Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Master Thesis

Generating and Solving Temporal Logic Problems with Adversarial Transformers

submitted by

Jens Ulrich Kreber

on February 10, 2022

Supervisor

Prof. Bernd Finkbeiner, Ph.D.

Advisor

Dr. Christopher Hahn

Reviewers

Prof. Bernd Finkbeiner, Ph.D.

Prof. Dr. Isabel Valera

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 10 February 2022

Jens Ulrich Kreber

Abstract

The applications of deep learning models have recently advanced towards symbolic domains that require complex reasoning and on which previously, only classic algorithmic approaches have been used. One such domain is hardware and software verification, where linear-time temporal logic (LTL) is commonly used as specification language. Recent work made use of the neural Transformer architecture to obtain satisfying traces for given LTL formulas. Firstly, this thesis continues the idea and tackles the LTL satisfiability problem both with standard Transformer encoders and universal ones, where especially the latter achieve a remarkable accuracy. To enable training, a new data generation process for obtaining balanced classes of LTL formulas based on flexible patterns is presented.
Secondly, this thesis addresses a fundamental problem of learning in symbolic domains: Obtaining a representative training dataset when no rule-based generation process is available, but only a small set of collected real-world instances. To this end, a new type of generative adversarial network (GAN) for symbolic domains built with Transformer encoders is presented. It is able to synthesize large amounts of syntactically correct and diverse LTL formulas without autoregression. Two variants, a standard GAN and a Wasserstein GAN with gradient penalty are compared with respect to their performance and training requirements. On the LTL satisfiability problem, it is demonstrated that GAN-generated data can be used as a substitute for real training data when training a classifier and, especially, that training data can be generated from a dataset that is too small to be trained on directly. The GAN setting also allows for altering the target distribution: By adding a classifier uncertainty part to the generator objective, a new dataset is obtained that is even harder to solve than the original one.

Contents

1 Introduction

Temporal logics, first introduced by [81], play an important rule in verification, where they are commonly used to specify system behavior [85], such as in the IEEE standard PSL [49]. The algorithmic validation of specifications is computationally expensive: Checking the satisfiability of a propositional linear-time temporal logic (LTL) formula is PSPACE-hard [93]. A possible alternative are learned models, which could potentially identify common patterns in specifications and thereby circumvent expensive checking. Recent developments in machine learning encourage the training of a deep neural network architecture directly on the LTL satisfiability problem, which is presented in Section 4 of this thesis. A problematic aspect of applying a learning approach to a domain where classically, only algorithmic solvers are used, is the absence of reliable training data. Existing datasets, e.g. from competitions [11, 51], are too small to train a deep neural solver directly. A solution approach to this is presented in Section 5: A new type of generative adversarial network (GAN) that can generate realistic and challenging temporal specifications from only a small data source.

Temporal logic satisfiability

Logics play a central role in mathematics and computer science. With them, properties of real world objects or objects of thought can be precisely expressed. Similar to natural language, existing phrases can be combined to build arbitrarily complex ones. A basic yet ubiquitous logic is propositional logic (PL), which reasons about a fixed set of propositions $(a, b, ...)$ that can either be true or false. By using boolean connectives such as $\neg$ "not" and $\wedge$ "and", larger expressions like $a \wedge \neg b$ "a and not b" can be formed. While this already has many applications, in other contexts one would like to describe things that change over time. This is especially important for reactive systems such as real-word devices (e.g. an elevator) or computer programs, for which it is desirable to formally describe their temporal behavior. Systems can then be monitored [72] or even proven to satisfy their specification in any given scenario, which is known as model checking [17]. One of the most widely used logics to describe temporal system behavior is propositional linear-time temporal logic (PLTL or simply LTL, [81]), both in science and industry [85, 89]. Like standard propositional logic, it reasons over a fixed set of propositions, but their truth value may change over time. An LTL formula is evaluated on a so-called *trace*, which corresponds to an infinitely long observation of a system in fixed time steps, where at each time step an arbitrary subset of the propositions can hold or not. Exemplary, in Figure 1, the beginning of a simple trace is shown. With the LTL formula $\bigcirc\Box b$ (read: "next globally b"), one can express that starting from time step 1, b holds indefinitely. The more complex formula $\Box(a \leftrightarrow \bigcirc\neg a)$ states that a holds exactly every second time step. The trace shown in Figure 1 satisfies both of these formulas. Note that also a trace with the truth values of a swapped would satisfy them both.
As a more realistic example why such specifications are useful, consider an arbiter that controls a single shared medium between two parties (Figure 2). Both can request usage of the medium (modeled as propositions r_1 and r_2) at any time, which can be granted

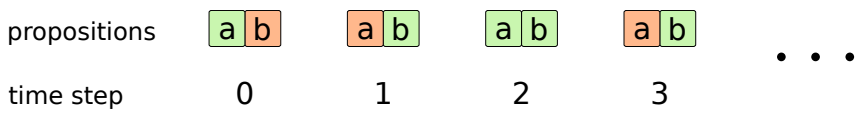


Figure 1: Simple trace with two propositions. Green background indicates that the proposition holds in that step, red that it does not.

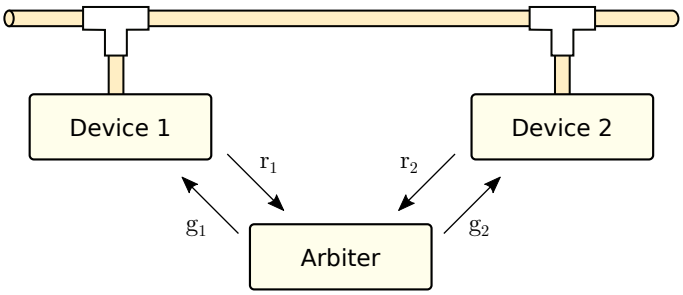


Figure 2: Example of an arbiter controlling access to a shared medium

by the arbiter (g_1, g_2). Obviously, to avoid data collision, the two parties must never be granted access to the medium simultaneously, which can be conveniently expressed in LTL as $\square\neg(g_1 \wedge g_2)$ and is typically called a safety requirement. Since this could be trivially fulfilled by never granting any request, additional liveness requirements such as $\square(r_1 \rightarrow \diamondsuit g_1)$ (r_1 implies that eventually g_1 happens) need to be added to the specification to achieve the desired behavior. Given a logical description of a system (e.g. the arbiter) and a specification, a model checking tool can then examine whether it actually adheres to the specification in any possible scenario.
In large scale, complex systems, specifications get increasingly complicated. As they are formulated by humans, they may contain mistakes, even making the specification itself contradictory. Since model checking is resource and time intensive and to keep debugging effort to a minimum, it is therefore beneficial to detect unsatisfiable specifications. Satisfiability checking refers to determining whether some trace that adheres to the specification formula exists at all. It is a common task in computer science and a widely studied problem for propositional logic (commonly referred to as "SAT"). For PL, it is NP-complete [19] and therefore impossible to solve efficiently at scale, but many highly optimized solvers exist that can solve problems with several hundred thousands of variables and millions of clauses relatively quickly [42]. The satisfiability problem of propositional LTL (called LTL-SAT here) is even more difficult due to its temporal components and PSPACE-hard [93]. Classically, the problem is solved by translating the LTL formula into a Büchi automaton that is then checked for emptiness [86]. Alternatively, some direct approaches exist [65, 10]. However, while these algorithms can solve arbitrary LTL formulas, it is well possible that in a particular application, only formulas with certain common patterns need to be checked. Then, it could be quicker to identify only certain aspects of the formula instead of using a general solver. To this end, recent advancements in machine learning pose the question if a fully learned approach is able to determine LTL satisfiability, at least on some classes of formulas.

Deep learning for symbolic problems

During the last decade, breakthroughs in machine learning and deep learning in particular have made those approaches very popular and spread their use to a wide variety of fields. While the most well-known successes were made in the area of image recognition and machine translation (e.g. [59, 41, 95, 5, 99]), applications have since reached many different research areas. Naturally, a tempting question is if deep learning approaches can even be applied to problems that so far are solved by human-written algorithms. This is particularly interesting since it allows to observe whether learned models are able to mimic the high-level understanding of a problem's nature that allows humans to solve it. Experiments have been conducted on various algorithmic tasks, ranging from very simple

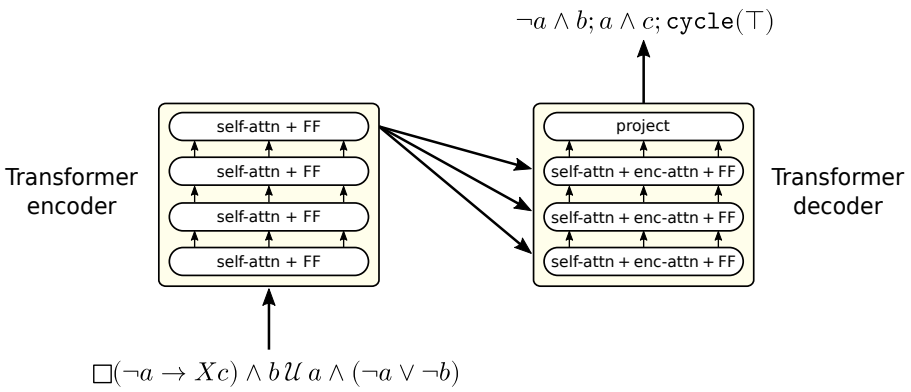


Figure 3: Setup in [39]: The neural Transformer architecture is used to generate satisfying traces from LTL formulas.

ones such as mirroring a sequence of letters, over integer multiplication to large-scale optimization problems [54, 87, 9]. While typical baseline architectures such as recurrent neural networks (RNNs) are able to learn some small and simple tasks, they are quickly overwhelmed on instances any human could solve with ease [87, 105]. Consequently, many authors came up with very specialized architectures featuring interesting theoretical properties such as StackRNN [53], the Differentiable Neural Computer [36], NeuralGPUs [54] and the Universal Transformer [20]. Still, general algorithmic problem solving remains a big challenge.

However, particularly in the area of mathematics and logics, there have been many demonstrations of deep learning approaches successfully solving typical problems: In [62], Transformers are used for symbolic integration of mathematical expressions which could even outperform their algorithmic baselines on some instances. [87] lets different sequence-based approaches carry out various mathematical reasoning tasks and [26] uses the parse tree of logical formulas to determine the validity of implications. As a significant milestone, the message-passing architecture of [92] is able to determine the satisfiability of propositional logic formulas in conjunctive normal form (CNF) with high accuracy. While not being competitive to existing SAT-solvers, this shows that a NP-complete problem can, to some extend, be solved by a learning approach without much assistance.

Most recently, the first application of deep learning to LTL was made by [39]. In the presented setting, Transformers [96], a deep learning architecture widely used in natural language processing, generate satisfying traces for a given input LTL formula. The setup is visualized in Figure 3. The model is trained using formula-trace pairs obtained from an algorithmic solver (`spot`, [21]). Training formulas are obtained by a synthetic generation process that randomly concatenates typical LTL formula patterns. The surprising finding is that the trained model constructs correct, satisfying traces for the input formula in a very high amount of cases and often, these differ from the trace found by the algorithmic solver. This suggests that the model indeed learned to solve the problem according to the LTL semantics and not just tries to mimic the exact syntactic output of the algorithmic solver. Given the high consistency of output formulas, it also shows that the Transformer architecture is well suited to symbolic processing. This thesis builds on [39] and also takes unsatisfiable LTL formulas into account. To this end, a learned LTL satisfiability checker is constructed in Section 4.

It is important to note that the approaches mentioned so far are only one way of tackling logic problems with deep learning: They do so in an end-to-end manner, that is the model is provided with the full problem as input and trained to produce the desired output. This leaves much freedom for how to find the solution, but accordingly poses a very hard

problem. The alternative is to use an algorithm and augment it with a learned component, which usually serves as a heuristic. Several works have implemented this approach, e.g. in theorem proving [98, 70, 78], QBF [63] and SMT solving [6] as well as SAT-solving [102, 91]. The obvious gain is that full correctness can be guaranteed and the size and complexity of the problem instance do not matter. The training process then becomes a reinforcement learning problem about guiding the algorithm and not solving the problem itself. However as [91] demonstrates, learning to solve the problem from scratch may lead to an architecture that can later be integrated into an algorithmic solver. This is also a desirable perspective for the approach in this thesis.

Data generation for symbolic problem domains

When training machine learning models, the design of the training dataset is of utmost importance. Specifically for classification tasks (as, for example, LTL satisfiability), it is an intricate challenge to ensure that the model actually learns to solve the intended task and not only exploits peculiarities in the dataset. For example, for formulas consisting of multiple conjunctions, their length alone can give away information about their chance of satisfiability (less likely the longer the formula). When generating random temporal formulas naively, it may also occur that unsatisfiability only ever stems from purely boolean contradictions, relieving the model of acquiring any temporal reasoning abilities. For standard propositional logic, there exists a practical solution to these problems as e.g. utilized by [92]: Random formulas in conjunctive normal form (CNF) exhibit a sharp phase transition in their probability of satisfiability depending on the ratio of variables and clauses [33]. With an appropriately designed incremental generation process, two formulas that are identical except for a single literal can be constructed, where one is satisfiable while the other one is not. Consequently, there is no easy way of finding out which instance was chosen except for solving the intended problem. However no similar procedure exists for LTL formulas, so the generation process has to be carefully designed to produce reliable training data. Additionally, the question arises what kind of formulas actually fit real-world use cases. [39] relied on formula patterns identified by literature to construct potentially realistic formulas. Section 3 expands this approach to also generate unsatisfiable and more diverse formulas and compares several generation procedures.

However, having to rely on engineered synthetic generation processes to hopefully produce realistic instances is unsatisfactory. This is a general problem for training deep learning models in symbolic domains where algorithms are usually employed, since naturally, the latter do not require any training data. Ideally, one would make use of instances collected from real-world applications, but existing collections, e.g. benchmark sets, are typically too small to train a deep model on them. This gives rise to the idea of using generative models, which can be trained on a small existing dataset and ideally capture the high-level appearance of typical instances. If training were to succeed, a much bigger dataset could be sampled from the model to finally train a neural solver on. This procedure is realizable especially on a logic domain, where it can be easily checked if an instance is syntactically valid by examining its parse tree. Invalid generated instances can simply be discarded. Additionally, labels can easily be computed by using existing algorithmic solvers. Contrary to many other domains, here it is not problematic to obtain correct labels, but rather realistic problems.
An intuitive candidate for the generative model are generative adversarial networks (GANs) due to their data efficiency [101]. However they are usually applied on continuous domains like images, since they require differentiating through generated instances. Additionally, they are not directly applicable to the variable-length sequences of logic problems. This

thesis aims to solve both problems by presenting a new kind of GAN architecture built with Transformer encoders in Section 5.
An additional, interesting opportunity of applying GANs in a logic domain is combining them directly with a neural solver. By propagating the gradient through the solver and generated samples back into the generator, the generator can be trained to produce instances that are hard to classify for the solver. Since an algorithmic solver is available, each such generated instance can be labeled and the solver trained on the real label. This potentially allows for a self-guided learning process which learns to both generate and solve increasingly harder problems. This approach is examined in Section 5.4.

Notes on this thesis

Implementation The thesis is accompanied by an implementation of all described approaches including data generation, classifiers for LTL-SAT and GANs for synthesizing LTL formulas. Its code is publicly available at `https://github.com/ju-kreber/Transformers-and-GANs-for-LTL-sat`. Every result presented in this thesis can be reproduced with the provided code. To this end, each table and figure is annotated with a reproduction notice found in Section 8 that details the necessary steps and parameters for the respective experiment.

Published content Some of the content in this thesis, in particular parts of Sections 3.2, 5.2 and 5.4, has already been published in the following paper:

[58] Jens U. Kreber and Christopher Hahn. *Generating Symbolic Reasoning Problems with Transformer GANs*. 2021. arXiv: `2110.10054 [cs.LG]`

2 Preliminaries

2.1 Propositional and linear-time temporal logic

2.1.1 Propositional logic

Before introducing temporal logic, standard propositional logic is quickly recapitulated formally. The main object of interest is a set of (atomic) propositions AP, whose elements are represented as characters $a, b, c, ...$ throughout this thesis. They can be combined by *boolean connectives* or *operators* such as "and" $\wedge$, "or" $\vee$ and "not" $\neg$ to form larger expressions. To obtain the semantics (denoted by $[\![\cdot]\!]$) of a given formula, which is a boolean value, each proposition is assigned to either "true" t or "false" f by a given assignment σ and the formula is recursively evaluated. Here, an assignment is a subset of AP assigning t to those propositions that are contained in it and f to all others.

$$\begin{aligned} [\![\alpha]\!]_\sigma &= t \text{ iff } \in \sigma \text{ for } \alpha \in AP \\ [\![\varphi \wedge \psi]\!]_\sigma &= t \text{ iff } [\![\varphi]\!]_\sigma = t \text{ and } [\![\psi]\!]_\sigma = t \\ [\![\neg\varphi]\!]_\sigma &= t \text{ iff } [\![\varphi]\!]_\sigma = f \end{aligned} \tag{1}$$

These two operators suffice for full expressiveness, but commonly additional connectives are used:

$$\begin{array}{lll} \varphi \vee \psi & := \neg(\neg\varphi \wedge \neg\psi) & \text{“or”} \\ \varphi \rightarrow \psi & := \neg\varphi \vee \psi & \text{“implies”} \\ \varphi \leftrightarrow \psi & := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) & \text{“equivalent”} \\ \varphi \oplus \psi & := \neg(\varphi \leftrightarrow \psi) & \text{“exclusive or”} \\ \top & := \alpha \vee \neg\alpha \text{ for some } \alpha \in AP & \text{“top”} \\ \bot & := \alpha \wedge \neg\alpha \text{ for some } \alpha \in AP & \text{“bot”} \end{array} \tag{2}$$

The atoms $\top$ and $\bot$ denote something that is always true or false, respectively.

An assignment σ is said to *satisfy* a formula φ, written $\sigma \vDash \varphi$ if and only if $[\![\varphi]\!]_\sigma = t$. If, for a given formula, some satisfying assignment exists, the formula is called *satisfiable*, otherwise it is *unsatisfiable*. Additionally, there are two special cases: If every possible assignment satisfies the formula, it is called a *tautology*. If it is instead false under every assignment, it is referred to as a *contradiction*.

2.1.2 Linear-time temporal logic

In [81], a logic that is referred to as linear-time temporal logic or LTL for short is introduced. There exist many different variants [24]; however for most use cases only the propositional fragment is relevant (more precisely abbreviated as PLTL). In this thesis, only this propositional fragment is referred to when mentioning LTL.

Where in standard propositional logic the objects of interest were single assignments, in temporal logic those are execution traces. A trace is a countably infinite sequence of assignments, meaning that given again a set of atomic propositions AP, a trace π maps each time step i to a subset of AP. Consequently, at each time step, any combination of propositions can hold or not. LTL extends standard propositional logic by temporal operators to allow the expression of relations across different time steps. The basic temporal operators are "next" $\bigcirc$ (also written $\mathcal{X}$) and "until" $\mathcal{U}$. In the following, let $\pi[i]$ denote

the i-th time step of trace π (starting at 0) and $\pi[i:]$ the trace starting at the i-th time step (inclusive).

$$\begin{aligned}
&\pi \vDash a && \text{iff } a \in \pi[0] \text{ for } a \in AP \\
&\pi \vDash \neg\varphi && \text{iff } \pi \nvDash \varphi \\
&\pi \vDash \varphi \wedge \psi && \text{iff } \pi \vDash \varphi \text{ and } \pi \vDash \psi \\
&\pi \vDash \bigcirc \varphi && \text{iff } \pi[1:] \vDash \varphi \\
&\pi \vDash \varphi \,\mathcal{U}\, \psi && \text{iff } \exists i.\ \pi[i:] \vDash \psi \text{ and } \forall j < i.\ \pi[j:] \vDash \varphi
\end{aligned} \tag{3}$$

Additionally, there are some commonly used derived operators:

$$\begin{aligned}
&\Diamond \varphi && := \top \,\mathcal{U}\, \varphi && \text{"eventually", also written } \mathcal{F} \\
&\Box \varphi && := \neg \Diamond \neg \varphi && \text{"globally", also written } \mathcal{G} \\
&\varphi \,\mathcal{R}\, \psi && := \neg(\neg\varphi \,\mathcal{U}\, \neg\psi) && \text{"release"} \\
&\varphi \,\mathcal{W}\, \psi && := (\varphi \,\mathcal{U}\, \psi) \vee \Box \varphi && \text{"weak until"}
\end{aligned} \tag{4}$$

For a more detailed reference of LTL and its variants consider [24]. In this thesis and its implementation, all binary operators are left-associative with the exception of the implication $\rightarrow$ and the precedence is the following (from strong to weak, separated by $\Rightarrow$):

$$\neg \bigcirc \Box \Diamond \quad \Rightarrow \quad \mathcal{U}\,\mathcal{W}\,\mathcal{R} \quad \Rightarrow \quad \wedge \quad \Rightarrow \quad \vee \quad \Rightarrow \quad \rightarrow \quad \Rightarrow \quad \leftrightarrow \oplus \tag{5}$$

This allows to omit e.g. the following parentheses:

$$\Box a \wedge b \,\mathcal{U}\, c \equiv (\Box a) \wedge (b \,\mathcal{U}\, c) \qquad\qquad \Diamond a \rightarrow \neg b \rightarrow c \equiv (\Diamond a) \rightarrow ((\neg b) \rightarrow c)$$

Consider the following example: Given the set $AP = \{a, b, c\}$, two valid traces are given on the left. The notation used here consists of a finite *prefix* followed by an infinitely often repeated *cycle*, denoted by the superscripted ω, to represent the infinite sequence. The table on the right marks which exemplary LTL formulas hold on which trace.

1. $\{a\}, \{a, b\}, (\{a\})^\omega$
2. $\{a\}, \{c\}, \{a, c\}\ (\{b\}, \{a\})^\omega$

formula	1	2	formula	1	2
$\Diamond b$	✓	✓	$\Box \Diamond b$	✗	✓
$\Box \neg c$	✓	✗	$a \,\mathcal{U}\, c$	✗	✓
$\Diamond \Box \neg c$	✓	✓	$a \,\mathcal{W}\, c$	✓	✓
$\bigcirc b$	✓	✗	$\Box(a \leftrightarrow X \neg a)$	✗	✓

Intuitively, the $\Diamond$ operator expresses that something should happen *eventually*, no matter when or how often. With $\Box$ one can express something that should always or never hold. Operators can be combined to form more interesting statements: $\Box\Diamond$ requires a subformula to hold *infinitely often* (like b in the examples), whereas the other way round $\Diamond\Box$ relaxes the global requirement to start holding at some later point in time (like the $\neg c$ in example trace 2). The difference between $a \,\mathcal{U}\, c$ and $a \,\mathcal{W}\, c$ is also exemplified with the shown traces: While $\mathcal{U}$ requires both its right-hand side to happen eventually and its left-hand side to hold up to that point, for $\mathcal{W}$ it is sufficient for the left-hand side to hold indefinitely as an alternative. As final example, consider the last example formula $\Box(a \leftrightarrow X\neg a)$ which gives a hint on how operators can be combined to express more sophisticated behavior: It requires the proposition a to exactly hold alternately every second time step.

2.2 Deep learning in the sequence domain

2.2.1 Token sequence domain

Token sequences are an important application domain of deep learning, with natural language text falling into this category. Typically, a vocabulary V of words is defined and each sentence (or document) is represented as a sequence of indices into that vocabulary. This tokenization process does not necessarily only occur at the word level; sub-word tokenization or even character-wise tokenization is also possible. Conveniently, for symbolic logic problems, tokens are directly available. The vocabulary used in this thesis has 24 entries including 10 atomic propositions. Neural architectures typically work on continuous vector representations. To feed discrete tokens into such an architecture, a so-called *embedding* is either provided or learned (e.g. [74]). It can be simply though of as a $|V| \times d$ matrix that maps the indices of different vocabulary tokens each to some continuous vector of fixed dimension d. Figure 4 visualizes the embedding process for a logic vocabulary as used in this thesis. While the step of constructing *one-hot* vectors is not technically necessary (embedding matrices can be indexed directly), they allow to express the embedding process more conveniently as matrix multiplication and are important later in section 5.2. Formally, given a token described by its index in the vocabulary $t \in [1, |V|]$, the corresponding one-hot vector has dimensionality $|V|$ and contains all zeros except for dimension t, where it contains a 1. A vector $\mathbf{p} \in [0, 1]^{|V|}$ subject to $\sum_{i=1}^{|V|} p_i = 1$ therefore represents a probability distribution over the existing tokens, where a one-hot vector is the extreme case of a determined token. This probability distribution occurs when tokens are output from a neural architecture as depicted in Figure 5. After the final neural processing layer, a logit with dimensionality $|V|$ is output. This logit is subsequently normalized, usually by applying a *softmax* and thereby represents the output token probability distribution. During supervised training, this distribution is used directly for loss computation, where usually a crossentropy against the desired token's one-hot representation is applied (in practice, it is implemented differently). In order to obtain the most probable token, an *argmax* can simply be taken over the distribution and the result looked up in the vocabulary.

Secondly, the variable length of sequences requires an architecture designed to work on this type of input. Historically, recurrent architectures like recurrent neural networks (RNNs, [73]), were commonly employed for this (e.g. early work of [75]). The sequence is passed into the network one sequence element after the other and its internal state is incrementally updated. Their design was refined over the years, most importantly by employing gating (e.g. LSTMs, [43]), an encoder-decoder split [16, 94] and the concept of *attention* [5]. However in recent years, the *Transformer* architecture introduced in the following section became a very popular choice for sequence tasks.

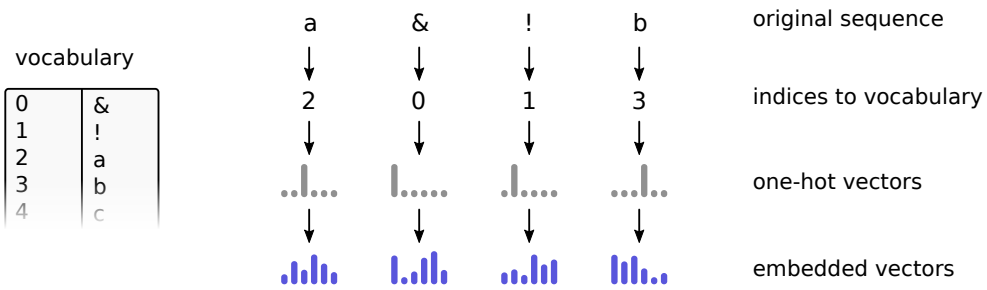


Figure 4: Embedding process for token sequences

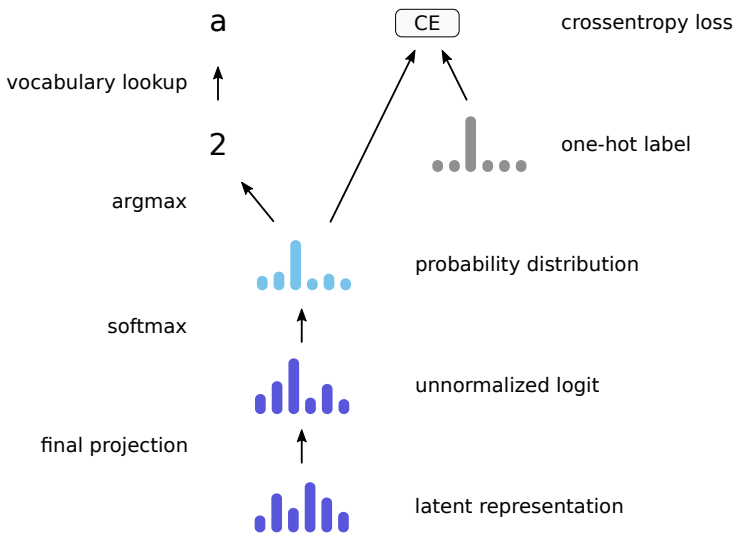


Figure 5: Output processing for a single token

2.2.2 The Transformer

The Transformer architecture introduced by [96] relies heavily on *self-attention* [79, 69] between all tokens of the sequence. In contrast to RNNs and their derivatives, the whole input sequence is processed simultaneously, with multiple steps of attention-based processing executed one after the other in a multi-layer, feed-forward way. Besides offering unrestricted global information flow (e.g. from first to last position in a single step), this allows for much faster training since the sequence does not have to be unrolled as in RNN-based approaches.

Input data processing Figure 6 depicts the fundamental architecture of a Transformer encoder. The input tokens are mapped by a learned embedding into a continuous vector, similarly as in other neural token sequence approaches. The number of dimensions d_{emb} of this vector is an important hyperparameter and used throughout the full Transformer architecture. To these embedding vectors, a positional encoding is added. The originally proposed encoding from [96] is based on sine functions, thereby encoding absolute positions in the input sequence. With increasing position, a sine wave of higher frequency is applied, where the argument of the sine is the respective dimension in the embedding vector. Notably, this positional encoding is the only piece of information the Transformer has about the sequence structure. In RNNs, the order of elements is implicitly encoded by sequential computations; in the Transformer, all positions are invariant (except for said positional encoding).

Data processing is carried out in multiple consecutive layers. Each layer firstly manipulates all sequence elements via an attention mechanism and then applies a two-layer feed-forward neural network individually to each element. Importantly, the results of both stages (attention and feed-forward NN) are *added* to the respective previous embeddings; thereby the Transformer employs a type of *residual connection*: Forwarding the sum $x + f(x)$ of original value x and the result value $f(x)$ instead of only the result allows the gradient to flow backwards much more freely, thereby increasing the performance of multi-layered architectures [40].

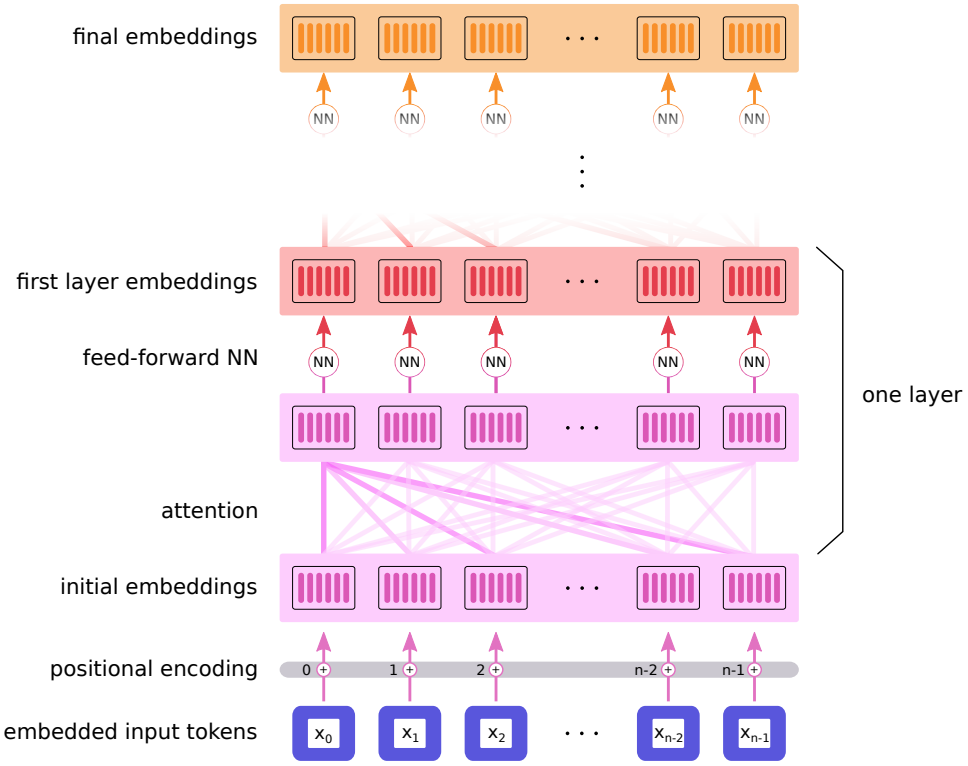


Figure 6: Transformer encoder architecture

Attention In the attention stage of each layer, so-called *scaled dot-product attention* [96] is employed. Basically, it uses the dot-product between two elements as a similarity measure. Notably, the dot-product is not calculated directly on the embeddings; they are firstly transformed by a learned linear mapping. Additionally, the comparison is not symmetric: The mappings differ depending on whether the attention is calculated *from* the current element or *towards* it. Element i "queries" information from the "keys" of all other elements. The attention score α_{ij} then describes how relevant position j is for the information need of i. Consequently, for this particular attention score, a latent embedding h_i is transformed by a query mapping A_Q and h_j by a key mapping A_K. Formally, α_{ij} is computed by

$$\alpha_{ij} = \frac{\exp(A_Q^T h_i \cdot A_K^T h_j)}{\sqrt{d_{\text{emb}}} \sum_{k=1}^{\ell} \exp(A_Q^T h_i \cdot A_K^T h_k)} \tag{6}$$

where ℓ represents the total sequence length. The retrieved information however is not obtained from the keys, but a distinct "value" representation obtained by applying A_V of the respective elements. This value representation is then weighted by the respective attention value and summed over the whole sequence. Consequently, a context vector c_i, the result of the self-attention process for an individual element i, is calculated as

$$c_i = \sum_{k=1}^{\ell} \alpha_{ik} A_V^T h_k \,. \tag{7}$$

Actually, the calculations described above are carried out on multiple, smaller subspaces in parallel (which are called *heads*, therefore this process is termed *multi-head attention*); consider [96] for details.

The Transformer was originally designed for sequence-to-sequence tasks (like LTL trace generation in [39]) and therefore has an encoder and decoder part. A Transformer decoder differs from the encoder in that instead of one attention step per layer, it first performs self-attention (over the output sequence) and then encoder-decoder attention, where queries come from the decoder (output sequence), but keys and values from the final layer of the encoder. For this thesis however, except for a small dataset comparison in Section3.3.3, no Transformer decoder is used.

2.2.3 Universal Transformer

While the Transformer is designed to efficiently process sequences, it only provides a very small number of interdependent computation steps, namely one in each layer. While this is arguably beneficial for training in many scenarios, specifically on algorithmic tasks, an architecture that performs some kind of iterative computation would be desirable. This is due to the fact that problems with non-constant complexity require more computation steps as they are scaled up in size. An architecture that incorporates this feature into the Transformer is the *Universal Transformer* [20]. Contrary to the multiple feed-forward layer of the standard Transformer encoder, it applies a single layer recurrently to the whole sequence. The number of iterations can either be specified beforehand or dynamically determined. For this thesis, only the encoder part of the Universal Transformer is relevant and applied in Section 4.2.

2.3 Generative adversarial networks

2.3.1 Overview

The goal of generative models in general is to fit a data distribution p_r as close as possible while being provided with samples from that distribution, $x \sim p_r$. The model itself thereby represents a generative distribution p_g, which should ideally converge to p_r. Naturally, one would like to be able to conveniently draw samples from p_g, which are potentially very high-dimensional, such as images. This is usually solved by training a generative model to map some well known, low-dimensional prior distribution p_z to the actual generative distribution p_g. This way, one can sample $z \sim p_z$, input it to the model G and obtain $g = G(z)$ which is equivalent to $g \sim p_g$. Typically, Gaussian and uniform distributions are used for p_z.
A natural choice for such a model are *autoencoders*, or more concretely the popular VAE architecture (variational autoencoder [56]). They are trained to represent data instances in a low-dimensional, probabilistic latent space and reconstruct it from there. In addition to the reconstruction objective, a prior distribution is enforced on the latent space. Ideally, the model then learns a bijective mapping between the underlying data space and the latent space. During generation, one can simply sample a point in the latent space (using the prior distribution p_z) and the decoder transforms it into a realistically-looking sample.

Generative adversarial networks (GANs) [34] on the other hand use an entirely different approach. As depicted in Figure 7, they consist of two networks, a *generator* and a *discriminator* with opposing goals, hence the name "adversarial". The generator behaves much like the decoder in a autoencoder: It is supplied with a random latent space sample z and builds realistic samples from it. The discriminator is given both generated (fake) instances from the generator and real instances from the training data distribution. It is trained to correctly classify the origin of an instance into real or fake. By this, it learns to effectively identify the samples created by the generator. Now, as the discriminator

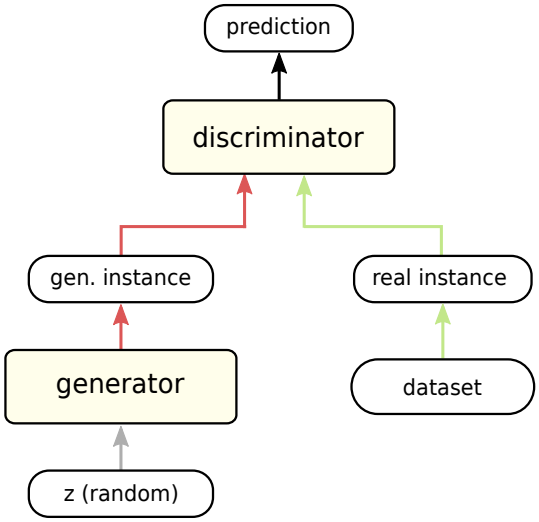


Figure 7: GAN architecture

is fully differentiable, one can propagate the gradient of its classification decision back through the generated instances to the generator's parameters. The generator's goal is to fool the discriminator, that is, producing instances the discriminator can not tell apart from real ones. By backpropagation of the gradient, the discriminator (unwillingly, in a way) teaches the generator how to improve. Ideally, this results in generated instances that are indistinguishable from real ones, in which case the discriminator can only guess randomly and the generative distribution p_g coincides with the real data distribution p_r. Importantly, the objectives of both parts of the GAN conflict and highly depend on each other: If the generator produces only very poor instances, the discriminator does not need to learn meaningful ways to identify them. Similarly, if the discriminator is overwhelmed by telling real and fake samples apart, the generator has no way to improve. This leads to potentially unstable training dynamics and convergence is not guaranteed. In the following, Section 2.3.2 introduces the formal objectives of the GAN formulation and Section 2.3.3 covers the practical training.

2.3.2 Formulation

The original [34] GAN objective formulation is the following:

$$\min_G \max_D V(G,D) \qquad \text{with} \qquad V(G,D) = \mathop{\mathbb{E}}_{x\sim p_r}[\log D(x)] + \mathop{\mathbb{E}}_{z\sim p_z}[\log(1-D(G(z)))] \quad (8)$$

with the generator G, discriminator D, real data distribution p_r and prior distribution p_z. For the generator, this is essentially a typical classification loss but on two different probability distributions. Since sampling $z \sim p_z$ and transforming it by $G(z)$ is equivalent to sampling from the implicitly defined probability distribution p_g, the discriminator objective for a fixed generator can be rewritten as

$$\max_D \mathop{\mathbb{E}}_{x\sim p_r}[\log D(x)] + \mathop{\mathbb{E}}_{x\sim p_g}[\log(1-D(x))]\ . \quad (9)$$

This maximum is attained for a discriminator that equals

$$D^*_G(x) = \frac{p_r(x)}{p_r(x)+p_g(x)} \quad (10)$$

[34], still for a fixed generator. Plugging this optimal discriminator back into the original objective function yields

$$\begin{aligned} V(G, D_G^*) &= \mathop{\mathbb{E}}_{x \sim p_r} \left[\log \frac{p_r}{p_r + p_g} \right] + \mathop{\mathbb{E}}_{x \sim p_g} \left[\log \frac{p_g}{p_r + p_g} \right] \\ &= \mathrm{KL}\left(p_r \,\|\, p_r + p_g\right) + \mathrm{KL}\left(p_g \,\|\, p_r + p_g\right) \\ &= 2\,\mathrm{JS}\left(p_r \,\|\, p_g\right) - 2\log 2 \end{aligned} \tag{11}$$

[34], which ultimately results in the *Jensen-Shannon* (JS) divergence between the real and generated distributions. The JS divergence is a characteristic property of GANs and sets them apart from most other architectures that use the *Kullback-Leibler* (KL) divergence. Note that for optimization, using the KL divergence and crossentropy are equivalent and both correspond to a maximum-likelihood estimation. In contrast to KL, the JS divergence is symmetric by comparing both distributions to an average distribution $\frac{1}{2}(p_r + p_g)$.
Nonetheless, this means that from a theoretic view, the GAN discriminator is trained to approximate the JS divergence between the real and current generative distribution. The generator then tries to minimize this approximation, thereby (implicitly) pushing p_g towards p_r. Note that this rests on the assumption of a discriminator trained to optimality, which is not the case in typical training scenarios. However, a GAN training loop usually has multiple discriminator iterations for each generator iteration to provide a good and up-to-date JS divergence approximation.

2.3.3 Training

In practice, training GANs is challenging. Training dynamics are often unstable and rarely converge to optimality (minimum possible divergence). The observed generated sample quality can improve and degrade indefinitely during training. A major problem is that the generator ceases to improve when the discriminator performs very well, which is counterintuitive to the previous derivation which assumes an optimal discriminator. The issue has been discussed in several works (including [3, 48, 4]) and different remedies have been proposed, of which some are also used in this thesis.

The standard GAN loss functions read

$$\begin{aligned} \mathcal{L}_D &= -\mathop{\mathbb{E}}_{x \sim p_r} \left[\log C(x)\right] - \mathop{\mathbb{E}}_{z \sim p_z} \left[\log(1 - D(G(z)))\right] , \\ \mathcal{L}_G &= \mathop{\mathbb{E}}_{z \sim p_z} \left[\log(1 - D(G(z)))\right] . \end{aligned} \tag{12}$$

In [34], the authors argue that a good discriminator will yield low estimates for $D(G(z))$, thereby saturating the logarithm and yielding gradients close to zero. They therefore propose the use of the alternative loss function

$$\mathcal{L}_G = -\mathop{\mathbb{E}}_{z \sim p_z} \left[\log D(G(z))\right] \tag{13}$$

for the generator, which behaves relatively similar to the original one, but breaks the underlying theoretic properties.

In [4], discriminator saturation is avoided by using a different underlying measure that is approximated. Instead of the JS divergence, the authors propose using the Wasserstein

distance which has better properties as objective for the generator. The Wasserstein-1 distance is defined as

$$W(p_r, p_g) = \inf_{\gamma \in \Pi(p_r, p_g)} \mathop{\mathbb{E}}_{(x,y)\sim\gamma} [\|x - y\|] \tag{14}$$

where $\Pi(p_r, p_g)$ is the set of all possible joint distributions with p_r and p_g as marginal distributions [4]. Intuitively, if the (two-dimensional) distributions p_r and p_g are imagined as piles of soil, the Wasserstein-1 distance yields the minimal amount of earth that has to be moved around in order to make both piles look identical. Hence it is also called "earth-mover" distance.
The authors of [4] find that the term

$$\max_{f \in \mathcal{F}} \mathop{\mathbb{E}}_{x\sim p_r} [f(x)] - \mathop{\mathbb{E}}_{x\sim p_g} [f(x)] \tag{15}$$

yields K times the Wasserstein distance between p_r and p_g where $\mathcal{F}$ is the set of all K-Lipschitz functions $\mathcal{X} \to \mathbb{R}$. Consequently, if a neural network can be ensured to be K-Lipschitz, it can be trained to approximate this distance, similar to the discriminator in the original GAN architecture. This is achieved in [4] by clipping the network's weights to a small numerical range. Importantly, the objective function 15 does not ask for classification as the original GAN one (Equation 8, $D : \mathcal{X} \to [0, 1]$), but allows any real number as output. Because of this, the authors refrain from calling this network a discriminator and use the more appropriate term *critic*. The loss for the critic C then reads

$$\mathcal{L}_C = - \mathop{\mathbb{E}}_{x\sim p_r} [C(x)] + \mathop{\mathbb{E}}_{z\sim p_z} [C(G(z))] \tag{16}$$

(with clipped weights) and for the generator

$$\mathcal{L}_G = - \mathop{\mathbb{E}}_{z\sim p_z} [C(G(z))] \tag{17}$$

without any further constraints. This formulation is called Wasserstein GAN or WGAN for short. In this thesis, both standard GANs and WGANs are used and compared. When not referring to particular differences between them, the term *GAN* is used as a hypernym for both variants. Since in this work Wasserstein GANs play a slightly more important role and the difference between a GAN discriminator and WGAN critic is marginal implementation-wise, the term *critic* is used as hypernym for both.

In [37], the authors notice negative effects of the weight clipping in the standard WGAN formulation and propose an alternative way of ensuring Lipschitz-continuity of the critic: Penalizing a deviation from a gradient of 1 for data points in between real and generated ones. Generally, if the gradient's norm $\|\nabla_x f(x)\| \leq 1$ almost everywhere, then f is 1-Lipschitz [37]. Since this is intractable to enforce for the whole data space, [37] argues that it is sufficient to do this on straight lines between randomly sampled instances from the real and generated data distributions. Concretely, for a real and a fake sample $x_r \sim p_r, x_g \sim p_g$, a value $\tau \sim U[0; 1]$ is sampled uniformly from the interval $[0, 1]$. With this, an intermediate data point $\tilde{x} = \tau\, x_r + (1 - \tau)\, x_g$ is calculated. For this point, the gradient of the critic with respect to the point itself is evaluated and its norm is penalized towards 1 via mean-squared error. The gradient penalty loss term reads

$$\mathcal{L}_{GP} = \lambda_{GP} \mathop{\mathbb{E}}_{\tilde{x}\sim p_{\tilde{x}}} \left(\|\nabla_{\tilde{x}} C(\tilde{x})\|_2 - 1\right)^2 \tag{18}$$

where $p_{\tilde{x}}$ denotes the implicitly defined distribution for obtaining an intermediate sample as described above. This algorithm is called WGAN-GP.

Algorithm 1 shows the training in the WGAN-GP variant [37] as used in this thesis. θ_C and θ_G refer to the parameters of the critic and generator, respectively and bs denotes the batch size. n_c (inner optimization steps), λ_{GP} and bs are hyperparameters. The classic GAN algorithm is identical except for the used losses (and intermediate data points).

Algorithm 1 WGAN-GP training algorithm as used in this thesis

for *num_steps* **do**
  obtain a batch of real data $\mathbf{x}_r$ from the dataset
  for n_c **do**
    sample a batch of random latents $\mathbf{z} \sim p_z$
    $\mathbf{x}_g := G(\mathbf{z})$
    sample a batch of random intermediate coefficients $\tau \sim U[0; 1]$
    $\tilde{x}^i := \tau^i \, x_r^i + (1 - \tau^i) \, x_g^i$
    $d\theta_C := \nabla_{\theta_C} \left[-\frac{1}{bs} \sum_{i=1}^{bs} C(x_r^i) + \frac{1}{bs} \sum_{i=1}^{bs} C(x_g^i) + \lambda_{GP} \frac{1}{bs} \sum_{i=1}^{bs} \left(\|\nabla_{\tilde{x}^i} C(\tilde{x}^i)\|_2 - 1 \right)^2 \right]$
    $\theta_C := Adam(\theta_C, d\theta_C)$
  end for
  sample a batch of random latents $\mathbf{z} \sim p_z$
  $d\theta_G := \nabla_{\theta_G} \left[-\frac{1}{bs} \sum_{i=1}^{bs} C(G(z^i)) \right]$
  $\theta_G := Adam(\theta_G, d\theta_G)$
end for

3 Data generation

To build a proper dataset for supervised training of LTL-SAT, several thousand example instances are required. The main training dataset used by [39] for example contains 1.3 M instances. Each example consists of the input (a LTL formula) and the desired output or label (boolean value for satisfiability). The satisfiability of a formula can simply be determined by using an existing LTL-SAT tool, such as `spot` [21] or `aalta` [66], which are both directly supported in the implementation. The intricate part however is generating meaningful LTL formulas that, at least to some extend, resemble real-world use cases and for which determining satisfiability is nontrivial.

3.1 Existing approaches

3.1.1 Simple random formulas

Arguably the most straightforward way of generating a formula of given size *size* and predetermined token probabilities *probs* is the following pseudocode:

Algorithm 2 Simple fixed-size formula generator

```
1:  function RANDOM_FORMULA(size, probs)
2:      if size = 1 then
3:          res_probs := RESTRICT_TO_LEAFS(probs)
4:      else if size == 2 then
5:          res_probs := RESTRICT_TO_UNARY(probs)
6:      else
7:          res_probs := RESTRICT_NO_LEAFS(probs)
8:      end if
9:      size := size − 1
10:     token, arity := SAMPLE(res_probs)
11:     if arity = 2 then
12:         left_size := RANDINT(1, size − 1)
13:         left_child := RANDOM_FORMULA(left_size, probs)
14:         right_child := RANDOM_FORMULA(size − left_size, probs)
15:         return BINARYNODE(token, left_child, right_child )
16:     else if arity = 1 then
17:         child := RANDOM_FORMULA(size, probs)
18:         return UNARYNODE(token, child)
19:     else
20:         return LEAFNODE(token)
21:     end if
22: end function
```

The various RESTRICT_ functions set probabilities of tokens with unfit arity to 0 and renormalize the total probability to 1. This ensures that always a formula of intended size is created. The random formula generator in `spot` works in a similar way. In [39], the secondary dataset `LTLRandom35` is created like this. While this simple approach does create syntactically random formulas, they are not necessarily useful for training dataset generation. For example, formulas obtained this way are almost always satisfiable, which is analyzed in detail in the upcoming Section 3.3, and encode uninteresting problems.

3.1.2 Concatenating prototype patterns

Generally, one can simply concatenate multiple sub-formulas with the `and` connective to increase the probability of rendering the whole formula unsatisfiable. For example, [92] does this in the conjunctive normal form of propositional logic until the whole formula becomes unsatisfiable and through clever design extracts a pair of satisfiable and unsatisfiable formulas that only differ in the sign of a single literal. Also for LTL, concatenating sub-formulas is a reasonable approach and also reflects the way real-world specifications are built from multiple parts. [39] uses a collection of 55 formulas called "dac patterns" that are instantiated with random variables and concatenated until a termination criterion like maximal total length or maximum number of concatenations is met. These formulas are based on [23], which introduced a systematic classification of typical specification patterns occurring in practice. A dataset similar to the one from [39] is evaluated in Table 1. There are two aspects of this generation procedure that could be improved upon: Firstly, the exact same 55 patterns are used for generation of all formulas. Even though their variables change, their structure in terms of logical operations remains the same. Secondly, concatenation formulas produced in this way are almost exclusively satisfiable as apparent from Table 1, which makes them unsuited for an LTL-SAT classification task. This also indicates that their diversity in terms of possible solutions could be limited. In the next subsection, this generation process is therefore refined.

3.2 New generation process: Rich pattern concatenation

The idea of concatenating specification patterns used in [39] is picked up and improved on in the following. Similarly, the pattern system introduced by [22] is used as a basis. The patterns identified in that work can be categorized into both a type and a scope. Types denote an intuitive meaning, such as "absence" (something never happens) or "response" (something happens after something else has happened). The scope limits the patterns to temporal constraints, such that it only becomes active after some event or between two distinct events, for example. For example, the "global absence of P" simply reads $\square \neg P$. However "Q responds to P before R " is more complex: $\diamondsuit R \rightarrow (P \rightarrow (\neg R \,\mathcal{U}\, (S \wedge \neg R)))\,\mathcal{U}\, R$. While this system may be well-organized and reflect real-world applications, it is not directly suited for training data generation in the context of this thesis. The main problem with patterns in general is that they form a net of dependencies between several events, but most often some "trivial" solution avoiding most of these dependencies exists, rendering large parts of the formula irrelevant. For example all patterns that are not in global scope can be "deactivated" by never making their scoping propositions true. In the context of model checking this is not a problem since the checker must consider all traces of the system, but for satisfiability, a single trace suffices.

This problem is tackled by a "richer" pattern-based generation scheme. Firstly, instead of using single propositions for the placeholders in the pattern prototypes, a whole subformula is generated. This may include additional temporal operators, but only with very low probability. Instead, creating small propositional formulas like $(a \wedge b) \vee \neg c$ is much more likely. The incorporation of multiple atomic propositions ties the dependencies between multiple patterns more closer together. Secondly, instead of only concatenating patterns together, with a relatively high probability (45%) a so-called "grounding" term is added. Its purpose is to increase the odds of actually triggering an event that "activates" another pattern somewhere in the whole formula. Groundings can be simple time-shifted propositional facts like $\bigcirc\bigcirc a \wedge \neg b$ or eventualities like $\square\diamondsuit c$. To amplify this intended ef-

fect of activating other patterns, groundings have a higher probability of `and`s while scope events in patterns have a higher probability of `or`s. Lastly, while the concatenation progresses, newly generated atomic propositions are biased towards those already existing in the formula to increase the odds of an interplay. For more details of the generation process, the interested reader may consider the code in `data_generation/spec_patterns.py`.

Main dataset A dataset of 3.4 M instances generated by the rich pattern scheme with groundings forms the basis of the main dataset used in this thesis. Formulas are restricted in size from 1 to 100. After removing duplicate instances, it is referred to as `RPC100-raw` in the following and can be reproduced by following Rep. 1. Its size distribution is plotted in Figure 8. As depicted in Figure 9, its proportion of satisfiable and unsatisfiable instances is still very unbalanced and depends on formula size. Consequently, it is filtered to have identical proportions for every size (depicted in Figure 11), following Rep. 2. The resulting length distribution is shown in Figure 10. This set is called `RPC100` and used extensively throughout this thesis. It is split into a training set of 1.4 M and a validation set of 0.2 M instances.

3.3 Evaluation

3.3.1 Comparison of generation methods

Table 1 exemplary compares the three major generation schemes discussed above. In order to build a dataset for LTL-SAT, the most important value is the fraction of unsatisfiable examples. As apparent from the table, the simple generation process with equal operator probabilities produces very few unsatisfiable examples and even a severe adjustment of the probabilities (with $\wedge$ being 15 times more likely than $\vee$, besides others) increases the fraction only to some extend. Surprisingly, the dac pattern scheme produces almost no unsatisfiable instances at all. The rich pattern scheme however already produces a significant unsatisfiable fraction when only patterns but no groundings are concatenated (see Section 3.2). With additional grounding, the fraction is again increased significantly. While 26% unsatisfiable instances still falls short of an ideal 50%, minor filtering can easily overcome this gap.

Table 1 contains several additional values that could potentially serve as indicators for formula interestingness. The average number of time steps in the solution trace (including prefix and cycle; only relevant for satisfiable instances) gives a hint on the temporal complexity of the solution. A purely propositional problem would be solved by a single timestep, however the solver `spot` in such cases usually outputs the propositional solution in the prefix and $\top$ in the cycle. Therefore, as a rule of thumb, a trace should have at least three steps to be considered temporally interesting. It should also be mentioned that

method	properties	unsat frac	avg steps	avg aps per step	avg solve time
simple	equal op dist.	2.0%	2.34	1.15	136.7
simple	many ands	11.7%	2.76	1.95	95.5
dac concat		0.1%	1.91	2.87	130.4
rich concat	patterns only	12.2%	2.19	2.22	31.5
rich concat	patterns+grounding	26.0%	3.67	2.05	35.05

Table 1: Data generation process comparison. All produce formulas of size 1 up to 100. Results can be reproduced by following Rep. 3

`spot` does not prefer the shortest solution or simplifies traces, so this metric should be considered more of a soft guideline. However, it can be seen that most schemes produce relatively similar values on the low end of temporal interestingness, with only the rich pattern scheme with grounding yielding a significant increase. The table additionally shows the average number of atomic propositions occurring in a single trace step on average. It is arguably low at around 2 for most generation schemes, which means that the solutions to satisfiable formulas are still not especially tightly constrained. Lastly, the solving time of the reference `spot` solver is also included for reference. While this number may seem as a simple universal reference for formula hardness, it can be misleading. `spot` is not primarily designed to solve the LTL-SAT problem, but constructs Büchi automata from LTL formulas which can then be checked for emptiness to determine satisfiability [21]. This means that the automata construction can take a long time. Interestingly, in the rich pattern scheme with grounding, unsatisfiable formulas only took 3.4 ms on average while generally being longer than satisfiable formulas, for which the computation took 46.2 ms on average. This heavily indicates that the solving time of `spot` should not be used as an indicator for temporal interestingness.

3.3.2 Temporal relaxation

While several metrics in Table 1 describe the solution trace to satisfiable formulas, there is no straightforward way to determine how easy it is to check the unsatisfiability of a formula. Certificates for unsatisfiability that could be analyzed are temporal resolution proofs, but these are out-of-scope for this thesis. However it would be reassuring to know that solving the generated examples indeed requires some sort of temporal reasoning. It could, for example, be well imaginable that all unsatisfiable cases come from purely propositional, non-temporal contradictions such as the left example in the following:

$$\underset{\text{temporally trivial unsat}}{(a \vee b) \wedge \neg b \wedge \Box \neg a} \qquad \underset{\text{temporally non-trivial unsat}}{\neg a \,\mathcal{U}\, b \wedge \Box \neg b \wedge \Diamond a} \tag{19}$$

The right example would require actual temporal reasoning. To identify formulas that do not need the consideration of temporal relations to determine their unsatisfiability, a relaxation from LTL to propositional logic is employed. Let the relaxation be called Rel. It must suffice the following criterion for any LTL formula φ:

$$\forall \pi . \pi \vDash \varphi \rightarrow \pi[0] \vDash Rel(\varphi) \tag{20}$$

In other words, for each solution trace π to φ, its first position $\pi[0]$ is a solution to the propositional relaxation $Rel(\varphi)$. In the context of satisfiability, actually the contraposition of this is used:

$$\forall \pi . \pi[0] \nvDash Rel(\varphi) \rightarrow \pi \nvDash \varphi \,, \tag{21}$$

which means that if some assignment does not satisfy the relaxation, no trace with this assignment at first position can ever satisfy the original formula. As a consequence, if the relaxation is found to be unsatisfiable, also the original formula must be. The relaxation

is defined as follows:

$$\begin{aligned}
Rel(\varphi * \psi) &= Rel(\varphi) * Rel(\psi) \quad \text{for } * \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\} \\
Rel(\varphi * \psi) &= Rel(\varphi) \vee Rel(\psi) \quad \text{for } * \in \{\mathcal{U}, \mathcal{W}\} \\
Rel(\varphi \mathcal{R} \psi) &= Rel(\psi) \\
Rel(\bigcirc \varphi) &= \top \\
Rel(\Box \varphi) &= \varphi \\
Rel(\Diamond \varphi) &= \top \\
Rel(\alpha) &= \alpha \quad \text{for } \alpha \in AP \cup \{\top, \bot\} \\
Rel(\neg \alpha) &= \neg \alpha \quad \text{for } \alpha \in AP \cup \{\top, \bot\}
\end{aligned} \tag{22}$$

Notably, negation is only allowed at the level of atoms. Each LTL formula can be rewritten in a negation normal form (NNF), where only operators $\wedge, \vee, U, R, X$ occur anywhere and negations only before atoms. Consequently, the relaxation can be applied to each LTL formula by first bringing it to NNF. By not considering anything after $\bigcirc$ and $\Diamond$, the relaxation is relatively loose, but simple and easy to apply. It will not catch certain types of temporally trivial unsatisfiable formulas like $\Box a \wedge \Diamond \neg a$ or $\bigcirc a \wedge \Diamond \neg a$ (both relaxed satisfiable), but many others like the left example in Equation 19 (relaxed unsatisfiable). Temporally non-trivial unsatisfiable formulas like the right example in Equation 19 are relaxed satisfiable. Figure 9 shows the distribution of satisfiable and unsatisfiable examples in `RPC100-raw` per length with the satisfiability of the relaxation included. As it can be seen, formulas are more likely to be unsatisfiable with increasing size. This indicates that filtering to balance classes should be done per length value to even this out. Additionally, there is a significant fraction of unsatisfiable formulas that do not require temporal reasoning as determined by the propositional relaxation, but this is only a minority. This means that chances are good that actual temporal reasoning is required to solve most of the instances. Figure 11 shows the resulting proportions of `RPC100`.

3.3.3 Evaluation of `RPC100` for LTL trace generation

As comparison with the dataset used in [39] ("dac concat"), `RPC-100` is subsequently tested for the trace generation problem studied in [39] with an identical architecture. Since trace generation is only possible for satisfiable formulas, for this experiment, unsatisfiable ones are dropped. Firstly, the "dac concat" dataset from [39] is used to train a model for LTL trace generation (Rep. 6). Figure 12 shows the results when using a standard positional encoding and Figure 13 for a tree positional encoding. While the first setup produces a significant amount of invalid and incorrect traces, with the tree PE, a very high accuracy is achieved for each formula size. With training and evaluating on `RPC100`, performance is much worse, as depicted in Figure 14 for standard PE and Figure 15 for tree PE. For a large range of formula sizes, there is a high percentage of incorrect traces and with increasing size, more and more outputs become invalid. Interestingly, the tree PE variant performs even slightly worse. This suggests that as intended, instances in `RPC100` encode challenging temporal problems.

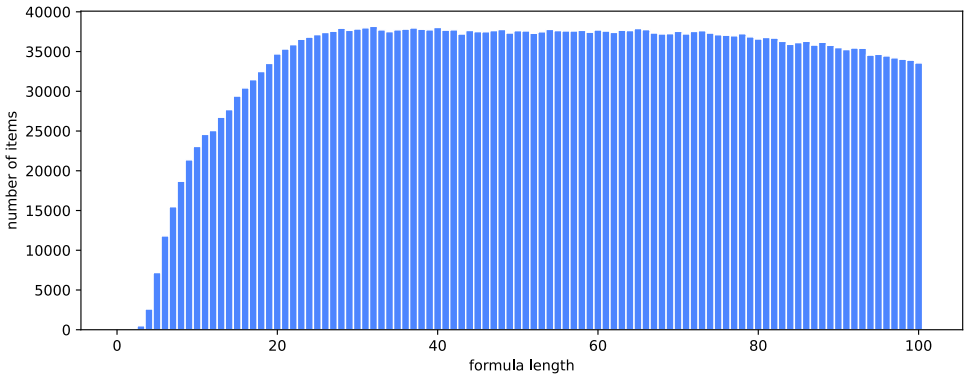


Figure 8: Formula length distribution of `RPC100-raw` (Rep. 4)

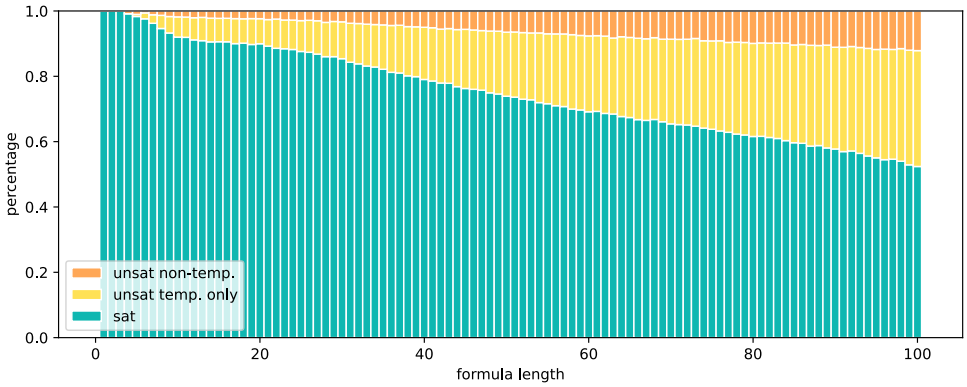


Figure 9: Class proportions by formula length of `RPC100-raw` (Rep. 4)

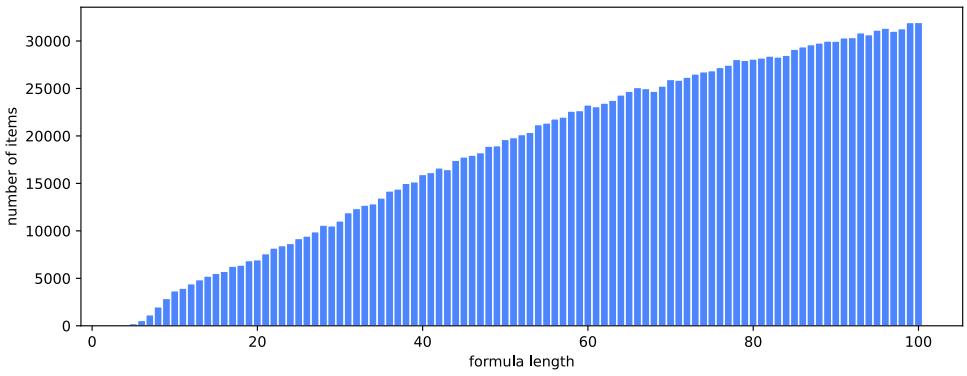


Figure 10: Formula length distribution of `RPC100` (Rep. 5)

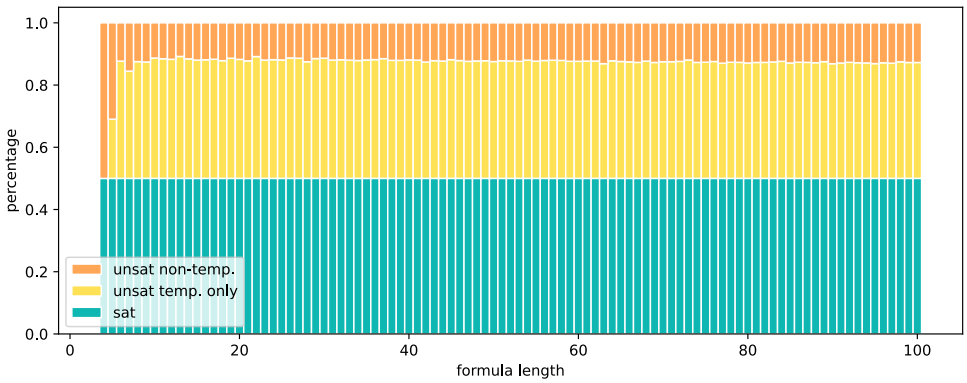


Figure 11: Class proportions by formula length of `RPC100` (Rep. 5)

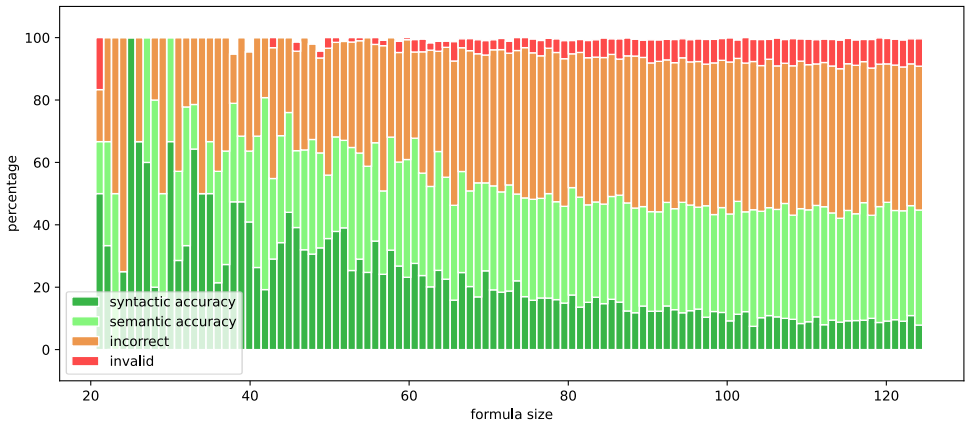


Figure 12: LTL trace generation on dac concat, standard PE, 100k steps (Rep. 6)

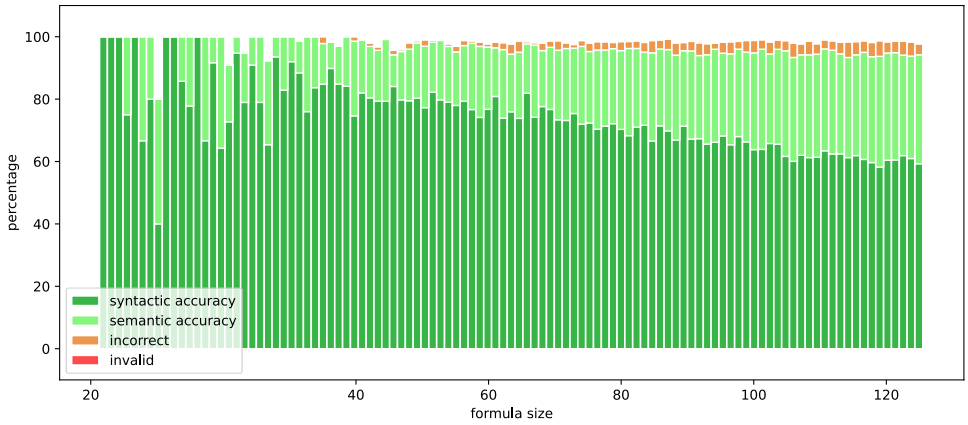


Figure 13: LTL trace generation on dac concat, tree PE, 100k steps (Rep. 6)

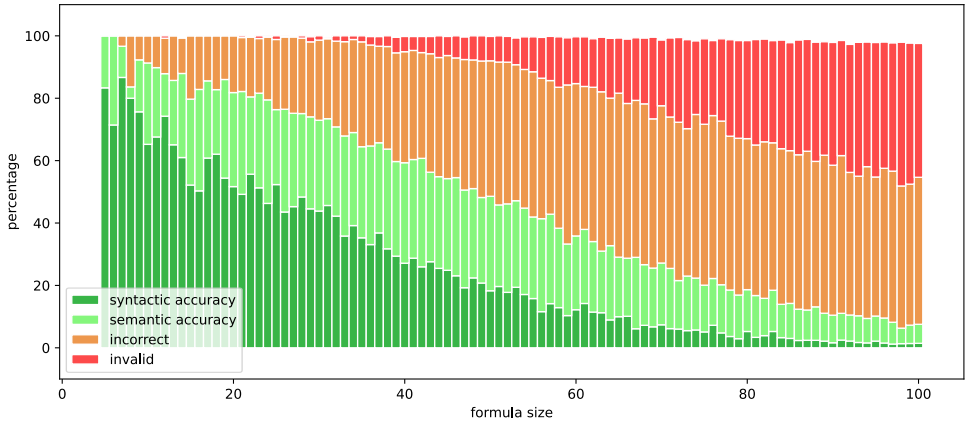


Figure 14: LTL trace generation on `RPC100`, standard PE, 100k steps (Rep. 7)

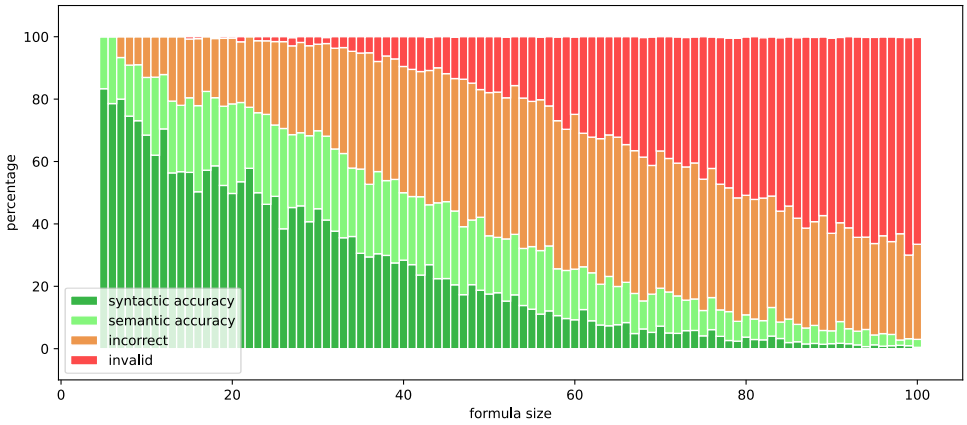


Figure 15: LTL trace generation on `RPC100`, tree PE, 100k steps (Rep. 7)

4 Transformer classifier for LTL-SAT

In this section, different approaches to build a neural classifier that determines the satisfiability of LTL formulas end-to-end are presented. Following the observation in [39] how well Transformers are able to handle logic formulas, the studied classifiers rely on Transformers as their main component. The first variant (Section 4.1) uses standard Transformer encoders [96] and starts with an architecture similar to [39], exploring the effects of different hyperparameters. The second variant (Section 4.2) uses Universal Transformers [20] and evaluates several extensions.

Evaluation All experiments are conducted on the `RPC100` dataset (see Section 3.2), so results are comparable. For a batch size of 1024, 40 epochs correspond to roughly 56k optimization steps, which is where most experiments are evaluated. Some selected runs use double the steps (80 epochs for batch size 1024, corresponding to ca. 112k steps). The metric used for evaluation is the error rate (1− accuracy) on the dataset's validation split. In tables, the column "err @ 56k" therefore denotes the validation error rate after 56,000 training steps. Error rates are given in percent and represent the average over several runs with identical parameters. The value obtained from a run represents the exact result of a test performed at the indicated training step, which means there is some inherent noise in those values. Standard deviations between the results of different runs are also provided, denoted by "σ".

4.1 Standard Transformer approach

4.1.1 Architecture

While the originally proposed Transformer architecture was designed for sequence-to-sequence tasks and therefore consists of distinct encoder and decoder stacks [96], for a classification task, only a single output value, the class prediction, is required. The first step to build a Transformer classifier is therefore determining a way of adjusting the standard Transformer architecture to best suit the classification scenario. Experiments (Table 2) showed that no decoder is required and the most reliable architecture is the following: The input formula is fed into a standard Transformer encoder, yielding a tensor of shape $bs \times l \times d_{\text{emb}}$, where bs is the batch size, l the maximum sequence length and d_{emb} the embedding dimension of the Transformer. A position-wise dense layer (an affine transformation) with output dimension 1 is applied to obtain a per-position prediction logit. These are averaged over all positions and finally, a logistic sigmoid is applied, which represents the final class prediction probabilities. A standard cross-entropy loss of this prediction against real class labels, averaging over batch elements, is employed.

Input formulas are treated similar to [39]: They are represented in polish notation which does not require any parentheses. Each token is encoded by a predefined vocabulary and the encoder is provided with the indices. The maximal formula length of 100 is fixed for the architecture; shorter formulas are padded and the padding mask is given to the encoder. After the initial learned embedding layer and scaling by $\sqrt{d_{\text{emb}}}$, a standard positional embedding [96] is added to the representation. A tree positional encoding like in [39] is supported, but proved to be unnecessary and is therefore not used in any subsequent experiments. The interested reader can enable it by specifying `tree_pe = true` in the parameters (compare the reproducibility Section 8).

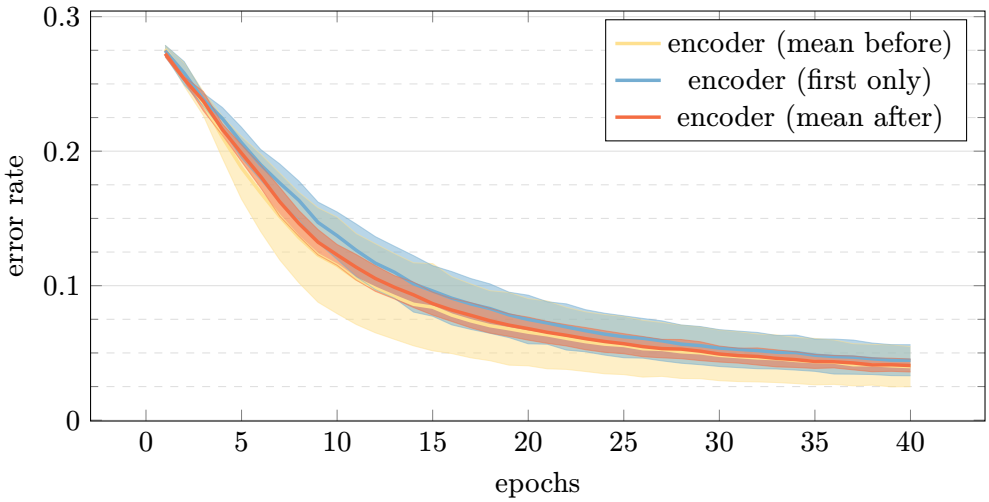


Figure 16: Comparison of different prediction aggregation methods during training, with depicted standard deviations between runs. 5 runs (Rep. 9)

For training, the standard Transformer optimization setup [96] is employed: The Adam optimizer [57] with $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 1e-9$ and a decaying learning rate with 4000 warmup steps. The default model uses an embedding dimension of $d_{\text{emb}} = 128$, $n_l = 6$ layers, $n_h = 8$ heads, a feed-forward dimension of $d_{\text{FF}} = 1024$ and a batch size of $bs = 1024$ (compare Rep. 10).

4.1.2 Results

Prediction aggregation

To find the most suitable architecture for classification, several adaptations of the standard Transformer architecture are compared. As baseline, the exact architecture from [39] including a decoder is used. However, instead of a trace, a single target output symbol, `0` or `1`, is given. To match the architecture exactly, the same hyperparameters (e.g. $n_l = 8$ layers) are used (compare Rep. 8). This baseline is compared with different possible encoder-only architectures. To obtain a single prediction for an input formula, the shape of the encoder output tensor must be reduced two times: Projecting the embedding dimensions to 1 and aggregating over the different positions of the sequence. The most simple tested variant ("first only") drops all final encoder embeddings except for the very first position and applies the learned affine transformation only to it. Another variant ("mean before") takes the mean of all final embeddings over all positions for each embedding dimension independently and applies the transformation afterwards. The last variant ("mean after") first transforms all embeddings to dimensionality 1 and then averages.
As evident from Table 2, the variant with distinct decoder performed the worst, so for the

architecture	err @ 56k	σ
encoder + decoder	4.87	0.76
encoder (first only)	4.46	1.16
encoder (mean before)	3.99	1.49
encoder (mean after)	4.06	0.48

Table 2: Comparison of different prediction aggregation methods. 5 runs (Rep. 8, Rep. 9)

d_{emb}	n_l	n_h	d_{FF}	bs	err @ 56k	σ
128	2	4	1024	1024	8.75	0.30
	4	4	1024	1024	4.39	0.52
	6	4	1024	1024	3.51	0.51
		8	1024	512	10.39	0.94
				1024	3.19	0.42
				2048	1.86	0.05
			2048	1024	3.59	0.28
	8	8	1024	1024	2.93	0.72
				2048	1.86	0.10
256	6	8	2048	1024	3.31	0.15

Table 3: Hyperparameter comparison. 3 runs (Rep. 10)

d_{emb}	n_l	n_h	d_{FF}	bs	err @ 112k	σ
128	6	8	1024	512	6.63	0.68
				1024	2.26	0.23
				2048	1.62	0.09
	8	8	1024	1024	2.35	0.07
				2048	1.56	0.07

Table 4: Hyperparameter comparison, double training steps (112k). 3 runs (Rep. 11)

rest of this section, purely encoder-only architectures are considered. Regarding them, combining information from all positions seems to have a positive effect on the error rate. However, they differ surprisingly strongly in their variance, which is more closely depicted in Figure 16. The strong plot lines are the mean of the evaluation error rate between different runs after each epoch. The light area surrounding each curve represents the standard deviation between them. Taking the position-wise mean after projecting down the dimensionality yields a much lower variance at each step during training. This is therefore the default architecture used in every subsequent experiment.

Hyperparameters

To explore the capabilities and limits of a Transformer-based LTL-SAT classifier, a study on the effects of different hyperparameters is performed in Table 3. Additionally, Table 4 lists results of selected parameter combinations for double the training steps. The default model with aforementioned parameters $d_{\text{emb}} = 128$, $n_l = 6$, $n_h = 8$, $d_{\text{FF}} = 1024$ and $bs = 1024$ serves as reference point. Notably, performance increases steadily with more layers, while increasing them from 6 to 8 only yields diminishing improvements. Increasing the embedding dimension d_{emb} as well as the feed-forward dimensionality d_{FF} yields slightly worse performance than the reference.
The batch size however has a decisive impact on training, the difference between $bs = 512$ and $bs = 2048$ is tremendous. Not only does it strongly affect the final error rate, where there is a big difference even for long training, but also the variance between runs, which is significantly smaller for $bs = 2048$ in all scenarios. Figure 17 visualizes the effect of different batch sizes during training for altered version of the reference model. Similar to Figure 16, the lighter areas represent the standard deviations between different runs with identical parameters. Note that variants with larger batch sizes consequently require more epochs of training on the dataset. Figure 17 takes the training steps as common reference, however naturally, training with higher batch sizes takes considerably longer

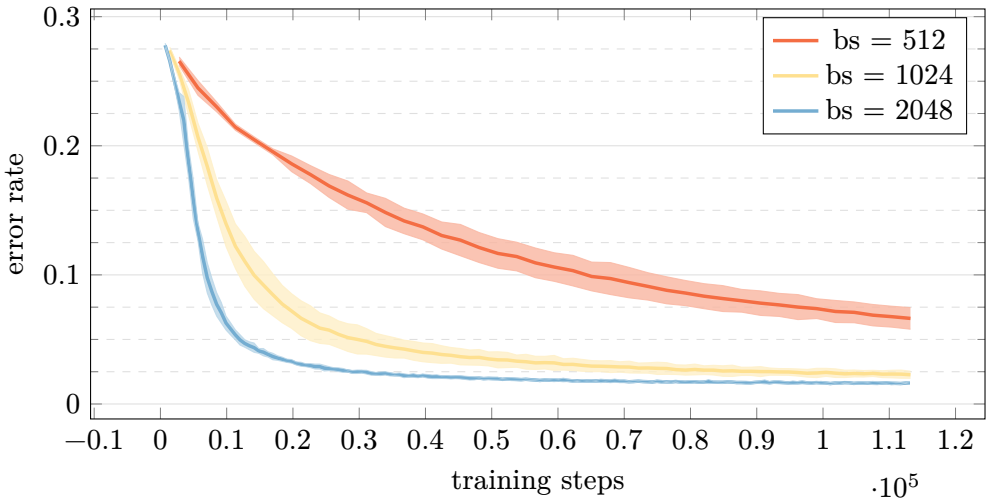


Figure 17: Comparison of different batch sizes during training, with depicted standard deviations between runs. 3 runs (Rep. 11)

for the same amount of training steps. The use of large batch sizes is also limited by the available memory of the computing device. If this is available, however, results strongly suggest using the highest possible size.

4.2 Universal Transformer approach

Given the algorithmic nature of the LTL satisfiability problem and the fact that worst-case computation steps required scale with the problem length, it would be desirable to have a neural classifier whose number of computation steps is not fixed, but can be scaled dynamically. The Universal Transformer ([20], compare Section 2.2.3) is an architecture that could potentially fit this requirement. With its recurrent way of computation, the number of iterations can be freely specified and even adjusted given the current prediction. This section presents a classifier architecture based on the Universal Transformer.

4.2.1 Architecture

The architecture is largely similar to the previous approach with identical input and output processing (including the aggregation of the final prediction). However, instead of the standard Transformer encoder, a Universal one is used. This means that as a central new hyperparameter, the number of recurrent iterations must be specified.
The implementation of the Univeral Transformer encoder in this slightly differs from the original proposed one [20]. In the original architecture, the positional encoding was added at each iteration together with an incremented temporal encoding. Both is omitted in this implementation since it showed to moderately improve performance and no negative effects could be observed. Similar to the standard Transformer approach, the positional encoding is only added after the initial embedding. Additionally, this implementation allows to use multiple different layers that are then executed recurrently, called base layers.
As an implementation detail, it is noteworthy that specifying a large number of iterations (more than 10 on a GPU with 40 GB memory) requires large amounts of memory to store intermediate results for gradient backpropagation. Consequently, a gradient checkpointing training loop was implemented: On a forward pass, the intermediate results between

recurrent applications of the Transformer layer are saved, but no gradients of the operations inside. When the gradient is backpropagated, a single recurrent application is executed again for each saved intermediate value from last to first, where this time all gradients of that application are tracked. While it increases training time by essentially requiring two forward passes, it significantly reduces memory consumption. This behavior must be enabled manually by setting `ut_gradient_method` to `checkpoint` instead of `tape` (compare Rep. 12).

For training, the same optimization parameters as for the standard Transformer approach are used. The default model uses a single recurrent layer (base layer count $bl = 1$) which is applied for $it = 6$ iterations. Other parameters are identical to the standard Transformer model with $d_{\text{emb}} = 128$, $n_h = 8$, $d_{\text{FF}} = 1024$ and $bs = 1024$.

4.2.2 Main results (hyperparameter study)

Several hyperparameters are compared in Table 5 (standard training steps) and Table 6 (double training steps). As a baseline comparison with the previous standard Transformer approach, models with as many base layers as iterations are trained, effectively applying each layer exactly once. Results closely match the baseline (compare the last two rows of Table 5 with Table 3). However, interestingly, using multiple base layers ($bl = 2$) with equally many total applications ($it = 6$, applying each base layer three times) leads to significantly worse results. Having only a single base layer with 6 iterations performed best of these variants by a significant margin. Consequently, for all subsequent experiments, a single base layer $bl = 1$ is used.

The low error rate of the $bl = 1, it = 6$ pure Universal Transformer variant is surprising. It already outperforms the standard Transformer one, which has equally many forward passes but much more trainable parameters and is trained twice as long. This suggests that generally, having many layers and thus many trainable parameters is harmful, but this is overshadowed by allowing more forward passes in the standard Transformer, which is again beneficial. Indeed, increasing the number of iterations leads to even better results with almost only 1% errors for $it = 12$. Increasing iterations even further requires longer training (compare Table 6) without improving results significantly.

Similar to the standard Transformer approach, increasing the batch size leads to a considerably lower error rate for each number of iterations. Setting $bs = 2048$ is the only way to benefit from more than 12 iterations, with the lowest error rate observed in this thesis, 0.35%, at 20 iterations.

d_{emb}	n_h	d_{FF}	bl	it	bs	err @ 56k	σ
128	8	1024	1	6	1024	1.82	0.06
				9	1024	1.25	0.02
				12	1024	1.07	0.02
					2048	0.59	0.02
				16	1024	1.15	0.02
				20	1024	1.58	0.09
			2	6	1024	3.91	0.32
				12	1024	2.94	0.14
			6	6	1024	2.89	0.56
			8	8	1024	2.87	0.20

Table 5: Hyperparameter comparison. 2 runs (Rep. 12)

d_{emb}	n_h	d_{FF}	bl	it	bs	err @ 112k	σ
128	8	1024	1	9	1024	0.81	0.04
				12	1024	0.63	0.02
					2048	0.47	0.02
				20	1024	0.62	0.07
					2048	0.35	0.01

Table 6: Hyperparameter comparison, double training steps (112k). 2 runs (Rep. 13)

4.2.3 Additional experiments

Training with iteration curriculum Since previous experiments show that training with many iterations (more than 12) required many training steps and a high batch size, the question arises whether this can be optimized. To this end, a training scheme with an iteration curriculum is tested: The iteration count for the Universal Transformer is not fixed, but is increased stepwise every few training epochs. At the first epoch, only 2 iterations are specified, and at epoch 40 (equivalent to 56k steps), 20 iterations. Results in Table 7 show that this approach performs very similar to the one with fixed 20 iterations, but even slightly worse. While training is faster due to omitting a high iteration count in the beginning, an iteration curriculum therefore proves to be not beneficial.

Pre- and postprocessing layers Even though multiple recurrent base layers proved to be harmful for performance, it could be possible that the Universal Transformer architecture benefits from pre- or postprocessing with different, non-recurrent Transformer layers. To this end, two variants were trained that feature additional standard Transformer layers before the actual Universal Transformer and after it, respectively. These are compared to a pure Universal Transformer architecture with the same amount of total forward passes. However, as apparent from Table 8, no positive effect can be observed, both for low and medium iteration counts. Interestingly, there is a difference between pre- and postprocessing layers, which could be related to more difficult gradient updates through the recurrent applications. No tests with higher iterations were performed since this would require implementing gradient checkpointing to this variant, which did not seem worth the effort.

Evaluation at non-target iterations In principle, the Universal Transformer should be flexible to changes in the iteration count. This paragraph examines the accuracy of the architecture when evaluated with a different iteration count than trained with. As first experiment, a standard model with 12 iterations is trained for the default 56k steps and then evaluated at different iterations (compare Rep. 16). Results are shown in the first row of Table 9. When compared to previous results in Table 5, performance is consistently worse than for any models trained directly for said iteration count. This suggests that when trained with fixed iterations, the Universal Transformer architecture should also only be evaluated at that count.
As second experiment, the architecture is trained with random iterations. Concretely, after each iteration, there is a probability of $p = 0.0833$ to stop the recurrent process and

it	max it	bs	err @ 56k	σ @ 56k	err @ 112k	σ @ 112k
curriculum	20	1024	1.63	0.14	0.70	0.05

Table 7: Variant with iteration curriculum. 2 runs (Rep. 14)

bl	*bs*	Σ passes	pre	*it*	post	err @ 56k	σ @ 56k	err @ 112k	σ @ 112k
1	1024	6	3	3	0	3.08	0.03	2.00	0.13
			0	6	0	1.82	0.06	-	-
			0	3	3	2.34	0.09	1.58	0.02
		9	3	6	0	2.79	0.34	1.95	0.25
			0	9	0	1.25	0.02	0.81	0.04
			0	6	3	2.12	0.36	1.19	0.02

Table 8: Variant with additional pre- or postprocessing layers. 2 runs (Rep. 15)

it	max *it*	*bs*	err @ 6 it	err @ 9 it	err @ 12 it	err @ 16 it	err @ 20 it
12	12	1024	22.1	2.9	1.1	1.3	2.3
$\mathbb{E}[it] = 12$	20	1024	43.0	27.0	-	1.4	1.0

Table 9: Evaluation at different iterations and randomized iteration count for training, both trained for 56k steps. 2 runs (Rep. 16, Rep. 17)

proceed with the final prediction aggregation. This results in a geometric distribution of iterations performed with expected value $\mathbb{E}[it] = 12$ (compare Rep. 17). Results in the second row of Table 9 show that performance is seriously deteriorated for low iteration counts, much more than for the fixed variant even though this one was trained many times on exactly those iterations. However, surprisingly, the performance for higher iterations counts is improved, even though by the probability distribution, they only appeared for a relatively low amount of times. For 20 iterations, the result is even much better than for a model trained solely on that count (compare to Table 5). However, it is not significantly better than the best model with fixed iterations at 56k steps. While these observations do not build a clear image of which way is best to train the Universal Transformer architecture for many iterations, they generally encourage both training and evaluating it only at fixed and identical iterations.

5 Transformer GAN for LTL formula generation

In this section, the application of generative adversarial networks (GANs) to generate LTL formulas is studied. The use of the original GAN formulation [34] in a such a sequential symbolic domain is not directly possible, since GANs fundamentally operate on continuous vectors with fixed dimensionality. To cope with the variable length sequences and inspired by the previously observed great performance of Transformer-based architectures on logic formulas, a GAN built with Transformer encoders is presented in this section. Both a standard GAN and a Wasserstein GAN with gradient penalty (WGAN-GP) variant are examined.
After defining some metrics to assess GAN performance in Section 5.1, two methods to cope with the non-differentiability of symbols are explored. The first one (Section 5.2) uses a continuous one-hot encoding with added noise to represent the symbols. The second approach (Section 5.3) learns a continuous embedding beforehand on a classification task. Finally, a classifier is integrated into the GAN architecture in Section 5.4 to explore ways of synthesizing more challenging instances than those from the base dataset.

Some of the content in this section has already been published in [58]. The presented GAN architecture works for any domain of variable-length token sequences and in has been applied to both LTL formulas and symbolic math expressions in that paper. For simplicity and keeping the application to temporal logics, in this thesis, only LTL formulas are considered.

Dataset Specifically, all subsequent experiments, if not specified differently, operate on the `RPC100` dataset with formula length limited to 50. This length limit is enforced during training by setting `max_encode_length = 50` and does not require modifying the dataset. However, to avoid confusion, in the following, the term `RPC50` is used to refer to this implicitly defined dataset. Note that classes (satisfiable and unsatisfiable) are still balanced since this was enforced for each length equally (refer to Section 3). Similar to Section 4, formulas are represented in polish notation to make parentheses obsolete.

Naming conventions An individual object from the dataset (here, an LTL formula) is referred to as *instance* or also *sample* if obtained from the GAN. Since the instances' sequential structure is coped with by the Transformer encoder, which processes each sequence element in parallel, the sequential axis is mostly disregarded in the following. The focus lies on the representation and processing of individual sequence elements (like a and $\Diamond$), which are consequently referred to as *elements*.

5.1 GAN metrics

It is not obvious to determine a measure of quality for generated formulas. While the GAN critic outputs a value for each generated instance, it is only meaningful in the context of the current training step. Only a critic trained to optimality (with a fixed generator) would present at least an approximation of the true divergence between the real and generative distributions. This would also be achieved if training converges at some point, but this was not observed in any experiments.
Additionally, the previously mentioned disconnection between the actual discrete symbols and the continuous vectors on which the GAN operates means that the critic only assesses quality of the latter, which may not correlate with quality of the actual symbol sequence. Consequently, an objective quality measure should be independent of the GAN and operate on the discrete symbolic output sequence.

Parsing metrics The simplest objective measures rely on the inherent syntactical requirements of a formula: It must form a tree with binary ($\wedge$, $\mathcal{U}$) and unary ($\neg$, X) operators and leaf nodes (1, a). A binary evaluation is therefore obtained by checking the following proposition: *The sequence is parsable as tree with no remaining tokens.* The fraction of instances that fulfill this test is called the *fully correct* (fc) metric, which is similar to a binary accuracy.
A more fine-grained description of a syntactically incorrect formula can be obtained by examining the parsing process: *Try to parse a valid tree from the list of tokens and if tokens remain, start again with the remaining ones.* For example the token sequence $\wedge\neg abb\bigcirc c$ (polish notation) would break into the three fragments $\neg a \wedge b$, b, $\bigcirc c$ (infix notation). The mean amount of obtained fragments is called the *parse fragments* (pf) metric, which should ideally converge to 1. Note that it is also possible for formulas to be incorrect with only a single fragment (e.g. $\bigcirc\bigcirc\bigcirc$).

Sequence entropy An important characteristic of formulas is not only syntactical correctness, but also heterogeneity. The formula $\square\square\square\square a$, while being syntactically correct, consists mostly of repetitions of the same operator. This is a particular issue with unary operators, since they can be repeated arbitrarily often without affecting the syntactical correctness of the formula. To quantify this, a *sequence entropy* (se) metric is employed: *Calculate the relative number of occurrences of a token with respect to the length of the whole sequence and compute the entropy over this relative number for each distinct occurring symbol.* For example, the formula above achieves a sequence entropy of $-4/5 \log(4/5) - 1/5 \log(1/5) \approx 0.5$. Note that the maximally achievable entropy depends on the sequence length. Values obtained by this metric are therefore only comparable between sequence sets with equal length distribution.

Observed quantities and plots Contrary to Section 4, which could rely on the accuracy of the validation set computed after each epoch for evaluation, the GAN is assessed by metrics of its immediate output. These are computed directly on the generated instances each few training steps (determined by `gan_trainsteps_infer_interval`, 10 is default) and change rapidly and non-smoothly. Moreover, they typically show a large variance between individual runs with identical parameters (especially the fully correct metric). Because of this, results with only few runs (e.g. 2) should be interpreted qualitatively. Additionally, the instability of the metrics over time require strong smoothing to be applied to obtain interpretable plots. If not specified otherwise, an exponential smoothing with $\alpha = 0.95$ is applied to all examined quantities obtained during GAN training. For results in tables, also the smoothed value at the respective training step is given. Before, an average over all runs is taken.

5.2 Direct one-hot approach

To obtain a continuous, differentiable representation for the GAN to work on, for this approach, the one-hot encoding of a real instance's elements is used (compare Section 2.2.1). This results in a $|V|$-dimensional vector (V is the token vocabulary), which contains a single one at the token's index and zeros otherwise. By allowing real values between zero and one, a continuous output space for the generator is already found, even though the real elements are all corner points of it. The following subsection 5.2.1 explains the architecture of a Transformer GAN operating on this representation in detail. Key results are presented in Section 5.2.2. Section 5.2.3 demonstrates the use of the architecture to create a large, meaningful dataset from much fewer examples. Finally, some more in-depth experiments are conducted in Section 5.2.4.

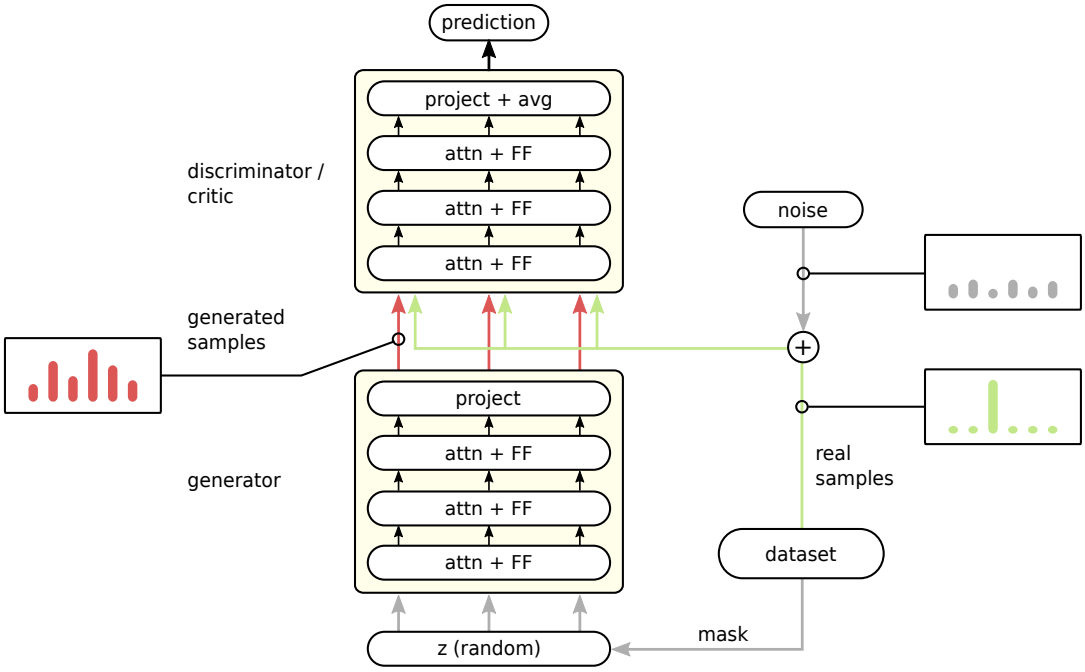


Figure 18: Direct GAN architecture with noise

5.2.1 Architecture and training setup

As main components of the proposed GAN architecture, Transformer encoders are used for both the critic and the generator. Notably, the generator encoder computes its output in a single pass, thereby not requiring autoregression to build an output sequence. This means there are only very few steps through which consistency of the whole formula must be achieved. An overview of the architecture is depicted in Figure 18 and the following paragraphs detail its components.

Generator The random input z to the generator is sampled from a uniform distribution $[0, 1]$ per element, constituting a tensor of shape $bs \times l$. An affine transformation (a dense layer with bias) maps the one-dimensional input noise to the encoder embedding dimension d_{emb}. Afterwards, it is processed by a standard Transformer encoder stack with residual layers of self-attention and fully-connected mappings.
A specialty is the padding mask, which is normally determined by the input sequence length in a standard Transformer. For the presented GAN architecture, a target sequence length is likewise determined before processing and a padding mask computed accordingly. There are two ways of choosing the sequence length: Sampling it randomly or copying it from a batch of real examples from the dataset. Since for the Wasserstein approach (compare subsequent paragraph on gradient penalty) it is desirable for the ith instances in the real and generated batch to have identical lengths, this copying procedure is used by default. Experiments with standard GANs have shown no significant difference between using a copied mask or a randomly determined one. While identical padding masks for real and generated samples are desirable for training Wasserstein critics, but irrelevant for training Wasserstein generators, further experiments have shown that even using a copied mask for the training of the generator did not affect performance. Due to this, for the sake of simplicity the copying is also employed for generator training.
Regarding the further architecture, after the final Transformer layer, the embedding dimension is projected to the vocabulary size by an affine transformation and a `softmax`

is applied. Thereby, both real and generated elements formally have the properties of a probability distribution over vocabulary tokens in that their components are non-negative and sum up to 1. Experiments showed that achieving the sum constraint by applying the softmax leads to far better training results than having the generator learn this property by itself.

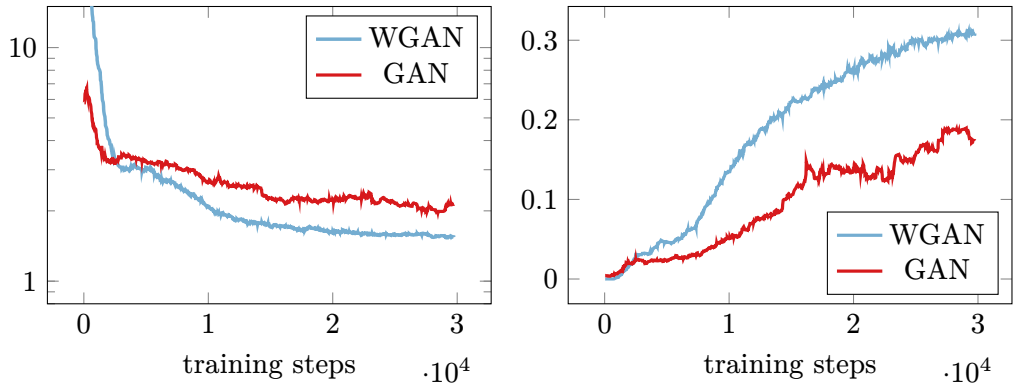
Discriminator/critic A GAN discriminator and WGAN critic are virtually identical in terms of their architecture. The only difference is that a critic outputs an unconstrained real scalar value where a discriminator outputs a probability. The latter is achieved by applying an additional logistic sigmoid function. For input processing, each $|V|$-dimensional element is first mapped to d_{emb} by an affine transformation before being processed by a standard Transformer encoder stack. After the last layer, the final embeddings are aggregated over the sequence by averaging, similar to the classifier described in Section 4.1, and projected to the scalar output value by an affine transformation.

Additive noise for real samples Working in the token probability space poses the problem that the elements of real samples (one-hot vectors) are exclusively corner points. Good critic functions would therefore be very non-smooth and consequently bad for gradient-based optimization. Less formally, if the discriminator/critic learns to identify real elements by their characteristic one-hot distribution, this would make GAN training likely impossible. To alleviate this problem, Gaussian noise $N(0, \sigma^2_{\text{real}})$ samples are obtained and their absolute value is added to the one-hot representation of real elements. The vectors are then re-normalized to sum up to 1, thus they lie again in the probability space, but not exactly at its corners. In Figure 18, the three cornered boxes depict the different appearance of the vectors. Section 5.2.4 studies the effect of different added noise levels more closely.

Gradient penalty for Wasserstein GAN Wasserstein GANs require a way of enforcing smoothness, which is usually achieved by a gradient penalty as proposed in [37]. However, in a variable length sequence domain, it is not obvious how to adapt the formulation. One could consider the padded, fixed-length sequences as target vector or only the variable-length sequences with actual relevant information. Moreover, it could be meaningful to consider the sequence as single vector for calculating the norm or each position (element) individually, using an element-wise average. From experiments, it proved to be best to use a norm over the full variable-length sequence. Therefore, the norm is calculated as follows: For input x with shape $bs \times l \times |V|$, calculate $\nabla_x C(x) =: d_x$ where C is the critic function. Sum d_x^2 over the last (vocabulary) axis and mask out padded positions (multiply by zero). Sum over the sequence axis and take the square root. Formally this reads

$$l_{dx} := \sqrt{\sum_{i=1}^{l} (1 - \text{pad}(i)) \sum_{j=1}^{|V|} (d_x^2)_{i,j}}\,, \tag{23}$$

where pad indicates whether the respective position is padded. The result l_{dx} is an effective gradient length for each instance in a batch.
Since this value must be computed on linearly interleaved real and fake samples, they must have identical length to preserve meaningfulness. This is why, as mentioned previously, the padding mask used in the generator is copied from a respective real instance from the dataset. This ensures equal length and the gradient penalty can be computed in a straightforward way.



(a) parse fragments (logarithmic) (b) fully correct fraction

Figure 19: Parse metrics for GAN and WGAN variants. 3 runs (Rep. 18 and Rep. 19)

Training setup and default hyperparameters Both GAN and WGAN variants are trained with Algorithm 1 introduced in Section 2.3.3, with their respective objectives. Notably, the GAN variant is trained with the alternative loss from Equation 13. For the WGAN, the gradient penalty is enforced towards a gradient target length of 1 with $\lambda_{GP} = 10$. As default, $n_c = 2$ critic training steps per generator training step and a batch size of $bs = 1024$ is used. Gaussian noise is added to real samples with a standard deviation of $\sigma_{\text{real}} = 0.1$ for all models to get comparable results. The generator and critic encoders both use an embedding dimension of $d_{\text{emb}} = 128$, $n_h = 8$ attention heads and a feed-forward network dimension of $d_{\text{FF}} = 1024$ per default. The default generator has $n_{lG} = 6$ layers, where the default critic only has $n_{\text{lC}} = 4$. Both are trained with the Adam optimizer [57] with parameters $\beta_1 = 0, \beta_2 = 0.9$ and constant learning rate $lr = 1e-4$, similar to [37]. Training with a higher value for β_1 led to very unstable training as did the use of the Transformer learning rate schedule [96]. In Section 5.2.4, different hyperparameter settings are compared.

5.2.2 Key results

Generating syntactically correct formulas During training of GAN and WGAN models with default parameters, generated instances are sampled periodically and converted to their text representation, which is achieved by taking the *argmax* on the vocabulary axis. This text sequence is used to calculate GAN metrics (compare Section 5.1), displayed in figures 19 and 20.
Both GAN and WGAN variants improve in performance relatively continuously, but eventually reach their limit around 30k training steps. Still, both generators are able to produce a large fraction of fully correct formulas, despite the length of the instances (up to 50 tokens) and the non-autoregressive, fixed-step architecture. Two examples are listed in the following (↩ is a line break):

$$\neg(h \to h)\,\mathcal{W}\,\bigcirc(g \vee h) \wedge (g \wedge g) \wedge \Diamond\Box\neg\Box j \wedge \neg\Box j\,\mathcal{W}\,\neg b \wedge \Box(\Box h \wedge \bigcirc j \to \Box j) \wedge \Diamond\Diamond j \quad ,$$

$$\bigcirc\bigcirc\bigcirc(c \vee i) \wedge \neg d \wedge \neg\Diamond\bigcirc c\,\mathcal{W}\,\neg c \wedge \Box(\Box d \wedge \neg((b \leftrightarrow c) \hookleftarrow$$
$$\leftrightarrow \Box c) \to \bigcirc(c \leftrightarrow \Box d)) \wedge \Box(b \wedge d \to \Diamond\Diamond\Box\Box d) \wedge c \quad .$$

Differences in homogeneity Interestingly, when comparing valid generated formulas from both variants, it appears that the standard GAN would often produce instances in the likes of

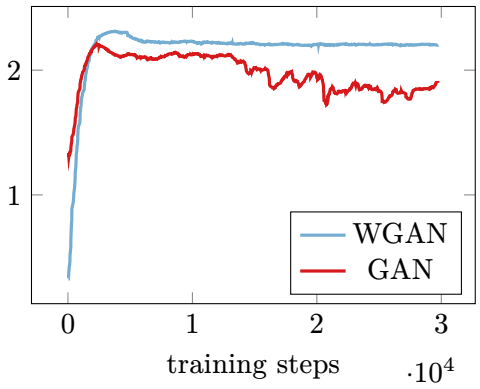


Figure 20: Sequence entropy for GAN and WGAN variants. 3 runs (Rep. 18 and Rep. 19)

$$\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\Diamond i \wedge \bigcirc\bigcirc i \wedge \bigcirc\bigcirc\bigcirc\bigcirc \neg \bigcirc \Box \neg\neg \Diamond i \wedge \neg\neg(g \wedge g \wedge i) \quad ,$$

which contains many repetitions, especially of the $\bigcirc$-operator. In fact, some GAN runs achieved fully correct fractions above 30% (higher than WGAN), but these exclusively produced formulas with such low internal variety. To quantify this, the average sequence entropy during training is calculated. Figure 20 shows that indeed this metric decreases for the GAN variant during training but remains stable for WGANs. A possible interpretation of this phenomenon is that the critic incidentally learns to check syntactic validity to some extend and some generators "exploit" this fact by producing correct, but repetitive formulas.
Due to this general trend of standard GANs to perform poorly in terms of formula homogeneity, most consecutive experiments, especially Section 5.4 exclusively use the WGAN variant.

Discriminator / critic predictions During training, the GAN discriminator quickly learns to identify real and generated instances, as depicted in Figure 21. The prediction curves very quickly reach values above 0.99 and below 0.01, respectively, and never change their directions. Similarly, the WGAN critic's Wasserstein distance estimate soon reaches a value of around 0.4 at which it remains for the rest of training. For this behavior, one would expect the generator to not improve significantly, which is contrary to the observed improvements in objective formula quality. This underlines the importance of independent metrics and shows that GAN training can be successful despite never converging.

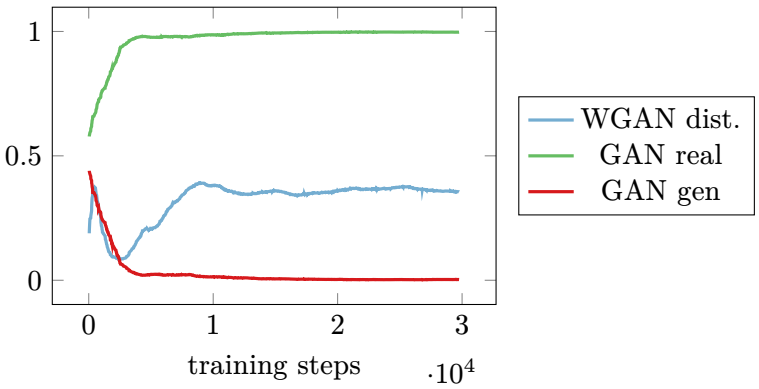


Figure 21: GAN real/generated predictions and WGAN Wasserstein distance estimate. 3 runs (Rep. 18 and Rep. 19)

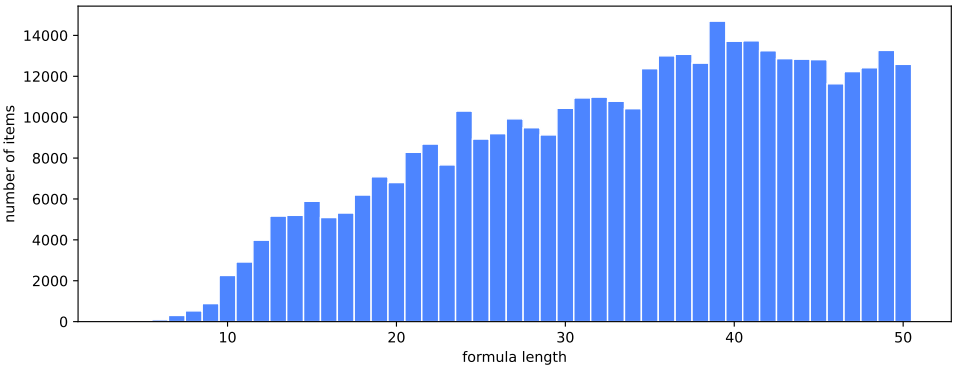


Figure 22: `Generated` dataset size distribution (average size 33.6). Rep. 23

5.2.3 Application for dataset substitution

In this subsection, the usability of the presented GAN architecture is demonstrated by showing that the large `RPC50` dataset can be substituted with generated training data when training a classifier for LTL-SAT.

Generated dataset Firstly, a small random subset containing 10k instances from the `RPC50` training set is obtained. This dataset is called `RPC50-10k` (Rep. 20). On this dataset, a WGAN with default parameters but smaller batch size of 512 is trained (Rep. 21) for 15k steps. After training, 800k syntactically correct generated formulas are sampled from it, which constitute the dataset `Generated-raw` (Rep. 22). Notably, only two instances of `RPC50` ($(\square\neg a) \wedge (\square\bigcirc a)$ and $(\square\square e) \wedge (\square\neg e)$), i.e. only 0.02%, reappeared in the 800k sampled instances. Additionally, only 2.3k of the 800k (0.28%) generated formulas were duplicates, which displays an enormous degree of variety.
This `Generated-raw` set is processed similar to the original dataset: Duplicates are removed and satisfiable and unsatisfiable instances are balanced to equal amounts per formula size. 400k instances are kept randomly and the resulting dataset is called `Generated` (Rep. 23). Its size distribution is plotted in Figure 22.

Results Several Transformer classifiers are trained on the original set `RPC50`, the small `RPC50-10k` and the `Generated` set. For completeness, also a 100k-sized subset `RPC50-100k` is used. The classifiers use $n_l = 4$ layers and a batch size of $bs = 1024$ and are trained for 50k steps (Rep. 24). Notably, the target epoch count must be adjusted to match the different sizes of the datasets.
Resulting accuracy values are compared in Table 10 and selected training curves are

trained on	bs	train acc @ 30k	val acc @ 30k	train acc @ 50k	val acc @ 50k
`RPC50`	1024	96.6% (0.5)	95.5% (0.4)	98.1% (0.3)	**96.1%** (0.3)
	512	92.4% (0.7)	93.0% (0.8)	95.4% (0.5)	95.0% (0.8)
`RPC50-100k`	512	95.3% (0.7)	88.3% (0.9)	98.1% (0.3)	87.8% (1.0)
`RPC50-10k`	512	100% (0.1)	76.4% (1.7)	100% (0.0)	75.5% (1.5)
`Generated`	1024	95.4% (0.2)	93.6% (1.0)	97.1% (0.1)	**93.9%** (0.3)

Table 10: Accuracies of small Transformer classifiers trained on different datasets and evaluated on `RPC50` (standard deviations in parentheses). 5 runs (Rep. 24)

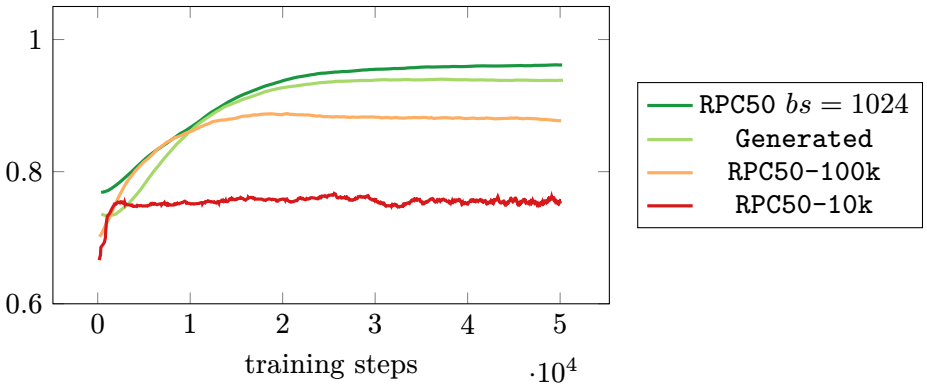


Figure 23: Validation accuracy (on `RPC50`) during training of Transformer classifiers on different datasets. 5 runs (Rep. 24)

depicted in Figure 23. Importantly, the validation accuracy is computed only on the original `RPC50` dataset for all classifiers. This simulates a setting where only limited training data is available and the performance on a ground-truth set is the quantity of interest. As apparent from the table, a reduced number of training examples strongly decreases performance on the original dataset. Having only 10k instances available leads to immense overfitting and poor accuracy. Out of several tries, no setting was found that allowed training a classifier on `RPC50-10k` with significantly higher validation accuracy.

A classifier trained on the `Generated` set however achieves almost the identical validation accuracy on the original `RPC50` set as the classifier that was actually trained on it. Importantly, as data source, only `RPC50-10k` was used to train the GAN from which the `Generated` set was subsequently constructed. This shows that that the data produced by the presented GAN approach is highly valuable as it can serve as substitute for the complete original training data even when provided with much fewer examples.

5.2.4 Additional experiments

Hyperparameter comparison Hyperparameters that both the GAN and WGAN variants share are compared in Table 11. Interestingly, the number of critic steps per generator step (n_c) has no big influence on any variant. Using fewer steps appears to be even more beneficial. Regarding layers, it seems to be beneficial to have slightly more for the generator than for the critic. While the WGAN variant can handle a deep $n_{lG} = 8, n_{lC} = 6$ configuration albeit not profiting from it significantly, the GAN becomes unstable for this. Generally, the WGAN seems to handle more layers better that the GAN, which can even achieve high fc fractions for a small architecture. The batch size has a much less important meaning than for the classifier in Section 4. Still, large values of 2048 seem to have a positive effect. As described before, the sequence entropy consistently degrades for the GAN with long training. However every single WGAN setting achieved a stable value of 2.2 even for long training. Generally, it is hard to accurately determine positive and negative effects of the considered hyperparameters without many more runs, due to the high variance of outcomes. However they also do not prevent successful training or lead to much better results, so any combination in the considered parameter range should work in principle. The effect of the overall architecture, WGAN compared to GAN, plays a much stronger role than individual hyperparameters. An exception is the amount of additive noise applied to real samples, as studied in the following section.

variant	n_c	n_{lG}	n_{lC}	bs	fc (15k)	fc (30k)	se (15k)	se (30k)
GAN	1	6	4	1024	14.9%	32.5%	2.0	1.8
	2	4	2	1024	14.2%	29.4%	2.0	1.9
			4	1024	11.7%	29.2%	2.0	1.8
		6	4	512	10.0%	16.6%	2.0	1.8
				1024	10.2%	17.9%	2.1	1.9
				2048	28.2%	38.2%	1.9	1.8
			6	1024	11.0%	17.0%	2.0	1.9
		8	6	1024	11.1%	4.5%	1.8	1.7
	3	6	4	1024	11.0%	19.9%	2.0	1.8
WGAN	1	6	4	1024	15.1%	25.4%	2.2	2.2
	2	4	2	1024	13.7%	20.4%	2.2	2.2
			4	1024	15.9%	24.9%	2.2	2.2
		6	4	512	17.5%	25.1%	2.2	2.2
				1024	21.2%	31.2%	2.2	2.2
				2048	24.2%	36.2%	2.2	2.2
			6	1024	22.2%	30.5%	2.2	2.2
		8	6	1024	22.1%	32.1%	2.2	2.2
	3	6	4	1024	19.2%	23.7%	2.2	2.2

Table 11: Hyperparameter comparison at 15k and 30k steps. 2-3 runs (Rep. 25)

Effect of additive noise on one-hot representation The impact of different amounts of noise σ_{real} applied to real elements is compared in Table 12. It strongly affects the performance of the GAN scheme, which is unable to work without added noise. Stronger noise improves this variant's performance. WGAN models on the other hand are not significantly influenced by added noise and are even able to be trained without it. As one would assume, too high amounts of noise hinder successful training for both variants. The table also shows the much higher metric variance between runs of the GAN variant compared to WGAN.

Additionally, in Figure 24, it is compared how the GAN discriminator rates unmodified generated instances and *argmaxed* versions thereof. This is to asses how important the appearance of elements in token probability space is as a criterion to the discriminator. Ideally, this should not be the determining factor (given that it lies sufficiently close to a corner), but rather the global, relative position of individual elements in the sequence. While the score for unmodified instances immediately decreases at the start of the training, it initially rises for the *argmaxed* ones, which could indicate that the local, element-focused information is decisive at this point. Naturally, this distinction is impeded with higher

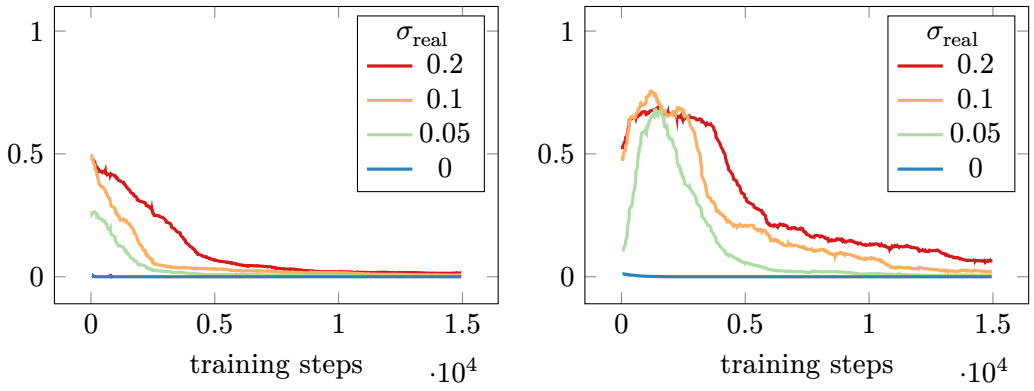
σ_{real}	fc	fc σ	se	se σ
0	0%	0	-	-
0.05	15%	4.1	1.7	0.04
0.1	17%	6.5	1.8	0.03
0.2	41%	4.0	1.9	0.04
0.4	11%	0.1	2.2	0.02

(a) GAN variant

σ_{real}	fc	fc σ	se	se σ
0	26%	1.5	2.2	0.00
0.05	25%	2.5	2.2	0.00
0.1	31%	1.1	2.2	0.01
0.2	25%	0.5	2.2	0.00
0.4	3%	0.1	2.4	0.03

(b) WGAN variant

Table 12: Performance comparison with different amounts of additive noise σ_{real}, with standard deviation for metrics (indicated by "σ"). 3 runs (Rep. 26)



(a) generated, original

(b) generated, arg-maxed

Figure 24: GAN discriminator predictions for generated samples with different noise levels σ_{real} on real samples. 3 runs (Rep. 26)

noise values. However, after a while of training, the scores of the *argmaxed* samples quickly deteriorate and, at least for lower values of σ_{real}, approach their soft-valued counterparts. This could indicate that, indeed, the discriminator eventually shifts its focus away from local, element-wise criteria towards structural, global features. Were this transition not to occur, it is likely that no syntactically correct formulas would be produced.

Wasserstein GAN parameters and quantities In addition to Table 12b, which analyzed the effect of additive noise to the WGAN with respect to performance metrics, Table 13a also shows its effect to several other quantities. The *soft entropy* is the mean entropy over the last axis of the generator's output, the token probability distribution. It shows two things: As expected, with higher noise levels σ_{real} (which is only applied to the *real* instances), the generated instances exhibit a higher entropy. Secondly, with no added noise, the soft entropy is very close to zero, which means that the generator is impressively certain of which token to output at which position. For an intuitive perspective, given that there are 24 possible tokens, an entropy of 0.15 corresponds to a probability distribution over 24 items where the strongest has a probability of roughly 98% and the other ones uniformly share the remaining probability. Notably, the soft entropy is not unique to the WGAN, but as shown in Table 12b, the pure GAN could not be trained without noise, preventing such an analysis.

Interestingly, Table 13a also shows that the additive noise strongly affects both the gradient length for the gradient penalty (l_{dx}) as well as the estimated Wasserstein distance at the end of training. As a reminder, l_{dx} is enforced towards 1 with scaling parameter $\lambda_{\text{GP}} = 10$. Consequently, with low noise levels, the critic can tell real and generated samples easier apart, which aligns with the observations from the previous paragraph regarding the GAN discriminator for different noise levels. Note that still, as shown before in Table 12b, performance is almost not affected at all by this. The bigger values for l_{dx} could indicate that the critic has a tendency of becoming less smooth for lower noise levels. In Table 13b, for fixed noise level $\sigma_{\text{real}} = 0.1$ (default), the effect of different values of λ_{GP} is studied. While as expected the gradient length l_{dx} behaves inversely to the parameter, the estimated Wasserstein distance remains relatively constant. It seems to be relatively independent from performance metrics, which are the highest for $\lambda_{\text{GP}} = 10$.

σ_{real}	soft entr	W-dist	l_{dx}
0	0.15	1.97	1.06
0.05	2.24	0.64	1.02
0.1	2.64	0.36	1.01
0.2	2.84	0.11	1.00

(a) Effect of σ_{real} on several quantities. 3 runs (Rep. 26)

λ_{GP}	fc	se	W-dist	l_{dx}
1	24.7%	2.2	0.41	1.12
5	30.3%	2.2	0.30	1.02
10	35.8%	2.2	0.36	1.01
20	29.3%	2.2	0.33	1.00

(b) Effect of gradient penalty λ_{GP} on WGAN metrics. 2 runs (Rep. 27)

Table 13: Effect of important hyperparameters to several WGAN quantities

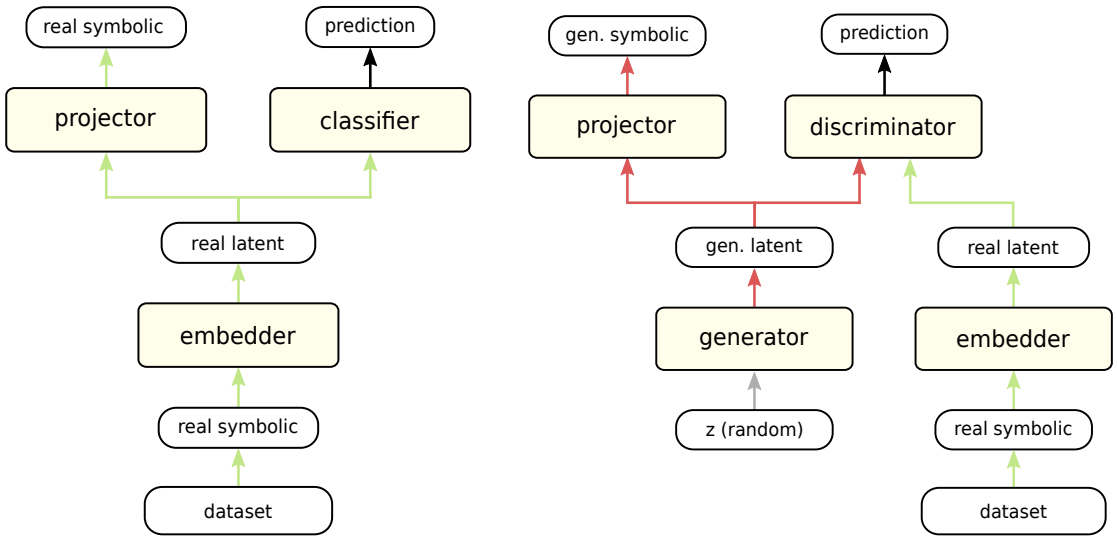
5.3 Learned embedding approach

This section builds on a similar GAN architecture as presented in the previous section, but tries to alleviate the problem of identifying real samples by their characteristic one-hot distribution in embedding space. To this end, an embedding of discrete symbols into a continuous space is learned on an auxiliary classification task and transferred into the GAN setting. Concretely, a Transformer classifier for LTL-SAT is trained, its lowermost layers are extracted and used to embed real instances. While this approach may result in an embedding that is better suited for processing with Transformers than the previous one in token probability space, an important property is lost: Generated instances can no longer be easily transformed back to a discrete form, which was possible by simply taking the *argmax* over the last axis previously. To reconstruct interpretable symbols, in fact a reverse transformation has to be learned. However such an operation can only be trained for instances where the correct discrete representation is known, which corresponds to the dataset of real instances. Consequently, the reverse transformation is never trained on generated symbols and it is well possible that the GAN may learn to construct formulas perfectly, but they cannot be observed since they are incorrectly projected back to symbolic space.
In the following subsection 5.3.1, the extensions to the existing Transformer GAN architecture are detailed. Two variants of extracting a learned embedding are tested: A *deep* one which uses the complete lower two layers of the trained classifier and a *shallow* one that only uses the very first embedding matrix during input processing. Section 5.3.2 then presents the results achieved with these approaches.

5.3.1 Architecture and training setup

The overall architecture is depicted in Figure 25. If differs from the direct one-hot architecture in several ways. Real samples are processed by an *embedder* before being handed to the critic. Generated samples are in turn transformed by a *projector* back into token probability space, where an *argmax* can be taken. Since this approach works on a latent representation, it can be chosen more freely, which corresponds to changes in the pretrained classifier architecture or the exact point of extraction (for the deep variant).

Critic The critic works similar to the direct one-hot approach. However, there, since real samples were unprocessed except for added noise, it had to perform initial input processing, which consists of applying an embedding transformation, scaling by $\sqrt{d_{\text{emb}}}$ and adding the positional encoding. With the approach in this section, all this is already performed by the learned embedder. Thereby, the critic's Transformer layers can be applied directly on its output. This is denoted as the "as-is" variant in the following. Alternatively, the critic can be equipped with an additional initial affine transformation, for example to allow for differences in embedding dimensionality. This is denoted by



(a) Pretraining architecture for learning an embedding

(b) Main architecture for GAN training

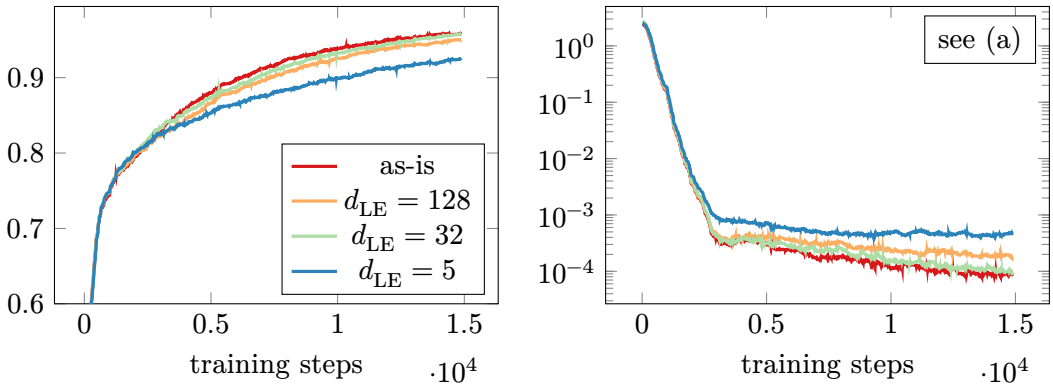
Figure 25: Components of the learned embedding architecture in pretraining and GAN training configuration

"upper projection" (or simply "upper", referring to the critic's position in the processing pipeline). Regarding an additional scaling by $\sqrt{d_{\text{emb}}}$, experiments showed that this consistently seriously degrades performance.

Generator The generator's architecture is almost unaltered from the direct approach, except for the omission of the final *softmax*. Depending on the design of the embedder, the generator does not even need its final affine transformation since its output can likewise be directly handed to the critic. Its presence is denoted by "lower projection".

Embedder As mentioned before, the embedder corresponds to the first few processing steps of a Transformer classifier trained beforehand. Importantly, in the GAN framework, there exists no objective for the embedder; its parameters are loaded from the fully trained classifier before GAN training and remain unchanged throughout it. Similar to the generator, for a "lower projection" variant, a final affine transformation is added. Note that classifier pretraining must be adapted accordingly to include respective additional transformations. All this is not relevant for the shallow embedding variant, where only the initial embedding matrix is learned. Still, in all cases, the embedder performs full input processing (applying the embedding matrix, scaling and adding the positional encoding). Note that in order to be better comparable to the direct one-hot embedding approach, a Gaussian noise is still added to the real samples before being handed to the embedder. However experiments showed that this has no significant effect on training.

Projector During pretraining of the classifier whose parts are later used for the embedder, the projector is trained simultaneously, but independently. Its objective is reconstructing the original token sequence while being provided with the output of the embedder (part of the classifier). Its input processing is similar to the critic: Except for the "upper projection" setting, where an additional affine transformation is applied, the input is fed directly to the Transformer layers. Its output processing is similar to that of the generator in the direct one-hot approach, where after a final affine transforma-



(a) training accuracy

(b) reverse projection crossentropy

Figure 26: Classifier pretraining for deep embedding. 2 runs (Rep. 28)

tion, a softmax to token probability space is applied. On this, the cross entropy towards original tokens is computed as loss. Note that similar to the embedder, it is provided with the real padding mask. Also note that similar to the critic, it is not provided with positional information at all. That successful reverse projection is possible shows that there is still enough positional information present in the embedder's output. In contrast to the embedder, further training of the projector is possible during GAN training since real instances are continuously provided for reference. However experiments showed that it is unnecessary since the reverse projection loss is very small shortly into pretraining already.

Training setup The embedding classifier is trained for 15k steps on `RPC50`. It has $n_l = 6$ layers, of which two are used as embedder in the deep variant. The projector similarly has 2 layers in the deep setting and none in the shallow setting. There, it must reconstruct the symbols by its single affine transformation. For all parts, the optimized transformer schedule from Section 4 is used. The GAN and WGAN variants trained afterwards use identical settings to the direct one-hot approach ($n_{lG} = 6$ generator layers, $n_{lC} = 4$ critic layers).

5.3.2 Results

Learning a deep embedding As first step, it is examined how different architectural variants perform during classifier pretraining. This includes the "as-is" variant with no additional affine transformations as well as "lower" and "lower+upper" variants with some additional ones. Also, reducing the embedding dimensionality at the embedder output is tested, which requires both transformations. This dimensionality is denoted by d_{LE} in the

d_{LE}	affine trans.	train err	val err	reverse CE
128	none	4.2	3.6	$< 1\text{e}{-4}$
128	lower only	4.8	3.8	$1.2\text{e}{-4}$
128	lower+upper	5.0	3.4	$1.7\text{e}{-4}$
32	lower+upper	4.2	3.2	$< 1\text{e}{-4}$
5	lower+upper	7.4	6.8	$4.5\text{e}{-4}$

Table 14: Classifier pretraining for deep embedding (validation error not smoothed). 2 runs (Rep. 28)

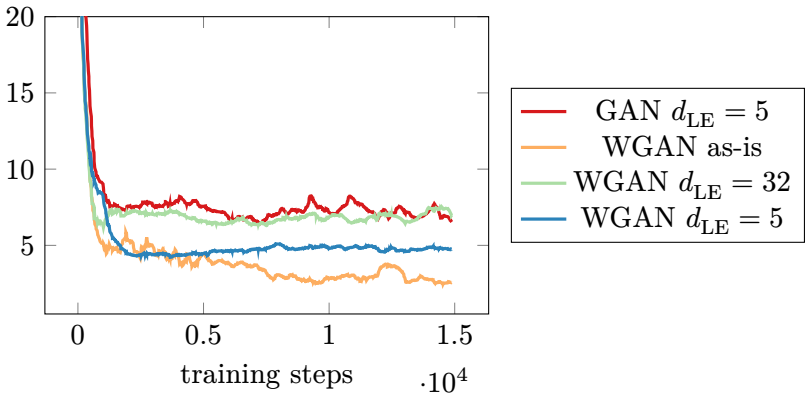
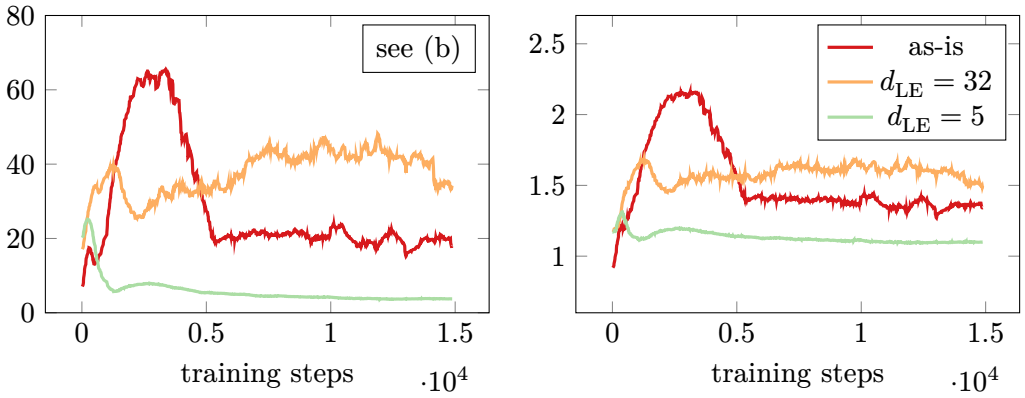


Figure 27: GAN and WGAN parse fragments with deep embedding. 2 runs (Rep. 29, Rep. 30)



(a) Wasserstein distance

(b) gradient length l_{dx}

Figure 28: WGAN quantities with deep embedding. 2 runs (Rep. 30)

following. Since the vocabulary contains $|V| = 24$ tokens, the next power of 2, $d_{\text{LE}} = 32$ as well as a theoretical binary representation, $d_{\text{LE}} = \log_2 32 = 5$ are tested.
As apparent from Table 14 and Figure 26, for pretraining, all variants perform relatively similar. Except for $d_{\text{LE}} = 5$ which seriously limits information flow, they achieve satisfyingly low error rates. Interestingly, the reverse projection is able to very quickly converge to virtually perfect reconstruction with a crossentropy below 10e−3 after only 1k steps for almost all variants ($d_{\text{LE}} = 5$ needs 2k). Note that this is not recognizable in Figure 26b due to the applied smoothing.

GAN training with deep embedding For each of the previously described pretrained variants, a standard GAN and a WGAN are trained, respectively. In all settings except for $d_{\text{LE}} = 5$, the standard GAN performed very poorly and no significant improvement in the metrics was noticeable at all. Neither low parse fragments nor a stable sequence entropy could be observed. While $d_{\text{LE}} = 5$ achieves a satisfying sequence entropy of 2.2, its parse fragments fail to consistently decrease and therefore no consistent formulas are output. The WGAN performs much better overall, with all variants achieving a stable 2.2 sequence entropy and a relatively stable number of parse fragments. Figure 27 depicts the performance during training. However "as-is" is the only one that successfully achieves acceptable parse fragments and a fully correct fraction above 1%. In Figure 28, the progress of WGAN-specific quantities are shown (for reasons of clarity, some variants

were omitted since they behaved similarly). Interestingly, in terms of both Wasserstein distance and gradient length l_{dx}, $d_{\text{LE}} = 5$ is the only one with curves relatively similar to the direct one-hot approach. All others variants show much higher values, where especially the far distance from target length 1 is concerning. While with an even higher gradient penalty λ_{GP}, the gradient length could be slightly reduced, the Wasserstein distance and overall performance were largely unaffected in additional experiments. This behavior is unexpected compared to the previous study in the direct one-hot approach (Table 13b). Also, there were critical performance differences between runs with identical parameters. For example, one WGAN run with $d_{\text{LE}} = 5$ and $\lambda_{\text{GP}} = 20$ achieved a fully correct rate of up to 5% (smoothed), while an identical run failed to even stabilize the parse fragments metric (with values above 20). Likewise, while the "as-is"-variant performed comparatively well in general, there were runs of it that did not achieve 1% fully corrects.

Shallow embedding For the shallow embedding variant, only the very first embedding matrix of the pretrained classifier is used. Due to implementation reasons, a separate classifier is trained to learn the embedding (Rep. 31), although it is architecturally identical to the "as-is" variant of the deep learned embedding with a total of $n_l = 6$ layers for classification. Note that the embedder and projector in this setting only consist of a single affine transformation, respectively. Additional "lower" and "upper" transformations therefore only affect the generator and critic. Note that in this setting, $d_{\text{LE}} = d_{\text{emb}}$ is implied. Table 15 presents results of different GAN and WGAN variants. Generally, GAN variants with shallow embedding perform better than their deep counterparts. Noticeable, all settings except only upper projection achieve a stable sequence entropy of 2.2. While they are able to lower the parse fragment count, no setting achieves a consistent fully correct count of 1%. Surprisingly, regarding the metrics in Table 15, the WGAN performs partially worse than the standard GAN. Some settings significantly degrade sequence entropy, which was not observed in any other experiment in this thesis. However, in contrast to the standard GAN, some settings achieve fully correct counts around 1%, namely the "as-is" and "lower+upper" variant.

Consequently, these experiments show that GAN training with a learned embedding is possible in principle and some correct formulas can be constructed, but in the current state it performs far below the level of the direct one-hot approach. It is not obvious whether this results from training in a high-dimensional, only weakly constrained latent space or from the discrepancy between actually generated instances and the real ones the projector was trained on, thus with reconstruction as a limiting factor. Regarding the latter, experiments were made with a setup where embedder and projector collaborate to reconstruct the latent representation output by the generator, but no beneficial effects on training could be observed. While using a learned embedding poses an interesting approach to GAN training in non-differentiable domains that could certainly be improved upon in the future, subsequent experiments resort again to the direct one-hot approach due to its high performance and ease of use.

affine trans.	*se*	*pf*
none	2.2	5.5
lower only	2.2	5.6
lower+upper	2.2	4.8
upper only	1.6	7.8

(a) GAN variants

affine trans.	*se*	*pf*
none	2.2	6.2
lower only	2.1	6.3
lower+upper	2.2	5.0
upper only	1.9	5.9

(b) WGAN variants

Table 15: GAN and WGAN variants with shallow embedding at 15k steps, values smoothed with $\alpha = 0.99$. 2 runs (Rep. 32, Rep. 33)

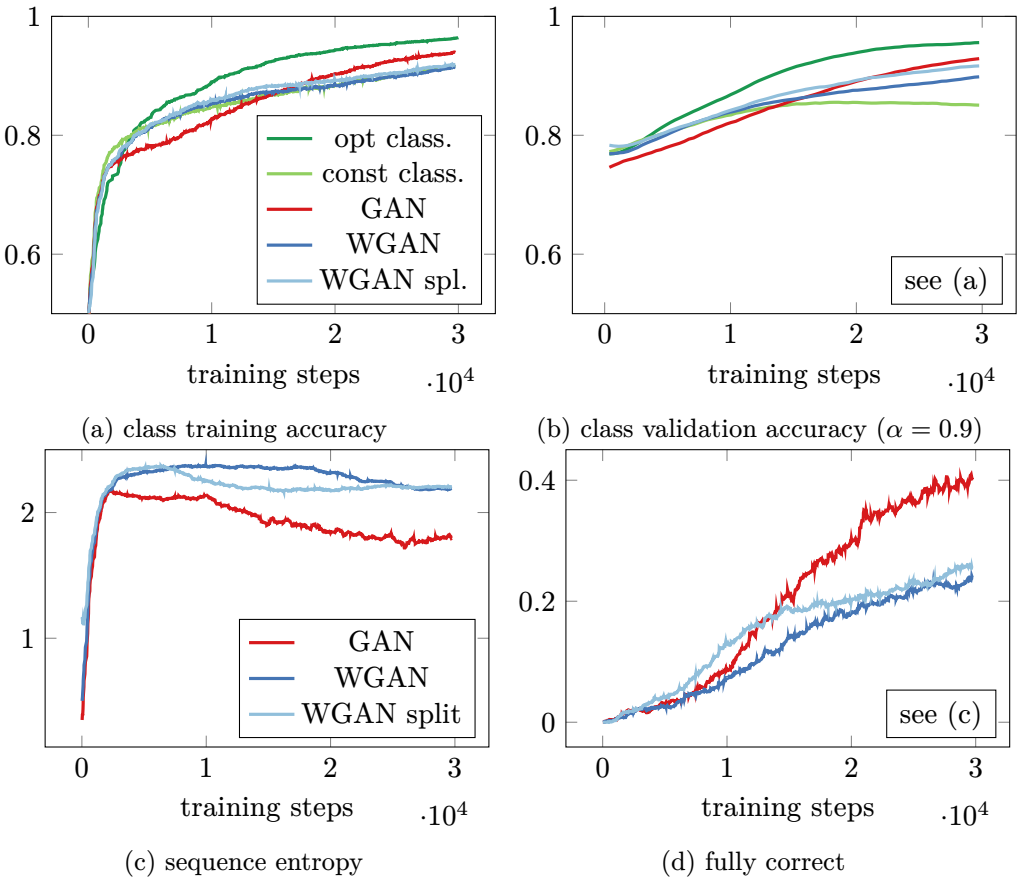
5.4 Additional class uncertainty objective

All preceding subsections focused on the fundamental target of generative adversarial networks: Imitating the real distribution of data instances. However, the general context of this thesis is LTL-SAT, so it would be desirable to make the GAN aware of classes, or concretely the satisfiability of a particular LTL formula. To this end, first, in Section 5.4.1 a classification objective on real instances is added to the critic and it is analyzed how this impacts GAN training. Then, in Section 5.4.2, the uncertainty of the critic with respect to the classes of generated samples is used as additional objective for the generator. The usability of such a method to synthesize new datasets that are harder to classify is subsequently demonstrated in Section 5.4.3. Finally, in Section 5.4.4, the critic is supplied with the classes of newly generated samples and the resulting closed-loop training dynamics are examined.

5.4.1 Additional classification objective for critic

Setup As first step, during GAN training, the critic is tasked simultaneously with classification of LTL formulas into satisfiable and unsatisfiable ones. The architecture is unaltered except that the critic's output dimensionality is increased from 1 to 2. On this second output, a crossentropy towards the actual class label is computed as loss. Naturally, this additional objective is only applied for real samples, where the label is known. The classification objective is weighted by a factor λ_{class}. Standard GANs proved to require only small factors, so their default value is left at $\lambda_{\text{class}} = 1$ (Rep. 36). WGANs on the other hand performed significantly worse on classification and require large values of e.g. $\lambda_{\text{class}} = 10$ to achieve a comparable accuracy (Rep. 37). This difference is likely due to the different type of loss applied to both outputs, where for standard GANs it is a crossentropy in both cases.

Results Classification and GAN performance is compared in Figure 29. As reference, also two pure classifiers are included: The "optimized classifier" (Rep. 34) uses the optimizer settings tuned for Transformers from Section 4, where the "constant classifier" (Rep. 35) uses the exact same parameters as the WGAN architecture with classification objective, but without any GAN objective. Its naming stems from the fact that in contrast to the optimized classifier, it only uses a constant learning rate. Regarding training accuracy, both GAN variants perform on par with the constant reference classifier and in terms of validation accuracy even outperform it by far, since the constant classifier seems to start overfitting the training set. The standard GAN can achieve significantly better validation accuracy than the WGAN, while both are outperformed by a large margin by the optimized classifier. Regarding GAN metrics, the standard GAN achieves very similar results than without any additional classification objective (compare Figure 19b). The WGAN however performs a bit worse in terms of the fully correct fraction. Note



(a) class training accuracy

(b) class validation accuracy ($\alpha = 0.9$)

(c) sequence entropy

(d) fully correct

Figure 29: Classification and GAN metrics for reference classifiers and (W)GAN variants with additional classification objective. 2 runs (Rep. 34, Rep. 35, Rep. 36, Rep. 37, Rep. 38)

that for all compared models, including the pure classifiers, the default Gaussian noise with $\sigma_{\text{real}} = 0.1$ was added to real samples. However this generally does not affect a pure classifier's performance at all.

WGAN with split heads To alleviate the performance loss of the WGAN, an architecture where the critic shares only some layers and uses distinct upper layers for the classification and critic targets, respectively, is tested. Table 16 lists classification and GAN metrics for a variable number of shared layers. As expected, the fully correct fraction increases with fewer shared layers. Sharing all layers also decreases classification accuracy. Since in following subsections, a gradient is backpropagated through the classification output, however, it is desirable to not reduce the number of shared layers too

shared layers	se	fc	val acc
0 / 4	2.2	31.8% (0.2)	89.9% (2.3)
2 / 4	2.2	26.6% (1.7)	92.5% (0.1)
3 / 4	2.2	24.9% (0.3)	92.1% (0.6)
4 / 4	2.2	24.0% (1.2)	90.5% (0.5)

Table 16: Different number of shared layers for WGAN with included classifier, 30k steps, standard deviations in parentheses. 2 runs (Rep. 39)

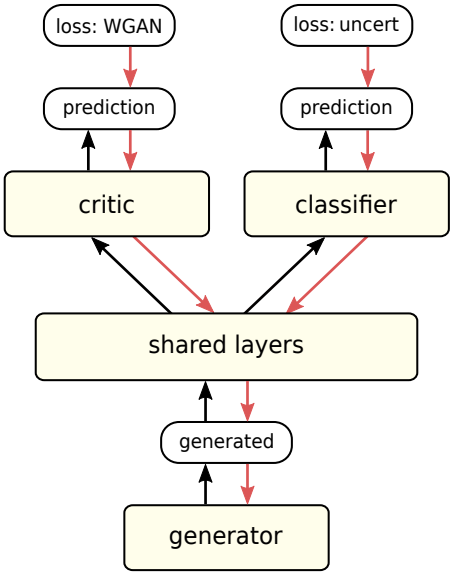


Figure 30: Data flow of generated samples and gradient backpropagation with WGAN and uncertainty losses for a split WGAN architecture

much, so that critic smoothness on that path is lost. In fact, experiments showed that the approach in Section 5.4.2 performed worse with no shared layers in the critic. As a result, a compromise of 3 shared layers out of 4 in total is used for all subsequent experiments. This setting is termed "WGAN split" (Rep. 38) in Figure 29 and while performing on par with the standard WGAN on GAN metrics, it achieves a significantly higher classification accuracy, rivaling the standard GAN variant.

5.4.2 Class uncertainty objective for generator

The goal of the approach presented here is to change the target of GAN training: Instead of just imitating an existing dataset, it is tested to what extend a generator can be trained to produce instances that are harder to classify than the original ones. In addition to the classification objective of the critic introduced in the previous section, a new objective is therefore added for the generator: the uncertainty of the critic with respect to the class of generated samples, which should be maximized. Note that while the generator works against the critic with this objective, the scenario is different than the two original GAN objectives: The critic classification objective is solely applied to real samples, for which the class labels are known. The critic is never trained for classification of generated samples directly, these are only scored regarding their authenticity. Nonetheless, the gradient of the class prediction of a generated instance can be backpropagated into the generator, which is utilized in this scenario. This setup is visualized in Figure 30. Importantly, both objectives for the generator, authenticity of samples and uncertainty of their class must be balanced, to which end a factor λ_{unct} is introduced to scale the uncertainty part. Starting from this subsection, standard GANs are not considered anymore due to their low sequence entropy. All following experiments use a WGAN with included classifier and split heads (3 shared, 1 distinct layer).

Uncertainty measures A natural choice to determine uncertainty is the entropy. For a class prediction p of the classifier, it reads $H(p) = -p \cdot \log(p) - (1-p) \cdot \log(1-p)$. Consequently, the term $-\lambda_{\text{unct}} H(p)$ is added to the generator's loss function. $H(p)$ is maximized at $p = 0.5$, so the generator is encouraged to produce instances which "confuse"

uncertainty obj.	pretrained	train acc	val acc	*se*	*fc*
none	no	93.3%	91.7%	2.2	25.8%
entropy	no	92.0%	90.7%	2.2	11.2%
absolute	no	91.5%	91.0%	2.2	10.9%
entropy	yes	95.2%	93.2%	2.2	12.4%
absolute	yes	94.6%	93.1%	2.2	12.4%

Table 17: Comparison of models trained with uncertainty objective, evaluated at 30k steps for non-pretrained variants (Rep. 38, Rep. 40, Rep. 41) and 15k steps for pretrained variants (Rep. 42), validation accuracy smoothed with $\alpha = 0.9$. 2 runs

the critic classifier and have it assign almost equal probability to both classes. For a WGAN architecture, $\lambda_{\text{unct}} = 2$ is chosen as default, but changing this parameter did not significantly affect results.
The entropy, however, becomes unhandy to compute for values close to 0 and 1. Therefore, a pragmatic alternative measure for (un)certainty is used: the absolute value of the classification logit l. This corresponds to the immediate output of the critic classifier before applying a logistic sigmoid to transform it into a probability ($p = \sigma(l)$). Logit values close to zero lead to predictions around 0.5, while positive values correspond to predictions closer to 1. Consequently, as as alternative to the entropy loss, the term $\lambda_{\text{unct}}|l|$ is added to the generator's loss function. For this variant, a default value of $\lambda_{\text{unct}} = 0.5$ in used.

Impact on training performance Adding any of the two versions of uncertainty objectives leads to a noticeable performance loss during GAN training for both parse metrics and classification accuracy on real samples. Table 17 shows the difference between split WGAN models with classification objective and the same with additional uncertainty objective. The sequence entropy remains for all variants. Due to the negative effects of the uncertainty objective, also two variants were pretrained without it (thereby being identical to the scheme from the previous section and the first row in Table 17) for 30k steps and afterwards trained with added uncertainty objective for further 15k steps. Interestingly, while this results in a comparable performance hit in the fully correct fraction, the critic's classification accuracy is not deteriorated at all and even slightly improves during further training. Consequently, for all further experiments with an uncertainty objective, a pretraining for 30k steps without it is employed.

Resulting classification uncertainty To determine if the incorporation of an uncertainty objective for the generator indeed increases uncertainty in the critic classifier, three different quantities are depicted in Figure 31 during the course of training for both pretrained variants: Firstly the mean classification entropy of real samples, which the critic classifier is trained to minimize (albeit not directly). It starts fairly low due to the previously observed high class certainty achieved during pretraining and is slightly decreased during further training as expected. The final entropy of roughly 0.13 corresponds to a probability of approximately 97% in a binary setting. Note that the actual accuracies for training (94.5%) and validation (93.1%) are much lower.
Secondly, the classification entropy of unaltered generated samples is depicted, which, after a short rise in the beginning, remains relatively constant. This shows several things: It already starts very high, which means the critic classifier is generally uncertain about generated samples, even without the additional objective. This again shows how differently the critic behaves on generated and real samples. Note that the maximally achievable

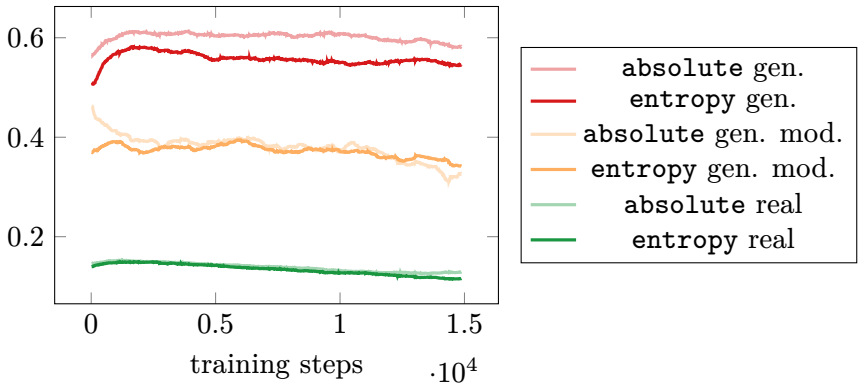


Figure 31: Classification entropy for different kinds of samples on entropy and absolute objective variants, $\alpha = 0.97$. 2 runs (Rep. 42)

entropy for equally probable predictions is $-\log \frac{1}{2} \approx 0.693$. Apart from that, the initial rise in the metric shows that it is very easy for the generator to shift to samples with a higher classification entropy. While it remains high throughout training, there remains a significant margin to the maximum value which means it is not trivial to max it out. This fits previous observations in reduced parse metrics, which indicates that the generator indeed has to sacrifice parts of its WGAN objective for the uncertainty one. An entropy of 0.6 roughly corresponds to a probability of 71%.
Lastly, also the classification entropy for generated samples that are processed to look more like real ones is depicted in Figure 31. They are *argmaxed* and Gaussian noise is added to them, so they should be indistinguishable from real samples locally. A similar comparison has been conducted previously in Figure 24. Values lie between those of real and unaltered generated samples, which shows that a significant amount of uncertainty indeed comes from the shape of the token probability distribution. However the gap to real samples is still large (0.4 corresponds to approximately 86% probability), which means that the intended goal of constructing samples that are harder to classify than the original ones was indeed reached. In the next section, this is validated on complete generated datasets.

5.4.3 Application for generating challenging datasets

In this subsection, similar to Section 5.2.3, new datasets are sampled from trained GAN models and the classification accuracies of reference classifiers trained on those sets are compared. The goal is to use the uncertainty objective introduced in the previous subsection 5.4.2 to produce a dataset with instances that are harder to classify than those from the original `RPC50` dataset.

Generated datasets WGAN variants with both entropy and absolute error as losses are trained similarly to previous section for 15k steps after with pretraining for 30k steps. From the trained models, similar to Section 5.2.3, samples are taken and processed to constitute datasets `Uncert-e` (from the entropy model) and `Uncert-a` (from the model with absolute error loss). Both sets contain 400k instances, which are balanced in size with respect to their class label. Figure 32 depicts the size distribution of the resulting `Uncert-e` set. Additionally, mixed datasets which contain random 200k instances from `RPC50` and 200k from the respective `Uncert` sets are constructed and denoted by `Mixed-e` and `Mixed-a`, respectively.

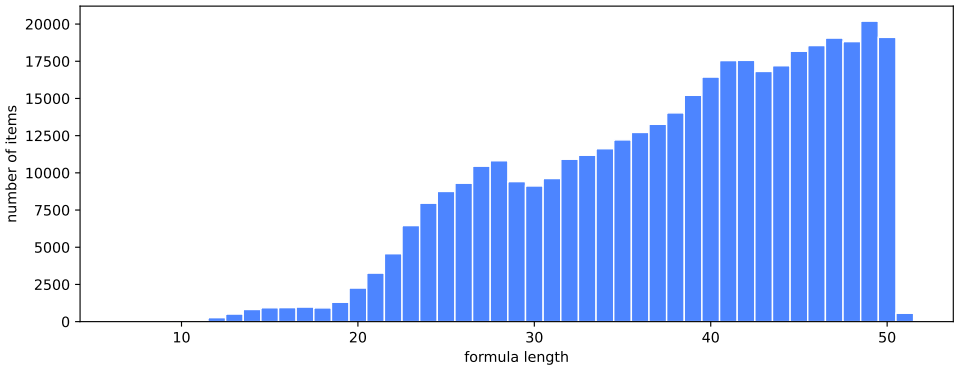


Figure 32: `Uncert-e` dataset size distribution (average size 38.0). Rep. 44

Results Similar to Section 5.2.3, several small reference Transformer classifiers are trained on the generated datasets. They have $n_l = 4$ layers and use the default Transformer optimization setup from Section 4. The classifier trained on `RPC50` serves as reference with 94.8% accuracy on the validation split of the same set. Training on the `Uncert` sets however allows the classifier to achieve only 91% and 90.5% accuracy on the validation splits of the respective datasets. Experiments conducted with 50k training steps also showed no significant improvement. Consequently, the datasets produced by the WGAN with added uncertainty objective are indeed harder to classify than the original `RPC50` dataset.

To validate this, classifiers were additionally trained on the `Mixed` sets, which contain equal parts of existing and newly generated hard formulas. They are tested on the validation splits of both the original and generated sets and show a strong difference: The accuracy on the `Uncert` sets is 5.1 respective 4.5 percent points lower than on `RPC50`. Additionally, the performance on the original dataset is not deteriorated at all and even slightly higher after training on the `Uncert-e` set. This shows that training GANs with the presented uncertainty objective yields instances that are not only hard to classify locally for the respective critic classifier, but also for freshly trained Transformer classifiers in general.

5.4.4 Incremental learning

The approaches considered so far do not train the critic classifier at all on generated samples. They are only used in the WGAN objective of the critic. While this already led to harder-to-classify datasets as examined in the previous section, the remaining question is what happens when the critic classifier is given the chance to actively adapt to generated, confusing samples.

trained on	tested on	accuracy
`RPC50`	`RPC50`	94.8% (0.3)
`Uncert-e`	`Uncert-e`	91.0% (0.5)
`Mixed-e`	`Uncert-e`	**90.2%** (0.9)
`Mixed-e`	`RPC50`	**95.3%** (0.4)

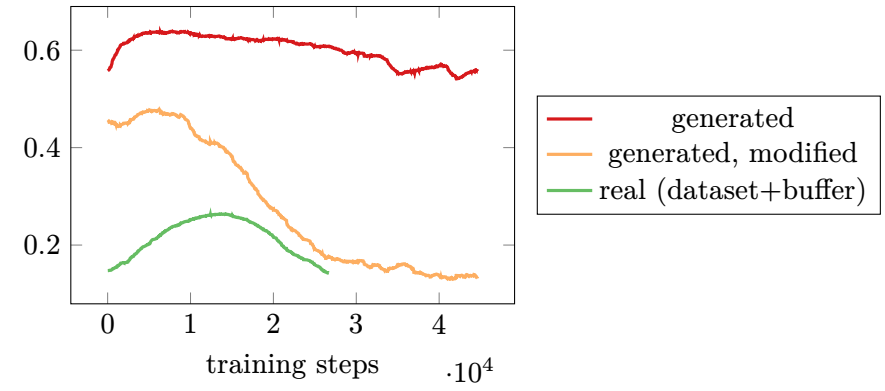
trained on	tested on	accuracy
`Uncert-a`	`Uncert-a`	90.5% (0.5)
`Mixed-a`	`Uncert-a`	89.6% (0.4)
`Mixed-a`	`RPC50`	94.1% (0.4)

Table 18: Performance of classifiers trained on datasets generated with uncertainty objectives at 30k steps, not smoothed, with standard deviations in parentheses. 5 runs (Rep. 45)

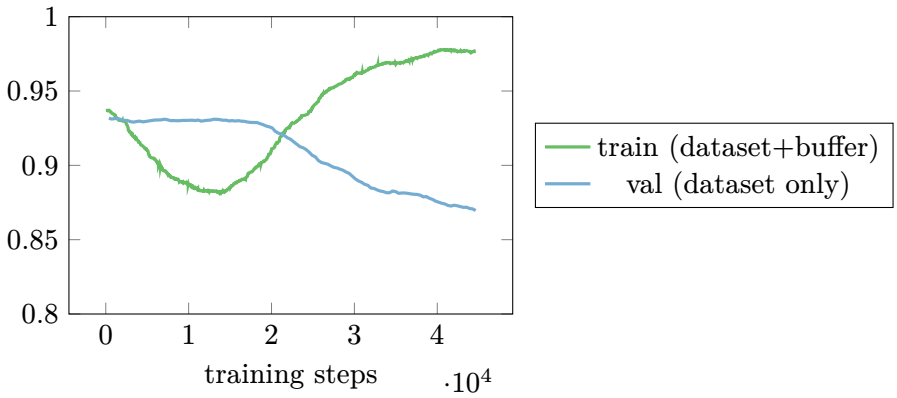
Setup The training process is expanded as follows: Every few steps (adjustable by `gan_trainsteps_process_interval`, default 5), a batch of generated samples is obtained from the GAN. These are transferred to their symbolic form (*argmaxed* and looked up in the vocabulary) and passed to an algorithmic LTL satisfiability solver (`aalta` by default). Instances are filtered to equal proportions of satisfiability and, together with their computed label, input into a buffer. Reservoir sampling is implemented with this buffer, so that even though it has a fixed size, at every point in time, every generated instance has an identical probability of being contained. The critic is now trained on a mixture of instances from the original dataset and the buffer. Notably, both sources are now considered real samples. The mixing parameter shifts over time, transitioning the real samples that are input to the critic from coming only from the dataset to coming only from the buffer. This significantly changes the target of GAN training: Instead of imitating an external dataset, the generator now tries to create instances similar to those it created in the past, but dissimilar in that they should be are harder to classify for the current critic. Importantly, since the critic is continuously trained on the buffer instances, the generator has to find new ways of making them hard to classify. Ideally, this would result in a dynamic setting with the current output distribution of the generator currently changing, but never being identical to a previous timestep, thereby creating a constantly diversifying set of formulas.

Results Like the variants with only fixed uncertainty objective from the previous two sections, a model with only critic classification objective is pretrained for 30k. For actual training, only a split WGAN with absolute error as uncertainty objective is used here, since for variants based on entropy, over- or underflowing values in the objective computation aborted training halfway through training. Note this is implementation-specific and could certainly be fixed with an improved calculation. For the buffer holding labeled generated samples, a size of 20k instances is chosen, and it is completely filled after around 11.6k training steps. The mixture transition period is chosen from step 1 to 20k, during which the mixture parameter linearly increases towards only using the buffer as real training samples. Figure 33 shows the entropy and accuracy progression during training of a respective model. The differences to the previous approach without an adjusting critic (Section 5.4.2) and in particular Figure 31 are immediately visible: While the classification entropy for unaltered generated samples remains largely constant, the entropy for modified ones decreases down to the same value as original real samples. This shows that the critic classifier indeed learns to precisely classify those samples and the generator is not able to keep up the uncertainty. Importantly, this also implies that no dynamics of a constantly shifting distribution emerge. While obviously the generator distribution (meaning its symbolic output formulas) differs from the original real dataset, once the critic classifier is familiar with it, it does not change further. The generator is apparently stuck on a fixed generative distribution.

Apart from this, the critic classifier's behavior is as expected: Initially, classification entropy rises and respectively training accuracy falls, as more and more samples are taken from the generated buffer, which follow a different distribution. Starting from 20k steps, all samples originate from the buffer. Until this point, validation accuracy on the original dataset remains stable, but afterwards it quickly deteriorates, which is expected due to the different distribution. Interestingly, until the end of training, training accuracy rises much higher than on the dataset, which is counter-intuitive given the results in Section 5.4.3. Classification entropy in fact becomes so low that the calculation underflowed, which is why in Figure 33a its plotting stops around 28k steps.



(a) entropies ($\alpha = 0.97$)



(b) accuracies (train $\alpha = 0.97$, val $\alpha = 0.7$)

Figure 33: Incremental learning. 2 runs (Rep. 46)

To summarize, results from Section 5.4 showed that it is possible to alter the target distribution of GAN training towards instances that are harder to classify for learned solvers. This holds globally, even solvers trained from scratch achieve lower final accuracies on such instances. In contrast, local dynamics of critic and generator continuously changing the generative distribution in a closed-loop incremental learning setting could not be observed so far.

6 Related work

Temporal logics Temporal logics were firstly introduced to computer science by [81]. LTL satisfiability has been studied extensively [66, 86, 89] and several solving procedures have been proposed [65, 90, 10]. Related research topics include synthesis [30, 29, 12, 27], monitoring [8, 31, 72] and model checking [17, 18]. Temporal specification patterns have been identified by several authors [22, 25, 44, 80]. First applications of deep learning to this topic are [39] for LTL trace generation and [88] for circuit synthesis.

Deep learning for symbolic problems A very active application area of deep learning is automated theorem proving [98, 70, 78, 71, 32, 50, 55, 7, 45, 100, 82, 64, 84, 77, 83]. Approaches also exist for satisfiability checking of propositional logic [92, 102, 91] and quantified boolean formulas (QBF) [63] as well as satisfiability modulo theories (SMT) solving [6]. Another closely related application domain is symbolic math [62, 104, 2, 84].

GANs in the sequence domain In discrete domains, GANs have been applied especially for text generation with a reinforcement learning setting to circumvent non-differentiability [15, 103, 14, 68, 28, 67, 38, 107]. As alternative, a Gumbel-Softmax distribution has been used to describe the sampling operation [61, 76]. Objective functions inspired by the GAN framework have also been incorporated into language models for text generation to alleviate problems arising from the discrepancy between training and inference [48, 35]. An approach with a pretrained embedding is pursued by [60]. The authors of the WGAN-GP scheme [37] also showed that a direct generation without sampling can work. Approaches closely related to this thesis are are [46] and [106], which also combine Transformers in an adversarial setting. The former rely on a Gumbel-Softmax and the latter extract a style code from reference examples. Transformers and GANs have also been combined in the domain of computer vision [97, 52, 47]. There, GANs in general have already been used for data augmentation [1, 13].

7 Conclusion and future work

This thesis tackled the satisfiability problem of temporal specifications in linear-time temporal logic with deep learning approaches based on the Transformer architecture. Firstly, a new data generation process based on a system of specification patterns was proposed, that is able to construct a dataset of highly constrained formulas with a large probability of being unsatisfiable. Such datasets may serve as basis for future applications of learning approaches to temporal logic. Secondly, LTL satisfiability classifiers were trained end-to-end using standard and universal Transformers. The latter achieved a remarkable classification accuracy of more than 99.6% on LTL formulas up to length 100 and outperformed standard Transformers by a large margin, underlining the importance of iterative architectures for solving algorithmic problems. This performance demonstrates that learned models can indeed be used as viable alternative to algorithmic solvers. A natural next goal for future work is the incorporation of a learned component into an algorithm to benefit from both, guaranteed correctness and fast predictions.
This thesis also approached the fundamental problem of obtaining reliable training data in symbolic reasoning domains in general. To this end, the capabilities of standard GANs and Wasserstein GANs equipped with Transformer encoders to generate valid and meaningful temporal specifications were studied. It was shown that both can be trained directly on one-hot encoded tokens when adding Gaussian noise, without autoregression and sampling. In particular the Wasserstein GAN produced a large fraction of syntactically consistent formulas and exhibited an impressive degree of variety in its output. It was shown that using this GAN, a new dataset can be synthesized from only few provided examples, which enabled the successful training of a neural classifier by using the GAN-generated data as full substitute for the original data. This approach could be applied to various other symbolic reasoning domains where only small data collections exist and make the training of learned solvers possible there. Additionally, a solver component was incorporated directly into the GAN to study the extend to which such an architecture can learn to autonomously synthesizing more difficult problems. From this, a dataset was created that proved to be indeed significantly harder to solve for newly trained Transformer classifier than the original dataset. This poses an interesting view on the application of learned solvers to symbolic problems in general: Similar future approaches could be used to identify weaknesses in neural solver architectures and potentially enable the training of solvers completely from scratch, without providing any initial training data at all.

8 Reproducibility

The implementation is contained in the Python module `dlsgs` ("Deep LTL-SAT generator and solver"). Refer to the `README` file for installation instructions. All following reproductions have been tested using Python 3.7.

Dataset generation

Reproduction 1 (Dataset `RPC100-raw`)

```
cd datasets
python -m dlsgs.data_generation.generator -od RPC100 --problem ltl
    --subproblem decision+witness --allow-unsat --frac-unsat None
    --include-solve-time --splits all_gen:1 -nv 10 -ne 3500000 -ts 1-100
    --timeout 2 --formula-generator spec_patterns
cd RPC100
python -m dlsgs.utils.update_dataset all_gen.txt unique - | python -m
    dlsgs.utils.update_dataset - relaxed_sat all_raw.txt
```

Reproduction 2 (Dataset `RPC100`)

```
cd datasets/RCP100
python -m dlsgs.utils.update_dataset all_raw.txt balance_per_size all.txt
python -m dlsgs.utils.update_dataset all.txt
    shuffle+split=train:8,val:1,test:1
```

Reproduction 3 (Table 1 on page 19)

Simple generation process, equal operator distribution

```
python -m dlsgs.data_generation.generator -od datasets/simple_equal
    --problem ltl --subproblem decision+witness --allow-unsat
    --frac-unsat None --include-solve-time --splits all:1 --token-dist
    'ap=!,false=0,true=0,not=1,F=1,G=1,X=1,equiv=0,implies=1,xor=0,R=0,
    U=1,W=1,M=0,and=1,or=1' -nv 10 -ne 20000 -ts 1-100 --timeout 2
```

Simple generation process, many ands

```
python -m dlsgs.data_generation.generator -od datasets/simple_manyands
    --problem ltl --subproblem decision+witness --allow-unsat
    --frac-unsat None --include-solve-time --splits all:1 --token-dist
    'ap=!,false=0,true=0,not=4,F=2,G=2,X=3,equiv=0,implies=1,xor=0,R=0,
    U=2,W=2,M=0,and=15,or=1' -nv 10 -ne 20000 -ts 1-100 --timeout 2
```

Concatenation of dac patterns

```
python -m dlsgs.data_generation.generator -od
    datasets/dac_pattern_concat --problem ltl --subproblem
    decision+witness --allow-unsat --frac-unsat None
    --include-solve-time --splits all:1  -nv 10 -ne 20000 -ts 1-100
    --timeout 2 --formula-generator dac_patterns
```

Rich pattern concatenation without groundings

Modify `dlsgs/generation/spec_patterns.py: SpecPatternGen.run()` to never call `SpecPatternGen.ground()`, then

```
python -m dlsgs.data_generation.generator -od
    datasets/rich_pattern_concat_nogroundings --problem ltl --subproblem
    decision+witness --allow-unsat --frac-unsat None
    --include-solve-time --splits all:1 -C 'no groundings' -nv 10 -ne
    20000 -ts 1-100 --timeout 2 --formula-generator spec_patterns
```

For rich pattern concatenation with groundings, see `RPC100-raw`.

For the final table, for each dataset individually, call

```
python -m dlsgs.utils.dataset_hist <path>/all.txt
```

Reproduction 4 (Figure 8 on page 22, Figure 9 on page 22)

```
python -m dlsgs.utils.analyze_dataset datasets/RPC100/all_raw.txt
    formula,sat_relaxed
```

Reproduction 5 (Figure 10 on page 22, Figure 11 on page 22)

```
python -m dlsgs.utils.analyze_dataset datasets/RPC100/all.txt
    formula,sat_relaxed
```

LTL-SAT classifier

A solver can be trained by running

```
python -m dlsgs.train.solver --run-name NAME --params-file
    configs/CONFIG.json
```

with a JSON file containing non-default parameters. In the following, parameters to be written into these config files are specified.

Reproduction 6 (LTL trace generation on dac concat set)
Train with

```
"d_embed_enc" : 128,    "d_embed_dec" : 128,
"num_layers"  : 8,      "num_heads"   : 8,
"d_ff"  : 1024,         "batch_size"  : 512,
"epochs"      : 39,     "ds_name"       : "DeepltlDacConcat",
"witness"       : true, "only_sat"      : true
```

For tree PE, add

```
"tree_pe" : true
```

Evaluated with

```
python -m  dlsgs.train.solver --test --run-name <name> --batch-size 512
    --samples 50000 --beam-size 4
```

Reproduction 7 (LTL trace generation on `RPC100` set)
Change Rep. 6 by

```
"epochs"      : 71,     "ds_name"       : "RPC100"
```

Reproduction 8 (Aggregation full Transformer architecture)

```
"d_embed_enc" : 128,     "d_embed_dec" : 128,
"num_layers"  : 8,       "num_heads"   : 8,
"d_ff"  : 1024,          "batch_size"  : 1024,
"epochs"      : 40,      "ds_name"       : "RPC100"
```

Reproduction 9 (Aggregation encoder only)

```
"implementation": "enc",        "d_embed_enc" : 128,
"num_layers"  : 8,       "      num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 100,            "ds_name"       : "RPC100",
"enc_accumulation" : "first"
```

For the other architectures, change `enc_accumulation` key to `mean-before` or `mean-after`, respectively

Reproduction 10 (Hyper parameters)
Base parameters

```
"implementation": "enc",        "d_embed_enc" : 128,
"num_layers"  : 6,              "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 40,             "ds_name"       : "RPC100",
"enc_accumulation": "mean-after"
```

Change accordingly. For batch size changes, adapt epoch count accordingly.

Reproduction 11 (Hyper parameters long)
Change Rep. 10 by

```
"epochs" : 80
```

Reproduction 12 (Universal Transformer encoder) Base parameters:

```
"implementation": "univenc",    "d_embed_enc" : 128,
"num_heads"   : 8,              "d_ff"  : 1024,
"batch_size"  : 1024,           "epochs"      : 40,
"ds_name"       : "RPC100",     "enc_accumulation": "mean-after",
"ut_base_layers"  : 1,          "ut_max_iterations": 9,
"ut_gradient_method" : "tape"
```

Change accordingly. For batch size changes, adapt epoch count accordingly. For higher iteration count, e.g. starting from 10 iterations, use

```
"ut_gradient_method" : "checkpoint"
```

Reproduction 13 (UT long training)
Change Rep. 12 by

```
"epochs"      : 80
```

Change accordingly. For batch size changes, adapt epoch count accordingly.

Reproduction 14 (UT with iteration curriculum)
Change Rep. 13 by

```
"ut_iterations" : "schedule",    "ut_max_iterations" : "20"
```

Reproduction 15 (UT with preceding and successive layers)
Adapt Rep. 13 by setting the parameters `ut_pre_layers`, `ut_max_iterations` and `ut_post_layers` accordingly.

Reproduction 16 (UT with intermediate evaluation)
Adapt Rep. 12 by adding the parameter

```
"ut_test_iterations" : [ 6, 9, 16, 20]
```

Reproduction 17 (UT with random training iterations)

```
"implementation": "univenc",     "d_embed_enc" : 128,
"num_heads"    : 8,              "d_ff"  : 1024,
"batch_size"   : 1024,            "epochs"      : 40,
"ds_name"        : "RPC100",             "enc_accumulation": "mean-after",
"ut_base_layers"   : 1,          "ut_max_iterations": 20,
"ut_test_iterations" : [ 6, 9, 16, 20],
"ut_stop_base" : 0.0833,         "ut_stop_map_certainty" : false
```

GANs

A GAN can be trained by running

```
python -m dlsgs.train.gan --run-name NAME --params-file
    configs/CONFIG.json
```

Similarly to the solver reproductions, in the following, parameters for the JSON config file are specified.

Reproduction 18 (GAN default model)

```
"d_embed_enc"    : 128,
"num_heads"    : 8,                      "d_ff"           : 1024,
"batch_size"   : 1024,                   "epochs"         : 80,
"ds_name"        : "RPC100",             "max_encode_length" : 51,
"objectives"     : "gan",                "num_layers" : 4,
"gan_generator_layers" : 6,              "gan_critic_steps" : 2,
"gan_critic_target_fn" : "sigmoid-log", "gan_critic_target_mode" : "one-minus",
"gan_generator_target_fn" : "sigmoid-log",
"gan_generator_target_mode" : "direct"
```

Reproduction 19 (WGAN default model)

```
"d_embed_enc"    : 128,
  "num_heads"    : 8,                    "d_ff"           : 1024,
  "batch_size"   : 1024,                 "epochs"         : 80,
  "ds_name"      : "RPC100",             "max_encode_length" : 51,
  "objectives"   : "gan",                "num_layers" : 4,
  "gan_generator_layers" : 6,            "gan_critic_steps" : 2,
  "gan_critic_target_fn" : "logits",     "gan_critic_target_mode" : "direct",
  "gan_generator_target_fn" : "logits", "gan_generator_target_mode" : "direct",
  "gan_gradient_penalty" : 10.0
```

Reproduction 20 (Dataset `RPC50-10k`)

```
cd datasets/RPC100
mkdir ../RPC50-10k
python -m dlsgs.utils.update_dataset train.txt limit_length=50 - |
    python -m dlsgs.utils.update_dataset - shuffle+split=train_len50:1
head -n 10000 train_len50.txt > ../RPC50-10k/train.txt
```

Reproduction 21 (WGAN on `RPC50-10k`)

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 512,
"epochs"      : 790,            "ds_name" : "RPC50-10k",
"val_set" : "datasets/RPC100",   "max_encode_length" : 51,
"objectives" : "gan",           "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_critic_steps" : 2,
"gan_critic_target_fn" : "logits",      "gan_critic_target_mode" : "direct",
"gan_generator_target_fn" : "logits",   "gan_generator_target_mode" : "direct",
"gan_gradient_penalty" : 10.0
```

Reproduction 22 (Dataset `Generated-raw` obtained by sampling from Rep. 21)
Make sure that the `load_from` parameter is set to the run name that was used for training the model.

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 512,
"epochs"      : 100,            "ds_name": "RPC50-10k",
"val_set" : "datasets/RPC100",  "max_encode_length" : 51,
"objectives" : "",              "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_critic_steps" : 0,
"load_from" : "<run name here>", "gan_trainsteps_process_interval" : 1,
"gan_process_generated_samples" : true,
"gan_filter_generated_entropy_threshold" : 0.0,
"gan_save_valid_samples" : true,        "gan_solve_valid_samples" : true,
"gan_balance_valid_samples" : false,     "gan_solve_tool" : "aalta"
```

Copy the file `generated_samples.txt` to `datasets/Generated/raw.txt`

Reproduction 23 (Dataset `Generated`)

```
cd datasets/Generated
python -m dlsgs.utils.update_dataset raw.txt unique - | python -m
    dlsgs.utils.update_dataset - balance_per_size - | python -m
    dlsgs.utils.update_dataset - shuffle+split=all_random.txt
head -n 400000 all_random.txt > train.txt
head -n 10000 all_random.txt > val.txt
```

Reproduction 24 (Reference classifiers for Section 5.2.3)
Note: Even though here only a classification objective is used, the GAN training code is used. Be sure to adjust the dataset name and epoch count to match 50k steps.

```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024,         "batch_size"  : 1024,
```

```
"epochs"      : 129,    "ds_name"       : "Generated",
"val_set" : "datasets/RPC100", "max_encode_length" : 51,
"objectives" : "class",         "num_layers" : 4,
"gan_critic_steps" : 1
```

Reproduction 25 (GAN and WGAN hyperparameter comparison)
Adjust the parameters of Rep. 18 and Rep. 19, respectively.

Reproduction 26 (Different additive noise levels) Figure 24 on page 42]
Adapt Rep. 18 and Rep. 19 by setting the parameter `gan_sigma_real` to the desired value.

Reproduction 27 (Different gradient penalties for WGAN)
Adapt Rep. 19 by setting the parameter `gan_gradient_penalty` to the desired value.

Reproduction 28 (Deep learned embedding)
Parameters for the "as-is" variant are provided in the following. Change `gan_latent_lower_proj`, `gan_latent_upper_proj` and `gan_latent_dim` accordingly for the other variants.

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 40,             "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "class,embed,project",   "num_layers" : 4,
"gan_generator_layers" : 0,     "gan_embedder_layers" : 2,
"gan_projector_layers" : 2,     "gan_critic_steps" : 1,
"gan_latent_upper_proj" : false,        "gan_latent_lower_proj" : false,
"gan_latent_dim" : 128
```

Reproduction 29 (GAN with deep learned embedding)
Parameters for the "as-is" variant are provided in the following. Change `gan_latent_lower_proj`, `gan_latent_upper_proj` and `gan_latent_dim` accordingly for the other variants, similarly to Rep. 28. Be sure to adjust the loading name to match the learned embedding run.

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 40,             "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "gan",           "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_embedder_layers" : 2,
"gan_projector_layers" : 2,     "gan_critic_steps" : 2,
"gan_critic_target_fn" : "sigmoid-log", "gan_critic_target_mode" : "one-minus",
"gan_generator_target_fn" : "sigmoid-log",
"gan_generator_target_mode" : "direct",
"gan_latent_upper_proj" : false,        "gan_latent_lower_proj" : false,
"load_from" : "<learned embedding>",    "load_parts" : "embedder,projector"
```

Reproduction 30 (WGAN with deep learned embedding)
Parameters for the "as-is" variant are provided in the following. Change `gan_latent_lower_proj`, `gan_latent_upper_proj` and `gan_latent_dim` accordingly for the other variants, similarly to Rep. 28. Be sure to adjust the loading name to match the learned embedding run.

```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024,         "batch_size"  : 1024,
"epochs"      : 40,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "gan",           "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_embedder_layers" : 2,
"gan_projector_layers" : 2,     "gan_critic_steps" : 2,
"gan_critic_target_fn" : "logits",      "gan_critic_target_mode" : "direct",
"gan_generator_target_fn" : "logits",   "gan_generator_target_mode" : "direct",
"gan_gradient_penalty" : 10.0,          "gan_latent_upper_proj" : false,
"gan_latent_lower_proj" : false,        "gan_latent_upper_scale_magic" : false,
"gan_latent_upper_add_pe" : false,      "load_from" : "<learned embedding>",
"load_parts" : "embedder,projector"
```

Reproduction 31 (Shallow learned embedding)

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 40,             "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "class,embed,project",   "num_layers" : 6,
"gan_generator_layers" : 0,     "gan_embedder_layers" : 0,
"gan_projector_layers" : 0,     "gan_critic_steps" : 1
```

Reproduction 32 (GAN with shallow learned embedding)
Parameters for the "as-is" variant are provided in the following. Change `gan_latent_lower_proj` and `gan_latent_upper_proj` accordingly for the other variants. Be sure to adjust the loading name to match the learned embedding run.

```
"d_embed_enc" : 128,            "num_heads"   : 8,
"d_ff"  : 1024,                 "batch_size"  : 1024,
"epochs"      : 40,             "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "gan",           "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_embedder_layers" : 0,
"gan_projector_layers" : 0,     "gan_critic_steps" : 2,
"gan_critic_target_fn" : "sigmoid-log", "gan_critic_target_mode" : "one-minus",
"gan_generator_target_fn" : "sigmoid-log",
"gan_generator_target_mode" : "direct",
"gan_latent_upper_proj" : false,        "gan_latent_lower_proj" : false,
"load_from" : "<learned embedding>",    "load_parts" : "embedder,projector"
```

Reproduction 33 (WGAN with shallow learned embedding)
Parameters for the "as-is" variant are provided in the following. Change `gan_latent_lower_proj` and `gan_latent_upper_proj` accordingly for the other variants. Be sure to adjust the loading name to match the learned embedding run.

```
"d_embed_enc" : 128,    "num_heads"   : 8,
```

```
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 40,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "gan_latent_mode" : true,
"objectives" : "gan",   "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_embedder_layers" : 0,
"gan_projector_layers" : 0,     "gan_critic_steps" : 2,
"gan_critic_target_fn" : "logits",      "gan_critic_target_mode" : "direct",
"gan_generator_target_fn" : "logits",   "gan_generator_target_mode" : "direct",
"gan_gradient_penalty" : 10.0,  "gan_latent_upper_proj" : false,
"gan_latent_lower_proj" : false,        "load_from" : "<learned embedding>",
"load_parts" : "embedder,projector"
```


Reproduction 34 (Optimized reference classifier for Section 5.4.1)


```
"d_embed_enc" : 128, "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 80, "ds_name"   : "RPC100",
"max_encode_length" : 51,
"objectives" : "class", "num_layers" : 4,
"gan_critic_steps" : 1
```


Reproduction 35 (Constant reference classifier for Section 5.4.1)


```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 80,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "objectives" : "class",
"num_layers" : 4,       "gan_critic_steps" : 2,
"gan_objweight_class" : 10.0,   "gan_force_constant_lr" : true
```


Reproduction 36 (GAN with classification objective)


```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 80,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "objectives" : "gan,class",
"num_layers" : 4,       "gan_generator_layers" : 6,
"gan_critic_steps" : 2, "gan_critic_target_fn" : "sigmoid-log",
"gan_critic_target_mode" : "one-minus",
"gan_generator_target_fn" : "sigmoid-log",
"gan_generator_target_mode" : "direct"
```


Reproduction 37 (WGAN with classification objective)


```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 80,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "objectives" : "gan,class",
"num_layers" : 4,       "gan_generator_layers" : 6,
"gan_critic_steps" : 2, "gan_critic_target_fn" : "logits",
"gan_critic_target_mode" : "direct",    "gan_generator_target_fn" : "logits",
"gan_generator_target_mode" : "direct", "gan_gradient_penalty" : 10.0,
"gan_objweight_class" : 10
```

Reproduction 38 (WGAN with split heads and classification objective)
Adapt Rep. 37 by

```
"num_layers" : 3,       "gan_critic_class_layers" : 1,
"gan_critic_critic_layers" : 1,
```

Reproduction 39 (WGAN with split heads and classification objective)
Adapt Rep. 38 so that `num_layers` is the number of shared layers and both `gan_critic_class_layers` and `gan_critic_critic_layers` are set to the remaining number.

Reproduction 40 (WGAN with split heads, classification and entropy uncertainty objective)

```
    "d_embed_enc" : 128,        "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 80,     "ds_name"       : "RPC100",
"max_encode_length" : 51,       "objectives" : "gan,class",
"num_layers" : 3,       "gan_critic_class_layers" : 1,
"gan_critic_critic_layers" : 1, "gan_generator_layers" : 6,
"gan_critic_steps" : 2, "gan_critic_target_fn" : "logits",
"gan_critic_target_mode" : "direct",    "gan_generator_target_fn" : "logits",
"gan_generator_target_mode" : "direct", "gan_gradient_penalty" : 10.0,
"gan_objweight_class" : 10,     "gan_generate_confusion" : true,
"gan_objweight_confusion" : 2,  "gan_confusion_loss" : "entropy",
"gan_delay_confusion_steps" : 0
```

Reproduction 41 (WGAN with split heads, classification and absolute error uncertainty objective)
Adjust Rep. 40 by

```
  "gan_objweight_confusion" : 0.5,      "gan_confusion_loss" : "mae"
```

Reproduction 42 (Pretrained WGANs with uncertainty objectives)
Identical to Rep. 40 respective Rep. 41, but load a model trained with Rep. 38 for 30k steps with

```
"load_from" : "<pretrained model>"
```

Reproduction 43 (Dataset `Uncert-e-raw` obtained by sampling from Rep. 42)
Make sure that the `load_from` parameter is set to the run name that was used for training the model.

```
"no_save" : true,       "d_embed_enc" : 128,
"num_heads"   : 8,      "d_ff"  : 1024,
"batch_size"  : 1024,   "epochs"      : 40,
"ds_name"       : "RPC100",     "max_encode_length" : 51,
"objectives" : "",      "num_layers" : 4,
"gan_generator_layers" : 6,     "gan_critic_steps" : 0,
"load_from" : "<run name here>",        "gan_trainsteps_process_interval" : 1,
"gan_process_generated_samples" : true,
    ↪ "gan_filter_generated_entropy_threshold" : 0.0,
"gan_save_valid_samples" : true,        "gan_solve_valid_samples" : true,
"gan_balance_valid_samples" : false,    "gan_solve_tool" : "aalta"
```

Copy the file `generated_samples.txt` to `datasets/Uncert-e/raw.txt`. Identical procedure for `Uncert-a-raw` with the respective model.

Reproduction 44 (Dataset `Uncert-e`)
Proceed similar to Rep. 23, using the file obtained from Rep. 43. Identical procedure for `Uncert-a` with the respective raw dataset.

Reproduction 45 (Reference classifiers for Section 5.4.3)

```
"d_embed_enc" : 128, "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 129, "ds_name"  : "Uncert-e",
"max_encode_length" : 51,
"objectives" : "class", "num_layers" : 4,
"gan_critic_steps" : 1
```

Similar procedure for `Uncert-a`. Evaluate on a different dataset by

```
python -m dlsgs.train.gan --run-name NAME --test --ds-name DATASET_NAME
```

Reproduction 46 (WGAN for incremental learning)
Load from Rep. 38

```
"d_embed_enc" : 128,    "num_heads"   : 8,
"d_ff"  : 1024, "batch_size"  : 1024,
"epochs"      : 120,    "ds_name"       : "RPC100",
"max_encode_length" : 51,       "objectives" : "gan,class",
"num_layers" : 3,       "gan_critic_class_layers" : 1,
"gan_critic_critic_layers" : 1, "gan_generator_layers" : 6,
"gan_critic_steps" : 2, "gan_critic_target_fn" : "logits",
"gan_critic_target_mode" : "direct",    "gan_generator_target_fn" : "logits",
"gan_generator_target_mode" : "direct", "gan_gradient_penalty" : 10.0,
"gan_objweight_class" : 10,     "gan_generate_confusion" : true,
"gan_objweight_confusion" : 0.5,        "gan_confusion_loss" : "mae",
"gan_process_confusion_samples": true,  "gan_save_confusion_samples" : true,
"gan_incremental_learning_mode" : true, "gan_solve_tool" : "aalta",
"gan_incremental_usage_zero_step" : 100,
"gan_incremental_usage_full_step" : 20000,
"gan_delay_confusion_steps" : 0,
"gan_filter_confusion_entropy_threshold" : 0.0,
"load_from" : "<pretrained model>"
```

References

[1] Antreas Antoniou, Amos J. Storkey, and Harrison Edwards. "Augmenting Image Classifiers Using Data Augmentation Generative Adversarial Networks". In: *Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III*. Ed. by Vera Kurková et al. Vol. 11141. Lecture Notes in Computer Science. Springer, 2018, pp. 594–603. DOI: 10.1007/978-3-030-01424-7_58.

[2] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. "Towards solving differential equations through neural programming". In: *ICML Workshop on Neural Abstract Machines and Program Induction (NAMPI)*. 2018.

[3] Martín Arjovsky and L. Bottou. "Towards Principled Methods for Training Generative Adversarial Networks". In: *ArXiv* abs/1701.04862 (2017).

[4] Martín Arjovsky, Soumith Chintala, and L. Bottou. "Wasserstein Generative Adversarial Networks". In: *ICML*. 2017.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473 (2015).

[6] Mislav Balunović, Pavol Bielik, and Martin Vechev. "Learning to Solve SMT Formulas". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 10338–10349.

[7] Kshitij Bansal et al. "HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 454–463.

[8] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), 14:1–14:64. DOI: 10.1145/2000799.2000800.

[9] Irwan Bello et al. *Neural Combinatorial Optimization with Reinforcement Learning*. 2017. arXiv: 1611.09940 [cs.AI].

[10] Matteo Bertello et al. "Leviathan: A New LTL Satisfiability Checking Tool Based on a One-Pass Tree-Shaped Tableau". In: *IJCAI*. 2016.

[11] Armin Biere and K Claessen. "Hardware model checking competition". In: *Hardware Verification Workshop*. 2010.

[12] Aaron Bohy et al. "Acacia+, a Tool for LTL Synthesis". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 652–657. DOI: 10.1007/978-3-642-31424-7_45.

[13] Christopher Bowles et al. "GAN Augmentation: Augmenting Training Data using Generative Adversarial Networks". In: *CoRR* abs/1810.10863 (2018). arXiv: 1810.10863.

[14] Tong Che et al. "Maximum-Likelihood Augmented Discrete Generative Adversarial Networks". In: *CoRR* abs/1702.07983 (2017). arXiv: 1702.07983.

[15] Liqun Chen et al. "Adversarial Text Generation via Feature-Mover's Distance". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[16] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”. In: *ArXiv* abs/1406.1078 (2014).

[17] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[18] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[19] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[20] Mostafa Dehghani et al. “Universal Transformers”. In: (July 10, 2018). arXiv: 1807.03819v3 [cs.CL].

[21] Alexandre Duret-Lutz et al. “Spot 2.0 — A Framework for LTL and ω-Automata Manipulation”. In: *Automated Technology for Verification and Analysis*. Ed. by Cyrille Artho, Axel Legay, and Doron Peled. Cham: Springer International Publishing, 2016, pp. 122–129.

[22] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Patterns in Property Specifications for Finite-State Verification”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. ICSE ’99. Los Angeles, California, USA: Association for Computing Machinery, 1999, pp. 411–420. DOI: 10.1145/302405.302672.

[23] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Property Specification Patterns for Finite-State Verification”. In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*. FMSP ’98. Clearwater Beach, Florida, USA: Association for Computing Machinery, 1998, pp. 7–15. DOI: 10.1145/298595.298598.

[24] E. Allen Emerson. “CHAPTER 16 - Temporal and Modal Logic”. In: *Formal Models and Semantics*. Ed. by JAN VAN LEEUWEN. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990, pp. 995–1072. DOI: https://doi.org/10.1016/B978-0-444-88074-1.50021-4.

[25] Kousha Etessami and Gerard J. Holzmann. “Optimizing Büchi Automata”. In: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*. Vol. 1877. Lecture Notes in Computer Science. Springer, 2000, pp. 153–167. DOI: 10.1007/3-540-44618-4_13.

[26] Richard Evans et al. “Can Neural Networks Understand Logical Entailment?” In: (Feb. 23, 2018). arXiv: 1802.08535v1 [cs.NE].

[27] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. “BoSy: An Experimentation Framework for Bounded Synthesis”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17.

[28] William Fedus, Ian J. Goodfellow, and Andrew M. Dai. "MaskGAN: Better Text Generation via Filling in the _____". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[29] Bernd Finkbeiner and Sven Schewe. "Bounded synthesis". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 519–539. DOI: 10.1007/s10009-012-0228-z.

[30] Bernd Finkbeiner and Sven Schewe. "Uniform Distributed Synthesis". In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 2005, pp. 321–330. DOI: 10.1109/LICS.2005.53.

[31] Bernd Finkbeiner and Henny Sipma. "Checking Finite Traces Using Alternating Automata". In: *Formal Methods Syst. Des.* 24.2 (2004), pp. 101–127. DOI: 10.1023/B:FORM.0000017718.28096.48.

[32] Thibault Gauthier et al. "TacticToe: Learning to Prove with Tactics". In: *J. Autom. Reason.* 65.2 (2021), pp. 257–286. DOI: 10.1007/s10817-020-09580-x.

[33] Ian P. Gent and T. Walsh. "The SAT Phase Transition". In: *ECAI*. 1994.

[34] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014.

[35] Anirudh Goyal et al. "Professor Forcing: A New Algorithm for Training Recurrent Networks". In: *NIPS*. 2016.

[36] Alex Graves et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626 (2016), pp. 471–476.

[37] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. NIPS '17. Curran Associates, Inc., 2017.

[38] Jiaxian Guo et al. "Long Text Generation via Adversarial Training with Leaked Information". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 5141–5148.

[39] Christopher Hahn et al. "Teaching Temporal Logics to Neural Networks". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[40] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[41] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[42] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda, eds. *Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2018, p. 77.

[43] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9 (1997), pp. 1735–1780.

[44] Jan Holeček et al. *Verification results in Liberouter project*. 2004.

[45] Daniel Huang et al. "GamePad: A Learning Environment for Theorem Proving". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[46] Fei Huang et al. "A Text GAN for Language Generation with Non-Autoregressive Generator". In: (2020).

[47] Drew A Hudson and C Lawrence Zitnick. "Generative adversarial transformers". In: *arXiv preprint arXiv:2103.01209* (2021).

[48] Ferenc Huszár. "How (not) to Train your Generative Model: Scheduled Sampling, Likelihood, Adversary?" In: *ArXiv* abs/1511.05101 (2015).

[49] IEEE-Commission et al. "IEEE standard for property specification language (PSL)". In: *IEEE Std 1850-2005* (2005).

[50] Geoffrey Irving et al. "DeepMath - Deep Sequence Models for Premise Selection". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. 2016, pp. 2235–2243.

[51] Swen Jacobs et al. "The first reactive synthesis competition (SYNTCOMP 2014)". In: *Int. J. Softw. Tools Technol. Transf.* 19.3 (2017), pp. 367–390. DOI: 10.1007/s10009-016-0416-3.

[52] Yifan Jiang, Shiyu Chang, and Zhangyang Wang. "Transgan: Two transformers can make one strong gan". In: *arXiv preprint arXiv:2102.07074* (2021).

[53] Armand Joulin and Tomas Mikolov. "Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Cited by MathReasoning-Analysis. Curran Associates, Inc., 2015, pp. 190–198.

[54] Łukasz Kaiser and Ilya Sutskever. "Neural GPUs Learn Algorithms". In: (Nov. 25, 2015). arXiv: 1511.08228v3 [cs.LG].

[55] Cezary Kaliszyk, François Chollet, and Christian Szegedy. "HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[56] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

[57] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2015).

[58] Jens U. Kreber and Christopher Hahn. *Generating Symbolic Reasoning Problems with Transformer GANs*. 2021. arXiv: 2110.10054 [cs.LG].

[59] A. Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60 (2012), pp. 84–90.

[60] Sachin Kumar and Yulia Tsvetkov. "End-to-End Differentiable GANs for Text Generation". In: *Proceedings on "I Can't Believe It's Not Better!" at NeurIPS Workshops*. Vol. 137. Proceedings of Machine Learning Research. PMLR, 2020, pp. 118–128.

[61] Matt J. Kusner and José Miguel Hernández-Lobato. "GANS for Sequences of Discrete Elements with the Gumbel-softmax Distribution". In: *ArXiv* abs/1611.04051 (2016).

[62] Guillaume Lample and Franccois Charton. "Deep Learning for Symbolic Mathematics". In: *ArXiv* abs/1912.01412 (2020).

[63] Gil Lederman et al. "Learning Heuristics for Quantified Boolean Formulas through Deep Reinforcement Learning". In: (July 20, 2018). arXiv: 1807.08058v3 [cs.LO].

[64] Dennis Lee et al. "Mathematical Reasoning in Latent Space". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[65] Jianwen Li et al. "Aalta: An LTL Satisfiability Checker over Infinite/Finite Traces". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 731–734. DOI: 10.1145/2635868.2661669.

[66] Jianwen Li et al. "LTL Satisfiability Checking Revisited". In: *Proceedings of the 2013 20th International Symposium on Temporal Representation and Reasoning*. TIME '13. TIME '13. USA: IEEE Computer Society, 2013, pp. 91–98. DOI: 10.1109/TIME.2013.19.

[67] Jiwei Li et al. "Adversarial Learning for Neural Dialogue Generation". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 2157–2169. DOI: 10.18653/v1/D17-1230.

[68] Kevin Lin et al. "Adversarial Ranking for Language Generation". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 3155–3165.

[69] Zhouhan Lin et al. *A Structured Self-attentive Sentence Embedding*. 2017. arXiv: 1703.03130 [cs.CL].

[70] Sarah Loos et al. "Deep Network Guided Proof Search". In: *In Thomas Eiter and David Sands, editors, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21). EPiC Series in Computing, vol. 46, pages 85-105, EasyChair, 2017. ISSN 2398-7340* (Jan. 24, 2017). LPAR '17. arXiv: 1701.06972v1 [cs.AI].

[71] Sarah M. Loos et al. "Deep Network Guided Proof Search". In: *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 85–105.

[72] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

[73] Larry R Medsker and LC Jain. "Recurrent neural networks". In: *Design and Applications* 5 (2001).

[74] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *ICLR*. 2013.

[75] Tomas Mikolov et al. “Recurrent neural network based language model.” In: *INTERSPEECH*. Ed. by Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura. ISCA, 2010, pp. 1045–1048.

[76] Weili Nie, Nina Narodytska, and Ankit B. Patel. “RelGAN: Relational Generative Adversarial Networks for Text Generation”. In: *ICLR*. ICLR ’19. 2019.

[77] Aditya Paliwal et al. “Graph Representations for Higher-Order Logic and Theorem Proving”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 2967–2974.

[78] Aditya Sanjay Paliwal et al. “Graph Representations for Higher-Order Logic and Theorem Proving”. In: *ArXiv* abs/1905.10006 (2020). AAAI ’20.

[79] Ankur Parikh et al. “A Decomposable Attention Model for Natural Language Inference”. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2249–2255. DOI: 10.18653/v1/D16-1244.

[80] Radek Pelánek. “BEEM: Benchmarks for Explicit Model Checkers”. In: *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*. Vol. 4595. Lecture Notes in Computer Science. Springer, 2007, pp. 263–267. DOI: 10.1007/978-3-540-73370-6_17.

[81] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[82] Stanislas Polu and Ilya Sutskever. “Generative Language Modeling for Automated Theorem Proving”. In: *CoRR* abs/2009.03393 (2020). arXiv: 2009.03393.

[83] Markus N. Rabe and Christian Szegedy. “Towards the Automatic Mathematician”. In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 25–37. DOI: 10.1007/978-3-030-79876-5_2.

[84] Markus Norman Rabe et al. “Mathematical Reasoning via Self-supervised Skip-tree Training”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[85] Kristin Y. Rozier. “Linear Temporal Logic Symbolic Model Checking”. In: *Computer Science Review* 5.2 (2011), pp. 163–203. DOI: https://doi.org/10.1016/j.cosrev.2010.06.002.

[86] Kristin Y. Rozier and Moshe Y. Vardi. “LTL Satisfiability checking”. In: *Model Checking Software*. Ed. by Dragan Bošnački and Stefan Edelkamp. Vol. 12. Springer Berlin Heidelberg, 2007, pp. 149–167. DOI: 10.1007/978-3-540-73370-6_11.

[87] David Saxton et al. “Analysing Mathematical Reasoning Abilities of Neural Models”. In: (Apr. 2, 2019). arXiv: 1904.01557v1 [cs.LG].

[88] Frederik Schmitt et al. “Neural Circuit Synthesis from Specification Patterns”. In: *ArXiv* abs/2107.11864 (2021).

[89] Viktor Schuppan and Luthfi Darmawan. “Evaluating LTL Satisfiability Solvers”. In: *Automated Technology for Verification and Analysis*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 397–413.

[90] Stefan Schwendimann. “A New One-Pass Tableau Calculus for PLTL”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by Harrie de Swart. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 277–291.

[91] Daniel Selsam and Nikolaj Bjørner. “Guiding High-Performance SAT Solvers with Unsat-Core Predictions”. In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikoláš Janota and Inês Lynce. SAT 2019. Cham: Springer International Publishing, 2019, pp. 336–353.

[92] Daniel Selsam et al. “Learning a SAT Solver from Single-Bit Supervision”. In: *ArXiv* (Feb. 11, 2018). ICLR 2019. arXiv: 1802.03685v4 [cs.AI].

[93] A. Prasad Sistla and Edmund M. Clarke. “The Complexity of Propositional Linear Temporal Logics”. In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis et al. ACM, 1982, pp. 159–168. DOI: 10.1145/800070.802189.

[94] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *NIPS*. 2014.

[95] Yaniv Taigman et al. “DeepFace: Closing the Gap to Human-Level Performance in Face Verification”. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition* (2014), pp. 1701–1708.

[96] Ashish Vaswani et al. “Attention Is All You Need”. In: (June 12, 2017). arXiv: 1706.03762v5 [cs.CL].

[97] Carl Vondrick and Antonio Torralba. “Generating the future with adversarial transformers”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 1020–1028.

[98] Mingzhe Wang et al. “Premise Selection for Theorem Proving by Deep Graph Embedding”. In: *NIPS*. NIPS ’17. 2017.

[99] Y. Wu et al. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *ArXiv* abs/1609.08144 (2016).

[100] Kaiyu Yang and Jia Deng. “Learning to Prove Theorems via Interacting with Proof Assistants”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6984–6994.

[101] Xin Yi, Ekta Walia, and Paul Babyn. “Generative adversarial network in medical imaging: A review.” eng. In: *Medical image analysis* 58 (Dec. 2019), p. 101552.

[102] Emre Yolcu and B. Póczos. “Learning Local Search Heuristics for Boolean Satisfiability”. In: *NeurIPS*. 2019.

[103] Lantao Yu et al. “SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient”. In: (Sept. 18, 2016). arXiv: 1609.05473v6 [cs.LG].

[104] Wojciech Zaremba, Karol Kurach, and Rob Fergus. “Learning to Discover Efficient Mathematical Identities”. In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani et al. 2014, pp. 1278–1286.

[105] Wojciech Zaremba and Ilya Sutskever. “Learning to Execute”. In: (Oct. 17, 2014). arXiv: 1410.4615v3 [cs.NE].

[106] Kuo-Hao Zeng, Mohammad Shoeybi, and Ming-Yu Liu. “Style example-guided text generation using generative adversarial transformers”. In: (2020). eprint: arXiv:2003.00674.

[107] Yizhe Zhang et al. “Adversarial Feature Matching for Text Generation”. In: (June 12, 2017). arXiv: 1706.03850v3 [stat.ML].