Predicting Timed Traces With Neural Networks

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Ayham Omar

Saarbrücken, April 2022

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Dr. Christopher Hahn
	Niklas Metzger
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Dr. Sebastian Hack
Submission:	14 April, 2022

Abstract

Recently, deep learning has been applied to the field of logical reasoning delivering promising results as a complement to classical algorithms. Particularly the Transformer architecture has been proved, in existing work, to be proficient not only in predicting the satisfiability of propositional and Linear Time Temporal Logic (LTL) formulas but also in the ability to construct satisfying solutions for these formulas. However, applying deep learning algorithms to formal specifications in the more complex continuous-time domain has not been explored yet.

In this work, we introduce the problem of predicting a satisfying timed trace for a Metric Interval Temporal Logic (MITL) formula to a state-of-the-art Transformer neural network. Specifications in MITL contain explicit time intervals to reason about the behavior of real-time systems, thus enforcing the Transformer to predict more profound traces.

We describe the Transformer architecture and explain the methods used in generating meaningful training data for a supervised training approach. Furthermore, we conduct several experiments to determine to what extent the model learns the semantics of MITL. We, to this end, differentiate between the semantic and syntactic accuracy of the solutions predicted by the model.

We find that Transformers prove proficient in solving MITL formulas, reaching over 90% of accuracy in some experiments. We also observe that a trained Transformer can predict correct solutions that deviate from the ones constructed by the data generator, demonstrating signs of generalizing to the semantics of the logic. This generalization property was even evident when challenging the Transformer with formulas much longer than it encountered during training or even formulas on which the data generator timed out. Concerning the stage of studying the effects of the size of the time intervals on the Transformer, we find that the Transformer continues to deliver good results when faced with much bigger intervals. An interesting result since solving formulas containing big intervals is particularly expensive for most classical approaches.

Acknowledgements

I would like to acknowledge Prof. Moritz Weber for the program "*Preparatory math courses for studies in MINT subjects* (2015-2018)"[1]. Passing this program is what qualified me to begin my bachelor's study at Saarland University. But foremost, I would like to express my sincere gratitude to my supervisor Prof. Bernd Finkbeiner, who contributed to my interest in the subject and offered me this project. I also genuinely thank Prof. Sebastian Hack for acting as the second reviewer for my thesis. And an earnest thank you to both of my advisors, Christopher Hahn and Niklas Metzger, for their unconditional support during my work on this project and for proofreading the thesis. Last but not the least, many thanks to my sister for her continuous support and proofreading this thesis.

[1]`https://www.math.uni-sb.de/ag/speicher/MathRefresh.html`

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

Statement in Lieu of an Oath
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, 14 April, 2022

Contents

Chapter 1

Introduction

The advancements of machine learning in computer science made it possible to solve tasks for which composing a classical algorithm would be inefficient. Deep learning, in particular, proved capable of matching or even outperforming humans in tasks such as medical imaging evaluation [1], classification [2] and video games [3].

However, the utilization of deep learning in the field of logical reasoning where many problems have high computational complexity is not yet voluminous. The rationale behind using machine learning to tackle such problems is mostly the competency in which a well-trained model can devise a solution to a given problem regardless of its complexity. This ability makes deep learning an admirable substitute for classical algorithms in practice. For instance, the predicted solutions of a model can be checked for accuracy, a procedure that for many problems turns out to be more direct than constructing the solution from scratch. In the infrequent cases where the model predicts faulty solutions, one can then refer to the slower but sound classical algorithms, this technique leads to overall enhanced performance.

Existing works have demonstrated promising results in training neural networks to decide the satisfiability of different types of logic and even in predicting satisfying assignments to logical formulas. In [4] for example, authors have successfully trained a neural network on solving the satisfiability problem of propositional logic (SAT). Equivalently, authors in [5] went beyond that and managed to predict satisfying traces for the Linear Temporal Logic (LTL) [6] in the discrete-time domain. None of the previous results, however, have dealt with the problem of predicting timed traces in the more complex continuous-time domain. Therefore, in this thesis, we will expand existing work in applying deep learning algorithms to reason about the Metric Interval Temporal Logic (MITL) [7].

MITL is a temporal language used to describe the timing properties of real-time systems. As an extension LTL, MITL is more proficiently adept at expressing requirements where we argue about the state of the system inside designated time periods. MITL facilitates this by bounding its temporal operators with time intervals. One can use MITL to express statements such as a proposition p must hold for some time between three to five-time units in the future ($\Diamond_{[3,5]}\, p$) or a proposition p must hold for the entire duration of a given time window ($\Box_{[3,5]}\, p$), that is for all time units between (and including) three and five.

MITL expressiveness abilities led MITL to become the specification language of choice for applications in fields like runtime verification of real-time systems [8, 9, 10] as well as in model-checking in which there is a developing interest in translating MITL formulas into timed automata [11, 12, 13].

In the objective of training a deep learning model on solving MITL formulas, we use the prominent Transformer neural network as the basis of our model. Transformers are sequence-to-sequence neural networks based on the encoder-decoder architecture. In this architecture, the input sequence is passed through a stack of identical encoders resulting in a prepared representation of the input sequence that is propagated afterward into N identical decoders in the decoding block. The decoders then use a combination of the prepared input and the embeddings of the earlier generated output sequence to predict the following output word.

Since its introduction in [14], the Transformer architecture was the core of state-of-the-art NLP machine learning techniques such as BERT [15] and GPT-3 [16]. The novelty of the Transformer architecture was essentially consuming the input sequence as a whole, thus achieving higher parallelization during training and consequently consuming a shorter time to train. This, along with applying positional encoding techniques and attention mechanisms, permits the Transformer to preserve the context of the processed sentence more efficiently, as well as to emphasize the relevance between related input words.

In this project, we use a supervised training approach to train a Transformer on pairs consisting of an MITL formula and a satisfying timed trace. We then perform multiple experiments to evaluate the accuracy of the model in predicting satisfying timed traces when provided with an MITL formula from a held-out test set. For instance, giving the MITL formula $(\mathsf{c}\ \mathcal{U}_{[1,11]}\ \Diamond_{[1,9]}\, \mathsf{e})$ our model predicts the timed trace $\mathsf{c} \wedge \mathsf{e}\, [0,1) \rightarrow \mathsf{c} \wedge \mathsf{e}\, [1,2) \rightarrow \mathsf{c} \wedge \mathsf{e}\, [2,\infty)$. When checking the predicted trace against the MITL formula for satisfiability, we find the predicted solution to be accurate (more on the evaluation process and data representation in Section 4).

On the fact that solutions to an MITL formula are not unique, we attempt to answer the interesting question: Has the trained model learned the actual semantics of the temporal logic, or is it imitating the algorithm of the data generator? We, consequently, differentiate between the semantic and syntactic accuracy of the solutions predicted by the model and seek to answer this question by testing the model on formulas with a different pattern than the ones utilized in the training set. We also investigate if the model can generalize to formulas longer than it has ever encountered during training and study the effect positional encoding has on this ability. Moreover, we study the impact the size of the time intervals has on the Transformer's performance.

For the rest of this thesis, we commence by providing an overview of related work. Next, we briefly introduce neural networks and then elaborate on explaining the encoder-decoder architecture of the Transformer neural network. Then, we introduce the syntax and semantics of MITL, answering the question of when a timed trace is considered a valid solution for an MITL formula. Afterward, we present the procedure used to obtain the required data sets of MITL formulas and their solutions, we discuss the characteristics of each data set as well. Thereafter, we present the experiments conducted, alongside discussing the results. Finally, we conclude in Section 6.

Chapter 2

Related Work

The supervised training of a Transformer neural network to predict satisfying solutions to LTL formulas [5] is directly related to this work. Results showed that the Transformer achieved high accuracy up to 96.8% on a held-out test set; it even predicted correct solutions for 83% of a set of formulas on which the classical tool used to generate the training data was timed out. Examining the advantages of using tree-positional encoding demonstrated that its use results in the model generalizing to longer LTL formulas than it saw during training, as opposed to using the standard positional encoding which didn't cause the model to generalize. Transformers, in addition, were shown to generalize to the semantics of LTL by predicting different solutions from those in the test set, yet those predicted solutions were correct. In a successor work of the same authors, the further complex problem of synthesizing hardware circuits out of LTL specifications was considered [17]. This time the authors used a tweaked version of the Transformer neural network called the hierarchical Transformer [18] achieving adequate results. In this project, we train a Transformer to conduct similar experiments as in [5], yet this time, on the problem of finding a satisfying solution for MITL formulas. Here the model has to learn the time intervals linked to the temporal operators in the formulas, it further has to predict the intervals in the solution trace, in which each step of the trace holds.

Another work employed deep learning in the scope of logical reasoning, trained a graph neural network (GNN) [19] as a classifier for predicting the satisfiability of propositional logic formulas [4]. The model in this work was trained on pairs of propositional logic formulas in their conjunctive normal form (CNF) and a single bit designating if the formula is satisfiable or not. The model herewithin has also generalized to longer formulas and formulas of novel distributions. Surprisingly, despite only being trained as a classifier, the activations of the model demonstrated that it learned to solve the formulas and the solutions could be decoded from the network's activations. In our work, we straightforwardly train the model to predict satisfying solutions rather than only predicting satisfiability. Moreover, the logic we utilize does not have to be transformed into any special form and it is more exacting and includes more involved operators.

Examples of different works demonstrating the makings of applying neural networks as a component of larger logical frameworks to ameliorate performance: In [20], authors trained a Recurrent Neural Network (RNN) to avail determine a control strategy for a system while ascertaining that a specification provided in Signal Temporal Logic (STL) is not infringed. STL is a similar logic to MITL that defines predicates over real values. In [21], authors utilized neural networks to improve the heuristics in SMT solvers ameliorating their performance up to $100\times$. In a subsequent work to [4], the authors in [22] used a simplified version of the exact architecture to predict unsatisfiable cores of real problems, enhancing the performance of several high-performance SAT solvers. In [23] the GNN architecture was used as an alternative for syntax trees to represent formulas instead, leading to improvements in the performance of higher-order theorem provers. However, In our work, we train a neural network as a standalone end-to-end MITL solver.

Other areas where Deep Learning additionally proved capable, is detecting variable misuses in source code and electing the proper variable to be used [24]. Moreover, in symbolic mathematics [25], where the authors used a Transformer neural network to predict the solutions of integration and differential equations.

Chapter 3

Background

Throughout this chapter, we briefly introduce the basics of artificial neural networks, afterwards, we further describe the architecture used in this project.

3.1 Artificial Neural Networks

Artificial Neural Networks are a class of machine learning algorithms inspired by the biological neural networks found in animal brains. Mimicking the way the brain works, neural networks learn to predict a value from a target set for a given input through analyzing a substantial number of pre-labeled examples in a process called training, in which hidden patterns are detected and the parameters of the network are adjusted accordingly.

With the gradual advancement of computational power and the influx of data, the fixation on applying neural networks has increased drastically. Computer algorithms predicated on neural networks reached and surpassed human-level accuracy in solving problems in sundry relevant applications, as mentioned in Section 1.

In this section, we introduce the building blocks of neural networks and their learning process. We focus on the feedforward architecture and the supervised learning approach as they are the relevant methods used in this project.

3.1.1 Neurons

Neural networks comprise the interconnections of numerous simple computational units known as Neurons. As depicted in Figure 3.1 an artificial neuron receives input values from other connected neurons, applies what is called an activation function on the weighted sum of the inputs, and progresses to output the processed data to other neurons in the network. A bias b is typically added to the input of a neuron, permitting the output of the activation function to fit the predictions better. Each input connection

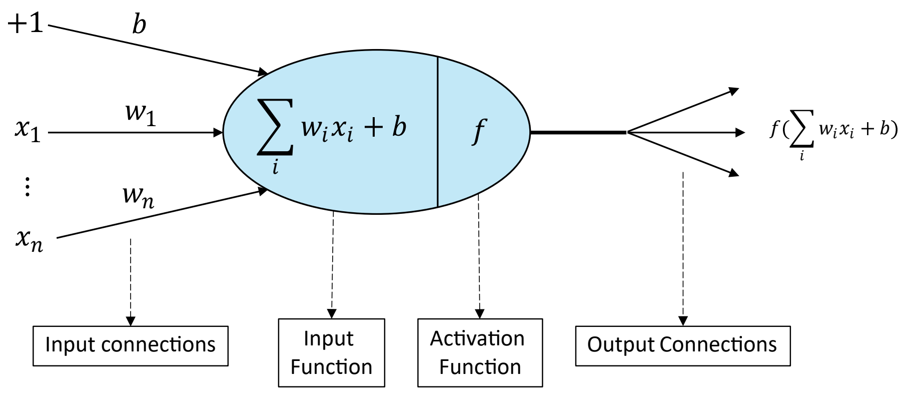


Figure 3.1: In addition to a bias value b, the artificial neuron receives n input values from the neurons in the preceding layer. It then applies a mathematical function f on the weighted sum of the inputs and proceeds to propagate the end result to the neurons in the next layer.

is initially weighted with a random value, the weights are then updated during training and they reflect the strength of the corresponding connection and to what extent the input will influence the output of the neuron.

Activation functions (also called transfer functions) interpret the input signals into output signals. Some commonly used functions are:

- The Step function:

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

- The Sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}}$$

- The Rectified Linear Unit (ReLu) function:

$$f(x) = max(0, x)$$

We view next how we can align neurons in connected layers to create a feedforward neural network and demonstrate the training process.

3.1.2 Feedforward Neural Networks

As the most fundamental neural network architecture, the feedforward neural network (FFNN) consists of an input layer, an output layer, and one or more hidden layers in between. Each layer is composed of several independent neurons that carry weighted connections to one or more neurons in the adjacent layer. In the case where each neuron connects to all neurons found in the next layer, we call the network a fully-connected

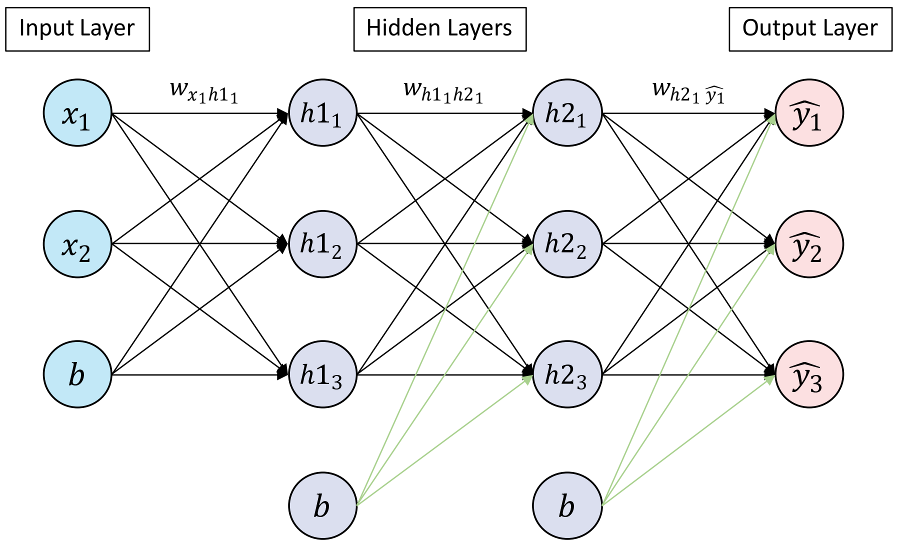


Figure 3.2: A fully-connected feedforward neural network with 2 hidden layers (weights are only partially denoted for clarity)

FFNN. As it is a feedforward network, the neurons of each layer only process the outputs provided by the neurons of the previous layer, restricting signals to only travel in one direction, from the input layer to the output layer.

Figure 3.2 resembles a FFNN with two hidden layers. Training FFNNs to solve a certain task using a supervised learning algorithm is conducted by subjecting the network to a data set composed of examples of inputs and their corresponding outputs. The training process can be considered as learning to approximate the underlying resulting function in this data set.

While being trained, the FFNN analyzes each example in the data set by first feeding each element of the input vector into the neurons of the first layer. Then the input is transmitted as it is into the first hidden layer, i.e the input layer does not apply any activation functions to the input. The neurons of the first hidden layer afterward apply the computation that was explained in the previous section, on all the inputs they receive and output the result of the activation function into the successive hidden layer, if existent, or into the output layer. Finally, the neurons of the output layer are responsible for producing the network's prediction.

Since the weights and biases of the FFNN are randomly initiated, the predictions of the network in the early stages of training are inaccurate. To improve the accuracy of its predictions, the FFNN utilizes what is called a cost (or loss) function.

The cost function evaluates the accuracy of the predictions compared to the ground truth (the correct output in the data set). Using the cost function, the loss is computed over n data samples.

As an example we demonstrate this computation using the Mean Squared Error (MSE) cost function:

$$\mathrm{MSE} = \frac{1}{n}\sum_{n=0}^{n}(Y_i - \hat{Y}_i)^2$$

Where n is the number of training examples analyzed, Y_i the ground truth, and $\hat{Y}_i$ the predicted output. Here, the actual learning is realized using an optimization algorithm that updates the weights of the FFNN minimizing the cost function, i.e., to improve the accuracy of future predictions. The most common optimization algorithms used are based on the gradient descent optimization approach. The goal of this approach is to minimize the loss function to its local minimum. More details on the gradient descent strategy and its related algorithms are elaborated on in [26].

3.2 Transformer Neural Network

The Transformer neural architecture was first introduced in [14] and since has replaced Recurrent Neural Networks (RNNs) as the leading architecture for Natural Language Processing (NLP). Transformers are FFNNs that employ attention mechanisms to manage to consume the input sequence fully instead of using recurrent cells to process the input sequentially. This capability of consuming the sequence fully allows for a superior parallelization that leads to more speedy training times and further allows for capturing the dependencies and context between divergent words regardless of the distance between those words. Preserving dependencies between distant words was an obstacle for former models and though attention techniques were used to overcome this obstacle in sequence to sequence models such as RNN [27], Transformers were still superior as being uniquely based on attention.

Transformers are established on the encoder-decoder structure, in which both the encoder and decoder blocks are composed of a stack of N identical encoders and decoders respectively (six encoders and six decoders in the original paper [14]). The encoder stack is used to encode the input sequence in its entirety resulting in an internal representation of the input. This representation is afterward passed to the decoder stack which uses it to predict the upcoming output. An overview of the complete architecture can be seen in Figure 3.4.

In this section, we are attempting to explicate the various layers and features of the Transformer architecture and then take a step back to introduce the workflow of the entire Transformer network.

3.2.1 Word Embeddings

Not unique to Transformers, word embedding is a paramount process in deep learning and especially in NLP, where each word in the input lexicon is encoded as a realvalued vector of a designated dimension ($d_{model} = 512$ in the original paper [14]). The resulting embedding vectors from this process can also help to encapsulate the syntactic and semantic kindred attributes of the input words, where words similar in meaning are mapped closer to each other in the embedding space. There are various word embedding algorithms that use neural networks to learn this encoding process, the most prominent is WORD2VEC [28].

3.2.2 Positional Encoding

Since Transformers consume the input sequence as a whole and not sequentially, the concept of position for the different sequence tokens gets lost. To preserve the apprehension about the relative and absolute positions of the different words, we map each word to a vector called the positional embedding (PE) vector. This vector is of the same size as the input embedding vector, which allows the summation of the two as we are going to perceive later. In [14] $sine$ and $cosine$ functions were used to compute the PE vector as follows:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

where pos is the position of the input word, and i is the dimension in the PE vector.

3.2.3 Attention

The innovation of the Transformer architecture was depending totally on self-attention to process the entire input sequence altogether, rather than recurrently processing input words in sequence. The motive behind this term is to provide additional context information when encoding the sequence of tokens. This additional information aids the Transformer to be more heedful to the most pertinent parts of the sentence whilst encoding or decoding a token.

3.2.4 Scaled Dot-Product Attention

Scaled Dot-Product Attention depicted in Figure 3.3 is the method used in [14] to calculate the attention matrix. For this method, we first need to generate three vectors for each of the sequence tokens. These vectors are a query vector q, a key vector k, and a value vector v. This is done by multiplying the embedding vector of each token by three weight matrices $W(Q)$, $W(K)$, and $W(V)$. These matrices are arbitrarily initialized and learned in the course of the training. To speed up computation and enhance parallelization, we pack all resulting vectors into matrices Q, K of dimension d_k, and V of dimension d_v. Next, the computation of the final output matrix can be conveyed with this straightforward formula:

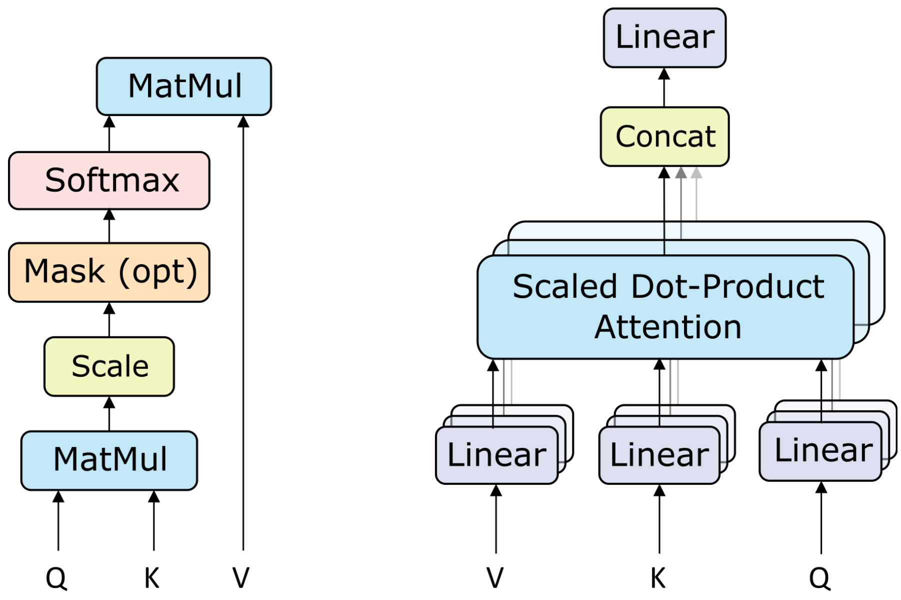


Figure 3.3: (To the left) Scaled Dot-Product Attention which is used to reveal dependencies in a sequence of tokens. (To the right) Multi-Head Attention which is the act of applying the attention method multiple times on the same sequence [14].

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

In the above formula, we multiply the Queries and Keys matrices, divide the outcome by the square root of the Keys matrix dimension and then apply the Softmax function on the output to obtain weights on the values. The scores of the Softmax function are lastly multiplied by the values matrix V increasing the values of the relevant tokens and decreasing the irrelevant ones. Scaling by $\sqrt{d_k}$ serves to halt pushing the Softmax function into regions where it has extremely small gradients [14].

3.2.5 Multi-Head Attention

Authors in [14] found it beneficial to perform the Scaled Dot-Product Attention function not only once, but multiple times in parallel for each token. This novel attention mechanism is called "multi-head attention" in which multiple attention sub-layers (heads) are employed as depicted in Figure 3.3. Each head will then perform the attention function on the entire sequence as elucidated in Subsection 3.2.4 with the only distinction that each head is using a different set of the Q, K and V weight matrices. The output of each head is then concatenated and multiplied with yet another weight matrix W_O resulting in the eventual attention representation of the sequence.

The benefit of replicating the attention computation with distinct randomly generated and learned weight matrices is to enhance the detection of relevant words for each token, where each head would disclose different dependencies.

The multi-head attention computation can be conceivable formalized as follows:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(head_1, ..., head_h)W^O$$
$$\text{Where } head_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Here W^O is another randomly initiated weight matrix that is jointly trained with the model. The dimension of the weight matrices are $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$.

To bar introducing any overhead while using the multi-head attention mechanism, we reduce the dimensions of the weight matrices in each head in consonance with their number h, that is $d_k = d_V = d_{model}/h = 64$. This reduction together with training the heads in parallel advances to a comparable computation cost to that of single-head attention with complete dimensionality of $d_{model} = 512$.

3.2.6 The FFNN Sub-Layer

Adjacent to attention sub-layers, Transformers employ a FFNN sub-layer composed of two linear transformations with a ReLU activation in between. This sub-layer is trained jointly with the model and intends to prepare the output of one layer to the successive one in the encoders, decoders stacks by potentially enriching its representation. The FFNN is applied to each position in the input sequence independently whilst preserving the dimension of the input, that is $d_{input} = d_{output} = d_{model}$. The FFNN administers the following computation:

$$\mathrm{FFNN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Here, W_1 and W_2 are weight matrices of dimensions $d_{model} \times d_{ff}$ and $d_{ff} \times d_{model}$ respectively with $d_{ff} = 2048$ and $b_1 \in \mathbb{R}^{d_{ff}}$, $b_2 \in \mathbb{R}^{d_{model}}$ are the biases.

3.2.7 The Complete Architecture

Figure 3.4 illustrates the full architecture of the Transformer Neural Network. We forthwith observe the exact steps a Transformer performs to predict the translation of input sequences amid training.

The Encoder Block

In the encoder block, the multi-head attention sub-layer of the first encoder receives the prepared input sequence that consists of the embedding and PE matrices combined. Then, with the aid of a residual connection [29], both the input and output of the Multi-head attention sub-layer are added and normalized. Next, the normalized matrix is fed

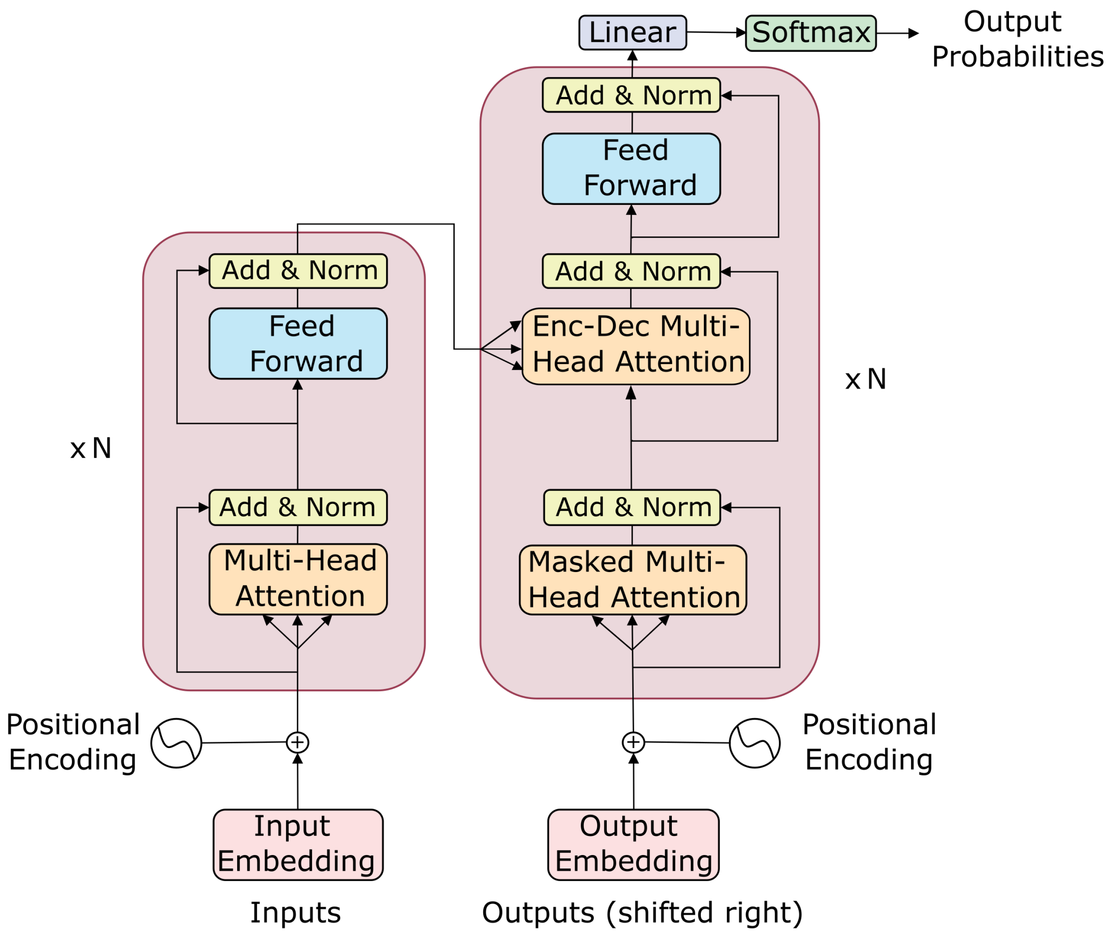


Figure 3.4: The complete Transformer architecture represented by its main encoding/decoding blocks and the different processing sub-layers in each [14].

into an FFNN that is likewise encompassed by a residual connection and followed by an add and normalization sub-layer. Afterward, the output of the encoder is moved as an input to the multi-head attention sub-layer of the following encoder.

The output of the top encoder of the stack is transformed into attention vectors K and V and sent into each decoder in the decoder block. More precisely, it is sent into the encoder-decoder multi-head attention sub-layer of each decoder.

The Decoder Block

In a similar aspect to the first encoder layer, the first decoder receives the embeddings and positional information of the present prediction sequence (initially the sequence only contains a special start symbol). That is when the prepared sequence is fed into a modified multi-head attention sub-layer that is similar to the one used in the encoder block with the key contrast that all values in the input of the Softmax function correspond to connections between the earlier predicted words and all future ones are masked out (set to $-\infty$), ensuring that the later prediction generated by the network depends singularly on the outputs predicted thus far, ergo compelling the network to learn.

The normalized attention matrix from the preceding step is fed into another Multi-head attention sub-layer. This sub-layer is as well distinct in that it gets its K and V attention matrices from the encoder block and its Q matrix from the masked layer underneath it. At last, a set of FFNN and normalization sub-layers will arrange the attention matrix and pass it to the following decoder in the stack.

The output of the top decoder is pushed through two final linear and softmax layers which will convert this output into next-token probabilities where the token with the highest probability is selected. As a result, the predicted token is concatenated to the prediction sequence and the decoding process is copied with this new sequence. Decoding is complete when a special end symbol is predicted.

Chapter 4

Data Generation

Before evaluating our model and conducting any experiments, we need to create sufficient diversified training data to attain a well equitable trained model. As our model is designed to learn through example in a supervised training approach, we need to train the model to approximate a function that maps MITL formulas to satisfying traces. As such, our data sets should consist of pairs of MITL formulas and their solution traces.

To form the fundamental data sets, we need to find a method to (i) generate satisfiable MITL formulas (ii) construct a satisfying trace for a given MITL formula, and (iii) check predicted traces against MITL formulas for satisfiability. On the basis that we need to generate an abundance of samples, all of the above steps have to be done efficiently.

In the course of this chapter, we elucidate the methods we used to construct our data sets. However, we institute by presenting our logic of interest MITL.

4.1 Metric Interval Temporal Logic

Metric Interval Temporal Logic (MITL) [7] is an extension of Linear Temporal Logic (LTL) [6], in which temporal operators are rather bounded by non-punctual timed intervals. As a real-time logic, MITL can describe more qualitative specifications in the continuous-time domain such as deadlines and time windows. For instance, if we consider the timed automaton depicted in Figure 4.1 modeling a simple coffee machine. One can describe specifications like "the coffee must be prepared within 2 to 10 time units" or "the user has a limitation of 15 time units to order after inserting a coin". These specifications can be represented using the MITL formulas $\Box(\mathtt{coffee\_ordered} \rightarrow \Diamond_{[2,10]}\, \mathtt{coffee\_prepared})$ and $\Box(\mathtt{coin\_inserted} \rightarrow (\Diamond_{[0,15)}\, \mathtt{coffee\_ordered} \vee \Diamond_{[15,\infty)}\, \mathtt{coin\_refunded}))$, respectively.

MITL is also considered a restriction of the original Metric Temporal Logic (MTL) [30], in which the constraining intervals of the temporal operators are singular, leading full MTL over infinite words to be undecidable for both the satisfiability and model check-

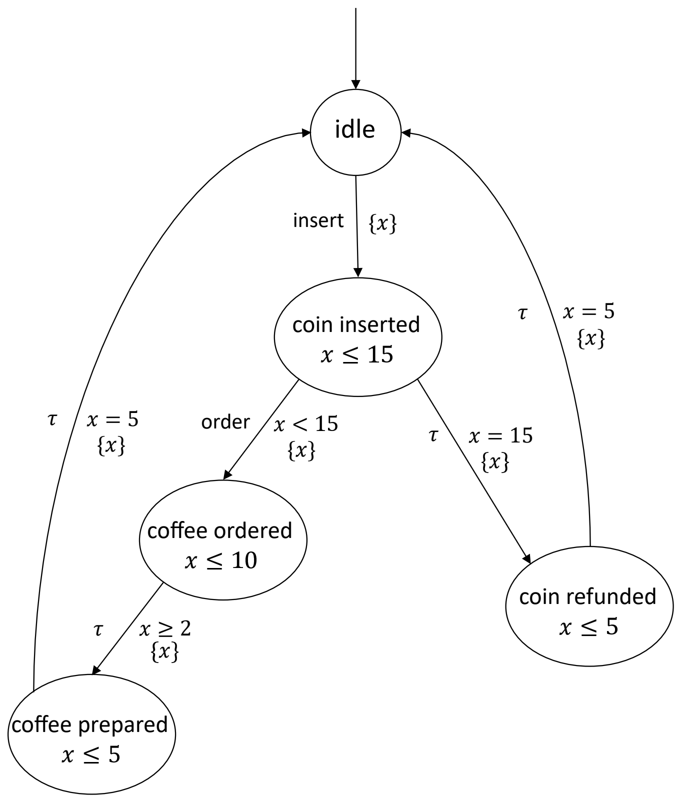


Figure 4.1: A simple timed automaton representing a coffee machine.

ing problems [31]. Prohibiting punctual intervals in MITL allowed for an EXPSPACE decision procedure for both model checking and satisfiability [7].

In this section, we present the syntax and semantics of MITL as viewed in the original paper [7].

4.1.1 Syntax

The syntax of MITL is similar to the syntax of LTL, which will prove conducive later on in this chapter when generating MITL formulas in Section 4.2. MITL formulas, for that reason, are composed of propositions connected by the regular logical operators along with the time-constrained temporal operator $\mathcal{U}$.

Definition 1 (MITL Syntax)

The syntax of MITL is defined over a set of atomic propositions AP by the following grammar

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathcal{U}_I \, \varphi_2$$

Where $p \in AP$ is a proposition and I is a non-singular interval with $l(I) < r(I)$ and $l(I), r(I)$ integer constants. The right end-point of I may also be unbounded $r(I) = \infty$. Restricting intervals to integer end-points serves purely for simplification.

To improve readability of MITL formulas, we derive the temporal operators time-constrained eventually ($\Diamond_I$) and time-constrained always ($\Box_I$) from the $\mathcal{U}$ operator seen above as follows:

$$\Diamond_I \varphi \equiv \text{true}\, \mathcal{U}_I\, \varphi \qquad \text{and} \qquad \Box_I \varphi \equiv \neg \Diamond_I \neg \varphi$$

Where ($\Diamond_{[1,4]}\, a$) is interpreted as "eventually within 1 to 4 time units a should hold" and ($\Box_{[1,4]}\, a$) as "always between 1 and 4 time units a should hold". We also derive the usual propositional operators from the $\neg$ and $\wedge$ operators.

4.1.2 Semantics

As observed in our timed automaton example viewed in Figure 4.1, MITL is interpreted over timed states. A timed state delineates the time period in which this system is found in the said state. As such, we are moving forward from defining the semantics of MITL, to formally instituting timed state sequences, together with its two components state and interval sequences.

Definition 2 (State and Interval Sequences)

A state sequence $\bar{s} = (s_0, s_1, s_2, ...)$ is a possibly infinite sequence of states $s_i \subseteq AP$. Similarly an interval sequence $\bar{I} = (I_0, I_1, I_2, ...)$ a possibly infinite sequence of intervals such that:

- I_0 is left-closed and $l(I_0) = 0$;
- I_i and I_{i+1} are adjacent for all $i \geqslant 0$;
- every time $t \in \mathbb{R}_{\geqslant 0}$ belongs to some interval I_i.

At this moment, pairing state and interval sequences is going to result in having the requested timed state sequence. In these types of sequences, by virtue of the adjacency of the intervals in the interval sequence, one can perceive the state of the system at any possible time point. As so, this concept will form the fundamentals of the notation of the timed traces predicted by our network, more on this matter is found in Section 4.3.

Definition 3 (Timed State Sequence)

A timed state sequence $\tau = (\bar{s}, \bar{I})$ is a pair of a state sequence $\bar{s}$ and an interval sequence $\bar{I}$. We define a function $\tau^* : \mathbb{R}_{\geqslant 0} \to 2^{AP}$, which provides a system state at every time instant. That is, $\tau^*(t) = s_i$ for all $i \geqslant 0$ and $t \in I_i$.
We can also represent the timed state sequence τ by the possibly infinite sequence:

$$(s_0, I_0) \rightarrow (s_1, I_1) \rightarrow (s_2, I_2) \rightarrow ...$$

Eventually, we show under which directives a timed state sequence satisfies an MITL formula.

Definition 4 (MITL Semantics)
We determine the semantics of an MITL formula φ over a timed state sequence $\tau = (\bar{s}, \bar{I})$ inductively as follows:

$\tau \models p$	iff	$s_0 \models p$;
$\tau \models \neg\varphi$	iff	$\tau \not\models \varphi$
$\tau \models \varphi_1 \wedge \varphi_2$	iff	$\tau \models \varphi_1$ and $\tau \models \varphi_2$
$\tau \models \varphi_1 \,\mathcal{U}_I\, \varphi_2$	iff	for some $t \in I, \tau^t \models \varphi_2$, and for all $t' \in (0, t), \tau^{t'} \models \varphi_1$

We name the timed state sequence τ a model of the formula φ or say τ satisfies φ iff $\tau \models \varphi$. We also denote the set of models of φ as $L(\varphi)$, where φ is satisfiable iff $L(\varphi) \neq \emptyset$.

Example 4.1.1. In this example, we illustrate the semantics of all temporal operators available in MITL through displaying a prototype MITL formula and a comparable satisfying timed state sequence. The green color ushers when the timed state sequence becomes satisfying to the correlated formula.

$a \,\mathcal{U}_{[1,3]}\, b$	$a\ [0,2) \rightarrow b\ [2,3) \rightarrow a\ [3,4) \rightarrow \mathtt{True}\ [4,\infty)$
$\Diamond_{[1,3]}\ a \equiv \mathtt{true}\ \mathcal{U}_{[1,3]}\ a$	$b\ [0,2) \rightarrow b\ [2,3) \rightarrow a\ [3,4) \rightarrow \mathtt{True}\ [4,\infty)$
$\Box_{[1,3]}\ a \equiv \neg\Diamond_{[1,3]}\ \neg a$	$b\ [0,1) \rightarrow a\ [1,4) \rightarrow \mathtt{True}\ [4,\infty)$

$\triangle$

4.2 Generating MITL Formulas

We make use of the fact that the syntax of both LTL and MITL is almost identical. In fact, apart from comprising the next operator, LTL can be viewed as a special case of MITL where all intervals in the formula are of the form $[0,\infty)$. Therefore, we will use a familiar platform for LTL and ω-automata manipulation called Spot [32].

Spot acquires a tool called RandLTL [33] that provides a constructive and manageable measure to generate a great number of unique LTL formulas. With RandLTL, one can specify a size interval for the generated formulas along with a probability distribution for all the possible nodes in the formulas.

To generate the required number of MITL formulas for our project, we utilize RandLTL and subsequently interpret each resulting LTL formula into an MITL one. We achieve this by plainly concatenating random time intervals of a specified size; to each of the temporal operators located in the LTL formula.

Example 4.2.1. Using the command line interface of Spot, we can execute this RandLTL command to generate 5 unique LTL formulas:

```
>randltl -n5 a b c --tree-size=12 --ltl-priorities implies=1, equiv=1, U=1, F=1,
    G=1, or=1, and=1
G!(c -> Ga) U F(a & c)
c <-> (!(a <-> Fb) & (b | Fa))
G(a <-> (a | b)) -> F(a -> Ga)
a <-> (Gb <-> !(Gb | (c U b)))
(c & (b U (b | c))) -> Fb
```

In order to transform the LTL formulas into MITL, all we need is to append time intervals to the temporal operators:

```
G[1,4]!(c -> G[2,4]a) U[4,5] F[2,7](a & c)
c <-> (!(a <-> F[1,3]b) & (b | F[1,5]a))
G[2,7](a <-> (a | b)) -> F[1,4](a -> G[2,5]a)
a <-> (G[1,6]b <-> !(G[2,6]b | (c U[1,6] b)))
(c & (b U[4,6] (b | c))) -> F[2,3]b
```

△

To complete the construction of our data sets, we have yet to find a method to build satisfying solutions for the generated formulas. We likewise have to be able to investigate solutions against MITL formulas for satisfiability. In the next section, we display the two steps.

4.3 Constructing & Evaluating Solutions

For our setup, we are not only attempting to construct satisfiable traces for MITL formulas but also, we are taking into account that MITL formulas do not have unique solutions; therefore, we are seeking to validate the solutions that are predicted by the model, which are syntactically different from the solutions in the data set.

Spot, for example, constructs an equivalent Büchi Automaton for a given LTL formula. This automaton A_φ precisely accepts the language that defines the model of the LTL formula φ, i.e., $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$. One can then, by searching for an accepting run in A_φ, construct and evaluate satisfying traces for φ. However, Spot does not support MITL.

A similar decision procedure for MITL was introduced in [11]. In this work, the authors implemented a tool to translate MITL formulas into a timed automaton written in the XML format supported by the acclaimed model-checker UPPAAL [34]. Similar to Spot, it is presumed that one can use these automata to generate or evaluate traces. Nonetheless, this proved to be capricious, due to the restrictions of the UPPAAL interface which essentially is aimed at model checking, and as UPPAAL not being simply configurable.

A different approach we explored for solving MITL formulas was introduced in [35]. In this work, the authors translated MITL formulas into an intermediate logic called

Constraint LTL over clocks (CLTLoc). This logic has a decision procedure by reducing it into a decidable Satisfiability Modulo Theories (SMT) problem, then using an over-the-shelf SMT solver to generate the solution traces. The translation process, however, in addition to the time needed for the SMT solver to generate the final solution, is drawn out to the extent that solving voluminous numbers of formulas is considered unattainable. Adding the fact that this procedure does not support validating traces led us to search for other options.

At last, we reached the decision to utilize an MITL manipulation tool written in python, called Py-MTL [36] though it is unable to solve MITL formulas directly, yet with the assistance of Spot, permit us to jointly generate satisfying traces for MITL formulas and evaluate solutions.

Constructing Solution Traces

In order to construct a satisfying trace for a given MITL formula we use the discretization feature of Py-MTL. This feature translates the MITL formula into an equivalent LTL formula that solely consists of the temporal operator Next ($\bigcirc$), the propositional operators conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and atomic propositions. Afterward, we use Spot to construct a satisfying trace to this LTL formula (and thereby to the original MITL formula) if existent.

To better replicate a timed state sequence (Definition 3), we rewrite the output traces of Spot by merging identical successive steps into one step and pairing it with a left-closed and right-opened time interval. These intervals indicate the time span during which their corresponding step holds. Intervals in neighboring steps are also adjacent, allowing us to observe the values of the propositions at each possible instant.

We call this composition of the timed state sequences timed traces. For instance, the MITL formula $(\Box_{[1,2]} \neg a \wedge \Diamond_{[2,4]} a)$ is satisfied by the timed trace: $\neg a\ [1,3);\ a\ [3,\infty)$.

In the following example, we will expound on the procedure of solving an MITL formula using the afore-described method:

Example 4.3.1. Discretizing the MITL formula $(\Box_{[4,6]}(a \wedge b) \vee \Diamond_{[2,4]} c)$ with Py-MTL results in the following equivalent LTL formula:

$$\neg(\neg(\bigcirc^4(a \wedge b) \wedge \bigcirc^5(a \wedge b) \wedge \bigcirc^6(a \wedge b)) \wedge (\bigcirc^2 \neg c \wedge \bigcirc^3 \neg c \wedge \bigcirc^4 \neg c)) \tag{4.1}$$

Next, we use Spot to construct a satisfying trace for (4.1):

$$\mathtt{true};\ \mathtt{true};\ c;\ cycle\{1\} \tag{4.2}$$

As a final step, we merge the similar steps in (4.2) and add the corresponding intervals to acquire the following timed trace:

$$a\ [0,2);\ c\ [2,\infty) \tag{4.3}$$

For further implicated formulas, steps in the timed trace can comprise propositional formulas containing only the operators $\wedge$, $\vee$, and $\neg$. △

It is vital to note that when converting the Spot trace into a timed trace we discard any symbolic true steps. These steps, to begin with, allow for any combination of propositions in their respective positions in the trace. However, we concretize the trace by arbitrarily selecting one combination. This is done to adapt the timed traces to our evaluation process, described next, which can only operate with explicit traces. In other words, we avoid evaluating all possible concrete traces whose number can be exponential in the number of APs for each symbolic step. This procedure is essential, since many solutions predicted by the model vary from the solutions in the data set, in such cases, we have to evaluate these solutions for accuracy. This action would however yield testing our trained model on a sufficient number of unseen samples unfeasible.

It is worth mentioning here that, although this discretization approach assisted with Spot proves adequate, it still undergoes the impotence to solve MITL formulas that enclose infinite intervals. As a consequence, our data sets are going to comprise finite MITL formulas.

Evaluating Traces

One more feature of the Py-MTL tool is evaluating whether a timed trace satisfies a given MITL formula. In this case, one has to rewrite the timed trace into a python dictionary where each atomic proposition in the formula has its own list. This list comprises pairs of a time point and the logical value of the atomic proposition at this time point.

Example 4.3.2. To evaluate the satisfiability of trace (4.3) in Example 4.3.1 against the MITL formula $(\Box_{[4,6]}(a \wedge b) \vee \Diamond_{[2,4]} c)$ using the Py-MTL tool, we are required to transform the trace into the following python dictionary:

```python
trace = {
    'a': [(0, 1), (1, 1), (2, 0)]
    'b': [(0, 0), (1, 0), (2, 0)],
    'c': [(0, 0), (1, 0), (2, 1)]
}
```

△

We will employ this feature to ascertain the correctness of those predicted timed traces that are syntactically varied from the target solutions. This will permit us later in Section 5] to better assimilate to what extent our model has fathomed the semantics of MITL.

In the following section, we proceed to put the tools analyzed to work and generate our required data sets.

4.4 Data Sets

To demonstrate the competence of the Transformer neural network in predicting satisfying timed traces and to investigate the generalization properties of Transformers, we used the methods elucidated in Sections 4.2 and 4.3 to generate several data sets in two different fashions. Said fashions are presented in Sections 4.4.1 and 4.4.2.

Each of our data sets is split into three parts. The first part contains 80% of all samples and is used to train the model. The second part is a validation set used to evaluate the model after each epoch to help avoid overfitting, it contains 10% of the samples. As for the remaining 10% of the samples, they are gathered in a held-out test set that is used as a final resolution of accuracy for the wholly trained model.

We also elected the Polish notation as the format of the formulas and the steps of the timed traces. This notation permits a unique representation of logical formulas without the need for parenthesis.

The maximum size of the time intervals in the formulas is limited to a specific bound for each of the data sets. The rationale behind limiting the size of the intervals is to obviate constructing MITL formulas whose corresponding equivalent discretizations are too large. This is due to the fact that constructing satisfying traces for such formulas can easily become unfeasible under the specified timeout limit of two seconds (one second for the discretization process, and one second for constructing the satisfying trace). For instance, 77.71% of the formulas in our smallest data set `MITLRandom35Full` (displayed later on) have an equivalent discretization formula larger than 1K in size and 10.55% larger than 10K. For some formulas, the discretization sizes even exceeded 100K.

Four Intel Xeon CPU E7-8867 v4 processors were used for the generation of all data sets, in which 140 processes were utilized to generate samples simultaneously.

4.4.1 Specification Pattern

In this data set, we construct samples from a pool of 139 temporal specification patterns collected from different sources in the literature [37, 38, 39, 40, 41]. These patterns contain both derived and non-derived propositional and temporal operators as well as up to six unique atomic propositions. Here are three randomly selected patterns:

```
G((a & !b & Fb) -> ((c -> (!b U (!b & d))) U b))
!(Ga | Gb | (G(a | FGc) & G(b | FG!c)))
FGa | GFb
```

To generate sufficient unique samples out of this limited number of specifications, we conjoined various patterns after randomly substituting their atomic propositions and appending the necessary time intervals to the temporal operators. We halted the conjunction process once the size of the resulting formula either succeeded 82 or once our trace construction method, from Section 4.3, timed out (> two seconds).

For this data set, time intervals were randomly selected under the constraint that their size does not exceed 100. Through using this approach, we successfully generated 319 082 samples in 18.27 hours.

In the following, we illustrate two randomly chosen samples from the MITLPattern82 data set. Although the model is trained only on samples written in the polish notation, we provide the infix notation for the purpose of clarity:

```
formula (polish): &->F[1,6]cU[29,42]ec|U[36,43]F[0,1]cb!F[0,1]U[10,41]c&cb
formula (infix) : F[1,6]c -> (e U[29,42] c) & (F[0,1]c U[36,43] b) | !F[0,1]
                  (c U[10,41] (c&b))
trace   (polish): !e [0,1); !c [1,∞)

formula (polish): &F[21,57]|faG[16,22]->bU[3,26]aU[0,6]!a|cd
formula (infix) : F[21,57](f | a) & G[16,22](b -> (a U[3,26] (!a U[0,6] (c | d)))
trace   (polish): f [0,16); !b [16,21); |&a!b&!bf [21,22); !b [22,∞)
trace   (infix) : f [0,16); !b[16,21); (a & !b) | (!b & f) [21,22); !b [22,∞)
```

4.4.2 Random Formulas

Using the RandLTL tool disclosed in Section 4.2, we arbitrarily generated LTL formulas and later converted those LTL formulas into MITL by appending random integral intervals of a delimited size to the temporal operators (See Example 4.2.1).

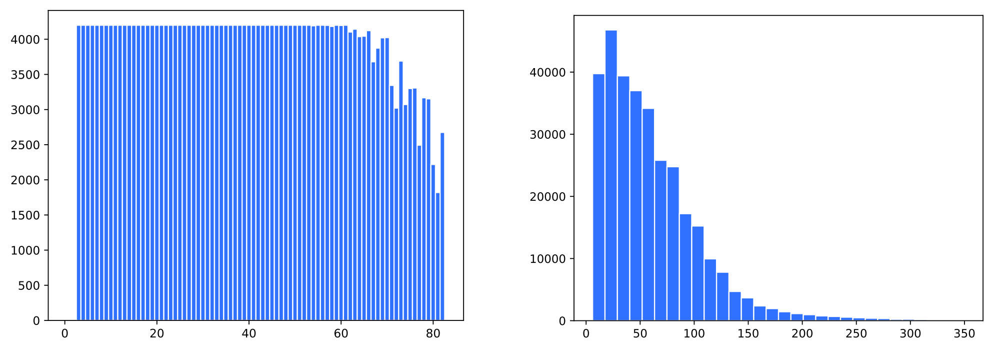
In this approach, the generated formulas can accommodate up to five different atomic propositions. For the node distribution, we select equal weights on all the temporal and propositional operators, as well as on the constants True and False. Atomic propositions, however, can appear with a higher probability of three times. Moreover, we filter the generated formulas to boost the consistency of the distribution in size.

In the following paragraph, we list the data sets generated by using this random approach. Intervals here were bounded to size 120.

MITLRandom35Full This data set encloses 654 984 samples. Formulas here have a maximal size of 35 and contain both derived and non-derived temporal and propositional operators, along with the constants True and False.

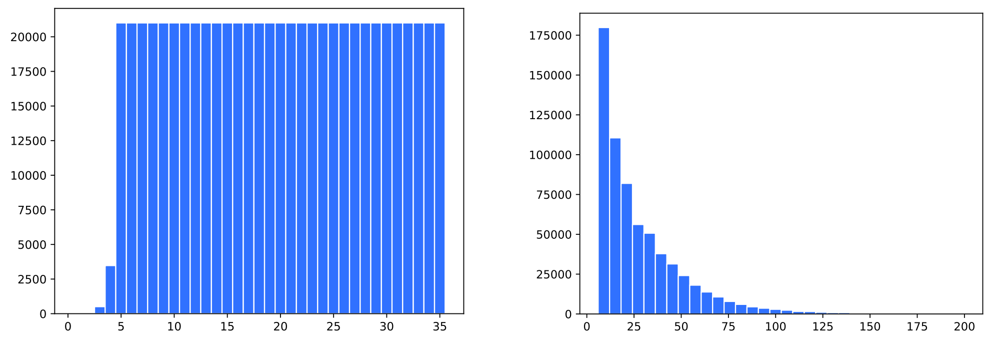
MITLRandom35Simple This data set encloses 624 930 samples. This data set varies from the MITLRandom35Full data set by the absence of derived operators. Here, only the operators $\wedge, \neg, \mathcal{U}$ are allowed.

MITLRandom82Simple As the name suggests, this data set encloses formulas with a maximal size of 82 that in its turn only contain non-derived operators. It consists of only 10K samples for the purpose of evaluating the accuracy of our models when challenged with larger-sized formulas.



(a) (b)

Figure 4.2: Size distribution for formulas (a) and traces (b) of the `MITLPattern82` data set. The size of the formulas and traces is on the x-axis ,while the number of samples is on the y-axis.



(a) (b)

Figure 4.3: Size distribution for formulas (a) and traces (b) of the `MITLRandom35Full` data set. The size of the formulas and traces is on the x-axis ,while the number of samples is on the y-axis.

The evident size distribution of two of our largest data sets can be observed in Figures 4.2 and 4.3. We consider the size of an MITL formula to be the number of nodes in the original LTL formula plus the number of added intervals, for example, the formula $(\Diamond_{[1,3]} \neg a)$ has the size four. Additionally, we indicate the different data sets presented in Sections 4.4.1 and 4.4.2 in Table 4.4.

4.4.3 Scaled Data Sets

Limiting the size of the time intervals in the previous data sets, though necessary, imposes restrictions on the number of unique intervals established in the data sets. Yet, this limitation contradicts actual real scenarios where time intervals can be of any size. In order to compensate for this limitation and to be able to evaluate the performance of

Data set	#Samples	Int. Size	#APs	OPs
MITLPattern82	319 082	100	6	$\mathcal{U}, \square, \diamond, \wedge, \vee, \rightarrow, \Leftrightarrow, \neg$
MITLRandom35Full	654 984	120	5	$\mathcal{U}, \square, \diamond, \wedge, \vee, \rightarrow, \Leftrightarrow, \neg$
MITLRandom35Simple	624 930	120	5	$\mathcal{U}, \wedge, \neg$
MITLRandom82Simple	10K	120	5	$\mathcal{U}, \wedge, \neg$

Table 4.4: The data sets used for training and evaluating the Transformer; together with the number of samples, maximum interval size, number of possible atomic propositions, and employed operators. The maximum size of the formulas can be deducted from the data sets labels.

the Transformer when faced with bigger, more diverse time intervals, we scale existing data sets into ones that possess such intervals.

This is achieved as follows: for each pair of formulas and its satisfying timed trace, we multiply both of the end-points of the time intervals with a single scalar. The scalar is selected randomly for each pair in the data set, with the limitation that none of the scaled time intervals exceeds the new selected size limit. As a result, we gain a new data set with the same number of samples, still with more variety of intervals due to the increased allowed size done by scaling. To demonstrate the validity of the resulting data sets, we introduce the following Proposition:

Proposition 1. *Multiplying the endpoints of all time intervals in an MITL formula ϕ and a satisfying timed state sequence τ with a single scalar μ, yields a new formula ϕ' and a new timed state sequence τ' that satisfies ϕ'.*

Proof. Since the semantics of MITL is defined inductively (Definition 4), and since only temporal operators acquire a time interval, it suffices to prove the lemma for the basic case ($\phi = \varphi_1 \, \mathcal{U}_I \, \varphi_2$):

Based on the assumption $\tau = (\bar{s}, \bar{I}) \models \phi$, there exists a time point $t \in I \cap I'_i$, with $\tau(t) \models \varphi_2$, where $I'_i \in \bar{I}$. It also must hold that for all $t' \in (0, t)$, $\tau(t') \models \varphi_1$.

Since scaling intervals in τ does not change their corresponding states, we have $t^* \in (I^*) \cap (I'^*_i)$, and $\tau'(t^*) \models \varphi_2$. Here the superscript * represents the time points and intervals scaled by μ. For the same reason we also have for all $t'^* \in (0, t^*)$, $\tau'(t'^*) \models \varphi_1$. Based on the semantics of MITL, we conclude that $\tau' \models \phi' = \varphi_1 \, \mathcal{U}_{I^*} \, \varphi_2$. □

In the following chapter, we will use this approach to scale some of the data sets introduced above and compare the performance of the Transformer for different time interval size limits.

Chapter 5

Experiments & Results

Based on the DeepLTL tool [42], we implement a Transformer neural network and train several models to predict satisfying timed traces for MITL formulas from the different data sets presented in Section 4.4. Alongside the Transformer's implementation, DeepLTL utilizes beam search [43], which is a heuristic best-first search algorithm that aids to decode the best possible output prediction up to predetermined beam size. For the following experiments, we used a beam size of two.

Since an MITL formula can have multiple accurate solutions, we differentiate between two accuracy measures when testing our models, syntactic accuracy and semantic accuracy. Syntactic accuracy plainly means that the prediction made by the model is identical to the target solution in the data set. Semantic accuracy, on the other hand, construes the percentage of predicted solutions that are, although not identical to the ones in the data set, still considered a valid solution to the formula. To better evaluate the performance of our trained models, we also differentiate between false solutions and syntactically invalid solutions.

All models in our experiment were trained using the popular Adam optimizer [44] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. Moreover, we used 4K warm-up steps for the computation of the learning rate. The configuration parameters of Adam and the number of warm-up steps, both were proposed in the original Transformer paper [14].

To circumvent overfitting; an occurrence where the model fits its training data too precisely it loses its generalization abilities; we employ the early stopping method. While training, the model is evaluated on the validation set after each epoch. When the validation error fails to decrease for more than a predetermined number of epochs, the learning process is interrupted and the model with the lowest validation error is restored. In this project, we elected a threshold of eight epochs for early stopping.

All our models were trained and tested on a single NVIDIA A100 SXM4 GPU with a 40 GB memory size.

Embedding	Layers	Heads	FC size	Batch	Epochs	Syn. Acc.	Sem. Acc.
128	4	4	512	100	175	67.82%	93.29%
128	4	4	512	100	300	69.86%	93.61%
256	4	8	512	100	175	68.13%	92.74%
128	4	4	512	512	134	71.18%	94.45%
128	4	4	512	512	203	72.53%	95.31%
128	5	6	1024	512	134	74.36%	**95.97%**
128	5	6	1024	750	151	74.03%	95.69%
256	5	6	1024	512	77	73.10%	94.88%
256	5	6	1700	512	86	73.91%	95.26%
128	6	8	1024	512	131	71.58%	94.52%
128	6	8	1024	512	142	72.03%	94.66%

Table 5.1: The hyperparameter analysis results of testing different Transformers on the `MITLRandom35Full` data set.

5.1 Hyperparameter Analysis

In this section, we evaluate the performance of multiple Transformers, each trained and tested on the `MITLRandom35Full` data set to examine the impact of the most significant hyperparameters. The principal parameters we will be focusing on, are the number of layers in each of the encoder and the decoder stacks, the number of heads in each layer, the size of the embedding vectors, the dimensionality of the fully-connected FFNN, and finally the number of examples processed in each training step (batch size). The maximum encoding size, hence the maximum number of tokens in an input sequence, was confined to 90 and the maximum decoding size was confined to 200. Table 5.1 summarizes the training results of the different Transformers along with the chosen values of their hyperparameters.

Starting with a Transformer with four layers, four heads, and a small training step of 100 examples, the results were already encouraging. With an embedding vector size of 128, this Transformer predicted the exact target solutions for 67.82% of the test samples and accurate yet new solutions for 25.47% of these samples. When training the same Transformer for more epochs or increasing its embedding size and the number of heads, the syntactic accuracy; although improved; the overall accuracy remarked little to no improvement. However, increasing the batch size did not only decrease the training time (182 seconds vs 249 seconds per epoch) but also lead to a better overall accuracy despite learning for fewer epochs.

Later in the table, we see that equipping a Transformer with five layers and six heads, in addition to increasing the dimension of the FFNN to 1024, yields our best-performing model. Even while keeping the embedding size at 128 this Transformer achieves an overall accuracy of 95.97%, with most of the gains being the result of the improvement of the syntactic accuracy.

Further experimenting with bigger Transformers in terms of the number of layers, heads, or size of embedding, FFNN brought no advantage. We, therefore, stick with the former more efficient Transformer for all of the forthcoming experiments.

5.2 Solving Pattern MITL Formulas

To better assess how the Transformer architecture would perform on specifications presented in practice, we trained a model of the same size as in the last section on the `MITLPattern82` data set.

Since formulas in this data set are considerably bigger than the ones we experimented with up to this point, we expanded the maximum encode and decode sizes for this Transformer to 175, and 350 respectively.

After training the model for 139 epochs, we performed two experiments. Firstly, we tested the trained model on the held-out segment of the `MITLPattern82` data set containing 10K examples. Secondly, we tested the same model on a set of 4 160 formulas from the same distribution, but for which the classical algorithm failed to construct a solution within the determined timeout limit of two seconds. Figure 5.2 terms the results of these experiments.

The results of the first experiment appeared slightly less accurate than the results of the previous models we tested thus far. This is, however, foreseen as a result of the increase in the size of the formulas. Regardless, results were satisfactory with the model predicting 93.66% of solutions precisely. The percentages of syntactic and semantic accuracy were similar to those of the previous experiments, 73.80% and 19.86% respectively. Only rarely did the model predict invalid traces (0.12%).

In the second experiment, the more structurally involved formulas; for which the construction of a satisfying timed trace timed out, evinced more challenging for the trained model. Out of the 4 160 formulas, the model predicted correct solutions for 3 068 formulas leading to an accuracy of 73.75%. The model was also more likely to predict syntactically invalid traces (2.60%). The superior speed of the Transformer is also worth noting. Compared to the classical algorithm that timed out after two seconds, the Transformer took around 10 milliseconds to predict a solution for each formula.

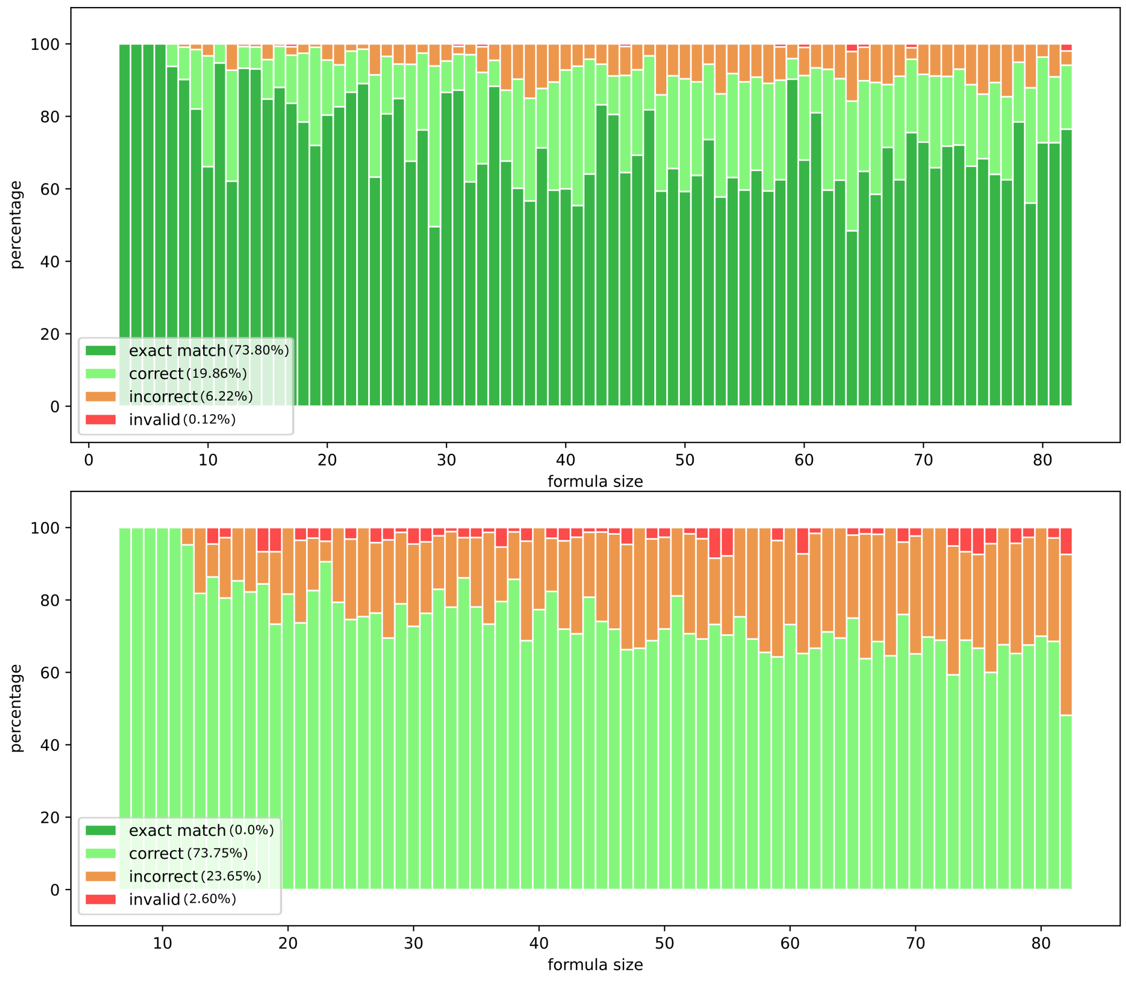


Figure 5.2: Performance of a model (only trained on `MITLPattern82`) that was tested on a held-out test set (on the top) and on a set of formulas for which our trace construction method timed out (on the bottom).

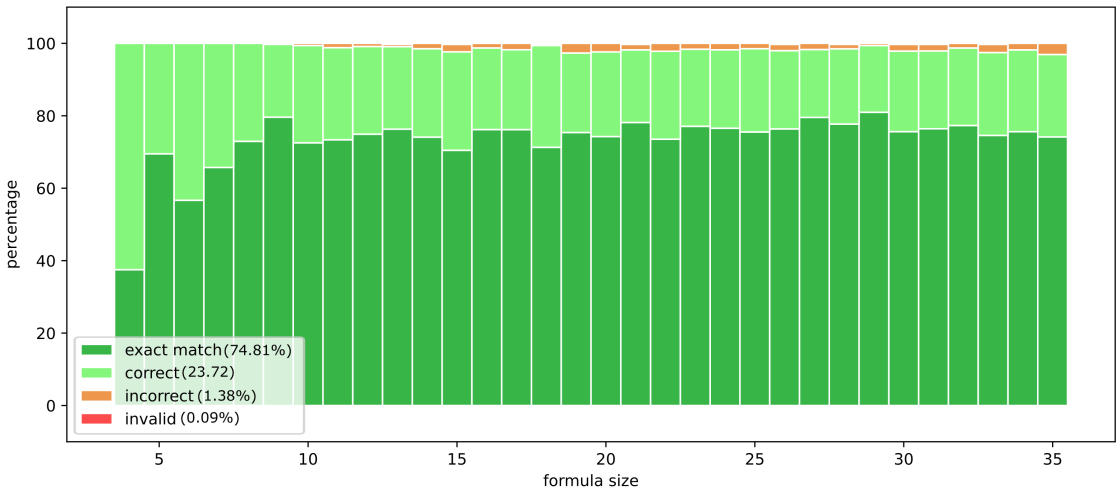


Figure 5.3: Results of testing our best performing model on the `MITLRandom35Simple` data set. The syntactic accuracy is illuminated in green and the semantic accuracy in light green.

5.3 Solving Randomly Generated MITL formulas

In order to achieve even finer results, we attempt to simplify the underlying problem by training and testing the model on formulas containing only the non-derived operators $\wedge, \neg$ and $\mathcal{U}$. This is especially significant since any logical formula can be easily converted into an equisatisfiable equivalent enclosing only non-derived operators.

For this experiment, we used the same hyperparameter settings proven to be the best in Section 5.1 and trained a model on the `MITLRandom35Simple` data set for a total of 87 epochs. We then tested this model on 10K samples from the held-out test set of the same data set.

The elimination of the derived operators, did indeed, simplify the problem for the Transformer. This time, The model was capable of predicting correct solutions for 9853 formulas out of the total 10K test samples, resulting in an overall accuracy of 98.53%. 74.81% of these accurately predicted solutions were syntactically identical to the target solutions found in the data set, whereas, 23.72% were semantically correct. Still, the model predicted false solutions for only 147 out of the total 10K tested samples, with solely nine of these false predictions being syntactically invalid. A demonstration of the test results can be seen in Figure 5.3.

5.4 Generalization Properties

Presented in Figure 5.3, our model provided accurate solutions for 23.72% of the formulas in spite of these solutions being dissimilar from the ones generated by the classical algorithm. This percentage proposes that the model is learning the semantics of the underlying problem, rather than only attempting to match the outcome of the classical algorithm. In this section, we desire to further investigate the competence of our model when faced with out-of-distribution formulas. Thereby, this will help reveal the generalization potentials of the model when coming across such unfamiliar input.

5.4.1 Generalizing to the Semantics of MITL

To begin with, we want to evaluate the model from Section 5.2, which is only trained on `MITLPattern82`, on the `MITLRandom82Simple` data set. Though both of these data sets comprise formulas of the same size, the random nature of the latter leads the formulas to be more concentrated in the number of temporal and propositional operators. Especially in the number of the $\mathcal{U}$ operator, since it is the only temporal operator allowed in this data set. This concentration in its turn leads to more nesting of operators in the formulas of the `MITLRandom82Smiple` data set.

Predictably, the syntactic accuracy; after running this experiment on 10K test samples, was very low (2.24%). We attribute this to the aforementioned differences between the two data sets. Similarly, The model presented a slightly higher tendency toward predicting invalid traces. These invalid predictions comprise 6.15% of the total predictions. Notwithstanding the two aforementioned mishaps, the model was able to achieve adequate total accuracy of 58.46% by predicting semantically accurate solutions.

Hither, we demonstrate three selected test samples with the corresponding predicted timed traces to manifest the possible semantic understanding gains of the model:

```
input : !a U[8,17] b & TRUE U[23,29] !b & !(d U[15,23] !a U[9,28] c)
output: !a & !b [0,8); !a & !b [8,9); !a & b [9,10); !a [10,23); !a & !b [23,24);
        !a & b [24,∞)
target: !a & !d [0,1); !a [1,8); !a & b [8,9); !a [9,18); a [18,23); !b [23,∞)

input : !b U[3,21] d
output: !b [0,3); !b & !d [3,4); !b & d [4,∞)
target: !b [0,3); !b & !d [3,13); !b & d [13,∞)

input : b & e & TRUE U[26,27] a & e U[23,24] b & !(( c & e ) U[17,18] d)
output: b & !c & e [0,1); e [1,23); !b & e [23,24); b & e [24,25); e [25,26); !a
        & e [26,27); a & e [27,28); e [28,45); !b & e [45,46); b & e [46,∞)
target: b & c & e [0,1); c & e [1,17); c & !d & e [17,18); !d & e | !c & e
        [18,19); e [19,23); b & e [23,24); e [24,25); b [25,26); a [26,∞)
```

5.4.2 Generalizing To Bigger MITL Formulas

Following the results from the previous section, we desire to further evaluate our model on another type of unseen formulas, i.e. on formulas of bigger sizes. We ,therefore, will test our model from Section 5.3, that is only trained on `MITLRandom35Simple`, on the `MITLRandom82Simple` data set.

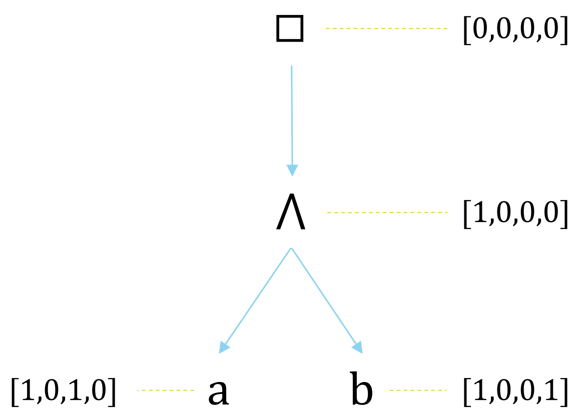


Figure 5.4: Tree positional encoding of the formula $(\Box_{[1,4]}(a \wedge b))$. All the tokens of the interval $[1, 4]$ share the corresponding encoding vector as the operator they belong to.

The `MITLRandom82Simple` data set contains formulas that are more than double the size of what the model has encountered during training. Still, the model in this experiment was able to predict correct solutions to 93.12% of the formulas, with 28.31% of the correct predictions diverging from the target solutions. Conversely, 6.88% of the solutions suggested by the model were false with only 0.17% thereof being invalid.

As an attempt to improve the previous result, we retrained an identical model on the same `MITLRandom35Simple` data set, this time using tree-positional encoding in place of the standard positional encoding we established in 3.2.2.

The authors in [45] introduced tree-positional encoding as a more apt technique for models optimized for tree-based problems. The basic idea is that the position of a token in the input sequence is determined by its node placement in the corresponding syntax tree. For example, the atomic proposition b in the simple MITL formula $(\Box_{[1,4]}(a \wedge b))$ is encoded with the vector $[1, 0, 0, 1]$, where $1, 0$ represents traversing one step to the left and $0, 1$ one step to the right in the syntax tree. Figure 5.4 presents the full encoding of the formula.

As a result of utilizing tree-positional encoding, the already good performing model witnessed little improvement. Specifically, the overall accuracy increased only by 0.49% reaching 93.61%. Upon deeper inspection, however, we observed that tree-positional encoding indeed improved the ability of the model to generalize to longer formulas rather remarkably. For instance, when restricting our test data set for this experiment to only formulas that are bigger than the ones seen during training (>35), the superiority of the tree-positional encoding becomes more evident. As it were, our model predicted the accurate solutions for 91.32% of the formulas, whereas a model trained with the standard positional encoding had an accuracy of 86.89%. However, this boost in accuracy was thwarted by the decrease in performance in relation to formulas of familiar sizes upon utilizing tree-positional encoding. We elucidate this actuality in Figure 5.5.

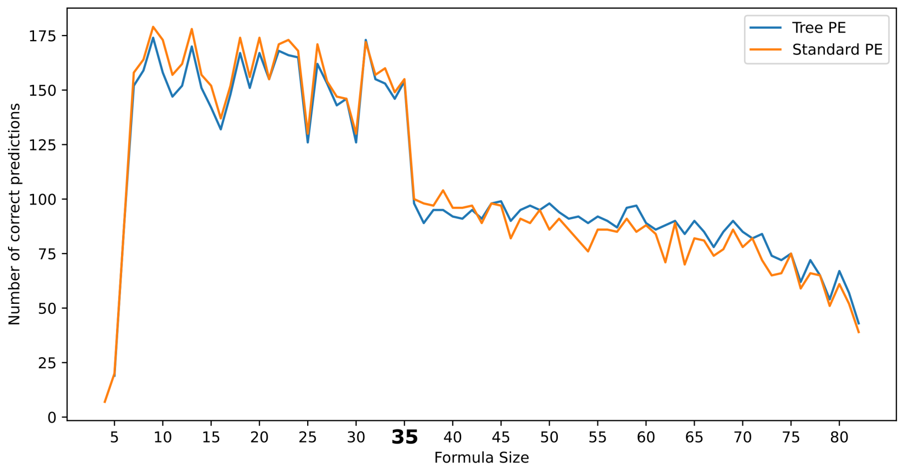


Figure 5.5: Performance of a model utilizing tree-positional encoding against a model solely relying on standard positional encoding. We observe finer generalization ability to formula sizes that the model was not trained on (> 35).

5.5 Solving Scaled MITL Formulas

In this experiment, we desire to work out the impact that the size of the time intervals has on the performance of the Transformer. We want to dismiss any suspicions that the results of the previous experiments are uniquely valid when the time intervals are small. Therefore, we apply the scaling approach introduced in Section 4.4.3 to generate multiple scaled versions of the `MITLRandom82Simple` and `MITLPattern82` data sets. We, after, retrain and test our best performing model on these scaled versions of the two data sets.

Since the scaled data sets are identical to their original equivalent up to the time intervals, we will study the effects that the number of unique integers; and consequently the number of unique intervals; have on the model's ability to predict precise solutions. We will additionally compare the number of epochs the model trained for, and the changes in relation to the semantic and syntactic accuracy. All results are listed in Tables 5.6 and 5.7.

As a matter of course, we notice an increase in the number of unique intervals proportional to the increase in the size limit of the intervals. The observed increase in the number of intervals is followed by an increase in the number of trained epochs for each model before the early stopping condition was met. We attribute the latter increase to the building up of information to learn caused by the new intervals.

As for the results, in the case of the `MITLRandom35Simple` data set, increasing the interval size was responsible for a reduction in both the syntactic and semantic accuracy. The accuracy result of the original data set with intervals of size 120, has dropped from 98.53% to a minimum of 85.74% for the scaled version with intervals up to 100K in size.

interval size	#integers	#intervals	#epochs	accuracy	syn. acc.	sem. acc.
120 **(orig.)**	121	10 011	87	98.53%	74.81%	23.72%
5K	5 665	295 890	121	93.20%	70.87%	22.33%
10K	10 657	473 741	132	90.33%	70.30%	20.03%
25K	24 317	836 122	136	86.95%	69.81%	17.14%
100K	82 152	1 645 391	168	85.74%	69.02%	16.72%

Table 5.6: Performance results of multiple models that are trained and tested on different scaled versions of the `MITLRandom35Simple` data set.

interval size	#integers	#intervals	#epochs	accuracy	sen. acc.	sem. acc.
100 **(orig.)**	101	9 566	139	93.66%	73.80%	19.86%
100K	84 814	1 636 788	168	73.71%	53.42%	20.29%

Table 5.7: Performance of two models, the first model is trained on the original `MITLPattern82` data set, while the second one is trained on a scaled version.

In the case of the `MITLPattern82` data set, only the syntactic accuracy has declined, yet the semantic accuracy has rather improved. The overall accuracy dropped 19.95% in the original data set with intervals of size 100 to reach 73.71% when stretching the intervals up to 1K times.

To emphasize the differences in size between the original data sets and their biggest scaled versions, we indicate the encoding and decoding sizes for each of the experiments: The encoding and decoding maximum length for the original `MITLRandom35Simple` data set was set to 90, 200 respectively. These had to be gradually increased for each scaled version, reaching up to 140, 320 for the 100K scaled version. Likewise, the 175, 350 encoding/decoding length limitations used for the `MITLPattern82` data set, were raised up to 240, 380 for the scaled version.

It is also important to state that the decline in the semantic accuracy noted in Table 5.6 could have been exaggerated by the need to filter some of the test samples. This is caused by setting a timeout limit for validating the predicted traces that contain huge intervals. For instance, the validation process of 1 273 out of the 10K test samples of the biggest scaled version of the `MITLRandom35Simple` data set has timed out, even when selecting a timeout limit of 30 seconds. Those predictions were deemed as incorrect, possibly negatively influencing the semantic accuracy results of the model, but not the syntactic accuracy, since syntactic accurate predictions do not need to be validated. The same applies to the scaled version of the `MITLPattern82` data set. In which case, 1 840 of the test samples were deemed as incorrect due to the timeout limit, possibly preventing the semantic accuracy from improving even more.

All things considered, we believe that the increase in interval size was not overly challenging for the Transformer. The dwindling in the syntactic precision of the predictions, for example, could be illustrated by the expansion in the size of the problem, while the reduction in the semantic accuracy could be, to a certain extent, linked to the inability to evaluate all the predicted solutions as explained above.

Ultimately, we assess and examine three selected test samples along with their semantically correct predicted traces:

```
input : (F[7959,30699]a -> !a U[1137,35247] (!a & c & F[0,1137](!a U[26151,31836]
        b))) & (f | b U[27288,46617] f)
output: a & f [0,1137); a [1137,7959); !a [7959,∞)
target: a & b & !f [0,1137); b [1137,7959); !a & b [7959,27288); !a & b & !f
        [27288,28425); !a & b & f [28425,29562); !a [29562,∞)

input : (b & TRUE U[23430,32802] !(a & !c)) U[1562,35926] !(!c U[21868,54670] b)
output: b [0,1562); b & c [1562,3124); b [3124,23430); !a | c [23430,∞)
target: b [0,1562); b & c [1562,3124); b [3124,6248); b [6248,23430); !a | c
        [23430,∞)

input : !d U[12208,45780] b U[39676,54936] e & !(!c U[0,18312] (c & (a & b)
        U[3052,51884] !e) & !(e U[15260,54936] d U[0,30520] a))
output: !d [0,12208); b & !d [12208,27468); b [27468,51884); b & e [51884,54936); b
        [54936,∞)
target: !d [0,12208); b & !d [12208,21364); b [21364,51884); b & e [51884,54936); b
        [54936,∞)
```

In the first sample, the output of the model comprises a less complicated solution, than the one constructed by the data generator. Precisely in the second conjunct of the formula, the model chose to satisfy the first disjunct (f) instead of the more involved sub term $(b\,\mathcal{U}_{[27288,46617]}\,f)$.

In the preceding two samples, we notice the Transformer predicting similar steps to the target solutions while predicting different corresponding time intervals. Even more interestingly in the second sample, the Transformer discarded an unnecessary stutter step in the trace, while precisely merging the two intervals.

Chapter 6

Conclusion

In this project, we contributed to the continuously growing field of machine learning that supports logical reasoning. Motivated by the favorable results of works such as [4] and [5] we explored the competence of neural networks in the realm of the more expressive real-time logic. With respect to the neural network architecture, we appointed the most prominent sequence to sequence neural network architecture, the Transformer. We consequently chose MITL as the elemental logic for our project. The end goal was to fulfill the inquiry, can the Transformer neural network predict accurate solutions in the form of timed traces to a given MITL formula?

Attempting to answer this question, we started by introducing the methodology we used to adequately generate and construct satisfying timed traces for MITL formulas. We, as well, defined how we can validate whether the predicted timed traces satisfy their corresponding formulas. Applying these methods, we then proceeded to generate the necessary data sets to train and evaluate our neural network. Meanwhile, we aimed to generate numerous data sets, with different formula sizes, and different fundamental structural patterns. The prime objective was attempting to disclose the volume of comprehension that the Transformer has for the semantics of the problem. Moreover, allowing us to reveal the generalization potentials of the Transformer while adding the value of strengthening the confidence in our implemented Transformer model, shall it be faced with real-world problems.

Commencing with training the Transformer on a data set comprising formulas that follow patterns found in literature, our best performing model correctly solved 93.66% of the test samples. Furthermore, it managed to correctly predict solutions for 73.75% of a subset of formulas that our data generator failed to solve under the given timeout limit. In like manner, the Transformer accomplished even better success once trained and tested on smaller sized and randomly generated formulas. It achieved a total accuracy of 98.53%. In both experiments, around 20% of the predicted solutions were semantically accurate.

To assert the generalization properties of the Transformer, we conducted two cross-data sets experiments. In these experiments, the model's performance was tested on a different data set than the set it was trained on. Thus, a model only trained on formulas following strict patterns, still performed well when predicting solutions for formulas that are randomly generated. The variations between the two data sets resulted in the model failing to make syntactically accurate predictions (only 2.24%). However, The model still managed to predict correct solutions for 58.46% of the formulas as a virtue of its ability to make semantically correct predictions.

Not only did the Transformer generalize to the semantics of the logic, but also to bigger-sized formulas. A very useful feature in practice, taking into consideration that bigger formulas present a greater challenge for classical algorithms. As an example, a model trained on formulas of size 35, accomplished significant results on formulas more than double that size (82). It prevailed in solving 93.61% of the formulas correctly. We followed this experiment with a comparison between the traditional positional encoding of the input and output sequences and the more natural tree-positional encoding. This comparison confirmed the tree-positional encoding being more convenient when working with MITL formulas, leading to enhanced results, principally when the Transformer had to deal with formulas bigger than those it is familiar with.

Finally, we wanted to evaluate the Transformer when it was trained and tested on data sets that comprise more variety of intervals both in quantity and size. An evaluation we could not conduct in the previous experiments due to the infeasibility of constructing satisfying traces for formulas containing big intervals in a reasonable time. We, therefore, introduced a technique where we scale an existing formula and satisfying trace pair into a valid pair that contains bigger time intervals. This technique made it feasible to obtain data sets that include formulas with much bigger time intervals, skipping the trouble of having to construct new satisfying traces for these formulas. Rerunning the training and evaluation routine of our best-performing model on scaled versions of our original data sets exhibited only a small decrease in the overall accuracy. Strictly when increasing the size limit of the time intervals 1K times, did the accuracy decrease by over 19%.

Taking account of all results; deep learning proves auspicious in the field of logical reasoning for real-time systems. The results suggest that deep learning might advocate evolving hybrid tools, particularly, classical algorithms augmented with deep learning that relies on more expressive real-time logic such as in this case MITL. Nevertheless, we cannot ignore the fact that the efficiency of the classical approaches is still fundamental as neural networks are prone to making false predictions. In such a case, the slower yet complete classical algorithms have to step in. Yet most of the time, the Aforementioned hybrid tools would benefit from superior performance speeds owing to the velocity at which the Transformer can predict solutions, combined with the fact that it frequently predicts correct solutions. The sole action still needed is to validate the accuracy of the predictions, a procedure that is especially vital in the case of critical systems, in which mistakes cannot be tolerated. This, however, is generally more simple than having to construct the solution afresh.

Bibliography

[1] Wenying Zhou, Y. Yang, Cheng Yu, Juxian Liu, X. Duan, Z. Weng, Dan Chen, Q. Liang, Qin Fang, Jiaojiao Zhou, H. Ju, Z. Luo, Weihao Guo, Xiaoyan Ma, Xiao-yan Xie, Ruixuan Wang, and Luyao Zhou. 2021. Ensembled deep learning model outperforms human experts in diagnosing biliary atresia from sonographic gallbladder images. *Nature Communications*, 12.

[2] Antoine Buetti-Dinh, Vanni Galli, Sören Bellenberg, Olga Ilie, Malte Herold, Stephan Christel, Mariia Boretska, Igor V. Pivkin, Paul Wilmes, Wolfgang Sand, Mario Vera, and Mark Dopson. 2019. Deep neural networks outperform human expert's capacity in characterizing bioleaching bacterial biofilm composition. *Biotechnology Reports*, 22, e00321. ISSN: 2215-017X. DOI: `https://doi.org/10.1016/j.btre.2019.e00321`. `https://www.sciencedirect.com/science/article/pii/S2215017X18301954`.

[3] The AlphaStar team. 2019. Alphastar: mastering the real-time strategy game starcraft ii. Retrieved 09/11/2021 from `https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii`.

[4] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. `https://openreview.net/forum?id=HJMC%5C_iA5tm`.

[5] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. 2021. Teaching temporal logics to neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. `https://openreview.net/forum?id=dOcQK-f4byz`.

[6] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1*

November 1977. IEEE Computer Society, 46–57. DOI: 10.1109/SFCS.1977.32. https://doi.org/10.1109/SFCS.1977.32.

[7] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. 1996. The benefits of relaxing punctuality. *J. ACM*, 43, 1, 116–146. DOI: 10.1145/227595.227602. https://doi.org/10.1145/227595.227602.

[8] Dejan Nickovic, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2020. AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. *Int. J. Softw. Tools Technol. Transf.*, 22, 6, 741–758. DOI: 10.1007/s10009-020-00582-z. https://doi.org/10.1007/s10009-020-00582-z.

[9] Doron Drusinsky. 2000. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings* (Lecture Notes in Computer Science). Klaus Havelund, John Penix, and Willem Visser, editors. Volume 1885. Springer, 323–330. DOI: 10.1007/10722468_19. https://doi.org/10.1007/10722468_19.

[10] Oded Maler and Dejan Nickovic. 2004. 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* (Lecture Notes in Computer Science). Yassine Lakhnech and Sergio Yovine, editors. Volume 3253. Springer, 152–166. DOI: 10.1007/978-3-540-30206-3_12. https://doi.org/10.1007/978-3-540-30206-3_12.

[11] Thomas Brihaye, Gilles Geeraerts, Hsi-Ming Ho, and Benjamin Monmege. 2017. Mightyl: A compositional translation from MITL to timed automata. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I* (Lecture Notes in Computer Science). Rupak Majumdar and Viktor Kuncak, editors. Volume 10426. Springer, 421–440. DOI: 10.1007/978-3-319-63387-9_21. https://doi.org/10.1007/978-3-319-63387-9_21.

[12] Oded Maler, Dejan Nickovic, and Amir Pnueli. 2006. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings* (Lecture Notes in Computer Science). Eugene Asarin and Patricia Bouyer, editors. Volume 4202. Springer, 274–289. DOI: 10.1007/11867340_20. https://doi.org/10.1007/11867340_20.

[13] Claudio Menghi, Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2020. Model checking MITL formulae on timed automata: A logic-based approach. *ACM*

Trans. Comput. Log., 21, 3, 26:1–26:44. DOI: 10.1145/3383687. https://doi.org/10.1145/3383687.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762. arXiv: 1706.03762. http://arxiv.org/abs/1706.03762.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805. arXiv: 1810.04805. http://arxiv.org/abs/1810.04805.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165. arXiv: 2005.14165. https://arxiv.org/abs/2005.14165.

[17] Frederik Schmitt, Christopher Hahn, Markus N. Rabe, and Bernd Finkbeiner. 2021. Neural circuit synthesis from specification patterns. *CoRR*, abs/2107.11864. arXiv: 2107.11864. https://arxiv.org/abs/2107.11864.

[18] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. 2021. Isarstep: a benchmark for high-level mathematical reasoning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Pzj6fzU6wkj.

[19] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Trans. Neural Networks*, 20, 1, 61–80. DOI: 10.1109/TNN.2008.2005605. https://doi.org/10.1109/TNN.2008.2005605.

[20] Wenliang Liu, Noushin Mehdipour, and Calin Belta. 2020. Recurrent neural network controllers for signal temporal logic specifications subject to safety constraints. *CoRR*, abs/2009.11468. arXiv: 2009.11468. https://arxiv.org/abs/2009.11468.

[21] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. 2018. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, 10338–10349. https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html.

[22] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding high-performance SAT solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings* (Lecture Notes in Computer Science). Mikolás Janota and Inês Lynce, editors. Volume 11628. Springer, 336–353. DOI: 10.1007/978-3-030-24258-9_24. https://doi.org/10.1007/978-3-030-24258-9%5C_24.

[23] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. 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, 2967–2974. https://aaai.org/ojs/index.php/AAAI/article/view/5689.

[24] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=B1lnbRNtwr.

[25] Guillaume Lample and François Charton. 2019. Deep learning for symbolic mathematics. *CoRR*, abs/1912.01412. arXiv: 1912.01412. http://arxiv.org/abs/1912.01412.

[26] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747. arXiv: 1609.04747. http://arxiv.org/abs/1609.04747.

[27] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Yoshua Bengio and Yann LeCun, editors. http://arxiv.org/abs/1409.0473.

[28] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Yoshua Bengio and Yann LeCun, editors. http://arxiv.org/abs/1301.3781.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR*, abs/1512.03385. arXiv: 1512.03385. http://arxiv.org/abs/1512.03385.

[30] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2, 4, 255–299. DOI: 10.1007/BF01995674. https://doi.org/10.1007/BF01995674.

[31] Joël Ouaknine and James Worrell. 2006. On metric temporal logic and faulty turing machines. In *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings* (Lecture Notes in Computer Science). Luca Aceto and Anna Ingólfsdóttir, editors. Volume 3921. Springer, 217–230. DOI: `10.1007/11690634\_15`. `https://doi.org/10.1007/11690634%5C_15`.

[32] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A framework for LTL and \omega -automata manipulation. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings* (Lecture Notes in Computer Science). Cyrille Artho, Axel Legay, and Doron Peled, editors. Volume 9938, 122–129. DOI: `10.1007/978-3-319-46520-3_8`. `https://doi.org/10.1007/978-3-319-46520-3_8`.

[33] [n. d.] Randltl. Retrieved 09/27/2021 from `https://spot.lrde.epita.fr/randltl.html`.

[34] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1, 1-2, 134–152. DOI: `10.1007/s100090050010`. `https://doi.org/10.1007/s100090050010`.

[35] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2016. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, 53, 2, 171–206. DOI: `10.1007/s00236-015-0229-y`. `https://doi.org/10.1007/s00236-015-0229-y`.

[36] Marcell Vazquez-Chanlatte. 2019. Mvcisback/py-metric-temporal-logic: v0.1.1. (January 2019). DOI: `10.5281/zenodo.2548862`. `https://doi.org/10.5281/zenodo.2548862`.

[37] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1998. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*. Mark A. Ardis and Joanne M. Atlee, editors. ACM, 7–15. DOI: `10.1145/298595.298598`. `https://doi.org/10.1145/298595.298598`.

[38] Kousha Etessami and Gerard J. Holzmann. 2000. Optimizing büchi automata. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings* (Lecture Notes in Computer Science). Catuscia Palamidessi, editor. Volume 1877. Springer, 153–167. DOI: `10.1007/3-540-44618-4\_13`. `https://doi.org/10.1007/3-540-44618-4_13`.

[39] Radek Pelánek. 2007. BEEM: benchmarks for explicit model checkers. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings* (Lecture Notes in Computer Science). Dragan Bosnacki and Stefan Edelkamp, editors. Volume 4595. Springer, 263–267. DOI: `10.1007/978-3-540-73370-6\_17`. `https://doi.org/10.1007/978-3-540-73370-6_17`.

[40] Fabio Somenzi and Roderick Bloem. 2000. Efficient büchi automata from LTL formulae. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings* (Lecture Notes in Computer Science). E. Allen Emerson and A. Prasad Sistla, editors. Volume 1855. Springer, 248–263. DOI: `10.1007/10722167\_21`. `https://doi.org/10.1007/10722167_21`.

[41] Jan Holeček, Tomas Kratochvila, Vojtech Rehák, David Safránek, and Pavel Simecek. 2004. Verification results in liberouter project. In.

[42] Christopher Hahn, Frederik Schmitt, Jens U Kreber, Markus N Rabe, and Bernd Finkbeiner. Deepltl 2020. `https://github.com/reactive-systems/deepltl`.

[43] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: bridging the gap between human and machine translation. *CoRR*, abs/1609.08144. arXiv: `1609.08144`. `http://arxiv.org/abs/1609.08144`.

[44] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Yoshua Bengio and Yann LeCun, editors. `http://arxiv.org/abs/1412.6980`.

[45] Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, 12058–12068. `https://proceedings.neurips.cc/paper/2019/hash/6e0917469214d8fbd8c517dcdc6b8dcf-Abstract.html`.