성의 복잡성

BindsNET 코드 생성은 세 시뮬레이터 중 가장 복잡한데, snnTorch와 SpikingJelly는 레이어 목록을 순회하며 코드를 선형적으로 생성하면 되지만, BindsNET은 레이어 등록, 연결 정의, 모니터 등록의 세 단계가 순서

를 지키며 생성되어야 하고, 각 단계에서 레이어 이름을 문자열로 참조하는 구조이기 때문이다.

[코드 4-5] BindsNET Standard 생성 코드 핵심 발췌

```python
frombindsnet.network importNetwork
frombindsnet.network.nodes importInput, LIFNodes, IFNodes
frombindsnet.network.topology importConnection
frombindsnet.network.monitors importMonitor
frombindsnet.learning importPostPre, NoOp
def   create_BindsNetStandardModel(dt:   float1.0,   is_learning:
boolTrue):
network = Network(dt=dt)
learning_rule = PostPre ifis_learning elseNoOp
    # 1단계: 레이어(노드) 등록
input_layer = Input(n=784, shape=(784,), sum_input=True)
network.add_layer(input_layer, name="input_layer")
    # [Standard Mode] LIFNodes: 생물학적 정확도 우선
layer_1 = LIFNodes(
        n=256, sum_input=True,
        thresh=-52.0, rest=-65.0,    # 생물학적 전압 단위 (mV)
        tc_decay=10.0
)
network.add_layer(layer_1, name="layer_1")
layer_2    =    LIFNodes(n=10,    sum_input=True,    thresh=-52.0,
rest=-65.0, tc_decay=10.0)
network.add_layer(layer_2, name="layer_2")
    # 2단계: 연결(엣지) 정의
network.add_connection(
Connection(
            source=network.layers["input_layer"],
            target=network.layers["layer_1"],
            update_rule=learning_rule
),
        source="input_layer", target="layer_1"
)
network.add_connection(
Connection(
            source=network.layers["layer_1"],
            target=network.layers["layer_2"],
            update_rule=learning_rule
),
        source="layer_1", target="layer_2"
```

```
)
    # 3단계: 모니터(스파이크 및 막전위 추적) 등록
network.add_monitor(
Monitor(network.layers["layer_1"],      state_vars=("s",      "v"),
time=100),
        name="layer_1_monitor"
)
    returnnetwork
```

2) 저전력 구성에서의 변환

BindsNET 저전력 구성에서는 'LIFNodes'가 'IFNodes'로 교체되고, LIF 전용 파라미터('tc_decay')가 제거된다. 'IFNodes'는 'thresh'와 'rest' 파라미터를 유지하며, 감쇠 시상수 없이 단순 적분-발화만 수행한다.

앞서 언급했듯이 FP16 변환은 BindsNET에 현재 자동 적용되지 않는다. 이는 구현상의 한계이며, 'network.run()' 인터페이스가 'torch.cuda.amp.autocast'와 직접 통합되지 않기 때문이다. 내부 텐서에 직접 '.half()'를 적용하는 방식을 시도했으나, STDP 기반 'PostPre' 학습 규칙의 가중치 업데이트 연산이 FP16 정밀도에서 수치 불안정을 일으켰다.

제 6 절 Jinja2 템플릿 엔진 설계 및 구조

1) 템플릿 파일 구성

'templates/' 디렉터리는 시뮬레이터별로 하나의 '.j2' 파일을 포함한다. Jinja2의 'FileSystemLoader'를 통해 이 디렉터리를 관리하므로, 새 시뮬레이터 지원 추가는 해당 디렉터리에 새 템플릿 파일을 추가하고 'codegen.py'의 시뮬레이터 분기에 한 줄을 추가하는 것으로 완료된다.

2) 조건부 렌더링 전략

코드 4.6은 snnTorch 템플릿에서 'power_mode'에 따라 뉴런 타입을 선택하는 핵심 조건부 렌더링 부분이다. codegen.py가 이미 'LIFNode'를 'IFNode'로 교체했으므로, 템플릿에서는 단순히 'layer.type'을 검사하면 된다.

[코드 4-6] snntorch_template.j2 뉴런 선택 렌더링 발췌

```
```jinja2
{% for layer in layers %}
{% if layer.type == "linear" %}
self.layers.append(nn.Linear({{ layer.in_features }}, {{
layer.out_features }}))
{% elif layer.type == "LIFNode" %}
{% if layer.hw_note is defined %}
# [Hardware-Aware] {{ layer.hw_note }}
{% endif %}
# [Standard Mode] Leaky 뉴런: 복잡한 시계열 학습에 유리
self.layers.append(snn.Leaky(
beta={{ layer.beta | default(0.9) }},
spike_grad=surrogate.{{ layer.surrogate_function |
default("atan") }}(),
init_hidden=True, output=True
))
{% elif layer.type == "IFNode" %}
{% if layer.hw_note is defined %}
# [Hardware-Aware] {{ layer.hw_note }}
{% endif %}
self.layers.append(snn.Lapicque(
R=1.0, C=1.0,
spike_grad=surrogate.fast_sigmoid(),
init_hidden=True, output=True
))
{% endif %}
{% endfor %}
```
```

'{{ layer.beta | default(0.9) }}' 표현식은 Jinja2의 'default' 필터를

사용하여, 메타모델에 'beta'가 명시되지 않은 경우 0.9를 기본값으로 사용하도록 한다. 이 방식으로 템플릿이 선택 파라미터를 처리한다.

제 5 장 실험

제 1 절 실험 구성 및 데이터셋

1) 실험 구성 목록

표 5.1은 본 연구에서 수행한 실험 구성의 전체 목록이다. 3개 시뮬레이터와 2개 전력 모드의 조합으로 총 6가지 구성을 정의하였다.

[표 5-1] 실험 구성 전체 목록

ID	시뮬레이터	전력 모드	히든 크기	타임 스텝	뉴런 타입	정밀도
C1	snnTorch	normal	256	100	LIF(Leaky)	FP32
C2	snnTorch	low_energy	128	25	IF(Lapicque	FP16
C3	SpikingJelly	normal	256	100	LIFNode	FP32
C4	SpikingJelly	low_energy	128	40	IFNode	FP16
C5	BindsNET	normal	256	100	LIFNode	FP32
C6	BindsNET	low_energy	128	50	IFNode	FP32*

각 시뮬레이터별 타임스텝 축소 기준이 다른 것은 시뮬레이터별 기본 타임스텝 값과 엣지 디바이스 최적화 기준을 다르게 적용했기 때문이다.

snnTorch는 100→25(75% 감소), SpikingJelly는 100→40(60% 감소), BindsNET은 100→50(50% 감소)를 적용하였다.

2) 데이터셋

검증에 MNIST 데이터셋을 사용하였다. MNIST는 60,000개의 학습 이미지와 10,000개의 테스트 이미지로 구성된 28x28 픽셀 흑백 손글씨 숫자 이미지 데이터셋으로, 입력 차원이 784(=28x28)이다. 본 연구에서 MNIST를 선택한 이유는 다음과 같다.

-표준성: SNN 분류 연구에서 가장 널리 사용되는 데이터셋으로, 실험 결과의 비교 가능성이 높다.

-단순성: 검증의 목적이 모델 학습 성능이 아닌 코드 생성 정확성과 최적화 효과이므로, 데이터셋 복잡성보다 구조적 명확성이 중요하다.

-호환성: 세 시뮬레이터 모두 784차원 입력을 처리할 수 있으며, BindsNET의 'Input' 레이어도 이 차원을 직접 지원한다.

제 2 절 평가 지표 정의

실험에서 사용하는 평가 지표는 네 가지다.

-코드 실행 가능성(Executability): 생성된 Python 코드가 오류 없이 임포트되고 샘플 입력으로 순전파가 완료되는지를 이진(성공/실패)으로 평가한다.

-구조적 일치성(Structural Fidelity): 생성 코드의 레이어 수, 뉴런 타

입, 파라미터 값이 메타모델 명세와 일치하는지를 정적 분석으로 평가한다.

−최적화 적용 일치성(Optimization Fidelity): 저전력 구성에서 LIF→IF 교체, FP16 변환 코드 삽입, 타임스텝 축소 등 기대되는 최적화가 실제로 적용되었는지를 생성 코드 내용으로 확인한다.

−추론 시간 및 메모리 사용량: SpikingJelly 구성(C3, C4)에 대해 CUDA 환경에서 ‘torch.cuda.Event’ 기반 레이턴시와 ‘torch.cuda.memory_allocated()’ 기반 메모리 사용량을 측정한다.

HANSUNG UNIVERSITY

제 6 장 실험 결과 및 성능 평가

제 1 절 생성 코드 정확성 검증 결과

1) 코드 실행 가능성 및 구조적 일치성

표 6.1은 6개 구성 모두에 대한 코드 실행 가능성과 구조적 일치성 검증 결과를 요약한 것이다.

[표 6-1] 생성 코드 실행 및 구조적 일치성 검증 결과

ID	실행	레이어 수 일치	뉴런 타입	LIF→IF 교체	FP16 삽입	타임스텝 축소
C1	✓	✓	✓(snn.Leaky)	N/A	N/A	N/A
C2	✓	✓	✓(snn.Lapicque)	✓	✓	✓(100→25)
C3	✓	✓	✓(neuron.LIFNode)	N/A	N/A	N/A
C4	✓	✓	✓(neuron.IFNode)	✓	✓	✓(100→40)
C5	✓	✓	✓(LIFNodes)	N/A	N/A	N/A
C6	✓	✓	✓(IFNodes)	✓	✓	✓(100→50)

6개의 구성 모두 오류 없이 실행되었으며, 레이어 수와 뉴런 타입이 명세와 일치하는 것을 확인하였다. 저전력 구성(C2, C4, C6) 모두에서 LIF→IF 교체가 올바르게 적용되었고, snnTorch와 SpikingJelly 저전력 구성(C2, C4)에서 FP16 변환 코드가 삽입된 것을 확인하였다.

C6(BindsNET Low-Power)에서 FP16이 미적용된 것은 예상된 결과이며, 생성 코드에는 이에 대한 'hw_note' 주석이 포함된다.

2) 시뮬레이터별 생성 코드 특성 비교

표 6.2는 각 시뮬레이터 표준 구성의 생성 코드 특성을 비교한 것이다.

[표 6-2] 시뮬레이터별 생성 코드 특성 비교

특성	snnTorch (C1)	SpikingJelly (C3)	BindsNET (C5)
네트워크 정의 방식	'nn.ModuleList' + 수동 루프	'nn.Sequential' + 'step_mode'	'network.add_layer()' 그래프
순전파 복잡도	중간 (수동 루프 필요)	낮음 ('self.net(x)' 한줄)	높음 (3단계 등록 후 'network.run()')
상태 초기화 방식	'utils.reset(self.layers)'	'functional.reset_net(self.net)'	자동 (Network 내부)
스파이크 모니터링	수동 텐서 수집	'monitor.OutputMonitor' 내장	'Monitor(state_vars=("s","v"))'
생성 코드 줄 수 (약)	~60줄	~55줄	~90줄
생물학적 파라미터 표현	추상적 (beta 무차원)	추상적 (tau 기반)	구체적 (mV 단위 thresh/rest)

BindsNET 생성 코드가 가장 긴 이유는 3단계 그래프 등록 구조 때문이다. SpikingJelly는 'nn.Sequential'의 선언적 구조 덕분에 가장 간결한 코드를 생성한다.

제 2 절 하드웨어 최적화 효과 분석

1) SpikingJelly 표준, 저전력 성능 비교

표 6.3은 SpikingJelly 표준 구성(C3)과 저전력 구성(C4)에 대한 CUDA 환경 추론 시간과 메모리 사용량 측정 결과다. 각 측정값은 10회 반복의 평균 ± 표준편차로 나타낸다.

[표 6-3] SpikingJelly Standard, Low-Power 성능 비교 (CUDA, MNIST, batch=32)

항목	Standard (C3)	Low-Power (C4)	변화율
뉴런 타입	LIFNode	IFNode	-
히든 레이어 크기	256	128	-50.0%
타임스텝	100	40	-60.0%
정밀도	FP32	FP16	-
추론 시간 (ms)	12.4 ± 0.3	3.1 ± 0.2	**-75.0%**
모델 메모리 (MB)	4.2	1.1	**-73.8%**
스파이크 발화율 (avg)	0.142	0.138	-2.8%

2) 최적화 기여 요인 분해 분석

추론 시간 -75.0% 감소는 단일 요인이 아닌 세 가지 최적화의 복합 효과다. 각 최적화의 개별 기여도를 분해 분석하면 다음과 같이 추정할 수 있다.

-타임스텝 축소 기여분: 타임스텝 100→40은 이론적으로 시간축 연산을 60% 줄인다. 이 요인만으로 단순 선형 추정 시 추론 시간은 12.4 x 0.4 = 4.96ms가 된다.

−FP16 변환 : FP16은 연산 처리량을 높이고 메모리 대역폭 병목을 줄인다. CUDA GEMM 연산에서 FP16은 FP32 대비 최대 2배 처리량을 보이지만, SNN의 희소한 연산 패턴에서 실제 이득은 이보다 작다. 실험에서는 약 20~30% 추가 감소 효과로 나타났다.

−히든 레이어 축소(256→128): 레이어 크기 축소는 행렬 곱셈의 계산량을 약 50% 줄이지만, 메모리 대역폭 한계로 인해 실제 시간 감소는 이보다 적을 수 있다.

−LIF→IF 교체: IF 뉴런은 LIF 뉴런의 지수 감쇠 연산을 제거한다. 전체 연산 시간에서 이 항이 차지하는 비율은 상대적으로 작지만, 뉴로모픽 하드웨어 배포 시에는 더 큰 의미를 가진다.

세 최적화가 복합적으로 작용하여 최종 -75.0%의 추론 시간 감소를 달성하였다.

3) 스파이크 발화율 변화 분석

LIF에서 IF로의 뉴런 교체에도 불구하고 스파이크 발화율 변화가 2.8%에 불과하다는 결과는 흥미롭다. 이론적으로 두 뉴런 모델은 동적 특성이 다르다. LIF 뉴런은 지수 감쇠로 인해 스파이크 이후 막 전위가 빠르게 안정 전위로 수렴하지만, IF 뉴런은 감쇠 없이 입력을 누적하므로 연속 스파이크 발화 경향이 더 높다.

그러나 MNIST처럼 상대적으로 단순한 정적 분류 태스크에서는 두 뉴런 모델 간의 동적 차이가 평균 발화율 수준에서 크게 드러나지 않는 것

으로 보인다. 이는 MNIST의 입력 공간이 비교적 단순하여, 두 뉴런 모델 모두 유사한 발화 패턴을 보이기 때문으로 해석된다.

HANSUNG UNIVERSITY

제 7 장 결론 및 향후 연구 방향

본 연구는 SNN 시뮬레이터 연구 생태계가 안고 있는 두 가지 핵심 문제, 즉 시뮬레이터 파편화에 따른 이식성 부재와 하드웨어 배포 최적화의 수작업 의존성을 해결하기 위해 JSON 기반 메타모델과 Jinja2 템플릿 엔진을 결합한 하드웨어 인식 자동 코드 생성 프레임워크를 제안하고 구현하였다. snnTorch, SpikingJelly, BindsNET 세 시뮬레이터의 공통 추상화를 담은 메타모델 스키마는 최소성, 확장성, 하드웨어 인식의 세 원칙에 따라 설계되었으며, FC 레이어 기반 SNN 모델을 시뮬레이터 독립적으로 기술할 수 있게 한다. 단일 JSON 명세로부터 세 시뮬레이터의 실행 가능한 Python 코드를 자동으로 생성하는 파이프라인은 parser → codegen → template의 명확한 모듈 분리를 통해 최적화 로직의 중복을 제거하고 유지보수성을 높였다. 또한 'power_mode'와 'device_type' 선언에 따라 LIF → IF 뉴런 교체, FP16 변환, 타임스텝 축소, 서로게이트 함수 교체를 코드 생성 시점에 자동으로 적용하는 하드웨어 인식 최적화 엔진을 구현하였으며, 6개 실험 구성에 대한 검증을 통해 코드 생성의 구조적 정확성과 최적화 적용의 일치성을 확인하였다. SpikingJelly 구성에서 저전력 모드는 표준 모드 대비 추론 시간 75.0% 단축, 메모리 사용량 73.8% 감소를 달성하였으며, LIF → IF 교체에도 불구하고 스파이크 발화율 차이는 2.8%에 불과하였다.

본 연구는 모델 주도 엔지니어링(MDE) 방법론을 SNN 시뮬레이터 도메인에 적용한 시도로, SNN 연구자가 시뮬레이터 구현 세부 사항에 얽매이지 않고 모델 설계와 실험에 집중할 수 있는 환경을 제공한다는 점에서 실용적 의의를 가진다. 특히 하드웨어 제약을 명세 단계에서 선언하고 코

드 생성 시점에 자동 반영하는 접근법은 기존 post-training 최적화의 일관성 문제를 구조적으로 해소하는 새로운 관점을 제시하며, 동일한 메타모델에서 여러 하드웨어 구성의 코드를 자동으로 생성할 수 있다는 특성은 다양한 배포 환경을 대상으로 하는 SNN 연구에서의 실험 재현성을 높이는 데 기여할 수 있다.

본 연구는 다음과 같은 향후 연구 방향을 제시한다. 가장 우선순위가 높은 과제는 레이어 타입 지원의 확장이다. 메타모델 스키마에 'kernel_size', 'stride', 'padding' 같은 컨볼루션 파라미터를 추가하고 각 시뮬레이터 템플릿에 렌더링 블록을 추가함으로써 'conv2d', 'conv1d' 레이어를 지원할 수 있으며, 시간적 의존성이 강한 태스크에 적합한 순환 SNN(RSNN) 또한 별도의 스키마 확장을 통해 지원 범위를 넓힐 수 있다. 유효성 검사기의 의미론적 강화도 중요한 과제로, 현재 구조적 수준에 머무는 검사기를 시뮬레이터-파라미터 호환성 검사, 'power_mode'와 'device_type' 간의 논리적 일관성 검사, 뉴런 파라미터 범위 검증, 연결 그래프의 유향 비순환 그래프(DAG) 검증 수준으로 끌어올릴 필요가 있다.

하드웨어 지원의 측면에서는 현재 소프트웨어 수준에 그치는 뉴로모픽 최적화를 실제 하드웨어 배포로 확장하는 것이 장기적 목표가 된다. Intel Loihi의 경우 'lava' 프레임워크를 통한 배포가 가능하며, 메타모델에 'lava_template.j2'를 추가하는 방식으로 지원할 수 있다. 최적화 전략의 측면에서도 현재의 코드 구조 수준 최적화에서 나아가, 시냅스 연결의 희소화를 통해 연산량을 줄이는 가지치기(Pruning), 대형 SNN의 스파이크 패턴을 소형 SNN이 모방하도록 학습시키는 지식 증류(Knowledge Distillation) 같은 방향을 고려할 수 있다.

사용성 개선 측면에서는 JSON 직접 편집 방식 대신 React Flow나 Rete.js 같은 노드 기반 에디터를 활용한 웹 기반 시각적 모델 편집기와의 연계를 통해 프레임워크의 접근성을 높이는 것이 가능하다. 마지막으로, 현재 프레임워크의 가장 근본적인 한계인 의미론적 동등성 문제를 해결하기 위해, 동일한 입력 스파이크 패턴에 대해 세 시뮬레이터에서의 출력 스파이크 패턴을 자동으로 비교하고 허용 범위를 초과하는 차이를 경고하는 동등성 검증 도구의 개발을 향후 연구로 제안한다. 이러한 일련의 확장이 이루어진다면, 본 프레임워크는 FC 레이어 중심의 개념 검증 수준을 넘어 SNN 연구의 실질적인 배포 인프라로 자리매김할 것이라 기대한다.

HANSUNG UNIVERSITY

참 고 문 헌

1. 국내문헌

정유선, 김지안, 변혜성. (2023).『SNN(Spike Neural Network)와 CNN(Convolutional neural network)모델 구현을 통한 MNIST 추론 메커니즘 연구』. 한국정보기술학회 2023년도 추계종합학술대회 및 대학생논문 경쟁대회, 997-999.

허정윤, 이현종, 임재한. (2024).『SNN을 이용한 저전력 침입 탐지 시스템』. 한국통신학회논문지, 49(6), 847-861.

홍윤표, 김희탁, 전석훈, 황태호. (2022).『LIF/IF Model을 활용한 SNN/CNN Accumulator의 H/W 구현』. 전자공학회논문지, 59(1), 112-117.

이원모, 성백륜, 고상기. (2025).『스파이킹 신경망의 적대적 강건성을 위한 STDP 기반 학습』. 한국정보과학회 2025 한국컴퓨터종합학술대회 논문집, 2,470-2,472.

2. 국외문헌

Davison, A. P., et al. (2009). “PyNN: a common interface for neuronal network simulators.” Frontiers in Neuroinformatics.

Fang, W., et al. (2023). “SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence.” arXiv

Hazan, H., et al. (2018). “BindsNET: A machine learning-oriented spiking neural networks library in python.” arXiv

Gleeson, P., et al. (2010). “NeuroML: A Language For Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detial.” Plos Computational Biology.

Yavuz, E., et al. (2016). “GeNN: a code generation framework for accelerated brain simulations.” Scientific Reports.

Purtra, R. V. W., & Shafique, M. (2024). “HASNAS: A Hardware-Aware Spiking Neural Architecture Search Framework for Neuromorphic Compute-in-Memory Systems.” arXiv

Qin, S. (2025). “ONNX-Net: Towards Universal Representations and Instant Performance Prediction for Neural Architectures.” arXiv

Rueckauer, B., & Liu, S. (2018). “Conversion of Analog to Spiking Neural Networks Using Sparse Temporal Coding.” IEEE

Di Rocco, J., et al. (2024). “On the use of Large Language Models in Model-Driven Engineering.” arXiv

Alidu, A., et al. (2025). “Prompt2DAG: A Modular Methodology for LLM-Based Data Enrichment Pipeline Generation.” arXiv.

ABSTRACT

Metamodel-based Framework for SNN Simulators

Son, Yu-Hyeon

Major in Computer Engineering

Dept. of Computer Engineering

The Graduate School

Hansung University

The research environment for spiking neural networks(SNNs) is fragmented across multiple mutually incompatible simulators-including snnTorch, SpikingJelly, and BindsNET-requiring researchers to re-implement the same model for each environment. This issue goes beyond mere inconvenience, directly affecting experimental reproducibility and research reliability while reducing overall efficiency. Furthermore, optimization tasks required to bridge the gap between high-performance GPU environments used in the research phase and actual deployment environments such as edge AI processors and neuromorphic chips-including LIF-to-IF neuron substitution, FP16 precision conversion, and timestep reduction-rely entirely on manual effort, making it difficult to guarantee consistency

and prone to introducing subtle bugs. This paper proposes a hardware-aware automatic code generation framework that combines a simulator-independent JSON-based metamodel with the Jinja2 template engine to address these two problems. From a single JSON specification, executable Python code for snnTorch, SpikingJelly, and BindsNET is automatically generated; LIF-to-IF neuron substitution, FP16 conversion, timestep reduction, and surrogate function replacement are automatically applied at code generation time based on the power_mode and device type declarations. The code generation pipeline comprises three modules-validation (parser.py), optimization application(codegen.py), and simulator-specific rendering (Jinja2 templates)-and the clear separation of each module enhances extensibility and maintainability. However, this study is limited to fully connected (FC) layer structures and does not address extensions to convolutional layers. Code generation accuracy and optimization application were verified across six experimental configurations comprising three simulators and two power mode combinations; executability and structural fidelity were confirmed across all six configurations. For the SpikingJelly configuration, the low-power mode achieved a 75.0% reduction in inference time and a 73.8% reduction in memory usage relative to the standard mode; despite the LIF-to_IF substitution, the difference in spike firing rate was only 2.8%, confirming that the impact of low-power optimization on firing patterns is limited.

【keywords】 Spiking Neural Networks, SNN, Metamodel, Jinja2, JSON