Mixed-Integer Non-linear Formulation for Optimisation over Trained Transformer Models

Master Thesis

Siân Hallsworth

TU Delft
Delft University of Technology

Mixed-Integer Non-linear Formulation for Optimisation over Trained Transformer Models

Master Thesis

by

Siân Hallsworth

to obtain the degree of Master of Science in Computer & Embedded Systems Engineering
at the Delft University of Technology,
to be defended publicly on 17th December 2024 at 9:00 AM.

Graduation Committee:

Dr. N. Yorke-Smith,	TU Delft, Chair
Dr. J. T. van Essen,	TU Delft, Core member
Dr. A. M. Schweidtmann,	TU Delft, Core member
Dr. T. Karia,	TU Delft, Daily co-supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Acknowledgements

I would like to express my heartfelt gratitude to everyone who supported me throughout this thesis. I sincerely thank my supervisors for their guidance and expertise, which helped me grow my knowledge and skills in this field of research. I am deeply thankful to Dr. Tanuj Karia whose daily supervision and insights greatly enriched this project. I look forward to applying everything I have learnt in my future endeavours. Lastly, I would like to thank my friends and family. Your support and encouragement have been truly appreciated.

Siân Hallsworth
Delft, December 2024

Abstract

In the past few years, rapid strides have been made in the modelling of complex systems due to the advent of machine learning (ML) technologies. In particular, the transformer neural network (TNN) has gained a lot of attention due to its powerful “sequence-to-sequence” modelling for several tasks in science and engineering. Predictive ML models are trained to learn the relationship between a set of input and output data so that new input data can be mapped to their expected output values. This gives us the ability to understand the key factors impacting a system and to make predictions about its future behaviours. This research investigates how to use TNNs for decision making by optimising the output of a TNN. To this end, a non-convex mixed-integer non-liner programming (MINLP) formulation for a trained TNN is proposed. The proposed formulation facilitates solving problems with TNNs embedded to global optimality. The effectiveness of the formulation is tested on three case studies consisting of an optimal trajectory problem, a verification problem, and a reactor optimization case study. Results show that optimisation over small TNNs can be achieved in under 3 minutes. However the tractability of the formulation quickly vanishes for larger models, highlighting the need for further research to refine the proposed formulation.

Nomenclature

Abbreviations

Acronym	Definition
FBBT	Feasibility-based bound tightening
FFN	Feed-forward neural network
GNN	Graph neural network
GRU	Gated recurrent units
LSTM	Long-short-term memory
MHA	Multi-head attention
MILP	Mixed integer linear program
MINLP	Mixed integer non-linear programming
ML	Machine learning
NLP	Natural language processing
NN	Neural network
ReLU	Rectified linear unit
RNN	Recurrent neural network
SiLU	Sigmoid linear unit
TNN	Transformer neural network

Symbols

Sets

Symbol	Definition
$\mathcal{D}$	Set of indices for the embedding dimensions
$\mathcal{F}$	Set of indices for the dimension of FFN
$\mathcal{H}$	Set of indices for the number of attention heads in MHA
$\mathcal{I}$	Set of indices for the sequence length dimension of the input
$\mathcal{I}^{\mathsf{dec}}$	Set of indices for the sequence length dimension of the decoder input
$\mathcal{J}$	Set of indices for the feature dimension of the input
$\mathcal{K}$	Set of indices for the dimension of each attention head in MHA

Parameters

Symbol	Definition
$\mathbf{b}_d^{e}$	Learned bias parameter for embedding indexed over set $\mathcal{D}$
$\mathbf{b}^{\mathfrak{h}}$	Scalar bias for a single hidden node in FFN
$\mathbf{b}_f^{\mathfrak{H}}$	Learned bias parameter for the first FFN layer indexed over set $\mathcal{F}$
$\mathbf{b}_{h,k}^{K_\alpha}$	Learned bias parameter for key variable indexed over sets $\mathcal{H}, \mathcal{K}$
$\mathbf{b}_d^{o_\alpha}$	Learned bias parameter for MHA output indexed over set $\mathcal{D}$
$\mathbf{b}_{h,k}^{Q_\alpha}$	Learned bias parameter for query variable indexed over sets $\mathcal{H}, \mathcal{K}$
$\mathbf{b}_{h,k}^{V_\alpha}$	Learned bias parameter for value variable indexed over sets $\mathcal{H}, \mathcal{K}$
$\mathfrak{d}$	Feature dimension of the embedded input

Symbol	Definition
$\mathfrak{d}^k$	Feature dimension of MHA heads
I	Sequence length
M^-	Lower bound parameter of the first FFN layer output
M^+	Upper bound parameter of the first FFN layer output
$M^{C'}$	Big-M value for max compatibility calculation
$\mathbf{W}^{e}_{j,d}$	Learned weight parameter for embedding indexed over sets $\mathcal{J}, \mathcal{D}$
$\mathbf{W}^{\mathfrak{h}}$	Weight vector for a single hidden node in FFN
$\mathbf{W}^{\mathfrak{H}}_{d,f}$	Learned weight parameter for the hidden layer of FFN indexed over $\mathcal{D}, \mathcal{F}$
$\mathbf{W}^{K_\alpha}_{d,h,k}$	Learned weight parameter for key variable indexed over sets $\mathcal{D}, \mathcal{H}, \mathcal{K}$
$\mathbf{W}^{\mathfrak{o}}_{f,d}$	Learned weight parameter for the output layer of FFN indexed over $\mathcal{F}, \mathcal{D}$
$\mathbf{W}^{o_\alpha}_{d,h,k}$	Learned weight parameter for MHA output indexed over sets $\mathcal{D}, \mathcal{H}, \mathcal{K}$
$\mathbf{W}^{Q_\alpha}_{d,h,k}$	Learned weight parameter for query variable indexed over sets $\mathcal{D}, \mathcal{H}, \mathcal{K}$
$\mathbf{W}^{V_\alpha}_{d,h,k}$	Learned weight parameter for value variable indexed over sets $\mathcal{D}, \mathcal{H}, \mathcal{K}$
β^{ℓ}_{d}	Layer normalization offset parameter indexed over $\mathcal{D}$
γ^{ℓ}_{d}	Layer normalization scaling parameter indexed over $\mathcal{D}$
ϵ^{ℓ}	A parameter added to the variance in layer normalization to prevent division by 0
$\boldsymbol{\pi}_{i,d}$	Positional embedding parameter indexed over sets $\mathcal{I}, \mathcal{J}$
Ψ	Number of elements in the power series approximation of the exponential function

Variables

Symbol	Definition
$\boldsymbol{A}^{\alpha}_{h,i,k}$	Attention score variable in multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{K}$
$\boldsymbol{C'}^{\max}_{\alpha_{h,i}}$	Maximum compatibility variable indexed over sets $\mathcal{H}, \mathcal{I}$
$\boldsymbol{C}^{\alpha}_{h,i,i}$	Compatibility variable of multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{I}$
$\boldsymbol{E}^{\alpha}_{h,i,i}$	$\mathrm{Exp}(\boldsymbol{C}^{\alpha}_{h,i,i})$ in multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{I}$
$\boldsymbol{K}^{\alpha}_{h,i,k}$	Key variable of multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{K}$
$\boldsymbol{Q}^{\alpha}_{h,i,k}$	Query variable of multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{K}$
$\boldsymbol{S}^{\alpha}_{h,i}$	Sum of $\mathrm{Exp}(\boldsymbol{C}^{\alpha}_{h,i,i})$ in multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}$
$\boldsymbol{V}^{\alpha}_{h,i,k}$	Value variable of multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{K}$
$\boldsymbol{x}_{i,j}$	Input variable indexed over sets $\mathcal{I}, \mathcal{J}$
$\boldsymbol{x}^{\mathfrak{H}}_{i,d}$	Output variable of FFN layer indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{+}_{i,d}$	Output variable of residual layer indexed over $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\alpha}_{i,d}$	Output variable of multi-head attention layer indexed over $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{e}_{i,d}$	Embedded input variable indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\ell}_{i,d}$	Output variable of layer normalization layer indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\pi}_{i,d}$	Positionally encoded input variable indexed over $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\mathsf{dec}}_{i,d}$	Input to decoder indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\mathsf{enc}}_{i,d}$	Input to encoder indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{x}^{\mathsf{pool}}_{d}$	Output variable of pooling layer indexed over sets $\mathcal{D}$
$\boldsymbol{z}^{\mathfrak{H}}_{i,f}$	Binary variable used for ReLU activation formulation of FFN indexed over sets $\mathcal{I}, \mathcal{F}$
$\boldsymbol{z}^{C'}_{h,i,i'}$	Binary variable for max compatibility calculation indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{I}$

Symbol	Definition
$\alpha_{n \to m}$	A scalar value representing the attention or importance of element m in the sequence to element n
$\boldsymbol{\delta}_i^{\ell}$	Variance variable of layer normalization input indexed over sets $\mathcal{I}$
$\boldsymbol{\Theta}_{i,d}^{\ell}$	Standard deviation of layer normalization input indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{\xi}_i^{\ell}$	Sum of layer normalization input variable indexed over sets $\mathcal{I}$
$\boldsymbol{\sigma}_i^{\ell}$	Standard deviation variable in layer normalization input indexed over sets $\mathcal{I}$
$\boldsymbol{\upsilon}_{i,d}^{\ell}$	Mean centered value of layer normalization input variable indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{\Upsilon}_{i,d}^{\ell}$	Squared mean centered value of layer normalization input indexed over sets $\mathcal{I}, \mathcal{D}$
$\boldsymbol{\omega}_{h,i,i}^{\alpha}$	Attention weight variable in multi-head attention layer indexed over sets $\mathcal{H}, \mathcal{I}, \mathcal{I}$
$\boldsymbol{\Omega}_{i,f}^{\mathfrak{h}}$	Output variable of ReLU activation layer of FFN indexed over sets $\mathcal{I}, \mathcal{F}$

Contents

Acknowledgements **i**

Abstract **ii**

Nomenclature **iii**

1 Introduction **1**

2 Background **3**
2.1 Neural Networks . . . 3
2.1.1 Feed-Forward Neural Networks . . . 3
2.1.2 Transformer Neural Networks . . . 4
2.1.3 TNN Illustrative Example . . . 11
2.2 Neural Networks for Optimisation . . . 16
2.2.1 Formulations . . . 16
2.2.2 Implementation . . . 19

3 Methods **20**
3.1 Transformer Formulation . . . 20
3.1.1 Input Embedding . . . 20
3.1.2 Encoder . . . 21
3.1.3 Decoder . . . 23
3.1.4 Output Prediction Head . . . 24
3.2 Bounds Tightening . . . 24
3.2.1 Linear Layer . . . 24
3.2.2 Residual . . . 25
3.2.3 Multi-head Attention . . . 25
3.3 Strengthening Formulation . . . 25
3.3.1 Layer Normalisation . . . 26
3.3.2 Multi-head Attention . . . 26
3.4 Overview of Bounds & Cuts . . . 30
3.5 Implementation . . . 31

4 Case Studies **33**
4.1 Optimal Trajectory Problem . . . 33
4.1.1 Problem Definition . . . 33
4.1.2 Experimental Set Up . . . 35
4.1.3 Numerical Results . . . 36
4.2 Verification Problem . . . 40
4.2.1 Problem Definition . . . 40
4.2.2 Experimental Set Up . . . 40
4.2.3 Numerical Results . . . 41
4.3 Reactor Case Study . . . 45
4.3.1 Problem Definition . . . 45
4.3.2 Experimental Set Up . . . 45
4.3.3 Numerical Results . . . 47
4.4 Key Results . . . 47

5 Conclusions **49**
5.1 Research Outcomes . . . 49
5.2 Limitations . . . 50
5.3 Outlook and Future Work . . . 50

References **52**

A Academic paper **56**

1 Introduction

Optimisation problems are all around us: we encounter them every time we try to maximize or minimize an outcome. This could be minimizing travel time by finding the shortest path between a set of locations or planning a diet so that the consumption of certain minerals is maximized. Optimisation problems become more difficult to solve as the size of the problem increases. Over the years, many have investigated how to efficiently solve these problems to optimal (or near optimal) solutions. This work has produced two overarching categories of optimisation techniques: (meta-)heuristic and exact [43]. Heuristic based methods, generally provide quick solutions to a problem with the caveat that there is no proof of optimality. On the other hand, exact methods often require more computational effort but can prove a solution's global optimality [43].

Exact optimisation approaches formulate optimisation problems using a mathematical model known as a mathematical program. Here, the goal is described as a function of variables in the problem (objective function), and the constraints on the problem are described by a set of constraints, often equality and in-equality equations [47].

Mathematical programs require that every part of the problem being modelled has a clear mathematical representation; which is not always the case. This can limit mathematical programming to applications where the underlying mechanisms of the system are explicitly understood.

Looking to the field of machine learning (ML), this limitation has already been surpassed, whereby we can model complex systems using data-driven approaches. The high levels of accuracy achieved by these models has lead to them becoming one of the most common modelling techniques. A lot of attention has been garnered by neural networks (NNs) for their ability to create accurate yet simplified models of complex or currently indescribable systems [1]. Consequently, NNs have been widely applied in fields of science and engineering [52][12].

These advancements in our modelling capabilities open the door to optimizing systems that could not previously be modelled as part of a mathematical program. This in turn allows for informed decision making for such systems. Current research has investigated how to create efficient mathematical formulations of NNs, that can be solved in a reasonable amount of time. Formulations have been proposed for the feed-forward NN, recurrent NN, convolutional NN and graph NN. These NN formulations have been applied to solve a variety of optimisation problems. For example, the optimisation of oil production systems with multi-phase flows [16], molecular design [39] and NN verification [14][56].

In recent years, a new NN model known as the transformer neural network (TNN) was developed by Vaswani et al. [57]. Although the model was originally proposed for natural language processing, it has been shown to have outstanding predictive abilities for other tasks such as time series modelling and computer vision [35]. Their success in other predictive tasks has been attributed to the TNN's ability to capture complex dependencies in the input data [59]. Moreover, the model's parallelized architecture, has resulted in faster training times than other similar NNs like long-short-term memory (LSTM) and recurrent NNs (RNNs) [26]. Due to the TNN's high performance and wide applicability,

this NN has quickly gained a lot of research attention. Lin et al. [35] provide an extensive survey of various transformer architectures and their applications.

Given the wide variety of applications of TNNs and their popularity as a modelling technique, this work aims to create an efficient formulation of a trained TNN for mathematical programming. With this goal in mind, the following research questions were devised:

1. How can a trained TNN be formulated as part of a mathematical program?
2. What is the solving performance of the formulated trained TNN?
3. How can solving be improved for efficient global optimisation?

The remainder of this thesis is structured as follows. First, background information about mathematical programming, TNNs and related works is presented in Chapter 2. Next, a MINLP formulation for a trained TNN is proposed in Chapter 3. The results of computational experiments are discussed in Chapter 4. Finally, the main takeaways of this thesis and recommendations for future research are described in Chapter 5.

2

Background

This chapter provides the necessary theoretical background for different types of neural networks (NN) and serves as a basis for the formulated transformer neural network (TNN) used in this thesis.

2.1. Neural Networks

Neural networks take inspiration from the biological neural systems. These models learn the relationship between given input and output data during a process called training. During training the model is essentially figuring out how to transform and combine the input values to arrive at the desired output value. These models rely on a large number of connected neurons and a large collection of data to discern complicated patterns and relationships [53]. Typically, their ability to capture difficult relationships is improved by using large data for training and by increasing the size of the NN [53].

These data-driven models have become a popular topic in research due to their ability to represent high-dimensional non-linear systems with precision. Within engineering fields, there has been a rapid uptake of NNs as a way to simplify the modelling of complex physical phenomena, and as a means to gain insights into behaviours that cannot be modelled by traditional approaches [37]. NNs provide new avenues for approximating the behaviour of such systems, which in turn has stimulated research into leveraging these less computationally intensive models. This research includes predicting future behaviours of a system as well as investigating how to configure systems to achieve desired outcomes.

2.1.1. Feed-Forward Neural Networks

The feed-forward neural network (FFN) has the simplest NN architecture and is the building block from which other NNs are created. FFNs are made up of units called neurons (or nodes), which receive, process and transmit data. These neurons are organised by layers, such that the neurons of a given layer transmit their outputs to a subsequent layer. The layers can be categorised into three groups: the input layer, the hidden layers, and the output layer. The input layer feeds the input data into the FFN. The hidden layers model the relationship between the input and output data. Lastly, the output layer gives the prediction of the FFN depending on the task at hand. An example of a FNN with 1 hidden layer is shown in Figure 2.1.

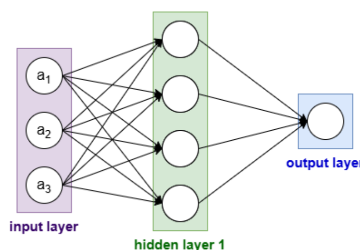


Figure 2.1: Architecture of feed-forward Neural Network

The implementation of the FFN relies on two main components: linear transforms, which adjust the value input to each neuron, and activation functions in the hidden layers, which allow for non-linear relationships to be modelled. In Figure 2.1, every arrow can be seen as a scaling operation to adjust the input to the next node, while each node can serve as an indication of a bias operation. These linear transforms are implemented using weight and bias parameters learnt during training.

Aside from the weight and bias operations, nodes in the hidden layers apply an activation function as shown in Figure 2.2. There are many activation functions that can be chosen when designing a FFN. For example, one of the most popular is the rectified linear unit (ReLU) which caps negative values to zero while allowing positive values to remain unchanged. Another popular activation function is the sigmoid linear unit (SiLU) or swish function which provides a smooth approximation of the ReLU function.

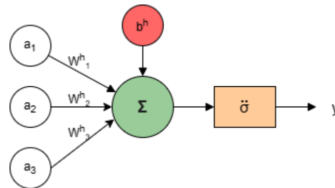


Figure 2.2: Computation at a hidden node for an arbitrary activation function $\ddot{\sigma}$

For an arbitrary activation function, $\ddot{\sigma}$, the computation at each hidden node can be written as,

$$\mathbf{y} = \ddot{\sigma}(\mathbf{a}\mathbf{w}^{\mathfrak{h}} + b^{\mathfrak{h}}) \tag{2.1}$$

where $\mathbf{a}$ is a vector of inputs, $[a_1, a_2, a_3]$, $\mathbf{w}^{\mathfrak{h}}$ is a vector of associated weights $[w_1^{\mathfrak{h}}, w_2^{\mathfrak{h}}, w_3^{\mathfrak{h}}]$, and $b^{\mathfrak{h}}$ is a scalar bias term. At each hidden node, the dot product of the input vector and the weight vector is computed, and the bias is added to produce the node's input to the activation function. This process transforms the input vector $\mathbf{a}$ into an output $\mathbf{y}$ based on the activation function $\ddot{\sigma}$.

To describe an entire layer of hidden nodes, this operation is extended by combining the weight vectors of each node into a weight matrix, and the biases into a bias vector. This allows the computation for the entire layer to be performed efficiently as a single matrix operation as shown in Equation 2.2, where $\mathbf{W}^{\mathfrak{H}}$ and $\mathbf{b}^{\mathfrak{H}}$ are the weight matrix and bias vector for the hidden layer, similarly $\mathbf{W}^{\mathfrak{o}}$ and $\mathbf{b}^{\mathfrak{o}}$ are the weight and bias for the output layer.

$$\mathrm{FFN}(\mathbf{a}) = \mathbf{W}^{\mathfrak{o}}(\ddot{\sigma}(\mathbf{a}\mathbf{W}^{\mathfrak{H}} + \mathbf{b}^{\mathfrak{H}})) + \mathbf{b}^{\mathfrak{o}} \tag{2.2}$$

2.1.2. Transformer Neural Networks

Transformers are currently one of the most popular NNs. They were originally applied to Natural Language Processing (NLP) tasks such as translation, text summarising and question answering, bringing about one of the most well-known AI applications Chat-GPT [41]. However, they are now being used for a plethora of tasks. For example, recent works show its application for time series forecasting [3][17], molecular property prediction [44][49], and computer vision for both images and videos [6][4][51]. This expansion of the TNNs application has come about from viewing it as a model that converts an input data sequence to an output data sequence (sequence-to-sequence model). Thus, the TNN can be applied to any task that uses sequential data.

The first transformer model, often referred to as the "Vanilla Transformer", was proposed in 2017 by Vaswani et al. [57]. It introduced the novel construction of multi-head attention components which extract information about how each element in a sequence interacts with the other elements. The transformer combines information about element similarity, proximity, and dependencies to create a representation of each element that is informed by its context. This contextual understanding can for example allow the transformer to distinguish between when the word "Apple" refers to the fruit and when it refers to the company.

The multi-head attention unit set the transformer apart from other sequence-to-sequence models due to its ability to capture long-range dependencies and highly parallelized architecture; which leads to

faster training times [26]. Moreover, all input tokens can be simultaneously fed into transformer models instead of processing one token at a time like recurrent neural networks (RNNs), long-short-term memory (LSTM) and gated recurrent Units (GRUs) [3].

Another notable advantage of transformer models is their minimal reliance on assumptions about the structure of input data [35]. In contrast, convolutional neural networks (CNNs) are inherently biased toward translation invariance and assume that local patterns are spatially repeated. Similarly, RNNs are constrained by their reliance on locality, as they assume that the current state depends primarily on recent past states. These inductive biases make CNNs highly efficient for image processing and RNNs well-suited for time-series tasks. However, they also limit these architectures to specific applications, as they struggle to represent and capture global relationships in data. Transformers, on the other hand, offer a more universal and flexible framework, allowing them to become a powerful tool in a wide range of applications. The only caveat is that the lack of structural information makes the transformers prone to overfitting on small datasets [35].

Implementation

This section describes the Vanilla Transformer model in the context of a translation task. Although numerous transformer architectures exist, they all share the same foundational components. At a high level, a transformer neural network comprises input embedding layers, an encoder block, a decoder block, and an output prediction head, as illustrated on the left side of Figure 2.3. Typically, the encoder is responsible for building a contextual understanding of the input data, while the decoder handles the predictive task.

Examining these two components in greater detail reveals several key modules: multi-head attention, addition and normalization, and feed-forward neural network layers (depicted on the right side of Figure 2.3). Notably, Figure 2.3 highlights the structural similarity between the encoder and decoder. The primary distinction is the decoder's inclusion of an additional masked multi-head attention layer, as well as corresponding addition and normalization layers. These additional layers are essential for enabling the transformer to autoregressively generate a sequence of output predictions.

The functionality and roles of these modules within the transformer architecture are outlined below.

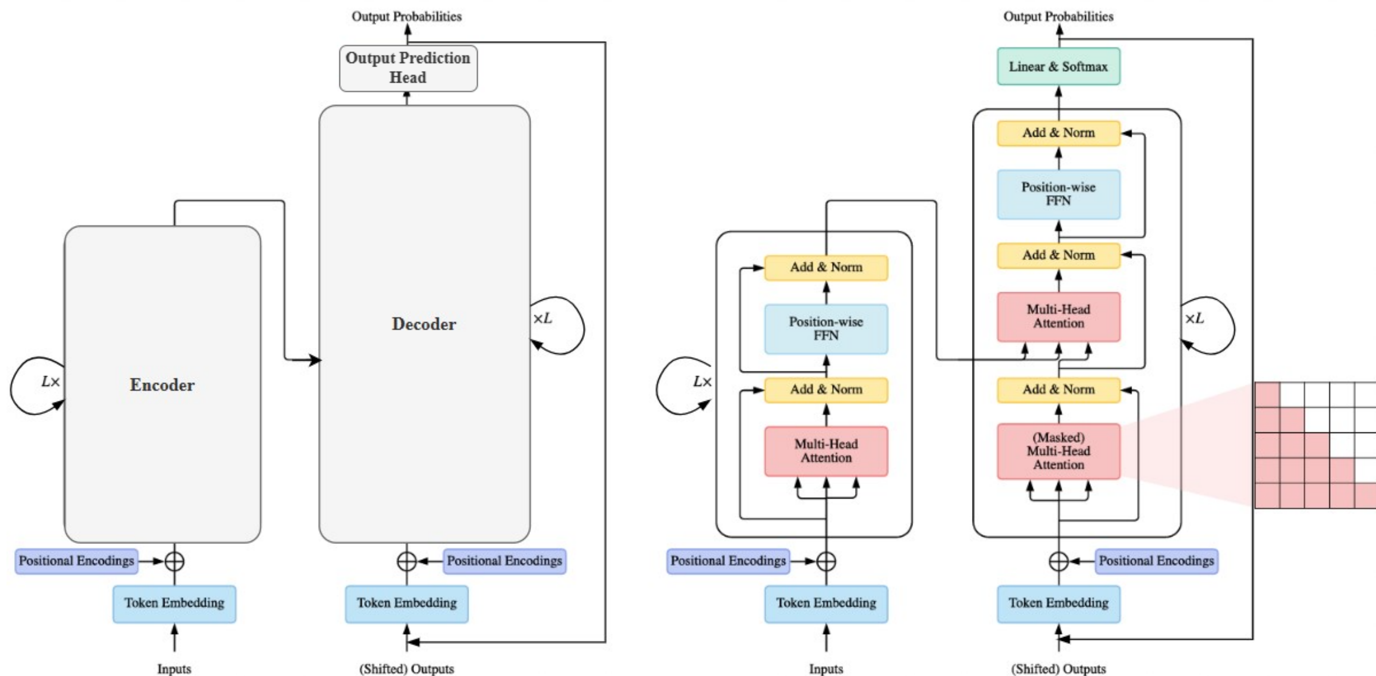


Figure 2.3: Vaswani et al. [57] transformer showing an overview of main components (left) and sub-components (right) with L encoder/decoder layers. Image modified from [35]

Token Embedding

In the first step of the training task, the input sequence is tokenized and embedded into a numeric vector. The tokenization process breaks down the input sequence into smaller chunks called tokens each with an associated value [58]. There are many `Python` libraries available to segment text at varying levels of granularity. The most simple of them is word-level tokenization which divides a sentence such that there is one partition (or token) for each word. In this case, each token (or word) is assigned a unique token ID. This segmentation of the input text is important for creating structured data with clearly defined numeric elements. A tokenized sequence ($\boldsymbol{x}$) is thus given by Equation 2.3 where N is the number of tokens.

$$\boldsymbol{x} = [x_0, ..., x_N]^T \tag{2.3}$$

After the sequence is tokenized, each token transformed into a numeric vector with $\mathfrak{d}$ dimensions. The role of these dimensions is to represent the tokens based on their meaning in relation to the other tokens [3]. Each embedding dimension can be interpreted as capturing a specific attribute or feature of the data. This allows the model to understand the similarity of a token in relation to the other tokens in the vocabulary. Taking the example from [34], when the embeddings of the words woman, man, king, and queen are plotted in a 2-dimensional plane, we can visualize their similarities in terms of royal status and femininity. For instance, we see queen-king are more similar (closer) than king-woman. The embeddings can capture similarities to the degree that if a calculation is performed to find the value of king - man + woman, the resulting value is approximately equal to the representation of the word queen.

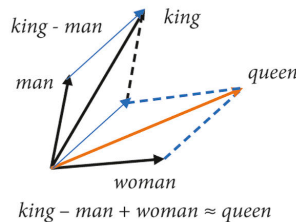


Figure 2.4: A word embedding example retrieved from [34]

Mathematically, the embedding step performs a linear transformation. It multiplies the input and output tokens by a learnt weight matrix of size $(1 \times I)$ [57]. The result is then scaled by a factor $\sqrt{\mathfrak{d}}$. The choice of this scaling factor is not explicitly motivated in [57] however, it is expected that this value is chosen to ensure that the magnitude of the embedded input and the positional encoding are of comparable size. In this architecture, the positional encoding is implemented using sine and cosine functions which are in the range [-1,1]. The value of the learnt weight ($\boldsymbol{W^e}$) is initialised as $\mathcal{N}(0, \frac{1}{\sqrt{\mathfrak{d}}})$. So, without scaling, the positional encoding values would outweigh the embedding values and as a result, the model would lack descriptive information about the input. Therefore, this scaling factor is only necessary due to how the Vanilla Transformer has been configured.

$$\boldsymbol{x}^e = \sqrt{\mathfrak{d}}\,(\boldsymbol{x} \cdot \boldsymbol{W^e}) \tag{2.4}$$

For numeric data, there is no tokenisation but the embedding layer is often kept, to transform the input data using learnt weights and biases. Much like the NLP case, this transformation helps the model to represent the data in a way that can better inform predictions.

Positional Encoding

As transformer models process all elements of the input sequence concurrently, a Positional Encoding (PE) component is needed to feed information about the order of elements in the sequence. Vaswani et al. [57] use a sinusoidal encoding method that gives a unique vector identifier for each position in the sequence. Positional encodings, $\boldsymbol{\pi}$, for each sequence position, $P \in \{0, \ldots, N\}$, and embedding dimension, $i \in \{0, \ldots, \mathfrak{d} - 1\}$, are given by:

$$\boldsymbol{\pi}_{P,\,2i} = \sin(P/10000^{2i/\mathfrak{d}-1}) \tag{2.5a}$$

$$\boldsymbol{\pi}_{P,\,2i+1} = \cos(P/10000^{2i/\mathfrak{d}-1}) \tag{2.5b}$$

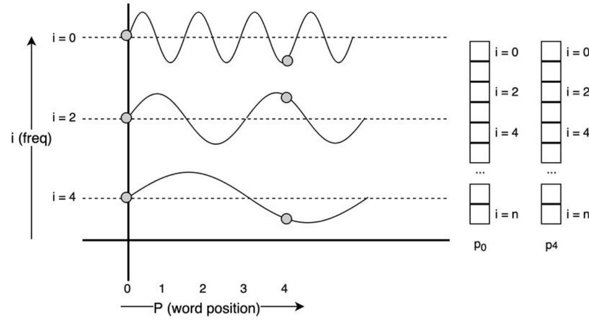


Figure 2.5: Illustration of positional encoding for embedding dimensions and sequence position. Image retrieved from Haque, Ghani, and Saeed [23].

Figure 2.5 illustrates how these positional encodings are created. For each embedding dimension, i, a sinusoid is constructed with a different angular frequency ($1/10000^{2i/\mathfrak{d}-1}$). This allows a wide range of unique values to be generated even though cyclic functions are applied. The even embedding dimensions are encoded using the sine functions while cosine functions are used for the odd. The trigonometric properties of these functions implicitly store information about the relative positions of elements. Consider an element at position $P+q$. The positional encoding of this value, $\boldsymbol{\pi}_{P+q}$, can be written as,

$$\boldsymbol{\pi}_{P+q,\,2i} = \sin(P')\cos(q') + \cos(P')\sin(q') \tag{2.6}$$

$$\boldsymbol{\pi}_{P+q,\,2i+1} = \cos(P')\cos(q') - \sin(P')\sin(q') \tag{2.7}$$

where $P' = P/10000^{2i/\mathfrak{d}-1}$ and $q' = q/10000^{2i/\mathfrak{d}-1}$. Since q is a fixed offset, the positional encoding of $P+q$ is a linear transform of $\boldsymbol{\pi}_P$. As a result, every element's encoding is a phase-shifted version of the previous position. This gives information about the ordering of the sequence to subsequent layers. The main benefit of this encoding approach is that it reduces training time as less time is needed to decipher the relative positions of elements. Once calculated, these positional encodings are added to the embedded input (Equation 2.8).

$$\boldsymbol{x}^{\pi} = \boldsymbol{x}^{e} + \boldsymbol{\pi} \tag{2.8}$$

An example of the patterns created by sinusoidal encodings is depicted in Figure 2.6. At sequence position 0, an oscillating pattern between dark green and yellow can be seen. This pattern occurs because the encoding of the embedding dimensions will lead to $\sin(0) = 0$ for even dimensions and $\cos(0) = 1$ for odd dimensions. As the position increases (columns), more detailed patterns are developed, due to the different angular frequencies used for each embedding dimension (rows). The colours of the rows gradually change at slower rates. This indicates how the wavelength increases with higher embedding dimensions as shown earlier in Figure 2.5. As a result, the encodings show similar patterns for nearby positions (columns) and similarity decreases with distance in the sequence.

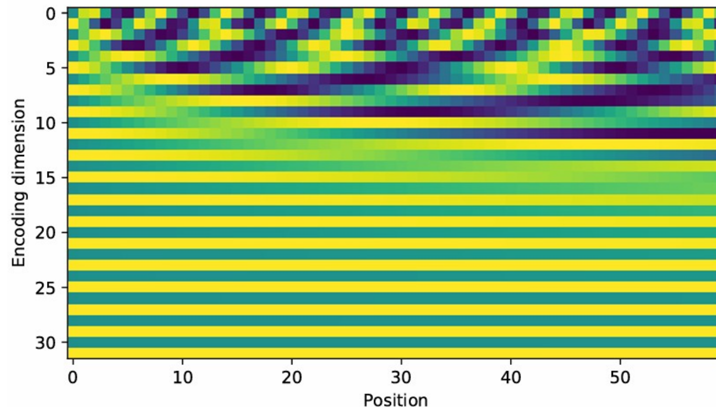


Figure 2.6: An example of token position in sequence vs embedding dimension with positional encoding values (π) shown via the heat map. Image retrieved from [28].

Other positional encodings are also seen in the literature. For example, computer vision applications tend to use learnable position encoding vectors [6][51]. Alexey Dosovitskiy et al. [4] and Adi Haviv et al. [2] also consider TNNs without positional information. Experiments by Alexey Dosovitskiy et al. (2021) show that there is only slight performance degradation for their computer vision example. While experiments by Adi Haviv et al. (2022) on an NLP use case report that excluding the positional encoding layer can lead to poor performance depending on the TNN architecture. This shows that the need for positional encodings is application and architecture dependent.

Encoder

The encoder's primary role is to build a contextual understanding between the tokens in the input sequence. These contextualised representations are typically used later by the decoder to make predictions. The encoder consists of multiple layers of multi-head attention, add and normalisation, and FFN components. The attention module forms the core of the encoder, learning dependencies and relationships which are further refined in the subsequent normalisation and FFN layers. Attention is computed using three projections of the input sequences called the Query, Key and Value. These matrices determine the pair-wise importance of each element in the sequence to all others, and inform the model of how much attention a given element should pay to the other elements. This allows the model to focus on the most significant relationships for every element in the sequence, which is useful for modelling complex data.

Each encoder layer refines the output of the previous layer, progressively developing more nuanced representations of the input sequence. In the Vanilla Transformer, the output of the final encoder layer is used as input to the decoder. However, some popular transformer architectures, like BERT and RoBERTa, use the encoder's output to directly make predictions; bypassing the need for a decoder [35]. These architectures, known as encoder-only models, are generally less computationally intensive and require fewer resources to train due to their smaller size. However, encoder-only models tend to perform worse than their encoder-decoder counterparts.

Decoder

The role of the decoder is to predict the output sequence based on a given input sequence. The decoder does this by predicting a single output token at a time, and then appending this output token to the output sequence. The new output sequence is fed back into the decoder each time to predict the next token. The masked multi-head attention block facilitates this by only permitting each token in the output sequence to attend to positions up to the current position. The subsequent attention step (cross-attention) between itself and the encoder enables the decoder to focus on relevant parts of the input sequence. In this case, the multi-head attention unit takes input values from both the encoder and decoder. More precisely, the Keys and Values are from the final encoder layer, and the Query values are from the decoder.

However, in some cases, decoder-only architectures are preferred. While decoder-only models capture less contextual information than encoder-decoder models, they achieve sufficient performance while reducing the size of the model. Decoder only are typically used for sequence generation [35] (e.g. GPT-3, ChatGPT, GoogleLaMDA).

Multi-Head Attention

The concept of multi-head attention (MHA) is an expansion of self-attention [57]. Self-attention refers to the process of determining relationships between each pair of elements in a given sequence. This allows for the machine learning models to represent elements with regard to their value as well as their interactions with other elements, thereby extracting the dependencies for a particular context. Multi-head attention essentially repeats the self-attention process in order to capture information from different representations of the relationships between elements in a sequence [57]. Each attention head can be seen as projecting the embeddings to a lower dimension in order to focus on a particular attribute of the data. Taking the example of a transformer comparing the words king, queen, man and woman, one attention head could focus on relationships based on gender and the other based on royal status.

The multi-head attention unit first calculates the Query ($\boldsymbol{Q}$), Key ($\boldsymbol{K}$), and Value ($\boldsymbol{V}$) matrices. $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are computed using learnt weights for each head, $h \in \{1, \ldots, H\}$; where H is a hyperparameter

of the transformer model which is chosen as a factor of the embedding dimension, $\mathfrak{d}$ (Equation 2.9) [54]. The learnt weights have a size $(\mathfrak{d} \times \mathfrak{d}^k)$ where $\mathfrak{d}^k$ is the ratio of the embedding dimension $(\mathfrak{d})$ and the number of attention heads (H) and the biases have a size $(\mathfrak{d}^k)$.

$$\boldsymbol{Q}^h(\boldsymbol{x}^\pi) = \boldsymbol{x}^\pi \cdot \boldsymbol{W}_h^Q + \boldsymbol{b}_h^Q \tag{2.9a}$$

$$\boldsymbol{K}^h(\boldsymbol{x}^\pi) = \boldsymbol{x}^\pi \cdot \boldsymbol{W}_h^K + \boldsymbol{b}_h^K \tag{2.9b}$$

$$\boldsymbol{V}^h(\boldsymbol{x}^\pi) = \boldsymbol{x}^\pi \cdot \boldsymbol{W}_h^V + \boldsymbol{b}_V^h \tag{2.9c}$$

Next a single attention weight (α) is calculated for each pair of tokens $\boldsymbol{x}_n^\pi$ and $\boldsymbol{x}_m^\pi$ where $n, m \in \{1, \ldots, N\}$. This calculation is shown in Equation 2.10.

$$\alpha_{n \to m}^h = \mathsf{softmax}\left(\frac{\boldsymbol{Q}^h(\boldsymbol{x}_n^\pi) \cdot \boldsymbol{K}^{h^\top}(\boldsymbol{x}_m^\pi))}{\sqrt{\mathfrak{d}^k}}\right) \tag{2.10a}$$

$$\mathsf{softmax}(X_{i,j}) = \frac{e^{X_{i,j}}}{\sum_{p=1}^N e^{X_{i,p}}} \tag{2.10b}$$

Attention weights are used to describe the relative importance of a given token's relationship with another token. This is computed using the compatibility, or multiplication between the $\boldsymbol{Q}$ and $\boldsymbol{K}$. The compatibility compares two sequence elements, $\boldsymbol{x}_n^\pi$ and $\boldsymbol{x}_m^\pi$. While the softmax function determines the relative importance. The softmax function creates a probability distribution from the compatibility scores of each token $(\boldsymbol{x}_{\pi_n})$ with all other tokens (Figure 2.7). This essentially ranks the relationships between elements since the sum of all compatibility scores related to an element, $\boldsymbol{x}_{\pi_n}$, will be 1.

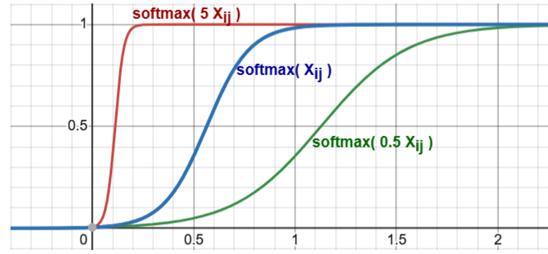


Figure 2.7: Softmax function for increasingly large input values

Note, that before the softmax function is applied, the compatibility is scaled by a factor $\frac{1}{\sqrt{\mathfrak{d}^k}}$. Vaswani et al. explain that without scaling, as the value of $\mathfrak{d}^k$ grows, so will the magnitude of the dot products. Consequently, the softmax function is pushed into regions with small gradients. As shown in Figure 2.7, as the input to the softmax function increases, the function saturates more quickly, which severely degrades learning. The scaling by $\frac{1}{\sqrt{\mathfrak{d}^k}}$ thus serves to prevent these unwanted effects.

The weighted sum of the attention weights and Value matrix are used to compute a new representation of the input tokens at each head, $\boldsymbol{x}_n^h$ with dimensions $(1 \times \mathfrak{d}^k)$ as shown in Equation 2.11.

$$\boldsymbol{x}_n^h = \sum_{m=1}^N \alpha_{n \to m}^h \cdot \boldsymbol{V}^h(\boldsymbol{x}_m^\pi)) \qquad \forall h \in \{1, \ldots, H\} \tag{2.11a}$$

$$\boldsymbol{x}_n = [\boldsymbol{x}_n^1, ..., \boldsymbol{x}_n^H] \tag{2.11b}$$

The vector representations from each attention head are then concatenated and combined to produce the multi-head attention output, $\boldsymbol{x}^\alpha$ with size $(N \times d)$ (Equation 2.12) [54].

$$\boldsymbol{x}_n^\alpha = \sum_{h=1}^H \boldsymbol{W}_h^\top \cdot \boldsymbol{x}_n \tag{2.12a}$$

$$\boldsymbol{x}^\alpha = [\boldsymbol{x}_1^\alpha, ..., \boldsymbol{x}_N^\alpha]^\top \tag{2.12b}$$

The multi-head attention block appears three times in the Vanilla TNN, once in the encoder and twice in the decoder. In the decoder the MHA layers are adapted slightly but work in almost the same way. The decoder produces the output sequence auto-regressively, taking the current output sequence as its input. Each token in the output sequence depends on the previous tokens in the output sequence (via masked multi-head attention) and the input sequence (via multi-head attention between the encoder and decoder). The masked multi-head attention layer alters the compatibility calculation to ensure that each predicted sequence element only depends on the previous sequence values. This is done by setting the compatibility to $-\inf$ for all subsequent positions (i.e. where $m > n$ in Equation 2.10), giving an attention weight of zero after the softmax function has been applied. As a result, future sequence values will have no influence on the calculation of previous values.

A different alteration appears in the second MHA block in the decoder. This block computes the cross attention between the encoder and decoder, such that values from both the encoder and decoder are used in the computations. Cross-attention is implemented by using the output of the previous decoder component to compute the Query matrix and the encoder output to compute the Key and Value.

Add & Norm

The Add & Norm component enhances training robustness and is especially important for large networks. It incorporates a residual layer which adds the input from the previous component ($\boldsymbol{x}^{\text{res}}$) to the current input of the Add & Norm component ($\boldsymbol{x}$) (Equation 2.13). A residual layer refers to this addition of a previous output back into the network at a later stage. It allows the model to preserve information from previous layers while learning new representations. He et al. [25] note that this is necessary because it is difficult for deep networks to maintain identity mappings when they consist of a stack of non-linear layers.

$$\boldsymbol{x}^{+} = \boldsymbol{x} + \boldsymbol{x}^{\text{res}} \tag{2.13}$$

Then layer normalization is performed to re-scale the data's mean and variance. This calculation is shown in Equation 2.14 where γ, β are learnable affine transforms and ϵ is a sufficiently small number used to avoid division by zero. The mean and standard deviation are calculated across the set of dimensions ($\mathcal{D} = \{0, \ldots, \mathfrak{d} - 1\}$) for each sequence position, n.

$$\boldsymbol{x}^{\ell}_{ni} = \mathsf{LayerNorm}\left(\boldsymbol{x}^{+}_{ni}, \gamma, \beta\right) \qquad \forall i \in \mathcal{D} \tag{2.14a}$$

$$= \gamma \frac{\boldsymbol{x}^{+}_{ni} - \mathsf{mean}(\boldsymbol{x}^{+}_{n})}{\mathsf{std}(\boldsymbol{x}^{+}_{n}) + \epsilon} + \beta \tag{2.14b}$$

$$\mathsf{mean}(\boldsymbol{x}^{+}_{n}) = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \boldsymbol{x}^{+}_{ni} \tag{2.14c}$$

$$\mathsf{std}(\boldsymbol{x}^{+}_{n}) = \sqrt{\frac{\sum_{i \in D} (\boldsymbol{x}^{+}_{ni} - \mathsf{mean}(\boldsymbol{x}^{+}_{n}))^2}{|\mathcal{D}|}} \tag{2.14d}$$

This re-scaling is important to stabilize the gradient descent during the training of NNs. When a NN is trained, the loss gradients are back-propagated through the network to update the learnt parameters. However, if the input distribution changes drastically for the next input sample, then learning is slowed down and may even stop in extreme cases.

Position-wise FFN

FFNs, as described in Section 2.1.1, are applied in both the encoder and decoder. In the work of Vaswani et al. [57], a FFN with 1 hidden layer and a ReLU activation function is applied to the representation of the input tokens ($\boldsymbol{x}$). This is shown in Equation 2.15, where W_1 and b_1 denote the learnt weights and biases for the hidden layer and W_2 and b_2 for the FFN output layer.

$$FFN(\boldsymbol{x}) = \max(0, \boldsymbol{x} \cdot \mathbf{W}^{\mathfrak{h}} + \mathbf{b}^{\mathfrak{h}})\mathbf{W}^{\mathfrak{o}} + \mathbf{b}^{\mathfrak{o}} \tag{2.15}$$

The purpose of the FFN in the transformer architecture is not explicitly stated by its creators, however, Sonkar and Baraniuk [50] perform experiments to investigate its effect. These experiments show that

the FFN is used to spread out the token embeddings from the multi-head attention component. Without the FFN the token embeddings eventually converge to one single embedding and lose information about individual tokens.

Output Prediction Head

Additional layers may also be added to the transformer output to create what is known as an output prediction head, which serves as the final layer in the transformer architecture. This prediction head is especially crucial in natural language processing tasks, where its role is to convert numerical token representations back into words. For such tasks, the transformer predicts the next token by estimating the probability of each possible token in the vocabulary being the next one. The token with the highest probability is then selected as the predicted next token.

Vaswani et al. [57] implement this process using a combination of a linear layer and the softmax function. The linear layer projects the transformer's output into a space with dimensions equal to the vocabulary size, using the same weight matrix as the embedding layer. This weight sharing reduces the number of trainable parameters, improving efficiency. Subsequently, the softmax function converts the linear layer's output into a probability distribution over all tokens in the vocabulary, ensuring that the probabilities sum to 1. This probabilistic output (y) enables the model to select the most likely next token using the decoder output ($\boldsymbol{x}$) as shown in Equation 2.16

$$y = \mathsf{softmax}\left(\boldsymbol{x} \cdot \boldsymbol{W}_s\right) \tag{2.16}$$

In non-NLP applications, output prediction heads may also be included. These prediction heads are frequently made up of pooling and linear layers. Pooling layers compress the transformer's output into a fixed-size representation by applying operations such as average pooling (computing the mean across tokens) or max pooling (selecting the maximum value across tokens). This compression step reduces the sequence dimension, enabling the model to produce a fixed-size output regardless of the input sequence length. The compressed representation can then be passed through a linear projection layer, which maps it to the required output dimensions for the specific predictive task.

2.1.3. TNN Illustrative Example

In the previous subsection, the key components of the transformer architecture are introduced, using the Vanilla Transformer as a reference to show how each module is implemented and its role in the model. This provides a foundational understanding of the transformer's structure and functionality which will be expanded upon in this subsection. Here an illustrative example of a TNN is presented that shifts the focus from NLP to learning from numeric data sequences. The example considers an image classification use case, where a TNN is trained to determine the number displayed in a figure (see Figure 2.8).

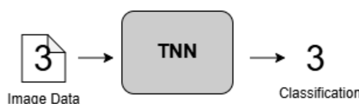


Figure 2.8: Transformer for image classification of hand written digits

The first step in training the TNN would be to prepare the data by formatting it as a sequence. For images, this involves converting each image in the training dataset into a matrix. The matrix is constructed such that each cell corresponds to a pixel in the image and the value of the cell represents the colour at the given pixel. For gray-scale images, where values range from 0 (white) to 1 (black), this transformation creates a structured representation, as shown in Figure 2.9. Once the matrix is created, it can be flattened to create a single vector through row concatenation to create a data sequence which can be fed directly into the transformer. In this example, an encoder-only model is considered as these are the most popular for computer vision applications [22]. The TNN architecture is defined in Figure 2.10 where the input image refers to the sequentially transformed image. This architecture shares similar characteristics as the Vanilla Transformer and all components work in the same manner as described in the previous subsection.

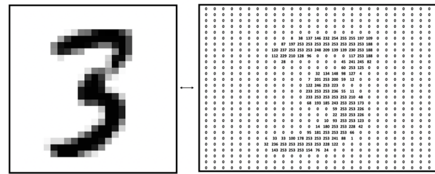


Figure 2.9: Conversion of grey-scale image to matrix representation. Image retrieved from [9].

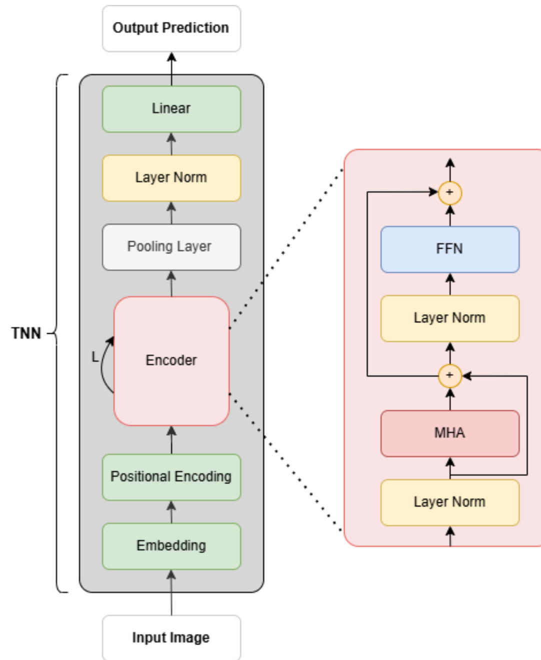


Figure 2.10: Example vision transformer

Given an image with dimensions $I \times I$, the input image, $\mathbf{x}$, will have dimensions $I^2 \times 1$. To generalise this, let the sequence length dimension, I^2 be denoted by N and the feature dimension, 1, be denoted by F. Figure 2.11 illustrates how the learnt embedding is used to transform the input data to be represented in a new feature dimension, D.

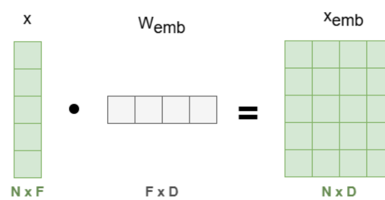


Figure 2.11: Input embedding operation

The next layer in the transformer is the positional encoding layer. This provides a unique value that indicates an element's position in the sequence and its proximity to other values in the sequence. The positional encoding is essential in this case, without it the TNN would only have information about the values of pixels in the image but no understanding of their configuration. As discussed previously, the positional encoding could be a learnt parameter or calculated using the sinusoidal encoding proposed by Vaswani et al. (2017). The definition of the encoding is merely a design choice in the TNNs architecture. From an implementation standpoint, the positional encoding is a fixed-valued matrix of size $N \times D$ which is added to the embedded input image, $\mathbf{x}_{\text{emb}}$ (Figure 2.12).

The resulting positionally encoded input values, $\mathbf{x}_{\pi}$ are then passed through the encoder layers. Each

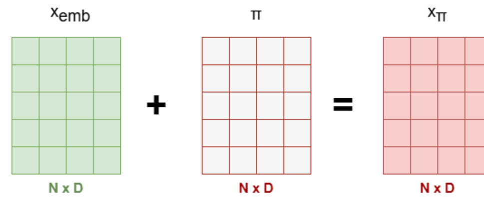
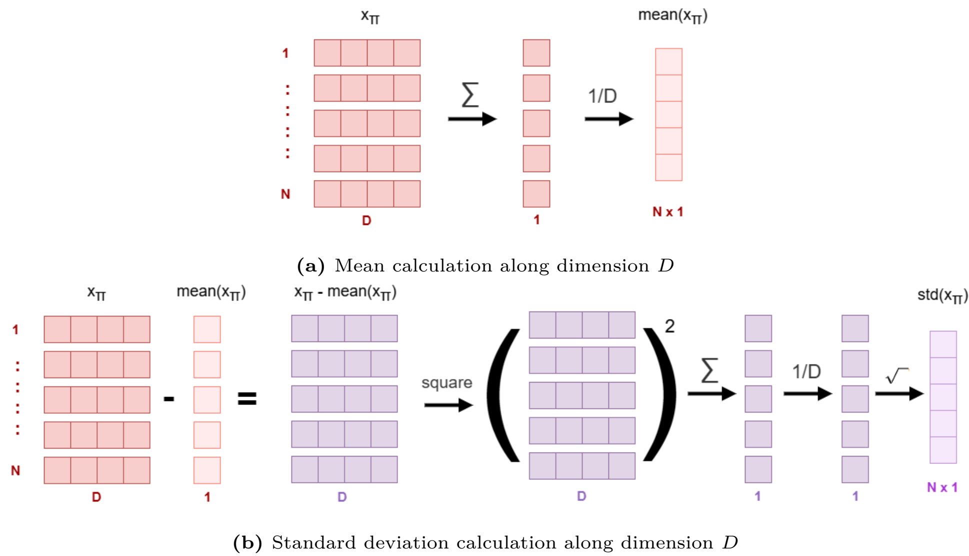


Figure 2.12: Positional encoding operations

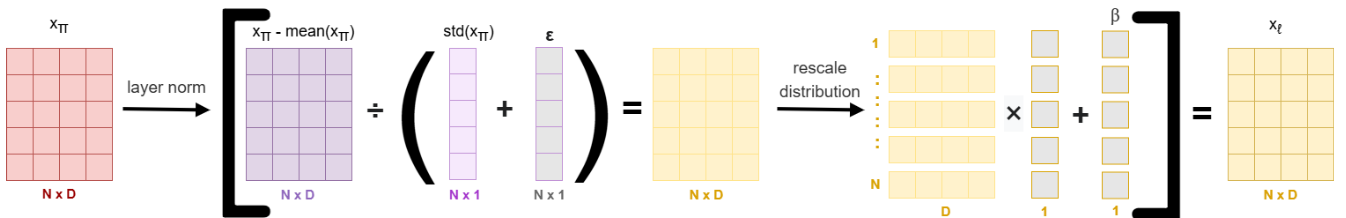
encoder layer consists of normalisation, multi-head attention, residual, and feed-forward NN modules. These submodules are applied sequentially and will be discussed in the order of their execution.

The first submodule, layer normalisation, re-scales and shifts values along the D dimension. These features in the D dimension collectively represent a single pixel in the original image. Layer normalisation ensures that the distribution of feature values remains within a fixed range. The range is determined by fixed gain (λ) and offset (β) parameters, which are learnt during training. Another fixed parameter, denoted by ϵ, is used to prevent division by zero. Figure 2.13 depicts the transformation performed in the layer normalisation module. The mean and standard deviation are computed along the D dimension. The results from these operations are then supplied to the layer normalisation calculation.



(a) Mean calculation along dimension D

(b) Standard deviation calculation along dimension D



(c) Layer normalisation of positionally encoded input ($\mathbf{x}_\pi$). Arrows indicate that the function placed above it is being applied.

Figure 2.13: Layer normalisation operation

At this stage, the input image has been transformed through a series of preparatory steps, resulting in a representation denoted by $\mathbf{x}_\ell$. The next step involves applying the multi-head attention mechanism, which is designed to extract patterns and build a contextual understanding of the input image. The attention mechanism creates the Query, Key and Value projections (Figure 2.14). These matrices have

three dimensions, the first for the sequence length (N), the second for the attention heads (H) and the last for the size of each attention head (S). Each attention head is viewed as a way to capture a different attribute about the relationship between pixels. The head size parameter can be increased to enable more nuanced insights to be gathered at each head.

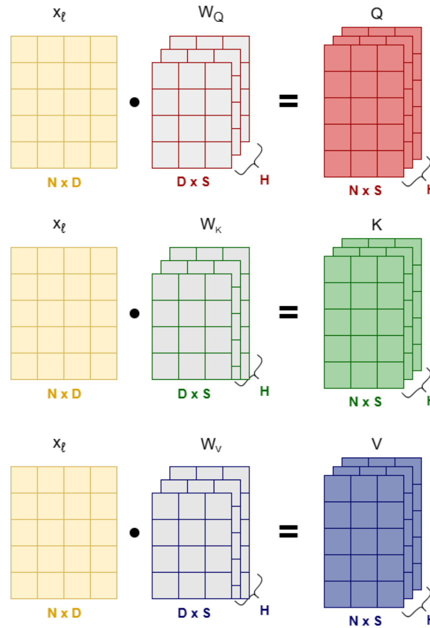


Figure 2.14: Calculation of Query, Key and Value projections

The Query, Key and Value matrices are used to compute the importance of each pixel in the image in relation to the other pixels. This results in a compatibility score which can be interpreted as determining how much "attention" a given pixel should allocate to others. This attention is what allows the TNN to gain contextual insights and to capture long-range dependencies between elements in the flattened image.

To compute the compatibility scores, the Query matrix is multiplied by the transpose of the Key matrix, producing a measure of the similarity between each pair of pixels. In Figure 2.15, it can be seen more clearly how each pair of pixels $n, m \in N$ is examined in the dot product between the Query and Key matrices for each head $h \in H$. This result is scaled by the factor $|\sqrt{S}|$ to prevent large values from saturating the softmax function. The softmax function is then applied to the scaled compatibility scores, converting them into attention weights. These weights form a probability distribution, guaranteeing that their values sum to 1 for each pixel. This probability distribution is useful because it allows the model to consider the relative importance of the other pixels.

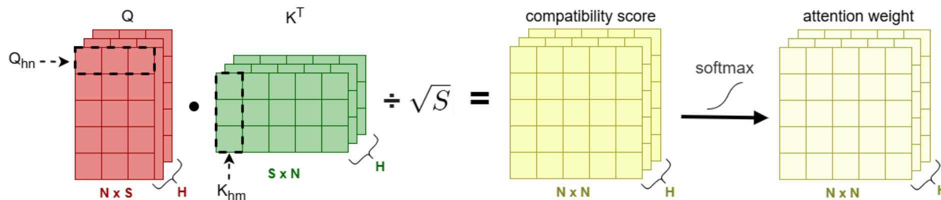


Figure 2.15: Attention weight calculation

Given these attention weights, the dot product between the attention weight and Value matrix elements is found for each attention head. This step is the most crucial part of the attention mechanism. The weighted sum of the Value matrix effectively aggregates a representation of each pixel based on the attention distribution. The resulting context-informed pixel representations are then converted back to

their original embedded dimensions, $N \times D$. This procedure is carried out in two steps. The first step projects the pixel representations to the embedding dimension, D, using a learnt weight matrix (W_O). The second step aggregates the 3-dimensional matrices by summing over the head dimension (H). The calculation of the MHA output, $\mathbf{x}_\alpha$, is depicted in Figure 2.16, highlighting how pixels are transformed and aggregated to produce the final output.

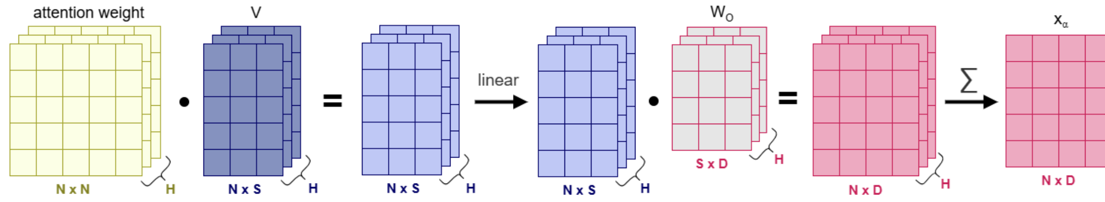


Figure 2.16: Calculation of multi-head attention output

Next, the output of the MHA module is passed through a residual layer, which helps preserve linear relationships by adding the output of previous TNN layers back into the model. Following the residual layer, the representation of the image data is normalized as shown in Figure 2.17.

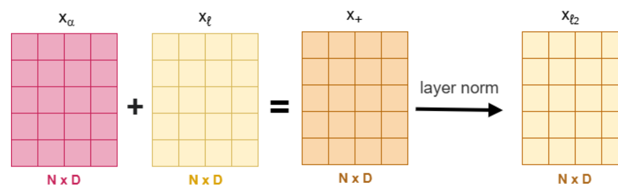


Figure 2.17: Post multi-head attention residual and normalisation calculation

The normalized representation is then passed to through the feed-forward neural network (FFN) (Figure 2.18). The FFN has one hidden layer with R nodes and uses a ReLU activation function.

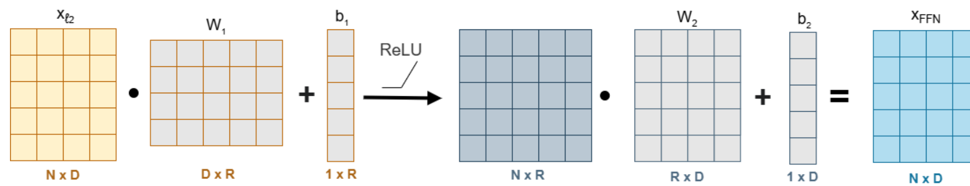


Figure 2.18: Feed-forward neural network operation

The final step in the encoder is a residual connection, where the output of the MHA layer is added to the result of the FFN layer (Figure 2.19). The result of the residual connection is then used as input to the next encoder layer. The above processes are repeated for each encoder layer, refining and constructing more intricate representations of the pixel values with each iteration. This allows the model to capture higher-level features and relationships within the input image sequence.

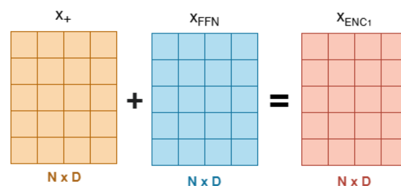


Figure 2.19: Final step in encoder layer

The following pooling, normalisation and linear layers act as an output prediction head for the vision

transformer. The pooling layer condenses the encoder output along the sequence element dimension, N (Figure 2.20). In this example, max pooling is used, which selects the largest value in the N dimension for each feature dimension, D.

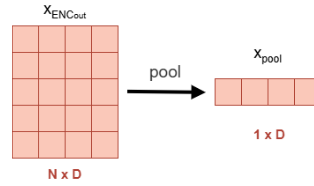


Figure 2.20: Pooling operation

The result of pooling, $\mathbf{x}_{\text{pool}}$, is normalised and passed through a linear layer to make the classification prediction. The linear layer is used to create the classification prediction for each digit (0 to 9). The output of this layer is a vector, storing the associated likelihood for each digit in a corresponding cell (TNN_{out}). The image can then be classified as the digit with the highest likeliness score (Figure 2.21). If the TNN is given an input image showing the number 3, then it can be expected that the highest likeliness score is found at position 3 in the TNN_{out} array; granted that the transformer is well-trained. This indicates that the image is classified correctly as the number 3.

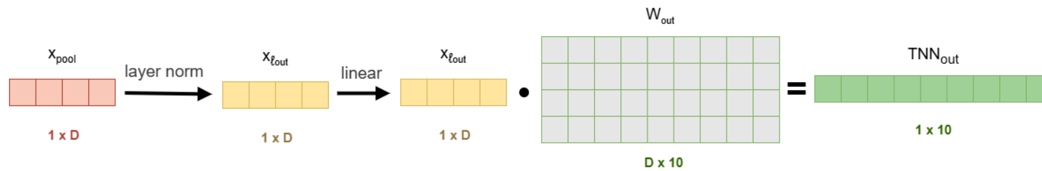


Figure 2.21: Output prediction head operations

2.2. Neural Networks for Optimisation

This section describes the state-of-the-art in the integration of NNs within optimisation problems. NNs excel at capturing sophisticated non-linear interactions between variables. Such NNs are useful in creating complex models to represent systems with unknown or difficult-to-represent mathematical structures [16] [5]. Research has investigated the use of ML models in a mathematical optimisation setting and the development of tools to integrate these representations into existing optimisation frameworks. These advancements have brought about a way to optimise systems with complex components to meet given objectives.

Current works propose optimisation formulations for various NNs and activation functions. These studies highlight the challenge of efficiently optimising over mathematical programs that include a large number of non-linear, non-smooth and non-convex functions. These complex functions increase the difficulty of exploring the search space and can lead to slow or computationally infeasible models. This has lead to research into the design efficient NN formulations that allow for fast global optimisation. These designs aim to create tractable mathematical models for realistic NN models, which are often large.

The following sub-sections describe the state-of-the-art in the integration of ML models into optimisation problems across a variety of real-world applications.

2.2.1. Formulations

Grimstad and Andersson (2019) consider a feed-forward Neural Network (FFN) with ReLU activation functions [16]. The authors model the piece-wise linear ReLU function using Big-M constraints and develop a mixed integer linear program (MILP) formulation of the ReLU FFN. This formulation is evaluated on an oil production optimisation problem that uses 10 ReLU FFNs to model the multi-phase flow in the pipes. Grimstad et al. note that the feasibility of their exact formulation quickly fades with the size of the network. Even after applying bound tightening techniques, it is not possible

to solve optimisation problems with more realistic ReLU FFNs involving thousands of hidden nodes. This results from the number of variables in the formulation scaling linearly with the number of nodes in the trained NN. Anderson et al. (2020) study how the search space of such a Big-M FFN formulation can be reduced [5]. The authors extend upon the MILP Big-M formulation, proposing a MILP formulation of the ReLU FFN that constructs the convex hull of each node in the FFN, drastically reducing the search space. Like the previous Big-M formulation, this approach also becomes infeasible for large FFNs [56].

Tsay et al. (2021), take a different approach in formulating the FFN. The authors propose an adaptive formulation which partitions the FFN formulation, creating distinct upper and lower bounds for each partition [56]. At one extreme, using one partition for each node constructs the convex hull for individual nodes, which requires an exponential number of constraints and auxiliary variables like the formulation created by Anderson et al. [5]. At the other extreme, having one partition for all nodes constructs the convex hull over all nodes, which is more compact but has weaker relaxations. Therefore, the partition size can be used to balance computational efficiency, due to the number of constraints, and relaxation strength. The partition-based formulation was evaluated for shallow FFNs and CNNs with varying partition sizes. The experimental results show a speed up in solve time and an increase in the number of solved problem instances as compared to the Big-M representation proposed by Grimstad and Andersson (2019).

Fischetti and Jo (2018) investigate MILP formulations for FFNs and CNNs with ReLU, max pooling and average pooling activations [14]. The MILP formulation and the effect of proposed bound tightening techniques are assessed using feature visualisation and adversarial machine learning optimisation problems. These examples highlight how optimising over NNs can be beneficial for assessing NNs. In the first application, the optimisation objective is to find inputs that maximise a given activation. This provides insights into how predictions are made in black box NN models. In the second application, the aim is to find the minimum input modifications required to produce an incorrect classification. This provides a measure of the NNs robustness against adverserial inputs. The computational experiments show the possibility to optimise over small NNs within a few seconds. However, the solving times increase drastically to the order of hours for larger NNs.

In an effort to extend optimisation to larger FFNs, Schweidtmann and Mitsos (2019) develop a reduced space formulation of a FFN with hyperbolic tangent activation functions [45]. The formulation presented reduces the dimensionality of the optimisation problem by "hiding" the FFN equations from the solving algorithm. This is implemented by implicitly solving the FNN layers and substituting this result into an optimisation of the over-arching NN. In this way, complex activation functions are not visible during the optimisation of the NN. Furthermore, the formulation is strengthened by constructing tight concave and convex envelopes for the hyperbolic tangent function via its envelopes. Results show that it is indeed possible to optimise over larger FFNs using this approach. A FFN with 700 neurons was solved in under $30s$. In Doncevic et al. (2020), the reduced formulation is extended to Recurrent NNs (RNNs) and applied to solve a model predictive control problem. Additional reformulation techniques are proposed to create a hybrid formulation which exposes one of the hidden variables to the optimiser. Results indicate that reduced formulation of the RNN, and the proposed reformulation techniques both work to lower the computational cost of optimisation. The creation of hybrid reduced space formulations is emphasized as a promising technique for speeding up the global optimisation of complex models.

Yang, Balaprakash, and Leyffer (2022) propose two formulations, similarly investigating scalability to larger FNNs [61]. A MILP-based and a non-convex non-linear program formulations are proposed. The first approach employs Big-M constraints to model the ReLU activation function. This replaces the non-linear max operation with a set of affine linear constraints and introduces a binary auxiliary variable for each hidden node. As a result, this creates a mixed-integer formulation with a convex continuous relaxation. The authors further strengthen the MILP by carefully determining Big-M values; which impacts the size of the search space. Optimality-based bound-tightening techniques are applied to determine the bounds on each node which are then used to instantiate the Big-M values. The non-convex formulation, on the other hand, uses complementary constraints to model the ReLU activation function. This adds one continuous slack variable for every hidden node and avoids the complexity of including integer variables. The formulations are evaluated in three case studies: an engine design problem to minimize emissions, an optimal adversary problem and an oil pump configuration problem.

The results show that the non-convex formulation finds a feasible solution in significantly less time than the MILP formulation using standard non-linear programming solvers; likely due to the lack of integrality constraints. However, these solvers only produce locally optimal solutions to the non-convex problem. Although the MILP version is slower to solve, its convex properties make it capable of solving to global optimality. The proposed formulations are shown to scale to FNNs with hundreds of hidden nodes. Nevertheless, the number of variables in both of the formulations increases with the number of neurons, thus the formulations become infeasible for larger NNs.

Formulations for graph neural networks have also been established. McDonald et al. (2024) propose two GNN formulations [39]. Namely an exact MINLP for the graph convolution neural network by Kipf and Welling [29] and a MILP representation of the popular GraphSAGE model by Hamilton, Ying, and Leskovec [21]. The formulations are assessed on a computer-aided molecular design case study, whereby the GNN models the relationship between a molecular structure and its boiling point. The results show that in most cases the benchmark Genetic Algorithm (GA) finds a solution of equal quality in a shorter time than solving the proposed formulations. However, unlike the MILP formulation, the GA gives no guarantees of converging to the global optimum.

Similarly, Zhang et al. (2023) propose two generalized GNN formulations that support various GNN architectures [64]. The first is a bi-linear formulation while the second is a Big-M formulation that extends upon the MILP GraphSAGE formulation in [39]. A later paper by Zhang et al. (2024) strengthens the Big-M GNN formulation of [63] by introducing symmetry-breaking constraints. The formulation is evaluated using a molecular design case study. The results show that this method can handle larger molecules than the previous works on embedding GNNs ([64],[39]). This emphasises the importance of building efficient formulations and the progress that can be achieved through the strengthening of existing formulations.

Thus far, the studies discussed focus on trained NNs, with fixed weight and bias parameters, applied to model components within optimisation problems. However, other existing works formulate NNs for different purposes. For instance, Thorbjarnarson and Yorke-Smith (2023) and Bernardelli et al. (2024) explore the use of MILP solvers for training NNs, instead of traditional gradient-based approaches. In these works, the learnt weights and biases are decision variables that must be found by the solver.

Thorbjarnarson and Yorke-Smith (2023) formulate binarised and integer-valued FNNs (BNNs and INNs) with sign activation functions as MILPs using Big-M constraints [55]. The authors demonstrate a training strategy which finds the optimal learnt parameters while simultaneously optimising the number of neurons in the architecture. This means that the size of the architecture does not need to be predefined, resulting in more memory-efficient and compact models. Furthermore, this approach eliminates the need for hyper-parameter tuning to configure the gradient descent method (e.g. learning rate, decay, epochs). The main limitation of using MILP solvers for training is that optimisation becomes intractable for large models. This currently limits solver-based training to small NNs with restricted domains (e.g. discritised FNNs) and small amounts of training data. Although the authors propose batch training techniques which greatly increase the amount of training data, it is only possible to optimise over small models (single layer and few neurons) with relatively small datasets in a reasonable amount of time.

Bernardelli et al. (2024) further extend this research, again considering BNNs and INNs [8]. In their work a multi-objective approach is described to concurrently optimise the accuracy, robustness to small perturbations and sparsity of the NN. This creates NNs that require little post-training refinements and can move more quickly from development to deployment. Compared to the state-of-the-art in solver-based training, their method achieves superior test accuracy with up to 75.3% of NN connections removed. Although these MILP-based training methods are currently constrained by model size, they present a promising alternative to traditional gradient-based techniques. These approaches shift the machine learning paradigm away from resource-intensive models that demand significant memory and specialised hardware such as GPUs, towards more compact models with proven optimal configurations.

The literature surveyed has outlined methods and applications of mathematical models for the FFN, RNN, CNN and GNN. The formulations consider how to effectively model the various NN layers as well as commonly used activations, namely the sign, hyperbolic tan and ReLU functions. Currently, no works examine how the TNN and softmax activation functions can be implemented in an optimisation setting. This research aims to address this gap.

2.2.2. Implementation

The extensive research into formulating the FFN has led to the development of libraries to implement them in various coding languages and optimisation frameworks. López-Flores, Ramírez-Márquez, and Ponce-Ortega (2024) survey the existing FFN formulations and the libraries that can be used to instantiate them [37]. Of note are the `OMLT` package [11] which links to the `Pyomo` framework [10] [24] and the `GurobiML` [18] for the `Gurobipy` framework [19].

`Pyomo` is an open-source optimisation framework which interfaces to numerous open-source and commercial solvers. The `OMLT` package allows for ML models to be constructed within the `Pyomo` framework. It offers a variety of FFN formulations with differing levels of complexity and available activation functions. For example, it includes the Big-M, reduced space and complementary constraint formulations of the FFN mentioned in the previous sub-section. This in addition to the numerous solvers available in the `Pyomo`, provides a very flexible modelling environment.

The `Gurobipy` framework on the other hand provides a more limited modelling environment since it only interfaces to the `Gurobi`. `Gurobi` is a state-of-the-art commercial solver which includes procedures for solving MINLPs to global optimality; like the proposed TNN formulation. The `GurobiML` package provides a Big-M formulation of the FFN with ReLU activations. While this modelling environment is less flexible, it allows access for the proposed TNN formulated to be evaluated using a high-performing MINLP solver.

3
Methods

This work focuses on embedding the TNN in a mathematical program, whereby the transformer models part of the problem definition. To achieve this, the ML model must be recreated in a format that is compatible with global optimisation techniques. This chapter outlines methods for constructing such a formulation of a trained TNN. As such, the formulation is designed to mimic the behaviour of the trained model and preserve its predictive performance. In this context, both the inputs and outputs of the transformer are decision variables that must be determined during the optimisation process. This contrasts with traditional ML approaches, which use fixed inputs to infer an outcome. Instead, this approach optimises the output of the transformer to meet a specific objective, thus constraining the values that the input can take. As a result, optimising over the TNN provides the input conditions required to meet a predefined goal.

This chapter begins by presenting the mathematical formulations for each constituent component of the TNN architecture. Subsequently, proposed methods for strengthening the formulation and enhancing the solving performance are described. Finally, the software implementation of the transformer is explained. This provides practical insights into how the software interfaces with popular python libraries, such as `HuggingFace`, `Pytorch` and `Keras`, to convert trained ML models into optimisation-based formulations.

3.1. Transformer Formulation

A non-convex MINLP formulation of the TNN is presented in the following sub-sections for the Vanilla TNN architecture [57]. Although linearisation and convexification techniques could be applied to create a more tractable mathematical model, they are not used as this would lead to an approximated version of the transformer. Refraining from these techniques ensures that the formulation gives the same predictions and behaviour as the trained transformer. An overview of the sets, parameters, variables, and equations used in the following component descriptions is shown in Table 3.1.

3.1.1. Input Embedding

Given an input variable, $\boldsymbol{x}_{i,j}$, with sequence dimensions $i \in \mathrm{I}$ and feature dimensions $j \in \mathrm{J}$, this input variable can be embedded to a new set of feature dimensions, D as follows. The multiplication between the input and weight matrices is computed for each element at position (i, d), by summing the product of the input and learned weight ($\mathbf{W}^e$) over J. Then a learned bias vector ($\mathbf{b}^e$) is added (Equation 3.1).

$$\boldsymbol{x}^e_{i,d} = \sum_{j \in J} \boldsymbol{x}_{i,j} \mathbf{W}^e_{j,d} + \mathbf{b}^e_d \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.1}$$

Positional encoding is implemented using a linear layer which adds the encoding vector ($\mathbf{b}^\pi$) to the embedded input. The encoding vector is a fixed parameter that is determined by the positional encoding method used in the trained transformer. Fixed sinusoidal values or learned values are commonly used

TNN Layer	Sets	Parameters	Variables	Formulation Eqns
Linear	$\mathcal{I}, \mathcal{J}, \mathcal{D}$	$\mathbf{W}^e_{j,d}, \mathbf{b}^e_d$	$\boldsymbol{x}^e_{i,d}$	3.1
PE	$\mathcal{I}, \mathcal{D}$	$\mathbf{b}^\pi_{i,d}$	$\boldsymbol{x}^\pi_{i,d}$	3.2
Layer Norm	$\mathcal{I}, \mathcal{D}$	$\epsilon^\ell, \boldsymbol{\gamma}^\ell_d, \boldsymbol{\beta}^\ell_d$	$\boldsymbol{\xi}^{\ell_1}_i, \overline{\boldsymbol{x}}^{\alpha^+}_i, \boldsymbol{\upsilon}^{\ell_1}_{i,d}, \boldsymbol{\Upsilon}^{\ell_1}_{i,d}, \boldsymbol{\delta}^{\ell_1}_i, \boldsymbol{\sigma}^{\ell_1}_i, \boldsymbol{\Theta}^{\ell_1}_{i,d}, \boldsymbol{x}^{\ell_1}_{i,d}$	3.14–3.21
Residual	$\mathcal{I}, \mathcal{D}$	-	$\boldsymbol{x}^+_{i,d}$	3.13
MHA	$\mathcal{I}, \mathcal{D}, \mathcal{H}, \mathcal{K}$	$\mathbf{W}^{Q^\alpha}_{d,h,k}, \mathbf{W}^{K^\alpha}_{d,h,k}, \mathbf{W}^{V^\alpha}_{d,h,k}, \mathbf{W}_{o^\alpha_{d,h,k}}, \mathbf{b}^{Q^\alpha}_{h,k}, \mathbf{b}^{K^\alpha}_{h,k}, \mathbf{b}^{V^\alpha}_{h,k}, \mathbf{b}^o_{\alpha_d}$	$\boldsymbol{Q}^\alpha_{h,i,k}, \boldsymbol{K}^\alpha_{h,i',k}, \boldsymbol{V}^\alpha_{h,i',k}, \boldsymbol{\omega}^\alpha_{h,i,i'}, \boldsymbol{A}^\alpha_{h,i,k}, \boldsymbol{x}^\alpha_{i,d}, \boldsymbol{C}^\alpha_{h,i,i'}, \boldsymbol{E}^\alpha_{h,i,i'}, \boldsymbol{S}^\alpha_{h,i},$	3.3–3.11, 3.37–3.42
FFN	$\mathcal{I}, \mathcal{D}, \mathcal{F}$	$\mathbf{W}^{\mathfrak{H}}_{d,f}, \mathbf{W}^{\mathfrak{o}}_{f,d}, \mathbf{b}^{\mathfrak{H}}_f, \mathbf{b}^{\mathfrak{o}}_d, M^-, M^+$	$\boldsymbol{\Omega}^{\mathfrak{H}}_{i,f}, \boldsymbol{z}^{\mathfrak{H}}_{i,f}, \boldsymbol{x}^{\mathfrak{H}}_{i,d}$	3.22–3.27

Table 3.1: An overview of the sets, parameters, variables, and equations used for each layer.

as encodings.

$$\boldsymbol{x}^\pi_{i,d} = \boldsymbol{x}^e_{i,d} + \mathbf{b}^\pi_{i,d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.2}$$

3.1.2. Encoder

The encoder consists of residual, layer normalization, multi-head attention and FFN layers (see Figure 2.3). For each encoder layer, the following constraints are added to the mathematical model.

First, the embedded input is linearly transformed to be represented by the Query ($\mathbf{Q}^\alpha$), Key ($\mathbf{K}^\alpha$) and Value ($\mathbf{V}^\alpha$) matrices. The embedding dimensions (D) are projected to H attention heads, each with K dimensions. These calculations use learned weights ($\mathbf{W}$) and biases ($\mathbf{b}$).

$$\boldsymbol{Q}^\alpha_{h,i,k} = \sum_{d\in\mathcal{D}} \boldsymbol{x}^\pi_{i,d}\mathbf{W}^{Q^\alpha}_{d,h,k} + \mathbf{b}^{Q^\alpha}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.3}$$

$$\boldsymbol{K}^\alpha_{h,i',k} = \sum_{d\in\mathcal{D}} \boldsymbol{x}^\pi_{i',d}\mathbf{W}^{K^\alpha}_{d,h,k} + \mathbf{b}^{K^\alpha}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i' \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.4}$$

$$\boldsymbol{V}^\alpha_{h,i',k} = \sum_{d\in\mathcal{D}} \boldsymbol{x}^\pi_{i',d}\mathbf{W}^{V^\alpha}_{d,h,k} + \mathbf{b}^{V^\alpha}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i' \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.5}$$

The attention weight (Equation 2.10) is computed in multiple steps. The compatibility ($\boldsymbol{C}$) between query ($\boldsymbol{Q}$) and value ($\boldsymbol{V}$) matrices is determined by:

$$\boldsymbol{C}^\alpha_{h,i,i'} = \frac{1}{\sqrt{|\mathcal{K}|}} \sum_{k\in\mathcal{K}} \boldsymbol{Q}^\alpha_{h,i,k}\boldsymbol{K}^\alpha_{h,i',k} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.6}$$

where $\mathcal{K}$ is the dimension of each attention head (head size). Given the compatibility, the exponential of the compatibility ($\boldsymbol{E}^\alpha$) and its sum ($\boldsymbol{S}^\alpha$) are calculated in Equation 3.7 and Equation 3.8.

$$\boldsymbol{E}^\alpha_{h,i,i'} = \exp(\boldsymbol{C}^\alpha_{h,i,i'}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.7}$$

$$\boldsymbol{S}^\alpha_{h,i} = \sum_{i'\in\mathcal{I}} \boldsymbol{E}^\alpha_{h,i,i'} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.8}$$

The attention weight ($\boldsymbol{\omega}^\alpha$) is determined as the ratio of the exponential of the compatibility and its sum over I. To avoid division by zero, this relation is formulated as the following product:

$$\boldsymbol{\omega}^\alpha_{h,i,i'} \cdot \boldsymbol{S}^\alpha_{h,i} = \boldsymbol{E}^\alpha_{h,i,i'} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.9}$$

The attention score ($\boldsymbol{A}^{\alpha}$), is determined from the product of the attention weight and Value matrices in Equation 3.10. The attention score is then used to compute the attention-informed representation of the input, $\boldsymbol{x}^{\alpha}$.

$$\boldsymbol{A}^{\alpha}_{h,i,k} = \sum_{i' \in \mathcal{I}} \boldsymbol{\omega}^{\alpha}_{h,i,i'} \boldsymbol{V}^{\alpha}_{h,i',k} \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.10}$$

$$\boldsymbol{x}^{\alpha}_{i,d} = \mathbf{b}^{o_{\alpha}}_{d} + \sum_{h \in \mathcal{H}} \sum_{k \in \mathcal{K}} \boldsymbol{A}^{\alpha}_{h,i,k} \mathbf{W}^{o_{\alpha}}_{d,h,k} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.11}$$

$$\tag{3.12}$$

The residual connection between the MHA input and output is handled by the following summation.

$$\boldsymbol{x}^{\alpha^{+}}_{i,d} = \boldsymbol{x}^{\alpha}_{i,d} + \boldsymbol{x}^{\pi}_{i,d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.13}$$

The first layer normalization layer (ℓ_1) is applied to the residual layer's output ($\boldsymbol{x}^{\alpha^{+}}$). The layer normalisation calculation (Equation 2.14) is broken down into the following steps. The mean of the input representation ($\overline{\boldsymbol{x}}^{\alpha^{+}}$) over the embedding dimension, D, is given by,

$$\boldsymbol{\xi}^{\ell_1}_{i} = \sum_{d \in \mathcal{D}} \boldsymbol{x}^{\alpha^{+}}_{i,d} \qquad \forall i \in \mathcal{I} \tag{3.14}$$

$$\overline{\boldsymbol{x}}^{\alpha^{+}}_{i} = \frac{\boldsymbol{\xi}^{\ell_1}_{i}}{|D|} \qquad \forall i \in \mathcal{I} \tag{3.15}$$

where $\boldsymbol{\xi}^{\ell_1}_{i}$ is the sum of the input values for each element i in the sequence. The standard deviation, $\boldsymbol{\sigma}^{\ell_1}$, is derived from the variance, $\boldsymbol{\delta}^{\ell_1}$; and a small constant ϵ^{ℓ_1} is added to prevent division by 0. It is important to use the same value of ϵ^{ℓ_1} that is used during training to minimize discrepancies between the formulated TNN's output and the trained TNN's output.

$$\boldsymbol{\upsilon}^{\ell_1}_{i,d} = \boldsymbol{x}^{\alpha^{+}}_{i,d} - \overline{\boldsymbol{x}}^{\alpha^{+}}_{i} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.16}$$

$$\boldsymbol{\Upsilon}^{\ell_1}_{i,d} = (\boldsymbol{\upsilon}^{\ell_1}_{i,d})^2 \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.17}$$

$$\boldsymbol{\delta}^{\ell_1}_{i} = \frac{1}{|D|} \sum_{d \in \mathcal{D}} \boldsymbol{\Upsilon}^{\ell_1}_{i,d} \qquad \forall i \in \mathcal{I} \tag{3.18}$$

$$(\boldsymbol{\sigma}^{\ell_1}_{i})^2 = \boldsymbol{\delta}^{\ell_1}_{i} + \epsilon^{\ell_1} \qquad \forall i \in \mathcal{I} \tag{3.19}$$

The layer normalization output, $\boldsymbol{x}^{\ell_1}_{i,d}$ is determined using the learned scaling and offset parameters ($\boldsymbol{\gamma}^{\ell_1}_{d}$, $\boldsymbol{\beta}^{\ell_1}_{d}$) along with the mean and standard variation variables.

$$\boldsymbol{\Theta}^{\ell_1}_{i,d} = \frac{\boldsymbol{\upsilon}^{\ell_1}_{i,d}}{\boldsymbol{\sigma}^{\ell_1}_{i}} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.20}$$

$$\boldsymbol{x}^{\ell_1}_{i,d} = \boldsymbol{\gamma}^{\ell_1}_{d} \boldsymbol{\Theta}^{\ell_1}_{i,d} + \boldsymbol{\beta}^{\ell_1}_{d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.21}$$

After normalization, the transformed input is passed through the feed-forward NN (FFN). The FFN is implemented using a Big-M formulation from existing libraries. The Big-M formulation of the ReLU FFN is shown in Equations 3.22–3.26. Parameters M^{-} and M^{+} refer to the upper and lower bounds of the function $f(x) = x \cdot W + b$. These big-M parameters are not chosen, but rather determined in the function that implements the FFN. More information about this formulation can be found in [5].

$$\boldsymbol{\Omega}^{\mathfrak{H}}_{i,f} \geq 0 \qquad \forall i \in \mathcal{I}, \forall f \in F \tag{3.22}$$

$$\boldsymbol{\Omega}^{\mathfrak{H}}_{i,f} \geq \sum_{d\in\mathcal{D}} \boldsymbol{x}^{\ell_1}_{i,d}\mathbf{W}^{\mathfrak{H}}_{d,f} + \mathbf{b}^{\mathfrak{H}}_{f} \qquad \forall i \in \mathcal{I}, \forall f \in F \tag{3.23}$$

$$\boldsymbol{\Omega}^{\mathfrak{H}}_{i,f} \leq (\sum_{d\in\mathcal{D}} \boldsymbol{x}^{\ell_1}_{i,d}\mathbf{W}^{\mathfrak{H}}_{d,f} + \mathbf{b}^{\mathfrak{H}}_{f}) - M^{-}(1 - \boldsymbol{z}^{\mathfrak{H}}_{i,f}) \qquad \forall i \in \mathcal{I}, \forall f \in F \tag{3.24}$$

$$\boldsymbol{\Omega}^{\mathfrak{H}}_{i,f} \leq M^{+}\boldsymbol{z}^{\mathfrak{H}}_{i,f} \qquad \forall i \in \mathcal{I}, \forall f \in F \tag{3.25}$$

$$\boldsymbol{x}^{\mathfrak{H}}_{i,d} = \sum_{f\in F} \boldsymbol{\Omega}^{\mathfrak{H}}_{i,f}\mathbf{W}^{\mathfrak{o}}_{f,d} + \mathbf{b}^{\mathfrak{o}}_{d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.26}$$

$$\boldsymbol{z}^{\mathfrak{H}}_{i,f} \in \{0,1\} \qquad \forall i \in \mathcal{I}, \forall f \in F \tag{3.27}$$

Finally, the residual connection between FFN input and output is added and the output of this residual layer is normalized (equations 3.28 - 3.36). This is implemented in the same way as the previously described residual and layer normalization components.

$$\boldsymbol{x}^{1^+}_{i,d} = \boldsymbol{x}^{\mathfrak{H}}_{i,d} + \boldsymbol{x}^{\ell_1}_{i,d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.28}$$

$$\boldsymbol{\xi}^{\ell_2}_{i} = \sum_{d\in\mathcal{D}} \boldsymbol{x}^{1^+}_{i,d} \qquad \forall i \in \mathcal{I} \tag{3.29}$$

$$\overline{\boldsymbol{x}}^{1^+}_{i,d} = \frac{\boldsymbol{\xi}^{\ell_2}_{i}}{|D|} \qquad \forall i \in \mathcal{I} \tag{3.30}$$

$$\boldsymbol{\upsilon}^{\ell_2}_{i,d} = \boldsymbol{x}^{\alpha^+}_{i,d} - \overline{\boldsymbol{x}}^{1^+}_{i,d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.31}$$

$$\boldsymbol{\Upsilon}^{\ell_2}_{i,d} = (\boldsymbol{\upsilon}^{\ell_2}_{i,d})^2 \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.32}$$

$$\boldsymbol{\delta}^{\ell_2}_{i} = \frac{1}{|D|}\sum_{d\in\mathcal{D}} \boldsymbol{\Upsilon}^{\ell_2}_{i,d} \qquad \forall i \in \mathcal{I} \tag{3.33}$$

$$(\boldsymbol{\sigma}^{\ell_2}_{i})^2 = \boldsymbol{\delta}^{\ell_2}_{i} + \epsilon^{\ell_2} \qquad \forall i \in \mathcal{I} \tag{3.34}$$

$$\boldsymbol{\Theta}^{\ell_2}_{i,d} = \frac{\boldsymbol{\upsilon}^{\ell_2}_{i,d}}{\boldsymbol{\sigma}^{\ell_2}_{i}} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.35}$$

$$\boldsymbol{x}^{\ell_2}_{i,d} = \boldsymbol{\gamma}^{\ell_2}_{d}\boldsymbol{\Theta}^{\ell_2}_{i,d} + \boldsymbol{\beta}^{\ell_2}_{d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.36}$$

3.1.3. Decoder

The decoder component of the TNN can be implemented in a similar manner. The only differences occur in the masked multi-head attention layer and the cross attention layer between the encoder and the decoder. The masked MHA is implemented by adapting the calculation of the compatibility (3.6). Let I_{dec} be the set denoting the sequence elements for the decoder input. Then masking is applied such that each element at position i in the sequence only computes the compatibility between itself and preceding sequence elements at positions $i' \leq i$. The following sequence elements are masked by leaving the compatibility values unconstrained for $i' \leq i$, and setting the exponential of the compatibility ($\boldsymbol{E}^{\alpha}_{h,i,i'}$) to 0. This results in attention weights ($\boldsymbol{\omega}^{\alpha}_{h,i,i'}$) equal to zero ensuring that subsequent elements have no influence on preceding ones.

$$\boldsymbol{C}^{\alpha}_{h,i,i'} = \frac{1}{\sqrt{|\mathcal{K}|}}\sum_{k\in\mathcal{K}} \boldsymbol{Q}^{\alpha}_{h,i,k}\boldsymbol{K}^{\alpha}_{h,i',k}, \qquad \text{if } i' \leq i \quad \forall h \in \mathcal{H}, \forall i,i' \in \mathcal{I}^{\mathsf{dec}} \tag{3.37}$$

$$\boldsymbol{E}^{\alpha}_{h,i,i'} = 0, \qquad \text{if } i' \geq i \quad \forall h \in \mathcal{H}, \forall i,i' \in \mathcal{I}^{\mathsf{dec}} \tag{3.38}$$

$$\tag{3.39}$$

For cross attention between the encoder and decoder equations 3.3– 3.5 are adapted. The Query variable for this block is computed using the output of the previous decoder layer ($\boldsymbol{x}_{\mathsf{dec}}$). The Key and Value variables, however, are calculated from the final output of the encoder ($\boldsymbol{x}_{\mathsf{enc}}$).

$$\boldsymbol{Q}^{\alpha}_{h,i,k} = \sum_{d \in \mathcal{D}} \boldsymbol{x}^{\text{dec}}_{i,d} \mathbf{W}^{Q^{\alpha}}_{d,h,k} + \mathbf{b}^{Q^{\alpha}}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I}^{\text{dec}}, \forall k \in \mathcal{K} \tag{3.40}$$

$$\boldsymbol{K}^{\alpha}_{h,i',k} = \sum_{d \in \mathcal{D}} \boldsymbol{x}^{\text{enc}}_{i',d} \mathbf{W}^{K^{\alpha}}_{d,h,k} + \mathbf{b}^{K^{\alpha}}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i' \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.41}$$

$$\boldsymbol{V}^{\alpha}_{h,i',k} = \sum_{d \in \mathcal{D}} \boldsymbol{x}^{\text{enc}}_{i',d} \mathbf{W}^{V^{\alpha}}_{d,h,k} + \mathbf{b}^{V^{\alpha}}_{h,k} \qquad \forall h \in \mathcal{H}, \forall i' \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.42}$$

3.1.4. Output Prediction Head

Pooling activation layers are often used in output prediction heads. The `OMLT` library has an existing pooling layer function that implements max pooling (3.43). For average pooling, equation 3.44 is used. Generally, the input to the pooling layer ($\boldsymbol{x}_{i,d}$) is the output of the TNN and is followed by a linear projection.

$$\boldsymbol{x}^{\text{pool}}_{d} = \max_{i \in \mathcal{I}}(\boldsymbol{x}_{i,d}) \qquad \forall d \in \mathcal{D} \tag{3.43}$$

$$\boldsymbol{x}^{\text{pool}}_{d} = \frac{1}{|I|} \sum_{i \in \mathcal{I}} \boldsymbol{x}_{i,d} \qquad \forall d \in \mathcal{D} \tag{3.44}$$

3.2. Bounds Tightening

Global optimization seeks to find the best solution of an objective function over a well-defined feasible domain. This domain is defined by bounds and constraints on the decision variables [46]. When the domain of decision variables is restricted, the search space becomes smaller and there are fewer candidate solutions to evaluate which can significantly improve solve efficiency.

One popular method of bounding variables is feasibility-based bound tightening (FBBT). The FBBT method applies interval arithmetic to propagate the bounds of an input variable, to restrict the domain of other associated variables [7]. Although FBBT techniques can provide weaker bounds as the number of propagations increases, they are a fast and computationally efficient way of reducing variable domains.

The application of these low-cost bounds to transformer components are described in the following subsections. Applying pre-computed propagated bounds complements the bounding strategies employed by state-of-the-art solvers by providing a smaller initial search space, which can lead to faster solve times. This is especially helpful for large or complex models like the proposed TNN formulation. These propagated bounds are extended upon in Section 3.3, to provide the solver with function-specific information that can further reduce the search space.

3.2.1. Linear Layer

Let $\mathsf{LB}\,(\boldsymbol{x}_{i,j})$ and $\mathsf{UB}(\boldsymbol{x}_{i,j})$ denote the upper and lower bounds on the input variable at sequence position i and dimension j. Given that the linear layer consists of a weighted sum of the input plus a constant, the following bounds are added to the model.

$$\mathsf{LB}(\boldsymbol{x}^{e}_{i,d}) = \mathbf{b}_d + \sum_{j \in J} \min(\mathsf{LB}(\boldsymbol{x}_{i,j})\mathbf{W}_{j,d}, \mathsf{UB}(\boldsymbol{x}_{i,j})\mathbf{W}_{j,d}) \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.45a}$$

$$\mathsf{UB}(\boldsymbol{x}^{e}_{i,d}) = \mathbf{b}_d + \sum_{j \in J} \max(\mathsf{LB}(\boldsymbol{x}_{i,j})\mathbf{W}_{j,d}, \mathsf{UB}(\boldsymbol{x}_{i,j})\mathbf{W}_{j,d}) \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.45b}$$

For each element in the input, the weighted value of its lower and upper bounds are compared. This is necessary to account for negative weights.

3.2.2. Residual

Given an inputs $\boldsymbol{x}_{i,j}$ and $\boldsymbol{x}'_{i,j}$ to the residual layer, the bounds on the input variable are propagated as follows:

$$\mathsf{LB}(\boldsymbol{x}^{+}_{i,d}) = \mathsf{LB}(\boldsymbol{x}_{i,d}) + \mathsf{LB}(\boldsymbol{x}'_{i,d}) \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.46a}$$

$$\mathsf{UB}(\boldsymbol{x}^{+}_{i,d}) = \mathsf{UB}(\boldsymbol{x}_{i,d}) + \mathsf{UB}(\boldsymbol{x}'_{i,d}) \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.46b}$$

3.2.3. Multi-head Attention

In the MHA layer, many bounds are calculated by propagation. To begin with, the bounds on the compatibility variable, $\boldsymbol{C}^{\alpha}$, are computed as follows. Let $\otimes_{\mathsf{min}}(a,b)$ and $\otimes_{\mathsf{max}}(a,b)$, refer to the following operations on two arbitrary variables (a) and (b),

$$\otimes_{\mathsf{min}}(a,b) = \min(\mathsf{LB}(a)\mathsf{UB}(b), \mathsf{LB}(a)\mathsf{LB}(b), \mathsf{UB}(a)\mathsf{LB}(b), \mathsf{UB}(a)\mathsf{UB}(b)) \tag{3.47a}$$

$$\otimes_{\mathsf{max}}(a,b) = \max(\mathsf{LB}(a)\mathsf{UB}(b), \mathsf{LB}(a)\mathsf{LB}(b), \mathsf{UB}(a)\mathsf{LB}(b), \mathsf{UB}(a)\mathsf{UB}(b)) \tag{3.47b}$$

The bounds on the compatibility are then derived as,

$$\mathsf{LB}(\boldsymbol{C}^{\alpha}_{h,i,i'}) = \frac{1}{\sqrt{|\mathcal{K}|}} \sum_{k \in \mathcal{K}} \otimes_{\mathsf{min}}(\boldsymbol{Q}^{\alpha}_{h,i,k}, \boldsymbol{K}^{\alpha}_{h,i',k}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.48a}$$

$$\mathsf{UB}(\boldsymbol{C}^{\alpha}_{h,i,i'}) = \frac{1}{\sqrt{|\mathcal{K}|}} \sum_{k \in \mathcal{K}} \otimes_{\mathsf{max}}(\boldsymbol{Q}^{\alpha}_{h,i,k}, \boldsymbol{K}^{\alpha}_{h,i',k}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.48b}$$

where $\mathsf{LB}(\boldsymbol{Q}^{\alpha}_{h,i,k}), \mathsf{UB}(\boldsymbol{Q}^{\alpha}_{h,i,k}), \mathsf{LB}(\boldsymbol{K}^{\alpha}_{h,i',k})$ and $\mathsf{UB}(\boldsymbol{K}^{\alpha}_{h,i',k})$ are found using Equations 3.45a – 3.45b. Since the exponential function is monotonically increasing,

$$\exp(\mathrm{LB}(C^{\alpha}_{h,i,i'})) \leq \exp(C^{\alpha}_{h,i,i'}) \leq \exp(\mathrm{UB}(C^{\alpha}_{h,i,i'}))$$

Therefore, bounds for the exponential of compatibility ($\boldsymbol{E}^{\alpha}$), and the sum of exponential compatibility ($\boldsymbol{S}^{\alpha}$) variables can be constructed as:

$$\mathsf{LB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) = \exp(\mathsf{LB}(\boldsymbol{C}^{\alpha}_{h,i,i'})) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.49a}$$

$$\mathsf{UB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) = \exp(\mathsf{UB}(\boldsymbol{C}^{\alpha}_{h,i,i'})) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.49b}$$

$$\mathsf{LB}(\boldsymbol{S}^{\alpha}_{h,i}) = \sum_{i' \in \mathcal{I}} \mathsf{LB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I} \tag{3.50a}$$

$$\mathsf{UB}(\boldsymbol{S}^{\alpha}_{h,i}) = \sum_{i' \in \mathcal{I}} \mathsf{UB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I} \tag{3.50b}$$

The limits of the attention score ($\boldsymbol{A}^{\alpha}$) are determined in the same way as the compatibility limits (Equations 3.51).

$$\mathsf{LB}(\boldsymbol{A}^{\alpha}_{h,i,k}) = \sum_{i' \in \mathcal{I}} \otimes_{\mathsf{min}}(\boldsymbol{\omega}^{\alpha}_{h,i,i'}, \boldsymbol{V}^{\alpha}_{h,i',k}) \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.51a}$$

$$\mathsf{UB}(\boldsymbol{A}^{\alpha}_{h,i,k}) = \sum_{i' \in \mathcal{I}} \otimes_{\mathsf{max}}(\boldsymbol{\omega}^{\alpha}_{h,i,i'}, \boldsymbol{V}^{\alpha}_{h,i',k}) \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I}, \forall k \in \mathcal{K} \tag{3.51b}$$

Finally, the values of the MHA output are restricted using the linear layer's bound equations (Equations 3.45a – 3.45b).

3.3. Strengthening Formulation

This section describes the methods used to strengthen the non-convex MINLP formulation of the TNN. The formulation is strengthened by adding additional cuts and tightening the variable bounds. Cuts

remove infeasible regions from the search space by injecting additional information about model variables. While bounds tightening imposes smaller domains on variables and provides tighter relaxations [36]. This reduces the search space and can lead to faster solve times.

Strengthening techniques are outlined per TNN layers in the following subsections. Some of these techniques make use of the propagated bounds defined in Section 3.2. No additional bounds or cuts are proposed for the FFN layer. This layer is implemented using existing approaches since it has already been widely investigated in the literature. The impact of the proposed strengthening techniques on the solving process is examined in Section 4.1.

3.3.1. Layer Normalisation

The purpose of this layer is to rescale input values to a normal distribution with a mean of β and a standard deviation of γ. The properties of this distribution indicate the percentage of values that lie within N standard deviations (std) of the mean (Figure 3.1). This information is used to set the bounds on the layer normalisation output ($\boldsymbol{x}^{\ell_1}_{i,d}$).

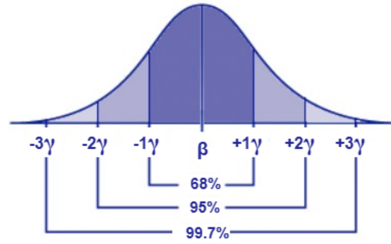


Figure 3.1: Normal distribution with mean, μ and standard deviation, σ. Image retrieved from [48].

Given that the bounds must not remove any feasible regions from the search space, the values of $\boldsymbol{x}^{\ell}_{i,d}$ are restricted to lie within 5 std of the mean. It is expected that 99.9994% of values lie in this range. The upper and lower bounds for the layer normalisation are stated in equations 3.52a and 3.52b.

$$\mathsf{LB}(\boldsymbol{x}^{\ell}_{i,d}) = \boldsymbol{\beta}^{\ell}_{d} - 5\boldsymbol{\gamma}^{\ell}_{d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.52a}$$

$$\mathsf{UB}(\boldsymbol{x}^{\ell}_{i,d}) = \boldsymbol{\beta}^{\ell}_{d} + 5\boldsymbol{\gamma}^{\ell}_{d} \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.52b}$$

Additionally, the lower bounds of nonnegative variables (e.g. variance, standard deviation) are set to 0.

$$\mathsf{LB}(\boldsymbol{\delta}^{\ell}_{i}), \mathsf{LB}(\boldsymbol{\sigma}^{\ell}_{i}), \mathsf{LB}(\boldsymbol{\Upsilon}^{\ell}_{i,d}) = 0 \qquad \forall i \in \mathcal{I}, \forall d \in \mathcal{D} \tag{3.53}$$

3.3.2. Multi-head Attention

The multi-head attention layer relies on matrix multiplications involving the Query, Key, and Value matrices. Specifically, the compatibility scores are derived from the Query and Key matrices and then used to compute the attention weight variables. The bounds on the compatibility variables depend on the bounds on the input to the MHA and the learned weight and bias parameters. However, for high-dimensional matrices or deep transformers with many encoder layers, the quality of the propagated bounds degrades. This is a result of compounding approximations across layers, and non-linear operations such as softmax and exponential functions.

One method of addressing this is to create an alternate formulation for the compatibility variable. The main idea is to use max-normalisation to introduce fixed bounds for the exponential compatibility and sum of exponential compatibility variables while scaling these values to smaller domains. These operations are performed without changing the behaviour of the TNN. Since solvers operate with finite precision, scaling the variables improves numerical stability of the exponential function and reduces the search space. This can make it easier for the solver to handle these non-linear exponential functions. The following paragraphs elaborate on how this is implemented.

First, a new variable $\boldsymbol{C'^{\alpha}}$ is introduced whereby,

$$\boldsymbol{C'^{\alpha}_{h,i,i'}} = \frac{1}{\sqrt{|K|}} \sum_{k \in \mathcal{K}} \boldsymbol{Q}^{\alpha}_{h,i,k} \boldsymbol{K}^{\alpha}_{h,i',k} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.54}$$

The compatibility $\boldsymbol{C}^{\alpha}$ is then calculated by subtracting the $\max_{i' \in \mathcal{I}}(\boldsymbol{C'^{\alpha}_{h,i,i'}})$ from $\boldsymbol{C'^{\alpha}_{h,i,i'}}$. This is implemented as,

$$\boldsymbol{C'^{\alpha}_{h,i,i'}} \leq \boldsymbol{C'^{\max}_{\alpha_{h,i}}} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.55}$$

$$\boldsymbol{C'^{\alpha}_{h,i,i'}} \geq \boldsymbol{C'^{\max}_{\alpha_{h,i}}} - (M^{C'} \cdot (1 - \boldsymbol{z}^{C'}_{h,i,i'})) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.56}$$

$$\sum_{i' \in \mathcal{I}} \boldsymbol{z}^{C'}_{h,i,i'} = 1 \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I} \tag{3.57}$$

$$\boldsymbol{C}^{\alpha}_{h,i,i'} = \boldsymbol{C'^{\alpha}_{h,i,i'}} - \boldsymbol{C'^{\max}_{\alpha_{h,i}}} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.58}$$

where $\boldsymbol{z}^{C'}_{h,i,i'}$ is a boolean variable that is 1 at the position corresponding to the maximum compatibility value and 0 otherwise. $M^{C'}$ is chosen as the upper bound of $\boldsymbol{C'^{\alpha}}$; since $\mathsf{UB}(\boldsymbol{C'^{\alpha}}) \geq \boldsymbol{C'^{\max}_{\alpha_{h,i}}} \geq \boldsymbol{C'^{\max}_{\alpha_{h,i}}} - \boldsymbol{C'^{\alpha}_{h,i,i'}}$.

The softmax function can then be applied to this max-normalized compatibility variable, resulting in what is known as the max-normalised softmax function [15]. The max-normalised softmax function is written as,

$$softmax(\boldsymbol{C'^{\alpha}_{h,i,i'}} - \quad \boldsymbol{C'^{\max}_{\alpha_{h,i}}}) = \frac{e^{\boldsymbol{C'^{\alpha}_{h,i,i'}} - \quad \boldsymbol{C'^{\max}_{\alpha_{h,i}}}}}{\sum_{p \in \mathcal{I}} e^{\boldsymbol{C'_{\alpha_{h,i,p}}} - \boldsymbol{C'^{\max}_{\alpha_{h,i}}}}} \tag{3.59}$$

From this equation it can be seen that when $\boldsymbol{C'^{\alpha}_{h,i,i'}}$ is equal $\boldsymbol{C'^{\max}_{\alpha_{h,i}}}$, the compatibility, $\boldsymbol{C'_{\alpha_{h,i,p}}}$, will be zero, leading to a numerator of e^0, which is equal to 1. For any compatibility value that is not the maximum, the numerator will be the exponential of a negative value, resulting in a non-negative value less than 1 (Figure 3.2). Thus the compatibility will maximally be 0 and the exponential of the compatibility will lie in the range $(0, 1]$. Similarly, a fixed lower bound for the denominator can be

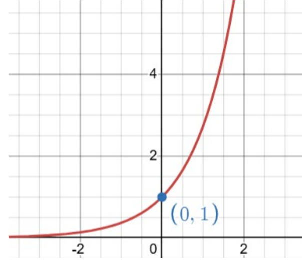


Figure 3.2: Graph of exponential function

established. In the denominator, the sum over the exponential of the compatibility is computed. At least one value in this sum will be 1, therefore the denominator must be greater than or equal to 1. Considering that the maximum value of $\boldsymbol{E}^{\alpha}$ is 1, then $\sum_{i' \in \mathcal{I}} UB(\boldsymbol{E}^{\alpha}_{h,i,i'}) = |\mathcal{I}|$. Thus $\boldsymbol{S}^{\alpha}$ is bounded above by the number of elements in the sequence $|\mathcal{I}|$.

This information can be used to update the bounds for the compatibility ($\boldsymbol{C'_{\alpha}}$), exponential of compatibility ($\boldsymbol{E}^{\alpha}$), and sum of exponential compatibility ($\boldsymbol{S}^{\alpha}$) variables. These adaptations are written in Equations 3.60a – 3.50b.

$$\mathsf{LB}(\boldsymbol{C}^{\alpha}_{h,i,i'}) = \mathsf{LB}(\boldsymbol{C'}^{\alpha}_{h,i,i'}) - \max_{r}(\boldsymbol{C'}^{\alpha}_{h,i,r}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.60a}$$

$$\mathsf{UB}(\boldsymbol{C}^{\alpha}_{h,i,i'}) = 0 \tag{3.60b}$$

$$\mathsf{LB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) = \exp(\mathsf{LB}(\boldsymbol{C}^{\alpha}_{h,i,i'})) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.60c}$$

$$\mathsf{UB}(\boldsymbol{E}^{\alpha}_{h,i,i'}) = 1 \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.60d}$$

$$\mathsf{LB}(\boldsymbol{S}^{\alpha}_{h,i}) = \max(1, \sum_{i' \in \mathcal{I}} LB(\boldsymbol{E}^{\alpha}_{h,i,i'}) \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I} \tag{3.60e}$$

$$\mathsf{UB}(\boldsymbol{S}^{\alpha}_{h,i}) = |\mathcal{I}| \tag{3.60f}$$

The introduction of the max-normalised softmax function tightens the bounds on $\boldsymbol{E}^{\alpha}$ and $\boldsymbol{S}^{\alpha}$ which would otherwise increase disproportionately with changes in the bounds of $\boldsymbol{C'}_{\alpha}$. This is expected to be effective at reducing the search space since $\boldsymbol{E}^{\alpha}$ and $\boldsymbol{S}^{\alpha}$ collectively account for $|\mathcal{H}||\mathcal{I}|^2 + |\mathcal{H}||\mathcal{I}|$ variables. Furthermore, it can help the formulation to scale to longer sequence lengths ($|\mathcal{I}|$) and larger head dimensions ($|\mathcal{H}|$).

It must be noted that while the input to the softmax function is changed, the output remains the same because the max value can be cancelled out of the calculation. This can be seen more clearly by rewriting Equation 3.59 to Equation 3.61.

$$softmax(\boldsymbol{C'}^{\alpha}_{h,i,i'} - \boldsymbol{C'}^{\max}_{\alpha_{h,i}}) = \frac{e^{-\boldsymbol{C'}^{\max}_{\alpha_{h,i}}} \cdot e^{\boldsymbol{C'}^{\alpha}_{h,i,i'}}}{e^{-\boldsymbol{C'}^{\max}_{\alpha_{h,i}}} \cdot \sum_{p \in \mathcal{I}} e^{\boldsymbol{C'}_{\alpha_{h,i,p}}}} \tag{3.61}$$

Therefore, this max-normalised softmax function effectively scales the inputs of the softmax function to a smaller domain with fewer values to explore without affecting subsequent calculations. Furthermore, this approach increases the robustness of the softmax function against under and overflow errors [15].

Further constraints are added to exclude infeasible regions from the search space, with regards to the compatibility and attention score variables. Auxiliary variables are introduced to present the products of $\boldsymbol{Q}^{\alpha} \cdot \boldsymbol{K}^{\alpha}$ and $\boldsymbol{\omega}^{\alpha} \cdot \boldsymbol{V}^{\alpha}$, and McCormick envelopes are applied to bound their values with concave and convex outer approximations [38]. Figure 3.3 shows an example of over-estimators created for a product term, $a \cdot b$. The blue and green solid lines in this figure represent the McCormick envelope, while the dashed lines show the interval arithmetic bounds. This comparison highlights how these outer approximations can be used to tighten bounds and improve search efficiency.

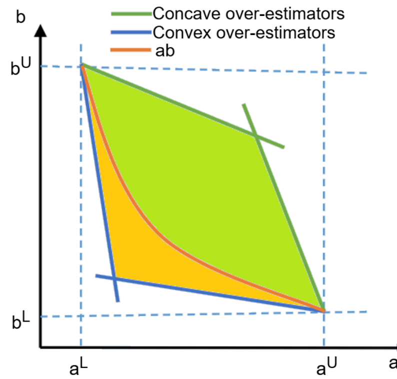


Figure 3.3: McCormick envelope for a bi-linear function $m(a, b) = ab$. Image retrieved from [40]

For a function $m(a,b) = ab$, the McCormick envelope is defined as,

$$m(a,b) \geq a^L \cdot b + a \cdot b^L - a^L \cdot b^L \tag{3.62a}$$
$$m(a,b) \geq a^U \cdot b + a \cdot b^U - a^U \cdot b^U \tag{3.62b}$$
$$m(a,b) \leq a^U \cdot b + a \cdot b^L - a^U \cdot b^L \tag{3.62c}$$
$$m(a,b) \leq a \cdot b^U + a^L \cdot b - a^L \cdot b^U \tag{3.62d}$$

These constraints put restrictions on the values that the auxiliary variables can take. This in turn acts to tighten the bounds on the compatibility and attention weight variables' feasible values.

The attention weight variable can also be constrained using properties of the softmax function. First, the values of the attention weight can be restricted to the interval $[0,1]$ (Equation 3.63a). Second, the sum of attention weights associated to each sequence element must equal to 1 (Equation 3.63b). These constraints stem from the fact that the softmax function's output is a probability distribution.

$$0 \leq \boldsymbol{\omega}^{\alpha}_{h,i,i'} \leq 1 \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.63a}$$
$$\sum_{i' \in \mathcal{I}} \boldsymbol{\omega}^{\alpha}_{h,i,i'} = 1 \qquad \forall h \in \mathcal{H}, \forall i \in \mathcal{I} \tag{3.63b}$$

While these constraints supply information about the feasible attention weight values, they do not provide tight bounds on the non-linear softmax function. To improve this, outer approximators can be created that tighten the bounds on the softmax function. Previous research demonstrated methods for deriving envelopes for similarly shaped non-linear functions. For example, Schweidtmann and Mitsos describe techniques for constructing a convex-o-concave envelope for the hyperbolic tan activation function [45] and Wilhelm, Wang, and Stuber for the sigmoid activation function [60]. This work applies similar strategies to create convex and concave approximators for the softmax function.

This strategy utilizes the fact that the softmax function can be decomposed into a convex and a concave region. A convex approximation can then be created which is equivalent to the softmax function in the convex region and smoothly transitions into a secant segment that under-approximated the concave region (see Figure 3.4). The same principle can be applied to create the concave over-approximation. The construction of these approximators is outlined in the following paragraphs.

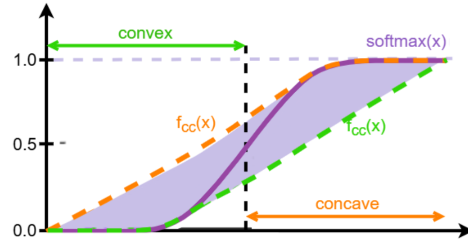


Figure 3.4: Over-approximators for softmax function (f_{cv}, f_{cc}) with bounded region shaded in lilac

Let $\boldsymbol{a}_{i,j}$ be the input to the softmax function with bounds $[\boldsymbol{a}_{\text{LB}}, \boldsymbol{a}_{\text{UB}}]$. The inflection point occurs when $e^{\boldsymbol{a}_{i,j}} = \sum_{j \neq p} e^{\boldsymbol{a}_{i,p}}$, thus when $\mathsf{softmax}(\boldsymbol{a}_{i,j}) = 0.5$. This can be derived by taking the second partial derivative of Equation 3.64; where by the sum term is considered as a constant.

$$\mathsf{softmax}(\boldsymbol{a}_{i,j}) = \frac{e^{\boldsymbol{a}_{i,j}}}{e^{\boldsymbol{a}_{i,j}} + \sum_{j \neq p} e^{\boldsymbol{a}_{i,p}}} \tag{3.64}$$

Therefore, if $\mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) \geq 0.5$, then $\mathsf{softmax}(\boldsymbol{a}_{i,j})$ lies in the concave region of the softmax function. Similarly, if $\mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) \leq 0.5$, then $\mathsf{softmax}(\boldsymbol{a}_{i,j})$ lies in the convex region. In other cases, the $\mathsf{softmax}(\boldsymbol{a}_{i,j})$ can be in the either region. As such the convex under-estimator (f_{cv}) and the concave

over-estimator (f_{cc}) functions are defined as,

$$f_{cv}(\boldsymbol{a}_{i,j}) = \begin{cases} \mathsf{softmax}(\boldsymbol{a}_{i,j}) & \text{, if } \mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) \leq 0.5 \\ \mathsf{sct}(\boldsymbol{a}_{i,j}) & \text{, if } \mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) \geq 0.5 \\ \mathsf{F}_{cv}(\boldsymbol{a}_{i,j}) & \text{, otherwise} \end{cases} \tag{3.65}$$

$$f_{cc}(\boldsymbol{a}_{i,j}) = \begin{cases} \mathsf{sct}(\boldsymbol{a}_{i,j}) & \text{, if } \mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) \leq 0.5 \\ \mathsf{softmax}(\boldsymbol{a}_{i,j}) & \text{, if } \mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) \geq 0.5 \\ \mathsf{F}_{cc}(\boldsymbol{a}_{i,j}) & \text{, otherwise} \end{cases} \tag{3.66}$$

where $\mathsf{sct}(\boldsymbol{a}_{i,j})$ is a secant function that constructs a line between the lower and upper bounds (Equation 3.67).

$$\mathsf{sct}(\boldsymbol{a}_{i,j}) = \frac{\mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) - \mathsf{softmax}(\boldsymbol{a}_{\text{LB}})}{\boldsymbol{a}_{\text{UB}} - \boldsymbol{a}_{\text{LB}}} \boldsymbol{a}_{i,j} + \frac{\boldsymbol{a}_{\text{UB}}\mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) - \boldsymbol{a}_{\text{LB}}\mathsf{softmax}(\boldsymbol{a}_{\text{UB}})}{\boldsymbol{a}_{\text{UB}} - \boldsymbol{a}_{\text{LB}}} \tag{3.67}$$

$\mathsf{F}_{cv}(\boldsymbol{a}_{i,j})$ defines a function that smoothly transitions from the concave part of the softmax function to a linear segment that under-approximates the concave region of the softmax function (Equation 3.68). A preliminary tie point variable ($\boldsymbol{a}_{i,j}^{cv*}$) is introduced such that the gradient of the softmax function at this point is equal to the gradient of a line between the tie point and $\boldsymbol{a}_{\text{UB}}$. The tie point variable, $\boldsymbol{a}_{i,j}^{cv}$ is then determined as $\max(\boldsymbol{a}_{i,j}^{cv*}, \boldsymbol{a}_{\text{LB}})$.

$$\mathsf{F}_{cv}(\boldsymbol{a}_{i,j}) = \begin{cases} \mathsf{softmax}(\boldsymbol{a}_{i,j}) & \text{, if } \boldsymbol{a}_{i,j} \leq \boldsymbol{a}_{i,j}^{cv} \\ \mathsf{sct}_{cv}(\boldsymbol{a}_{i,j}) & \text{, if } \boldsymbol{a}_{i,j} > \boldsymbol{a}_{i,j}^{cv} \end{cases} \tag{3.68}$$

$$\mathsf{sct}_{cv}(\boldsymbol{a}_{i,j}) = \mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) + \frac{\mathsf{softmax}(\boldsymbol{a}_{\text{UB}}) - \mathsf{softmax}(\boldsymbol{a}_{i,j}^{cv})}{\boldsymbol{a}_{\text{UB}} - \boldsymbol{a}_{i,j}^{cv}} \cdot (\boldsymbol{a}_{i,j} - \boldsymbol{a}_{\text{UB}}) \tag{3.69}$$

$\mathsf{F}_{cc}(\boldsymbol{a}_{i,j})$ is similarly defined as,

$$\mathsf{F}_{cc}(\boldsymbol{a}_{i,j}) = \begin{cases} \mathsf{softmax}(\boldsymbol{a}_{i,j}) & \text{, if } \boldsymbol{a}_{i,j} \geq \boldsymbol{a}_{i,j}^{cc} \\ \mathsf{sct}_{cc}(\boldsymbol{a}_{i,j}) & \text{, if } \boldsymbol{a}_{i,j} < \boldsymbol{a}_{i,j}^{cc} \end{cases} \tag{3.70}$$

$$\mathsf{sct}_{cc}(\boldsymbol{a}_{i,j}) = \mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) + \frac{\mathsf{softmax}(\boldsymbol{a}_{\text{LB}}) - \mathsf{softmax}(\boldsymbol{a}_{i,j}^{cc})}{\boldsymbol{a}_{\text{LB}} - \boldsymbol{a}_{i,j}^{cc}} \cdot (\boldsymbol{a}_{i,j} - \boldsymbol{a}_{\text{LB}}) \tag{3.71}$$

where $\boldsymbol{a}_{i,j}^{cc}$ is then determined as $\min(\boldsymbol{a}_{i,j}^{cc*}, \boldsymbol{a}_{\text{UB}})$ and $\boldsymbol{a}_{i,j}^{cc*}$ is a point on the softmax function that smoothly connects the concave region of the softmax function with a secant to the lower bound of $\boldsymbol{a}_{i,j}$. This envelope is applied to the restrict the values of the attention weight variable using Equations 3.72a and 3.72b.

$$\boldsymbol{\omega}_{h,i,i'}^{\alpha} \geq f_{cv}(\boldsymbol{C}_{h,i,i'}^{\alpha}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.72a}$$

$$\boldsymbol{\omega}_{h,i,i'}^{\alpha} \leq f_{cc}(\boldsymbol{C}_{h,i,i'}^{\alpha}) \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.72b}$$

3.4. Overview of Bounds & Cuts

The previous sections propose a number of bounds and cuts for each of the TNN layers. These proposed enhancements can be classified based on whether they are calculated from propagated bounds or not. The bounds and cuts calculated without propagated bound values are hereafter referred to as "independent" while the others are referred to as "dependent". Table 3.2 provides an overview of the independent (I) and dependent (D) enhancements for each TNN layer. It must be noted that there are conditional constraints that can lead to either independent or dependent enhancements. In these cases, both the I and D columns in Table 3.2 are checked.

In Section 4.1 computational experiments are carried out to evaluate the effectiveness of the proposed enhancements. Given the number of bounds and cuts, they are arranged into 6 groups before performing experiments. The six groups are created as follows:

#	Layer	Enhancement Descr.	Equations	I	D
1	Linear	bounds on layer output	3.45	-	x
2	Layer Norm	bounds on layer output	3.52	x	-
3	Layer Norm	lower bounds on non-negative variables	3.53	x	-
4	Residual	bounds on layer output	3.46	-	x
5	MHA	bounds on compatibility variable	3.48	-	x
6	MHA	bounds on exp(compatibility) variable	3.49	-	x
7	MHA	bounds on sum of exp(compatibility) variable	3.50	-	x
8	MHA	max-norm. softmax function	3.54–3.58	x	-
9	MHA	compatibility bounds with max-norm. softmax function	3.60	x	x
10	MHA	bounds on attention score variable	3.51	-	x
11	MHA	McCormick envelope for QK product in compatibility	3.62	-	x
12	MHA	McCormick envelope for ωV product in attention score	3.62	-	x
13	MHA	bounds on softmax output	3.63a	x	-
14	MHA	softmax sum constraint	3.63b	x	-
15	MHA	softmax convex and concave approximators	3.65 – 3.72	x	x
16	MHA	output of MHA layer	3.45	-	x

Table 3.2: Summary of dependent and independent strengthening methods used in each layer

i. Standard (STD): dependent bounds in linear and residual layers, lower bounds of non-negative variables in LN layer and max-normalized softmax constraints in MHA layer (1,3,4,8,9 in Table 3.2)

ii. Layer normalisation independent (LN_I): indepedent bounds in layer normalisation layer (2 in Table 3.2)

iii. Multi-head attention independent (MHA_I): independent bounds and cuts in the MHA layer (13,14 in Table 3.2)

iv. Multi-head attention dependent (MHA_D): dependent bounds and cuts in the MHA layer with exception of bi-linear and softmax envelope constraints and max normalised softmax constraints (5,6,7,10,16 in Table 3.2)

v. Multi-head attention McCormick (MHA_MC): bi-linear envelope constraints (11,12 in Table 3.2)

vi. Multi-head attention softmax enclosed (MHA_SE): constraints to construct concave/convex outer approximations for softmax function (15 in Table 3.2)

3.5. Implementation

The original TNN architecture of Vaswani et al. [57] is commonly used to design TNNs; however, there is no standardised architecture. In many cases, new architectures are preferred to tailor the model to a given application. Some choose to arrange the layers differently, while others omit specific layers [22][59]. For this reason, it is important to have a modular TNN implementation that can adapt to many different TNN architectures. The formulated transformer was created with this in mind. As such, each TNN layer is implemented as a `python` function which can be connected to any other layer as long as the input and output dimensions of the connected layers match.

Moreover, there are inconsistencies between the TNN implementations for various machine learning (ML) libraries. The encoder and decoder blocks are constructed in different arrangements with different default parameters. Furthermore, learned parameters are stored in different arrangements without clear documentation of the format of the parameters. This is likely because the learned parameters are typically not used outside of the given ML library. Nevertheless, this can make the parsing of TNN parameters tedious. Another challenge in parsing the trained TNN is determining the layers used and their ordering. Some layers like the residual, pooling and activation function layers are not instantiated as layer components of the trained TNN, but are rather implemented as functions in the forward pass through the TNN. As a result, TNNs architecture list of layer modules can be misleading because it will not mention these layers. For `Pytorch`'s TNN module, the model summary does not print the layers of the multi-head attention block in the order that they occur. Consequently, the documentation for

the various layers and TNN functions must be closely examined to ensure that the formulated TNN is constructed in the same way as the trained TNN.

The formulated TNN is implemented to offer solutions to the challenge of using the trained TNN models outside of their intended environments. First of all, three parsing functions are created to build TNN formulations for models created using the `Keras`, `Pytorch` and `HuggingFace` ML libraries. These parsing functions extract information from files containing trained TNNs, including where necessary information about the format of learned parameters, ordering of layers and any omitted layers. When updates are made to the TNN modules in these libraries, the parsing function will likely also need to be updated. For the `Pytorch` and `HuggingFace` libraries an additional function is included to build a TNN formulation based on the default TNN configuration of these libraries. A more flexible approach is taken for this function, whereby a forward pass through the trained TNN is carried out to identify the correct ordering of the TNN layers. In this case placement of the omitted residual layers can be inferred from the position of the normalisation layers.

The implementation also supports multiple mathematical modelling frameworks. This is necessary to have access to an assortment of solving algorithms and FFN implementations (see [37]). The implementation is developed to be compatible with `Pyomo` framework, which allows for a wide variety of open-source and commercial solvers and the `Gurobipy` framework which uses the `Gurobi` solver. Eventhough the `Gurobi` solver can be invoked in both frameworks, the `Pyomo` interface to `Gurobi` currently does not support the modelling of non-linear functions [42]. The lastest two `Gurobi` releases, `Gurobi 11` and `12`, delivered updates for performing global optimisation on MINLPs, like the proposed TNN formulation.

Strictly speaking, the formulation is constructed as a `Pyomo` mathematical model. For the `Gurobipy` framework, a python function is created to convert the `Pyomo` mathematical model to `Gurobipy`. The differences in the chosen optimisation framework gives some small differences in the overall formulation. First, different libraries are used for constructing the Big-M FFN formulation. For `Pyomo`, the `OMLT` package is used and for `Gurobipy`, the `GurobiML` package is used. Second, there are different methods used for implementing the exponential function. Currently, the exponential function in the `Pyomo` framework does not support inputs greater than 2. In this case, a static power series approximation is used (Equation 3.73). For the `Gurobi` framework, a more accurate approximation is applied via `Gurobi`'s exponential function (equation 3.7). `Gurobi`'s exponential function instantiates a dynamic piece-wise linear approximation of the exponential function which is computed in the vicinity of the current relaxation solution during solving (see [20]).

$$\boldsymbol{E}^{\alpha}_{h,i,i'} = \sum_{\psi=0}^{\Psi} \frac{{\boldsymbol{C}^{\alpha}_{h,i,i'}}^{\psi}}{\psi!} \qquad \forall h \in \mathcal{H}, \forall i, i' \in \mathcal{I} \tag{3.73}$$

4

Case Studies

This chapter discusses a series of case studies aimed at evaluating the performance of the proposed TNN formulation inside an optimisation problem. Three optimisation problems are considered to explore the performance the formulation on transformers of various sizes in a variety of applications.

First, an optimal trajectory problem showcases a simple problem containing a small encoder-only TNN. Next, a verification problem with a larger encoder-only transformer is examined. Finally, a reactor optimisation problem with an encoder-decoder model is studied.

In the subsequent experiments, the performance of the formulated TNN is evaluated based on the following metrics:

- **Speed**: This is measured as the time it takes to converge to the optimal solution (*Time*).
- **Optimality**: This is assessed using the objective value at the end of the run and the difference between the lower and upper bound of the objective value (optimality gap or *Gap*).
- **Computational complexity**: The average amount of search iteration per second during the solve is used as a proxy to indicate the formulation's complexity (*Iter./s*).
- **Ability to find feasible solutions**: This is assessed using the number of solutions found within the time limit (*# Solutions*) and the speed at which the first feasible solution is found.

All experiments are run on an Intel Core i7-10750H CPU @ 2.60 GHz with 6 physical cores and 12 threads. Furhtermore, the `Gurobi 11.0.1` solver is used with fixed for each experiment which creates a deterministic environment.[1] More details about the experimental set ups for each case study can be found in the following subsections and results in the form of solve logs can be retrieved from `https://zenodo.org/records/14261874`.

4.1. Optimal Trajectory Problem

4.1.1. Problem Definition

This is a simple optimal trajectory problem which is adapted from an example in the Optimal Control Estimation lecture notes by Yang [62]. This illustrative problem is defined as follows:

> "Consider a canon placed at the origin of a 2-dimensional plane, which we can fire with control over the initial velocities v_x, v_y; representing the decomposed velocities with respect to displacement (x) and height (y). What is the best trajectory that can be achieved, so that the cannonball follows a target path, defined by the coordinates $(x_{\text{target}}(t), y_{\text{target}}(t))$ at each time step t, as closely as possible?"

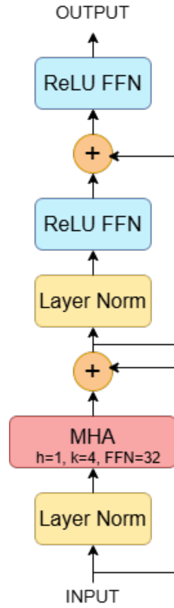
This optimisation problem can be mathematically formulated as shown in equation 4.1. Here the displacement, $x(t)$, and height, $y(t)$, over time is given by the projectile motion equations and the objective function is set to minimize the absolute difference between the cannonball and some fixed

[1] `Gurobi 11.0.1` was the most recent version at the time of this work

target positions. In this case, the target trajectory is defined by projectile motion equations with some random noise added to the y_{target} trajectory.

$$
\begin{aligned}
\min_{v_x, v_y} \quad & \sum_{t=0}^{T} (x(t) - x_{\text{target}}(t))^2 + (y(t) - y_{\text{target}}(t))^2 && && (4.1a) \\
\text{s.t.} \quad & x(t) = v_x t && \forall t \in \{0, ..., T\} && (4.1b) \\
& y(t) = (v_y t) - (0.5gt)^2 && \forall t \in \{0, ..., T\} && (4.1c) \\
& x_{\text{target}}(t) = v_{x_{\text{target}}} t && \forall t \in \{0, ..., T\} && (4.1d) \\
& y_{\text{target}}(t) = (v_{y_{\text{target}}} t) - (0.5gt)^2 + noise(t) && \forall t \in \{0, ..., T\} && (4.1e) \\
& g = 9.81 && && (4.1f) \\
& v_x, v_y \geq 0 && && (4.1g) \\
& x(0), y(0) = 0 && && (4.1h) \\
& noise(t) \in \mathbb{R} && \forall t \in \{0, ..., T\} && (4.1i)
\end{aligned}
$$

The problem definition is then modified to include an encoder-only TNN that models the projectile motion of the cannonball. A TNN is trained using the `Keras` library with an architecture as shown in 4.1. A dataset containing the x and y trajectories over time for 300 different initial velocities is generated and sampled to create 7200 training instances of sequence length 3. The TNN is trained to predict the x and y positions of the cannonball at the third time step, given the x and y positions at the first two time steps. The model has a mean absolute percentage error of 8.54% on unseen data. This poor performance is expected to be a result of the small size of the training dataset since TNNs tend to require large amounts of training data [32][35]. Another factor impacting predictive performance could be the lack of a positional encoding layer [2].



(a) TNN architecture. MHA has a head size (k) = 4, number of heads (h)= 1, FFN dimension = 32

Layers	Count
Input	1
Linear	0
Residual	2
Layer Norm	2
MHA	1
FFN	2

(b) Summary of components

Figure 4.1: Description of TNN for the optimal trajectory problem

When the TNN is added to the illustrative problem it removes the physical projectile constraints from the model and instead relies on the TNN's predictions. It must be noted that in this case, it also removes the control variables v_x and v_y from the problem, which should not occur in a decision problem. However,

this problem was deemed as suitable for the purpose of testing the correctness of the formulation. Nevertheless, the TNN finds the sequence of trajectory positions x and y, therefore the velocity is inherently modelled by the change in these variables and the control variables v_x and v_y can be derived from the solution.

The illustrative problem can be rewritten as,

$$\min_{\boldsymbol{x}_t, \boldsymbol{y}_t} \quad \sum_{t=0}^{T} (\boldsymbol{x}_t - x_{\text{target}}(t))^2 + (\boldsymbol{y}_t - y_{\text{target}}(t))^2 \tag{4.2a}$$

$$\text{s.t.} \quad \boldsymbol{x}_T, \boldsymbol{y}_T = TNN(\boldsymbol{x}_{t_1}, \boldsymbol{x}_{t_2}, \boldsymbol{y}_{t_1}, \boldsymbol{y}_{t_2}) \tag{4.2b}$$

$$x_{\text{target}}(t) = v_{x_{\text{target}}} t \qquad \forall t \in \{t_1, t_2, T\} \tag{4.2c}$$

$$y_{\text{target}}(t) = (v_{y_{\text{target}}} t) - (0.5gt)^2 + noise(t) \qquad \forall t \in \{t_1, t_2, T\} \tag{4.2d}$$

$$g = 9.81 \tag{4.2e}$$

$$\boldsymbol{x}_{t_1}, \boldsymbol{y}_{t_1} = 0 \tag{4.2f}$$

$$\boldsymbol{x}_{t_2}, \boldsymbol{x}_T, \boldsymbol{y}_{t_2}, \boldsymbol{y}_T \in \mathbb{R} \tag{4.2g}$$

$$noise(t) \in \mathbb{R} \qquad \forall t \in \{t_1, t_2, T\} \tag{4.2h}$$

$$T \geq t_2 \geq t_1 = 0 \tag{4.2i}$$

whereby the height (y) and distance (x) positions at the third time step, T, is a function of the previous y and x positions.

4.1.2. Experimental Set Up

Each experiment is run with a fixed time-limit. The solving process of the experiments is recorded using `Gurobi`'s logging tool. Results are then extracted from the log files to obtain information about the solving process.

In the first experiment, the standard MINLP formulation without any strengthening techniques is considered. In the second experiment, the computational correctness of the formulated transformer is verified by comparing the formulated TNN's and trained TNN's predictive results for the same input value. In the last set of experiments examine the impact of the proposed bounds and cuts on solve performance. Six different combinations of the proposed bounds and cuts are considered and the configuration with no additional enhancements is included for reference. These set up configurations are defined in Table 4.1.

Config	STD	LN_I	MHA_I	MHA_D	MHA_MC
`All`	1	1	1	1	1
`No_MC`	1	1	1	1	0
`I_only`	1	1	1	0	0
`LN_only`	1	1	0	0	0
`Prop_MC`	1	1	0	1	1
`LN_prop`	1	1	0	1	0
`None`	1	0	0	0	0

Table 4.1: An overview of the additional enhancements that are activated for each configuration; where 1 indicates that a given set of enhancements is activated and 0 indicates that it is deactivated. LN – refers to enhancements that are added to the layer normalization layer, MHA – refers to the multi-head attention layer, I – refers to independent enhancements that do not include propagated bounds, D – refers to dependent enhancements, MC – refers to the addition of bi-linear McCormick envelopes and, SE – refers to the addition of softmax envelopes. A more detailed description of these categories can be found in subsection 3.4.

Each of the configurations affects the number of constraints that constitute the formulated transformer. The number of constraints that they add to the model is shown in Table 4.2. The baseline setup with `None` has 248 constraints. Note well that these values do not account for the changes to the individual bounds on variables.

Config	Constraints
All	50
No_MC	2
I_only	2
LN_only	0
Prop_MC	16
LN_prop	0

Table 4.2: Caption

4.1.3. Numerical Results

TNN performance without enhancements

In this experiment, the TNN is given the initial positions $(x(0), y(0))$ as inputs and must solve for the values of the x and y positions at times t_2 and T. No additional bounds or cuts (enhancements) are added to the model. The results indicate that it is not possible to solve the optimal trajectory problem with the TNN to optimality within a time limit of 3 hours. Moreover, no feasible solutions are found before the time limit is reached despite 5.08×10^7 nodes having been explored.

The results show that even for this small TNN and simple illustrative problem, finding a feasible solution is not possible within the time limit. This highlights the additional computation overhead that is incurred due to the formulated transformer. Moreover, it emphasises the need to reduce the size of the feasible region that the solver must explore as the model is too complex to benefit from the advanced techniques applied by the `Gurobi` solver. The experiments in the following subsections investigate how performance can be improved by adding new bounds and cuts to the problem that are informed by the properties of the functions used in the TNN.

TNN Validation

In this experiment, the inputs to the TNN are fixed so that the solution to the illustrative problem only needs to be evaluated. First, the problem is run without any additional enhancements. The illustrative problem is not solved within 6 hours and no feasible solutions are found. Time out is reached with a best-bound value of 3.10×10^{-4} after exploring a larger number of nodes (9.37×10^7), revealing the difficulty the solver faces in finding a feasible solution in this large search space.

Next, all of the proposed additional bounds and cuts are added to the model. The result of solving the problem with fixed TNN inputs is shown in Figure 4.2 as the dashed lines. The expected solution based on the result of solving the original problem without the TNN is also included for reference (shown as solid lines).

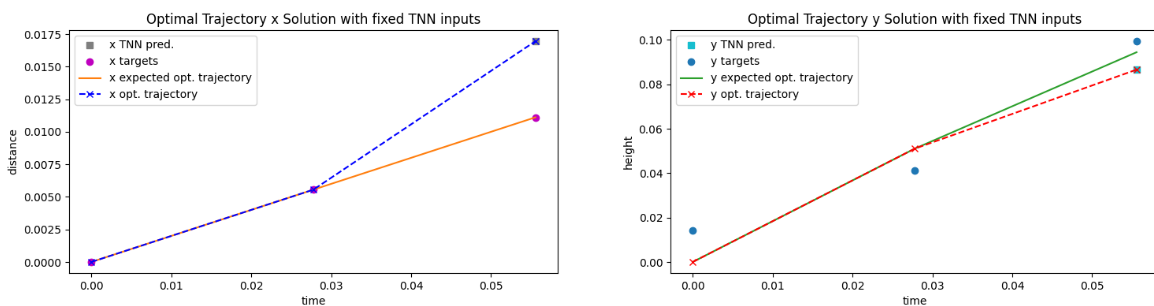


Figure 4.2: Optimal toy trajectory with fixed TNN inputs

The first key result is that it is possible to optimize over the formulated TNN when enhancements are added. Furthermore, it can be seen that the output of the formulated TNN is the same as the trained TNN output (shown as squares) when they are given the same input values. This proves that the formulated TNN calculates its output prediction in the same manner as the trained TNN and therefore validates the correctness of the formulation.

When the solution to the original problem (solid lines) and the solution to the problem with the TNN (dashed lines) are compared, it can be seen that the original problem finds a better solution. Solving the original toy problem results in an objective value of 3.30×10^{-4}, while solving the toy problem with the TNN results in an objective value of 5.06×10^{-4}. This difference is likely due to the poor predictive capabilities of the trained transformer because the formulated TNN's predictions reflect those of the trained model. With poor performing ML models, the true optimal solution may not be found simply because the ML model is incapable of predicting these values.

TNN performance with enhancements

This section describes the results of solving the illustrative problem with the TNN when different additional bounds and cuts are included in the TNN formulation. The results for 6 configurations of enhancements are considered. A 7th configuration with no enhancements is included for reference. The enhancement configuration summary is shown in Table 4.1. The MHA_SE enhancements are not considered in these experiments since preliminary results showed that they led to unsolvable models. This is likely due to the high computational overhead of this approach which can add up to ($2 \times$ number of heads $\times$ sequence length^2) tie-point variables, 10 slack variables and ($35 \times$ number of heads $\times$ sequence length^2) additional constraints.

The solve results for each configuration are shown in Table 4.3. At first glance, it can be seen that in contrast to the experiment with fixed transformer inputs, the objective value found in this case (2.06×10^{-4}) is lower than the optimal objective value when solving the original toy problem (3.30×10^{-4}). This is expected to occur because the trained TNN does not adequately model the physical principles of what constitutes a feasible projectile motion.

	Time	Iter./s	# Nodes	# Solutions	Gap	Objective
`All`	46 mins	9.03×10^4	9.16×10^6	**79**	3.73×10^{-6}	2.06×10^{-4}
`No_MC`	60 mins	8.69×10^4	1.16×10^7	43	3.97×10^{-1}	2.06×10^{-4}
`I_only`	12 mins	8.82×10^4	2.26×10^6	69	3.96×10^{-6}	2.06×10^{-4}
`LN_only`	13 mins	7.34×10^4	2.36×10^6	22	**0**	2.06×10^{-4}
`Prop_MC`	23 mins	$\mathbf{1.08 \times 10^5}$	4.35×10^6	55	8.69×10^{-5}	2.06×10^{-4}
`LN_prop`	**6 mins**	8.13×10^4	$\mathbf{1.09 \times 10^6}$	51	6.45×10^{-5}	2.06×10^{-4}
`None`	60 mins	1.97×10^5	5.08×10^7	0	inf	2.01×10^{-4}

Table 4.3: Summary solve results for various configurations of enhancements with a time limit of 1 hour. Here time refers to wall clock time.

When making an overall comparison between the enhanced configurations and the configuration with no enhancements (`None`), it can be seen that the enhancements allow for configurations with equally many (or more) constraints (`LN_prop, LN_only`) to be solved to global optimality in a fraction of the time. This shows that the benefit of adding the enhancements outweighs the drawback of having more constraints (or a more complex model). Furthermore, enhanced configurations explore fewer nodes indicating that the enhancements help to improve the search efficiency. The `LN_prop` configuration explores the least nodes (1.09×10^{-6}) before finding the optimal solution, showing that these enhancements best assist the `Gurobi` MINLP solver in exploring the search space. However, this configuration is one of the most computationally complex (8.13×10^4 iter./s). `LN_prop` takes about 3 times longer than the least complex (`None`) configuration to complete an iteration. Together, these results show that while the enhancements increase the toy problem complexity, they allow the solving algorithm to make better choices while exploring the search space, thus finding solutions faster.

Looking at the solution time in Table 4.3, the results show clear differences for the various configurations. The solve times for the enhanced formulations vary from 6 mins in the best case (`LN_prop`), to reaching time-out (`No_MC`). The configurations `All, No_MC,` and `Prop_MC` give the slowest runtimes. This is expected to arise from the added complexity of imposing more constraints on the same variables. For example, one of the `LN_I` enhancements adds bounds to the attention weight ($\boldsymbol{\alpha}$) so that it must lie between $[0, 1]$. While constraints are added to restrict the values of $\boldsymbol{\alpha} \cdot \boldsymbol{V}$ in the `MHA_MC` enhancements, and bounds are added to the compatibility ($\boldsymbol{Q} \cdot \boldsymbol{K}$), which is used to calculate $\boldsymbol{\alpha}$ (see Equation 2.10), in the `MHA_D` enhancements. On the other hand, the fastest solving configurations `I_only, LN_only`

and LN_prop each add new information about the variables in the layer normalization and multi-head attention layers without adding restrictions to the same variables. These results align with Kosheleva, Ceberio, and Kreinovich's [30] discussion about the counter-intuitive relationship between problem complexity and the difficulty of solving the problem. The authors note that in some cases the best way to solve a complex problem is to make it more complex by adding additional constraints. More specifically, they mention that when constraints are included in a problem that adds new information, the benefit of these constraints generally outweighs their added complexity.

Despite the differences in run time, all enhanced configurations converge to the optimal objective value and find multiple feasible solutions. Only No_MC does not find the optimal solution within the 1-hour time limit. However, it has a relatively small optimality gap, which is a positive indication that the optimal solution could be found with more resources. To better understand the convergence properties of the configurations, the changes in the incumbent and best bounds over time are examined (Figure 4.3).

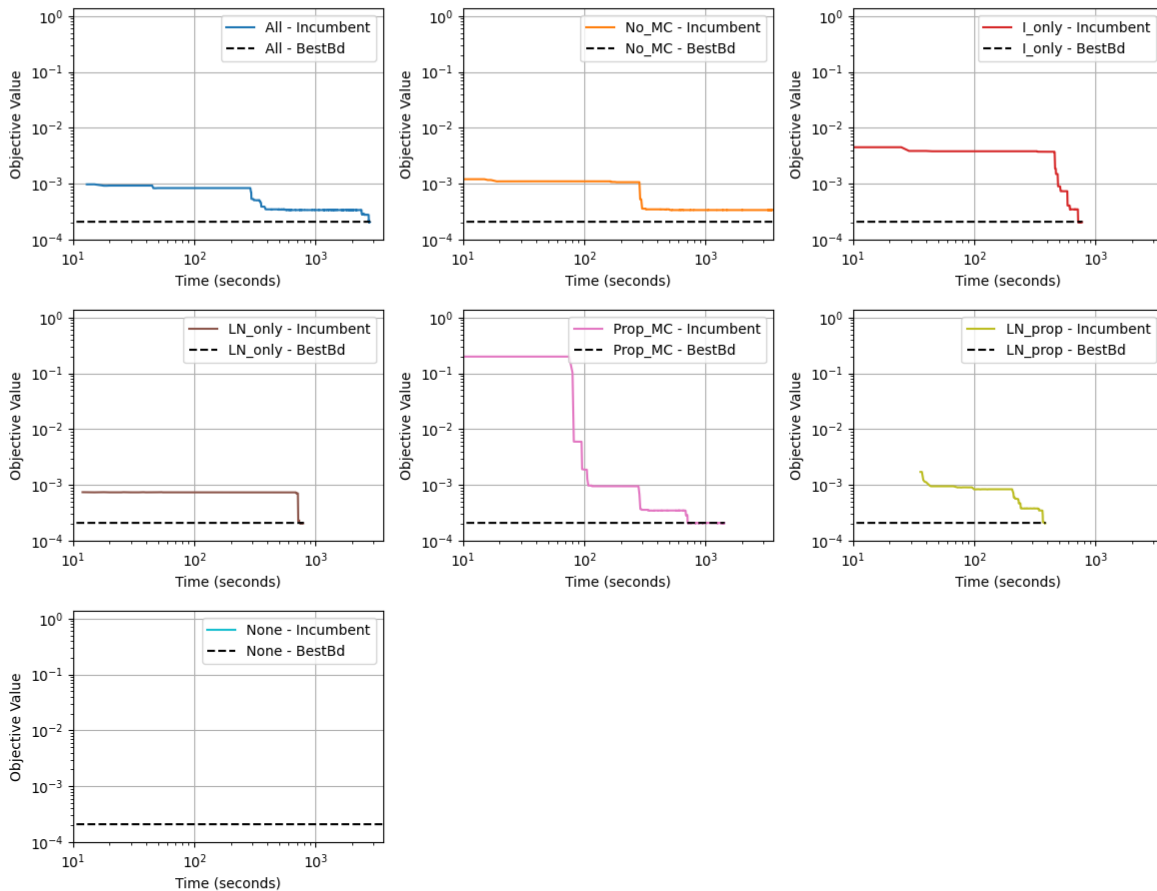


Figure 4.3: Convergence of the incumbent and best bound over time for each configuration

Regardless of the configuration, the best bound remains constant over time and near to the optimal solution. This indicates that the solver works well to tighten the lower bound of the objective value. For the enhanced configuration, the best bound is improved to be equal to the optimal solution, thus there is no room for improvement on these bounds.

The incumbent shows different initial gaps and convergence trends for the various configurations. For example, configurations No_MC, I_only, and Prop_MC all find feasible solutions almost instantaneously, but the quality of these solutions varies significantly. Prop_MC begins with a large optimality gap which can be expected, as compared to the No_MC and I_only configurations, because Prop_MC does not include independent bounds and cuts in the MHA layer. The independent enhancements constrain variables to fixed regions which helps to reduce the growth of the propagated bounds to large and uninformative values. Instead propagated bounds are applied in the MHA layer which are complimented

by McCormick envelopes on the Query-Key and attention weight-Value products. The trace shows that the gap progresses in large steps towards the optimum, and is one of the fastest configurations to significantly close the gap. This indicates that the solver frequently finds a solution that is significantly better than the previous one. When an improved feasible solution is found, the solver can eliminate parts of the search space that lead to solutions less than this value. This tightens variable bounds and leads to more informative propagated bounds and constructed envelopes, which are computationally inexpensive. This provides a model with relatively fast iterations (1.08×10^5iter./s) and bounds that effectively improve as better feasible solutions are found. This can explain how `Prop_MC` converges faster than other configurations which have tighter initial gaps.

The trace for `I_only` has similar characteristics as `Prop_MC` but shows better performance as it starts with a smaller optimality gap. This is expected to be a result of the fixed valued constraints on the layer normalisation and attention weight values. These constraints include that the attention weights but be in $[0, 1]$, and that the attention weights associated with a given element in the sequence must sum to 1. Even though these constraints do not provide the convex hull for the attention weight, they provide fixed domains that do not grow, which can be useful for scaling to larger transformer models. This configuration initially struggles to find improved feasible solutions (long plateau of the incumbent in Figure 4.3), then quickly converges. This is likely due the fact that these enhancements only improve the search space at the start of the solving process and provide little additional information to aid the solver in later stages. Nevertheless, this configuration has a good trade-off between the problem being quickly solved and the optimal solution being quickly found. Thus these fixed constraints create a sufficiently small search space that solver can efficiently traverse using its built-in techniques.

The `LN_only` configuration uses a subset of the enhancements applied in `I_only`. `LN_only` only constrains the layer normalisation output values to be in a fixed range based on the properties of the normal distribution. This configuration results in the smallest initial gap. This indicates that when the independent MHA costraints are added in `I_only`, these weaken the formulation leading to a larger search space and slower convergence. Therefore, these proposed MHA_I enhancements are not beneficial for solving this problem.

Instead when dependent MHA and independent layer normalisation enhancements are combined in `LN_prop`, the fastest convergence is achieved. However, this configuration is also the slowest to find an initial feasible solution. These features in `LN_prop`'s convergence are attributed over approximated propagated bounds on the MHA variables which can degrade solving in early stages but aid it as the search space reduces. As such `LN_prop` is able to find better feasible solutions earlier than `LN_only`.

`No_MC` is a combination of the `I_only` and `LN_prop` configurations. The `No_MC` is one of the fastest to find a feasible solution faster than most among the set of configurations. Only `LN_only` and `All` result in a better initial feasible solution but it takes a longer time. Here it can be seen how the combination of independent and dependent constraints on MHA variables leads to a smaller initial gap than in `I_only`. This combines the benefits of a smaller initial search space due to fixed constraints, and bounds that gradually improve during the search due to the propagated bounds. However, this combination of bounds and cuts also degrades convergence (long plateau of the incumbent at the end of solve in Figure 4.3). This effect is likely due to the `No_MC` configuration likely over-constraining the MHA variables, as mentioned earlier. This can make the non-convex search space with multiple local optima excessively complex and difficult to explore.

The `All` extends the `No_MC` to include McCormick envelopes on products in the MHA layer. McCormick envelopes are added to the product of the Query and Key matrices as well as the attention weight and Value matrices. These outer approximations construct tighter bounds that remove infeasible regions from the bounding box created by the propagated bounds (see Figure 3.3). The plots in Figure 4.3 show how adding these envelopes leads to faster convergence while exploring fewer nodes (9.16×10^6 nodes) than `No_MC` (1.16×10^7 nodes). So while some of the MHA variables are still over-constrained, and the search space is difficult to explore, there are fewer candidate solutions to explore in this case. As a result, convergence is improved when compared to `No_MC`. Furthermore, these enhancements appear to help the solver with finding improved feasible solutions since `No_MC` and `All` run for similar durations but `All` finds almost twice as many solutions. This ability to frequently find better solutions can help to quickly prune the search tree and improve solve efficiency.

Each of the configurations shows long periods of time when the incumbent does not improve. This indicates that the solver has difficulty in finding improved feasible solutions and that the formulation would benefit from further refinements to help close the gap faster. Another method to improve the solve could be to tune the solver's parameters. For example, `Gurobi` provides a MIPFocus parameter which makes the solver focus on finding feasible solutions quickly. Alternatively, warm-starting could be used to provide a good initial solution that is in the neighbourhood to the optimal solution. This could be helpful for these configurations because at some point in their solving process, a new feasible solution is found which improves the incumbent and then the bounds converge rather quickly.

4.2. Verification Problem

4.2.1. Problem Definition

Consider a verification problem in which the robustness of an image classification TNN model against perturbations to the input image is assessed. The input image, $\overline{\boldsymbol{x}}$, is sparsely perturbed to an image, $\boldsymbol{x}$, such that $||\boldsymbol{x} - \overline{\boldsymbol{x}}||_{\ell 1} \leq \epsilon$, where ϵ is a perturbation limit. Given the correct image classification label, l_{tgt} and an adversarial label, l_{adv}, the optimisation objective is to maximize the difference between the predicted probability of these labels ($\boldsymbol{y}_{l_{\text{tgt}}}$ and $\boldsymbol{y}_{l_{\text{adv}}}$). This gives an indication of the degree to which the classifier is deceived. Thus the optimisation problem is defined as,

$$\max_{\boldsymbol{x},\boldsymbol{\delta}} \quad (\boldsymbol{y}_{l_{\text{adv}}} - \quad \boldsymbol{y}_{l_{\text{tgt}}}) \tag{4.3}$$

$$\text{s.t.} \quad \boldsymbol{x}_p \leq \min(1, \quad \overline{\boldsymbol{x}}_p - \epsilon) \qquad \forall p \in P \tag{4.4}$$

$$\boldsymbol{x}_p \geq \max(0, \quad \overline{\boldsymbol{x}}_p + \epsilon) \qquad \forall p \in P \tag{4.5}$$

$$\boldsymbol{\delta}_p \geq \boldsymbol{x}_p - \overline{\boldsymbol{x}}_p \qquad \forall p \in P \tag{4.6}$$

$$\boldsymbol{\delta}_p \geq \overline{\boldsymbol{x}}_p - \boldsymbol{x}_p \qquad \forall p \in P \tag{4.7}$$

$$\sum_{p \in P} \boldsymbol{\delta}_p \leq \epsilon \tag{4.8}$$

$$\boldsymbol{y}_{l_{\text{adv}}}, \quad \boldsymbol{y}_{l_{\text{tgt}}} = \mathsf{TNN}(\boldsymbol{x}) \tag{4.9}$$

where P is the set of all pixels in the image and $\boldsymbol{\delta}_p$ is the perturbation at a pixel $p \in P$.

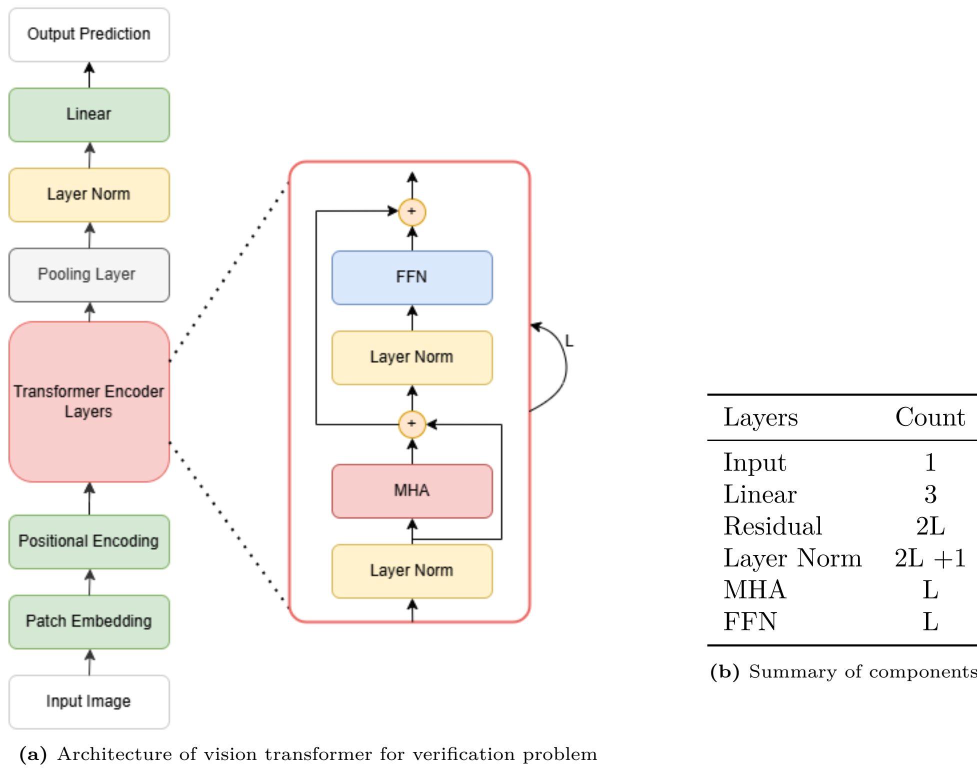
The TNN was trained to perform image classification of hand-written numbers (MNIST data [33]) using the `Pytorch` library. The TNN uses an encoder-only architecture as shown in Figure 4.4. The input images have a size of 4×4 pixels. The patch embedding layer reorganises these images as a set of patches with smaller dimensions, in this case 2×2 patches of a 4×4 image. Then a linear transform is applied to project each pixel to d, dimensions. This patch embedding step is common practice when training a Vision Transformer [22].

4.2.2. Experimental Set Up

In this section each experiment is run once with a fixed time-limit of 4.5 hours using the Gurobi solver (v11.0.1). Experiments are performed on a 4×4 image of the number 7 and the adversarial label is 1. The value of ϵ is set to 0.0001. Although this restricts the total perturbation to a small range, the search space still remains relatively large due to the nature of the problem definition. The solver must determine the perturbation to apply at each of the 16 pixels in the image such that all the perturbations sum to ϵ. These combinatorial constraints along with the double-precision computations of the `Gurobi` solver can lead to a large and complex feasible region. This choice of ϵ simplifies the problem in order to examine how the performance of the formulation changes for larger TNNs.

The experiments in this subsection investigate how the formulated transformer performs when applied to larger/more realistic models that increase the dimensions and depth of NN. In the first experiment, the impact of increasing the embedding dimensions of the TNN input is considered. This increases the number of variables and constraints created by each of the TNN layer functions and examines the scaling of the TNN model to larger inputs. In the second experiment, the impact of increasing the number of encoder layers is examined to determine the solve performance for deeper models with multiple encoder layers.

Transformers with embedding sizes of 6, 12 or 18 and either 1 or 2 encoder layers (depth) are considered.



(a) Architecture of vision transformer for verification problem

Layers	Count
Input	1
Linear	3
Residual	2L
Layer Norm	2L +1
MHA	L
FFN	L

(b) Summary of components

Figure 4.4: Description of vision TNN, where L denotes the number of encoder layers

An overview of the number of variables for these formulated TNNs is shown in Figure 4.5. The number of binary variables increases by the same factor as the depth while the number of continuous variables scales with both the depth and embedding size. Although there are far more continuous variables, the binary variables contribute significantly more to increasing the complexity of the formulation because more integrality constraints are added to the model. Therefore, scaling to larger embedding sizes is expected to be easier than scaling to deeper models, since the number of binary variables remains constant.

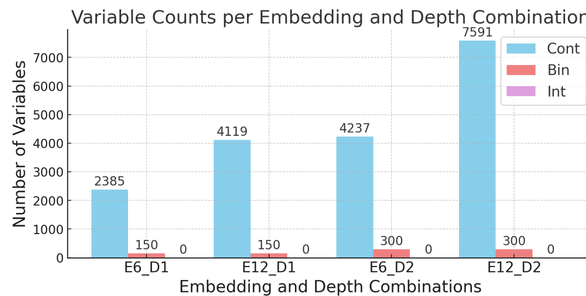


Figure 4.5: Number of continuous, binary, and integer variables for formulated transformers with embedding size (E) and depth (D).

4.2.3. Numerical Results

Scaling Input Size

The experiments in this subsection aim to evaluate how the proposed formulation scales to TNNs with large embedding dimensions. Typically, increasing the embedding size is regarded as a way to create a more descriptive models, however this can have adverse effects on the formulation. Increasing the embedding size results in larger matrices throughout the TNN, which significantly increases the number of decision variables and, consequently, the computational workload for the solver. More

specifically, as shown in the previous subsection,increasing the embedding size by a factor, f scales the number of continuous variables by almost the same amount (Figure 4.5). Additionally, these larger matrices introduce more variables in the dot product operations in the MHA layer which is expected to cause propagated bounds to become exaggerated after fewer propagations, further challenging the solver's efficiency. The following paragraphs describe how scaling the embedding dimension impacts the optimisation of the formulated transformer.

The transformer is solved using two enhancement configurations, namely `All` and `LN_prop`. Based on the optimal trajectory experiments, `All` finds an initial solution quickly that has a small gap. `All` is one of the best configurations in these regards. Furthermore, it produced the largest number of feasible solutions which can aid the solver in removing suboptimal regions from the search space. It is expected that proving optimality will become increasingly difficult for more complex models, given the formulation's non-convexivity. Thus a configuration that easily finds feasible solutions can be beneficial for solving. On the other hand, the `LN_prop` is slower to find an initial feasible solution but converges at a much faster rate. The `LN_prop` configuration supplies fewer and simpler additional bounds and cuts to the solver. As a result, it relies more on the solver's build-in techniques than `All`. In the optimal trajectory problem, this led to faster convergence. Therefore, these experiments compare a more informative yet slower to converge configuration (`All`) with a less informative configuration that explores the search space more efficiently (`LN_prop`).The results can be found in Table 4.5.

Config	E	D	Time	# Nodes	# Solutions	Gap	Objective
`LN_Prop`	6	1	15s	2.4×10^3	4	4.00×10^{-6}	-12.498
`All`	6	1	13s	3.4×10^3	4	5.00×10^{-6}	-12.498
`LN_Prop`	12	1	1m 3s	3.4×10^4	3	9.20×10^{-5}	-15.711
`All`	12	1	44s	2.8×10^4	3	8.60×10^{-5}	-15.711
`LN_Prop`	18	1	4h 30 min	2.6×10^6	0	inf	-16.735
`All`	18	1	2m 51s	4.4×10^3	3	1.00×10^{-5}	-16.754

Table 4.4: Results with varying embedding dimension (E) and constant depth (D) equal to 1.

One notable result in Table 4.5 is the differing objective values for each combination of embedding size and depth. This occurs because each of these combination represents a different trained TNN. As such these trained models can have dissimilar learnt parameters and predictions.

Similar to the optimal trajectory results, the verification problem reveals that the `All` configuration finds an initial feasible solution faster than `LN_prop` for all embedding sizes (Figure 4.6). Nevertheless, with an embedding size of 6, both configurations find the optimal solution within similar amount of time and find the same number of solutions (4). The configurations differ, however, in the number of nodes they explore. The `All` configuration explores 1×10^3 more nodes than `LN_prop` before finding the optimal solution, indicating that the solver was able to make better branching decisions for `LN_prop`. This is likely due to `LN_prop` imposing fewer constraints and thus a less complex search space.

The convergence plots in Figure 4.6 highlight the the model complexity trade-off when the transformer is scaled to larger embedding sizes. As the embedding size increases, the more complex `All` configuration outperforms `LN_prop`. This indicates that as models grow and problem definitions become harder, the advantages of more complex configurations outweigh their overhead. In this case, the additional McCormick envelopes and fixed parameter and constraints bounds included in `All` can be seen to aid in scaling the TNN. For `All` the solve times increase at a much slower rate than for `LN_prop`.

`All`, leads to tighter initial gaps which aligns with previous experimental results. This is expected to be a results of its fixed valued bounds and constraints. Furthermore, the relatively fast convergence for larger embedding sizes is likely due to the McCormick envelopes present in `All`, which create tighter bounds on the products in the MHA layer. Given that the verification problem's decision variables have small domains ($\boldsymbol{x}_p \in [0,1], \boldsymbol{\delta}_p \in [0, 0.0001]$) this can improve the effectiveness of the McCormick envelopes at removing infeasible regions from the search space. However, these envelopes are computed using propagated bounds and therefore become looser with the growing embedding size. This in turn diminishes the effectiveness of the McCormick envelopes. Given the trend in the convergence plots, it can be expected that further scaling will become infeasible.

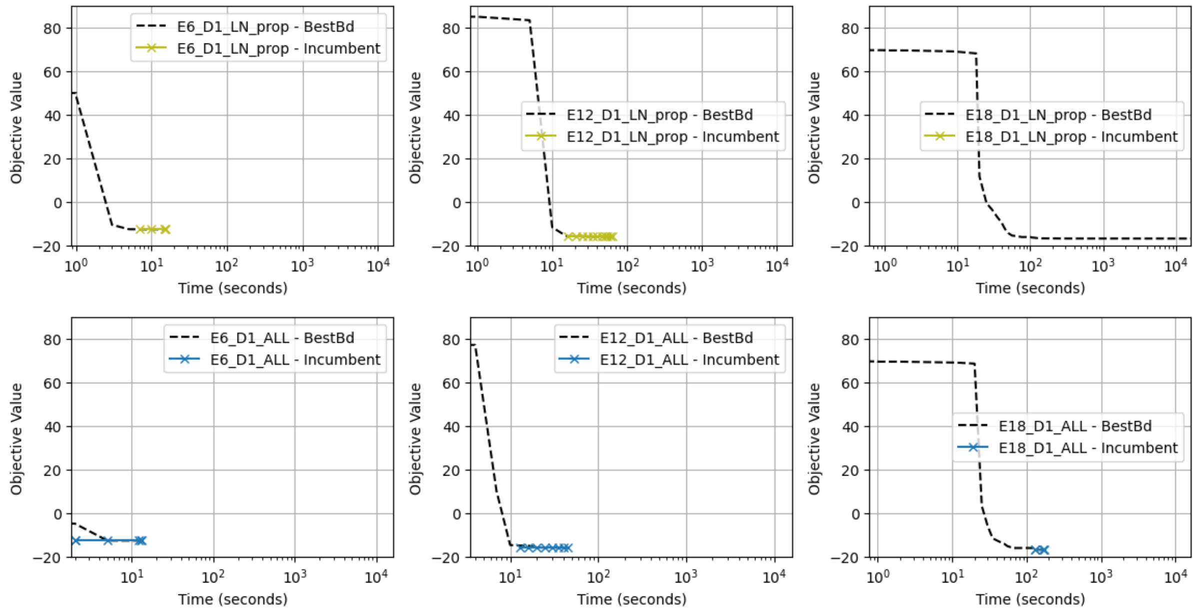


Figure 4.6: Optimality gap convergence with embedding sizes 6, 12 and 18, and 1 encoder layer

Scaling Depth

The following experiments examine the performance of the formulation for deeper transformer models. Deep architectures are very common in ML applications as they allow for hierarchical features to be captured by the model. For the TNN this means that more MHA and FFN components will be included in the model. This can be problematic for optimisation as there will be a larger number of non-linear and non-convex functions. Moreover, it drastically increases the number of decision variables since the MHA layer has many high-dimensional matrices. These factors are expected to result in models that have progressively more intricate search spaces and looser propagated bounds. Such models can struggle to find feasible solutions and give rise to poor incumbents which is detrimental to solving. Therefore, these experiments are run using the $\mathsf{MIPFocus} = 1$ parameter which prioritises finding feasible solutions over improving the objective bound.

In these experiments two trained TNNs with differing numbers of encoder layers (depth) are examined. The `No_MC` and `All` configurations are selected for these experiments based on their performance in previous experiments. Both of these configurations start with small initial gaps and are the fastest to find initial feasible solutions. These characteristics are useful for solving larger models for which it is harder to find feasible solutions, as was explained in the previous subsection. Moreover, these chosen configurations allow for a better examination of the impact of the McCormick constraints which are included in `All` but not `No_MC`.

An overview of the solve results can be found in Table 4.5. Interestingly, Table 4.5 indicates that the TNN with an embedding size of 6 does not find a feasible solution within a time limit of 12 hours, while the larger TNN with an embedding size of 12 reports a solve time of about 23 mins for both configurations. It is unusual that the a smaller model will be intractable while a larger one solves. These effects may be caused by the large number of binary variables that increases proportionately with depth. The binary variables in combination with learnt parameters (which impact the propagated bounds) appear to result in a search space that is difficult to traverse for the model with embedding size 6 and depth 2.

Looking at the convergence plots in Figure 4.7, vastly different bounds can be seen for the two transformer models with embedding size 6. The model with embedding size 6 and 1 encoder layer has significantly better best bounds that than the model with 2 encoder layers. This illustrates why there is such a stark difference between their solve times.

The convergence plots for the model with an embedding size 12 are shown in Figure 4.8. For both

Config	E	D	Time	# Nodes	# Solutions	Gap	Objective
No_MC	6	1	14s	7.0×10^3	3	9.80×10^{-5}	-12.498
All	6	1	10s	2.7×10^3	2	6.80×10^{-5}	-12.498
No_MC	6	2	12h 0m	1.1×10^7	0	inf	NaN
All	6	2	12h 0m	8.7×10^6	0	inf	NaN
No_MC	12	1	54s	2.4×10^4	3	9.90×10^{-5}	-15.711
All	12	1	1m 34s	4.9×10^4	3	9.80×10^{-5}	-15.711
No_MC	12	2	23m 20s	2.8×10^3	5	5.20×10^{-5}	-9.522
All	12	2	23m 11s	1.3×10^4	3	9.90×10^{-5}	-9.523

Table 4.5: Results with varying embedding dimension (E) and encoder depth (D).

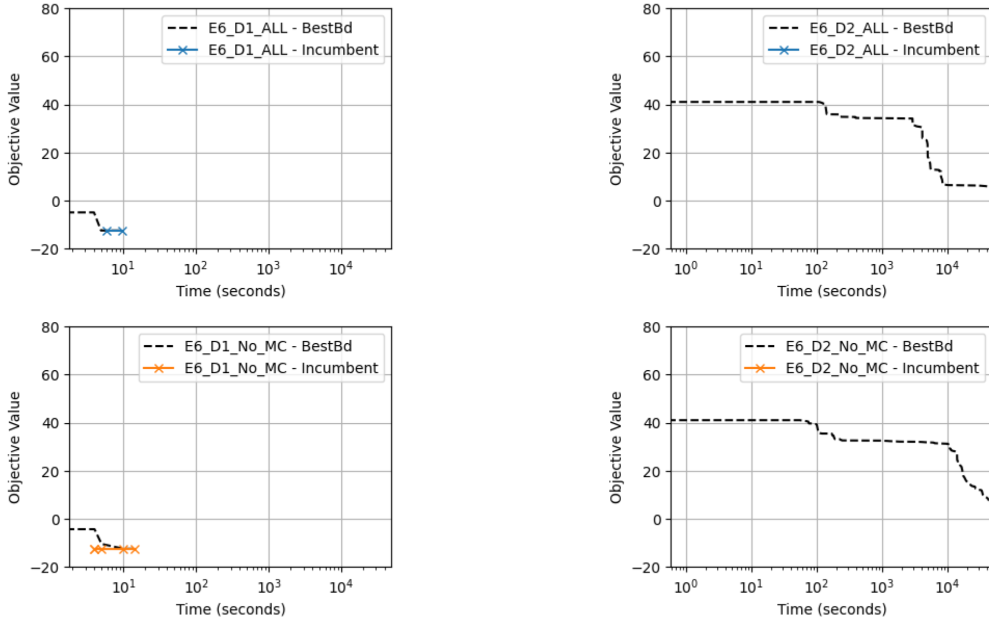


Figure 4.7: Optimality gap convergence with embedding size 6 and depths 1,2

depths, near equal solve times are achieved. The No_MC however finds more solutions than All for the deeper TNN. The convergence traces again show a drastic change in best bounds for a depth of 2 whereby it takes to solver a much longer time to improve these values. The result of scaling the depth by 2, is a solution time that is about 23 times longer. Therefore, it can be expected that larger TNNs will quickly be infeasible to solve.

Overall, for an embedding size of 12 the addition of the McCormick envelopes (All) leads to tighter initial bounds for a depth of 1 but a slightly slower solution time. However for a depth of 2, the solution time for All is somewhat better than No_MC. This gives the impression that the McCormick envelopes are more advantageous for harder problems. Nevertheless further experiments with deeper TNNs would be required to better understand the impact of the McCormick envelopes scalability.

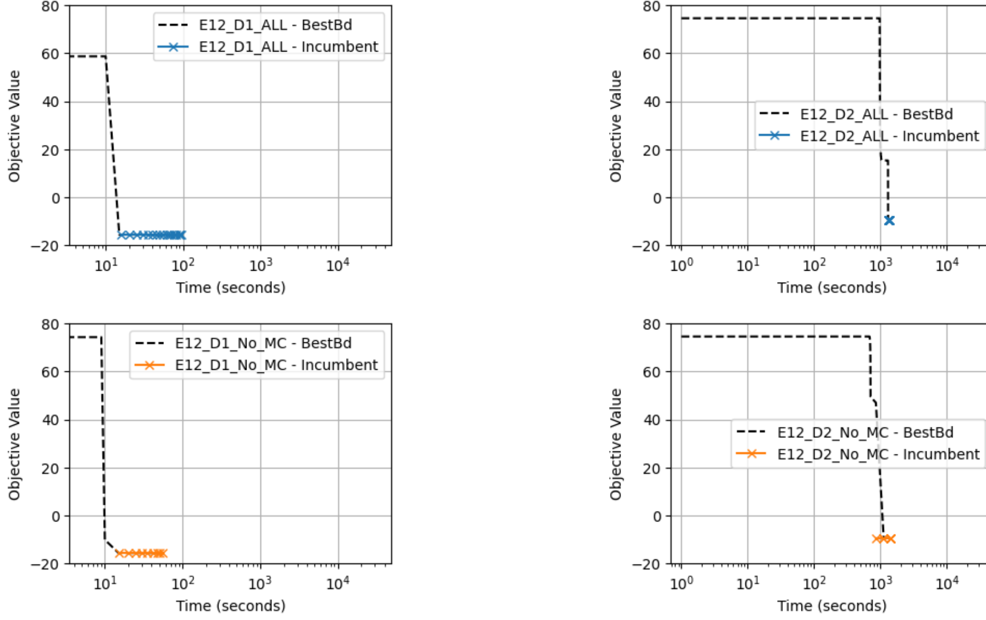


Figure 4.8: Optimality gap convergence with embedding size 12 and depths 1,2

4.3. Reactor Case Study

4.3.1. Problem Definition

The reactor problem determines the optimal temperature (T_{in}) and pressure (P_{in}) required at the reactor inlet to produce the maximum amount of methanol ($\dot{n}_{\text{CH}_3\text{OH}}$) at the reactor outlet. The reactor is modelled as a sequence of spatial components, starting from the inlet at position 0 to the outlet at position $\mathcal{L}$, using a trained TNN. The transformer takes the initial states ($\mathbf{x}_0$) as input and predicts the amount of methanol at the outlet. This is more formally represented as shown in the reactor optimisation problem below.

$$\max_{T_{\text{in}}, P_{\text{in}}} \quad \dot{n}_{\text{CH}_3\text{OH}}(\mathcal{L}) \tag{4.10}$$

$$\text{s.t.} \quad \dot{n}_{\text{CH}_3\text{OH}}(\mathcal{L}) = \mathsf{TNN}(\mathbf{x}_0) \tag{4.11}$$

$$\dot{n}_{\text{CO}}(\mathcal{L}) \leq 0.02 \tag{4.12}$$

$$450 \leq T_{\text{in}} \leq 550 \tag{4.13}$$

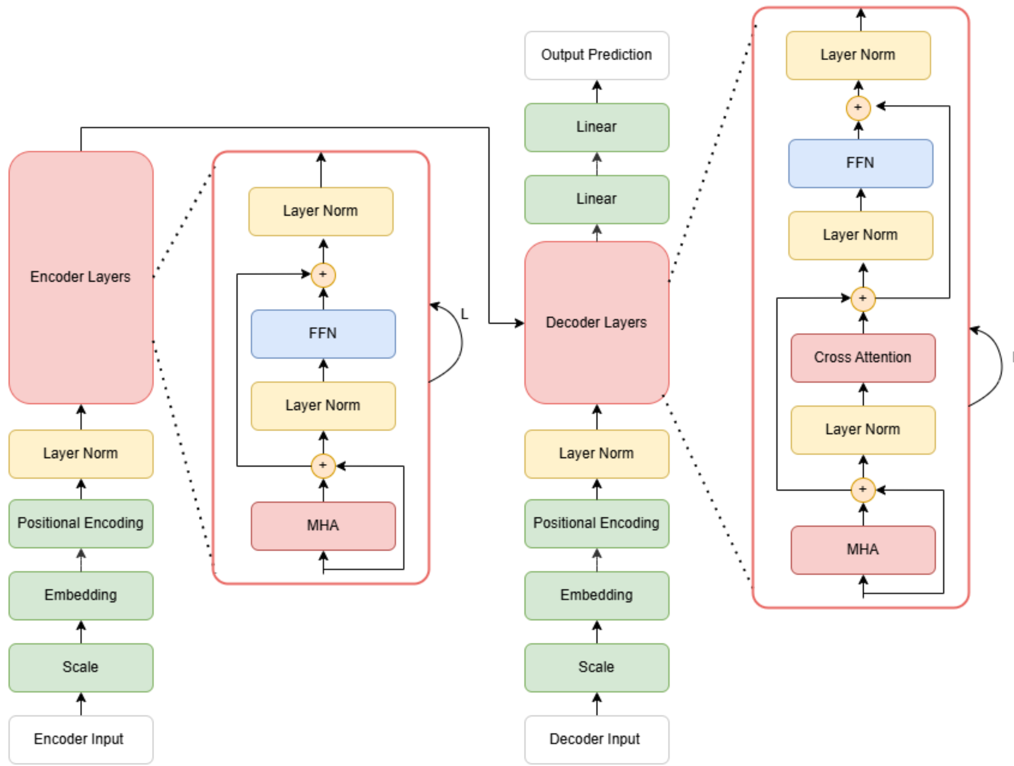
$$62 \leq P_{\text{in}} \leq 68 \tag{4.14}$$

The TNN model is trained on inlet data consisting of the temperature, pressure, and amounts of various compounds in the reactor stream: carbon monoxide ($\dot{n}_{\text{CO}}$), carbon dioxide ($\dot{n}_{\text{CO}_2}$), hydrogen ($\dot{n}_{\text{H}_2}$), methane ($\dot{n}_{\text{CH}_4}$), methanol ($\dot{n}_{\text{CH}_3\text{OH}}$), water ($\dot{n}_{\text{H}_2\text{O}}$), and nitrogen ($\dot{n}_{\text{N}_2}$). These variables make up the dimensions of the input variable $\mathbf{x}$, and are auto-regressively predicted at each segment of the reactor.

The trained transformer model was implemented using the time-series transformer class from the `HuggingFace` library. The model embeds the 9 input features to 64 dimensions and has two encoder and two decoder layers. The architecture of this model is shown in Figure 4.9. It must be noted that the FFNs in this architecture use the SiLU activation function, which is a smooth approximation of the ReLU activation function.

4.3.2. Experimental Set Up

The reactor problem experiments are run for a fixed time-limit of 24 hours with the `Gurobi` solver (v11.0.1). Three enhancement configurations are considered, namely the `All`, `No_MC`, and `LN_Prop` configurations. The `All` configuration is chosen as it was shown to be adept at scaling to larger



(a) TNN architecture for reactor problem with SiLU FFNs

Layers	Count
Input	2
Linear	8
Residual	5L
Layer Norm	5L + 2
MHA	3L
FFN	2L

(b) Summary of components

Figure 4.9: Description of reactor TNN, where L denotes the number of encoder and decoder layers

transformer models in the verification problem experiments. While, this configuration is fairly fast a finding an initial, `No_MC` is faster. `No_MC` is the fastest at finding an initial solution. Therefore, this configuration could potential provide a feasible solution to the problem and give an idea of the optimality gap early on. The last chosen configuration is `LN_Prop`. This configuration boosted the fastest solve times for the optimal trajectory problem but was not advantageous for the larger models used in the verification problem. The effectiveness of this configuration degrades rather quickly with size and as a result it can give insight into the limits of the proposed formulation for encoder-decoder models.

The trained transformer model results in a formulation with the following size: This model has a

Cont	**Bin**	**Int**
179255	388	0

Table 4.6: The number of continuous, binary, and integer variables for the formulated reactor TNN

similar number of binary variables as the previously discussed verification transformer. However it has considerably more continuous variables.

4.3.3. Numerical Results

Encoder-Decoder TNN

The experiments in this subsection investigates the performance of a larger encoder-decoder transformer applied as a surrogate model in a reactor optimisation problem. A summary of the solve results is presented in Table 4.7. These results reveal that none of the proposed configurations are capable of optimising the reactor problem or even finding a feasible solution within 24 hours.

Config	Time	# Nodes	# Solutions	Gap	Objective
No_MC	24h	1.78×10^3	0	inf	4.36×10^{-2}
LN_prop	24h	3.41×10^3	0	inf	4.36×10^{-2}
All	24h	1.38×10^3	0	inf	4.36×10^{-2}

Table 4.7: Solve results for reactor optimisation problem. Time is shown in hours.

Figure 4.10 depicts the convergence plots for these experiments. These graphs show that the best bound remains constant throughout the solve process for all configurations. No conclusions can be drawn about the best bound without knowing the optimality gap. Therefore it is not clear how good this upper bound to the objective value is.

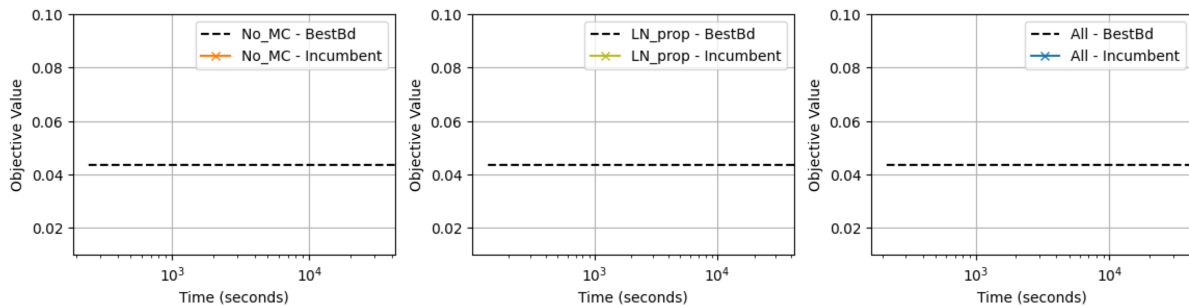


Figure 4.10: Optimality gap convergence for reactor transformer

Overall, these results emphasize the need for the transformer formulation to be further strengthened and extended upon before it can be useful for realistic TNNs with multiple encoder and decoder layers.

4.4. Key Results

This section serves as an overview of the key experimental results from the previous case studies. The optimal trajectory problem verified the correctness of the TNN and showed that the formulation alone is too complex to be solved even for a small illustrative example. The proposed enhancements improve solving efficiency and allow the for small formulated TNNs to be solved to global optimality. More work is needed, however, to optimise over larger NNs.

The solve performance for the different configurations of enhancements varies depending on the problem set up and TNN architecture. For smaller models, less complex enhancement configurations are more advantageous. However, as the problem size grows, applying more bounds and cuts became necessary for solving. In terms of problem set up, the initial domains of decision variables impacts the effectiveness of the proposed enhancements. Therefore, it is important to choose complementary enhancements that can help the solver. For example, if decision variables in the optimisation problem have large bounds then applying propagated bounds may only worsen the bounds in the problem. Instead, fixed founds, outer approximations or rescaling techniques can be used to ensure that the search space is not drastically expanded.

The proposed fixed bounds and cut are shown to be mostly beneficial for reducing the initial gap but are not very impactful in the later stages of solving. On the other hand the dependent bounds and cuts can lead to exaggerated initial bounds but aid in faster convergence to the optima. Nevertheless, scaling the larger and deeper transformers remains a challenge.

The main challenges in scaling these formulations to larger NNs are tight initial bounds and the inability to quickly find improved solutions. In many cases the propagated bounds lead to vast over approximations and create large search spaces. While some of the proposed bounds and cuts in the configurations help to mitigate against these effects, further work is required to construct tighter bounds for the continuous variables. For instance by using function approximators to construct bounds like the bi-linear McCormick envelopes. Furthermore, even though there are fewer binary variables than continuous, these variables lead to a significantly more intricate search space which greatly dimished scalability. Thus it would be beneficial to eliminate them from the formulation. This can be done, for example, by reformulating the max-normalised softmax function using a disjunctive approach rather than a Big-M approach.

5
Conclusions

In this section the outcomes in relation to the research question are outlined. Subsequently, possible future research directions are described.

5.1. Research Outcomes

In this work, the first mathematical optimisation formulation for a trained TNN is proposed (Section 3.1). To create this formulation, each component of the Vanilla transformer [57] is studied. The purpose of each layer, as well as the operations they apply, are presented in Section 2.1.2. The operations of the trained model are used to derive the proposed formulation in a manner which ensures that both models give the same result. This design choice was made so that the behaviour of the trained TNN would be preserved by the optimisation TNN. Thus the ML model can be converted to an optimisation model without a need to reverify its performance. The choice to preserve exactness of the models, leads to a non-convex MINLP formulation which is typically difficult to solve to global optimality. Simpler formulations could be established using convexification or linearisation techniques, but this would develop an optimisation TNN that only approximates the behaviour of the trained model.

In Section 4.1, a small TNN and a simple optimal trajectory problem are chosen to assess the formulation strategy. The results illustrate the complexity of the formulation, showing that in such a simple context the TNN cannot be optimised within 3 hours using the `Gurobi` solver. Thus, state-of-the-art solvers alone are not sufficient to optimise over the formulated TNN.

In the hope of solving the optimal trajectory problem, enhancements are proposed to aid the solver in its search. These enhancements focus on reducing the search space by excluding infeasible regions. Each layer of the TNN is examined to determine how information about its operations can be leveraged to create informed cuts or to reduce the bounds on variables. Other weaker bounds are also generated using bound propagation and interval arithmetic. Furthermore, concave and convex outer approximators for the dot product and softmax operations are applied as a way to strengthen variable bounds. These techniques are presented in Section 3.3.

In Section 4.1 various subsets of the proposed bounds and cuts (configurations) are applied to the optimal trajectory problem. Given the nature of non-convex MINLP problems, adding new bounds and cuts can increase the intricacies of the search space by introducing more local optima. At the same time, the enhancements can reduce the search space leading to fewer solve iterations and faster optimisations. These experiments aim to understand the subtle balance between computational complexity and solving efficiency when optimising over a TNN. The following key results are found.

First, the enhanced configurations, with the exception of `No_MC`, allow for the optimal trajectory problem to be solved to optimality. In the best and worst case the optimality gaps are 0 and 3.97×10^{-1}. Moreover, the initial best bounds are improved to be equal to the optimal solution. This indicates that by including the proposed enhancements, the solving efficiency of the formulation can be improved.

Nevertheless, the added benefit varies for each configuration. The configurations show a range of run

times between 6 mins to reaching the time limit of 1 hour. Furthermore, some explore almost 10 times more nodes before finding the optimum. The degradation in solve performance is found to be related to overly restricting the values of specific variables. In cases where multiple enhancements are applied to limit a variable's domain, it must be carefully considered how these enhancements interact and if all of them are necessary. As such, it would be beneficial to explore the enhancements added to the multi-head attention layer in greater detail. Currently, the groupings of enhancement configurations are too large. At this time, experiments are limited to either adding few enhancements or adding many restrictions to the same variables in the MHA components. It would be interesting to further investigate what would be the best combination of enhancements to achieve a small initial gap and frequent feasible solutions, thus enabling fast convergence to the optima.

The results showed that configurations with a combination of fixed bounds and propagated bounds performed better than other configurations. The fixed bounds were shown to be associated with smaller initial gaps, and infrequent improved solutions. On the other hand, the propagated, especially the tighter McCormick-based ones, were found to be associated with more frequent improved bounds which led to progressively smaller gaps. Some constraints, like the summation of attention weights to 1 did not provide any noticeable improvements and should be removed from other configurations to reduce the unnecessary complexity of this combinatorial constraint.

The proposed enhancements are capable of improving the search efficiency for small encoder-only transformer models. For 1 encoder layer, models with up to 18 embedding dimensions and for 2 encoder layers, models with up to 12 embedding are optimised. Still further work is required before larger NNs can included in optimisation problems. The main challenge comes from the lack of feasible solutions. Therefore, more cuts should be constructed to exclude infeasible regions from variable bounds, especially for the non-linear and non-convex functions. This work also constructed outer-approximations for the softmax function which were not examined in detail during experiments. It would be interesting to investigate these outer-approximation constraints as they could potentially help to solve larger models despite their high computational overhead.

5.2. Limitations

This research is limited in the following ways. One limitation lies in the exploration of the various proposed configurations. The current analysis of does not sufficiently explain interactions the interplay between various proposed bounds and cuts. Thus few recommendations can be made about useful combinations of enhancements. On the other side of things, the formulation may contain constraints do not aid in solving but add unnecessary complexity. A more extensive analysis would be beneficial to give a better understanding of what techniques would be promising to further strengthen the formulation. A useful approach for investigating this is full factorial analysis of variance.

Furthermore, the decoder component has slightly different mechanisms than the encoder. However, no clear examination of these differences and their impact on solving are presented. Given that there is a masking component in the decoder which sets some variables to 0, this can effectively reduce the number of decision variables the solver needs to determine. Therefore, it may be more tractable to optimize over larger decoder-only TNNs.

Another limitation is that the optimal trajectory and verification experiments may provide misleading optimisation results. The input values to these transformers are not restricted to be in a range close to the training data. Since ML models only capture relationships in the training data, therefore they are poor at extrapolating to data points that are significantly different to the training instances. This leads to modelling errors and unrealistic solutions. Yang, Balaprakash, and Leyffer (2022) provide methods for designing optimisation models with NNs that don not make predictions far outside the range of their training data [61]. It is imperative that formulated NNs include such techniques in order to avoid modelling errors.

5.3. Outlook and Future Work

Further research is still required to create a TNN formulation that solve more realistic scenarios with larger TNNs embedded in harder problems. This work shows that the size of the TNN both in embedding dimension and depth greatly diminishes solve efficiency. The main challenge is that the formulations

complexity and general lack of strong bounds makes it difficult to find feasible solutions. The main sources of complexity are the FFN and MHA layers. In terms of the FFN, other existing formulations could be investigated. For example, the partition-based formulation [56] could be used to balance the complexity and tightness of the FFN to improve the ability to find feasible solutions.

The MHA layer introduces a large number of continuous and binary variables into the model many of which are part of non-linear functions. Therefore, the complexity of this layer should be reduced. Binary variables can be removed by reformulating the max-normalised softmax function to use disjunctive constraints instead of the Big-M method. Furthermore, tight approximations or envelopes would be beneficial in reducing the search space by removing infeasible regions. This not only leads to fewer nodes to examine in the search tree but makes it easier for the solver to find feasible solutions. Furthermore, alternative formulations of the MHA layer can be investigated. For example, using a reduced space formulation techniques [45]. In this way the complexity of the MHA layers could be hidden from the solver and it would be easier to optimise over deeper TNNs.

Another avenue to research would be to investigate dynamically adding the proposed enhancements to the model. The results in the reactor case study show that after some time the bounds stagnate around the optimal solution and no further solving progress is made. It could be interesting to investigate the addition of more constraints at this stage. If a dynamic setting is used, less constraints could be used in the beginning to speed up solving, and then more complex constraints could be supplied when the gap stagnates (e.g., the softmax approximators). This would provide additional information to the solver, hence increase model complexity, only when it is necessary. These so called "lazy constraint" techniques have been proven to be beneficial and are already included in many MINLP solvers [31].

Besides the investigation of new formulation approaches, future research could consider pruning of TNNs as a means to optimisation over larger models. Using sparse networks can reduce the embedding size at various stages in the transformer, which would make the model easier to solve. This could then lead to less complex deeper models and potentially extend optimisation to more realistic TNNs. One drawback could be a reduction in predictive performance. However, experiments by Jaszczur et al. [27] report that sparse TNNs show comparable predictive abilities which indicates that this could be a promising approach.

Lastly, research could investigate in more detail the impact of TNN architecture on optimisation performance. Given that there is still no standard architecture used for this model, it would be worthwhile to investigate how to structure TNNs for optimisation. For example, encoder-only architectures without positional encoding components have no information about the ordering of sequence elements. As such, every ordering of the same input values should lead to the same result, resulting in an unnecessarily large search space. Symmetry breaking cuts can be used to reduce the search space by enforcing a non-decreasing ordering of sequence elements. This applies a similar approach as outlined in [64], which was shown to significantly reduce the search space for GNNs. It must be noted that such a TNN would not be applicable for cases where sequence ordering is of high importance (e.g. time-series modelling) or for TNNs with decoder components, due to their casual masking in the masked multi-head attention layer.

References

[1] Oludare Isaac Abiodun et al. “State-of-the-art in artificial neural network applications: A survey”. In: *Heliyon* 4.11 (2018).

[2] Adi Haviv et al. “Transformer Language Models without Positional Encodings Still Learn Positional Information”. In: *Findings of the Association for Computational Linguistics: EMNLP 2022*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 1382–1390. DOI: 10.18653/v1/2022.findings-emnlp.99. URL: https://aclanthology.org/2022.findings-emnlp.99.

[3] Sabeen Ahmed et al. “Transformers in time-series analysis: A tutorial”. In: *Circuits, Systems, and Signal Processing* 42.12 (2023), pp. 7433–7466.

[4] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=YicbFdNTTy.

[5] Ross Anderson et al. “Strong mixed-integer programming formulations for trained neural networks”. In: *Mathematical Programming* 183.1-2 (2020), pp. 3–39.

[6] Anurag Arnab et al. “Vivit: A video vision transformer”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 6836–6846.

[7] Pietro Belotti et al. *On feasibility based bounds tightening*. Jan. 2012.

[8] Ambrogio Maria Bernardelli et al. “Multiobjective Linear Ensembles for Robust and Sparse Training of Few-Bit Neural Networks”. In: *INFORMS Journal on Computing* (2024).

[9] Soha Boroojerdi and George Rudolph. “Handwritten Multi-Digit Recognition With Machine Learning”. In: *2022 Intermountain Engineering, Technology and Computing (IETC)*. 2022, pp. 1–6. DOI: 10.1109/IETC54973.2022.9796722.

[10] Michael L. Bynum et al. *Pyomo–optimization modeling in python*. Third. Vol. 67. Springer Science & Business Media, 2021.

[11] F. Ceccon et al. “OMLT: Optimization & Machine Learning Toolkit”. In: *Journal of Machine Learning Research* 23.349 (2022), pp. 1–8.

[12] Yingyi Chen et al. “A review of the artificial neural network models for water quality prediction”. In: *Applied Sciences* 10.17 (2020), p. 5776.

[13] Danimir T. Doncevic et al. “Deterministic Global Nonlinear Model Predictive Control with Neural Networks Embedded”. In: *IFAC-PapersOnLine* 53.2 (2020). 21st IFAC World Congress, pp. 5273–5278. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.1207. URL: https://www.sciencedirect.com/science/article/pii/S2405896320316037.

[14] Matteo Fischetti and Jason Jo. “Deep neural networks and mixed integer linear optimization”. In: *Constraints* 23.3 (2018), pp. 296–309.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. Chap. 4.

[16] Bjarne Grimstad and Henrik Andersson. “ReLU networks as surrogate models in mixed-integer linear programs”. In: *Computers & Chemical Engineering* 131 (Dec. 2019), p. 106580. ISSN: 0098-1354. DOI: 10.1016/j.compchemeng.2019.106580. URL: http://dx.doi.org/10.1016/j.compchemeng.2019.106580.

[17] Nate Gruver et al. “Large language models are zero-shot time series forecasters”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[18] Gurobi Optimization, LLC. *Gurobi Machine Learning Documentation.* Accessed: 2024-10-27. 2024. URL: https://gurobi-machinelearning.readthedocs.io/en/stable/index.html.

[19] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2024. URL: https://www.gurobi.com.

[20] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual - General Constraints.* Accessed: 2024-10-26. 2024. URL: https://www.gurobi.com/documentation/current/refman/general_constraints.html#subsubsection:GenConstrFunction.

[21] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Advances in neural information processing systems* 30 (2017).

[22] Kai Han et al. "A Survey on Vision Transformer". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.1 (2023), pp. 87–110. DOI: 10.1109/TPAMI.2022.3152247.

[23] Anwar Ul Haque, Sayeed Ghani, and Muhammad Saeed. "Image captioning with positional and geometrical semantics". In: *IEEE Access* 9 (2021), pp. 160917–160925.

[24] William E Hart, Jean-Paul Watson, and David L Woodruff. "Pyomo: modeling and solving mathematical programs in Python". In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260.

[25] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[26] Saidul Islam et al. "A comprehensive survey on applications of transformers for deep learning tasks". In: *Expert Systems with Applications* (2023), p. 122666.

[27] Sebastian Jaszczur et al. "Sparse is enough in scaling transformers". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 9895–9907.

[28] Herman Kamper. *Transformers Notes - NLP 817.* Accessed: 2024-11-23. 2024. URL: %5Curl%7Bhttps://www.kamperh.com/nlp817/notes/11_transformers_notes.pdf%7D.

[29] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[30] Olga Kosheleva, Martine Ceberio, and Vladik Kreinovich. "Adding constraints–a (seemingly counterintuitive but) useful heuristic in solving difficult problems". In: *Constraint programming and decision making* (2014), pp. 79–83.

[31] Jan Kronqvist and Andreas Lundell. "Convex minlp–an efficient tool for design and optimization tasks?" In: *Computer aided chemical engineering.* Vol. 47. Elsevier, 2019, pp. 245–250.

[32] Pedro Lara-Benítez et al. "Evaluation of the transformer architecture for univariate time series forecasting". In: *Advances in Artificial Intelligence: 19th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2020/2021, Málaga, Spain, September 22–24, 2021, Proceedings 19.* Springer. 2021, pp. 106–115.

[33] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[34] Wentao Liang et al. "Detecting Resource Release Bugs with Analogical Reasoning". In: *Scientific Programming* 2022 (Feb. 2022). DOI: 10.1155/2022/3518673.

[35] Tianyang Lin et al. "A survey of transformers". In: *AI Open* 3 (2022), pp. 111–132. ISSN: 2666-6510. DOI: https://doi.org/10.1016/j.aiopen.2022.10.001. URL: https://www.sciencedirect.com/science/article/pii/S2666651022000146.

[36] Jeff Linderoth and Jonas Schweiger. *MINLP: Theory, Algorithms, Applications: Lecture 6, Nonconvex Problems and Advanced Techniques.* https://www.imus.us.es/IMUS-MSRI2016/assets/media/docs/lectures/lec_lind_6.pdf. June 2016.

[37] Francisco Javier López-Flores, César Ramírez-Márquez, and José María Ponce-Ortega. "Process Systems Engineering Tools for Optimization of Trained Machine Learning Models: Comparative and Perspective". In: *Industrial & Engineering Chemistry Research* 0.0 (2024), null. DOI: 10.1021/acs.iecr.4c00632. eprint: https://doi.org/10.1021/acs.iecr.4c00632. URL: https://doi.org/10.1021/acs.iecr.4c00632.

[38] Garth P McCormick. "Computability of global solutions to factorable nonconvex programs: Part I—Convex underestimating problems". In: *Mathematical programming* 10.1 (1976), pp. 147–175.

[39] Tom McDonald et al. "Mixed-integer optimisation of graph neural networks for computer-aided molecular design". In: *Computers & Chemical Engineering* (2024), p. 108660. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2024.108660. URL: https://www.sciencedirect.com/science/article/pii/S0098135424000784.

[40] Syed Rameez Naqvi et al. "An optimization framework for codes classification and performance evaluation of RISC microprocessors". In: *Symmetry* 11.7 (2019), p. 938.

[41] OpenAI. *ChatGPT: Chat Generative Pre-trained Transformer*. Language model developed by OpenAI. 2023. URL: https://openai.com/chatgpt.

[42] Pyomo Developers. *Issue #3359: Pyomo fails to solve large MINLP models*. Accessed: 2024-11-06. 2023. URL: https://github.com/Pyomo/pyomo/issues/3359.

[43] Nibaldo Rodriguez et al. "Optimization Algorithms Combining (Meta)heuristics and Mathematical Programming and Its Application in Engineering". In: *Mathematical Problems in Engineering* 2018 (Sept. 2018), pp. 1–3. DOI: 10.1155/2018/3967457.

[44] Jerret Ross et al. "Large-scale chemical language representations capture molecular structure and properties". In: *Nature Machine Intelligence* 4.12 (2022), pp. 1256–1264.

[45] Artur M Schweidtmann and Alexander Mitsos. "Deterministic global optimization with artificial neural networks embedded". In: *Journal of Optimization Theory and Applications* 180.3 (2019), pp. 925–948.

[46] Yaroslav D Sergeyev and Dmitri E Kvasov. *Deterministic global optimization: An introduction to the diagonal approach*. Springer, 2017.

[47] David B. Shmoys. *Lecture Notes for ORIE 6300: Mathematical Programming I*. Accessed: 2024-10-07. 2019. URL: https://people.orie.cornell.edu/dsd95/teaching/orie6300/ORIE6300Fall2019notes.pdf.

[48] Sigmapedia. *Empirical Rule (68-95-99.7 Rule)*. Image, Accessed: 2024-10-27. 2024. URL: https://www.sigmapedia.com/includes/term.cfm?&word_id=2183&lang=eng.

[49] Yuanbing Song et al. "Double-head transformer neural network for molecular property prediction". In: *Journal of Cheminformatics* 15 (Feb. 2023). DOI: 10.1186/s13321-023-00700-4.

[50] Shashank Sonkar and Richard G Baraniuk. "Investigating the Role of Feed-Forward Networks in Transformers Using Parallel Attention and Feed-Forward Net Design." In: *CoRR* (2023).

[51] Robin Strudel et al. "Segmenter: Transformer for semantic segmentation". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 7262–7272.

[52] Gang Sun and Shuyue Wang. "A review of the artificial neural network surrogate modeling in aerodynamic design". In: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 233.16 (2019), pp. 5863–5872.

[53] Amey Thakur and Archit Konde. "Fundamentals of neural networks". In: *International Journal for Research in Applied Science and Engineering Technology* 9.VIII (2021), pp. 407–426.

[54] John Thickstun. "The Transformer Model in Equations". In: 2020. URL: https://api.semanticscholar.org/CorpusID:216559335.

[55] Tómas Thorbjarnarson and Neil Yorke-Smith. "Optimal training of integer-valued neural networks with mixed integer programming". In: *Plos one* 18.2 (2023), e0261029.

[56] Calvin Tsay et al. "Partition-based formulations for mixed-integer optimization of trained ReLU neural networks". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 3068–3080.

[57] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[58] S Vijayarani, R Janani, et al. "Text mining: open source tokenization tools-an analysis". In: *Advanced Computational Intelligence: An International Journal (ACII)* 3.1 (2016), pp. 37–47.

[59] Qingsong Wen et al. "Transformers in Time Series: A Survey". In: (Aug. 2023). Ed. by Edith Elkind. Survey Track, pp. 6778–6786. DOI: 10.24963/ijcai.2023/759. URL: https://doi.org/10.24963/ijcai.2023/759.

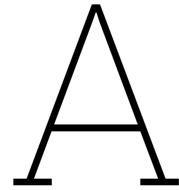
[60] Matthew Wilhelm, Chenyu Wang, and Matthew Stuber. "Convex and concave envelopes of artificial neural network activation functions for deterministic global optimization". In: *Journal of Global Optimization* 85 (Aug. 2022). DOI: 10.1007/s10898-022-01228-x.

[61] Dominic Yang, Prasanna Balaprakash, and Sven Leyffer. "Modeling design and control problems involving neural network surrogates". In: *Computational Optimization and Applications* 83.3 (2022), pp. 759–800.

[62] Heng Yang. *Optimal Control and Estimation*. Harvard University, 2023. URL: https://hankyang.seas.harvard.edu/OptimalControlEstimation/index.html.

[63] Shiqiang Zhang et al. "Augmenting optimization-based molecular design with graph neural networks". In: *arXiv preprint arXiv:2312.03613* (2024).

[64] Shiqiang Zhang et al. "Optimizing over trained GNNs via symmetry breaking". In: *Advances in Neural Information Processing Systems* 36 (2023).



Academic paper

Title: Mixed-Integer Non-Linear Formulation for Optimisation over Trained Transformer Models

Abstract: Neural networks (NNs) have become indispensable tools across various disciplines for predicting future outcomes. Recent works propose a new use case for these models which determines the inputs required to achieve a desired future outcome. To this end mathematical formulations for NNs are designed to be used in optimisation problems. Promising results have been achieved for various NNs, the Transformer Neural Network (TNN), a state-of-the-art model known for its multi-modal capabilities and exceptional contextual understanding—remains, unexplored in this context. This research develops an efficient mixed-integer non-linear formulation for a TNN, enabling the use of TNNs for decision making. The proposed formulation is evaluated on three case studies consisting of an optimal trajectory problem, a verification problem, and a reactor optimization case study. Results show that optimisation over small TNNs can be achieved in under 3 minutes. However the tractability of the formulation quickly vanishes for larger models, highlighting the complexity optimising over the TNN.

Potential Journal: Mathematical Programming