Finding Simplified Closed Form Approximations of MLPs.

László Dirks, Nicolai Radke

July 2021

1 Introduction

In this paper we investigate finding a simplified closed form approximation of a multilayer perceptron (MLP): Given a function template, for example $f(x) = a \cdot x + b$ and a trained MLP, we examine different methods to find values for the parameters a and b such that f approximates the MLP with certain error guarantees. We consider MLPs with piecewise linear activation functions and function templates, which can be encoded as a satisfiability modulo theories (SMT) formula. To do so, we focus on two different methods.

First, we look at an incremental SMT approach: given a set of samples S and a template, we use an SMT solver to fit the parameters of the template to the samples. In a second step, we use an SMT solver to find an outlier, meaning a sample $t \notin S$ such that the output of the MLP deviates at least a given ϵ from the function evaluated at this point. If such a sample is found, we add it to S and repeat the process.

The second, more traditional approach takes input-output pairs of the MLP and then uses existing least-squares curve fitting implementations to find parameters for the given template. Then we reuse the method to find outliers form the SMT approach to determine an upper and lower bound for the maximum difference between the MLP and the function found. Currently, only linear functions of arbitrary dimension and one-dimensional polynomials are supported for this approach.

Section 2 goes through some preliminaries before we give a more detailed description our work in Section 3. A very short overview of the implementation can be found in Section 4. The implementation is evaluated in Section 5. We give a short notion on related work in Section 6 before concluding the paper in Section 7.

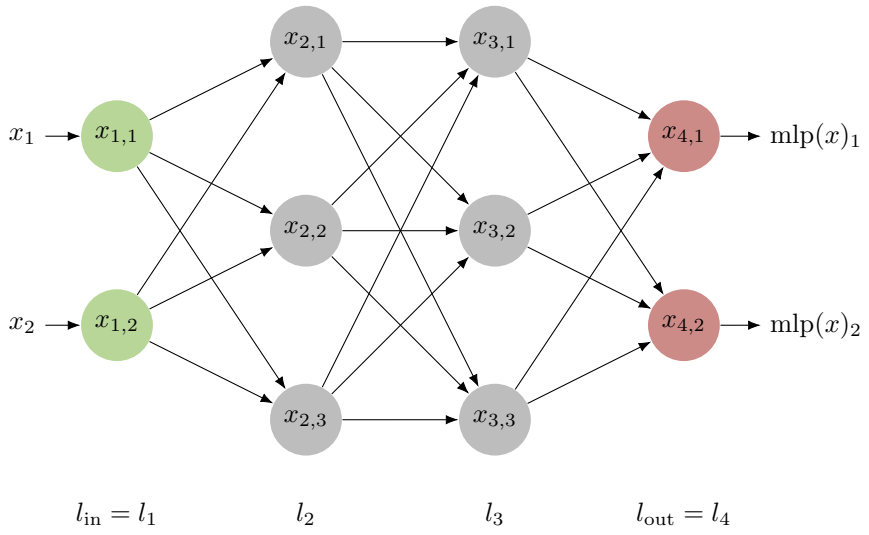

Figure 1: Structure of a fully connected MLP consisting of 4 layers: one input layer (●), two hidden layers(●) and one output layer (●). It computes a function $f : \mathbb{R}^2 \to \mathbb{R}^2, x = (x_2, x_2) \mapsto \text{mlp}(x) = (\text{mlp}(x)_1, \text{mlp}(x)_2)$

2 Preliminaries

2.1 Multilayer Perceptrons

A multilayer perceptron (MLP) is a type of neural network (NN). An MLP computes a function $f : \mathbb{R}^N \to \mathbb{R}^M, x \mapsto f(x)$. It can depicted using a number of layers consisting of nodes (see Figure 1). A multilayer perceptron has $L \geq 3$ layers: one input layer, at least one hidden layer and one output layer.

The input layer l_{in} consists of N nodes, the output layer l_{out} consists of M nodes, hidden layers can have an arbitrary number of nodes.

The j-th node in layer l is connected to a value $x_{l,j}$ which is computed from the values of all nodes in the previous layer $l-1$. We define $x_{l_{\text{in}},n} := x_n, n \in [1, ..., N]$ and $f(x)_m := x_{l_{\text{out}},m}, m \in [1, ..., M]$. For layers other than the input layer $l-1$ with I nodes and l with J nodes we define $x_{l,j} = h(\sum_{i=1}^{I} x_{l-1,i} \cdot \alpha_{i,j} + \beta_j)$ with an activation function h, weights $\alpha_{i,j} \in \mathbb{R}$ and biases $\beta_j \in \mathbb{R}$. Weights and biases are trained using error backpropagation. For details on this step we refer to [1]. There are multiple choices for the activation function. For the sake of simplicity, we only consider the rectified linear unit (ReLU) activation function, which is defined as follows:

$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{else} \end{cases} = \max(x, 0)$$

To be precise, the system described above is a fully connected feed forward

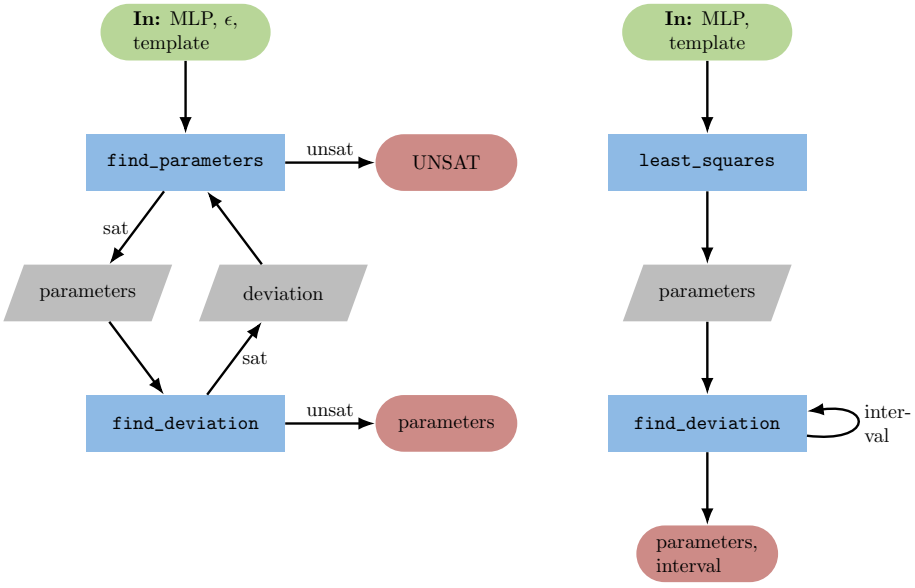

Figure 2: High-level flow-chart diagram for Algorithms 2, 3 (left) and 4 (right) using Algorithm 1.

neural network. In the following $\mathrm{MLP}(x) := \mathrm{mlp}(x)$ denotes the output of an MLP with input x.

2.2 Satisfiability and Logic

In this paper we use logical formulas to formalize the neural network and the function template. While the network can be described through linear constraints, this is not possible for the templates which may have an arbitrary form. Thus, we use quantifier free nonlinear real arithmetic (`QF_NRA`). To deal with these formulas, we use an SMT solver. We assume standard settings and notations.

3 Approximating an MLP

In this section we give a detailed description of the algorithms we implemented. A high-level flow-chart diagram of the algorithms can be found in Figure 2.

3.1 Finding deviations

```
def find_deviation(template, parameters, MLP,
                   ub, lb):

    # insert parameters in template to define f
    f = template(parameters)

    # encode conditions
    encoding = ∃x : |MLP(x) − f(x)| > ϵ ∧ (lb ≤ x ≤ ub)

    # use an external solver to find a solution
    Solver.add(encoding)
    result = Solver.check()

    if result == SAT:
        model = Solver.model()
        return model(x)
    return UNSAT
```

Algorithm 1: Method to find a deviation between the function and the MLP

All of the approaches we propose rely on the `find_deviation` method, which verifies whether found parameters for the function template yield a function that approximates the behaviour of the given MLP on a specified interval within some error bound ϵ. For this we encode the input-output relation of the MLP as introduced in [5]. Pseudocode of the presented algorithm can be found in Algorithm 1.

We construct a formula encoding the existence of values for the variables x and y such that $\text{MLP}(x) = y$. Accordingly, we encode the existence of values for the variables x' and y' such that $f(x') = y'$. Be lb a lower bound and ub an upper bound defining an input subspace. We assert both the encoding of the MLP and the function, with the found parameters to an of-the-shelf SMT-solver and further encode

$$x = x' \wedge x \geq lb \wedge x \leq ub$$
$$y - y' > \epsilon \vee y' - y > \epsilon$$

The concatenation of these formulas is satisfiable if and only if there is some input in the specified subspace such that the output of f and the MLP deviate more than ϵ. The model returned by the solver provides this input value.

To improve the solving time we developed a strategy to split the encoding of this problem into smaller sub-problems, which then can be solved in parallel. This is accomplished by omitting the encoding of the behaviour of the activation function ReLU for a node and instead solving two formulas, where the input of

the node is limited such that the output of the node is linear to the input. We refer to this as a *split*.

3.2 SMT-based Parameter Search

In this section, we describe two SMT-based variants for finding parameters of the given template. The left side of Figure 2 is an abstraction which works for both variants. The two variants differ in their realization of the `find_parameters` method.

3.2.1 Template Adjustment within ϵ

```
def find_parameters(template, MLP, ϵ, x):

    # encode conditions
    encoding = (∃parameters : f = template(parameters)
                ∧ |f(x) − mlp(x)| ≤ ϵ)

    # use an external solver to find a solution
    Solver.add(encoding)
    result = Solver.check()

    if result == SAT:
        model = Solver.model()
        return model(parameters)
    return UNSAT
```

Algorithm 2: Finding parameters within ϵ.

Our fist approach consists of the routine summarized in Algorithm 2. The input to our algorithm consists of a function template, an MLP, a maximal deviation ϵ and an lower and upper bound for the input lb, ub. We start by encoding the output y of the template function for some arbitrary, fixed input (e.g. $x = lb$) in dependence of the parameters and add the requirement that $y - \text{mlp}(x) \leq \epsilon \wedge \text{mlp}(x) - y \leq \epsilon$. Note that here $\text{mlp}(x)$ and x are constant values and ϵ is part of the input. The resulting formula has a model if and only if the there are parameters for the template such that the resulting function deviates at most ϵ from the MLP.

In the second step we call the `find_deviation` method to determine whether the found parameters are within the ϵ error bound for the entire subspace. If this is the the case, the method terminates. Otherwise the we use the counterexample provided by `find_deviation` to repeat the first step.

During the process the first step will be repeated for different inputs. For efficiency we use an incremental solver and assert the encoding for a different input in each iteration on top of the previous formulas. This way the resulting

parameters satisfy the requirement of the maximal deviation for the entire set of input samples.

3.2.2 Template Adjustment with Optimal ϵ

```
def find_parameters(template, MLP, X):

    # encode conditions
    encoding = ( ∃parameters, ε_max :
                   f = template(parameters)
                   ⋀_{x∈X}  |f(x) − mlp(x)| ≤ ε_max
                   ∧ (⋁_{x∈X}  |f(x) − mlp(x)| = ε_max) )

    # use an external solver to find a minimal
    # solution
    Solver.add(encoding)
    result = Solver.minimize(ε_max)

    if result == SAT:
        model = Solver.model()
        return model(parameters)
    return UNSAT
```

Algorithm 3: Finding parameters minimizing $\epsilon_{\max}$.

The approach described in this section is summarized in Algorithm 3. If we only consider function templates, which can be encoded using linear real arithmetic, we can use existing solvers [2] to find the minimal ϵ and parameters for the template such that for the resulting function f it holds that for all $x \leq ub, x \geq lb$, $|f(x) - \text{mlp}(x)| \leq \epsilon$.

This can be accomplished by modifying the first step of the previously introduced approach. Be $X = \{x_0, ..., x_{k-1}\}$ the set of input values in the k-th iteration. In stead of encoding the requirement $y_i - \text{mlp}(x_i) \leq \epsilon \ \wedge \ \text{mlp}(x_i) - y_i \leq \epsilon$ for all $x_i \in X$, we encode for each $x_i \in X$ the deviation of the MLP output and the function:

$$(y_i - \text{mlp}(x_i) \geq 0) \rightarrow (e_i = y_i - \text{mlp}(x_i)) \wedge$$
$$(\text{mlp}(x_i) - y_i > 0) \rightarrow (e_i = \text{mlp}(x_i) - y_i)$$

With that we can encode the maximal deviation:

$$\bigvee_{x_i \in X} (\epsilon_{\max} = e_i) \wedge$$
$$\bigwedge_{x_i \in X} (\epsilon_{\max} \geq e_i)$$

Finally, let the solver find a model of the encoding with the target function of minimizing $\epsilon_{\max}$.

3.3 Least-Squares Fit

```
def fit_verify(mlp_model, target_function, interval,
               size, epsilon, accuracy_steps):

    # take samples
    x, mlp(x) = sample(mlp_model, interval, size)

    # find parameters of target function
    parameters = least_squares(x, mlp(x),
                               target_function)

    # binary search to narrow down deviation
    # interval
    lower = 0
    upper = epsilon
    for _ in range(accuracy_steps)
        mid = (lower + upper)/2
        if find_deviation(mid):
            lower = mid
        else:
            upper = mid
    deviation_interval = [lower, upper]

    return parameters, deviation_interval
```

Algorithm 4: Method to fit parameters and find a deviation interval

Another approach we tested is using existing, traditional least-squares fit. Pseudo-code of the the method can be found in Algorithm 4, the right side of Figure 2 shows a high-level flow-chart diagram of the approach.

We used existing implementations of linear regression for linear functions of arbitrary dimension and polynomial fitting for 1D polynomials of arbitrary degree. Details on the functions used are in Section 4.

These methods take a number of $(x, f(x))$ pairs and then find the parameters for the best fitting curve. Thus, we take input/output samples of the neural network. The current implementation takes evenly spaced samples in a specified interval.

In contrast to the SMT methods from Section 3.2, we are not able to enforce the parameters to fulfill certain properties. Therefore, we are not able to use x values with large deviation to refine the parameters. We are only able

	SMT	Least-Squares Curve Fitting
similarities	• tries to find a curve fitting to the MLP • uses an SMT solver to check whether parameters fulfil given bounds	
differences	• *incrementally* includes outliers to find parameters • indirectly considers all points when finding new parameters • can guarantee that parameters with certain accuracy do not exist	• uses existing methods *once* to find parameters • only considers a finite set of points to find parameters • can only give guarantee for the accuracy of parameters found by the fitting function • incapable of improving parameters[1]

Table 1: Methodical comparison of the SMT approach and existing curve fitting approaches.

to check whether there exists a value x a with minimum deviation ϵ, meaning $\text{mlp}(x) - f(x) \leq \epsilon$. Therefore we use binary search to find an interval that is guaranteed to contain the maximal distance between the curve found and the output of the MLP. Since MLPs using only ReLU activation functions are picewise linear, it would also be possible to find the maximum deviation analytically. This could be part of further research.

4 Implementation

The implementation can be found in our GitHub repository [3]. We used the language Python for the implementation.

To create, train and store MLPs, we used `tensorflow` with `keras` and `sklearn`. For solving the formulas we used `z3`.

For curve fitting we used existing least-squares fit implementations. For linear regression we used `sklearn.linear_model.LinearRegression`, for polynomial fitting `numpy.polynomial.polynomial.polyfit`.

5 Evaluation

. Due to the novelty of this approach (see Section 6), it is impossible for us to compare it to existing ones. Due to the methodical differences of the presented

[1] It may be possible to improve the parameters through modifying meta-parameters of the fitting function, e.g. the number of samples. However, this does not guarantee improvement. Also including an outlier in the fitting process does not give any guarantees w.r.t. accuracy.

approaches (see Table 1), comparing them quantitatively with each other is also not sensible. Therefore we can (1) qualitatively evaluate the approaches and (2) quantitatively evaluate the approaches through comparing the performance of the same approach on different MLPs.

A methodical comparison can be found in Table 1.

Template	#Nodes	ϵ	#Splits	Runtime in s
1D linear function	12	0,5	0	1,23
			1	4,18
2D linear function	14	0,5	0	-
			1	6174.98
1D polynomial of degree 2	12	0,5	0	8,10
			1	7,44
1D polynomial of degree 3	17	0,5	0	6.34
			1	6.87
1D linear function	57	0,5	0	-
			1	-

Table 2: Test results for the method described in section 3.2.1. A timeout is denoted with -.

All tests were executed on the Intel Xeon Platinum 8160 Processors "Sky-Lake" (2.1 GHz) with 8 GB of memory. Each test with zero splits was given one CPU core and each test with one split was given two CPU cores and a timeout of 5 hours. Only the tests for the MLP with 57 nodes were given 16 GB of memory.

Table 2 and 3 show runtimes of the methods described in Section 3.2.1 and 3.2.2, respectively. The number of timeouts show that optimizing the parameters w.r.t. ϵ did not improve the runtime and lead to a notable number of timeouts. Splitting at certain nodes for parallelization did also not improve runtime. Table 4 shows the results of the method described in Section 3.3. We can see that this approach worked quite well and was also able to give relatively small deviation intervals using only 4 interval-refinement steps. However, `find_deviation` repeatedly lead to a timeout on a network with 57 nodes.

Template	#Nodes	ϵ	#Splits	Runtime in s
1D linear function	12	0,02	0	4,38
			1	5,87
2D linear function	14		0	-
			1	-
1D linear function	57		0	-
			1	-

Table 3: Test results for the method described in section 3.2.2. A timeout is denoted with -. The found bounds for ϵ are rounded to two decimal places.

Template	#Nodes	ϵ	#Splits	Runtime in s
1D linear function	12	[0,02; 0,03]	0	3,07
			1	5,25
2D linear function	14	[0,08; 0,09]	0	36,90
			1	30,57
1D polynomial of degree 2	12	[0,31; 0,33]	0	1,47
			1	3,53
1D polynomial of degree 3	17	[0,25; 0,27]	0	1,63
			1	3,80
1D linear function	57		0	-
			1	-

Table 4: Test results for the method described in section 3.3. A timeout is denoted with -.

We can generally observe that larger models lead to timeouts. Although repeatedly tried, we were not able to get results for any model with more than 20 nodes.

6 Related Work

To the best of our knowledge, extracting function from a trained MLP has not been investigated. However, part of our approach is MLP formalization and the verification of its properties (for details see Section 3.1). Neural network verification is a big field of study and there exists an abundance of approaches we could have used or adapted for our own implementation. It should be mentioned that the problem of network verification is mostly not interpreted as proving a direct functional relation between input and output. Thus, if an existing approaches is used for our task, we would most likely have to adapt it. For an in depth overview of verification algorithms for neural networks we refer to [4]. As performance was not a main priority, we omitted related work and did our own straight forward encoding and solving using `z3` as described in Section 4. Future work with focus on performance could use existing, more sophisticated approaches.

7 Conclusion & Outlook

We presented approaches for finding a closed form of MLPs which combined relatively simple ideas and existing methods.

The first approach described in Section 3.2 used a counterexample-guided search algorithm for parameters using a type of NN-verification. The evaluation showed that this approach does work in practice, however, scalability to larger networks or more complex functions is still an issue. Future work could try

different methods to scale the approach to more realistic network sizes. Another topic of interest could be to directly optimize the parameters w.r.t. the maximum deviation and not their sum.

Reusing the NN-verification of the first approach, the second approach used existing least-squares implementations to find parameters. Of course, finding parameters initially is much faster process. However, the least-squares the optimization considers only a finite set of points and optimizes w.r.t. least squares and therefore solves a different problem. As the function `find_deviation` was used in this approach as well, scalability issues were apparent too.

Another possible topic for future work, which would affect both approaches, could be to rewrite the function `find_deviation` so that it returns the maximum difference between the MLP and the function found. However, being an optimization problem, the runtime of this alternative implementation would be even longer.

In conclusion, we were able to present usable approaches for finding a closed form of MLPs. To be usable for real-world applications, future work is necessary.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: Lecture Notes in Computer Science. Springer, 2015.

[3] László Dirks and Nicolai Radke. *mlp_smt_closed*. URL: `https://github.com/ODE-Construction-with-SMT-at-ToHS/mlpSmtPrototype`.

[4] Changliu Liu et al. "Algorithms for verifying deep neural networks". In: *arXiv preprint arXiv:1903.06758* (2019).

[5] Ivan Papusha et al. "Incorrect by Construction: Fine Tuning Neural Networks for Guaranteed Performance on Finite Sets of Examples". In: *CoRR* (2020).