Computational Limits and Turing Completeness of Modern Neural Sequence Architectures in Regular Language Recognition

Pim Van den Bosch

January 2024

Abstract

This paper critically examines the computational capabilities and Turing completeness of modern neural sequence architectures, specifically focusing on their application in regular language recognition. At the heart of our inquiry lie three fundamental assumptions: infinite precision of weights, infinite recursion, and infinite context. These assumptions, omnipresent in theoretical proofs, serve as a foundation for asserting the Turing completeness of various models, including Recurrent Neural Networks (RNNs) and Transformers. We begin by exploring the historical and theoretical background of neural sequence models, from the inception of Seq2Seq models and RNNs to the advent of Transformer architectures. Our analysis then delves into the key theoretical works that assert the Turing completeness of these models, particularly under the lens of infinite assumptions. This theoretical exploration is juxtaposed with practical limitations, highlighted through a series of experiments designed to test these models' performance in recognizing regular languages. The experiments particularly focus on the asymptotic behavior of neural networks and their ability to approximate the indicator function of a language as the model parameters grow large. Our findings illuminate the gap between theory and practice, underscoring the complexity of implementing theoretically infinite assumptions in real-world scenarios.

1 Introduction

The fields of machine learning and artificial intelligence have witnessed a significant evolution with the introduction of Sequence-to-Sequence (Seq2Seq) models (Sutskever et al., 2014). These models commonly feature an encoder which learns to encode an input sequence in a vector, and a decoder, which learns to predict the next token in a target sequence. This encoder and decoder were originally implemented as a Recurrent Neural Network (RNN), with common variants being the Long Short-Term Memory network (LSTM) and the General Recursive Unit (GRU).

Bahdanau et al. improved upon the standard Seq2Seq architecture by introducing the attention mechanism, addressing the limitations of seq2seq models in handling long input sequences by enabling the model to focus selectively on different parts of the input sequence.

Building upon this, Vaswani et al. in 2017 introduced the transformer architecture; a significant improvement. Transformers replaced recurrent layers in the encoder and decoder with self-attention mechanisms, overcoming challenges related to parallelization and handling long-range dependencies. This shift from RNN-based approaches to transformers represented a significant advancement in the field, as this allowed Seq2Seq models solve problems in sequence recognition and generation that were previously deemed intractable [3].

With the rise of transformer architectures, a key question (re)emerged regarding their computational power, specifically their Turing completeness. Turing completeness, a fundamental concept in theoretical computer science, refers to a system's capability to simulate any Turing machine, thus performing any computation given the necessary resources. In 2019, Pérez et al. were the first to propose that transformers with positional encodings are Turing complete - under the assumption of infinite precision in the model's weights. This assumption suggested that with theoretically infinite precision, transformers could encode extensive computational capabilities within individual weights.

The path of inquiry then progressed with subsequent studies, including those by Bhattamishra et al. and Roberts (2023), which aimed to refine what constitutes Turing completeness in transformers. These studies questioned and sometimes overturned previous assumptions about the necessary components for Turing completeness. For instance, Bhattamishra et al. (2020) challenged the necessity of positional encoding, while Roberts (2023) suggested that a decoder-only architecture could suffice, even in configurations as minimal as a single layer with a single attention head.

A trend that marks the lines of research which assume infinite precision of real numbers - an assumption made originally by Siegelmann (1992) in this line of research - is the increasing disconnect between the assumptions made about model parameters or configurations and the actual implementations of those models when tested empirically. This theoretical exploration into Turing completeness has therefore been subjected to scrutiny and debate, particularly with regards to assumptions beyond the infinite precision assumption.

In 2020 Hahn provided the first theoretical work on the computational limitations of Transformers as proposed by Perez et al. The analysis highlighted the limitations of self-attention in transformers. Their analysis suggests that the transformers face intrinsic limitations in modeling specific formal languages and structures without increasing the model's depth or head count proportionally with input length.

In their analysis titled "Transformers Learn Shortcuts to Automata," Liu et al. reinforce this critique, classifying the theoretical approaches of Siegelmann (1992) - and by logical extension subsequent research utilizing the same assumptions - as "pathological". This term refers to assumptions about neural models or properties thereof that, while mathematically well defined, may not be realistic with regards to the true nature of the objects in question, representing extreme cases in the spectrum of computational principles.

Another critical perspective can be found in the paper 'Sequential Neural Networks as Automata' by Merrill (2019). Here the concept of asymptotic analysis for neural networks is introduced which claims to address the unrealistic assumptions by Siegelmann and by logical extension further investigations by Roberts (2023), Bhattamishra et al.(2020) and Perez et al. (2019). While limitations are pointed out, Merrill's work assumes that, as the parameters of neural networks grow very large, they will converge pointwise to an indicator function for any regular language, allowing for a theoretical view of NNs as Finite State Automata (FSA). Under this asymptotic assumption it is shown for RNNs that, since a language is regular if and only if there is a finite state automaton which accepts it, the languages acceptable by RNNs are equal to the regular languages. In addition they claim that this claim holds under the following conditions:

Quote:

> 1. Real-time: The network performs one iteration of computation per input symbol.
> 2. Bounded precision: The value of each cell in the network is representable by $O(\log n)$ bits on sequences of length n.

In this following sections we will start by giving the common definition of an RNN as it stands at the root of the work on Turing completeness by Siegelmann, whose work inspired Perez et al.(2019), Bhattamishra et al.(2020) and Roberts(2023). Seq2Seq models are defined as well in order to introduce the attention mechanism of Vaswani et al. and its role in defining the transformer model. After that we will describe the FSA and Turing machines and we will evaluate the main assumptions behind the proof that RNNs are Turing complete by Siegelmann (1992) which we will then relate to the work of Perez et al (2019) and similar research on Transformers. We will contrast this with an exposition on asymptotic analysis and the finite precision proof on simple RNNs by Merrill. Finally we will ground the theoretical exposition by setting up an experiment in the methods section using grid search, a common technique for hyperparameter optimization.[1] We test whether there is any indication of pointwise convergence of RNNs to indicator functions of given regular languages given the asymptotic acceptance construction by Merill(2019) for RNNs. We conclude by addressing future considerations and ongoing research efforts.

2 RNNs and Seq2Seq Models

2.1 Introduction to Recurrent Neural Networks (RNNs)

Before delving into the specifics of sequence-to-sequence (seq2seq) models and their evolution towards Transformer architectures, it is imperative to understand the fundamental building block upon which the original seq2seq model was based: the Recurrent Neural Network (RNN). We provide an exposition following the definitions of

[1]`https://en.wikipedia.org/wiki/Hyperparameter_optimization`

2.1.1 Core Mechanism of RNNs

Recurrent Neural Networks (RNNs), a generalization of feedforward neural networks to sequences, play a crucial role in processing sequential data. Given a sequence of inputs $(x_1, ..., x_T)$, an RNN computes a corresponding sequence of outputs $(y_1, ..., y_T)$ by iterating the following equations:

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1})$$
$$y_t = W_{yh}h_t$$

The mathematical description provided here pertains to a single layer of the network. In an RNN, each layer processes the input sequence one element at a time, maintaining a hidden state h_t that effectively encodes the information seen up to that point in the sequence.

1. **Hidden State h_t:** This is the memory of the network at time step t. It's updated at each step based on the current input x_t and the previous hidden state h_{t-1}.

2. **Weight Matrices:** There are three key weight matrices in a basic RNN layer:

- W_{hx}: Weights connecting the input x_t to the hidden layer.
- W_{hh}: Weights connecting the hidden state from the previous time step h_{t-1} to the current hidden state h_t.
- W_{yh}: Weights connecting the hidden state h_t to the output y_t.

3. **Activation Function σ:** A non-linear function like sigmoid or tanh. It's applied to the linear combination of inputs and the previous hidden state to produce the new hidden state.

4. **Output y_t:** The output at time step t, which is typically a function of the current hidden state h_t alone in the simplest form of RNNs.

If an RNN has multiple layers, each layer has its own set of weight matrices and processes the sequence in a similar manner, with the output of one layer serving as the input to the next layer. This allows the network to learn more complex representations at each subsequent layer. However, the basic computational principle described by the equations remains the same for each layer.

The key feature here is the ability to maintain a form of memory through the hidden state vectors. This recurrent structure enables RNNs to model temporal dynamics and dependencies in sequential data.

2.1.2 Addressing Long-Term Dependencies: LSTMs and GRUs

Despite their strengths, standard RNNs face challenges in capturing long-term dependencies due to issues like vanishing gradients. These gradients are required to update the weight matrices necessary to model the underlying data during the learning process. Long Short-Term Memory networks (LSTMs) and Gated Recurrent Units (GRUs) were introduced to overcome this. These variants incorporate gating mechanisms that regulate the flow of information, allowing the network to better capture long-range dependencies in the data.

2.2 The Role of RNNs in Seq2Seq Models

Seq2Seq models, which marked a significant advancement in machine learning, prominently feature RNNs in both their encoder and decoder components. These models in principle take a sequence of input data

2.2.1 Addressing Variable Sequence Lengths

One of the challenges in sequence-to-sequence learning, as highlighted in Sutskever et al.'s work "Sequence to Sequence Learning with Neural Networks," is handling input and output sequences of varying lengths with potentially complex and non-monotonic relationships. Standard RNNs inherently map sequences to sequences when the alignment between inputs and outputs is known ahead of time. However, this becomes less straightforward when the lengths of the input and output sequences differ or when their relationship is complex.

2.2.2 Seq2Seq Models as a Solution

To address this, the Seq2Seq model architecture was introduced, where one RNN (the encoder) maps the input sequence to a fixed-sized vector (the context vector). This vector encapsulates the essential information from the input sequence and serves as the initial state for the decoder RNN. The decoder then generates the output sequence, one element at a time, using the context vector and the previously generated elements as inputs. This structure allows for the flexible handling of sequences of varying lengths and is particularly effective in tasks like machine translation, where the length and structure of the output sequence are not directly aligned with the input.

2.2.3 Encoder

The encoder in a seq2seq model processes an input sequence $x = (x_1, \ldots, x_{T_x})$, where $x_t \in R^{d_x}$ is the input vector at time step t. The encoder updates its hidden state $h_t \in R^{d_h}$ for each time step using the equation:

$$h_t = f(x_t, h_{t-1})$$

where $h_{t-1} \in R^{d_h}$ is the previous hidden state, and f is a nonlinear function, commonly a simple RNN or an LSTM. The final hidden state h_{T_x} serves as the context vector c, encapsulating the entire input sequence information:

$$c = q(\{h_1, \ldots, h_{T_x}\})$$

Here, q is a function that transforms the sequence of hidden states into the context vector, often chosen as $q(\{h_1, \ldots, h_{T_x}\}) = h_{T_x}$.

2.2.4 Decoder

The decoder generates the output sequence $y = (y_1, \ldots, y_{T_y})$, where $y_t \in R^{d_y}$ is the output vector at time step t. The decoder is designed to predict the next word y_t given the context vector c, derived from the encoder, and all previously predicted words $\{y_1, \ldots, y_{t-1}\}$.To achieve this, the decoder defines a probability distribution over the sequence y by decomposing the joint probability into ordered conditional probabilities:

$$p(y) = \prod_{t=1}^{T_y} p(y_t | \{y_1, \ldots, y_{t-1}\}, c)$$

Each conditional probability $p(y_t|\{y_1, \ldots, y_{t-1}\}, c)$ is modeled using a function g :

$$p(y_t|\{y_1, \ldots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$$

In this equation, $s_t \in R^{d_s}$ is the hidden state of the decoder RNN at time t, and g is a nonlinear function (often a softmax layer) that generates the probability distribution over all possible output elements. The initial state of the decoder is often set to the context vector c, and s_t is updated recursively:

$$s_t = f'(y_{t-1}, s_{t-1}, c)$$

where f' is a nonlinear function, distinct from the encoder function f, and can also be an RNN or LSTM. The sequence of hidden states $(s_1, \ldots, s_{T_y})$ in the decoder captures the information of all previous outputs and the context vector, guiding the generation of the next output in the sequence.

3 Transformers

3.1 Seq2Seq as a Precursor to Attention and Transformers

Seq2Seq models set the groundwork for the development of Transformer architectures. The initial seq2seq framework, reliant on RNNs, encountered limitations in handling long sequences due to the sequential nature of RNN processing. This led to the introduction of attention mechanisms by Bahdanau et al., which enabled models to focus on different parts of the input sequence selectively. Vaswani et al. built upon this concept with the Transformer architecture, which replaced RNNs with self-attention mechanisms, offering advantages in parallelization and handling long-range dependencies.

3.2 Overview of Attention Mechanism

The attention mechanism is a critical component in the Transformer architecture that allows the model to dynamically focus on different parts of the input sequence. This mechanism was first introduced by Bahdanau et al.

(2014), and has since become a fundamental element in the design of advanced neural network models, particularly in the Transformer as proposed by Vaswani et al. (2017). The Transformer, as formalized by Pérez et al. (2019), leverages attention to improve the encoding of sequence information, enabling efficient and effective processing of long sequences.

3.3 Formalization of Attention

In the context of the Transformer architecture, the attention mechanism is formalized as follows by Perez et al.(2019):

Given a scoring function score : $Q^d \times Q^d \to Q$ and a normalization function $\rho : Q^n \to Q^n$, where $d, n > 0$, assume that $q \in Q^d$, and $K = (k_1, \ldots, k_n)$ and $V = (v_1, \ldots, v_n)$ are tuples of elements in Q^d.

A q-attention over (K, V), denoted by $\mathrm{Att}(q, K, V)$, is a vector $a \in Q^d$ defined as follows:

$$(s_1, \ldots, s_n) = \rho(\mathrm{score}(q, k_1), \mathrm{score}(q, k_2), \ldots, \mathrm{score}(q, k_n)) \tag{1}$$

$$a = s_1 v_1 + s_2 v_2 + \cdots + s_n v_n \tag{2}$$

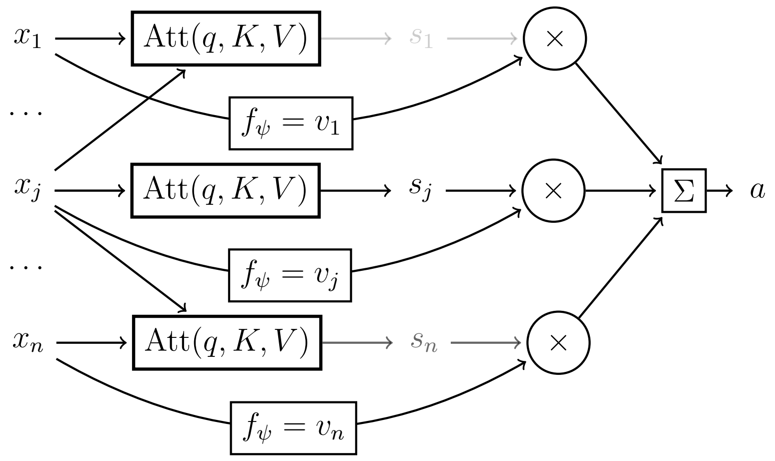

Figure 1: Illustration of the attention mechanism in the Transformer model. Each element x_i of the input sequence is transformed by a function f, possibly representing the computation of keys or values in the attention mechanism.

In this formulation, q is termed as the query, K as the keys, and V as the values. The scoring and normalization functions are not restricted, allowing for a variety of implementations. For example, a common choice for the scoring function is a feedforward network or the dot product between the query and keys. The normalization function often used is the softmax function, which converts scores into probabilities.

3.4 Transformer's Use of Attention

In the Transformer architecture, attention serves as the mechanism by which context is dynamically aggregated from different parts of the input sequence. The encoder and decoder components of the Transformer use attention in different ways:

3.4.1 Encoder's Self-Attention

The encoder uses self-attention to relate different positions of the input sequence to each other. This is achieved by computing attention scores (Equation 1) among all pairs of positions in the sequence, enabling the encoder to capture the interdependencies regardless of their distance in the sequence.

3.4.2 Decoder's Attention Mechanisms

The decoder employs two types of attention mechanisms: self-attention and encoder-decoder attention. The self-attention in the decoder similarly processes the output sequence, allowing each position in the decoder to attend to all positions in the decoder up to and including that position. The encoder-decoder attention provides a mechanism for the decoder to focus on relevant parts of the input sequence, using the encoder's output as keys and values.

3.4.3 Role in Contextual Understanding

The use of attention in the Transformer enables the model to adaptively focus and weigh different parts of the input sequence, facilitating a deeper understanding of the context. This is particularly effective in tasks like machine translation and text summarization, where the relevance of different parts of the input can vary greatly.

4 Finite State Automata and Turing Machines

4.1 Definition of a Finite State Automaton

A Finite State Automaton (FSA) is a theoretical model of computation used to represent and analyze the behavior of sequential systems. It consists of the following components:

- A finite set of states Q.
- A finite set of input symbols Σ, called the alphabet.
- A transition function $\delta : Q \times \Sigma \to Q$.
- An initial state $q_0 \in Q$.
- A set of accept states $F \subseteq Q$.

Mathematically, an FSA can be represented as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The automaton reads a string of symbols from Σ, and transitions from state to state according to the transition function δ. The string is accepted by the automaton if, after reading the entire string, the automaton is in one of the accept states.

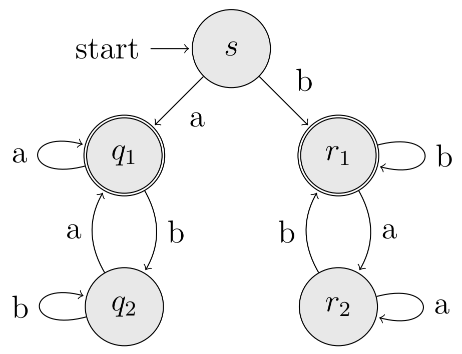

Figure 2

4.2 Definition of a Turing Machine

A Turing Machine (TM) is a more powerful theoretical model of computation that generalizes the concept of an FSA. It consists of:

- A finite set of states Q.

- A finite set of input symbols Σ.
- A finite set of tape symbols Γ, where $\Sigma \subseteq \Gamma$.
- A transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
- An initial state $q_0 \in Q$.
- A blank symbol $b \in \Gamma \setminus \Sigma$.
- A set of accept states $F \subseteq Q$.

The Turing Machine is defined as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$. It operates on an infinite tape divided into cells, each containing a symbol from Γ. The machine has a read-write head that moves along the tape and can read and write symbols and move the tape left or right. The transition function δ determines the machine's actions based on the current state and the symbol being read. The computation halts when the machine enters one of the accept states.

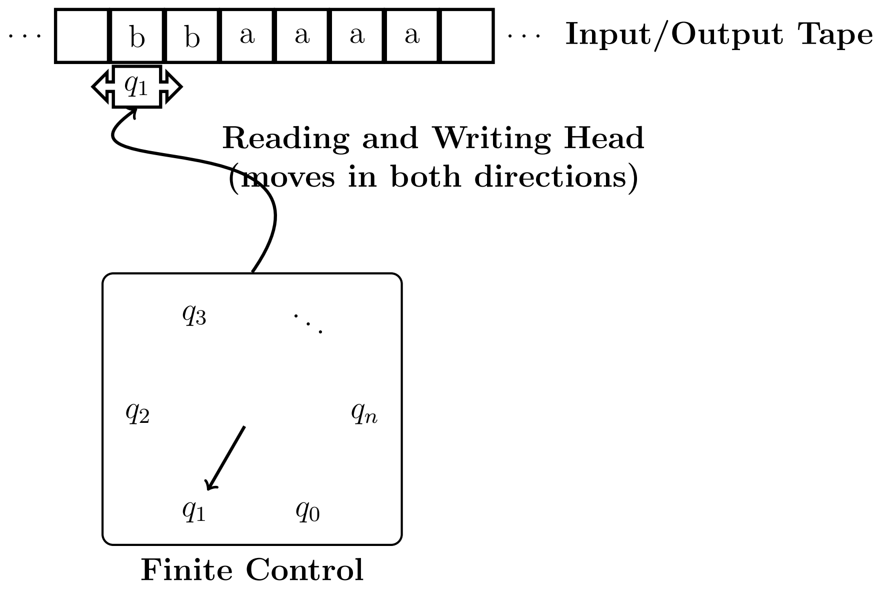

Figure 3

4.3 Formal Languages

A formal language is a set of strings of symbols, where each string is composed of symbols from a specific alphabet and is constrained by the rules of the language. The alphabet is a finite, non-empty set of symbols. Formal languages are central to theoretical computer science, particularly in the fields of automata theory and linguistics.

[Alphabet] An alphabet, denoted as Σ, is a finite, non-empty set of symbols.

[String] A string over an alphabet Σ is a finite sequence of symbols from Σ. Example: "abba" is a string over the alphabet $\{a, b\}$.

[Formal Language] A formal language L over an alphabet Σ is a set of strings composed of symbols from Σ. Each string in L adheres to the specific rules or patterns that define L. Example: (00+11)*

Formal languages can be classified based on their complexity and the type of computational models needed to recognize them. The Chomsky hierarchy classifies formal languages into several types:

- **Regular Languages:** Recognized by finite state automata. These are the simplest languages in the hierarchy.
- **Context-Free Languages:** Recognized by pushdown automata. These include languages that require a certain level of nesting or hierarchy.
- **Context-Sensitive Languages:** Recognized by linear bounded automata. These languages have more complex rules that depend on the context of the symbols.
- **Recursively Enumerable Languages:** Recognized by Turing machines. This class includes all languages that can be recognized by some computational model.

The study of formal languages involves understanding the properties of these languages, developing methods to describe them (such as grammars), and examining the computational power required to process them.

5 Turing Completeness of Transformers Assuming Infinite Precision and Recursion

5.1 Contextualizing Siegelmann's Assumptions in Modern Neural Network Research

The work of Siegelmann (1992) serves as a classic result in understanding the computational power of neural networks, particularly Recurrent Neural Networks (RNNs). Siegelmann's assertion of Turing completeness in RNNs under the assumption of infinite precision in weights forms a basis for subsequent research in the field. This assumption presents a dichotomy between the idealized models and their practical implementations, a theme that resonates throughout our exploration of modern neural architectures.

Pérez et al., Bhattamishra et al., and Roberts have extended Siegelmann's foundational concepts in their respective studies, for the transformer. These studies, while diversifying in their approach, inherently rely on assumptions similar to those posited by Siegelmann. The infinite precision assumption, alongside considerations for infinite recursion and context, remains a pivotal element in their arguments for Turing completeness.

In this section, we present an informal version original of the proof by Siegelmann, setting the stage for understanding the subsequent extensions and adaptations in the works of Pérez et al. and others. By dissecting Siegelmann's proof, we aim to highlight the key assumptions and methodologies that have influenced the trajectory of neural network research, particularly in the realm of Turing completeness.

5.1.1 Siegelmann's Proof of Turing Completeness in RNNs

Siegelmann's proof that Recurrent Neural Networks (RNNs) can achieve Turing completeness under certain assumptions is a milestone in computational theory. This proof relies on the assumption of infinite precision in the network's weights, allowing the RNN to encode an unlimited amount of information, akin to the infinite tape of a Turing Machine (TM). We present an abridged version of this proof, focusing on its core elements and implications.

5.1.2 Assumptions and Preliminaries

- Infinite Precision: The weights of the RNN are assumed to have infinite precision, enabling them to represent real numbers with an arbitrary number of decimal places. This assumption is key to encoding the states and transitions of a TM within the RNN.
- RNN Structure: The RNN considered here consists of neurons that compute a weighted sum of inputs followed by a non-linear activation function. The infinite precision of weights extends beyond typical RNNs, where weights are finite and of limited precision.
- Encoding Scheme: The proof utilizes a unique encoding scheme where the states and transitions of a TM are represented within the real number weights of the RNN.

5.1.3 Proof Outline

1. Turing Machine Simulation: The RNN is shown to simulate any TM step-by-step. Each neuron in the RNN corresponds to a component of the TM, with its state mirroring the TM's state transitions.
2. Step-by-Step Correspondence: At each step of computation, the RNN's state transition parallels the TM's transition. This is achieved through the intricate encoding of the TM's tape and states in the RNN's weights and states.
3. Encoding and Decoding Process: The RNN encodes the TM's tape, states, and transition functions in its structure, enabling it to simulate the TM's computational processes.
4. Completeness: The RNN's ability to simulate any TM implies its Turing completeness, as TMs can perform any computable function.

5.1.4 Key Considerations

- The assumption of infinite precision is critical for the RNN's ability to encode the TM's infinite tape.
- The gap between theoretical models and practical implementations is significant, as infinite precision is not feasible in real-world applications of RNNs.

5.1.5 Connecting to Modern Research

Siegelmann's proof, with its emphasis on infinite precision, directly influences the computational models explored by Pérez et al., Bhattamishra et al., and Roberts. These studies extend Siegelmann's concepts to transformer models and sequence-to-sequence architectures, maintaining the core assumption of infinite precision. Understanding Siegelmann's proof thus offers valuable insights into the foundational theories that shape current research on the computational capabilities of neural architectures.

5.2 Pérez et al.'s Proof of Turing Completeness in Transformer Networks

Pérez et al. have extended the conceptual framework of Turing completeness to Transformer networks, incorporating positional encodings. Their work demonstrates that Transformer networks, despite their architectural differences from traditional RNNs, also possess the capacity for Turing completeness under certain conditions. This section outlines their proof, connecting it back to the foundational principles established by Siegelmann.

5.2.1 Theorem and Proof Sketch

Theorem 3.4 (Pérez et al.): The class of Transformer networks with positional encodings is Turing complete.

Proof Sketch: The proof constructs a Transformer network, *TransM*, that simulates a given Turing machine M. This is achieved by representing the Turing machine's execution and state transitions within the Transformer's structure.

5.2.2 Construction

- The Turing machine M is defined as $M = (Q, \Sigma, \delta, q_{\text{init}}, F)$, where Q is the set of states, Σ is the tape alphabet, δ is the transition function, q_{init} is the initial state, and F is the set of final states.
- A string $w = s_1 s_2 \cdots s_n \in \Sigma^*$ is represented as a sequence X of one-hot vectors with corresponding positional encodings.

- The Transformer network *TransM* is designed to process this input sequence X and produce an output sequence $y_0, y_1, y_2, \ldots$ that encodes information about the state and tape symbol at each step of M's computation.

5.2.3 Inductive Proof Strategy

- *Base Case*: Assume that for the input sequence $y_0, \ldots, y_t$, each y_i contains the encoded state $q^{(i)}$ and tape symbol $s^{(i)}$ of M at time i.
- *Inductive Step*: Show that *TransM* can construct y_{t+1} containing the state $q^{(t+1)}$ and symbol $s^{(t+1)}$, thus simulating the next step of M.

5.2.4 Key Components of the Proof

- Implementation of M's transition function δ within the Transformer's first layer.
- Use of self-attention layers to compute and store the head position and symbol updates.
- Encoding and copying of the Turing machine's head movement and symbol writing into the Transformer's output sequence.
- Handling of special cases, such as the initial copying of input symbols and dealing with previously unvisited tape cells.

6 Asymptotic Analysis and the Proof on RNNs and Regular Languages

6.1 Asymptotic Analysis in Neural Network Architectures

In this section we lay out the definition of asymptotic analysis as defined by Merrill (2019). In doing so we formalize language acceptance in neural networks and what it means for a neural this section provides an introduction to the asymptotic behavior of neural networks, specifically RNNs, and establishes their capability to accept regular languages under realistic constraints. Asymptotic analysis in neural networks is a method of evaluating the behavior and capabilities of these networks as certain parameters approach their limits. This approach is particularly relevant in the context of understanding how neural networks, specifically Recurrent Neural Networks (RNNs), process and accept languages.

6.1.1 Formalizing Language Acceptance in Neural Networks

To begin, we consider the formal definition of a neural network accepting a language. This concept hinges on the network's ability to process a sequence of characters and return a probability indicating whether the sequence forms a valid sentence in a particular language. This is mathematically defined as follows:

Neural Sequence Acceptor: Let X be a matrix representation of a sentence where each row is a one-hot vector over an alphabet Σ. A neural sequence acceptor $\hat{1}$ is a family of functions parameterized by weights θ. For each θ and X, the function $\hat{1}_\theta$ is defined by:

$$\hat{1}_\theta : X \to p \in (0,1)$$

This definition encapsulates the general architecture and specific networks like LSTM, parameterized with learned weights.

6.1.2 Understanding Asymptotic Behavior in Neural Networks

The asymptotic behavior of a neural network is analyzed by considering the limit as the magnitude of its parameters becomes very large. This leads to the network's internal structure approximating a discrete computation graph, and its probabilistic output converging to the indicator function of a language.

Asymptotic Acceptance: Let L be a language with indicator function 1_L. A neural sequence acceptor $\hat{1}$ with weights θ asymptotically accepts L if:

$$\lim_{N\to\infty} \hat{1}_{N\theta} = 1_L$$

This definition allows us to view the network as an automaton and provides a framework for analyzing the network's computational capabilities.

6.1.3 State Complexity in Neural Networks

State complexity is a measure of the number of distinct states or values a node within a network can assume, based on the length of the input sequence. We introduce additional mathematical constructions to further explore this concept.

Hidden State: For any sentence X, let n be the length of X. For $1 \leq t \leq n$, the k-length hidden state h_t with respect to parameters θ is defined by the sequence of functions:

$$h_\theta^t : X \to v_t \in R^k$$

Configuration Set: For all n, the configuration set of hidden state h_n with respect to parameters θ is given by:

$$M(h_\theta^n) = \bigcup_{|X|=n} \lim_{N \to \infty} h_{N\theta}^n(X)$$

State Complexity: The state complexity of a hidden state h_n is the cardinality of the configuration set of h_n:

$$m(h_\theta^n) = |M(h_\theta^n)|$$

These definitions enable a rigorous analysis of the memory capacity and computational power of different neural network architectures, particularly in terms of their ability to process and remember information over long sequences.

6.2 Proof Finite State Complexity of RNNs

This proof aims to establish the finite state complexity of Recurrent Neural Networks (RNNs) with sigmoid activation functions, particularly as the length of the input sequence increases. It combines a detailed analysis of the sigmoid function's impact on individual hidden state components with an examination of the overall sequence of hidden states.

RNN Structure and Sigmoid Activation The RNN is defined by the recurrence relations:

$$\begin{aligned} h_t &= \sigma(W_{hx}x_t + W_{hh}h_{t-1}) \\ y_t &= W_{yh}h_t \end{aligned}$$

where h_t is the hidden state at time t, and σ is the sigmoid activation function, defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

6.2.1 Sigmoid Function's Saturation Effect

The sigmoid function is crucial for understanding the behavior of individual components in the hidden state. It squashes its input into a range between 0 and 1. Importantly, for values of z far from 0 (either positive or negative), the output of $\sigma(z)$ approaches 1 or 0, respectively. This is known as the saturation effect.

6.2.2 Impact on Individual Hidden State Components

Consider the i-th component of the hidden state h_t, denoted $h_{t,i}$:

$$h_{t,i} = \sigma\left(\sum_{j=1}^{d_x}(W_{hx})_{ij}x_{t,j} + \sum_{j=1}^{k}(W_{hh})_{ij}h_{t-1,j}\right)$$

As the input sequence length n increases, each component $h_{t,i}$ will be influenced by increasingly large or increasingly negative values due to accumulated inputs and recurrent connections. This leads to a saturation effect where $h_{t,i}$ tends towards either 0 or 1.

6.2.3 Analysis of the Entire Sequence of Hidden States

As we consider the entire sequence $\{h_1, h_2, ..., h_n\}$, it's important to recognize the cumulative effect of inputs over time. While individual components of each h_t are saturating towards 0 or 1, the recurrent nature of the RNN also means that each h_t is influenced by all previous $h_{t'}$ for $t' < t$. This cumulative effect reinforces the binary-like behavior of each component in the sequence.

6.2.4 Finite State Complexity of the RNN

Given the saturation behavior of the sigmoid function and the cumulative effect of recurrent connections:

1. **Binary-like Behavior of Components:** Each component $h_{t,i}$ in the sequence of hidden states tends to exhibit binary-like behavior, approaching either 0 or 1.
2. **Configuration of Hidden States:** For a hidden state vector of length k, this binary-like behavior implies that the number of distinct configurations that each vector can take is bounded by 2^k.
3. **State Complexity Across the Sequence:** Considering the entire sequence $\{h_1, h_2, ..., h_n\}$, as n becomes very large, we observe a finite set of distinct configurations for each hidden state. Thus, the state complexity $m(h_\theta^n)$ of the RNN over the entire sequence is bounded by 2^k.

6.3 Proof L(RNN) = RL

We provide a semiformal description of the proof by Merill(2019) which shows the languages recognizable by RNNs are the regular languages under the following assumptions:

6.3.1 Assumptions

1. **Real-time Computation**: Each RNN iteration, denoted as a function $h_t = f_\theta(h_{t-1}, x_t)$, corresponds to one input symbol $x_t \in \Sigma$, mirroring the step-by-step processing of a DFA.
2. **Bounded Precision**: The hidden state h_t of the RNN at each time step t holds a value representable in $O(\log n)$ bits for an input sequence of length n, thereby limiting its precision.
3. **Asymptotic Acceptance**: As $N \to \infty$, the RNN's output function $\hat{1}_{N\theta} : X \to p \in (0, 1)$ converges to the indicator function 1_L of a language L.

6.3.2 Proof Outline

The proof connects RNNs under these constraints with the behavior of Deterministic Finite Automata (DFA).

6.3.3 DFA and RNN Equivalence

1. **Finite States in RNNs**: Under bounded precision, RNNs effectively have a finite number of distinct states. For each time step t, the RNN's hidden state h_t can only assume a finite set of values, akin to a DFA's finite states.
2. **State Transition in RNNs**: In real-time computation, each input symbol induces a transition in the RNN's hidden state, similar to how a DFA transitions between states on each input symbol.
3. **RNN Saturation Effect and DFA States**: Due to the sigmoid function's saturation, RNN states become increasingly binary-like with longer sequences, mirroring the deterministic nature of DFA states.

6.3.4 RNN-DFA Formalism Alignment

- DFA is defined by a finite set of states Q, a finite set of input symbols Σ, a transition function $\delta : Q \times \Sigma \to Q$, a start state $q_0 \in Q$, and a set of accept states $F \subseteq Q$.
- Under the stated assumptions, an RNN can be mapped to a DFA-like structure:
  - States in DFA $\Leftrightarrow$ Distinct configurations of RNN's hidden states.
  - Transition Function $\delta \Leftrightarrow$ RNN's transition dynamics per input symbol.
  - Accept States $F \Leftrightarrow$ RNN's output at final time step for sequence acceptance.

6.3.5 Implication

Given the finite state nature and deterministic transitions in RNNs under real-time and bounded precision constraints, these networks effectively operate as DFAs. Therefore, the class of languages recognizable by such RNNs is equivalent to the class of regular languages, formalized as L(RNN) = RL.

This proof crucially hinges on the assumptions of finite computational precision and step-wise processing, aligning RNNs closely with the fundamental properties of DFAs in language recognition.

7 Methods

7.1 Experimental Setup

The primary objective of the experiment was to investigate the asymptotic behavior of Recurrent Neural Networks (RNNs) in recognizing regular languages, particularly in the context of asymptotic weight scaling. We aimed to understand how RNNs perform as their weights are scaled by increasingly large factors. We do this to test the assumption by Merrill(2019) pertaining assymptotic acceptance which states that in the limit of parameter size (Large N), the model converges pointwise to an indicator function for a regular language.

7.1.1 Dataset Generation

The datasets were generated based on predefined regular expressions (regexes) and consisted of sequences of varying lengths. Two types of sequences were created:

1. **Matching Sequences:** Strings that conformed to the given regex.
2. **Random Sequences:** Strings composed of random characters from the regex's alphabet, not necessarily conforming to the regex.

The proportion of matching to random sequences was maintained at 50%, and the sequence lengths varied across experiments.

7.1.2 One-Hot Encoding

Each character in the sequences was encoded using one-hot encoding, where each character was represented as a binary vector of length equal to the alphabet size. This representation was necessary for processing by the RNN.

7.1.3 Model Architecture

A simple RNN model with tanh activation was employed. The model comprised an input layer, a hidden layer with a variable number of neurons, and an output layer. The output layer utilized a sigmoid activation function to produce a probability value indicating whether a given sequence matched the regex.

7.1.4 Weight Scaling

A critical aspect of the experiment was scaling the RNN's weights by factors of N as. This scaling was applied to both the recurrent and output layer weights. The experiment varied the scale factor to observe its impact on the network's ability to recognize sequences conforming to the regex.

7.1.5 Training and Testing

The RNN was trained using the binary cross-entropy loss function and Adam optimizer. The dataset was divided into training and testing subsets, with the training set used to update the model's weights and the testing set used to evaluate its performance.

7.2 Evaluation Metrics

The performance of the RNN was assessed using two primary metrics:

1. **Accuracy:** The proportion of correctly classified sequences in the test set.

2. **Loss:** The binary cross-entropy loss, representing the model's prediction error on the training set.

7.3 Experimental Conditions

Experiments were conducted under various conditions, varying in scale factor, sequence length, and hidden layer size. This diversity allowed for a comprehensive analysis of the RNN's performance across different scenarios.

7.4 Implementation Details

The experiment was implemented using the PyTorch framework. The RNN model, along with the data generation and processing functions, was written in Python. The computations were executed on a device with CUDA support to leverage GPU acceleration.

8 Results

8.1 Impact of Noise on Model Performance

The initial part of our results focuses on understanding the impact of noise in the training process of Recurrent Neural Networks (RNNs) as suggested by Merrill. For this purpose, we analyzed the model's performance on the regular expression '(00+11)*' across different configurations of sequence length and hidden size, comparing scenarios with and without noise in the initial hidden state. The results are illustrated in the heatmap depicted below:

8.1.1 Analysis of Model Accuracy

The accuracy heatmaps reveal a significant performance drop as the sequence length increases. This trend is observable in both noise and no-noise scenarios. However, the introduction of noise seems to mitigate overfitting to some extent, leading to slightly more stable accuracy across different sequence lengths and hidden sizes.

8.1.2 Analysis of Model Loss

Similarly, the loss heatmaps show a clear increase in loss values with longer sequence lengths. The presence of noise appears to have a regularizing effect, slightly lowering the loss, especially for larger hidden sizes. This suggests that noise can be beneficial in preventing overfitting, aligning with Merrill's hypothesis.

8.1.3 General Observations

- The performance degradation with increasing sequence length highlights a limitation in the RNN's ability to capture long-term dependencies, a known challenge in sequence modeling.

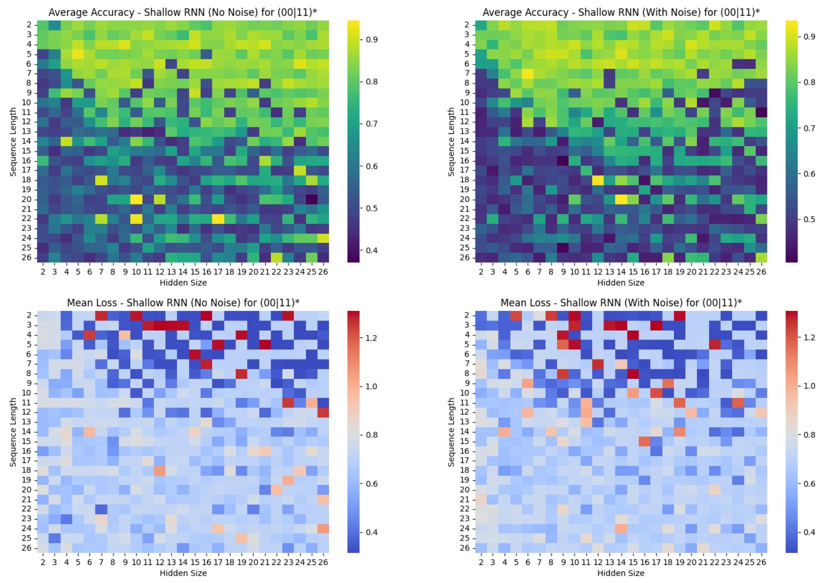

Figure 4: Heatmaps showing the effects of noise on the performance of RNNs for the regex '(00+11)*'. The sequence length is plotted along the y-axis, and the hidden size along the x-axis. The top row shows accuracy, and the bottom row shows loss. The left column represents the scenario without noise, and the right column with noise.

- The slight improvement with noise insertion aligns with the concept of noise acting as a regularizer, promoting generalization in the model.
- The effects of hidden size appear to be more nuanced, with no clear trend indicating an optimal size for balancing accuracy and loss.

8.2 Analysis of RNN Performance on Language Acceptance Tasks

In our analysis, we have focused on examining the performance of Recurrent Neural Networks (RNN) on language acceptance tasks, specifically under variations in scale factors and hidden sizes. We refer the reader to our github

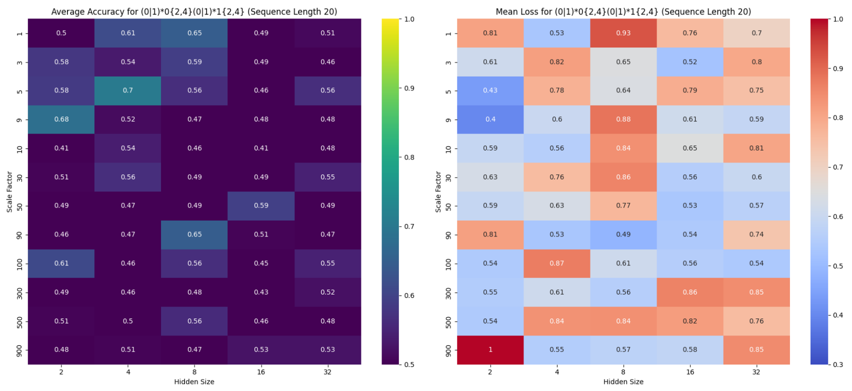

Figure 5: Accuracy and Loss for hidden size vs scale factor for a given regex

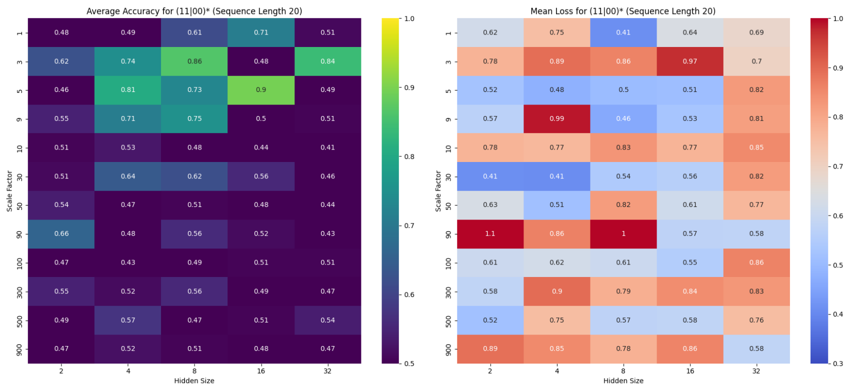

Figure 6: Accuracy and Loss for hidden size vs scale factor for a given regex

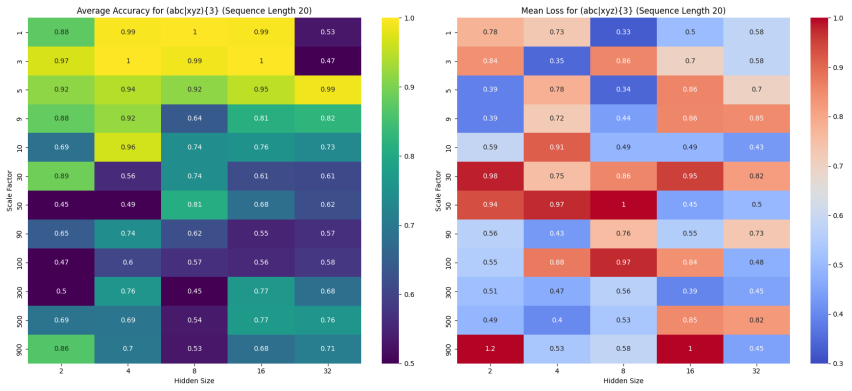

Figure 7: Performance of {`abc+xyz`}(`3`) Regex

for the full battery of tests.[2]

8.2.1 Accuracy Across Scale Factors and Hidden Sizes

We can make several observations about the accuracy:

- There is variability in accuracy across different scale factors and hidden sizes. This indicates that the model's ability to generalize the pattern depends on these parameters.
- The accuracy does not consistently improve with increasing scale factor (Big N), suggesting that simply scaling the weights does not guarantee better performance.
- Certain combinations of scale factors and hidden sizes appear to yield higher accuracy, but there is no clear trend that increasing scale factor always correlates with higher accuracy.

8.2.2 Mean Loss Across Scale Factors and Hidden Sizes

Observations regarding the mean loss include:

- Similar to accuracy, the mean loss shows variability across scale factors and hidden sizes.
- In some instances, mean loss increases with larger scale factors, which could indicate overfitting or instability due to the scaling of weights.
- For certain scale factors, particularly intermediate values, the model seems to perform better in terms of mean loss, suggesting a potential "sweet spot" in the scaling.

8.2.3 Pointwise Convergence for Big N

- The assumption of pointwise convergence under the "Big N" scaling refers to the idea that as the network's weights are scaled towards infinity, the output of the network will converge to a binary indicator function for language acceptance.
- The heatmaps do not show a consistent pattern of convergence towards better accuracy or lower loss as the scale factor increases. This might

[2]`https://github.com/Nabla7/TOg-State-Machines-and-Computability`

indicate that pointwise convergence is not occurring in practice or that the scale factors tested are not sufficient to approach the theoretical asymptotic behavior.

- Additionally, the heatmaps show that the model's performance varies significantly depending on the hidden size, which suggests that the capacity of the model also plays a crucial role in language acceptance, not just the scaling of weights.

8.2.4 Impact of Sequence Length

- For shorter sequence lengths (e.g., 5 and 10), we see higher accuracy across various hidden sizes and scale factors. As the sequence length increases, the accuracy seems to become more sensitive to the choice of hidden size and scale factor.
- The model's ability to maintain performance over longer sequences may be challenged, reflecting the increasing difficulty of the task as sequence length grows.

8.2.5 Main Takeaways Experiment

- The data suggests that the asymptotic behavior under large model parameter sizes does not universally lead to improved performance for this RNN model and task. The ideal scale factor and hidden size for optimal performance may depend on the specific characteristics of the task and the sequence length.
- These findings underscore the importance of considering both model capacity and the scaling of weights when designing neural networks for language acceptance tasks under the "Big N" assumption. The practical utility of "Big N" scaling may be limited, and further empirical analysis would be necessary to fully understand its implications for different architectures and tasks.

In summary, the heatmaps provide valuable insights but do not fully support the assumption of pointwise convergence for "Big N" in a practical, empirical context. The relationship between scale factor, hidden size, sequence length, accuracy, and loss is complex and warrants further investigation to determine the conditions under which the "Big N" assumption holds true.

Note: It is observed that the regular expression `{abc+xyz}(3)` demonstrates better performance than others, but even at large scales, its performance starts to degrade.

9 Discussion

9.1 The Complex Tradeoff in Proofs: Infinite Assumptions and Practical Realities

This paper's exploration of modern neural sequence architectures, specifically in the context of regular language recognition and Turing completeness, has consistently encountered a recurring theme: the reliance on theoretically infinite assumptions. These assumptions—infinitely precise weights, infinite recursion, and infinite context—form the bedrock of our theoretical models but also present a stark contrast to practical realities. This discussion aims to delve into these assumptions, connect them with the proofs and experiments presented, and explore their implications in real-world applications.

9.1.1 Infinite Precision of Weights

The assumption of infinite precision in weights is a cornerstone in theoretical models, particularly in demonstrating Turing completeness. This hypothesis allows models to theoretically encode an unbounded amount of information within individual weights. However, the practicality of this assumption is limited by current hardware capabilities, which restrict the precision of floating-point representations. Our experiments, while not directly testing infinite precision, highlight the challenges of balancing model complexity and computational feasibility. The asymptotic behavior observed in the experiments underscores the limitations when transitioning from theoretical models to practical applications.

9.1.2 Infinite Recursion and Context

Similarly, the assumptions of infinite recursion and context are vital in establishing the depth and breadth of a model's computational capabilities. Infinite recursion allows models to process input sequences of unbounded length, while infinite context enables the encompassing of an extensive range of dependencies and relations within the data. However, in practice, these assumptions run into limitations of computational resources and efficiency. Our experiments, particularly those involving long sequence lengths and varying hidden sizes, demonstrate the difficulty in maintaining performance as these parameters increase, reflecting the practical challenges of infinite recursion and context.

9.2 Connecting Theory and Practice

The theoretical framework of Turing completeness, grounded in infinite assumptions, provides a powerful lens to understand the potential capabilities of neural sequence architectures. Yet, the disconnect between this theoretical ideal and the empirical results of our experiments is evident. While the models demonstrate promising capabilities, they fall short of the theoretical ideal, constrained by the realities of finite precision, recursion depth, and context.

9.3 Implications and Future Directions

This exploration raises essential questions about the nature and direction of neural network research. The pursuit of theoretical models that push the boundaries of computational power is invaluable. Still, it must be tempered with an understanding of their practical limitations and applicability. The insights gained from our experiments suggest a need for a balanced approach that navigates these infinite assumptions while remaining anchored in practical realities.

Future research should focus on bridging this gap, exploring architectures and training methods that approximate the theoretical ideals under realistic conditions. Additionally, advancements in hardware and computational techniques may gradually alleviate some of the current limitations, bringing practical implementations closer to their theoretical counterparts.

9.4 Concluding Remarks

The journey from theoretical models based on infinite assumptions to practical, effective neural network architectures is complex and fraught with trade-offs. While the theoretical underpinnings provide a formal understanding of what is computationally possible, the practical application of these models requires a careful consideration of their limitations and optimizations to operate within real-world constraints. The balance between theoretical aspiration and practical application remains a dynamic and evolving frontier in the field of neural network research.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] Samuel Hahn. Theoretical limitations of self-attention in neural sequence models. *arXiv preprint arXiv:1906.06755*, 2020.

[3] John Jumper, Richard Evans, Alexander Pritzel, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583–589, 2021.

[4] Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *International Conference on Learning Representations*, Carnegie Mellon University; Microsoft Research NYC; University of Pennsylvania, 2023. ICLR.

[5] William Merrill. Sequential neural networks as automata. *arXiv preprint arXiv:1906.01615*, 2019.

[6] Javier Pérez, Emma Strubell, and Santiago Ontanón. On the turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*, 2019.

[7] Adam Roberts. Turing completeness of decoder-only transformer language models. *arXiv preprint arXiv:2305.17026*, 2023.

[8] Hava Siegelmann. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT, 1992.

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[10] Ashish Vaswani et al. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[11] Wikipedia contributors. Hyperparameter optimization — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Hyperparameter_optimization`, 2023. [Online; accessed 25-January-2023].