# [Logical Node Selection Heuristic for Refinement using Spurious Counterexample]

이름: 채승현
학번: 20180462
연구지도교수: 배경민

---

## 1 연구 목적 (Problem Statement)

DNN (Deep Neural Network)의 실생활 적용이 급격하게 늘어나며, autonomous driving car과 같은 robustness가 요구되는 safety-critical 분야에서도 활발히 활용 중이다. 이로 인해 모든 입력에 대해 관심 있는 특징 및 성질이 성립하는지를 검사하는 검증 과정이 중요시되고 있다. 앞으로도 더욱더 많은 영역에서 사용되리라 전망되는 DNN에 대한 검증은 해결해야 하는 필수적인 과제다. 현재까지 여러 접근과 방법이 제시되었지만, non-linearity를 제공하는 Sigmoid와 ReLU 와 같은 activation functions 등의 존재로 인해 다룰 수 있는 NN (Neural Network)의 종류나 크기가 제한적인 상황이다.

본 연구에서는 (1) DNN 검증에서 활용되는 대표적 접근 방법들인 symbolic interval propagation with node splitting refinement method와 branch and bound framework 및 이를 구현한 Neurify와 BaB를 분석하고 각 접근의 장점만 합치며, 새로운 heuristic 분석이 용이하도록 Neurify를 BaB framework에 integrate하여, BaB style Neurify를 구현하고자 한다. 그리고 (2) node splitting refinement method에서 현재 사용되고 있는 경험적 heuristic을 대체할 model checking에서 활용되는 counterexample based learning을 적용하여 spurious counterexample을 활용하는 logical node selection heuristic을 제시하고자 한다.

## 2 연구 배경 (Motivation and Background)

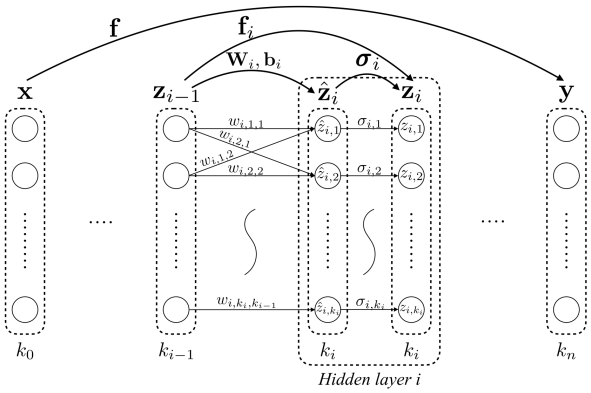


Figure 1: Neural Network Illustration

DNN 검증이란 입력 받은 input과 원하는 output set의 property가 있을 때, input set과 output set의 관계가 성립되는지를 확인하는 과정이다. 즉 f(x) = y (Fig 1)에서 (x, y) relation이 원하는 관계를 가지는지를 확인하는 것이다. 현재 수많은 검증 방법이 제안되었으며, 크게 사용한 analysis 기법을 기준으로 search, optimization, reachability 혹은, 3개의 기법의 혼합 분류로 나눌 수 있다.

그중에서 search + reachability 계열인 Symbolic interval propagation with node splitting refinement가 high-dimension input 상대 DNN 검증에는 state-of-the art이다. 이 방법은 Fig 2.에서 볼 수 있듯이 symbolic linear relaxation으로 ReLU를 abstract 한 후, input interval을 propagate하여 reachable interval을 계산한다. 그리고 linear constraints를 solve하여 counterexample이 존재하는지 확인한다. 이때, abstraction이 coarse하므로 실제

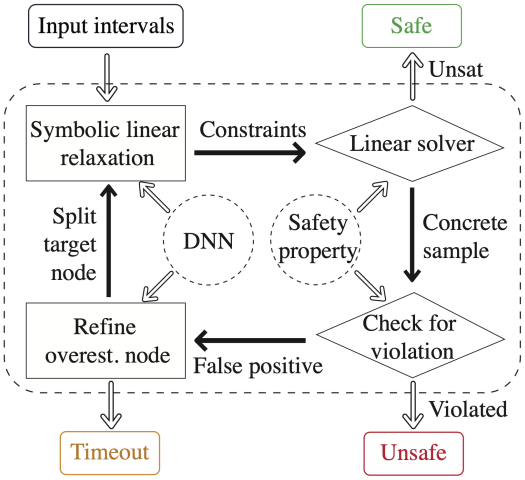


Figure 2: Neurify Workflow

counterexample인지 확인하기 위해 실제 NN에 입력하여 확인하고, false positive인 경우, 즉, spurious counterexample인 경우 다시 node splitting refinement를 하는 iterative process를 가진다.

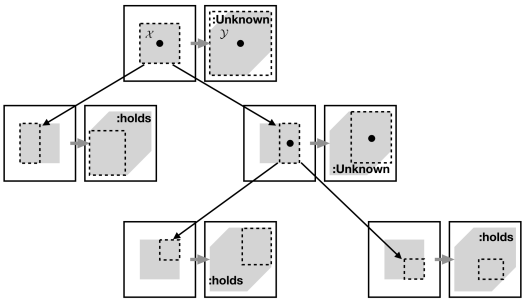


Figure 3: Node Splitting Refinement

Node splitting refinement (Fig 3)란 over-approximation 과정으로 인해 소개되는 over-estimation을 줄이기 위해, 특정 heuristic을 기반으로 node를 선정한 후 interval이나 abstract data structure을 refine하는 과정이다. refinement할 node를 선정하는 node selection heuristic으로는 여러 가지가 있으며 현재까지는 경험적인 heuristic이 전부다. 그중에서도 node interval에 대한 NN output의 gradient를 계산한 뒤 max gradient를 가진 node를 select하는 heuristic이 대표적이며, Neurify에서도 사용된다.

하지만 이러한 경험적 heuristic은, 상식적으로 타당한 방법 같지만, 논리적 근거와 접근이 부족하다. 그뿐만 아니라, spurious counterexample임이 확인된 후, counterexample이 제공하는 정보를 하나도 사용하지 않는다는 단점이 있다. Iterative refinement 단계에서 이전 과정의 counterexamples이 쌓이지만, 활용하지 않는다는 것이다.

$$
\begin{aligned}
\min_{\mathbf{x},\hat{\mathbf{x}}} \quad \hat{x}_n \qquad \text{s.t.} \quad & \mathbf{x}_0 \in \mathcal{C}, \\
& \hat{\mathbf{x}}_{k+1} = W_{k+1}\mathbf{x}_k + \mathbf{b}_{k+1} \quad k \in [\![0, n-1]\!] \\
& \mathbf{x}_k = \sigma_k\left(\hat{\mathbf{x}}_k\right) \qquad\qquad\quad k \in [\![1, n-1]\!]
\end{aligned}
$$

Figure 4: Branch and Bound Problem Formulation

BaB framework에서는 NN의 마지막 layer 뒤로 fully-connected layer를 추가하여, 다루는 검증 문제 자체 (linear contraints의 conjunctions)를 f(x) ≥ 0 꼴의 global optimization 문제 (Fig 4)로 변환한다. 이로 인해 property가 hold 하는지 violated 되는지는 0보다 큰지 작은지로 판단할 수 있게 된다.

**Algorithm 1** Branch and Bound

```
1:  function BAB(net, problem, ϵ)
2:      global_ub ← inf
3:      global_lb ← − inf
4:      probs ← [(global_lb, problem)]
5:      while global_ub − global_lb > ϵ do
6:          (_ , prob) ← pick_out(probs)
7:          [subprob_1, . . . , subprob_s] ← split(prob)
8:          for i = 1 . . . s do
9:              prob_ub ← compute_UB(net, subprob_i)
10:             prob_lb ← compute_LB(net, subprob_i)
11:             if prob_ub < global_ub then
12:                 global_ub ← prob_ub
13:                 prune_problems(probs, global_ub)
14:             end if
15:             if prob_lb < global_ub then
16:                 problems.append((prob_lb, subprob_i))
17:             end if
18:         end for
19:         global_lb ← min{lb | (lb, prob) ∈ probs}
20:     end while
21:     return global_ub
22: end function
```

Figure 5: Branch and Bound Algorithm for DNN Verification

그 후, 특정 heuristic에 따라 input domain을 계속 분할하며, 각 sub-domain의 minimum의 lower/upper bound를 계산한 뒤 subdomain의 minimum의 upper bound가 global upper bound보다 작은 경우 pruning을 진행한다 (Fig 5). global lower/upper bound 간의 차이가 특정 값보다 작아지면 branch 과정을 멈치며 NN를 검증한다. Fig 5. line 6에서 pick_out 함수는 branching할 다음 domain을 선택하는 함수로 여러 heuristic이 사용 가능하며, 이번 연구에서 logical heuristic이 사용될 예정이다.

# 3 연구 방법 (Design and Implementation)

연구 목적 (1), (2)에 따라 연구 방법도 마찬가지로 Part 1, 2를 나눠며 Part 1에서는 기존 접근 분석 및 Part 2에서 새롭게 제시된 heuristic을 테스팅하기 위해 새로운 도구 개발하며, Part 2에서는 spurious counterexample 분석 및 새로운 heuristic을 고안한다.

## 3.1 Part 1. Symbolic interval propagation with node splitting refinement method 와 BaB framework 분석 및 BaB style Neurify 개발

아래 3 단계를 거쳐 진행한다. 도구 내부 코드를 분석하며 실제로 사용해보고 장단점을 파악 및 정리한 후, 두 접근의 장점을 합친 새로운 BaB style Neurify를 개발한다.

1. 대표 도구인 Neurify와 BaB를 활용하여 Acas Xu 데이터셋 NN 검증
2. 도구 알고리즘 및 검증 결과 분석을 통한 장단점 정리
3. 두 접근/방법의 장점들을 합친 BaB style Neurify 개발

```julia
function solve(solver::BaB, problem::Problem)
    (u_approx, u, x_u) = output_bound(solver, problem, :max)
    (l_approx, l, x_l) = output_bound(solver, problem, :min)
    bound = Hyperrectangle(low = [l], high = [u])
    reach = Hyperrectangle(low = [l_approx], high = [u_approx])

    output = problem.output
    if reach ⊆ output
        return ReachabilityResult(:holds, [reach])
    end
    high(bound) > high(output) && return CounterExampleResult(:violated, x_u)
    low(bound)  < low(output)  && return CounterExampleResult(:violated, x_l)
    return ReachabilityResult(:unknown, [reach])
end
```

```julia
function solve(solver::Neurify, problem::Problem)
    isbounded(problem.input) || throw(UnboundedInputError("Neurify can only handle bounded input sets."))

    nnet, output = problem.network, problem.output
    reach_list = []
    domain = init_symbolic_grad(problem.input)
    splits = Set()
    for i in 1:solver.max_iter
        if i > 1
            domain, splits = select!(reach_list, solver.tree_search)
        end

        reach = forward_network(solver, nnet, domain, collect=true)
        # The first entry is the input set
        popfirst!(reach)
        result, max_violation_con = check_inclusion(solver, nnet, last(reach).sym, output)

        if result.status === :violated
            return result
        elseif result.status === :unknown
            subdomains = constraint_refinement(nnet, reach, max_violation_con, splits)
            for domain in subdomains
                push!(reach_list, (init_symbolic_grad(domain), copy(splits)))
            end

        end
        isempty(reach_list) && return CounterExampleResult(:holds)
    end
    return CounterExampleResult(:unknown)
end
```

Figure 6: Julia Code for BaB and Neurify

실제 구현 및 실행/테스팅은 컴퓨터공학과 서버 환경에서 "Algorithms for Verifying Deep Neural Networks"에서 제공된 Neurify와 BaB 알고리즘과 구현된 Julia 코드를 사용한다. 테스트 데이터셋 (Acas Xu NN)은 Marabou Github repo에서 제공된 nnet 형식 파일을 사용하고, 검증하고자 하는 Properties의 경우, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks"에 명시되어 있는 Acas Xu Property 5, 10을 사용한다.

### 3.2 Part 2. Node splitting refinement method에서 활용되는 spurious counterexample을 활용한 logical node selection heuristic 제시

1. NN 검증에서 spurious counterexample의 의미 분석
2. Spurious counterexample이 output 혹은 검증에 미치는 영향 측정 방법 고안
3. Spurious counterexample을 활용한 새로운 heuristic 제시

마찬가지로 3 단계를 거쳐, 활용하고자 하는 spurious counterexample의 의미를 먼저 분석한 후, 이를 활용할 수 있도록 output에 미치는 영향을 측정할 수 있는 방법과 새로운 heuristic을 고안한다.

## 4 연구 결과 및 평가 (Methodology and Evaluation)

### 4.1 Part 1.1 Neurify와 BaB 도구를 통한 Acas Xu 데이터셋 NN 검증

```julia
acas_file = "../mod_networks/ACASXU_experimental_v2a_4_5.nnet"
acas_nnet = read_nnet(acas_file, last_layer_activation = Id());

# ACAS PROPERTY 10
b_lower = [ 0.2689784272,  0.1114084602, -0.4999998960 ,  0.2272727273 ,  0.0      ]
b_upper = [ 0.6798577687,  0.4999998960 , -0.4984083465,  0.5,  0.5      ]

in_hyper  = Hyperrectangle(low = b_lower, high = b_upper)
out_hyper  = Hyperrectangle(low = [-0.1], high = [3.0])

problem_acas1_RR = Problem(acas_nnet, in_hyper, out_hyper)

solver = BaB()
println("$(typeof(solver)) - acas xu property 10")
timed_result = @timed solve(solver, problem_acas1_RR)
```

Figure 7: Julia code for verifying Acas Xu Property 10 with BaB

Acas Xu Property 5와 10을 Julia로 구현된 Neurify와 BaB가 입력 받을 수 있도록, Neurify의 경우 Input: Hyperrectangle, Output: HPolytope, BaB의 경우 Input: Hyperrectangle Output: Hyperrectangle 형식 (Fig 7)으로 표현했다.

```julia
# output5 <= output 1
outputSet1 = HPolytope([HalfSpace([-1.0, 0.0, 0.0, 0.0, 1.0], 0.0)])
problem_polytope_polytope_acas1 = Problem(acas_nnet, in_hyper, outputSet1);

# output5 <= output 2
outputSet2 = HPolytope([HalfSpace([0.0, -1.0, 0.0, 0.0, 1.0], 0.0)])
problem_polytope_polytope_acas2 = Problem(acas_nnet, in_hyper, outputSet2);

# output5 <= output 3
outputSet3 = HPolytope([HalfSpace([0.0, 0.0, -1.0, 0.0, 1.0], 0.0)])
problem_polytope_polytope_acas3 = Problem(acas_nnet, in_hyper, outputSet3);

# output5 <= output 4
outputSet4 = HPolytope([HalfSpace([0.0, 0.0, 0.0, -1.0, 1.0], 0.0)])
problem_polytope_polytope_acas4 = Problem(acas_nnet, in_hyper, outputSet4);
```

Figure 8: Julia code for verifying Acas Xu Property 10 with Neurify

이 과정에서 linear constraints의 conjunction을 한 번에 표현할 수 없는 제약 조건으로 인해, clause 하나당 verification 문제를 생성하여 결과를 확인했다. Property 10의 경우 검증하고자 하는 성질이 "the score for COC is minimal"이므로 해당되는 output node 1과 나머지 output node 2, 3, 4를 각각 비교하는 문제 4개를 생성했다 (Fig 8).

```
Neurify - acas
 - Time: 21.472895226 s
 - Output:
CounterExampleResult(:violated, [0.21466922000000005, 0.11140845999999999, -0.49840835, 0.3920202, 0.4])
```

Figure 9: Verification Result using Julia Tool

| | Neurify | BaB |
|---|---|---|
| Property 5 | 140.3601s | TIMEOUT |
| Property 10 | 92.593s | TIMEOUT |

Table 1: Neurify, BaB performance verifying Acas Xu property 5 and 10

그 후, Neurify와 BaB 도구를 사용하여 검증을 진행했으며, 검증 결과는 Figure 9 형식으로 출력되었다. 검증 과정을 property 5, 10에 대해 진행하고 총 소유 시간을 비교한 결과, Table 1과 같이 결과가 나왔다. (TIMEOUT 기준: 12h) 결과 분석은 Part 1.2에 나와 있다.

## 4.2 Part 1.2 Neurify, BaB 알고리즘 및 Part 1.1 검증 결과 분석을 통한 장단점 정리

Neurify의 경우, 경험적 heuristic (output에 영향을 미치는 정도를 gradient로 계산한 후, maximum 값을 가지고 있는 node 선정)을 사용하며, interval propagation 및 relu node abstraction으로 인해 high-dimension input 상대 state-of-the art이다. 그러나 연구 배경에서 나왔듯이 검증 과정에서 spurious counterexample을 사용하지 않는다는 단점이 있다.

| | Neurify | BaB |
|---|---|---|
| Pros | 경험적 heuristic 사용<br>high-dimension input 상대 state-of-the art | Global optimization problem으로 formulate<br>node selection heuristic에서 splitting decision이 output에 영향을 미치는 정도 측정하는데 용이 |
| Cons | Abstraction으로 인해 발생하는<br>spurious counterexample 미활용 | Naive한 heuristic 사용 |

Table 2: Pros and Cons of Neurify and BaB

BaB의 경우, naive heuristic (현재 구현상, domain의 크기가 가장 큰 domain을 선택)을 사용하여 시간이 오래 걸리는 반면, global optimization 문제로 formulate하며, 검증하고자 하는 성질을 output node 한 개에 해당하게 하

므로, node selection heuristic에서 node selection 후 splitting을 했을 때, selection 자체가 output에 어떻게 영향을 미치는지를 측정하는데 용이하다.

따라서 위와 같은 분석 결과를 통해 두 도구의 장점을 가지고 있는, BaB framework에서 input domain 대신 전체 unsplit ReLU nodes를 상대로 branch and bound를 하며 modular 하게 사용할 heuristic을 정할 수 있는 BaB style Neurify를 구현하고자 했다. Part 2.2, 3 에서 제시되는 spurious counterexample을 활용한 heuristic을 확인하고자 했다.

## 4.3 Part 2.1 Neural network verification에서 spurious counterexample이란

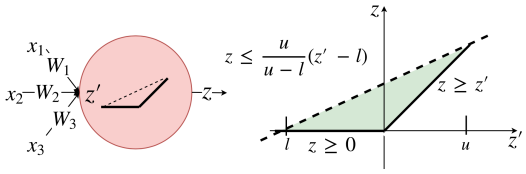


Figure 10: Linear relaxation of a ReLU node

검증하고자 하는 concrete NN를 함수 f 라고 하면, 현재 검증 방법은 f 에 approximate하는, 모든 network node의 value assignment가 만족하는 linear constraints system을 생성하는 것이다. 하지만 ReLU와 Sigmoid 등의 activation function은 linear하지 않기 때문에 Figure 10 과 같이 linear approximation을 통해 linear constraints로 표현할 수 있도록 한다.

위 과정과 같은 abstraction으로 인해 생성된 abstract NN은 concrete NN와 다르게 행동하게 되며, abstract NN에서 성립하는 property는 concrete NN에서도 성립하지만, 성립하지 않는 counterexample이 abstract NN에서 확인되는 경우, false positive일 수 있다. 즉, non-linear activation function의 linear approximation을 위한 abstraction에서의 coarseness로 인해 발생하는 false positive counterexample이 spurious counterexample이며, concrete NN와 abstract NN의 행동 차이의 결과물이다.

## 4.4 Part 2.2,3 Spurious counterexample이 미치는 영향 측정 방법 고안 및 counterexample을 활용한 logical heuristic 제시

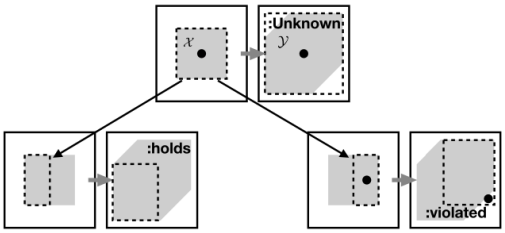


Figure 11: Node splitting process/refinement

고안한 영향력 측정 방법으로는 2가지가 있으며, BaB framework에 integrate 함으로써 확인하고자 하는 성질을 한 개의 output node로 표현할 수 있다는 점을 활용했다.

1. Split 후 subdomain의 lower bound와 0 사이 거리 측정
2. Split 후, original domain과 subdomain의 lower bound 차이

그리고 위 측정 방법을 사용한 새로운 logical heuristic으로 양쪽 모두 spurious counterexample이 존재하는 node들을 전부 확인한 후, 가장 크게 영향을 미치는 node를 선택하는 selection heuristic이다. 이 heuristic 외, 양쪽 모두 spurious counterexample이 존재하는 node들이 많을 때, 많지 않을 때, 한쪽에만 spurious counterexample이 확인되는 경우 등에 대한 logical heuristic도 조사 중이다.

### 4.5 Part 1.3 BaB style Neurify 개발

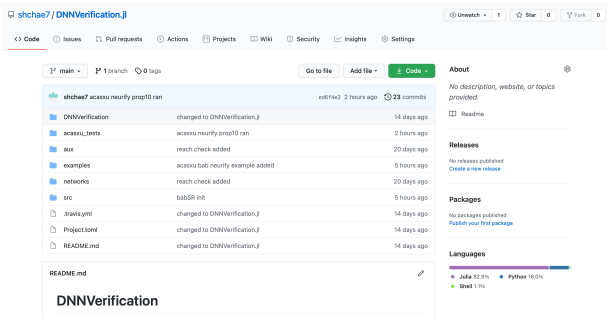


Figure 12: Github repo for BaB style Neurify

현재 Julia로 BaB style Neurify를 구현 중이며, 최대한 abstract하여 모든 component를 modular하게 사용할 수 있도록 제작하는 과정에서 시간이 예상보다 오래 걸렸다. 개발이 끝나자마자 Part 2.3에서 고안한 heuristic의 성능을 확인하고 결과를 분석할 예정이다.

## 5 토론 및 전망 (Discussion and Future Work)

추가로 spurious counterexample을 단순히 활용하는 heuristic에서 끝나지 않고, spurious counterexample에 관여하는 node 탐색 및 측정 방법도 현재 조사 중이며 abstraction의 종류에 따라, concrete counterexample값을 계산한 후, approximation의 면적을 계산하는 방법과 같은 접근을 본 연구에 적용할 예정이다.

본 연구에서는 Abstraction 과정에서 필히 생길 수밖에 없는 현재 효과적으로 활용되고 있지 않았던 spurious counterexample을 활용하는 heuristic을 제시하기 시작했다. 그리고 여러 heuristic을 modular하게 교체할 수 있는 framework를 개발하는 과정으로, 앞으로 새로 고안되는 logical heuristic을 손쉽게 확인할 수 있는 플랫폼을 제공할 수 있을 것이라고 기대된다. 그뿐만 아니라 앞으로 새로운 spurious counterexample 영향 측정 방법과 selection heuristic을 제시함으로써, 아직 기존 경험적 heuristic보다 성능이 낮다고 하더라도, 점점 더 성능이 좋아질 것이라고 예상된다. 궁극적으로 다룰 수 있는 NN의 크기 증가와 architecture/종류 다양성 향상을 이룰 수 있을 것이라고 예상된다.

## 참고문헌


[1] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A unified view of piecewise linear neural network verification, 2018.

[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[3] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks, 2017.

[4] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. *Lecture Notes in Computer Science*, page 43–65, 2020.

[5] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks, 2020.

[6] Jingyue Lu and M. Pawan Kumar. Neural network branching for neural network verification, 2019.

[7] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks, 2018.