Neural Networks as Artificial Specifications

I.S.W.B. Prasetya and Minh An Tran
Utrecht University, the Netherlands, s.w.b.prasetya@uu.nl

Abstract. In theory, a neural network can be trained to act as an artificial specification for a program by showing it samples of the program’s executions. In practice, the training turns out to be very hard. Programs often operate on discrete domains for which patterns are difficult to discern. Earlier experiments reported too much false positives. This paper revisits an experiment by Vanmali et al. by investigating several aspects that were uninvestigated in the original work: the impact of using different learning modes, aggressiveness levels, and abstraction functions. The results are quite promising.

Keywords: neural network for software testing, automated oracles

1 Introduction

Nowadays, many systems make use of external services or components to do some of their tasks, allowing services to be shared, hence reducing cost. However, we also need to take into account that third parties services may be updated on the fly as our system is running in production. If such an update introduces an error, this may affect the correctness of our system as well. One way to guard against this is by doing run time verification [2]: at the runtime the outputs of these services are checked against their formal specifications. Unfortunately, in practice it is hard to persuade developers to write formal specifications.

A more pragmatic idea is to use ‘artificial specifications’ generated by a computer. Another use case is automated testing. Tools like QuickCheck, Évosuite, and T3 [3,6,13] are able to generate test inputs, but if no specification is given, only common correctness conditions such as absence of crashes can be checked. Using artificial specifications would extend their range.

Although we cannot expect a computer to be able to on its own specify the intent of a program, it can still try to guess this intent. One way to do this is by observing some training executions to predict general properties of the program, e.g. in the form of ‘invariants’ (state properties) [5], finite state machine [12], or algebraic properties [4]. These approaches cannot however capture the full functionality of a program, e.g. [5] can only infer predefined families of predicates, many are simple predicates such as such as \(a \neq \text{null}\) and \(x + y \geq 0\). With respect to these approaches, neural networks offer an interesting alternative, since they can be trained to simulate a function [9].

The trade off of using artificial specifications is the additional overhead in debugging. When a production-time execution violates such a specification, the
failure may be either caused by an error triggered by the execution, or by an error in the training executions that were reflected in the predictions, or due to inaccuracy of the predictions. The first two cases expose errors (though the second case would take more effort to debug). However, the failure in the last case is a false alarm (false positive). Since we do not know upfront if a violation is a real error or a false positive, we will need to investigate it (debugging), which is quite labour intensive. If it turns out to be a false positive, the effort is wasted. Despite the potential, studies on the use of neural networks as artificial specifications are few: [16,11,10]. They either reported unacceptably high rate of false positives, or do not address the issue.

In this paper we revisit an experiment by Vanmali et al. [16] that revealed ≈ 16% rate of false positives—a rate of above 5% is likely to render any approach unusable in practice. The challenge lies in the discrete nature of the program used as the experiment subject, making it very hard to train a neural network. This paper explores several aspects that were left uninvestigated in the original work, namely the influence of different learning modes, aggressiveness levels, and abstraction. The results are quite promising.

2 Neural Network as an Artificial Specification

Consider a program $P$ that behave as a function $I \rightarrow O$. An artificial specification $\phi$ is a predicate $I \times O \rightarrow \text{bool}$; $\phi(x, P(x)) = \text{T}$ means that $P$’s output is judged as correct, and else incorrect. With respect to the intended specification $G$, $\phi$’s judgment is a true positive is when both $\phi$ and $G$ judge a $\text{T}$, a true negative is when they agree on the judgement $\text{F}$, a false positive is when $\phi$ judges $\text{F}$ and $G$ judges $\text{T}$, and a false negative is when $\phi$ judges $\text{T}$ and $G$ judges $\text{F}$.

An neural network (NN) is a network of ‘neurons’ [9] that behaves as a function $\mathbb{R}^M \rightarrow \mathbb{R}^N$. We will restrict ourselves to feed forward NNs (FNNs) where the neurons are organized in linearly ordered layers [9]; an example is below:

- The first layer is called the input layer, consisting of $M$ neurons connected to the inputs. The last layer is the output layer, consisting of $N$ neurons that produce the outputs. The layers in between are called hidden layers. An input neuron simply passes on its input, else it has $k$ inputs and an additional input called ‘bias’ whose value is always 1 [9]. Each input connector has a weight $w_i$. The neuron’s output is the weighted sum of its inputs, followed by applying a so-called activation function: $\text{out} = f(\Sigma_{0 \leq i \leq k} w_i x_i)$. A commonly used $f$ is the logistic function, which we also use in our experiments.

Any continuous numeric function $\mathbb{R}^M \rightarrow \mathbb{R}^N$, restricted within any closed subset of $\mathbb{R}^M$, can be simulated with arbitrary accuracy by an FNN [7], which implies that an FNN can indeed act as an artificial specification for $P$, if $P$ is injectable into such a numeric function. That is, there exists a continuous numeric function $F: \mathbb{R}^M \rightarrow \mathbb{R}^N$ and injections $\pi_I: I \rightarrow \mathbb{R}^N$ and $\pi_O: O \rightarrow \mathbb{R}^N$ such that $F$ encodes $P$: for all $x \in I$, $P(x) = \pi_O^{-1}(F(\pi_I(x)))$. However, finding a right FNN is hard. A common
审批（公民，州，地区，性别，年龄，婚姻，依赖，收入）{
  if (地区==5 || 地区==6) 金额=0；
  else if (年龄<18) 金额=0；
  else {
    if (公民==0) {
      金额 = 5000+1000*收入；
      if (州==0)
        if (地区==3 || 地区==4) 金额 = 金额+2；
        else 金额 = (int)(金额+1.50)；
      else 金额 = (int)(金额+1.10)；
      if (婚姻==0)
        if (依赖>0) 金额 = 金额+200+依赖；
        else 金额 = 金额+500；
      else 金额 = 金额+1000；
      if (性别==0) 金额 = 金额+500；
      else 金额 = 金额+1000；
    } else 
      金额 = 1000 + 800 * 收入；
      if (婚姻==0)
        if (依赖>2) 金额 = 金额+100+依赖；
      else 金额 = 金额+100；
      else 金额 = 金额+300；
      if (性别==0) 金额 = 金额+100；
      else 金额 = 金额+200；
    }
  if (金额==0) 批准=F else 批准=T；
  return (批准，金额)；
}

图 1。实验主题：一个信用审批程序[16]。

技术之一是通过训练一个FNN来模拟一组样本输入和输出，例如使用反向传播[9]算法。这可能更容易训练NN来模拟$\alpha \circ P$，其中$\alpha$是P的输出值的某些选择抽象。折衷是我们得到一个较弱的规格说明。

由于NN并非直接产生布尔值，我们需要将其输出向量$\bar{z}'=\text{NN}(\pi_I(\bar{x}))$与一个所谓的比较器$C: \mathbb{R}^N \to \mathbb{R}^N \to \text{bool}$和输出向量$\bar{z}=\pi_O(\alpha(P(\bar{x})))$进行比较，以进行判断。如果其值对每个元素都很‘远’，则判断为F，否则为T。通过调整‘远’的含义，我们可以调整规格说明的激进程度，而不必修改NN的内部。在我们的实验中，虽然等式$\text{id}=(\lambda x.x)$将被用作注入算子$\pi_I$和$\pi_O$。因为id仅仅传递其输入，所以它将被从公式中省略。

3 实验

图1显示了一个来自金融领域的信用审批程序，这是Vanniali等[16]作为实验主题所使用的。该程序有8个输入参数描述了一个客户。输出是b,y的对，其中b是表示信用请求是否被批准的布尔值，如果y则表示最大允许的信用。我们将忽略b，因为[16]已经证明一个FNN可以准确预测其值。尽管其规模，主题相当具有挑战性，因为该程序在有限的域上操作（数值值都是整数）。整个输入域有224000个可能值。我们将使用一个FNN，输入有8个（表示approve的输入）和一个隐藏层24个神经元（添加更多层和神经元并不会显著提高FNN的准确性）。

Fig. 1. The experiment subject: a credit approval program from [16].
Five variations of the FNN will be used, as listed below, along with the used comparator $C$. $C$ is parameterized with aggressiveness level $A$ (integer 0 (least aggressive) ... 5) that determines $C$’s policy to deal with non-clear-cut cases.

1. The FNN **direct** has one output, which is trained to simulate $y$. Its comparator $C_A$ uses Euclidian distance, with sensitivity linearly scaled by $A$: $C_A(y, y') = |y - y'| < \epsilon_{\text{max}} - 0.01A$, with $\epsilon_{\text{max}}=0.09$.

2. The FNN **unimin** has $N$ outputs, trained to simulate $\alpha_N \circ \text{approve}$. The abstraction $\alpha_N$ maps approve’s $y$ output to a vector $\bar{z} : [0..1.0]^N$ representing one of $N$ uniform sized intervals in $y$’s range $[0..18000]$, such that the $k$-th interval is represented by a vector of 0’s except a single 1 at the $k$-th position. If $v : [0..1.0]^N$, let $\text{winner}(v)$ be the index of the greatest element in $v$.

   The comparator is more complicated. An obvious case is when $z' = \text{NN}(\bar{x})$ and $\bar{z} = \alpha_{10}(\text{approve}(\bar{x}))$ report the same winner. If the NN’s winner is confident of itself, approve’s output is judged as correct. When they produce different winners and the NN’s winner is confident of itself, we judge approve to be incorrect. Other cases are non-clear-cut and judged depending on the aggressiveness level. The full definition of $C_A$ is shown below. The original work Vanmali et al. [10] only uses $A = 3$ aggressiveness level.

   function $C_A(\bar{z}, z')$
   
   \begin{align*}
   k, j & \leftarrow \text{winner}(\bar{z}), \text{winner}(z') \quad \text{agree} \leftarrow k = j \\
   & \text{if} \quad \text{agree} \land |\text{agree} - z'_j| < \text{th}_{\text{low}} \text{ then (obvious match) } T \\
   & \text{else if} \quad \neg\text{agree} \land |\text{agree} - z'_j| > \text{th}_{\text{high}} \text{ then (obvious mismatch) } F \\
   & \text{else (non-clear-cut cases) case } A \text{ of } \\
   & 0 : \text{(least aggressive: always accept) } T \\
   & 1 : \text{(reject when the NN contradicts agreement) } \neg(\text{agree} \land |\text{agree} - z'_j| > \text{th}_{\text{high}}) \\
   & 2 : \text{(always accept on agreement) } \text{agree} \\
   & 3 : \text{(Vanmali et al. [10]: accept on conflicting results) } \neg\text{agree} \lor |\text{agree} - z'_j| > \text{th}_{\text{high}} \\
   & 4 : \text{(only accept if NN’s winner supports } \bar{z}) |\text{agree} - z'_j| < \text{th}_{\text{low}} \\
   & 5 : \text{(most aggressive: never accept) } F \\
   \end{align*}

end function

The thresholds $\text{th}_{\text{low}}$ and $\text{th}_{\text{high}}$ are set to 0.2/0.8.

3. The FNN **unimin** is a less presumptuous variant of uni, with $\text{th}_{\text{low}}/\text{th}_{\text{high}}$ set to 0.1/0.9. This will cause more cases to be regarded as non-clear-cut.

4. The FNN **lower** is like unimin, but trained to simulate $\alpha_N \circ \text{low} \circ \text{approve}$. low is used to ’stretch’ $\alpha_N$ to divide $y$ into finer intervals in the lower region of $y$’s range, e.g. if we believe the region to be more error prone, and growing coarser towards the other end. We use the log function to do this: $K \times \log(1 + y/a)$ with $K=8000$ and $a=100$ controlling the steepness.

5. The FNN **center** is like unimin, but trained to simulate $\alpha_N \circ \text{ctr} \circ \text{approve}$. ctr is used to ’stretch’ $\alpha_N$ to divide $y$ into finer intervals in the center region of $y$’s range. We use logistic function $\text{ctr}(y) = M/(1 + e^{-a(y-0.5M)})$ where $M=18000$ ($y$’s maximum) and $a=0.0006$ determines the function’s steepness.

**Training.** We randomly generate 500 distinct inputs (from the space of 224000 values) and collect the corresponding approve’s outputs. This set of 500 pairs (input,output) forms the training data. For every type of FNN above and every aggressiveness level an FNN is trained. $N$ controls the granularity of the used abstraction, so we also try various $N$ (10..60). For each FNN, the connections’ weight is randomly initialized in $[-0.5..0.5]$. The training is done in a series
of epochs using the back propagation algorithm [9]. We tried both the incremental learning mode [9,8], where the FNN’s error is propagated back after each training input, and batch learning modes, where only the average error is propagated back, after the whole batch of training inputs (500 of them). Incremental learning is thus more sensitive to the influence of individual inputs.

**Evaluation.** To evaluate the FNNs’ ability to detect errors, we run them on 21 erroneous variations (mutants) of the subject as in [16] —they are listed in the Appendix. For each mutant, 500 distinct random inputs are generated, whose outputs are ‘error exposing’ (distinguishable from the corresponding outputs of the correct subject). As an artificial specification, an FNN should ideally reject all these error exposing outputs. Each rejection is a true positive. We also generate 500 distinct random inputs and feed it to the (unmutated) subject. The FNN should accept the corresponding outputs —each rejection is a false positive.

**Fig. 2.** The true positive and false positive rates (in %) of different FNNs.

Figure 2 shows some of the results. Except for direct, the training was done in 1500 epochs with learning rate 0.5. We can see that using abstraction improves the FNN’s performance: compare direct with uni30. The latter obtains a true positive rate 68% on aggressiveness 2, implying that out of two erroneous executions, uni30 is likely to detect at least one, while when the aggressiveness level is set low, its rate of false positives is only around 2%. Abstraction also makes training easier: after 1500 epochs uni30 produces a mean square error (MSE) of $\approx 0.0001$, whereas the shown results for direct is obtained after 10000 epochs (incrementally) with 0.1 learning rate, yielding an MSE $\approx 0.0004$.

The experiment in [16] uses unimin10. We believe [16] used batch learning because the reported MSE after 1500 epochs matches, namely $\approx 0.05$. However, as can be seen in Figure 2, this leads to poor performance (batched unimin10). Incremental learning yields a much more accurate FNN ($\approx 0.0001$ MSE), hence also better performance (unimin10). The performance of the FNN in [16] under our setup is indicated by the vannali-markers in Figure 2.

The effect of using different abstractions and abstraction granularity (the N parameter) is shown in Figure 3. Based on the results in Figure 2, we now use
the lowest aggressiveness level (0). The graph of uni shows that increasing $N$ can greatly improve the FNN’s ability to detect error, while keeping the false positive rate below 5%. We also see $\alpha_N$ and $\alpha_N \circ \text{low}$ perform significantly better than $\alpha_N \circ \text{ctr}$, implying that the choice of the abstraction function matters. Compared to $\alpha_N$, $\alpha_N \circ \text{low}$ and $\alpha_N \circ \text{ctr}$ introduce non-linear granularity. The results suggest that introducing more granularity in the region (of $P$’s output) which are more error prone pays off.

**Fig. 3.** The effect of different abstractions and the abstraction granularity ($N$). uni shows the TP and FP rates of the uni$_N$ configuration with its aggressiveness level set to 0 —recall that this FNN uses the function $\alpha_N$ as abstraction. center and lower show the TP and FP rates of the same FNN, but they use respectively $\alpha_N \circ \text{ctr}$ and $\alpha_N \circ \text{low}$ as the abstraction.

4 Conclusion
The experiment showed that, contrary to earlier attempts, it is possible to train Neural Networks, given an appropriate abstraction, to become an artificial specification for a non-trivial program with acceptable precision. As future work, more case studies are needed to see how this generalizes.

References

1. Aggarwal, K., Singh, Y., Kaur, A., Sangwan, O.: A neural net based approach to test oracle. ACM SIGSOFT Software Engineering Notes 29(3), 1–6 (2004)
2. Cao, T.D., Phan-Quang, T.T., Felix, P., Castanet, R.: Automated runtime verification for web services. In: Int. Conf. on Web Services (ICWS). IEEE (2010)
3. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ACM Sigplan Int. Conf. on Functional Programming (2000)
4. Elyasov, A., Prasetya, W., Hage, J., Rueda, U., Vos, T., Condori-Fernández, N.: AB=BA: Execution equivalence as a new type of testing oracle. In: 30th ACM Symposium on Applied Computing. ACM (2015)
5. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1), 35–45 (2007)
A Results on Individual Mutations

The table below shows each of the mutation used in our experiment for simulating errors. The mutations are the same as originally used in [16].

| line | mutation |
|------|----------|
| 2: Region == 5 || Region == 6 | M1: Region==5, M2: Region==5 && Region==6, M3: Region==4 || Region==5, M4: Region==3 || Region==4 |
| 3: Age < 18 | M5: Age > 18, M6: Age < 25 |
| 5: Citizenship == 0 | M17: Citizenship == 1 |
| 7: State == 0 | M8: State == 1 |
| 8: Region == 3 || Region == 4 | M9: Region==3, M10: Region==3 && Region==4, M11: Region==2 || Region==3, M12: Region==1 || Region==2 |
| 11: Marital == 0 | M13: Marital == 1 |
| 12: Dependents > 0 | M14: Dependents == 0, M15: Dependents < 0 |
| 15: Sex == 0 | M16: Sex == 1 |
| 20: Marital == 0 | M17: Marital == 1 |
| 21: Dependents > 2 | M18: Dependents > 2, M19: Dependents < 2, M20: Dependents <= 2 |
| 24: Sex == 0 | M21: Sex == 1 |
Figure 4 shows the true positive and false positive rates of the FNNs on individual mutants. Two results of two FNNs are shown. The first is \texttt{uni}_{30} with its aggressiveness level set to 0; recall that \texttt{uni}_{30} uses the function $\alpha_N$ as the abstraction function. The second is \texttt{lower}_{30}, with aggressiveness 0, but it uses $\alpha_N \circ \texttt{low}$ as the abstraction.

Let’s first consider the true positives (top graph). We see here that on mutants M16, M17, and M20 \texttt{uni} actually performs very poorly. See the top graph in Figure 4—\texttt{uni}’s results on these three cases are annotated in the graph.

In contrast, the hardest mutants for \texttt{lower} are M13 and M14, but even on these mutants \texttt{lower} has a true positive rate of >10%. Whether this 10% is good enough depends on the situation. We have defined the rate of true positives as the percentage of wrong executions that the FNN judges as wrong as well. In particular, note that the metric is not defined as the percentage of mutations that can be discovered. If we would define it like this, \texttt{lower} would have 100% rate of true positives because with enough test cases eventually it will be able to detect all mutants. So, 10% individual rate of true positives for e.g. M14 means thus that if we manage to trigger at least 10 distinct executions that expose the mutation, statistically the FNN has a good chance to detect at least one of them, and thus identifying the mutation. While this sounds very encouraging, note that the actual probability for detecting the error also depends on the probability of producing executions that expose it. In the experiments, the probability of the latter is simply 1: we knew upfront that there is a mutation, so generating the set of error exposing executions for each mutant was not problematic. In a real regression testing setup, it is not possible to steer the testing process towards exposing a particular error; we do not even know upfront if the new version of the program would contain any regression error at all. There are indeed tools to automatically generate test inputs capable of generating a large number of test cases [3,14]. However, it is hard to generate test cases that are evenly distributed over all control paths in the target program. Some paths may even be left uncovered because they are too difficult to cover, even by tools that employ more sophisticated techniques like an evolutionary algorithm [6] or symbolic calculation [15].

The bottom graph in Figure 4 shows the individual false positive rate of \texttt{uni} and \texttt{lower}. Each bar in this graph also has its own error bar to indicate the standard deviation $\sigma$ of the value the bar represents (the error bar is capped above at 100 and below at 0, since true/false positive rates can only range between [0..100]). For each mutant $M$, and each experiment (e.g. \texttt{lower}), the error $E_M$ of the false positive rate of the experiment is calculated by randomly dividing the set of 500 executions used in the experiment, into 5 bags of 100 elements and then we calculate the false positive rate of the experiment with respect to each bag. $E_M$ is defined as the standard deviation of these values. The error bars indicate that sometimes the false positive rate can peak above 5%, though in average both configurations, \texttt{uni} and \texttt{lower}, produce rates that are below 5%, for every individual mutant.
Fig. 4. The graph at the top shows the individual true positive rate of the uni30 with its aggressiveness level set to 0 (left bars) and that of the same FNN but using $\alpha_N \circ \text{low}$ as the abstraction function (right bars) on each mutant (M0..M20). The bottom graph shows the individual false positive rate.