Concolic Testing on Inference Decision Logic of Neural Network Models

Chih-Duo Hong[1], Wang Yu[1], Yao-Chen Chang[2], and Fang Yu[1]

[1] National Chengchi University, Taiwan
[2] Academia Sinica, Taiwan

Abstract. Concolic testing for deep neural networks alternates concrete execution with constraint solving to search for inputs that flip decisions. We present an influence-guided concolic tester for Transformer classifiers that ranks path predicates by SHAP-based estimates of their impact on the model output. To enable SMT solving on modern architectures, we prototype a solver-compatible, pure-Python semantics for multi-head self-attention and introduce practical scheduling heuristics that temper constraint growth on deeper models. In a white-box study on compact Transformers under small L_0 budgets, influence guidance finds label-flip inputs more efficiently than a FIFO baseline and maintains steady progress on deeper networks. Aggregating successful attack instances with a SHAP-based critical decision path analysis reveals recurring, compact decision logic shared across attacks. These observations suggest that (i) influence signals provide a useful search bias for symbolic exploration, and (ii) solver-friendly attention semantics paired with lightweight scheduling make concolic testing feasible for contemporary Transformer models, with potential utility for debugging and model auditing.

1 Introduction

In the past decades, deep neural networks have achieved remarkable success across various domains, including computer vision [1], speech recognition [2], and game playing [3], drawing significant attention from both academia and industry [4]. By leveraging neural networks, complex systems that are difficult for humans to design can be created, often outperforming traditional methods. However, the increasing use of neural networks in safety-critical and socially sensitive areas raises significant concerns about their verification and security. Neural network models, albeit trained on finite datasets, are expected to generalize to new inputs. However, they can behave unexpectedly when faced with adversarially crafted perturbations [5–7]. This unpredictability can lead to unsafe systems. Manual inspection of large models is infeasible due to their complexity, necessitating systematic and automated verification methods to ensure their reliability.

Significant efforts have been devoted to developing robust machine-learning models that can withstand environmental noise and adversarial attacks [8]. Specifically, various program analysis techniques, from formal verification [9–12]

to automated testing [13–16], have been proposed to enhance the dependability of AI systems. Although formal verification provides correctness guarantees, its scalability on practical networks is often limited. As a complementary path, we adopt *concolic testing*, where the tester alternates between concrete execution and SMT-based solving of path predicates to synthesize inputs [17, 18]. The objective is not certified worst-case robustness but the efficient discovery of concrete counterexamples. A central design choice in such testers is how to prioritize which symbolic branches to solve next. Prior neural-network testing frameworks often rely on coverage metrics, such as neuron coverage and contribution coverage [19, 20]. However, these metrics treat neurons equally without considering their relative importance in the model's decision-making process, which often leads to inefficient exploration of model behaviors and missed critical vulnerabilities [21, 22].

To address this prioritization challenge within our concolic testing loop, we leverage SHAP (SHapley Additive exPlanations) values [23–25] to estimate the influence of neurons on the current decision and to rank the corresponding branch predicates. This influence-guided strategy focuses solver calls on high-leverage parts of the model, prioritizing predicates expected to affect the output while deferring less informative branches. We further adopt the notion of abstract critical decision path [26], which summarizes recurring decision logic behind adversarial inputs and highlights neurons that frequently contribute to misclassification.

The emergence of Transformers [27] has posed new challenges for neural network testing [28–31]. Their unique attention mechanisms make them less amenable to traditional testing methods designed for simpler architectures like CNNs and RNNs. Our approach extends concolic testing to generate adversarial examples specifically for Transformer models. Prior constraint-based Transformer verification methods rely on convex relaxations/abstract interpretation or MILP encodings [28, 30]. To the best of our knowledge, our work offers the first (SMT-backed) concolic testing method that supports standard multi-head self-attention with softmax, thereby making constraint-based testing viable for Transformer architectures.

We conduct a preliminary empirical study of our influence-guided concolic testing on compact Transformer classifiers. For each seed input, we search for label flips by solving path constraints under bounded perturbations (e.g., 1–2 pixel edits of the input image) and a fixed per-seed time bound. Relative to a FIFO baseline, prioritizing branch predicates by SHAP-based influence improves search quality. On a single-layer Transformer we obtain 67 successful one-pixel attacks and 6 successful two-pixel attacks. On a deeper two-layer Transformer, two simple scheduling heuristics—*layer-prioritized* exploration and *time-capped* constraint building—help maintain steady progress under tight budgets. Finally, aggregating successful cases with our SHAP-based abstract critical decision path (ACDP) analysis reveals shared decision logic: with $\alpha = 20\%$ and $\beta = 0.5$, 245 of 4,430 neurons are marked critical for more than half of the adversarial inputs.

Contributions. We list our current contributions below.

1. **Influence-guided concolic testing.** We integrate SHAP-based influence into the concolic loop to rank path constraints by estimated impact on model decisions, steering SMT solving toward higher-leverage branches.
2. **Decision-logic synthesis.** We instantiate a SHAP-based ACDP analysis to summarize common failure mechanisms across adversarial inputs, surfacing compact sets of neurons that may guide debugging and repair; the evidence is preliminary but suggests recurring decision logic across successful attacks.
3. **Prototype Transformer support with practical scheduling.** We provide a solver-compatible semantics for multi-head self-attention (including softmax) within PyCT and introduce simple scheduling heuristics that trade off influence against constraint complexity on deeper models.

This paper is organized as follows. Section 2 surveys related work on constraint-based verification and testing of neural networks. Section 3 presents our influence-guided concolic testing framework, including SHAP-based predicate prioritization, a solver-compatible pure-Python semantics for multi-head self-attention, and the SHAP-based abstract critical decision path analysis. Section 4 reports preliminary experiments on compact Transformer classifiers, including ablations of two scheduling heuristics and a brief comparison with DeepConcolic. Section 5 concludes with limitations and directions for future work. The appendix provides the step-by-step attention semantics used by our prototype.

2 Related Work

Recent research demonstrates growing interest in employing constraint solvers to verify and test neural networks [32–36], where these solvers are applied to rigorously analyze complex decision boundaries and validate critical input-output relationships. Neural network verification through constraint solvers entails encoding network operations and constraints into logical formulations for solver analysis [8,37]. Pioneering constraint-solving frameworks such as Reluplex [9] and Marabou [12] have been pivotal in verifying feedforward networks with ReLU activations [38]. These tools formally validate a network model by identifying counterexamples to a specification or confirming compliance. To address the scalability challenges inherent in constraint-based approaches, various enhancements have been proposed, including symbolic reasoning [39], quantization [40, 41], and abstract interpretation [13, 15].

In addition to formal verification, constraint solvers have also been adopted to generate test cases systematically. Sun et al. [42] pioneered using a constraint solver to maximize neuron coverage in test case generation. This concept was implemented in DeepCover [43] and DeepConcolic [44], which employed concolic execution to balance between test coverage and efficiency. Our work distinguishes itself from these approaches: While DeepCover and DeepConcolic focus exclusively on feedforward networks with ReLU activation, our tool extends concolic testing capabilities to sigmoid and tanh activations. Furthermore, existing tools generate test cases that maximize coverage, which might be ineffective for discovering adversarial vulnerabilities [21]. Specifically, DeepConcolic formulates

neuron coverage as an optimization problem and generates constraints over input variables based on the neurons to activate. By contrast, PyCT computes path constraints induced by the attacked input variables and explores only the network behaviors affected by these variables. This approach arguably aligns more closely with the logical flow of input-output specifications, making our tool particularly suitable for applications requiring contract-based [45] and feature-specific adversarial testing [46].

Our testing framework exploits Shapley values (via DeepSHAP [47]) to attack the most influential input variable and explore the most influential branches. In comparison, gradient-based attacks [29,31,48–51] target features with large gradient values, assuming they have high sensitivity to prediction changes. Gradient-based methods are computationally efficient and applicable across diverse network architectures. However, they often lack interpretability and can be susceptible to gradient noise and local optima [48]. Our integration with DeepSHAP selects features directly based on their impacts on model behaviors, arguably offering more explainability and robustness. This capability positions our framework as a complementary alternative to traditional methods.

Besides gradient-based testing, coverage-guided fuzzing has also been widely utilized to uncover vulnerabilities in neural networks [52–54]. However, conventional neuron coverage does not account for the varying significance of neurons in influencing model outputs, making them ineffective in searching for adversarial examples [21]. Recent tools such as CriticalFuzz [55] incorporate the information of "critical neurons" in the selection and mutation of seed inputs, thereby achieving higher error detection rates compared to traditional fuzzing techniques. Specifically, neuron path coverage [26] measures the relevancy of test cases based on decision logic, motivating our influence-guided concolic testing approach. Its concept of critical decision paths is employed in our work to identify the most essential neurons, where we utilize SHAP values to prioritize branch constraints based on their impact on model decisions.

Prior concolic testing frameworks for neural networks either target FNN/CNN architectures [17,19,36] or present general-purpose engines without Transformer-specific reasoning [18, 56–59]. To the best of our knowledge, no existing work integrates concolic/symbolic path-constraint solving with the internals of Transformer attention blocks [27] to automatically synthesize adversarial inputs; our work is, therefore, the first concolic testing approach explicitly designed for and demonstrated on modern Transformer architectures.

3 Methodology

In this section, we present our methodology for adversarial example synthesis using concolic testing, along with the algorithms that drive our attack.

3.1 Object-Oriented Concolic Testing

We briefly revisit the concolic testing algorithm that serves as the foundation of PyCT [17,18,36]. This algorithm is built upon an architecture that elegantly

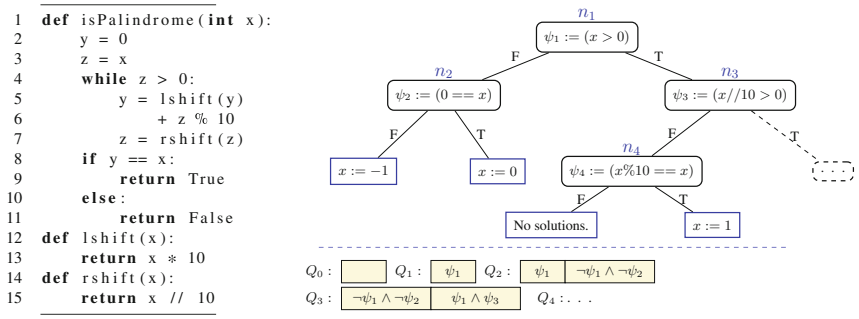

Fig. 1: A concolic testing example from Chen et al. [18]. Left: A program for checking if x is a palindrome. Right: The tree T and queue Q of the concolic testing process on the program.

integrates symbolic execution with Python's dynamic and object-oriented model [59]. The core idea behind the algorithm is to substitute basic Python data types (such as integers and strings) with concolic objects, which maintain both concrete and symbolic representations. Each concolic object includes two components: a concrete value and a symbolic expression representing the value in terms of input variables. For instance, a concolic integer c_a might be represented as $c_a = (2, x + 1)$, where 2 is the concrete value of the integer variable a, x is an input variable, and $x+1$ is an expression depicting the relationship between the variable a the and the input variable x, namely, $a = x + 1$.

The algorithm operates by overriding the member functions of Python's standard data types with concolic versions that support both symbolic and concrete computations. Following the previous example, an overridden member function $c_a.f()$ of the concolic integer c_x would update the concrete value 2 of c_a in the same way that the original function $a.f()$ does, and update the symbolic expression $x + 1$ to a new expression that captures the semantics of invoking f on $a = x + 1$. This design allows the concolic tester to handle Python's rich set of operations while maintaining symbolic representations wherever possible.

During testing, the algorithm utilizes concolic objects to track and update constraints in program execution. For each path executed, the algorithm maintains a path condition tree T and a queue Q of unexplored branches (Figure 1). The tree T records all explored paths by maintaining nodes with constraints encountered during execution. Simultaneously, Q stores formulas representing unexplored branches, which are derived by negating conditions in the current path. For every decision point in the tested code (such as an `if` statement), the algorithm creates a symbolic condition. If the current path takes the "false" branch, the condition for the "true" branch is negated and added to the queue Q. This queue holds unexplored branches that are yet to be tested. The strength of this approach lies in its ability to refine both T and Q iteratively. Each iteration draws a formula from Q, solves it using an SMT solver, and generates new in-

puts that guide the subsequent program execution down previously unexplored paths. By continuously updating T with new path constraints and expanding Q with new branch conditions, the algorithm incrementally generates new inputs to cover more code paths and detect potential program faults.

The order in which constraints are picked from the queue Q is crucial for this algorithm's performance. Although simple strategies like first-in-first-out can be adopted, a more intelligent approach that prioritizes more relevant constraints can lead to faster convergence and improved efficiency. As a result, selecting a suitable order for processing Q remains an essential issue in optimizing the search process.

3.2 Influence-Guided Concolic Testing

To target the most critical decisions in altering the model outcome, it is necessary to establish a computable ordering over the branch conditions. To this end, we assign to every branch condition an influence value derived from the SHAP value of the associated neurons. The core idea of the influence calculation is as follows: any branch condition, such as an `if` statement, must be evaluated during the computation of the values for a set of neurons. In other words, for any branch condition b, there should be a list of associated neurons assocNeurons(b) for which the branch condition b is evaluated.

Moreover, for a given seed input x, every neuron n in layer l, we should be able to compute $\texttt{shap}(n, x)$ by taking the SHAP values of n at x within the submodel spanning from layer l to the output layer, averaged by all the output neurons. With both assocNeurons and `shap`, each branch condition can be associated with a SHAP-based influence, defined as:

$$\texttt{influ}(b \mid x) \;:=\; \text{avg}(\{\; \texttt{shap}(n, x) \mid n \in \text{assocNeurons}(b) \;\}) \tag{1}$$

The function $\texttt{influ}(b \mid x)$ calculates for branch b the average SHAP values of its associated neurons on the output neurons at seed input x, indicating the general influence of b on the output at x.

Having established the ordering on the branch conditions, we propose a strategy to process the queue Q for PyCT, as depicted in Algorithm 1. Below, we explain the three most important operations in our SHAP-based path exploration.

Registering associated neurons during forwarding computations. In the forward computation within our translated models, the associated neurons are registered to the module to reflect the neurons targeted by the current computation. For each instruction in a forwarding function, the associated neurons are determined as follows: if the instruction involves the assignment of an output neuron of the layer, that output neuron is assigned as its only associated neuron; if an non-output variable is assigned, the associated neurons are the output neurons affected by this variable; otherwise, the associated neurons comprise the entire set of output neurons of the layer.

Algorithm 1 Influence-based concolic testing

```
1: Input
2:    Python function f : {arg_name : float} → any
3:    seed input x_0 : {arg_name : float | ConcolicFloat}
4: Output
5:    a map x' : {arg_name : float} such that x' ≠ x_0 and f(x') ≠ f(x_0)
6: Procedure
7:    y' ← f(x_0)
8:    x' ← x_0
9:    branches_to_explore ← new PriorityQueue()
10:   Do
11:      (y', bypassed_branches) ← concolic_execution(f, x')
12:      If (x' ≠ x_0 and y' ≠ f(x_0))
13:         Return x'
14:      Foreach branch in bypassed_branches
15:         branches_to_explore.push(branch.predicate, influ(branch | x'))
16:      While branches_to_explore is not empty
17:         predicate ← branches_to_explore.popMax()
18:         Match solver.check(predicate)
19:            Pattern UNSAT
20:               Continue
21:            Pattern SAT(solution)
22:               x' ← solution
23:               Break
24:   While branches_to_explore is not empty
```

Calculating the influence of all neurons. PyCT computes the influence of a neuron based on SHAP values. Originally, the SHAP value is a real-valued function, $\texttt{shap}(i, o, x \mid M, X)$, assuming a model M and a background dataset X for reference,[3] defined for an input neuron i of M, an output neuron o of M, and an input data point under test x. Based on this definition, we construct a real-valued function $\texttt{shap}(n, x \mid M, X)$ that computes the influence of a general non-output neuron n at layer l for the output at x, given by:

$$\texttt{shap}(n, x \mid M, X) := \\ \text{avg}(\{\ \|\texttt{shap}(n, o, M_{\leq l}(x) \mid M_{>l}, M_{\leq l}(X))\| \mid o \in M.\text{output}\ \})$$

Here, $M_{\leq l}$ denotes the submodel of M up to layer l and is used to transform the input x and background dataset X, whereas $M_{>l}$ is the submodel of M after layer l that includes neuron n as one of its input neurons. This ensures that the calculation of the SHAP value is well-defined and consistent. In the actual

[3] Influence ranks depend on the background dataset used to compute SHAP values. In our experiments, we sample a fixed background set from the training distribution and compute SHAP once per model prior to search (Algorithm 2). In practice, using different background sets may change the order of some predicates, but it does not invalidate the search loop.

Algorithm 2 Inner-SHAP influence calculation

Input
  model M
  background dataset X
  seed input x
Output
  a map $influences$ of type {(layer_number, indices) : float}
Procedure
  $influences \leftarrow$ new Map()
  $M_{>l} \leftarrow M$
  $X_l \leftarrow X$
  $x_l \leftarrow x$
  for $l = 0, \ldots, \|M.\text{layers}\| - 1$:
    for $n \in M.\text{layers}[l]$:
      $i \leftarrow \text{avg}(\{\|\texttt{shap}(n, o, x_l \mid M_{>l}, X_l)\| \mid o \in M.\text{output}\})$
      $influences.\text{set}((l, n), i)$
    if $M.\text{layers}[l+1] == M.\text{output}$:
      break
    $x_l \leftarrow \text{apply\_first\_layer}(M_{>l}, x_l)$
    $X_l \leftarrow \text{apply\_first\_layer}(M_{>l}, X_l)$
    $M_{>l} \leftarrow \text{remove\_first\_layer}(M_{>l})$
  return $influences$

implementation, `shap` is computed for all neurons in advance of all iterations as described in Algorithm 2.

Pushing branch conditions into the priority queue by SHAP-based influence. As shown in Algorithm 1, the goal of Line 15 is to execute the target model f at a given input x' (consisting of possibly concolic values), while recording the predicate expressions of bypassed branches, which are encountered but not entered. These predicate expressions are then added to the priority queue. In each iteration, constraints are continually popped from the priority queue until a satisfiable constraint is found. The solution of this constraint is then used as the input for the next iteration. If none of the constraints are satisfiable, the program terminates.

Queue invariants and termination. The priority queue in Algorithm 1 stores negations of bypassed predicates keyed by influence. Each iteration either (i) discovers a new concrete path (possibly a flip) or (ii) removes at least one predicate as UNSAT. With finite time/memory budgets, the loop terminates when the queue drains or when the global budget expires. In the absence of budgets and with a fair solver, the process is *semi-complete*: every satisfiable predicate in the reachable region will eventually be attempted; however, we make no claim of full path coverage due to path explosion.

Soundness boundary and validation oracle. Our engine proposes inputs by solving symbolic path predicates, but always validates them by re-executing the

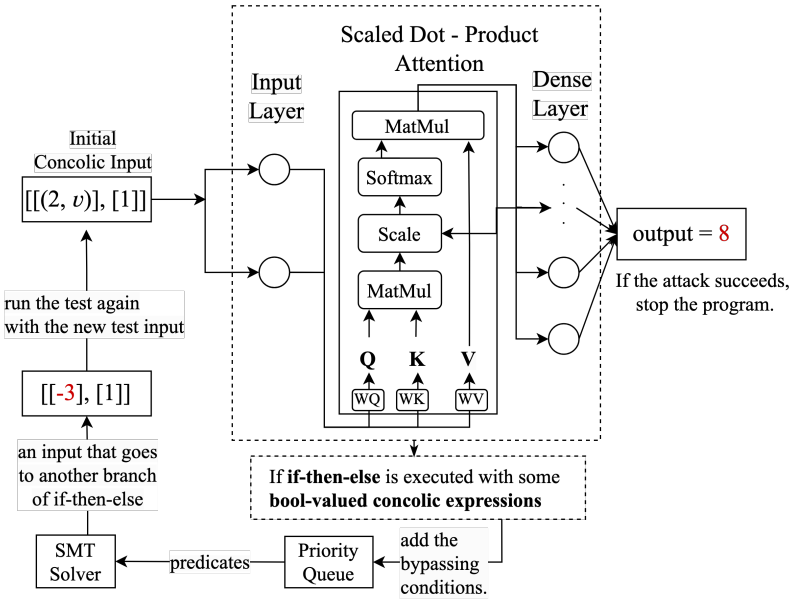

Fig. 2: A running example on a single-layer transformer

original model concretely. Hence, any reported label flip corresponds to an actual behavioral change of the given model under the chosen implementation. Indeed, if the testing loop returns an input x' with $f(x') \neq f(x_0)$ (Algorithm 1, Line 13), then this inequality holds for a concrete execution of the original model f. Therefore, a reported counterexample cannot be spurious.

3.3 Testing the Transformer Architecture

We propose to employ the PyCT tester to scrutinize Transformer models [27] based on SMT reasoning. However, PyCT currently only supports basic Python syntax and data types [18], and thus cannot handle libraries such as NumPy and Tensor used in modern Transducer implementations. To overcome this, we realized the Transformer model using basic Python syntax, eliminating dependencies on external libraries. The implementation involves rewriting the Keras Transformer model, particularly the multi-head attention layer, with fundamental Python constructs.

In the rest of this section, we demonstrate how PyCT computes an adversarial input for a Transformer model implemented in pure Python (Figure 2). Suppose that the seed input is $[[2], [1]]$, which leads to a model prediction of class 10. We set the initial concolic value as $[[(2, v)], [1]]$ and feed it into the tested Transformer. During the model computation, if an if-then-else statement is executed with boolean-valued concolic expressions, we add the bypassing conditions

Algorithm 3 Function for transforming and splitting

function tas(*vectors*, *weights*, *bias*):
  for $i = 0, \ldots,$ num_heads $- 1$
    for $j = 0, \ldots,$ key_dim_per_heads $- 1$
      $outputs[i][j] \leftarrow \sum_{k=0}^{\text{model\_dim}}(weights[k][i][j] \times vectors[k]) + bias[i][j]$
  return *outputs*

to the priority queue. These conditions are then prioritized based on SHAP values and popped as predicates for the SMT solver to solve. If an input directs the execution to an alternate branch of the if-then-else statement, PyCT re-runs the test with the new input. This process repeats until the output label changes.

Transforming and splitting of input sequence. In a single forward pass, the input sequence is first linearly transformed separately by weight matrices W_Q, W_K, W_V and bias matrices B_Q, B_K, B_V, resulting in attention matrices Q, K, V. The operational details are outlined in Algorithm 3. To compute Q, K, V, PyCT first needs to compute $\texttt{tas}(input, W_Q, B_Q)$, $\texttt{tas}(input, W_K, B_K)$, and $\texttt{tas}(input, W_V, B_V)$. Specifically, suppose that

input $= [[\![2, v]\!], [1]]$	num_heads $= 1$	key_dim_per_heads $= 2$
seq_len $= 2$	model_dim $= 1$	$W_Q = [[[1, 1]]]$
$W_K = [[[2, 1]]]$	$W_V = [[[1, 2]]]$	$W_O = [[[1], [1]]]$
$B_Q = [[1, 1]]$	$B_K = [[2, 1]]$	$B_V = [[1, 2]]$
$B_O = [1,]$		

We use $[\![2, v]\!]$ to denote a concolic variable such that 2 is the concrete value and v is the symbolic variable. Let w_{000} represent WQ[0][0][0], w_{001} represent WQ[0][0][1], b_{00} represent BQ[0][0], and b_{01} represent BQ[0][1]. For attention Q, the concolic computation by Algorithm 3 gives

$$\begin{aligned}
&[[w_{000} \times [\![2, v]\!] + b_{00},\ w_{001} \times [\![2, v]\!] + b_{01}],\ [w_{000} \times 1 + b_{00},\ w_{001} \times 1 + b_{01}]] \\
&\quad = [[1 \times [\![2, v]\!] + 1, 1 \times [\![2, v]\!] + 1],\ [1 \times 1 + 1, 1 \times 1 + 1]] \\
&\quad = [[[\![2, v]\!] + 1, [\![2, v]\!] + 1],\ [2, 2]]
\end{aligned}$$

Thus, the resulting concolic value of attention Q is $[[[\![2, v]\!] + 1, [\![2, v]\!] + 1], [2, 2]]$. Attentions K and V are computed similarly. After this step, the concolic values of attentions Q, K, and V are as follows:

$$\begin{aligned}
\text{Concolic matrix of } Q\text{: } &[[[\![2, v]\!] + 1,\ [\![2, v]\!] + 1],\ [2, 2]] \\
\text{Concolic matrix of } K\text{: } &[[2 * [\![2, v]\!] + 2,\ [\![2, v]\!] + 1],\ [4, 2]] \\
\text{Concolic matrix of } V\text{: } &[[[\![2, v]\!] + 1,\ 2 * [\![2, v]\!] + 2],\ [2, 4]]
\end{aligned}$$

Algorithm 4 Function for dot-product attention

1: **function** `dpa`(Q, K, V):
2: $attention\_scores \leftarrow$ matrix_multiply(Q, K^T)
3: $attention\_scores \leftarrow$ [
4: [$score/\sqrt{key\_dim\_per\_heads}$) for $score$ in $attention\_score$]
5: for $attention\_score$ in $attention\_scores$]
6: $attention \leftarrow$ matrix_multiply(`softmax`($attention\_scores$), V)
7: **return** $attention$

Computing the dot-product attention. In this step, we compute the dot-product attention based on the attention matrices Q, K, and V. Leveraging the query, key, and value matrices derived from the input sequence, we meticulously compute attention scores for each attention head. This computation is crucial in identifying salient features within the input sequence, facilitating effective information extraction. The operational details can be referred to in Algorithm 4. Here, we calculate `dpa`(Q, K, V). Firstly, K undergoes a transpose operation. Then, Q and K_T are subjected to matrix multiplication (Algorithm 4, Line 3). The resulting $Q \times K_T$ operation yields:

$$[[Q_{00} \times K_{T00} + Q_{01} \times K_{T10}, \quad Q_{00} \times K_{T01} + Q_{01} \times K_{T11}],$$
$$[Q_{10} \times K_{T00} + Q_{11} \times K_{T10}, \quad Q_{10} \times K_{T01} + Q_{11} \times K_{T11}]]$$

After the calculation, the matrix becomes

$$[[3 * [\![2, v]\!]^2 + 6 * [\![2, v]\!] + 3, \quad 6 * [\![2, v]\!] + 6], \ [6 * [\![2, v]\!] + 6, 12]]$$

Each value in the matrix is then divided by the square root of the dimension of the key, which in this case is 2. (Algorithm 4, Line 4). Upon applying softmax to the attention mechanism as illustrated in Algorithm 4, we will invoke a standard softmax implementation, as depicted in Algorithm 5. Subsequently, prior to exponentiation, each value is subtracted from its corresponding maximum value. This procedure aims to maintain numerical stability and mitigate issues related to overflow and underflow (cf. [4]).

In our example, when the `softmax` function is invoked, $x_{\max} = [(3 * [\![2, v]\!]^2 + 6 * [\![2, v]\!] + 3)/\sqrt{2}, (6 * [\![2, v]\!] + 6)/\sqrt{2}]$. Note that new constraints may be generated here: Recall that in PyCT, whenever a boolean expression with concolic values is evaluated, it triggers a branch listener that adds the negation of the current constraint to the constraint queue. In this case, when the if-statement in the `max` function is executed, it holds that $x[0][1] < x[0][0]$ since $(18/\sqrt{2}) < (27/\sqrt{2})$. PyCT thus pushes the first constraint φ_1: (x[0][1] < x[0][0]) into the priority queue. And in the next row, $x[1][1] < x[1][0]$, i.e., $(12/\sqrt{2}) < (18/\sqrt{2})$, PyCT adds the second constraint φ_2: (x[1][1] < x[1][0]) into the constraint queue. Moreover, for the branch listener to determine the priority of the branches added, we pass the positions of their associated neurons to the symbolic execution engine by calling `register_current_indices` before every if-statement. Recall that the calling of $\max(x, i)$ on i is used to calculate the i-th row of the $attention\_scores$

Algorithm 5 Functions for performing softmax and max

```
1: function softmax(x):
2:    x_max ← [ max(x, i) for x[i] in x ]
3:    e_x ← [ [ e^(x[i][j] − x_max[i]) for x[i][j] in x] ]
4:    e_x,sum ← [ sum(e_x[i]) for e_x[i] e_x ]
5:    result ← [ [e_x[i][j]/e_x,sum[i] for e_x[i][j] in e_x] ]
6:    return result

1: function max(x, i):
2:    register_current_indices([ (i, k) for k = 0, ..., model_dim − 1 ])
3:    max ← x[i][0]
4:    for x[i][j] in x[i]:
5:       if x[i][j] > max:
6:          max ← x[i][j]
7:    return max
```

matrix, whose value at the end affects the i-th row of the output neurons of the multi-head attention layer. Thus, the associated neurons here are assigned with the i-th row of the output neurons, as in Line 2 of the `max` function, where $model\_dim$, the dimension of the input encoding, is the second dimension of the output neurons.

Line 3 of the `softmax` function subtracts each element $x[i][j]$ by its corresponding row's maximum value $x_{\max}[i]$. This step is crucial to prevent issues such as exponential explosion or decay, as it stabilizes the computation. Subsequently, each element is exponentiated. In our running example, we have $e_x = [[\exp(0), \exp(3 - 3 * [\![2, v]\!]^2)/\sqrt{2})], [\exp(0), \exp(6 - 6 * [\![2, v]\!])/\sqrt{2})]]$. Note that our tool uses Python's `math.exp` function for exponentiation, which cannot handle concolic variables. Hence, all concolic variables inside $exp(\cdot)$ are downgraded to concrete values, yielding a new matrix where each element represents the weight of the corresponding element in x relative to the maximum value of its row. In our example, it leads to $e_x = [[1, \exp((-9)/\sqrt{2})], [1, \exp((-12)/\sqrt{2})]]$.

Next, the elements of e_x are summed for each row, resulting in a new matrix $e_{x,\text{sum}}$ where each element represents the total weight of its row. Finally, each element in e_x is divided by the corresponding row's total weight to obtain the final output of the `softmax` function, i.e., the normalized probability values:

$$\begin{bmatrix} \frac{1}{1+\exp(-9/\sqrt{2})} & \frac{\exp(-9/\sqrt{2})}{1+\exp(-9/\sqrt{2})} \\ \frac{1}{1+\exp(-12/\sqrt{2})} & \frac{\exp(-12/\sqrt{2})}{1+\exp(-12/\sqrt{2})} \end{bmatrix} \approx \begin{bmatrix} 0.998 & 0.002 \\ 0.986 & 0.014 \end{bmatrix}$$

The final step in the dot-product attention involves matrix multiplication between the computed attention scores and the attention values, as illustrated in Line 5 of Algorithm 4. The matrix multiplication of the attention scores and attention values (the symbolic result of `dpt`(Q, K, V)) is computed as follows:

$$\begin{aligned} attentions \;=\; & [[0.998 * [\![2, v]\!] + 1.002, \quad 1.996 * [\![2, v]\!] + 2.004], \\ & [0.986 * [\![2, v]\!] + 1.014, \quad 1.972 * [\![2, v]\!] + 2.028]] \end{aligned}$$

Algorithm 6 Function for concatenating and transforming

1: **function** `concat`(*attentions*, *weights*, *bias*):
2: $n \leftarrow$ num_heads
3: $m \leftarrow$ key_dim_per_heads
4: **for** $word = 0, \ldots,$ seq_len $- 1$
5: **for** $i = 0, \ldots,$ model_dim $- 1$
6: $outputs[word][i] \leftarrow \sum_{j=0}^{n} \sum_{k=0}^{m} (attentions[j][word][k] \times weights[j][k][i]) + bias[i]$
7: **return** *outputs*

Concatenation and transformation of attention heads. After computing attention scores, the individual attention heads' results are concatenated and further transformed to produce the final output sequence. This process involves the careful aggregation of attention head outputs and their transformation using output weights and biases. It ensures the integration of information from multiple heads into a coherent representation of the input sequence. Algorithm 6 gives the code of this computation. Here, the result of `concat`$(attentions, [[[1], [1]]], [1,])$ is $[[2.994 * [\![2, v]\!] + 4.006], [2.958 * [\![2, v]\!] + 4.042]]$.

Following the aforementioned multi-head attention layer, the computation process continues through subsequent flatten, reshape, and dense layers. The calculation flow of the concolic variable within these layers is analogous to the previously described algorithm.

Softmax concretization and under-approximation. In our implementation, concolic terms inside Python's `math.exp` are downgraded to their concrete values for numerical stability. This yields an *under-approximate* symbolic semantics for attention: the solver explores a subset of feasible branches around attention logits, and may therefore miss some flips, but any satisfying assignment that survives concrete re-execution is a valid counterexample. Practically, we mitigate search myopia by (i) prioritizing branch predicates from multiple layers/heads, and (ii) budgeting solver time per-constraint so that exploration does not stall on a single expensive softmax path.

3.4 SHAP-based Abstract Critical Decision Path Synthesis

Finally, we analyze the set of adversarial inputs by applying a variant definition of the *abstract critical decision path* in [26]. Specifically, assuming a classification model M and a background input dataset X (for calculating SHAP value), our calculation proceeds as follows:

First, we define a *SHAP-based relevance* $R(n, x)$ of a neuron n for an input x using the SHAP value:

$$R(n, x) \ := \ \texttt{shap}(n, M(x), M_{\leq l}(x) \mid M_{>l}, X)$$

Here, we again consider the SHAP value on the submodel $M_{>l}$ of M starting from layer l to the output layer, so that n is an input neuron of $M_{>l}$, ensuring its SHAP value is well-defined.

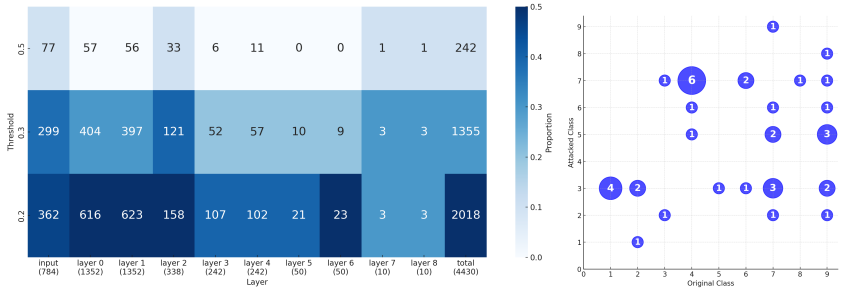

Fig. 3: Right: Adversarial attacks: original vs attacked classes; Left: The abstract critical decision paths of the adversarial inputs found for the CNN model.

Using this, we define the *SHAP-based critical decision neurons* $\texttt{cdn}(l, x \mid \alpha)$ in a layer l for an input x as the largest set of neurons in l such that:

1. $|\texttt{cdn}(l, x \mid \alpha)| \leq \alpha \cdot |l|$
2. For every neuron $n \in \texttt{cdn}(l, x \mid \alpha)$, $R(n, x) > 0$.
3. For every pair of neurons n and n' from layer l, if $n \in \texttt{cdn}(l, x \mid \alpha)$ and $n' \notin \texttt{cdn}(l, x \mid \alpha)$, then $R(n, x) \geq R(n', x)$.

In other words, $\texttt{cdn}(l, x \mid \alpha)$ represents the set of at-most-α neurons in the layer with the highest relevances.

Next, we define the *SHAP-based critical decision path*:

$$\texttt{cdp}(x \mid \alpha) \;:=\; \bigcup\nolimits_{l \,\in\, \text{Layers}(M)} \texttt{cdn}(l, x \mid \alpha)$$

We say a neuron n is *α-critical* if $n \in \texttt{cdp}(x \mid \alpha)$.

Finally, for any set of input data points A, we define its *SHAP-based abstract critical decision path* (ACDP) as the neurons that are α-critical in more than β of the data points in A:

$$\texttt{acdp}(A \mid \alpha, \beta) \;:=\; \{n \mid w(n, A \mid \alpha) > \beta\}$$

where the weight $w(n, A \mid \alpha) := |\{x \in A : n \in \texttt{cdp}(x \mid \alpha)\}| \;/\; |A|$ is the ratio of test inputs in the test suite A that have the neuron n in their α-critical decision path. Below, we exploit this definition to analyze PyCT's decision logic in generating adversarial attacks.

3.5 Common Decision Logic Behind Adversarial Inputs

We illustrate the effectiveness of abstract critical decision paths (ACDP) by inspecting real adversarial instances. We trained a CNN model over MNIST (Table 1, top); the model allows for 39 successful PyCT attacks using SHAP-based predicate prioritization (Table 3, top). We outline the joint distribution of the original and attacked class for the successful attacks in Figure 3. As the

Table 1: Architecture of the models used in our evaluation

Model Name	Layer Type	Output Shape	Param #
A toy CNN	conv2d	(None, 26, 26, 2)	20
	conv2d_1	(None, 11, 11, 2)	38
	dense	(None, 10)	510
	dense_1	(None, 10)	110
	Total params / Trainable params: 678		
Single-layer Attention Transformer	input_1 (InputLayer)	(None, 28, 28, 1)	0
	multi_head_attention	(None, 28, 28, 1)	897
	flatten	(None, 784)	0
	dense	(None, 10)	7850
	Total params / Trainable params: 8747		
Two-layer Attention Transformer	input_1 (InputLayer)	(None, 28, 28, 1)	0
	multi_head_attention	(None, 28, 28, 1)	897
	flatten	(None, 784)	0
	dense	(None, 128)	100480
	reshape	(None, 16, 8)	0
	multi_head_attention_1	(None, 16, 8)	1128
	flatten_1	(None, 128)	0
	dense_1	(None, 10)	1290
	Total params / Trainable params: 103795		

plot shows, the combinations of original and attacked classes distribute nearly uniformly. Indeed, the distribution exhibits an entropy $H \approx 4.23$, which is rather close to the maximal possible entropy $\lg 39 \approx 5.29$.

Despite the seemingly random distribution of original and attacked classes, we still identified a decision logic behind the attacks. Specifically, we examined the set A of all adversarial inputs generated by PyCT and calculated its SHAP-based abstract critical decision path $\texttt{acdp}(A \mid \alpha, \beta)$ as defined in Section 3.4. Figure 3 (left) illustrates the results of $\texttt{acdp}(A \mid \alpha, \beta)$ with $\alpha = 20\%$ and $\beta \in \{20\%, 30\%, 50\%\}$. Recall that $\texttt{acdp}(A \mid \alpha, \beta)$ represents the neurons that are α-critical for more than β of the dataset A. With $\beta = 30\%$, 1355 out of 4430 neurons are identified as the most 20%-critical neurons by 30% of the adversarial inputs. Even when $\beta = 50\%$, 245 neurons are still recognized as 20%-critical for more than half of the adversarial inputs.

This empirical evidence suggests that even under a relatively random distribution of successful attacks, we can still discern a common decision logic behind the generated adversarial instances. In practice, this logic may accelerate effective test generation by biasing the search toward high-weight ACDP neurons and provide actionable targets for model repair or regularization, such as weight damping or constraint-based fine-tuning on these neurons.

Table 2: Experimental results of the Single-layer Attention Transformer model with 1 & 2 pixels (top), and of the Two-layer Attention Transformer model with 1 pixel (bottom).

search strategy	#atk	#iter	#sat	#unsat	#gen cons.	#sol cons	wall	cpu
PQ, 1-pixel	67 (S)	4	5	172	946	164	1031	430
	33 (U)	9	11	605	1192	512	2960	1163
FIFO, 1-pixel	65 (S)	8	9	526	1292	321	1272	410
	35 (U)	13	15	993	2016	909	2962	878
PQ, 2-pixels	6 (S)	4	3	87	1743	90	1532	472
	25 (U)	8	9	499	2105	509	3600	1358
FIFO, 2-pixels	1 (S)	2	1	0	1684	1	536	5
	32 (U)	13	13	801	3459	817	3600	792

search strategy	#atk	#iter	#sat	#unsat	#gen cons	#sol cons	wall	cpu
PQ, Prioritized	8 (S)	2	1	18	3334	19	472	154
	21 (U)	1	0	13	1791	13	3429	2067
PQ, Limited	7 (S)	2	1	2	2929	13	628	596
	22 (U)	1	0	18	2148	28	3600	2443
FIFO	11 (S)	7	6	297	11219	303	561	310
	18 (U)	5	4	74	8682	79	3600	1123

4 Experiments

To evaluate the performance of our influence-guided concolic testing framework, we conducted preliminary experiments on a Windows 11 desktop computer equipped with an Intel i7-12700 CPU, 32GB of RAM, and an RTX 3060 GPU. We examine two Transformer models trained on the Fashion-MNIST dataset; models and parameter counts are listed in Table 1. In our experiments, the multi-head attention often leads to considerable constraints (exceeding 1GB) in the final layers, which can cause the tester to stall. To mitigate this issue, we explored the balance between high-influence and low-complexity constraints through two strategies for managing the priority queue:

Prioritizing Layers. This strategy prioritizes addressing the smallest constraints from earlier layers before tackling the largest ones in the later layers.

Limiting Runtimes. This strategy restricts the time allocated for building each constraint. If the process takes longer than a threshold, the ongoing constraint is bypassed to focus on the next one.

Unless noted, each seed is allotted a wall-time budget of 3600 s and we report: #atk (count), #iter (iterations until success/timeout), #sat/#unsat (solver outcomes per attack), #gen cons./#sol cons. (generated vs. submitted constraints), and CPU/wall time (averages over the corresponding sets). A successful attack is a concrete label flip under the specified pixel budget. We outline results for single- and two-layer Transformers, and 1- vs. 2-pixel budgets in Table 2.

4.1 Effectiveness of SHAP-Based Prioritization in Targeting Critical Decisions

We first compare a SHAP-prioritized queue (PQ) against a standard FIFO order when guiding PyCT to flip model decisions.

Single-layer Transformer, 1-pixel budget. On successful attacks, PQ reaches counterexamples in half the iterations of FIFO (4 vs. 8), solves roughly half as many constraints (164 vs. 321, about 49% fewer), and reduces unsatisfiable solver calls by about two-thirds (172 vs. 526, about 67% fewer). These savings translate into a ≈19% lower wall time (1031 s vs. 1272 s) with similar CPU time (430 s vs. 410 s). Importantly, PQ also finds slightly more successful cases (67 vs. 65). Overall, prioritizing high-influence branches improves search precision—fewer, better-chosen solver queries yield faster flips with comparable compute.

Single-layer Transformer, 2-pixel budget. With a larger perturbation budget, PQ uncovers substantially more counterexamples (6 vs. 1). The per-success cost is higher (e.g., about 3-times longer wall time), largely because PQ tackles harder but more promising constraints (90 vs. 1 solved constraints). These results suggest that, when the goal is *finding* more failures, PQ delivers markedly better yield; when the goal is *time-to-first* example, FIFO can be cheaper on this easier setting.

Two-layer Transformer, 1-pixel budget. On this deeper architecture, PQ variants cut iterations by 71% (2 vs. 7) and solved constraints by 94–96% (19 or 13 vs. 303). PQ with *layer prioritization* lowers both wall time (about 16% faster) and CPU time (about 50% lower) compared to FIFO, but attains fewer total successes (8 vs. 11). Therefore, on deeper models, PQ makes each success cheaper, though FIFO can still produce more total successes within the fixed time budget.

Summary. Across settings, SHAP-based prioritization consistently reduces search *effort per success* (iterations and solver calls). On shallow models or small budgets this also improves time-to-flip; on deeper models the advantage depends on how we trade influence for constraint complexity.

4.2 Trading Off High-Influence and Low-Complexity Constraints

Transformer constraints at later layers can explode in size because each node conjoins all ancestors in the path, making final-layer formulas dominate the budget. We therefore study two PQ realizations on the two-layer Transformer (Table 2, bottom): (i) *PQ (Prioritized Layers)*, which solves earlier-layer constraints first; and (ii) *PQ (Limited Runtimes)*, which retains high-influence ordering but skips any single constraint that exceeds a 30 s build cap.

When it comes to computation effort and latency, both PQ variants dramatically shrink the work per success relative to FIFO: iterations drop from 7 to

Table 3: Attacking CNN with PyCT and DeepConcolic.

search strategy	#atk	#iter	#sat	#unsat	#gen cons.	#sol cons	wall	cpu
PQ, ≤ 8 pixels	39 (S)	32	31	1127	4142	1158	612	181
	60 (U)	76	76	3820	10701	3897	1800	578
FIFO, ≤ 8 pixels	15 (S)	63	63	1240	13140	1304	608	188
	84 (U)	154	156	3278	30234	3435	1779	638
DeepConcolic (L_0)	94 (S)	-	-	-	-	-	-	38
	6 (U)	-	-	-	-	-	-	-
DeepConcolic (L_∞)	1 (S)	-	-	-	-	-	-	370
	99 (U)	-	-	-	-	-	-	-

2, and solved constraints from 303 to 19 (prioritized layers) or 13 (limited runtimes), i.e., 94–96% fewer. PQ with prioritized layers also reduces wall time from 561 s to 472 s, about 16% faster, and halves CPU time from 310 s to 154 s. By contrast, PQ with limited runtimes avoids getting stuck on pathological formulas but can spend more CPU overall (596 s) due to frequent constraint handovers.

For the effect on yield, FIFO attains more total successes (11 cases) within the fixed attack budget than either PQ variant (8 and 7 cases). This is consistent with FIFO opportunistically solving many *small* constraints, some of which happen to flip the output, whereas PQ invests in fewer, higher-influence (often larger) constraints.

Summary. Prioritized layers minimize iterations/solves and improve wall/CPU time, while FIFO maximizes the count of counterexamples within a hard time cap. By selecting the most critical constraints but limiting the build time, PyCT avoids getting stuck on expensive constraints while still benefiting from prioritizing high-influence constraints. Furthermore, constraints generated in early layers provide sufficient numbers of constraints with different SHAP values, which can be solved based on their priority values.

Based on these empirical results, we recommend using PQ with prioritized layers when the objective is to obtain quick, interpretable counterexamples and downstream triage (minimal solve footprint). On the other hand, we recommend using FIFO or PQ with limited runtimes when the objective is to maximize the number of distinct failures discovered under a strict time cap on deep models. In concept, it is possible to combine schedules to capture the best of both worlds: One can start with PQ with prioritized layers for rapid early wins; if no flip occurs within a short slice (e.g., 2–3 iterations), fall back to FIFO or to PQ with limited runtimes to diversify explored paths. Assessments of more sophisticated scheduling strategies for PQ are left as an important future work.

(a) Original images

(b) DeepConcolic, L_0 norm

(c) DeepConcolic, L_∞ norm

(d) PyCT, 2 pixels

Fig. 4: Example adversarial cases generated by DeepConcolic and PyCT. DeepConcolic successfully attacked all three images with L_0 norm, but failed on digit 7 with L_∞ norm. PyCT with a 2-pixel attack budget failed on digit 4.

4.3 Comparison with DeepConcolic

We conducted a preliminary comparison with DeepConcolic [19],[4] a state-of-the-art concolic testing tool designed specifically for generating adversarial inputs for ReLU-CNNs, over the model and dataset we used in Section 3.5. The results show that DeepConcolic has distinct attack performance for different perturbation norms. For perturbations measured by the L_0 norm, DeepConcolic achieves a success rate of over 90% with very short runtimes (approximately 30 seconds on average). However, this rapid performance comes at the cost of heavily altering the seed images (Figure 4). In contrast, the L_∞ norm tends to preserve the visual similarity between the adversarial and original images; DeepConcolic's success rate drops to around 1%, despite the examples being almost indistinguishable from the original images. PyCT takes an average runtime between those of DeepConcolic in L_0 and L_∞ settings, producing adversarial examples that are much more subtle than those generated under DeepConcolic's L_0 setup, yet avoiding the very low success rate observed in the L_∞ setting.

The contrasting outcomes between DeepConcolic and PyCT can be attributed to their distinct testing pipelines. Specifically, DeepConcolic is *coverage-driven*: a structural requirement (e.g., maximizing the node coverage metric) is selected and solved symbolically; adversariality is checked only post hoc by a robustness oracle. The solver is thus optimized to satisfy coverage constraints rather

[4] Tool was downloaded from `https://github.com/TrustAI/DeepConcolic`. We used the following command: python -m deepconcolic.main –model {model_name}.h5 –outputs out/{model_name} –max-iterations {max} –dataset {dataset} –criterion nc –norm {norm} –save-all-tests. In our experiments, {norm} was either l0 (L_0) or linf (L_∞), and {max} was replaced with the maximal possible iteration number that ensures a runtime within 1800 seconds.

than to change the model's decision. In contrast, PyCT is *flip-oriented*: each solver call targets a bypassed branch on the current concrete path, prioritized by SHAP-based influence on the output; solutions are immediately validated for a concrete label flip (Algorithms 1-2). This design choice trades volume of tests for decision-changing tests under tight attack budgets.

In DeepConcolic's own evaluation on ReLU-CNNs [19], the adversarial search allows generous budgets for L_0 (e.g., up to 100 pixels) and moderate budgets for L_∞ (e.g., $\varepsilon = 0.3$). Such L_0 budgets make it easy for a coverage-driven generator to cross many ReLU boundaries, explaining its very high L_0 success and the visible artifacts. Under small, uniform per-pixel changes, the same coverage-first search often achieves the requirement but not a label flip, producing the low success rate reported in Table 3 and the near-original visuals in Figure 4(c). Our PyCT experiments deliberately operate under small pixel budgets (e.g., ≤ 8 pixels), spending those edits where influence is highest. This yields subtle examples at an average runtime between the L_0 and L_∞ settings of DeepConcolic.

Summary. While both PyCT and DeepConcolic exploit concolic testing to generate adversarial attacks, they have distinct advantages. Empirically, if the goal is structural test generation (e.g., maximizing some coverage metric) under relaxed budgets, DeepConcolic is a strong choice. If the goal is to find visually subtle, decision-changing counterexamples under tight perturbation limits, PyCT's influence-guided, path-predicate strategy might be better aligned.

4.4 Limitations and threats to validity

Our evaluation has the following limitations and threats to validity. First, the SHAP ranking used in PyCT's branch exploration depends on the background dataset; different choices may alter branch priorities. Second, our implementation concretizes concolic terms inside Python's `math.exp` during softmax for numerical stability (Section 3.3), which under-approximates branching around attention logits and could bias the explored constraint set. Finally, although we report consistent trends, absolute counts and timings are specific to the models, datasets, and hardware in our evaluation.

5 Conclusion

We have reported preliminary results on influence-guided concolic testing, which aims to focus solver effort on high-leverage decisions and produce valid counterexamples. By leveraging SHAP values to prioritize influential neurons, our testing framework uncovers vulnerabilities that traditional coverage-based methods may overlook. As neural network architectures continue to grow in size and complexity, it is essential to optimize the scalability of our approach. Future work will focus on reducing the computational overhead of constraint solving and symbolic execution, as well as evaluating more versatile prioritization strategies to achieve effective path exploration of large models.

References

1. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
2. Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.
3. David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
4. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
5. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
6. Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436, 2015.
7. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
8. Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.
9. Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer-Aided Verification (CAV)*, pages 97–117. Springer, Springer, 2017.
10. Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems*, 31, 2018.
11. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium*, pages 1599–1614, 2018.
12. Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *Proceedings of the 31st International Conference on Computer-Aided Verification (CAV)*, pages 443–452. Springer, 2019.
13. Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
14. Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 31, 2018.

15. Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2018.
16. Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.
17. Fang Yu, Ya-Yu Chi, and Yu-Fang Chen. Constraint-based adversarial example synthesis. *arXiv preprint arXiv:2406.01219*, 2024.
18. Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. Pyct: A python concolic tester. In *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings 19*, pages 38–46. Springer, 2021.
19. Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Deepconcolic: Testing and debugging deep neural networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 111–114. IEEE, 2019.
20. Zhiyang Zhou, Wensheng Dou, Jie Liu, Chenxin Zhang, Jun Wei, and Dan Ye. Deepcon: Contribution coverage testing for deep learning systems. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 189–200. IEEE, 2021.
21. Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. Structural coverage criteria for neural networks could be misleading. *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 89–92, 2019.
22. Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.
23. Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.
24. Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMLR, 2017.
25. Hugh Chen, Scott M Lundberg, and Su-In Lee. Explaining a series of models by propagating shapley values. *Nature communications*, 13(1):4512, 2022.
26. Xiaofei Xie, Tianlin Li, Jian Wang, Lei Ma, Qing Guo, Felix Juefei-Xu, and Yang Liu. Npc: Neuron path coverage via characterizing decision logic of deep neural networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–27, 2022.
27. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
28. Gregory Bonaert, Dimitar I Dimitrov, Maximilian Baader, and Martin Vechev. Fast and precise certification of transformers. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 466–481, 2021.
29. Jieren Deng, Yijue Wang, Ji Li, Chao Shang, Hang Liu, Sanguthevar Rajasekaran, and Caiwen Ding. Tag: Gradient attack on transformer-based language models. *arXiv preprint arXiv:2103.06819*, 2021.

30. Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness verification for transformers. *arXiv preprint arXiv:2002.06622*, 2020.
31. Chuan Guo, Alexandre Sablayrolles, Hervé Jégou, and Douwe Kiela. Gradient-based adversarial attacks against text transformers. *arXiv preprint arXiv:2104.13733*, 2021.
32. Charis Eleftheriadis, Nikolaos Kekatos, Panagiotis Katsaros, and Stavros Tripakis. On neural network equivalence checking using SMT solvers. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 237–257. Springer, 2022.
33. Hai Duong, ThanhVu Nguyen, and Matthew Dwyer. A DPLL(t) framework for verifying deep neural networks. *arXiv preprint arXiv:2307.10266*, 2023.
34. Dario Guidotti, Laura Pandolfo, and Luca Pulina. Verifying neural networks with non-linear SMT solvers: a short status report. In *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 423–428. IEEE, 2023.
35. Soham Banerjee, Sumana Ghosh, Ansuman Banerjee, and Swarup K Mohalik. SMT-based modeling and verification of spiking neural networks: A case study. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 25–43. Springer, 2023.
36. Ming-I Huang, Chih-Duo Hong, and Fang Yu. Concolic testing on individual fairness of neural network models. *arXiv preprint arXiv:2509.06864*, 2025.
37. Luca Pulina and Armando Tacchella. Challenging SMT solvers to verify neural networks. *AI Communications*, 25(2):117–135, 2012.
38. Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286. Springer, 2017.
39. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Advances in neural information processing systems*, 31, 2018.
40. Luiz Sena, Xidan Song, Erickson Alves, Iury Bessa, Edoardo Manino, Lucas Cordeiro, et al. Verifying quantized neural networks using SMT-based model checking. *arXiv preprint arXiv:2106.05997*, 2021.
41. Thomas A Henzinger, Mathias Lechner, and DJordje Zikelic. Scalable verification of quantized neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 3787–3795, 2021.
42. Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119, 2018.
43. Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Testing deep neural networks. *arXiv preprint arXiv:1803.04792*, 2018.
44. Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Deepconcolic: Testing and debugging deep neural networks. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 111–114, 2019.
45. M Pezze. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, New York, 2008.
46. Lichao Feng, Xingya Wang, Shiyu Zhang, and Zhihong Zhao. Deepfeature: Guiding adversarial testing for deep neural network systems using robust features. *Journal of Systems and Software*, 219:112201, 2025.

47. Hugh Chen, Scott M. Lundberg, and Su-In Lee. Explaining a series of models by propagating shapley values. *Nature Communications*, 13(1), 2022.
48. Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
49. Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. *arXiv preprint arXiv:1710.06081*, 2018.
50. Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 120–131, 2018.
51. Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
52. Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 146–157, 2019.
53. Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111. IEEE, 2018.
54. Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.
55. Tongtong Bai, Song Huang, Yifan Huang, Xingya Wang, Chunyan Xia, Yubin Qu, and Zhen Yang. Criticalfuzz: A critical neuron coverage-guided fuzz testing framework for deep neural networks. *Information and Software Technology*, 172:107476, 2024.
56. Phillip Schanely. Crosshair - an analysis tool for python that blurs the line between testing and type systems. `https://github.com/pschanely/CrossHair`, 2017. [Online; accessed 20-Oct-2024].
57. Björn I. Dahlgren. Pysym documentation. `https://pythonhosted.org/pysym/`, 2016. (2016).
58. D. Barsotti, A.M. Bordese, and T. Hayes. Pef: Python error finder. In *Selected Papers of the XLIII Latin American Computer Conference (CLEI)*, volume 339 of *Electronic Notes in Theoretical Computer Science*, pages 21–41. Elsevier, 2017.
59. Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In *Dependable Software Systems Engineering*, pages 26–41. IOS Press, Amsterdam, 2015.

A Pure-Python Semantics of Multi-Head Attention with Concolic Execution

This appendix gives a step-by-step, pure-Python semantics for the multi-head attention layer we have implemented for PyCT. It specifies (i) shapes and data flow; (ii) how branch predicates arise during execution and are registered; and (iii) the precise under-approximation we adopt inside `exp` for solver compatibility (cf. Section 3.3).

Notation and shapes. Let the input sequence be $X \in \mathbb{R}^{L \times d_{\text{model}}}$ (row t is token x_t). Let h be the number of heads and d_k the per-head key/query dimension. Weights and biases are stored as plain Python lists in the following shapes:

$$W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times h \times d_k}, \qquad B_Q, B_K, B_V \in \mathbb{R}^{h \times d_k},$$
$$W_O \in \mathbb{R}^{h \times d_k \times d_{\text{model}}}, \qquad B_O \in \mathbb{R}^{d_{\text{model}}}.$$

For concolic execution, we view a scalar as a pair $\langle c, \phi \rangle$ (concrete value c and symbolic expression ϕ); ordinary scalars are identified with $\langle c, \perp \rangle$. All list/loop computations below operate element-wise on such pairs.

Step 1: Linear projections and head splitting (`tas`)

For each head $i \in [0, h)$, time step $t \in [0, L)$, and feature $j \in [0, d_k)$, the pure-Python linear transform computes:

$$Q[i,t,j] = \sum_{k=0}^{d_{\text{model}}-1} X[t,k] \cdot W_Q[k,i,j] + B_Q[i,j],$$
$$K[i,t,j] = \sum_{k=0}^{d_{\text{model}}-1} X[t,k] \cdot W_K[k,i,j] + B_K[i,j],$$
$$V[i,t,j] = \sum_{k=0}^{d_{\text{model}}-1} X[t,k] \cdot W_V[k,i,j] + B_V[i,j].$$

The result shapes are $Q, K, V \in \mathbb{R}^{h \times L \times d_k}$. In code, this is the `tas` function (Algorithm 3), expressed with nested `for`-loops only.

Step 2: Scaled dot-product scores (`dpa`)

For each head i, define the score matrix $S_i \in \mathbb{R}^{L \times L}$ by

$$S_i[t,u] \;=\; \frac{1}{\sqrt{d_k}} \sum_{j=0}^{d_k-1} Q[i,t,j] \cdot K[i,u,j].$$

This is implemented by a naive `matrix_multiply` followed by division by $\sqrt{d_k}$ (Algorithm 4). No external libraries are used.

Step 3: Numerically stable softmax (`softmax`)

For each head i and row t, define the row maximum $m_{i,t} = \max_{u \in [0,L)} S_i[t,u]$. We compute $m_{i,t}$ via a Python loop with an `if`-ladder, which evaluates boolean comparisons over concolic numbers and therefore registers branch predicates to the queue (the negations of every guard encountered). Formally, evaluating

`if (z > m)` emits the predicate $(z \le m)$ to the worklist with the associated output-row neurons $(t, \cdot)$.

To avoid overflow/underflow, we subtract the row max, then exponentiate and normalize:

$$\widehat{S}_i[t,u] \;=\; S_i[t,u] - m_{i,t}, \qquad P_i[t,u] \;=\; \frac{\exp(\,\mathsf{conc}(\widehat{S}_i[t,u])\,)}{\sum_{v=0}^{L-1} \exp(\,\mathsf{conc}(\widehat{S}_i[t,v])\,)}.$$

Here $\mathsf{conc}(\langle c, \phi\rangle) = c$ concretizes any concolic term inside `math.exp`, yielding an under-approximate symbolic semantics for softmax (path exploration may miss some branches), while preserving the *validation oracle* soundness because every synthesized input is re-executed concretely (Section 3.3). Each row $P_i[t, \cdot]$ is a probability simplex.

Step 4: Head outputs and value aggregation

For each head i and time step t, the attention output is

$$A_i[t,j] \;=\; \sum_{u=0}^{L-1} P_i[t,u] \cdot V[i,u,j], \qquad \text{so } A_i \in \mathbb{R}^{L \times d_k}.$$

This is a plain nested-loop matrix multiplication (Algorithm 4, Line 6).

Step 5: Concatenation and output projection (`concat`)

We conceptually concatenate heads along the last dimension and then apply the output projection using only base Python operations:

$$Y[t,\ell] \;=\; \sum_{i=0}^{h-1} \sum_{j=0}^{d_k-1} A_i[t,j] \cdot W_O[i,j,\ell] \;+\; B_O[\ell], \quad Y \in \mathbb{R}^{L \times d_{\text{model}}}.$$

The provided implementation computes this via `concat` (Algorithm 6) using list-of-lists; no reshapes require external libraries.

Step 6: Branch predicates generation and register

Branch predicates arise anywhere a Python boolean over concolic scalars is evaluated. In MHA this happens in the `max` routine used by softmax:

- For each row t of S_i, calling `max(row)` compares elements using `>` and triggers the branch listener. We call `register_current_indices`($[(t,k) \mid k < d_{\text{model}}]$) before the loop so that all guards in this row are associated with the output-row neurons $(t, \cdot)$.
- Each evaluated guard $(z > m)$ pushes its negation $(z \le m)$ into the priority queue with the SHAP-based influence score of the associated neurons. (See Algorithm 1 and Section 3.2 for the prioritize-solve-execute loop.)

Step 7: Properties and complexity

- *Row-stochasticity.* With concretization inside `exp`, every $P_i[t, \cdot]$ is non-negative and sums to 1.
- *Cost (single head).* Linear projections has time complexity $O(L\, d_{\text{model}}\, d_k)$; computing scores requires $O(L^2 d_k)$; aggregation requires $O(L^2 d_k)$. Multiply by h heads and output projection takes $O(L\, h\, d_k\, d_{\text{model}})$.
- *SMT-compatibility.* All steps use base Python control-flow, indexing, and arithmetic; the only transcendental is `math.exp`, guarded by $\mathsf{conc}(\cdot)$ as above.

Wrap-up: The end-to-end forward pass

Let $Q, K, V \in \mathbb{R}^{h \times L \times d_k}$ be as constructed in Steps 1–2, and define the per-head slices $Q_i := Q[i, :, :] \in \mathbb{R}^{L \times d_k}$, $K_i := K[i, :, :] \in \mathbb{R}^{L \times d_k}$, and $V_i := V[i, :, :] \in \mathbb{R}^{L \times d_k}$ for each head $i \in \{0, \ldots, h-1\}$. Form the scaled dot-product scores $S_i := \frac{1}{\sqrt{d_k}} Q_i K_i^\top \in \mathbb{R}^{L \times L}$ and the row-wise probabilities $P_i := \text{softmax}(S_i) \in \mathbb{R}^{L \times L}$ as in Step 3 (softmax uses `conc` inside `exp`). The per-head outputs are $A_i := P_i V_i \in \mathbb{R}^{L \times d_k}$ (Step 4). Concatenating heads and applying the output projection (Step 5) yields

$$Y[t, :] \;=\; \sum_{i=0}^{h-1} \sum_{j=0}^{d_k - 1} A_i[t, j]\; W_O[i, j, :] \;+\; B_O.$$

Branch predicates are registered only in the `max` used by `softmax`; arguments to `exp` are concretized as specified in Step 3, and every candidate input is validated by concrete re-execution.