What Really Counts: Theoretical and Practical Aspects of Counting Behaviour in Simple RNNs

CITY
UNIVERSITY OF LONDON
EST 1894

Nadine El-Naggar

Department of Computer Science

City, University of London

Submitted in partial fulfillment of the requirements for the Degree of

Doctor of Philosophy

May 2024

Dedicated to my family, who have always motivated and supported me unconditionally.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text. This dissertation contains less than 65,000 words including appendices, bibliography, footnotes, tables and equations and has less than 150 figures. I grant powers of discretion to the City, University of London librarian to allow the dissertation to be copied in whole or in part without further reference to myself (the author). This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgment.

Nadine El-Naggar
13 May 2024

Acknowledgements

This PhD has been an incredible learning experience, and I would like to express my utmost gratitude to the wonderful people who have encouraged and supported me every step of the way.

First and foremost, I would like to thank my supervisors Dr Tillman Weyde and Dr Pranava Madhyastha. I am extremely grateful for your unwavering support and invaluable guidance. This PhD would not have been possible without your impressive knowledge, enthusiasm, and mentorship. I have learned so much from working with you, and you have helped me grow into the researcher I am today. You are, and will continue to be, a great inspiration to me.

I would also like to thank my viva examiners Prof Artur Garcez and Prof Guillaume Rabusseau for taking the time to evaluate my thesis. Your comments, constructive feedback and interesting discussion during the viva were very insightful. A big thank you to Dr Ernesto Jimenez-Ruiz for chairing my viva.

To my lab mates, colleagues and friends at City, who have shared the many ups and downs of this PhD with me, thank you for the amazing time we spent together and the friendships we have made. Because of you, my PhD has been a fun and memorable experience that I will look back on fondly.

Last but certainly not least, I would like to express my sincerest thanks to my family and friends. Thank you for always believing in me, and supporting me through the best and toughest of times. You have celebrated my successes, and patiently motivated me to overcome challenges. You have been a constant reminder that I have an incredible support system. I am eternally grateful to have you in my life.

Abstract

Recurrent Neural Networks (RNNs) are commonly used in sequential tasks and have been shown to be Turing complete, and are therefore theoretically capable of computing any task if the correct configuration is used. However, there is a long standing debate on the systematicity of NN learning. There has recently been an increased interest in the abilities of RNNs to learn different systematic tasks such as counting. In this thesis, we develop a better understanding of RNN learning of counting behaviour.

We conduct experiments to evaluate the learning and generalisation of counting behaviour with different commonly used RNN models, using different initialisations and configurations. We formalise counting as Dyck-1 acceptance and focus on generalisation to long sequences. We find that in general RNN models do not learn to count exactly and eventually fail on longer sequences. We also find that weights correctly initialised for Dyck-1 acceptance are unlearned during training. Analysing the results, we find different failure modes for different models.

For single-cell linear RNNs and ReLU RNNs, we propose two theorems, where we establish Counter Indicator Conditions (CICs) on the weights of the model that result in exact counting behaviour. We formally prove that the CICs are necessary and sufficient for exact counting to be realised. However, we find in experiments that the CICs are not found and are even unlearned by correctly initialised models. In experiments on ReLU RNNs, we find that there is mismatch between the task and the loss function such that the correct model does not coincide with the minimal loss value. This indicates that gradient descent based optimisation is unlikely to reach exact counting behaviour with a standard setup.

As an alternative approach, we develop a discrete non-negative counter module to

use with RNNs. To allow for training with backpropagation, we equip the discrete non-negative counter with artificial gradients. The discrete non-negative counter is implemented and unit-tested. We find in experiments that RNNs equipped with this module achieve improved performance over the standard RNN models.

In this thesis, we address the ability of different Recurrent Neural Networks to learn counting behaviour. In the first part we conduct experiments and find that when RNNs learn to count they eventually fail to generalise to arbitrarily long sequences. In the second part, we propose and prove two theorems establishing Counter Indicator Conditions (CICs) in Recurrent Neural Networks. In our experiments, we find that the CICs are not reached by the models during training. We investigate the learning dynamics and find that there is a mismatch between the CICs and the minimum of the loss function. In part 3 we then propose a discrete non-negative counter module, which we design, implement and integrate into RNNs. The integration of the discrete non-negative counter module results in improved performance over our baseline models.

Contents

List of Figures

List of Tables

List of Abbreviations

BB	Balanced Bracket Language
ADT	Abstract Data Type
AI	Artificial Intelligence
BCE	Binary Cross Entropy
BPTT	Backpropagation Through Time
CE	Cross Entropy
CFG	Context Free Grammar
CIC	Counter Indicator Condition
DFA	Deterministic Finite Automata
DNC	Differentiable Neural Computer
DNNC	Discrete Non-Negative Counter
FFNN	Feed-Forward Neural Network
FPF	First Point of Failure
FSA	Finite State Automaton
GCM	General Counter Machine
GRU	Gated Recurrent Unit
LRN	Linear Recurrent Network

LSTM	Long Short Term Memory
MemNN	Memory Neural Network
MSE	Mean Squared Error
NALU	Neural Arithmetic Logic Unit
NLP	Natural Language Processing
NN	Neural Network
NNPDA	Neural Network Pushdown Automata
NTM	Neural Turing Machine
PDA	Pushdown Automata
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RTRL	Real-Time Recurrent Learning
SGD	Stochastic Gradient Descent
SINC	Stateless Incremental Non-Negative 1-Counter Machine
vv-WFA	Vector Valued Weighted Finite Automata
WFA	Weighted Finite Automata
WFSA	Weighted Finite State Automata

Chapter 1

Introduction

1.1 Motivation

Neural Networks (NNs) are the state-of-the-art tools used for many Artificial Intelligence (AI) tasks. Recurrent Neural Networks (RNNs) are highly expressive and have been shown to be Turing complete with both sigmoid and Rectified Linear Unit (ReLU) activation functions (Chen et al., 2018; Siegelmann and Sontag, 1992). This means that they can be used to solve any computable task when the correct configuration is used. Feed-Forward Neural Networks (FFNNs) are universal approximators (Cybenko, 1989; Funahashi and Nakamura, 1992), this is also true for RNNs (Schäfer and Zimmermann, 2006). RNNs have been heavily used for sequential tasks and achieved tremendous success. Although Transformers (Vaswani et al., 2017) have become increasingly popular in Natural Language Processing (NLP) applications, RNNs are also extensively used.

Despite the success of NNs, there is a long-standing debate on the ability of NNs to learn and generalise systematically (Fodor and Pylyshyn, 1988). Work by Marcus et al. (1999) shows that NNs fail to learn abstract patterns that 7-month-old infants learn in minutes. There have been different tests developed to evaluate systematicity in NN models, namely SCAN (Lake and Baroni, 2018) and BIG-Bench (Srivastava et al., 2022).

For a long time, there has been an interest in evaluating counting behaviour in RNNs. Counting is a systematic task that plays a part in many different aspects of language understanding (Gelman and Gallistel, 2004; Le Corre et al., 2006) and mathematical reasoning (Crollen et al., 2011; Domahs et al., 2010). It is a discrete process that plays a role in several sequential tasks (Rodriguez and Wiles, 1997; Trott et al., 2017). Counting is often formalised as formal language acceptance, most commonly a^nb^n (e.g. Gers and Schmidhuber (2001); Hölldobler et al. (1997); Rodriguez (2001)) and Dyck-1 (e.g. Bernardy (2018); Hewitt et al. (2020); Sennhauser and Berwick (2018); Suzgun et al. (2019a)). Since there is a well developed theory of formal languages and automata, drawing connections between RNNs and formal automata can help us develop a better understanding of RNNs. This is the approach we use to address counting behaviour in RNNs, using Dyck-1.

In the 1990s and early 2000s, beyond the theoretical work by Siegelmann and Sontag (1992), Hölldobler et al. (1997), and Kalinke and Lehmann (1998), there was mainly an interest in empirical studies on counting in RNNs (Bodén and Wiles, 2000, 2002; Elman, 1991; Gers and Schmidhuber, 2001; Rodriguez and Wiles, 1997; Wiles and Elman, 1995, inter alia). There has recently been a resurgence of interest in theoretical (Merrill, 2019; Weiss et al., 2018a, inter alia) and empirical (Suzgun et al., 2019a, inter alia) studies in counting behaviour in RNNs. By studying RNN behaviour from both empirical and theoretical perspectives we can bridge the gap between empirical and theoretical findings which can allow for the development of better solutions.

In most empirical studies, RNNs are trained with backpropagation (Rumelhart et al., 1986), which requires that the models are continuous and differentiable. The extent to which RNNs systematically learn exact counting behaviour via backpropagation and generalise to arbitrarily long sequences is unclear, and likely limited. To our knowledge, the weight configurations for RNNs that lead to exact counting behaviour, and consequently, perfect generalisation, have not previously been determined. Although alternative training regimes have been used to learn and generalise counting behaviour in RNNs (Lan et al., 2022; Mali et al., 2021), backpropagation is still the most commonly used training regime.

In this thesis, we address the systematic learning of counting behaviour in RNNs with backpropagation. We use 3 different approaches. The first approach we use is an empirical approach where we train and test RNN models on the Dyck-1 language. We find that the models do not learn counting behaviour that generalises to arbitrarily long sequences. We also find that models initialised with the correct weights deviate from these weights during training. The second approach is a theoretical approach, where we relate linear RNNs and ReLU RNNs to formal automata. We propose and prove two theorems where we define Counter Indicator Conditions (CICs) on their weights, which result in counting behaviour to arbitrarily long sequences. The third approach is constructive. We design and implement a Discrete Non-Negative Counter (DNNC) module. We equip the DNNC with artificial gradients to ensure that the DNNC can be used with RNNs trained with backpropagation. We evaluate the effect of using the DNNC in RNNs and observe an improvement over the baseline standard RNN models.

1.2 Research Questions, Objectives and Contributions

1. To what extent can RNNs be trained to count and generalise effectively beyond training data?

   Objectives:

   (a) Evaluate the ability of different RNN models to learn counting behaviour and effectively accept the Dyck-1 language.

   (b) Use different weight initialisations and activation functions and evaluate their effect on model performance.

   (c) Evaluate the limits of generalisation of trained RNN models using different longer strings and analyse the behaviour and failure of the models.

   Contributions:

   (a) We provide empirical results showing that RNNs generally fail to learn counting behaviour and therefore fail to generalise to longer strings.

   (b) We provide empirical evidence that RNN models struggle to learn counting behaviour and that RNN models correctly initialised for counting deviate from their initialisation when trained with the standard classification setup but not with a regression setup.

   (c) We evaluate the behaviour of RNN models and observe different failure modes for the different RNN models.

2. Under what conditions do RNNs count exactly?

   Objectives:

   (a) Theoretically identify counting conditions for some RNN models

   (b) Empirically evaluate the learning of these counting conditions by RNN models

   Contributions:

   (a) We provide two theorems where we determine Counter Indicator Conditions on the weights of linear and ReLU RNNs to indicate counting behaviour and prove that the Counter Indicator Conditions are necessary and sufficient

   (b) We find that the Counter Indicator Conditions are not found by the models in training, and upon further investigation find that with the dataset used the conditions are not located at the minimum loss value.

3. Can we design a discrete counter that can be integrated into RNNs trained with backpropagation and what is the effect of using it?

   Objectives:

   (a) Design and implement a discrete counter module in PyTorch with artificial gradients to integrate counting behaviour into NNs

   (b) Integrate the discrete non-negative counter module into RNN models and evaluate the effect of the module on the model performance and generalisation.

   Contributions:

   (a) We provide a discrete non-negative counter module with artificial gradients

   (b) We provide empirical results which show improvements in model performance with this module

1.3 Publications

1.3.1 Published Papers

The following publications are a direct result of this work:

- (El-Naggar et al., 2022a) El-Naggar, N., Madhyastha, P., Weyde, T. (2022, October). Experiments in Learning Dyck-1 Languages with Recurrent Neural Networks. In: Proceedings of the 3rd Human-Like Computing Workshop (Vol. 3227, pp. 24-28).
  Oral talk and poster.

- (El-Naggar et al., 2022b) El-Naggar, N., Madhyastha, P., Weyde, T. (2022, December). Exploring the Long-Term Generalization of Counting Behavior in RNNs. In: *I Can't Believe It's Not Better Workshop: Understanding Deep Learning Through Empirical Falsification* at NeurIPS 2022.
  Poster.

- (El-Naggar et al., 2023a) El-Naggar, N., Madhyastha, P. S., Weyde, T. (2023, May). Theoretical Conditions and Empirical Failure of Bracket Counting on Long Sequences with Linear Recurrent Networks. In: Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop (pp. 143-148).
  Poster.

- (El-Naggar et al., 2023b) El-Naggar, N., Ryzhikov, A., Daviaud, L., Madhyastha, P., Weyde, T. (2023, July). Formal and Empirical Studies of Counting Behaviour in ReLU RNNs. In: International Conference on Grammatical Inference (pp. 199-222). PMLR.
  Oral talk.

1.3.2 Presentations

- Poster presented at ALPS winter school (January 2021).
- Oral presentation at City's Research Vignettes (April 2021).

- Invited talk at Mohamed bin Zayed University of Artificial Intelligence (November 2023)

1.4 Thesis Outline

The remainder of this thesis details the work that has been done throughout this PhD. In Chapter 2, we describe the fundamental concepts and datasets that we use throughout the thesis. The remainder of the thesis is divided into 3 parts, where each part covers one approach we use to address counting behaviour in RNNs.

Part I focuses on the empirical analysis of counting behaviour using standard RNN models, this consists of Chapters 3,4 and 5. Chapter 3 provides context for the remaining chapters of this part. In Chapter 4, we evaluate the effect of different configurations and hyperparameters on the learning of counting behaviour in RNNs. In Chapter 5 we focus on evaluating the generalisation of counting in RNNs on very long sequences.

Part II focuses on theoretical understanding of linear and ReLU RNNs. Part II is made up of Chapters 6,7 and 8. We provide context for the remaining chapters of Part II in Chapter 6. Chapter 7 provides the theorem with proof and experiments for linear RNNs. Chapter 8 provides the theorem with proof and experiments for ReLU RNNs.

In Part III we describe our proposed solution of a discrete non-negative counter module (DNNC). Chapter 9 provides context and describes relevant related prior work. In Chapter 10, the discrete non-negative counter module is introduced and described. We integrate the DNNC into RNN models and conduct experiments in Chapter 11.

We conclude this thesis with Chapter 12 where we discuss our findings, the resulting implications, and propose ideas for future work.

Chapter 2

Background and Terminology

In this chapter, we introduce the concepts that we use throughout the thesis. We introduce the different NNs we use, specifically the different RNN types, and the activation and loss functions. We then describe formal languages, particularly the Dyck-1 language. We also introduce the counter machine terminology by Merrill (2020) which we use in Part II. Finally, we introduce the different datasets we use in our experiments.

2.1 Neural Networks

Neural Networks (NNs) are the building blocks for modern AI applications. There are different types of NNs, each of which is better suited to a set of different tasks. The simplest type of NN is a Feed-Forward Neural Network (FFNN), which processes all data it receives in one timestep. Recurrent Neural Networks (RNNs) were designed to process sequential data over multiple timesteps and allow for intermediate access of results. There are different types of RNNs, Simple RNNs, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU).

NNs are normally trained using gradient descent. The training tries to optimise the parameters of a model. This is done by comparing the predicted value with the ground truth using a function of their differences in order to try to eventually produce a predicted value that is as close as possible to the ground truth by applying the chain rule. The chain rule is applied to the components of the model iterating backward from the last layer and updating the parameters, this is known as backpropagation. In RNNs, backpropagation happens for each of the individual timesteps to update the model parameters, and this is known as backpropagation through time (BPTT).

We focus on Recurrent Neural Networks (RNNs), which differ from Feed-Forward Neural Networks in that they pass data from one timestep to the next, and are better suited to sequential data. Before the introduction of transformers (Vaswani et al., 2017), RNNs were the most commonly used tool for NLP applications. Transformers are not the focus of this thesis, as we are interested in the incremental differences between timesteps when a RNN is assigned a counting task. We first describe FFNNs because they are a building block for RNNs. We then describe the different RNN models and outline the differences.

2.1.1 Feed-Forward Neural Networks (FFNNs)

Feed-Forward Neural Networks (FFNNs) are the simplest NN models. The data only flows in one direction, and the connections between the nodes of a FFNN do not form a cycle. FFNNs consist of an input layer, hidden layers and output layer.

- **Input Layer:** The input layer consists of neurons that pass the input to the hidden layer. The input layer has no trainable parameters.

- **Hidden Layer:** The hidden layer applies the weights, biases and activation function to the input values and passes the result to the output layer to produce a label. The hidden layer can consist of multiple layers, where each of them applies its own weights, biases and a typically non-linear activation function. The weights and biases in the hidden layers are trainable and are updated during the backpropagation stage of training.

- **Output Layer:** The output layer receives the output of the last hidden layer and produces the output label. The output layer consists of a single layer of neurons which apply their own weights, biases and activation function. The weights and biases of the output layer are trainable, and updated during the backpropagation stage of training.

Trainable layers of a FFNN have the following update function:

$$y = A(Wx_t + W_b) \tag{2.1}$$

where:

- x_t is the input value at time t,
- W is the matrix of weights applied to the model input value x_t.
- W_b is the bias value,
- y is the output value,
- A is the activation function.

The activation function A could be one of a number of non-linearities, such as ReLU, sigmoid, softmax and tanh among others. We describe the different activation functions in 2.2.1.

FFNNs with sigmoid and ReLU activation functions are known to be universal approximators, which means that they can approximate any function with a large enough network (Hornik et al., 1989; Leshno et al., 1993).

In order to process a sequence using a feed-forward neural network, the entire sequence would have to be input at once, not token by token. Recurrent neural networks are designed to handle sequential data, by passing data from one timestep to the next.

2.1.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are descendents of FFNNs that pass data from one timestep to the next. They are designed to handle sequential data over multiple timesteps. For example, they can process a string of text one token at a time and provide the output of the intermediate steps of the hidden layers. They inherit the universal approximation property from FFNNs (Cybenko, 1989; Funahashi and Nakamura, 1992; Leshno et al., 1993). They are Turing complete (Siegelmann and Sontag, 1992), and can therefore, in theory, be used to compute any task if adequate weights and biases are used. There are different types of RNNs:

- Simple RNNs
- Gated Recurrent Units (GRUs)
- Long Short Term Memory (LSTM)

Simple RNNs

Simple RNNs are the first RNNs to be introduced. A Simple RNN is an extension of FFNN that is fed an additional input, which is the output of the previous timestep (h_{t-1}). This additional input also goes through a dedicated weight matrix (U) and is part of the update equation. Simple RNNs have the following update equation:

$$h_t = A(Wx_t + Uh_{t-1} + W_b) \tag{2.2}$$

where:

- x_t is the input value.
- W is the matrix of weights applied to the model input value x_t.
- W_b is the bias value.
- y is the output value.
- A is the activation function.
- h_t is the recurrent value that is propagated from one timestep to the next.
- U is the weight matrix applied to the recurrent value h.

Different types of Simple RNNs have different activation functions:

- **Elman RNNs:** Have a tanh activation function, and hence have the following update equation:
$$h_t = \tanh(Wx_t + Uh_{t-1} + W_b) \tag{2.3}$$
- **ReLU RNNs:** Have a ReLU activation function, and the following update equation:
$$h_t = max(0, Wx_t + Uh_{t-1} + W_b) \tag{2.4}$$

- **Linear RNNs:** Have no activation function, and the following update function:

$$h_t = Wx_t + Uh_{t-1} + W_b \tag{2.5}$$

Simple RNNs suffer from vanishing and exploding gradients and have difficulty learning sequences with long term dependencies (Bengio et al., 1994; Hochreiter, 1991; Hochreiter et al., 2001), and this has been theoretically verified by Pascanu et al. (2013). During backpropagation, the gradients of the recurrent component of Simple RNNs go through a number of matrix multiplications, which results in the gradients growing (exploding) or shrinking (vanishing) exponentially. This makes them hard to train and results in them failing to learn long-term dependencies. LSTM and GRU networks were proposed to remedy the problem of vanishing and exploding gradients.

Long Short Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks were developed in 1997 by Hochreiter and Schmidhuber (1997) to remedy the vanishing and exploding gradient problem in Simple RNNs. LSTMs are equipped with three gates and a second recurrent value called the context value (c_t) which contains additional information from previous timesteps that is strategically filtered and updated through the different gates. This makes LSTMs capable of learning long term dependencies. The LSTM gates are:

- **Forget gate:** The forget gate uses the input x_t and the recurrent output from the previous timestep h_{t-1} and produces a value between 0 and 1 for each value in the context. This is executed in Equation 2.6. This value f_t is then multiplied componentwise by the context. This way, the values of the context which are multiplied by 1 are completely retained and those multiplied by 0 are completely forgotten.

- **Input gate:** The input gate determines which parts of the input will be added to the context. It uses the input x_t and the recurrent output from the previous timestep h_{t-1} and produces a value between 0 and 1 for each value in the context.

This is executed in Equation 2.7. This value is then multiplied by a candidate value $\tilde{c_t}$ (Equation 2.9) then used to update the context.

- **Output gate:** The output gate determines how much of the context value is included in the recurrent output to the next timestep (h_t). It produces a value between 0 and 1 based on the input value x_t and the recurrent output from the previous timestep (h_{t-1}). This is executed in Equation 2.8.

An LSTM has the following update equations:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + W_{b_f}) \tag{2.6}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + W_{b_i}) \tag{2.7}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + W_{b_o}) \tag{2.8}$$

$$\tilde{c_t} = \tanh(W_c x_t + U_c h_{t-1} + W_{c_f}) \tag{2.9}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c_t} \tag{2.10}$$

$$h_t = o_t \circ \tanh(c_t) \tag{2.11}$$

where:

- W_f, W_i, W_c and W_o are the weight matrices applied to input x_t to calculate f_t, i_t, $\tilde{h_t}$ and o_t, respectively.

- U_f, U_i, U_c and U_o are the weight matrices applied to recurrent input h_{t-1} to calculate f_t, i_t, $\tilde{c_t}$ and o_t, respectively.

- W_{b_f}, W_{b_i}, W_{b_c} and W_{b_o} are the bias values used to calculate f_t, i_t, $\tilde{c_t}$ and o_t, respectively.

- h_t and c_t are the recurrent values passed from one timestep to the next.

Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) were proposed in 2014 by Cho et al. (2014) as another RNN with gating mechanisms to remedy the vanishing and exploding gradient problem. GRUs consist of only 2 gates, unlike LSTMs which have 3. GRUs have an update gate z_t (Equation 2.12) and a reset gate r_t (Equation 2.13), but do not have an output gate or context value passed from one timestep to the next. These gates retain the relevant information and remove the irrelevant information, each in its own way:

- **Update gate:** applies weight matrices W_z and U_z to input x_t and h_{t-1}, respectively then squashes the result between 0 and 1 using a sigmoid function. This helps the model determine how much of the previous data to retain and pass to future timesteps.

- **Reset gate:** This gate helps the model decide how much of the previous data to forget. Similar to the update gate, weight matrices are applied to x_t and h_{t-1} and the result is squashed between 0 and 1 with a sigmoid.

An intermediate memory step is used to keep relevant information from previous timesteps by applying the result of the reset gate (Equation 2.14). Finally, the model combines the relevant information from both previous and current timesteps and produces the output (Equation 2.15).

A GRU has the following update equations:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + W_{b_z}) \tag{2.12}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + W_{b_r}) \tag{2.13}$$

$$\tilde{h_t} = \tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + W_{b_h}) \tag{2.14}$$

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h_t} \tag{2.15}$$

where:

- W_z, W_r and W_h are the weight matrices applied to input x_t to calculate z_t, r_t and $\tilde{h_t}$, respectively.

- U_z, U_r and U_h are the weight matrices applied to recurrent input h_{t-1} to calculate z_t, r_t and $\tilde{h}_t$, respectively.
- W_{b_z}, W_{b_r} and W_{b_h} are the bias values used to calculate z_t, r_t and $\tilde{h}_t$, respectively.
- h_t is the recurrent value passed from one timestep to the next.

2.2 Activation and Loss Functions

Activation functions are applied to the output of a weight matrix in recurrent and feed-forward networks. Loss functions are used to calculate the error between the ground truth and the value predicted by a model. Both activation and loss functions are a fundamental part of neural network learning.

2.2.1 Activation Functions

Activation functions are used in the hidden and output layers of NN models. They are applied to the output of a weight matrix, and are usually nonlinear functions. There are a variety of activation functions, some which are bounded, and others which are not. Bounded or squashing activation functions map or squash their inputs to a value found in a limited range. Unbounded or non-squashing activation functions do not squash the input values to a limited range. We describe the following activation functions:

Definition 2.1. (Sigmoid Activation Function) A sigmoid activation function σ is a bounded activation function that maps the scalar input it receives to a positive real number between 0 and 1.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.16}$$

where e is Euler's number.

Definition 2.2. (Softmax Activation Function) A softmax activation function is a bounded activation function that maps a vector K of real values to a probability distribution over K possible outcomes.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, ..., K \text{ and } z = (z_1, ..., z_K) \in \mathbb{R}^K \tag{2.17}$$

where e is Euler's number.

Definition 2.3. (Tanh Activation Function) A tanh activation function restricts its input x to a value between -1 and 1. Similarly to sigmoid and softmax activation functions, tanh is a bounded activation function.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.18}$$

where e is Euler's number.

Definition 2.4. (ReLU Activation Function) A ReLU activation function rectifies any negative input it receives to 0, and outputs any positive inputs it receives unchanged.

$$ReLU(x) = max(0, x) \tag{2.19}$$

A ReLU activation function is an unbounded activation function.

Definition 2.5. (Clipping Activation Function) The Clipping activation function restricts the input it receives to a range between 0 and 1.

$$Clipping(x) = min(1, max(0, x)) \tag{2.20}$$

2.2.2 Loss Functions

Loss functions are used to calculate the error in a model's prediction. The error is calculated between the prediction and the ground truth label. The loss value is then used in backpropagation to apply the chain rule and tweak the model parameters. We use 2 different loss functions in the thesis, Mean Squared Error (MSE) and Cross Entropy (CE)/Binary Cross Entropy (BCE).

Definition 2.6. (Mean Squared Error (MSE) Loss Function) This loss function calculates the square of the difference between ground truth value Y and predicted value $\hat{Y}$ for a vector of n predictions.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y - \hat{Y})^2 \tag{2.21}$$

Definition 2.7. (Cross Entropy (CE) Loss Function) This loss function measures the true probability distribution and the probability distribution predicted by ML models.

$$CE = -\sum_{i=1}^{k} t_i \log(p_i) \tag{2.22}$$

for k classes, where t_i is the ground truth and p_i is the softmax probability for the i^{th} class.

Definition 2.8. (Binary Cross Entropy (BCE) Loss Function) This loss function is the version of the Cross Entropy loss function that is used in scenarios where there are only 2 possible outcomes.

$$BCE = -\sum_{i=1}^{2} t_i \log(p_i) = -[t \log(p) + (1-t) \log(1-p)] \tag{2.23}$$

2.3 Formal Languages

We are interested in the ability of RNN models to count and learn formal languages, specifically the Dyck-1 language. Here, we describe formal languages in general and also the Dyck-1 language.

A formal language L is a set of strings made up of symbols from a finite alphabet Σ, where Σ^* is the set of all possible strings over alphabet Σ, and $L \subseteq \Sigma^*$. A formal language can be specified by a specific set of rules that generates the language, like a grammar, or by a machine that accepts the language. A grammar characterises which strings of Σ^* are in L and which are not. Machines are said to accept a language if they distinguish between the elements of Σ^* that belong to language L and the elements that do not. The Chomsky hierarchy (Chomsky, 1956) is a containment hierarchy that identifies formal languages as regular, context-free, context-sensitive or recursively enumerable based on the language structure. Regular languages are the most restricted, while recursively enumerable languages are the most general. We focus on Dyck-1, which is a context-free language. Context free languages are accepted by pushdown automata.

2.3.1 The Dyck-1 Language

Dyck-n languages are strings of well formed parentheses with n different types of brackets. In Dyck languages, a corresponding closing bracket exists for every opening bracket in a Dyck string. However, the closing brackets cannot occur before their corresponding opening brackets. In the case of Dyck-n for $n > 1$, the different types of brackets cannot be crossed.

We focus on Dyck-1 which consists of only one type of bracket, $\Sigma = \{\langle, \rangle\}$. In traditional computer science, Dyck-1 strings can be parsed using a pushdown automaton or stack ADT, where an opening bracket is pushed to the stack and a closing bracket pops the stack. It is possible to cast these automata as Minsky machines in the case of Dyck-1, which are monosymbolic pushdown automata. We can further simplify the required automata to counters, since it is not necessary to store any opening brackets on the stack, but only to keep track of the number of unmatched opening brackets.

2.4 Counter Machines

In this work, we focus on the ability of RNNs to systematically learn counting behaviour to accept the Dyck-1 language. There has been a recent interest in the abilities of RNNs to learn counting behaviour, and establishing a connection between RNNs and automata in general. However, the Dyck-1 language can be accepted by a counter, so we relate RNNs to counter machines. Counter machines were formalised by Fischer et al. (1968). They have been later revisited by Merrill (2020), and we use the same notation as Merrill (2020) in our work.

Definition 2.9. (General Counter Machine) A k-Counter is a tuple $\langle \Sigma, Q, q_0, u, \delta, F \rangle$ with:

1. A finite alphabet Σ

2. A finite set of states Q

3. An initial state q_0

4. A counter update function

$$u : \Sigma \times Q \times \{0, 1\}^k \to (\{+m : m \in \mathbb{Z}\} \cup \{\times 0\})^k$$

5. A state transition function

$$\delta : \Sigma \times Q \times \{0, 1\}^k \to Q$$

6. An acceptance mask

$$F \subseteq Q \times \{0, 1\}^k$$

The counter machine computation is formalised in Definition 2.10. The finite mask of the current state is created using a zero-check function $z(\mathbf{v})$ for a vector v, where:

$$z(\mathbf{v})_i = \begin{cases} 0, & \text{if } v_i = 0 \\ 1, & \text{otherwise} \end{cases} \tag{2.24}$$

Definition 2.10. (Counter Machine Computation) Let $\langle q, \mathbf{c} \rangle \in Q \times \mathbb{Z}^k$ be a configuration of machine M. Upon reading input $x_t \in \Sigma$, we define the transition

$$\langle q, \mathbf{c} \rangle \to_{x_t} \langle \delta(x_t, q, z(\mathbf{c})), u(x_t, q, z(\mathbf{c}))(\mathbf{c}) \rangle$$

This definition can naturally be extended to sequences $s = [s_1, ..., s_l] \in \Sigma^*$, with counter machine M such that:

$$\langle q, \mathbf{c} \rangle \to_s^M \langle q_s, \mathbf{c}_s \rangle :=$$

$$\langle q, \mathbf{c} \rangle \to_{s_1} \langle q_1, \mathbf{c}_1 \rangle \to_{s_2} ... \to_{s_l} \langle q_s, \mathbf{c}_s \rangle$$

Definition 2.11. (Real-Time Acceptance) For any string $x \in \Sigma^*$ with length n, a counter machine accepts x if there exist states $q_1, ..., q_n$ and counter configurations

$\mathbf{c}_1, ..., \mathbf{c}_n$ such that

$$\langle q_0, 0\rangle \rightarrow_{x1} \langle q_1, \mathbf{c}_1\rangle \rightarrow_{x_2} .. \rightarrow_{x_n} \langle q_n, \mathbf{c}_n\rangle \in F.$$

Definition 2.12. (Real-Time Language Acceptance) A counter machine accepts a language L if, for each $x \in \Sigma^*$, it accepts x iff $x \in L$.

Definition 2.13. (Stateless Counter Machine) A counter machine is stateless if $Q = \{q_0\}$.

An input string x is processed by the counter one token x_t at a time.

2.5 Summary

In this chapter, we present the different concepts that are used in this thesis. We describe the RNN models that we use, as well as the different activation and loss functions. We describe the Dyck-1 language and the counter machine terminology that is used later on in the thesis.

Part I

Experiments Using Standard RNNs

Chapter 3

Empirical Analysis: Context

In this part, we address the first of the 3 overarching questions stated in Chapter 1: to what extent can RNNs be trained to count and generalise effectively beyond the training data? We address counting in terms of formal language acceptance, specifically Dyck-1, as already introduced in section 2.3.1. We motivate the research in this part in section 3.1. RNNs and formal languages have been studied for some time, as discussed in section 3.2, while prior work on RNNs' capacity to count is discussed in section 3.3. While RNNs can be configured to count, the result of learning is normally not exact which can be tested by long sequences using different sequences structures and tasks as outlined throughout the thesis. We discuss the literature on RNNs learning to count in section 3.4, which motivates our experimental setup in Chapters 4 and 5. We also discuss some alternative methods to backpropagation that people have used to train RNNs to count in section 3.5. The literature discussed in section 3.4 is directly relevant to the work discussed in this part, while the contents of the other sections provide more general background information.

3.1 Motivation

Recurrent Neural Networks (RNNs) are Turing complete with both simgoid and ReLU activation functions (Chen et al., 2018; Siegelmann and Sontag, 1992) and thus capable of computing any function if appropriately configured. RNN models are often challenging to interpret, and drawing connections between RNNs and well defined theoretical concepts, such as context-free grammars and formal automata, can help us understand the behaviour or RNNs.

Counting is a discrete process that plays a role in numerous different tasks. It can be evaluated using formal languages that can be accepted by counting, such as $a^n b^n$ and Dyck-1. According to the Chomsky-Schützenberger theorem (Chomsky and Schützenberger, 1959), a context-free language L can be represented by a structural description that is the intersection of a Dyck language and a regular language and a homomorphism from the structural description to the words. Thus, Dyck languages characterise an essential aspect of context-free languages. Dyck-1 is the simplest Dyck language, and we frame counting as Dyck-1 acceptance. Therefore,counting can thus contribute to understanding the acceptance of context-free languages with RNNs.

In most empirical studies, such as Wiles and Elman (1995), Rodriguez (2001) and Gers and Schmidhuber (2000), RNNs are trained via backpropagation (Rumelhart et al., 1986), where the RNNs are required to be continuous and differentiable. The extent to which RNNs learn to count via backpropagation and generalise to longer sequences has not yet been determined, and is expected to be limited. We are interested in evaluating the extent to which different RNNs learn to count with backpropagation and generalise to arbitrarily long sequences. We are also interested in the behaviour of the different models when tested on longer sequences.

3.2 Relating RNNs to Formal Automata and Formal Languages

In theoretical computer science, formal automata are abstract machines that are used to mathematically define and describe computations of machines. Formal languages are syntactic rules that apply to sets of strings. Formal languages are accepted by formal automata. A relationship between formal languages and formal automata has been defined in the Chomsky hierarchy (Chomsky, 1965), where each class of formal language is accepted by a specific formal automaton. Recently, a class of counter languages has been defined by Delétang et al. (2022), which includes formal languages that can be accepted by counter automata from different levels of the Chomsky hierarchy. Some context-free languages, such as a^nb^n and Dyck-1 are counter languages, and therefore a counter can also be used to accept the language.

There are different ways to relate RNNs to formal automata and formal languages. Both theoretical and empirical approaches have been commonly used. Empirical approaches usually involve training and testing RNNs on a formal language acceptance task to evaluate the extent to which RNNs learn to behave like the formal automaton that can accept this language. This is the approach used by Elman (1991), who evaluate the ability of Elman RNNs (Elman, 1990) to learn to behave like pushdown automata to represent sentences with hierarchical structures.

There are also different theoretical approaches to relate RNNs to formal automata. The expressive capacity of RNNs can be determined by mathematically proving the relationship between the inner workings and update functions of RNNs and formal automata, e.g. Peng et al. (2018) define the concept of rational recurrences and Merrill et al. (2020) categorise different RNNs into a hierarchy. Peng et al. (2018) define a rationally recurrent RNN as one whose update function can be represented by a weighted finite state automaton. Following the concept of rational recurrences, Merrill et al. (2020) construct their hierarchy of RNN architectures by determining the rational recurrence and space complexity of the RNN models, and categorising them accordingly. Alongside their definition of counter languages, which we mentioned previously, Delétang et al. (2022) also categorise different NN models into a hierarchy. However, instead of defining their own hierarchy like Merrill et al. (2020), they determine the corresponding levels for feed-forward, recurrent and Transformer (Vaswani et al., 2017) models on the Chomsky hierarchy.

To study the capacity of simple RNNs and LSTMs to express context-free grammars, Hewitt et al. (2020) follow the Chomsky-Schutzenberger theorem (Chomsky and Schützenberger, 1959), which states that any context-free language can be expressed using a combination of a Dyck language with a regular language. Context-free grammars are typically accepted by pushdown automata (Chomsky, 1956). Pushdown automata are finite automata augmented with a stack, and can be used to capture the hierarchical structure of language. Hewitt et al. (2020) mathematically show that simple RNNs and LSTMs can emulate stack-like behaviour to generate Dyck-1 with bounded nesting depth.

3.3 Theoretical Work on Counting in RNNs

Some work has been done towards determining whether counting can be implemented with RNNs in general and under limited numerical precision and the extent to which the RNNs can count.

Theoretically evaluating the extent to which RNNs can count has been of interest since the 1990s and has recently received attention as well. Both Hölldobler et al. (1997) and Collins et al. (2016) investigate the limits of counting in Elman RNNs, and find that the information that can be stored by an Elman RNN is limited. In the case of Hölldobler et al. (1997), they find that the largest counting depth is dependent on the precision of the hardware. Similarly, Collins et al. (2016) find that in general, the information that an Elman RNN can store is limited by the model parameters and the memory resources of the hardware.

In any practical case, the machines used operate with finite precision. Theoretically determining the ability of different RNNs to count to arbitrarily long sequences with finite precision can provide a better understanding of their behaviour. Like Hölldobler et al. (1997) and Collins et al. (2016), Weiss et al. (2018a) show that counting in Elman RNNs is limited. More generally, they show that RNNs with squashing activation functions, such as GRUs and Elman RNNs, do not have the capacity to count indefinitely with finite precision activation values. They show that this is due to the limited number of numeric values that can be represented in a squashed activation, hence eliminating the possibility of infinite counting. In practice, all computers operate with finite precision, as it is currently not possible for a computer to have infinite memory. Weiss et al. (2018a) also show that LSTM and ReLU RNNs do have the capacity for infinite counting with finite precision, unlike RNNs with squashing activations.

For LSTMs, the configuration that Weiss et al. (2018a) find that allows for infinite counting relies on saturating the activation functions. Saturating the activation functions replaces sigmoids with step functions, and these asymptotic (or saturated) RNNs are formalised by Merrill (2019) to simplify the model analysis. They show that saturated LSTMs can be used to count indefinitely.

3.4 Empirical Work on Counting in RNNs

As previously stated, empirically establishing connections between RNNs and formal languages involves training and testing RNN performance on formal language acceptance tasks. Here, we focus on empirical studies that empirically evaluate counting in RNNs. In most of these studies, $a^n b^n$ and Dyck-1 context-free language acceptance is used as the testable task, as these languages can be accepted by a counter. This task involves training the RNN model to predict the next possible character in a valid sequence at every timestep. In some cases, the models are tested on longer sequences to evaluate their ability to generalise systematically.

Earlier studies that empirically investigate the ability of RNNs to learn counting behaviour focus on Elman RNNs. These include the studies by Wiles and Elman (1995), Rodriguez and Wiles (1997), Rodriguez et al. (1999) and Bodén and Wiles (2000), who all find that Elman RNNs do not generalise effectively to longer sequences. Upon inspection of model dynamics, Wiles and Elman (1995) found that their models did not learn a counting mechanism and behaved like oscillators instead. Rodriguez and Wiles (1997), Rodriguez et al. (1999), and Bodén and Wiles (2000) all find that their models can sometimes struggle to learn counting behaviour from the data.

In more recent work, counting has been evaluated in other different RNN models such as GRUs and LSTMs. A well known paper on counting in LSTMs is that by Gers and Schmidhuber (2001). They take inspiration from previous works such as Wiles and Elman (1995). They focus on single-cell LSTMs and find that their models learn to count and accept $a^n b^n$ effectively. They also report that for most models, generalisation to longer sequences is limited, but in a few cases the models generalise effectively to longer sequences. This setup has been used again in in more recent literature such as Weiss et al. (2018a) and Suzgun et al. (2019a) who use the Dyck-1 language in their experiments. We also use the same setup in our experiments in Chapter 5. Weiss et al. (2018a) compare the performance of LSTM and GRU models, showing that LSTMs learn a counting counting mechanism more effectively than GRUs. In their experiments, both Gers and Schmidhuber (2001) and Weiss et al. (2018a) observed that their LSTMs generalised to some extent, but failed on longer sequences.

Suzgun et al. (2019a) evaluate the performance of LSTMs, GRUs and Elman RNNs on Dyck-1 acceptance. Like Weiss et al. (2018a) and Gers and Schmidhuber (2001), they find that the LSTMs can learn and generalise counting behaviour to a limited extent. They also observe that LSTMs learn counting behaviour more effectively than GRUs and Elman RNNs. However, they do not test their models on significantly longer sequences, and they do not provide much more insight into the behaviour of their models when tested on the longer sequences. Other papers that have evaluated bracket prediction in both natural and artificial strings with LSTMs are Karpathy et al. (2015), Bernardy (2018) and Skachkova et al. (2018). The results by Karpathy et al. (2015) show that LSTMs can learn to keep track of brackets and match closing brackets to their corresponding opening brackets but eventually fails on longer sequences. Skachkova et al. (2018) find that LSTMs generalise more effectively to longer sequences than Elman RNNs and GRUs. Bernardy (2018) learn to successfully match closing brackets to unmatched opening brackets, but the performance deteriorates with longer sequences.

In addition to Dyck-1, there have been studies investigating the ability of RNN models to learn higher order Dyck languages. In their work, Suzgun et al. (2019a) find that standard RNNs struggle to learn higher order Dyck languages. Bhattamishra et al. (2020) also investigate the ability of LSTMs to recognise and generalise higher order Dyck languages, specifically Dyck-2, 3 and 4. However, unlike Suzgun et al. (2019a), they also investigate the effect of restricting the nesting depth in training and generalising to larger nesting depths in testing. Their results show that the their models do not generalise effectively to larger nesting depths, even when the sequence length is the same.

3.5 Alternative Methods to Backpropagation for Training RNNs to Count

Backpropagation is the training regime used for the counting tasks we described earlier. There have also been alternative training methods proposed for RNN learning of counting behaviour. Mali et al. (2021) investigate alternative recurrent learning algorithms to backpropagation through time (BPTT) and evaluate the ability of LSTM, GRU and Elman RNN models to learn Dyck languages. They find that there are some alternatives to backpropagation that result in LSTMs generalising to longer sequences, but not always perfectly. Lan et al. (2022) use a Minimum Description Length score and a genetic algorithm to train ReLU, sigmoid and linear RNNs to count and accept a^nb^n, and find that their models are able to generalise effectively to longer sequences. However, the standard training algorithm used for NN models is backpropagation, therefore these alternative approaches are not standard.

3.6 Summary

In this chapter, we review different studies that evaluate counting behaviour in RNNs. We first introduce the different approaches to relating RNNs to formal automata and formal languages. We then specifically review theoretical and empirical work on counting in RNNs, and the alternative learning approaches that have been proposed.

Like Suzgun et al. (2019a), we use the Dyck-1 language in many of our experiments. When using the same setup as Gers and Schmidhuber (2001) with the Dyck-1 language, we can also view the task as a classification task, where we classify a subsequence at every timestep as valid or invalid in the Dyck-1 language. We evaluate the generalisation of our models on sequences which are significantly longer than the training sequences in order to better understand the long-term counting behaviour in the various RNN models we test. We also investigate the effects of different initialisations and setups.

Chapter 4

Effects of Different Configurations and Hyperparameters on Counting Behaviour in RNNs

In this chapter, we describe our initial experiments. We are interested in evaluating whether the RNN models can learn to count and accept the Dyck-1 language. The objective of these experiments is to determine if NN models can learn the counting behaviour required to systematically learn Dyck-1 strings in such a way that they can generalise effectively to longer strings. We also want to observe the effect of training our models from correct weights. In this chapter we use the terms string and sequence interchangeably to describe the elements of a dataset.

In this chapter, we address the first research question listed in Chapter 1: to what extent can RNNs be trained to count and generalise effectively beyond the training data? In 4.1, we attempt to train standard RNNs to count and systematically learn Dyck-1 strings, and subsequently classify them as valid or invalid. In 4.2 we design a linear bracket counter, allowing a negative count for simplicity. This linear counter is used to classify incoming bracket sequences as having a final bracket count greater than 0 or not greater than 0. We then design a more stack-like ReLU network counter in 4.3, which is used for Dyck-1 sequence classification. Similarly to the standard RNNs, the incoming Dyck-1 sequences are classified as valid or invalid. Different conditions and

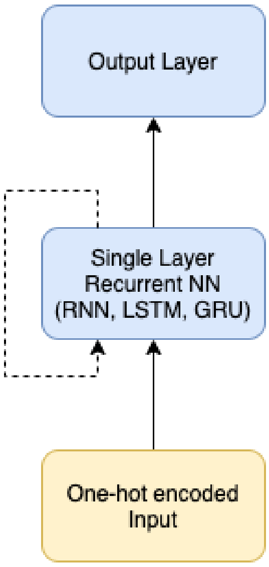


Figure 4.1: The architecture used for the Dyck-1 Acceptance task. We use Elman RNNs, LSTMs and GRUs with a single hidden layer and vary the number of neurons in the hidden layer.

configurations are used to test our models. All models are implemented using PyTorch. Sections 4.2.2 and 4.3.2 have been published in El-Naggar et al. (2022a).

4.1 Experiment 1: Dyck-1 Acceptance with Standard RNNs

In this section, we test the ability of standard RNN models to systematically learn to count and accept Dyck-1 strings from data. We use a classification task, where strings are classified as valid or invalid in the Dyck-1 language. Elman RNNs, LSTMs and GRUs are all used in this experiment.

We use a simple model for this experiment. The model consists of a fully connected input layer, a single layer Elman RNN/GRU/LSTM, and an output layer. We vary the number of hidden units in the Elman RNN/GRU/LSTM layer. The model is tested with hidden layers of sizes 2, 3 and 4 units. All layers are initialised using random weights and biases. The model is shown in Figure 4.1.

The goal of this experiment is to evaluate whether or not we can train traditional NN models to count and learn Dyck-1 sequences systematically, and hence extrapolate and generalise to unseen data.

In this experiment, we use Variant 1 of the Dyck-1 Validity datasets defined below to evaluate our models.

Dataset 1. (Dyck-1 Validity Datasets) These datasets are used to test a model's ability to accept the Dyck-1 language from the entire universe of possible sequences. We would also like to evaluate whether models can learn counting behaviour from minimal sequences. We use shorter sequences and provide feedback once at the end of each sequence. The sequences in this dataset are classified as:

- Valid: a valid Dyck-1 sequence.
- Invalid: an invalid Dyck-1 sequence.

The dataset is divided into:

- **Variant 1:**
  - **Short:** Bracket sequences of lengths 2 and 4 tokens, where the minority class is oversampled to ensure class balance. 30% of this dataset is used for testing and the remaining 70% is used for training. This dataset consists of 35 sequences.
  - **Long Test:** All bracket sequences ranging in length from 2 to 12 tokens. This dataset is also class balanced by oversampling the minority class and consists of approximately 10,000 sequences.
- **Variant 2:**
  - **Train 2:** All possible bracket strings of length 2.
  - **Train 4:** All possible bracket strings of lengths 2 and 4.
  - **Train 8:** All possible bracket strings of lengths 2, 4, and 8.
  - **Test 10:** All possible bracket strings of length 10.
  - **Test 20:** 150 randomly chosen strings of length 20 tokens.
  - **Test 50:** 150 randomly chosen strings of length 50 tokens.

Despite the overlap between the dataset variants, each of the variants of this dataset is a collection of datasets designed for a specific experiment and are not complementary to each other.

We use 70% of our short dataset for training and the remainder for testing. We completely exclude the long test set from the training. This longer dataset is used to test the model's generalisation to unseen data.

For each sequence, one token is passed to the model at every time step. During training, backpropagation occurs after an entire string has been passed through the model. Feedback is provided once for every string, which is similar to what is normally done. Furthermore, we do not want to provide too much feedback to the models to make the learning task more challenging. We use 2 different output activation functions for this model:

1. Sigmoid (see Definition 2.1).

2. Clipping: We bound the output activation to between 0 and 1 (see Definition 2.5).

Different hidden layer sizes are also used to test the model. We test our model using a single hidden layer consisting of 2, 3 and 4 Elman RNN/LSTM/GRU units. There is evidence from previous literature that counting behaviour can be realised with small RNN models (see Weiss et al. (2018a) and Suzgun et al. (2019a)). In their experiments on Dyck-1, Suzgun et al. (2019a) use 3 hidden units, and we use models with the same hidden unit size, as well as 2 and 4 hidden units for comparison. For each model and configuration used, 10 experiment runs are carried out. When 2 hidden units are used, we test the effect of freezing the input layer, to investigate the effect of feeding the one-hot encoding directly to the RNN/GRU/LSTM.

For training, we use 70% of the short dataset, and use the remaining 30% for testing. The dataset consists of sequences of lengths 2 or 4 brackets.

The model is trained over 1000 epochs. We use the Adam optimiser with a learning rate of 0.001. The MSE loss function is used for backpropagation. Feedback is only given to the model at the very end of each sequence, when all tokens in the sequence have been passed through the model.

After training, we test the model on the remaining 30% of the original dataset. We then test it on the longer dataset to evaluate its ability to generalise to data that has not been seen in training, specifically, longer sequences from the long test set. We do

Table 4.1: This table summarises the results of the Dyck-1 acceptance task using standard RNNs with a single hidden layer, specifically Elman RNNs, GRUs and LSTMs as shown in Figure 4.1. Each model is tested with 2, 3 and 4 hidden units. We also experiment with fixing the input weights and using both sigmoid and clipping activation functions.

	Hidden	Output	Fixed		Train			Test			Long	
Model	**Units**	**Activation**	**Weights**	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
RNN	2	Sigmoid	None	87.5	100	93.75	81.82	100	92.73	46.22	91.32	66.20
GRU	2	Sigmoid	None	95.83	100	99.58	72.73	90.91	81.82	45.65	75.57	57.49
LSTM	2	Sigmoid	None	95.83	100	99.58	81.82	100	92.73	47.67	85.68	65.56
RNN	2	Clipping	None	50	100	83.33	54.55	100	82.73	50	64.97	57.35
GRU	2	Clipping	None	50	100	73.75	54.55	100	77.27	48.22	66.06	52.58
LSTM	2	Clipping	None	50	100	80	54.55	90.91	74.55	50	89.60	67.2
RNN	2	Sigmoid	Input	87.5	100	95.83	72.73	90.91	81.82	58.14	76.01	62.73
GRU	2	Sigmoid	Input	100	100	100	90.91	100	97.27	48.58	70.96	57.21
LSTM	2	Sigmoid	Input	100	100	100	63.64	100	90.91	49.34	90.16	60.64
RNN	2	Clipping	Input	50	100	68.33	54.55	100	69.99	50	69.48	53.99
GRU	2	Clipping	Input	50	100	75	54.55	90.91	72.73	50	82.45	56.49
LSTM	2	Clipping	Input	50	100	79.58	54.55	100	79.99	50	87.89	54.83
RNN	3	Sigmoid	None	91.67	100	98.33	63.64	90.91	81.82	51.04	66.04	58.86
GRU	3	Sigmoid	None	100	100	100	90.91	100	91.82	44.71	62.58	53.16
LSTM	3	Sigmoid	None	100	100	100	90.91	100	99.09	57.86	92.95	80.07
RNN	3	Clipping	None	50	100	75	54.55	90.91	72.73	49.51	85.84	61.04
GRU	3	Clipping	None	50	100	80	54.55	90.91	74.55	50	60.83	52.62
LSTM	3	Clipping	None	50	100	90	54.55	100	80.91	48.55	91.38	55.25
RNN	4	Sigmoid	None	95.83	100	99.58	90.91	100	94.55	53.99	68.86	59.69
GRU	4	Sigmoid	None	100	100	100	63.64	90.91	80.91	51.65	70.87	58.92
LSTM	4	Sigmoid	None	100	100	100	72.73	100	88.18	49.34	93.1	75.64
RNN	4	Clipping	None	50	100	65	54.55	72.73	58.18	50	73.89	56.75
GRU	4	Clipping	None	50	100	70	54.55	90.91	66.36	50	60.55	52.23
LSTM	4	Clipping	None	50	100	65	54.55	90.91	64.55	50	93.33	62.19

this for the RNN, GRU and LSTM models.

4.1.1 Results

We perform 10 runs of training and testing on each of our models. The results are summarised in Table 4.1. They show that the models were all capable of training to 100%, but were not always able to achieve 100% on the short test set. None of the models and configurations used achieved 100% on the long test set. In most cases, the models using a sigmoid activation outperformed their counterparts with a clipping activation. We observe that the performance of models in general increased when the number of hidden units was increased from 2 to 3. However, we also observe a decline in performance when the number of hidden units is increased from 3 to 4. This agrees with the observation by Suzgun et al. (2019a) that 3 hidden neurons is a good number for accepting Dyck-1, even with our different setup. While we do not have any hypothesis as to why this happens, developing a more rigorous understanding of model behaviour may provide more insights.

Freezing the input layer for the model using the sigmoid activation with 2 hidden units resulted in an increase in the training accuracy, but only the GRU model achieved higher accuracy on the test set and a comparable accuracy on long test set with a frozen input layer. The accuracies of the RNN and LSTM models on the short and long test sets dropped slightly when the input layer was frozen. As an ablation study, we experiment with the input weights to evaluate both, end-to-end learning with a trainable input layer, as well as providing the encoding directly to the model, with the input weights frozen, to check if the remaining parts are being learned.

When a clipping output activation was used in combination with a frozen input layer, the training and long test accuracies of RNN and LSTM models decreased, in contrast to the GRU which had an increase in training accuracy when the input layer was frozen. The performance on the short test set is generally lower, except when the LSTM model was used.

The best overall performance and generalisation was achieved using an LSTM model with 3 hidden units, and a sigmoid activation function on the output layer. This model was able to generalise quite well, but not perfectly, to unseen data.

4.2 Experiment 2: Bracket Counting with Linear RNNs

In this experiment, we use the simplest RNN to evaluate counting behaviour, a single-cell linear RNN. A single-cell linear RNN cannot be used to accept Dyck-1, as will be shown in section 7.1, and additional mechanisms would be needed to detect out-of-order closing brackets. Therefore, we allow the counter to have both positive and negative values. We use a classification task for the final bracket count at the end of a sequence, where an opening bracket increments the count and a closing bracket decrements the count. Similar to Dyck-1 acceptance, this is a counting task. However, unlike Dyck-1 acceptance, we simplify the task by disregarding the order of brackets, and solely focusing on the final bracket count. The classes for this task are:

1. Bracket difference ≤ 0: the overall bracket difference is 0 or less.

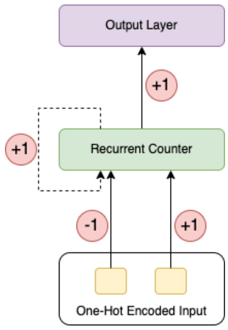


Figure 4.2: The architecture used for the Bracket Counting task. We use a single-hidden layer single-cell linear RNN.

2. Bracket difference > 0: the overall bracket difference is greater than 0.

We use the model shown in Figure 4.2. We define here the Bracket Difference datasets, which we use to evaluate our single-cell linear RNN models.

Dataset 2. (Bracket Difference Datasets) These datasets consist of bracket strings with varying numbers of opening and closing brackets. The elements of these datasets are labelled once at the end of each sequence. These datasets contain sequences with surplus closing brackets. These datasets are class balanced. They are used to evaluate whether or not a model can count, with a single label at the end of the sequence, and this allows us to test minimal linear RNNs. The task here is a bit simpler than Dyck-1 acceptance because in Dyck-1 acceptance, a mechanism would be needed to keep track of excess closing brackets. We use binary and ternary classification tasks to evaluate the effect of the different labels on the learning of the models.

- **Binary:** The bracket difference is classified as > 0 or ≤ 0.
  - **Train 2:** All possible bracket strings of length 2.
  - **Train 4:** All possible bracket strings of lengths 2 and 4.
  - **Train 8:** All possible bracket strings of lengths 2, 4, and 8.
  - **Test 10:** All possible bracket strings of length 10.
  - **Test 20:** 150 strings of length 20 tokens.
  - **Test 50:** 150 strings of length 50 tokens.
- **Ternary:** The bracket difference is classified as > 0, 0 or < 0.

  - **Train 2:** All possible bracket strings of length 2.
  - **Train 4:** All possible bracket strings of lengths 2 and 4.
  - **Train 8:** All possible bracket strings of lengths 2, 4, and 8.
  - **Test 10:** All possible bracket strings of length 10.
  - **Test 20:** 150 strings of length 20 tokens.
  - **Test 50:** 150 strings of length 50 tokens.

- **Cursory Binary:** The bracket difference is classified as > 0 or ≤ 0.
  - **Short:** All possible bracket strings of length 2 tokens. The dataset is then balanced by oversampling the minority class.
  - **Long:** 19 bracket strings of lengths 4 to 60 tokens.

4.2.1 Cursory Evaluation

The aim of this experiment is to perform a cursory evaluation whether a single-cell linear RNN can learn to behave as a bracket counter and investigate the effect of different conditions on model performance. We use the Bracket Difference Cursory Binary dataset (see Dataset 2) for this experiment. The class labels are encoded as 0 (bracket difference ≤ 0) or 1 (bracket difference > 0). We simplify the task, by allowing negative values in the counter, and we remove all biases. We use two different configurations for this model.

1. **Classification configuration:** Here a sigmoid activation is used for the output layer, and a BCE loss function.
2. **Regression-style configuration:** Here a clipping activation is used for the output layer. This bounds the output between 0 and 1. It is used alongside a MSE loss function. We introduce this setup because we are interested in determining whether an activation function that can produce an output identical to the ground truth combined with a loss function that can output a value of 0 results in more effective learning. Also, we would like to investigate the effect of this setup on correctly initialised weights in comparison to the standard classifcation setup.

For both configurations, the models are trained under a variety of different conditions:

1. Correct/Random weight initialisation
2. Correct/Random initialisation with weight discretisation
3. Correct/Random initialisation with regularisation (L1, L2, Dropout).
4. Correct initialisation with noise.

We show the correct weight configuration used in Figure 4.2. We also investigate the occurrence of unlearning when a model is correctly initialised. The different configurations are used to determine which activation function is better suited to this discrete task. We also plan to test the effect of noise on the models, and whether it is possible to recover from the noise. The generalisation to unseen data is also at the forefront of the things we aim to investigate in this experiment.

Before training and testing, to ensure that we assume correct weights and biases, we run the entire dataset through the models when they are initialised with the perfect weights.

The model was trained using the following conditions:

- 1000 epochs
- 70% of the dataset was used for training, and the remaining 30% used for testing.
- Learning rate of 0.005
- SGD optimiser

The feedback is only given to the network after the entire sequence has been passed through the model. Backpropagation does not occur after every time step, only at the very end of a sequence.

In order to test this model and its generalisation capability using different configurations, we use the remaining 30% of the dataset that was not used in training. We also use another separate test set consisting of much longer sequences. This length set consists of 19 sequences ranging in length between 4 and 60 brackets. The longer test set has been excluded from the training.

Table 4.2: Results of Cursory Evaluation for Bracket Counter model shown in Figure 4.2. We show Avg (Min/Max) Accuracy for Train, Test and Long datasets over 3 runs.

Configuration/Initialisation	Regularisation	Noise	Discretisation	Train	Test	Long
Classification/Correct	No	No	No	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Correct	No	No	Loss	100 (100/100)	100 (100/100)	63.2 (63.2/63.2)
Classification/Correct	No	No	Postprocessing	100 (100/100)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Classification/Correct	Dropout	No	No	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Correct	L1	No	No	100 (100/100)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Classification/Correct	L2	No	No	0.0 (0.0/0.0)	50.0 (50.0/50.0)	57.9 (57.9/57.9)
Classification/Correct	No	[-0.1 to 0.1]	No	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Correct	No	[-0.2 to 0.2]	No	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Correct	No	[-0.3 to 0.3]	No	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Correct	No	[-0.4 to 0.4]	No	100 (100/100)	100 (100/100)	71.1 (68.4/73.7)
Classification/Correct	No	[-0.5 to 0.5]	No	100 (100/100)	100 (100/100)	71.1 (68.4/73.7)
Classification/Random	No	No	No	100 (100/100)	100 (100/100)	57.9 (42.1/73.7)
Classification/Random	No	No	Loss	67.0 (0.0/100)	83.3 (50.0/100)	66.7 (57.9/73.7)
Classification/Random	No	No	Postprocessing	100 (100/100)	100 (100/100)	57.9 (57.9/57.9)
Classification/Random	Dropout	No	None	100 (100/100)	100 (100/100)	68.4 (68.4/68.4)
Classification/Random	L1	No	No	33.0 (0.0/100)	50.0 (50.0/50.0)	52.6 (42.1/57.9)
Classification/Random	L2	No	No	100 (100/100)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Regression/Correct	No	No	No	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.1 to 0.1]	Postprocessing	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.1 to 0.1]	No	100 (100/100)	100 (100/100)	54.4 (47.4/57.9)
Regression/Correct	No	[-0.2 to 0.2]	No	100 (100/100)	100 (100/100)	57.9 (57.9/57.9)
Regression/Correct	No	[-0.3 to 0.3]	No	100 (100/100)	100 (100/100)	60.1 (47.4/73.7)
Regression/Correct	No	[-0.4 to 0.4]	No	100 (100/100)	100 (100/100)	60.1 (57.9/63.2)
Regression/Correct	No	[-0.5 to 0.5]	No	100 (100/100)	100 (100/100)	60.1 (57.9/63.2)
Regression/Correct	Dropout	No	No	58.0 (25.0/75.0)	100 (100/100)	100 (100/100)
Regression/Correct	L1	No	No	50.0 (50.0/50.0)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Regression/Correct	L2	No	No	50.0 (50.0/50.0)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Regression/Random	No	No	No	100 (100/100)	100 (100/100)	42.1 (42.1/42.1)
Regression/Random	No	No	Loss	50.0 (50.0/50.0)	50.0 (50.0/50.0)	47.4 (42.1/57.9)
Regression/Random	No	No	Postprocessing	100 (100/100)	100 (100/100)	42.1 (42.1/42.1)
Regression/Random	Dropout	No	No	100 (100/100)	100 (100/100)	42.1 (42.1/42.1)
Regression/Random	L1	No	No	50.0 (50.0/50.0)	50.0 (50.0/50.0)	42.1 (42.1/42.1)
Regression/Random	L2	No	No	50.0 (50.0/50.0)	50.0 (50.0/50.0)	42.1 (42.1/42.1)

When the traditional configuration is used, the model never achieves 100% on the length set, but sometimes achieves 100% on the shorter test set. When the regression-style configuration is used, there are conditions under which the model can generalise to longer sequences.

We test the effect of adding noise to the correct weights before training to see if the models can recover and converge back to the correct weights in training. We are also interested in finding out the range of added noise that can be corrected in training. We also experiment with discretising the weights after training, which we refer to as postprocessing discretisation. We also test the effect of discretising the loss before backpropagation. In both cases, this discretisation is done by rounding the values to the nearest integer.

Table 4.3: Results for Bracket Counting shown in Figure 4.2: average, minimum, and maximum accuracy over 10 runs.

Setup Config/Init.	Train Length	Train Avg(Min/Max)	10 Tokens Avg(Min/Max)	20 Tokens Avg(Min/Max)	50 Tokens Avg(Min/Max)
Class./Random	2	100 (100/100)	69.8 (66.5/78.6)	69.2 (6.04/77.3)	69.0 (66.7/72.7)
Class./Random	4	100 (100/100)	74.8 (74.6/75.7)	94.8 (94.7/95.3)	89.3 (88.7/90.0)
Class./Random	8	100 (100/100)	100 (99.9/100)	96.9 (94.0/100)	92.7 (78.7/98.0)
Reg./Random	2	90.0 (50.0/100)	72.0 (50.0/99.9)	64.5 (50.0/96.0)	61.2 (50.0/91.3)
Reg./Random	4	86.1 (50.0/100)	84.4 (50.0/100)	83.9 (50.0/100)	82.9 (50.0/100)
Reg./Random	8	90.0 (50.0/100)	90.0 (50.0/100)	90.0 (50.0/100)	88.3 (50.0/100)
Class./Correct	2	100 (100/100)	67.9 (67.9/67.9)	71.3 (71.3/71.3)	69.3 (69.3/69.3)
Class./Correct	4	100 (100/100)	74.1 (74.1/74.1)	94.0 (94.0/94.0)	92.0 (92.0/92.0)
Class./Correct	8	100 (100/100)	100 (100/100)	96.7 (96.7/96.7)	93.3 (93.3/93.3)
Reg./Correct	Any	100 (100/100)	100 (100/100)	100 (100/100)	100 (100/100)

Results

The results for this model show that traditional classification techniques which use a sigmoid output activation with cross entropy loss unlearn the correct weights and biases. This in turn reduces the networks generalisation capabilities. The regression style models where a clipping classification activation was used with mean squared error loss did not unlearn the correct weights. However, both the traditional and regression-style approaches did not manage to find the correct weights from the data. When regression-style configuration is used, the model was able to recover from noise and generalise to longer sequences when the network weights and biases were discretised postprocessing. Overall, the regression-style models performed better and generalised better to longer sequences when the correct weights and biases were initialised. The results of the experiments for both configurations are summarised in Table 4.2.

4.2.2 Experimental Setup

The contents of this subsection are a modified version of part of El-Naggar et al. (2022a). We use the Binary variant of the Bracket Difference Datasets (see Dataset 2). We train the model shown in Figure 4.2 with each of the different training sets and test the model on the longer test sets. All datasets are class balanced. There is a label at the end of each sequence, with the following encoding, 1 (bracket difference > 0) and 0 (bracket difference ≤ 0). We use the two different setups used in the cursory evaluation in 4.2.1,

- **Classification:** sigmoid output activation with binary cross-entropy loss, (see Definitions 2.1 and 2.8, respectively)
- **Regression:** clipped output [0, 1] with mean squared error (MSE) loss, (see Definitions 2.5 and 2.6, respectively).

We use two initialisation schemes: random weights and correct weights according to Figure 4.2. There are no trainable biases in the models used in this experiment. We train for 100 epochs using the Adam optimiser and a learning rate of 0.005.

Results

In Table 4.3, we observe that the task is learned on the training set, but generalises less for longer sequences. Longer sequences in the training data do lead to improved generalisation, but still not perfect in most cases. Interestingly, starting from correct weights with the classification setup does not lead to better generalisation. This is intriguing, as the training apparently unlearns the correct weight values. The regression setup does not change the correct weights (last row in Table 4.3), but it does not learn well from random intialisation. When training from correct weights with the classification setup, which uses a sigmoid activation function with CE loss, we observe that output of the model is not identical to the ground truth and hence the resulting loss values in training are not 0. This results in the models unlearning the correct weights in the optimisation. In contrast, in the regression setup, the labels produced when trained from correct weights are identical to the ground truth, which result in a loss value of 0, and therefore the weights are not changed during the optimisation.

4.3 Experiment 3: Dyck-1 Acceptance with ReLU RNNs

For this experiment, we design a ReLU RNN model which can be configured to fully accept Dyck-1 strings. We evaluate whether or not this model can be trained to count brackets to correctly classify Dyck-1 sequences. This time, the bracket difference should be 0 and order of brackets does matter, because this is a Dyck-1 acceptance task, and closing brackets cannot precede their corresponding opening brackets in Dyck-1 strings.

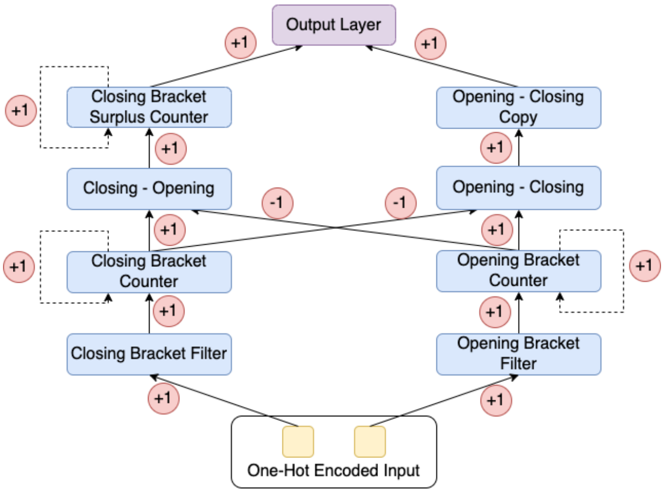


Figure 4.3: Architecture of the Dyck-1 ReLU Counter

This model is more complex than the previous model used in 4.2. We design a RNN model which can be used to accept Dyck-1 strings. The model consists of 4 hidden layers, where all neurons have a ReLU activation function (see Definition 2.4), and an output layer with a sigmoid activation (see Definition 2.1). Negative values are not allowed when a closing bracket is detected and the count is 0. There is a mechanism in the model which counts the number of excess closing brackets. The architecture of this model is shown in Figure 4.3.

The task for this model involves the classification of Dyck-1 sentences, we want the network to learn the grammar of Dyck-1. In order for a Dyck sentence to be valid, the number of opening and closing brackets should be the same, and they should occur in the correct order, where a closing bracket cannot precede its corresponding opening bracket. The model is used to classify sequences as:

1. Valid: a valid Dyck-1 sequence (encoded as 0).

2. Invalid: an invalid Dyck-1 sequence (encoded as 1).

4.3.1 Cursory Evaluation

We use Variant 1 of the Dyck-1 Validity Datasets (see Dataset 1). 30% of the short dataset is used for testing and the remainder is used for training. In order to test the model's generalisation capability, we use the long test dataset. The dataset is input to the model as one-hot encodings.

The goal of this experiment is to test whether a ReLU network can be trained to behave like a non-negative counter and consequently accept Dyck-1 strings. We use two different configurations for this model.

1. **Classification configuration:** Here a sigmoid output activation is used, and a Binary Cross-Entropy (BCE) loss function (see Definitions 2.1 and 2.8).
2. **Regression configuration:** Here a clipping output activation is used alongside a Mean Squared Error (MSE) loss function (see Definitions 2.5 and 2.6).

Both configurations are trained under a variety of different conditions:

1. Correct/Random initialisation
2. Correct/Random initialisation with weight discretisation
3. Correct/Random initialisation with regularisation (L1, L2, Dropout).
4. Correct initialisation with noise.

Similar to the previous experiment, we also test our models with the regression setup in order to observe the effects of using this setup which can result in an identical value to the ground truth and also a zero loss value in comparison to the classification setup. We want to investigate the effect of different activation functions on this discrete task. We also want to evaluate whether any unlearning happens, and which activation function is better suited to the task. The effect of noise on the model is also something we plan to test. We also focus on the generalisation to unseen data and how it is affected by the different conditions and configurations we use.

Before training and testing, to ensure that we assume correct weights we run the entire dataset through the models when they are initialised with the perfect weights.

The model was trained using the following conditions:

- 1000 epochs
- 2/3 of the dataset was used for training, and the remaining 1/3 used for testing.

- Learning rate of 0.005
- SGD optimiser

The feedback is only given to the network after the entire sequence has been passed through the model. Backpropagation does not occur after every time step, only at the very end of a sequence. The model is tested using 30% of the short dataset. It is then tested on the long test dataset.

Results

The results are summarised in Table 4.4. Like the linear bracket counter in 4.2, the results show that the regression-style configuration did not unlearn the correct initialisation. The model could not learn the correct weights from data using both configurations. The model can recover from minimal noise when the correct weights and biases are initialised, if weight discretisation is performed postprocessing.

4.3.2 Experimental Setup

This subsection is a modification of part of El-Naggar et al. (2022a). We use Variant 2 of the Dyck-1 Validity Datasets (see Dataset 1). We train the model on each of the training sets and test on the different test sets. There is a label at the end of each sequence, with the following encoding, 1 (bracket difference > 0) and 0 (bracket difference ≤ 0). We use the two different setups from the cursory evaluation,

- **Classification setup:** sigmoid output activation with binary cross-entropy loss (see Definitions 2.1 and 2.8).
- **Regression setup:** Clipped output [0, 1] with mean squared error (MSE) loss (see Definitions 2.5 and 2.6).

We use two initialisation schemes: random weights and correct weights according to Figure 4.3. We train for 100 epochs using the Adam optimiser and a learning rate of 0.005.

Table 4.4: Results of Cursory Evaluation of Dyck-1 acceptance with the Dyck-1 ReLU Counter shown in Figure 4.3. We show Avg (Min/Max) Accuracy for Train, Test and Long datasets over 3 runs.

Configuration/Initialisation	Regularisation	Noise	Discretisation	Train	Test	Long
Classification/Correct	No	No	No	100 (100/100)	100 (100/100)	52.3 (50.0/55.3)
Classification/Correct	No	No	Loss	97.0 (91.7/100)	100 (100/100)	83.0 (49.0/100)
Classification/Correct	No	No	Postprocessing	100 (100/100)	87.9 (72.7/100)	92.8 (92.8/92.8)
Classification/Correct	Dropout	No	No	100 (100/100)	100 (100/100)	54.8 (52.9/56.2)
Classification/Correct	L1	No	No	43.0 (33.3/50.0)	51.5 (45.5/54.5)	50.0 (50.0/50.0)
Classification/Correct	L2	No	No	37.0 (33.3/45.8)	51.5 (45.5/54.5)	50.0 (50.0/50.0)
Classification/Correct	No	[-0.1 to 0.1]	No	100 (100/100)	100 (100/100)	53.3 (49.9/55.0)
Classification/Correct	No	[-0.2 to 0.2]	No	100 (100/100)	100 (100/100)	58.3 (49.0/73.8)
Classification/Correct	No	[-0.3 to 0.3]	No	100 (100/100)	100 (100/100)	71.9 (56.8/87.2)
Classification/Correct	No	[-0.4 to 0.4]	No	100 (100/100)	100 (100/100)	61.3 (49.5/84.8)
Classification/Correct	No	[-0.5 to 0.5]	No	72.0 (50.0/100.0)	63.6 (45.5/81.8)	48.9 (46.8/50.0)
Classification/Correct	No	[-0.1 to 0.1]	Postprocessing	100 (100/100)	77.3 (72.7/81.8)	92.8 (92.8/92.8)
Classification/Correct	No	[-0.2 to 0.2]	Postprocessing	100 (100/100)	90.9 (90.9/90.9)	95.5 (95.5/95.5)
Classification/Correct	No	[-0.3 to 0.3]	Postprocessing	100 (100/100)	90.9 (90.9/90.9)	49.3 (49.3/49.3)
Classification/Correct	No	[-0.4 to 0.4]	Postprocessing	100 (100/100)	63.6 (63.6/63.6)	94.1 (94.1/94.1)
Classification/Correct	No	[-0.5 to 0.5]	Postprocessing	50.0 (50.0/50.0)	45.5 (45.5/45.5)	50.0 (50.0/50.0)
Classification/Random	No	No	No	42.0 (37.5/45.8)	48.5 (45.5/54.5)	50.0 (50.0/50.0)
Classification/Random	No	No	Loss	50.0 (45.8/54.2)	51.5 (45.5/54.5)	50.0 (50.0/50.0)
Classification/Random	Dropout	No	No	57.0 (37.5/91.7)	57.6 (45.5/81.8)	53.9 (50.0/61.8)
Classification/Random	L1	No	No	49.0 (41.7/58.3)	48.5 (45.5/54.5)	50.0 (50.0/50.0)
Classification/Random	L2	No	No	40.0 (37.5/45.8)	51.5 (45.5/54.5)	50.0 (50.0/50.0)
Regression/Correct	No	No	No	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.1 to 0.1]	No	100 (100/100)	100 (100/100)	86.7 (79.2/92.6)
Regression/Correct	No	[-0.2 to 0.2]	No	50.0 (50.0/50.0)	45.5 (45.5/45.5)	50.0 (50.0/50.0)
Regression/Correct	No	[-0.3 to 0.3]	No	50.0 (50.0/50.0)	45.5 (45.5/45.5)	50.0 (50.0/50.0)
Regression/Correct	No	[-0.4 to 0.4]	No	50.0 (67.0/100)	60.6 (45.5/90.9)	50.0 (50.0/50.0)
Regression/Correct	No	[-0.5 to 0.5]	No	50.0 (50.0/50.0)	45.5 (45.5/45.5)	50.0 (50.0/50.0)
Regression/Correct	No	[-0.1 to 0.1]	Postprocessing	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.2 to 0.2]	Postprocessing	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.3 to 0.3]	Postprocessing	50.0 (50.0/50.0)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.4 to 0.4]	Postprocessing	50.0 (50.0/50.0)	100 (100/100)	100 (100/100)
Regression/Correct	No	[-0.5 to 0.5]	Postprocessing	50.0 (50.0/50.0)	100 (100/100)	100 (100/100)
Regression/Correct	Dropout	No	No	100 (100/100)	100 (100/100)	100 (100/100)
Regression/Correct	L1	No	No	49.0 (41.7/54.2)	48.5 (45.5/54.5)	50.0 (50.0/50.0)
Regression/Correct	L2	No	No	50.0 (50.0/50.0)	45.5 (45.5/45.5)	50.0 (50.0/50.0)
Regression/Random	No	No	No	64.0 (50.0/91.7)	60.6 (45.5/81.8)	51.4 (50.0/54.3)
Regression/Random	No	No	Loss	50.0 (50.0/50.0)	48.5 (45.5/54.5)	50.0 50.0/50.0)
Regression/Random	Dropout	No	No	42.0 (33.3/50.0)	51.5 (45.5/54.5)	50.0 (50.0/50.0)
Regression/Random	L1	No	No	46.0 (41.7/50.0)	54.5 (54.5/54.5)	50.0 (50.0/50.0)
Regression/Random	L2	No	No	50.0 (50.0/50.0)	51.5 (45.5/54.5)	50.0 (50.0/50.0)

Table 4.5: Results for Dyck-1 acceptance with Dyck-1 ReLU Counter shown in Figure 4.3: average, minimum, and maximum accuracy over 10 runs.

Setup	Train	Train	10 Tokens	20 Tokens	50 Tokens
Config/Init.	Length	Avg(Min/Max)	Avg(Min/Max)	Avg(Min/Max)	Avg(Min/Max)
Class./Random	2	56.7 (50.0/83.3)	52.7 (50.0/65.3)	54.1 (50.0/71.3)	59.9 (50.0/71.3)
Class./Random	4	55.9 (50.0/79.4)	55.2 (50.0/76.1)	54.5 (50.0/72.7)	54.3 (50.0/71.3)
Class./Random	8	55.3 (50.0/76.6)	55.2 (50.0/76.1)	54.5 (50.0/72.7)	54.3 (50.0/71.3)
Reg./Random	2	56.7 (50.0/83.3)	55.2 (50.0/76.1)	54.5 (50.0/72.7)	54.3 (50.0/71.3)
Reg./Random	4	55.9 (50.0/79.4)	55.2 (50.0/76.1)	54.5 (50.0/72.7)	54.3 (50.0/71.3)
Reg./Random	8	55.3 (50.0/76.6)	56.6 (50.0/83.4)	54.5 (50.0/72.7)	54.3 (50.0/71.3)
Class./Correct	2	100 (100/100)	49.8 (49.8/49.8)	50.0 (50.0/50.0)	50.0 (50.0/50.0)
Class./Correct	4	100 (100/100)	100 (100/100)	86.7 (86.7/86.7)	80.67 (80.7/80.7)
Class./Correct	8	96.7 (96.7/96.7)	97.9 (97.9/97.9)	66.7 (66.7/66.7)	48.0 (48.0/48.0)
Reg./Correct	Any	100 (100/100)	100 (100/100)	100 (100/100)	100 (100/100)

Results

In Table 4.5, we observe that the task is learned on the training set, but generalises less for longer sequences. Longer sequences in the training data do lead to improved generalisation, but still not perfect in most cases. Interestingly, starting from correct weights with the classification setup does not lead to better generalisation. This is intriguing, as the training apparently unlearns the correct weight values. In order to avoid this problem, we developed the regression setup, which does not change the correct weights (last row in Table 4.5), but it does not learn well from random initialisation. The learning from random weights never leads to perfect training or generalisation. Even with correct initialisation, generalisation is mostly poor. This unlearning is likely the result of the classification setup not outputting an identical value to the ground truth and consequently, non-zero losses causing the model to deviate from the correct weights during training.

4.4 Summary

This chapter details our preliminary experiments. We test the ability of standard recurrent models, simple linear models and ReLU models to behave like bracket counters for different classification tasks. The models are tested under different conditions and different configurations are used.

We have a few interesting observations in our experiments with linear and ReLU

RNNs. Learning Dyck-1 sequences from random weights is a difficult task and the results do not generalise to long sequences in any setup not initialised with correct weights. Using longer (and thus more) training sequences leads to some improvements, but generalisation to longer sequences is still limited. Initialising the network with correct weights could help, but even that is not effective in a classification task. The tested approach of using a regression setup (clipping and MSE) can avoid unlearning of correct weights, but it hinders learning from random weight initialisation, and is thus not effective either.

The standard recurrent models, on the other hand, were able to train to 100% accuracy from random weights and biases, even though that was not always the case. However, the accuracy on the longer test set was significantly lower than the training or short test accuracies. The generalisation to longer sequences was not as good as we had hoped it would be for all the models we tested in our preliminary experiments.

The observation is that RNNs fail to learn symbolic patterns and the systematic behaviour required to count brackets and recognise Dyck-1 sequences. This is consistent with studies that focus on the abilities of NNs to learn systematic behaviour, such as Fodor and Pylyshyn (1988), Marcus et al. (1999), and Lake and Baroni (2018). These studies show that NNs struggle with tasks that are simple for humans to learn from a small number of examples. The difficulty to learn symbolic patterns in a systematic manner is not exclusive to larger models, and is evidently exhibited by our very small models.

Chapter 5

Exploring the Long Term Generalisation of Counting Behaviour in RNNs

In this chapter, we evaluate the abilities of LSTMs, GRUs, and ReLU RNNs to learn to count and accept the Dyck-1 language. We also investigate the limits of trained ReLU, LSTM and GRU models in generalising Dyck-1 acceptance to strings that are substantially longer than those in the training set and analyse when, how and why they fail. This chapter is a modified version of El-Naggar et al. (2022b).

5.1 Modelling Dyck-1 Strings

We evaluate the limit of three different single-cell RNN models trained on Dyck-1 strings in generalising to longer strings. Dyck-1 strings have the following properties:

a) One type of brackets,

b) An equal number of opening and closing brackets,

c) No closing brackets before their corresponding opening brackets.

We use the same task setup as Suzgun et al. (2019a) to evaluate our models, which is to classify which tokens are valid to follow in a Dyck-1 string after each token. This is equivalent to classifying complete vs incomplete Dyck-1 strings. This is limited as a Dyck-1 language acceptance task. While the string is classified at every time step, strings that violate property c) do not occur. Therefore, this setup does not allow for testing full language acceptance, as defined, e.g., in Merrill (2020).

In this chapter, we use Dataset 3, the Dyck-1 Prediction Datasets, defined below, which are the same types of datasets as reported in Suzgun et al. (2019a).

Dataset 3. (Dyck-1 Prediction Datasets) These dataset are generated in the same way as Suzgun et al. (2019a). These datasets consist of valid Dyck-1 strings with labels at every timestep indicating the possible next characters. At each timestep, there are 2 labels, one representing an opening bracket and another representing a closing bracket. The label for each bracket is 1 if its corresponding bracket is a possible next token in a valid Dyck-1 sequence and 0 otherwise. Opening brackets can occur at any timestep, and the corresponding output label is therefore always 1. A closing bracket cannot occur when a subsequence is itself a valid Dyck-1 sequence, and in this case, the target label for a closing bracket is 0, but is 1 otherwise. This dataset allows us to compare our work to Suzgun et al. (2019a) and perform more detailed investigations on a model's generalisation abilities because of the labels at every timestep. It allows us to find the exact point at which a model fails to extrapolate and produces an incorrect output label. This dataset can be used with ReLU models due to there never being a surplus of closing brackets at any point. ReLU models cannot produce a negative output and therefore cannot accept Dyck-1 sequences from the entire universe of possible sequences. If any excess closing brackets occurred when using a ReLU model, the ReLU would rectify the negative count to 0, which would result in the ReLU model not differentiating between 0 and negative counts. The sequences in this dataset are all valid Dyck-1 sequences. Therefore, surplus closing brackets that would result in a negative count do not occur, making this dataset straightforward to use with a ReLU model.

- **Train:** 10,000 Dyck-1 strings of lengths between 2 and 50 tokens.
- **Validation:** 5,000 Dyck-1 strings of lengths between 2 and 50 tokens.
- **Long Test:** 5,000 Dyck-1 strings of lengths between 52 and 100 tokens.

- **Very Long Test:** 100 Dyck-1 strings of length 1,000 tokens.
- **Zigzag:** 10 Dyck-1 strings of length 2000 tokens consisting of repetitions of j opening brackets followed by j closing brackets for $j = \{10, 20, 25, 50, 100, 125, 200, 250, 500, 1000\}$.

For training and initial testing, we use the Training, Validation and Long Test Sets. Our *Training Set* consists of 10000 strings with lengths of 2 to 50 tokens. We use a *Validation Set* with 5000 strings of length 2 to 50 tokens. We use a *Long Test Set* with 5000 strings of lengths 52 to 100 tokens. The disjoint Training, Validation and Long Test Sets are generated in the same way as in Suzgun et al. (2019a).

We use single-layer single-cell LSTMs, ReLUs and GRUs in our experiments. We include GRUs in these experiments, although they cannot perform unbounded counting, to compare how this limitation manifests itself empirically. We train online with the Adam optimizer (Kingma and Ba, 2014), and learning rates of 0.01 for the ReLU and LSTM models, and 0.001 for the GRU models. The training was run using PyTorch in a Linux environment.

We train 10 models of each type for 30 epochs and select the best model (out of the epochs) per run by validation loss. We observe convergence typically within 20 epochs of training, often less. We experimented with training for 100 epochs on a subset without observing any additional benefit. For ReLUs we train 35 models and select the 10 best runs, as many ReLUs do not learn to accept Dyck-1 strings at all.

5.2 Generalisation to Very Long Strings

We present the prediction performance results of our experiments in Table 5.1. We observe that all three models perform well on both Training and Validation Sets, but there is more variation on the Long Test Set. We also observe in Table 5.1 that ReLUs tend to show very variable performance. LSTMs perform best on average, and GRUs are on average between LSTMs and ReLUs.

We evaluate the generalisation limits of our models that train well on short strings and generalise well to some longer strings up to 100 tokens. We use the *Very Long*

Table 5.1: For Training, Validation, and Long Test Sets we report the accuracy in percentage of strings where a string counts as correctly accepted when all tokens are correctly classified. The numbers in the table are the averages and (minimum / maximum) over 10 models. For the Very Long strings, we report the First Point of Failure (FPF). A maximum FPF value *none* means that the best model(s) did not fail. The best results per column are highlighted in bold font.

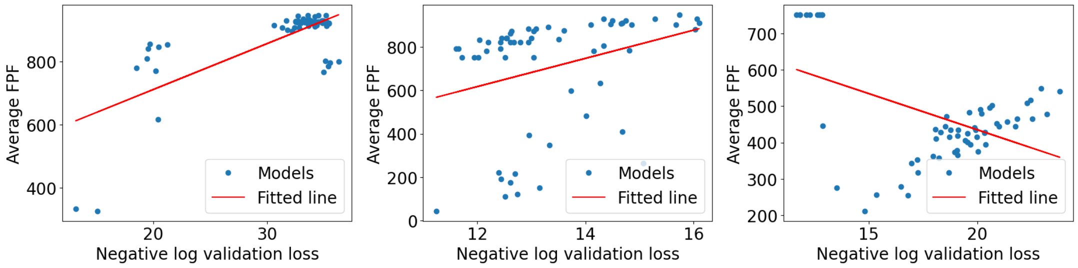
Model	**Training**	**Validation**	**Long**	**Very Long (FPF)**
LSTM	**100** (100 / 100)	**100** (100 / 100)	**100** (99.8 / 100)	**916.8** (802.4 / 946.4)
ReLU	97.9 (90.9 / 100)	97.5 (89.3 / 100)	74.2 (33.5 / 100)	871.7 (543.1 / ***none***)
GRU	**100** (100 / 100)	**100** (100 / 100)	92.5 (90.72 / 95.6)	472.5 (434.7 / 545.2)

Test Set from Dataset 3 which consists of 100 strings of length 1000 tokens, which are generated using same methods as presented in Suzgun et al. (2019a).

The results on the Very Long Test Set are presented in the last column of Table 5.1 and are shown as First Point of Failure because all models but one fail on these strings. Only one ReLU model has an accuracy of 4.0% while all the other ReLUs, the LSTMs and the GRUs do not accept a single Very Long string correctly. Previous work from Weiss et al. (2018a) shows that it is not possible for GRUs to perform unbounded counting, so it is not surprising that their performance deteriorates sharply for the longer strings. ReLUs and LSTMs do have the capacity for unbounded counting, but we observe that training dynamics does not lead to models that count precisely enough in the long run. Interestingly, the only model that accepts at least some of the Very Long strings correctly is a ReLU model, despite their lower average performance. We will discuss the reasons for this behaviour below.

This result does qualify the statement by Suzgun et al. (2019a) that LSTMs can perform dynamic counting, insofar as the counting does not generalise indefinitely. This is actually not surprising given the results by Weiss et al. (2018a), where LSTMs fail to generalise to very long string related tasks, which can be explained by counting not being precise enough, so that the errors accumulate. For example, if an increment adds a value of 1 and a decrement subtracts a value of 0.95 from the count, as the sequence gets longer, the difference between the total increments and total decrements increases and eventually results in the model producing an incorrect label.

5.3 Loss and Generalisation



(a) LSTM (b) ReLU (c) GRU

Figure 5.1: Scatter plot with regression lines of negative log of the average validation loss versus the average FPF on the Very Long Test Set. We use a set of 60 models from different stages of the training process of for each of ReLU, LSTM and GRU (see text for details).

In practice, finite counting abilities may be sufficient, hence it would be useful if it is possible during training (through either the training or validation dynamics) to gather sufficient signals in order to understand how well a model will generalise. This is because repeatedly testing on very long strings can be expensive. In this section, we study the utility of training dynamics towards answering the above question. We use the observation that lower loss values achieved by the models seem to be related to higher FPF values. We quantify this in a linear regression between the negative log loss on Training, Validation and Long Test Sets and the FPF on the Very Long Test Set. We use here a different set of models, to include different stages of training with different levels of validation loss. We take from each of the 10 runs the models after epoch 1, 5, 10, 15, 20, and 25, so that we have 60 models for every type. This is done in order to have a range of models including some where the loss values are relatively high because

Table 5.2: Linear regression between the loss values on the different datasets (Training, Validation and Long Test Sets) and the FPF on the Very Long Test Set with R^2 and p values.

	Training	**Validation**	**Long**
Model	R^2 / p	R^2 / p	R^2 / p
LSTM	0.434 / 1.06×10^{-8}	0.511 / 1.412×10^{-10}	0.593 / 6.36×10^{-13}
ReLU	0.002 / 0.75	0.094 / 0.017	0.201 / 0.0003
GRU	0.363 / 3.55×10^{-7}	0.218 / 0.00017	0.075 / 0.035

the training was still at the beginning. The R^2 and p values are reported in Table 5.2.

We observe that the losses and FPF are well correlated for the LSTMs. For the ReLUs, we see increased correlation for the losses on datasets with long strings, but overall the correlation is low. The GRUs show good correlation on the Training and Validation Sets, but lower correlation values for the Long Test Set, which is plausible given the lower performance on the Long Test Set.

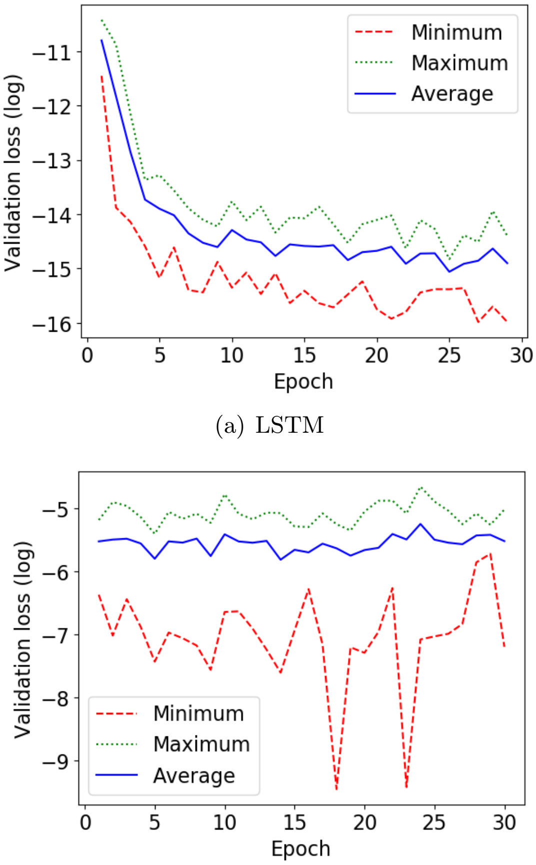
Figure 5.1 shows scatter plots with the regression lines for the Validation Set loss. For the LSTMs we see that the lower loss (higher negative log loss) does lead to higher FPF, but there are still many values that are 20-25% lower than the length of the strings. The ReLUs are much more variable, so that a low validation loss does not guarantee a high FPF. The GRUs, are interesting, where we observe that there are a few models with good FPF but relatively high loss. These models lead to a negative correlation overall and even for very low losses the FPF values are relatively low.

5.3.1 Effect of Training Duration on Loss

Given that low validation losses correlate with high FPF, a possible strategy for improving long term counting behaviour could be to train for more epochs to reduce the loss and thus extend the duration of correct counting. We show the progression of the validation loss during training for LSTM and ReLU models in Figure 5.2. We do not include GRUs as they do not have the capacity for precise counting, so that this strategy does not apply in any case.

The LSTM training behaves normally, i.e. the loss value tends to decrease throughout training for the LSTM, but the reduction becomes much slower as the training progresses. For ReLUs, there is no clear trend, but strong variations throughout.

In any straightforward solution of exact counting with LSTMs, like the one proposed by Weiss et al. (2018a), it is necessary to saturate some of the activation functions. In a cursory inspection we find only a few networks where any saturation occurs. In all the experimentation we do in the context of this paper, we do not encounter a single LSTM model that actually accepts any Very Long sequence, even with longer training. Thus it seems unlikely that saturation is achievable when training within common ranges.



(a) LSTM

(b) ReLU

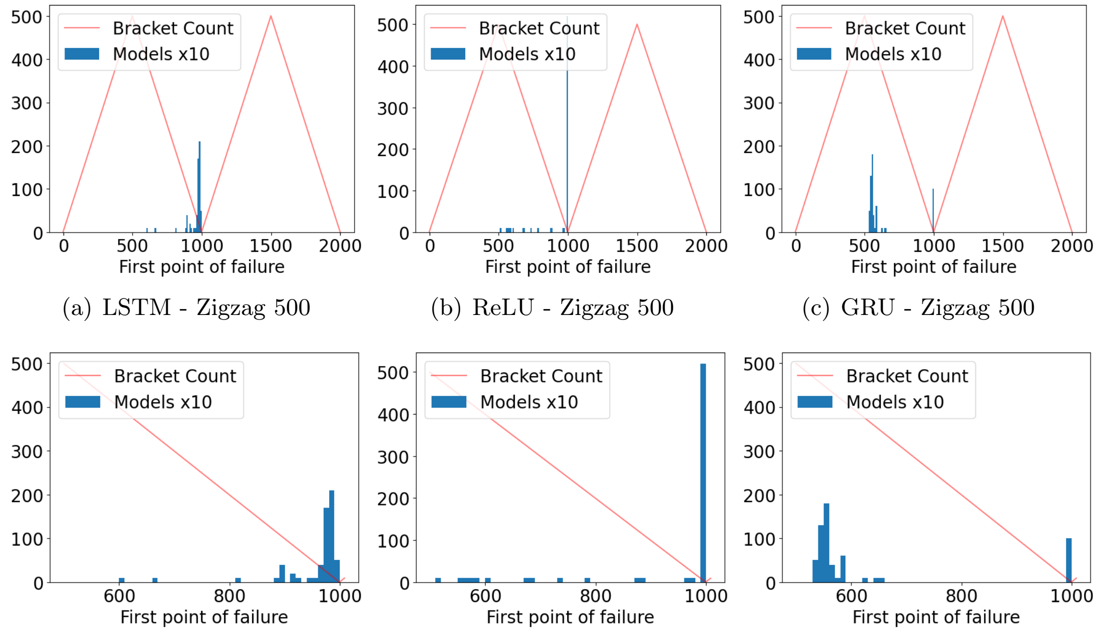
Figure 5.2: Progression of the validation loss during training for LSTM and ReLU models. We show average, minimum and maximum loss. The LSTMs show effective training while the ReLU results mostly stagnate.

Overall, extensive training is not a practical approach to achieve long term counting behaviour as it computationally expensive and not reliable.

5.4 Failure Modes

When trying to understand why and how the RNN models fail to count correctly for long strings, it is worth considering where the counting is represented within the RNN. In the single-cell ReLU network, the activation h_t of the ReLU is the only suitable variable. For the LSTM, the variable c_t (in the notation used by Weiss et al. (2018a), see Equation 2.10) is the only one that is not the result of a squashing function, and it has empirically been shown to perform counting in their experiments.

The only actual test needed of the counter is the comparison to zero to test whether



(a) LSTM - Zigzag 500 (b) ReLU - Zigzag 500 (c) GRU - Zigzag 500

(d) LSTM - Detail (e) ReLU - Detail (f) GRU - Detail

Figure 5.3: Histogram of the FPF distribution for the Zigzag string with $j = 500$. The red line represents the depth of the string (number of opening brackets without a closing bracket), and the histogram represents the number of models failing at different positions. The second row shows details of the distribution from position 500 to 1000, illustrating different failure modes of the different RNN types (see text for a detailed discussion).

the string is a complete Dyck-1 string (or $a^n b^n$ in the experiments by Weiss et al. (2018a)). In order to achieve zero activation at the appropriate points in the string, the increment when processing an opening bracket needs to be of the same absolute value but opposite sign as the decrement when processing a closing bracket. The datasets used in these experiments contain only valid Dyck-1 strings, so that the counter values should not become negative therefore the ReLU should behave linearly. For a linear model, the order of the tokens plays no role. In the solution provided by Weiss et al. (2018a) for the LSTM it is necessary that several variables saturate the activation functions to reach 1 or -1 for exact counting. In particular, if f_t (in their notation) does not saturate, the counter state will be reduced over time, independent of the opening or closing brackets. For GRUs, it is not possible to consistently have the same increment and decrement as discussed in Weiss et al. (2018a).

We expect these differences in the operation to result in different behaviours that will mainly occur around the points where counter should be 0, because the Dyck-1 string is complete. For this purpose we use the *Zigzag Set* that consists of repetitions of j opening brackets followed by j closing brackets for $j = \{10, 20, 25, 50, 100, 125, 200, 250, 500, 1000\}$. All strings in the Zigzag Set are 2000 tokens in length, and distinct. We test our 60 models for each RNN type at various stages of training, as in the previous section, on the Zigzag Set and record the FPF achieved for each model on each element of the dataset. The results on the Zigzag Set are that LSTM and GRU models perform better on the strings with smaller j, showing evidence of a non-linear influence of the counter value (which is proportional to the bracket count, i.e. the number of open brackets without a corresponding closing bracket).

To illustrate the different behaviours, we plot in Figure 5.3 the distribution of the FPF values for the Zigzag string with $j = 500$ and show in detail graphs positions 500-1000. At position 1000 we have a complete Dyck-1 string, so that the counter should be 0 and the RNN should predict the corresponding class. The LSTM models fail mostly at a short distance before position 1000 (Figure 5.3 (a) and (d)), indicating that the decrements are consistently greater by absolute than the increments. ReLUs fail mostly at position 1000, indicating that the decrement is less by absolute than the increment, but they also fail before (Figure 5.3 (b) and (e)), indicating that the position of failures is not dependent on the string depth. The GRUs' increment and decrement values vary throughout the string with the decrements relatively large when the counter value is greater. This leads to a failure soon after closing bracket string starts (Figure 5.3 (c) and (e)).

The GRU behaviour confirms the known limitations. The observations of the ReLUs confirm that they can have decrements greater or smaller than increments and independent of the string depth, but that finding correct weights by backpropagation with gradient descent is difficult. LSTMs confirm that exact counting does not occur in a normal training setup and saturation of the activation functions is rarely observed, preventing straightforward solutions.

This is a potential explanation for the fact that the best model in our experiments was a ReLU model. In the ReLU, saturation of an activation function is not possible and not needed, so that it is possible for the learning process to occasionally reach a good weight configuration even if the training process is not very effective on average.

5.5 Summary

In this chapter, we investigate the generalisation of RNNs trained to accept Dyck-1 languages. The counting behaviour of the trained models is effective on strings up to double the length of the training strings, but almost always fails on strings of 20 times the length (1000 tokens in this case). While Training, Validation and Long Test Set losses are predictive for very long term generalisation, the correlation is not perfect and training for longer to achieve better generalisation is not a practical solution as long training is not efficient on LSTMs and not reliable on ReLUs. The different failure patterns of LSTMs, ReLUs and GRUs correspond to our hypotheses of their operation and shows that despite the theoretical capacity of LSTMs and ReLUs, the networks do not reach exact solutions even in the simple noiseless test cases we tried.

Part II

Theoretical Analysis of Linear and ReLU RNNs

Chapter 6

Theoretical Analysis: Context

In Part I we empirically evaluate RNN learning and generalisation of counting behaviour using Dyck-1 languages as a framework. We find that the RNNs do not learn exact counting behaviour in practice and consequently fail to generalise to arbitrarily long sequences. In this part, we address the second of the 3 overarching questions stated in Chapter 1: under what conditions do RNNs count exactly? We address this question mathematically by relating formal languages and formal automata to RNNs. We explain the motivation for this part in section 6.1. Several earlier studies reveal fundamental theoretical findings about the capacity of NNs, which we discuss in section 6.2. Relating RNNs to formal languages and formal automata is often done using automata learning, which is discussed in section 6.3, or by theoretically analysing RNNs, which is discussed in section 6.4. The directly related work is outlined in section 6.4, while the earlier sections cover literature that provides background but is not directly related. In this part, we focus on theoretically relating single-cell linear RNNs and ReLU RNNs to formal counter automata in order to determine conditions on their weights that lead to counting behaviour. We focus on single-cell linear RNNs in Chapter 7, and on single-cell ReLU RNNs in Chapter 8.

6.1 Motivation

As previously stated, in theoretical computer science formal automata have long been used to model different computations and processes. Formal automata can be defined by the formal languages that they accept, and formal languages can be assigned to their corresponding level in the Chomsky hierarchy based on the automata that can be used to accept them (Chomsky, 1956). Therefore, mathematically relating RNNs to formal automata can help us better interpret and understand the behaviour of RNNs.

In Part I, we used Dyck-1 acceptance tasks to empirically test the ability of different RNNs to learn counting behaviour and found that learning exact counting is difficult with standard training with backpropagation.

As the next step to understanding counting behaviour in RNNs we aim to determine the weight conditions that result in exact counting behaviour. The conditions under which RNNs can count exactly and generalise to arbitrarily long sequences have not yet been established. Therefore, we focus on performing rigorous analysis to determine conditions on the weights of single-cell linear and ReLU RNNs that indicate exact counting behaviour.

6.2 Early Theoretical Findings about NNs

For many years there has been an interest in the theoretical abilities of NN models. An early prominent theoretical finding was by Minsky and Papert (1969) that a perceptron cannot be used to compute the XOR logic function. Rumelhart et al. (1986) address this problem and show that multilayer perceptrons can compute the XOR function and learn to do so by applying the chain rule. This consequently resulted in the widespread use of backpropagation. More recently, it has been mathematically proven that feed-forward neural networks are universal approximators (Cybenko, 1989; Funahashi and Nakamura, 1992). RNNs are also universal approximators (Schäfer and Zimmermann, 2006), and have also been shown to be Turing complete given unbounded precision and time with a sigmoid activation function (Siegelmann and Sontag, 1992). More recently, RNNs with ReLU activations were also shown to be Turing complete (Chen et al., 2018).

6.3 Automata Learning

Automata learning is an approach used to understand the behaviour of black-box functions by relating them to automata. Inputs are fed into the black-box and the outputs are analysed. This is commonly referred to as automata extraction. One well known automata learning algorithm is the L^* algorithm (Angluin, 1987), which is created to extract finite state automata and regular languages from black-box functions. Automata extraction is commonly used to understand the behaviour of trained RNNs.

The L^* algorithm is used in several studies to extract automata from trained RNNs. One example is the study by Weiss et al. (2018b) who extract Deterministic Finite Automata (DFA) from trained RNNs. They analyse the outputs of a trained RNN when longer sequences are used as inputs. They find that the automaton extracted from the RNN using longer sequences is not the same as the automaton that was used to generate the formal language.

Spectral learning is another approach used in automata learning. It is used to extract weighted automata. Ayache et al. (2019) use a spectral learning algorithm to extract weighted automata from black-boxes that compute real-valued functions from sequences of symbolic data. They find that linear projections defined by weighted automata are similar to non-linear projections of RNNs.

Rabusseau et al. (2017) introduce vector-valued Weighted Finite Automata (vv-WFA) and a corresponding spectral multitask learning algorithm. This multitask learning algorithm discovers a representation space that is shared by the different tasks being learned. They extend this multitask learning algorithm to relate WFAs and second-order RNNs with linear activation functions in Rabusseau et al. (2019). They prove that second order RNNs with linear activation functions are equivalent to vv-WFAs with continuous input vectors.

The previously mentioned papers all focus on extracting regular languages and finite automata from RNNs. The tasks we use in our experiments throughout this thesis are counting tasks formalised as Dyck-1 acceptance. Yellin and Weiss (2021) aim to extract context-free grammars, including Dyck languages, from trained RNNs in a way that results in successful generalisation to sequences of arbitrarily length and nesting depth.

They find that the extraction of Dyck languages from RNNs is successful to some extent. However, some of the productions in the grammar were not extracted, such as nesting relations.

6.4 Theoretical Analysis of RNNs

A theoretical approach can be used to analyse RNNs. Using a theoretical approach can help us determine the expressive capacity of RNNs and also establish connections to different automata. In section 3.2 we describe different literature that relates RNNs to formal languages and formal automata. In section 3.3, we focus on describing theoretical work on counting in RNNs.

Peng et al. (2018) propose the notion of rational recurrences to establish connections between the update functions of RNNs and weighted finite state automata. They explore possible approaches to derive neural models from weighted finite state automata, and consequently design more interpretable neural models. They define a rationally recurrent RNN as an RNN with a recurrent state that can be represented by a weighted finite automaton.

Another approach is to relate an NN model to an automaton and mathematically prove the connection. Merrill (2020) follow from the definition of k-counter machines by Fischer et al. (1968) and establish a hierarchy of counter machines, with focus on language recognition. For each of the counter machine variants, they detail the the respective abilities and limitations.

Weiss et al. (2018a) prove that RNNs with squashing (bounded) activation functions do not have the capacity to count indefinitely with finite precision activation values. They have also shown that LSTMs and ReLU RNNs are stronger than Elman RNNs with tanh activation functions and GRUs, and can more readily be used to implement counting behaviour. They explain how saturated LSTMs can be used to count exactly, and as a result, count indefinitely. Similarly, Merrill (2019) formalise the concept of saturated (or asymptotic) RNNs, where sigmoid functions are replaced with step functions. They show how asymptotic LSTMs can be used to exhibit counting behaviour.

6.5 Summary

In this chapter, we review the different approaches of relating RNNs to automata. We start by describing early theoretical work on NNs in section 6.2. We then describe different literature that focuses on automata learning to relate RNNs to formal automata in section 6.3. Finally, we review differernt theoretical studies that focus on the analysis of RNNs in section 6.4.

In this part, we relate single-cell linear RNNs and ReLU RNNs to counter automata. We then propose two theorems defining conditions on the weights of the models to achieve counting behaviour. We then empirically evaluate the models to test whether they find these conditions in training.

Chapter 7

Theoretical Conditions and Empirical Failure of Bracket Counting on Long Sequences with Linear Recurrent Networks

In this chapter, we examine the behaviour of the simplest form of RNNs: a linear single-cell RNN. We theoretically identify the necessary conditions for a linear RNN to have the ability to count, and explore how these conditions relate to the empirical behaviour of trained linear RNN models. We identify two conditions that indicate counting behaviour in linear RNNs and prove that these Counter Indicator Conditions (CICs) are necessary and sufficient for exact counting behaviour to be achieved. We then show empirically that linear RNNs generally do not learn exact counting and do not meet the CICs; and finally, we show empirical relationships between the length of the training strings and the CIC value distributions. This chapter is a modified version of El-Naggar et al. (2023a).

7.1 Counting Behaviour in Linear RNNs

In this section we formally define the Balanced Bracket Language, Balanced Bracket Counter and Linear Recurrent Network and we identify conditions for the network weights that indicate that the Linear Recurrent Network will behave as a Balanced Bracket Counter. We base our counter definitions on the General Counter Machine (GCM) as defined by Merrill (2020) (see Definition 2.9).

The GCM is defined by a vocabulary Σ, finite set of states Q, initial state q_0, counter update function u, state update function δ, and acceptance mask F. From Definition 2.9,

$$u : \Sigma \times Q \times \{0,1\}^k \to (\{+m : m \in \mathbb{Z}\} \cup \{\times 0\})^k$$

$$\delta : \Sigma \times Q \times \{0,1\}^k \to Q$$

Some components, such as states, can be empty. The counter computation also uses a zero check function. An input string x is processed by the counter one token x_t at a time. The counter update function u is used to update the counter value $\mathbf{c} \in \mathbb{Z}$ with integer increments $(+m)$. The counter updates are dependent on the current input token, the current state, and a finite mask of the current state. In our setting, a counter machine is said to accept a string if $\mathbf{c} = 0$ after the whole string is has been processed. A counter machine M is said to accept a language L if $M \text{accepts} \, s \iff s \in L$.

We focus on strings consisting of one type of bracket: $\Sigma = \{\langle, \rangle\}$. For a set w, let $card(w) = |w|$, which is the cardinality of the set. For sequence s of length T, Let $\text{count}(\langle, s) = card(\{t|s_t = \langle\})$, and $\text{count}(\rangle, s) = card(\{t|s_t =\rangle\})$, where $t = [1, ..., T]$ and $\text{count}(\langle, s) \in \mathbb{N}$ and $\text{count}(\rangle, s) \in \mathbb{N}$.

Definition 7.1. (Balanced Bracket Language BB)
The Balanced Bracket Language BB is defined as

$$BB = \{s \in \Sigma^* |\ \text{count}(\langle, s) = \text{count}(\rangle, s)\}.$$

The order of the opening and closing brackets does not matter for the BB language, only that the number of opening and closing brackets in a string is equal overall. Dyck-1 strings are a special case of BB strings where the number of closing brackets is never greater than the number of opening brackets at any point in the string, i.e. for all prefixes.

Our focus is on the counting abilities of single-cell linear RNNs. These networks do not have the capacity to accept Dyck-1 strings from the universe of all possible strings, because they would need to treat negative counts differently from positive activations to distinguish correctly ordered from incorrectly ordered bracket strings. However, that is not possible with a single linear neuron, and additional mechanisms would be needed to fully accept a Dyck-1 language from the entire universe of possible strings.

Previous work, such as Suzgun et al. (2019a), who train their single-cell RNN models to learn Dyck-1 languages only use valid Dyck-1 strings in their datasets, where there are never any excess closing brackets at any point in the strings. This seems unnecessarily limiting, however as the models are only exposed to a subset of possible bracket sequences where excess closing brackets never occur. This does not evaluate the ability of the model to accept Dyck-1 from Σ^*, which would require the detection of excess closing brackets to distinguish between valid and invalid Dyck-1 sequences. Therefore, we use the BB language which can be accepted from $\{\langle, \rangle\}^*$ by a single-cell linear RNN as we will show below.

Definition 7.2. (Balanced Bracket Counter)
A General Counter Machine is a Balanced Bracket Counter iff it accepts BB.

Definition 7.3. (Linear Recurrent Network)
A Linear Recurrent Network (LRN) is a network which receives an input x_t at every timestep t, which is used along with the activation from the previous timestep h_{t-1} and weights W, U, and W_b to produce activation h_t, which is then passed on to the next timestep with the update function:

$$h_t = Wx_t + Uh_{t-1} + W_b.$$

W is a vector of the same dimensionality as x_t, while U, h_{t-1} and W_b are scalars. Here, x_t is a one-hot-encoded input token, an LRN is similar to a stateless counter machine if we apply a zero check z function to h_t after processing the last input. The value h_t in an LRN corresponds to $\mathbf{c}$ in a stateless counter machine, and because only h is propagated from one timestep to the next, there is no state. A counter based on the LRN deviates from the definition by Merrill (2020) in that:

- The counter value $\mathbf{c}$ corresponds to h_t (it is the only value that propagates from one time step to the next), which is real instead of integer,
- The results of the update function ($+m$ in the Counter Machine) are real numbers, which we define as a and b, which are scalar values. The values a and b are defined for analysis to represent the value added to the recurrent value h_t in the LRN in the case of inputs $\langle$ and $\rangle$, respectively. Specifically:

  $a = Wx_t + W_b$ if $x_t = \langle$, and

  $b = Wx_t + W_b$ if $x_t = \rangle$.

We use a single-cell LRN for bracket counting. As a result, W is a vector of the same dimensionality as x_t and U, W_b and h_t are scalars.

In Theorem 7.1, we relate Balanced Bracket Counter behaviour of an LRN to specific conditions on its *weights*. We define two Counter Indicator Conditions (CICs) and show that they are necessary and sufficient for exact counting behaviour to be achieved in an LRN.

Theorem 7.1. (Linear RNN Counter Indicator Conditions (CICs))
The following two Counter Indicator Conditions are necessary and sufficient for a Linear Recurrent Network to accept the Balanced Bracket Language BB.

1. $\frac{a}{b} = -1$ (AB ratio)
2. $U = 1$ (recurrent weight).

Proof 7.1. We prove that the Counter Indicator Conditions in Theorem 7.1 are necessary and sufficient to accept the Balanced Bracket Language with a Linear Recurrent Network. We first prove that the CICs are necessary (Part 1) and then that they are sufficient (Part 2).

Table 7.1: Input sequences used to derive the CICs from Theorem 7.1.

Case	**Input**	**Output** (h)	**Findings**
1	$\langle$	$h_1 \neq 0$	$a \neq 0$
2	$\rangle$	$h_1 \neq 0$	$b \neq 0$
3	$\langle\rangle$	$h_2 = 0$	$b = -Ua$ and $U \neq 0$
4	$\langle\langle$	$h_2 \neq 0$	$U \neq -1$
5	$\langle\langle\rangle\rangle$	$h_4 = 0$	$b + Ub + U^2 a + U^3 a = 0$
6	$\langle\rangle\langle\rangle$	$h_4 = 0$	$b + Ua + U^2 b + U^3 a = 0$

Part 1: We prove that the Counter Indicator Conditions in Theorem 7.1 are satisfied if a Linear Recurrent Network accepts the Balanced Bracket Language by using different input strings (Table 7.1), from which we derive the CICs. If a Linear Recurrent Network accepts the Balanced Bracket Language, then equal numbers of opening and closing brackets result in an output activation $h_t = 0$, otherwise, $h_t \neq 0$. This is equivalent to zero check function $z(h_t)$ yielding 0 or 1. Therefore, we will not include the zero-check function in the following derivation.

Case 1: $seq = \langle, h_0 = 0, h_1 \neq 0$

$h_1 = a + Uh_0 = a$

$\therefore a \neq 0$

Case 2: $seq =\rangle, h_0 = 0, h_1 \neq 0$

$h_1 = b + Uh_0 = b$

$\therefore b \neq 0$

Case 3: $seq = \langle\rangle, h_2 = 0$

$h_2 = b + Ua$

$b + Ua = 0$

From cases 1,2: $a \neq 0$ and $b \neq 0$

$\therefore b = -Ua$, and $U \neq 0$

Case 4: $seq = \langle\langle, h_2 \neq 0$

$h_2 = a + Ua$

$a + Ua \neq 0$

$\therefore U \neq -1$

Case 5: $seq = \langle\langle\rangle\rangle, h_4 = 0$

$h_3 = b + Uh_2 = b + U(a + Ua)$

$h_4 = b + Uh_3 = b + U(b + U(a + Ua))$

$\therefore h_4 = b + Ub + U^2a + U^3a = 0$

Case 6: $seq = \langle\rangle\langle\rangle, h_4 = 0$

$h_3 = a + Uh_2 = a + U(b + Ua)$

$h_4 = b + Uh_3 = b + U(a + U(b + Ua))$

$\therefore h_4 = b + Ua + U^2b + U^3a = 0$

Combine the findings from cases 5 and 6.

$b + Ub + U^2a + U^3a = b + Ua + U^2b + U^3a$

Subtract $b + U^3a$ from both sides and divide both sides by U

$b + Ua = a + Ub$

Rearrange and factorise

$b - a = U(b - a)$

As a result, we get 2 possible situations:

(a) $U = 1$, which implies $a = -b$ by case 3

(b) $a = b$, where by cases 1 and 2 we know $a \neq 0$ and $b \neq 0$, and $U = -1$ follows from case 3, which contradicts case 4

$\therefore U = 1$ and $a = -b$, i.e., the Counter Indicator Conditions listed in Theorem 7.1 hold, if a Linear Recurrent Network accepts the Balanced Bracket Language.

Part 2: We prove by induction that if the Counter Indicator Conditions listed in Theorem 7.1 hold, a Linear Recurrent Network accepts the Balanced Bracket Language.

Each string consists of n opening brackets and m closing brackets, and the input token x_k is either $\langle$ or $\rangle$.

Base Case: $k = 1$

- $x_1 = \langle, n = 1$ and $m = 0$:

$h_1 = a + Uh_0$

$h_1 = a$

- $x_1 =\rangle, n = 0$ and $m = 1$:

  $h_1 = -a + Uh_0$

  $h_1 = -a$

For n opening and m closing brackets, the following equation satisfies the base case, and is therefore our induction hypothesis:

$$h_k = (n - m) \times a$$

We assume that this is true for strings of length k consisting of n opening brackets and m closing brackets. We prove by induction that if this holds for strings of length k tokens, it holds for strings of length $k + 1$ tokens. In our induction step, we use once $x_{k+1} = \langle$ and once $x_{k+1} =\rangle$.

Induction Step:

- If $x_{k+1} = \langle$:

  $h_{k+1} = ((n + 1) - m) \times a$

- If $x_{k+1} =\rangle$:

  $h_{k+1} = (n - (m + 1)) \times a$

From the premise, we can derive that:

$h_k = a + Uh_{k-1}$ if $x_k = \langle$, and $h_k = -a + Uh_{k-1}$ if $x_k =\rangle$

- If $x_{k+1} = \langle$:

  $h_{k+1} = a + h_k$

  Substitute $h_k = (n - m) \times a$

  $h_{k+1} = a + ((n - m) \times a)$

  $\therefore h_{k+1} = ((n + 1) - m) \times a$

- If $x_{k+1} =\rangle$:

  $h_{k+1} = -a + h_k$

Substitute $h_k = (n - m) \times a$

$h_{k+1} = -a + ((n - m) \times a)$

$\therefore h_{k+1} = (n - (m + 1)) \times a$

Therefore, we prove that if the Counter Indicator Conditions listed in Theorem 7.1 are satisfied in a Linear Recurrent Network, it accepts the Balanced Bracket Language. □

Next, we are going to show that it follows from this theorem and proof that an LRN cannot accept Dyck-1. We introduce the function $D(s) \in \mathbb{Z}$, which is the difference between the number of opening and closing brackets in a sequence s, formally,

$$D(s) = \text{count}(\langle, s) - \text{count}(\rangle, s).$$

We express the difference between the number of opening and closing brackets in a prefix of length t in a sequence s of length T as $D(s, t) \in \mathbb{Z}$, formally,

$$D(s,t) = \text{count}(\langle, s_t) - \text{count}(\rangle, s_t).$$

Definition 7.4. (The Dyck-1 Language)
The Dyck-1 Language is represented using the following grammar,

$$S \longrightarrow \epsilon | \langle S \rangle | SS.$$

Dyck-1 also possesses the following property,

$$\text{Dyck-1} \subseteq BB.$$

It follows immediately from the grammar that Dyck-1 is a subset of BB where there are never any excess closing brackets at any point in a sequence:

$$\text{Dyck-1} = \{s | s \in BB \text{ and } D(s_t) \geq 0 \ \forall t \in T\}.$$

Corollary 7.1. A Linear Recurrent Network cannot accept the Dyck-1 language.

Proof 7.2. We prove by contradiction that an LRN cannot accept Dyck-1. We start by assuming that an LRN $\mathcal{L}$ accepts Dyck-1. The sequences in Table 7.1 are preconditions for Part 1 of Proof 7.1 and the inputs listed are either in the intersection between BB and Dyck-1, or in the intersection of their complements. Therefore, the input-output relations in Table 7.1 hold for Dyck-1 as well as BB. By Part 1 of Proof 7.1, if the preconditions hold for $\mathcal{L}$, then the CICs hold for $\mathcal{L}$. By Part 2 of Proof 7.1, if the CICs are satisfied for $\mathcal{L}$, then LRN $\mathcal{L}$ accepts BB. This is a contradiction to the assumption that $\mathcal{L}$ accepts Dyck-1, as $\mathcal{L}$ can only accept one language. For example, given the sequence $\rangle\langle$ which is in BB but not in Dyck-1, $\mathcal{L}$ will produce a label of 0 because this sequence is in BB, and can therefore not produce a non-zero label indicating that the sequence is not in Dyck-1. Therefore, an LRN cannot accept Dyck-1. □

Table 7.2: Accuracy metrics of our previous binary classification experiments without bias (reported in Table 4.3, and published in (El-Naggar et al., 2022a)), ternary classification experiments without bias, and binary and ternary classification experiments with bias.

Classification Experiment	Train Length	Train Avg(Min/Max)	20 Tokens Avg(Min/Max)	50 Tokens Avg(Min/Max)
Binary (without bias)	2	100 (100/100)	69.2 (6.04/77.3)	69.0 (66.7/72.7)
Binary (without bias)	4	100 (100/100)	94.8 (94.7/95.3)	89.3 (88.7/90.0)
Binary (without bias)	8	100 (100/100)	96.9 (94.0/100)	92.7 (78.7/98.0)
Ternary (without bias)	2	90 (33.3/100)	55.6 (33.3/64.4)	51.4 (33.3/60.0)
Ternary (without bias)	4	100 (100/100)	79.5 (65.8/94.7)	67.2 (66.7/68.0)
Ternary (without bias)	8	100 (100/100)	94.4 (67.1/100)	85.7 (66.7/100)
Binary (with bias)	2	100 (100/100)	73.4 (63.3/100)	72.4 (60.0/93.3)
Binary (with bias)	4	100 (100/100)	95.3 (92.7/98.0)	86.0 (77.3/90.7)
Binary (with bias)	8	100 (100/100)	95.2 (85.3/100)	87.9 (70.0/98.0)
Ternary (with bias)	2	88.3 (66.7/100)	58.0 (38.2/67.6)	54.4 (43.6/67.5)
Ternary (with bias)	4	97.9 (79.2/100)	81.5 (64.4/100)	68.0 (65.3/73.3)
Ternary (with bias)	8	100 (100/100)	95.9 (83.6/100)	76.5 (65.3/100)

7.2 Counting in Linear RNNs in Practice

We conduct experiments to analyse the models and whether or not they satisfy the CICs defined in the previous section in training. We use 2 classification tasks to evaluate our models.

7.2.1 Task 1: Binary Classification

We use the same task and model as in our previous work in Chapter 4 (El-Naggar et al., 2022a), i.e., a linear RNN without biases with a single output neuron with sigmoid activation to classify the bracket difference of the sequence as > 0 or ≤ 0 (binary). The absence of a trainable bias reduces the degrees of freedom in the model, and is equivalent to having a bias (W_b) value that is fixed to 0, hence simplifying the learning task. We also use the same models with trainable biases. This allows us to evaluate model learning when part of the correct solution is fixed within the model and compare it to training the model to learn all the weight values from the data. Here we use the Binary Bracket Difference Dataset (see Dataset 2). The models are trained with strings of lengths 2, 4 and 8 tokens for 100 epochs in 10 runs. The initial count value (h_0) has a value of 0 for every string. We inspect the weights of our models and plot the distribution of the CIC values (a/b,U) of the trained models for each training set size in Figure 7.1. We observe that the models do not fulfill the CICs, but they do approach the CICs as the length of the training strings increases. We observe that the distributions of the a/b indicator have mean values above -1 but less so for longer training strings.

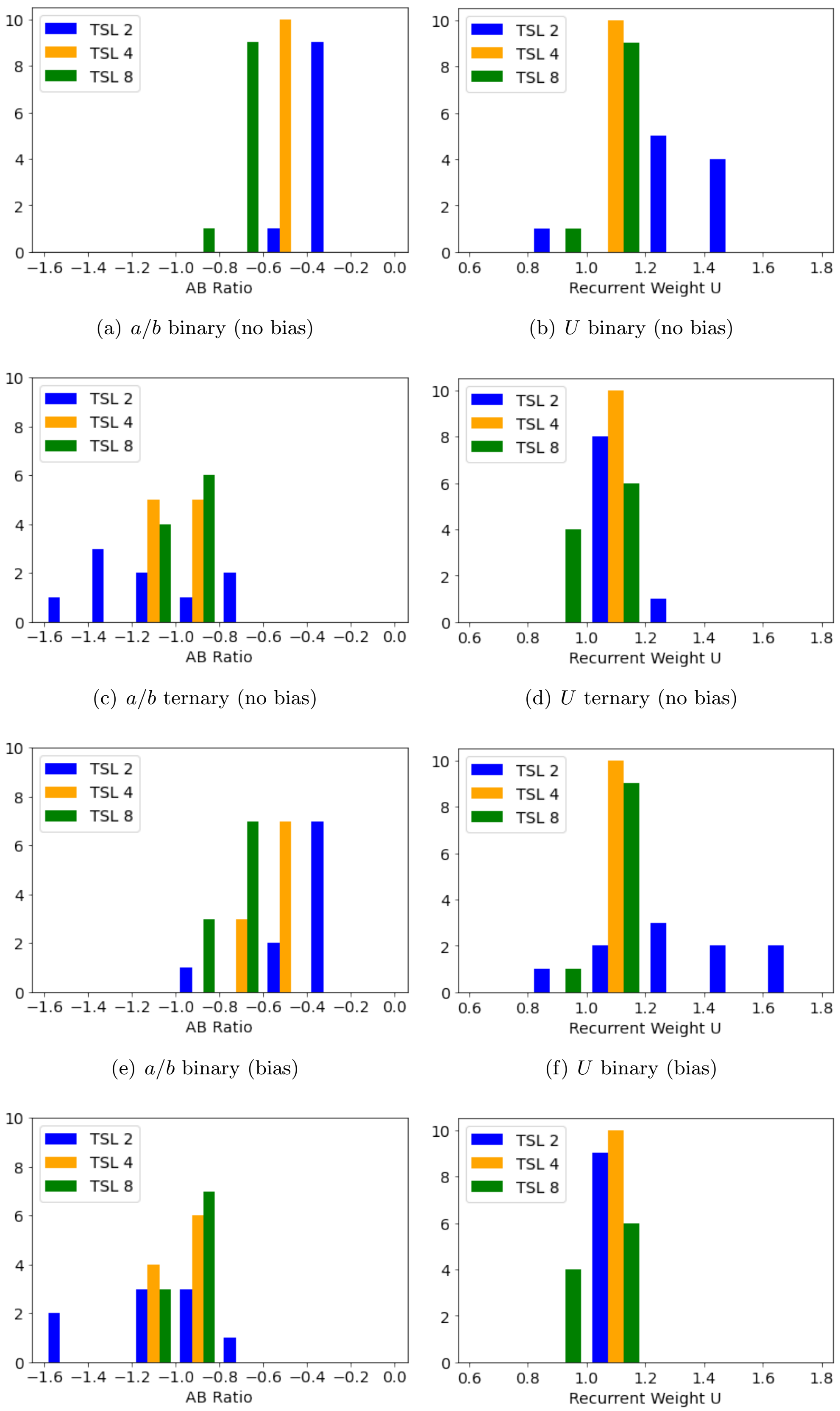
7.2.2 Task 2: Ternary Classification

We also apply a ternary classification: > 0, $= 0$ or < 0. Here we use the Ternary Bracket Difference Dataset (see Dataset 2). We use the same model as in Task 1, except that instead of a single output neuron with a sigmoid output activation, we use 3 output neurons and a softmax output layer with bias, which is the minimal configuration that can achieve this task. We also use the same models with trainable biases.

The initial count value (h_0) has a value of 0 for every string and the models are trained in the same manner as the models from Task 1. The ternary classification accuracy is slightly lower as can be expected with more classes. For shorter training strings, this may be related to the larger number of model parameters relative to the data points. The accuracy improves with longer training strings. Ternary classification does not show a mean of a/b above -1 as can be seen in Figure 7.1.

7.3 Summary

Although linear RNNs have the theoretical capacity for counting behaviour, previous research has shown that these models often struggle to effectively generalise counting behaviour to long strings. In this chapter, we present a set of necessary and sufficient conditions that indicate counting behaviour in linear RNNs and provide proof that meeting these conditions is equivalent to counting behaviour. To investigate the extent to which these conditions are met, we examine the parameters of models trained on strings of varying lengths for classification tasks. We use both binary and ternary classification tasks and find that the models do not fully meet these conditions, but do approach them and get closer as the string length increases.



(a) a/b binary (no bias)

(b) U binary (no bias)

(c) a/b ternary (no bias)

(d) U ternary (no bias)

(e) a/b binary (bias)

(f) U binary (bias)

(g) a/b ternary (bias)

(h) U ternary (bias)

Figure 7.1: CICs after training on binary and ternary classification without biases (top) and with biases (bottom) with different Training Sequence Lengths (TSL).

Chapter 8

Formal and Empirical Studies of Counting Behaviour in ReLU RNNs

In this chapter, we investigate the capability of single-cell ReLU RNN models to demonstrate precise counting behaviour. Formally, we start by characterising the semi-Dyck-1 language and semi-Dyck-1 counter machine that can be implemented by a single Rectified Linear Unit (ReLU) cell. We define three Counter Indicator Conditions (CICs) on the weights of a ReLU cell and show that fulfilling these conditions is equivalent to accepting the semi-Dyck-1 language, i.e. to perform exact counting. Empirically, we study the ability of single-cell ReLU RNNs to learn to count by training and testing them on different datasets of Dyck-1 and semi-Dyck-1 strings. While networks that satisfy the CICs count exactly and thus correctly even on very long strings, the trained networks exhibit a wide range of results and never satisfy the CICs exactly. We investigate the effect of deviating from the CICs and find that configurations that fulfil the CICs are not at a minimum of the loss function in the most common setups. This is consistent with observations in previous research indicating that training ReLU networks for counting tasks often leads to poor results. We finally discuss implications of these results. This chapter is a modified version of El-Naggar et al. (2023b), which was done in collaboration with Andrew Rhyzikov, Laure Daviaud, Pranava Madhyastha and Tillman Weyde. In particular, there was a significant contribution to the formulation and the proofs of the theorems in this chapter by Laure Daviaud and Andrew Rhyzikov.

8.1 Formalising Counting Behaviour in ReLU RNNs

In this section, we ask under which conditions a single-cell ReLU Recurrent Neural Network (ReLU RNN) can count, or more formally under which conditions a single-cell ReLU RNN accepts the semi-Dyck-1 language. The semi-Dyck-1 language is a variant of the Dyck-1 language (which is the language of well-formed bracket strings), adapted to the limitations of a single-cell ReLU RNN. In Theorem 8.1, we give a set of conditions that are necessary and sufficient for this to happen. The definitions used in the following are inspired by the notions of the General Counter Machine and Real-Time Language Acceptance presented in Merrill (2020) (see section 2.4).

8.1.1 Counter Machines and semi-Dyck-1 Language

We first define a particular variant of a counter machine and demonstrate its links with the semi-Dyck-1 language. This specific counter machine aligns with the computational constraints of a ReLU cell, which is incapable of producing *negative* activation values.

Definition 8.1 (Stateless Incremental Non-Negative 1-Counter Machine (SINC))**.** A Stateless Incremental Non-Negative 1-Counter Machine (SINC) is a tuple (Σ, u) where Σ is a finite alphabet and u is a counter update function:

$$u : \Sigma \to \{-1, +1\}$$

where $+1$ (resp. -1) denote the function $f(x) := x + 1$ (resp. $f(x) := x - 1$ if $x > 0$ and $f(0) = 0$) defined on $x \in \mathbb{N}$.

Definition 8.2 (Computations of SINC)**.** A configuration of a SINC is a non-negative integer value (the value in the counter). A computation on input $x = x_1 \cdots x_n$ where $x_i \in \Sigma$ for all $1 \leq i \leq n$ is a sequence of configurations:

$$c_0 \to c_1 \to \cdots \to c_n$$

such that $c_0 = 0$ is the initial configuration, and for all $1 \leq i \leq n$, $c_i = u(x_i)(c_{i-1})$. The value c_n is called the output on x of the SINC.

An alphabet Σ is a finite set of symbols, called tokens and a string on Σ is a sequence of tokens. In this paper we only consider finite strings, so finite sequences of tokens. The language accepted by a SINC is defined as the set of input strings that have output 0. In the following, we use the bracket characters $\langle$ and $\rangle$ to denote the characters in our alphabet, while other brackets have their usual meaning.

Note that a SINC is a particular instance of one-state pushdown automata with a one-token stack-alphabet, or can be seen as a one-state one-counter Minsky machine with an input alphabet.

Definition 8.3 (semi-Dyck-1 counter)**.** A SINC (Σ, u) is said to be a semi-Dyck-1 counter if $\Sigma = \{\langle, \rangle\}$, $u(\langle) = +1$ and $u(\rangle) = -1$.

The Dyck-1 language is the language of well-formed strings of brackets, e.g. $\langle\langle\langle\rangle\rangle\langle\rangle\rangle$. It is defined on the alphabet $\Sigma = \{\langle, \rangle\}$ and generated by the context-free grammar $s \to \varepsilon|\langle s\rangle s$, where ε denotes the empty string.

Definition 8.4 (semi-Dyck-1 language)**.** We define the semi-Dyck-1 language as generated by the context-free grammar $s \to \varepsilon|\langle s\rangle s|s\rangle s$.

This is the language of strings constructed from strings of the Dyck-1 language in which closing brackets can be inserted at any point.

Proposition 8.1. A semi-Dyck-1 counter accepts the semi-Dyck-1 language.

Proof 8.1. Consider a semi-Dyck-1 counter (Σ, u) with $\Sigma = \{\langle, \rangle\}$. Let us first prove that any string in the semi-Dyck-1 language is accepted. Let x be a string in the semi-Dyck-1 language. By definition, $u(\langle) = +1$ and $u(\rangle) = -1$, which means that the counter is incremented by 1 everytime an opening bracket is read and decreased by 1 when a closing bracket is read, unless the counter value is 0, in which case, it remains 0. It is easy to see that the output on x is 0. This can be formally proved by induction on the grammar generating the semi-Dyck-1 language. Indeed,

- The output on ε is 0;
- Suppose now that s and s' are strings in the semi-Dyck-1 language with output 0 and $x = \langle s\rangle s'$. Then, on input x, first the counter is incremented by 1 after the

first opening bracket. Then after reading s its value is either 1 (if s is in Dyck-1) or 0 (if s has at least one more closing bracket than opening ones) by induction hypothesis on s. In any case, after reading the closing bracket the counter has value 0, and after reading s', has again value 0, by induction hypothesis on s', so the output on x is 0;

- Suppose now that s and s' are strings in the semi-Dyck-1 language with output 0 and $x = s\rangle s'$. Then the counter is at 0 after reading s by induction hypothesis, remains at 0 after reading the closing bracket and is hence also at 0 after reading s' by induction hypothesis. So the output on x is 0.

On the other hand, we shall demonstrate that any accepted string belongs to the semi-Dyck-1 language. Consider an input string, x, which is accepted by the semi-Dyck-1 counter. We prove by induction that the output of x represents the quantity of opening brackets that remain unmatched by a closing bracket later on in the string.

- This is clearly true for ε.

- Suppose $x = x'\langle$ with the output on x' being the number of opening brackets that are not matched later on in x' by a closing bracket. Then, by definition of the semi-Dyck-1 counter, the output on x is the output on x' incremented by 1, and hence the induction hypothesis is satisfied for x.

- Suppose $x = x'\rangle$ with the output on x' being the number of opening brackets that are not matched later on in x' by a closing bracket. If this value is 0, then the same goes for x and the induction is satisfied. If this value is positive, then it is decremented by 1 for x, which also closes an opening bracket in x' so the induction is also satisfied.

This is adequate to conclude the proof, as the output of x equals zero, signifying that every opening bracket in x is subsequently matched by a closing bracket. Consequently, x is a member of the semi-Dyck-1 language.

8.1.2 ReLU Recurrent Neural Networks (ReLU RNNs) as Counters

We define here formally a ReLU Recurrent Neural Network (ReLU RNN) and state the conditions that we will prove to be necessary and sufficient for it to behave as a semi-Dyck-1 counter.

Definition 8.5. (ReLU Recurrent Neural Network (ReLU RNN)) A single-cell ReLU RNN is a tuple $(\Sigma, n, u, W, U, W_b)$ where Σ is a finite alphabet, n is a positive integer, u is a mapping from Σ to $\mathbb{R}^n$, W is a vector in $\mathbb{R}^n$ and U and W_b are real numbers. W, U, and W_b are called the weights of the ReLU RNN. A ReLU RNN takes as input a string $x = x_1 \cdots x_n$ on Σ, considering a token per timestep. An output activation function is computed at each timestep and defined as follows: $h_0 = 0$ and for all $t = 1, \ldots, n$,

$$h_t = max(0, Wu(x_t) + Uh_{t-1} + W_b)$$

The product $Wu(x_t)$ is to be understood as the scalar product of two vectors in $\mathbb{R}^n$. The output of the ReLU RNN on input x is h_n.

The language accepted by a ReLU RNN is defined as the set of input strings having output 0. We now fix $\Sigma = \{\langle, \rangle\}$, and given a ReLU RNN $\mathcal{R} = (\Sigma, n, u, W, U, W_b)$, we define the following reals:

$$a_{\mathcal{R}} = Wu(\langle) + W_b \quad \text{and} \quad b_{\mathcal{R}} = Wu(\rangle) + W_b$$

Note that with this definition, the update function becomes $h_t = max(0, a_{\mathcal{R}} + Uh_{t-1})$ if $x_t = \langle$ and $h_t = max(0, b_{\mathcal{R}} + Uh_{t-1})$ if $x_t =\rangle$.

Definition 8.6. (CIC Compliant ReLU Recurrent Neural Network) A single-cell ReLU RNN $\mathcal{R} = (\Sigma, n, u, W, U, W_b)$ is said to be CIC Compliant if it satisfies the three following conditions:

1. $a_{\mathcal{R}} = -b_{\mathcal{R}}$
2. $a_{\mathcal{R}} > 0$
3. $U = 1$

We call these conditions *Counter Indicator Conditions (CICs).* We now state our main result below.

Theorem 8.1. For all single-cell ReLU RNNs $\mathcal{R}$, the three following assertions are equivalent:

- $\mathcal{R}$ is CIC Compliant,
- $\mathcal{R}$ accepts the semi-Dyck-1 language,
- $\mathcal{R}$ accepts the same language as a semi-Dyck-1 counter.

First, we give two lemmas that will be used in the proof of Theorem 8.1.

Lemma 8.1. For all ReLU RNNs $\mathcal{R}$ on $\Sigma = \{\langle, \rangle\}$ accepting the semi-Dyck-1 language, for all positive integers n and non-negative integers m, the output of $\mathcal{R}$ on input $\langle^n\rangle^m$ is:

$$\begin{array}{ll} a_{\mathcal{R}}(1+U+\cdots+U^{n-1}) & \text{if } m=0 \\ a_{\mathcal{R}}U^m(1+U+\cdots+U^{n-1})+b_{\mathcal{R}}(1+U+\cdots+U^{m-1}) & \text{if } n>m\geq 1 \\ \max(0,(a_{\mathcal{R}}U^n+b_{\mathcal{R}})(1+U+\cdots+U^{n-1})) & \text{if } n=m \end{array}$$

Proof 8.2. The first item is proved by induction on n. For $n = 1$, the output on $\langle$ is $\max(0, a_{\mathcal{R}})$ which is $a_{\mathcal{R}}$ since $\langle$ is not in the semi-Dyck-1 language. Suppose now that this is true for some positive integer n. By definition of $\mathcal{R}$ and induction hypothesis, the output on $\langle^{n+1}$ is $\max(0, a_{\mathcal{R}} + U(a_{\mathcal{R}}(1 + U + \cdots + U^{n-1})))$, which is $\max(0, a_{\mathcal{R}}(1 + U + \cdots + U^n))$. Since $\langle^{n+1}$ is not in the semi-Dyck-1 language, this has to be positive so the output is $a_{\mathcal{R}}(1 + U + \cdots + U^n)$.

The second item is proved by induction on m, considering the induction hypothesis is true for all $n > m$. For $m = 1$, for any $n > 1$, the output on $\langle^n\rangle$ is deduced from the previous item and is $\max(0, b_{\mathcal{R}} + Ua_{\mathcal{R}}(1 + U + \cdots + U^{n-1}))$. This has to be positive so the output is $a_{\mathcal{R}}U(1 + U + \cdots + U^{n-1}) + b_{\mathcal{R}}$. Suppose now that this is true for some positive integer m, and let $n > m + 1$. Then, by induction hypothesis, on input $\langle^n\rangle^{m+1}$, the output is $\max(0, b_{\mathcal{R}} + U(a_{\mathcal{R}}U^m(1 + U + \cdots + U^{n-1}) + b_{\mathcal{R}}(1 + U + \cdots + U^{m-1})))$, which is $a_{\mathcal{R}}U^{m+1}(1 + U + \cdots + U^{n-1}) + b_{\mathcal{R}}(1 + U + \cdots + U^m)$ since this has to be positive.

The third item is directly derived from the second and first ones used on input '$\langle^n\rangle^{n-1}$'. If $n = 1$, the output on $\langle\rangle$ is $\max(0, a_{\mathcal{R}}U + b_{\mathcal{R}})$. Otherwise, for all positive integers $n > 1$, the output on '$\langle^n\rangle^n$' is $\max(0, b_{\mathcal{R}} + U(a_{\mathcal{R}}U^{n-1}(1+U+\cdots+U^{n-1}) + b_{\mathcal{R}}(1+U+\cdots+U^{n-2}))$, which is $\max(0, (a_{\mathcal{R}}U^n + b_{\mathcal{R}})(1 + U + \cdots + U^{n-1}))$.

Lemma 8.2. For all strings x on alphabet $\Sigma = \{\langle, \rangle\}$, the output value of a semi-Dyck-1 counter on x is n for some integer n if and only if the output in a CIC Compliant ReLU RNN $\mathcal{R}$ on x is $a_{\mathcal{R}}n$.

Proof 8.3. This follows from the definition of a CIC Compliant ReLU RNN. The activation function is now $h_t = max(0, a_{\mathcal{R}} + h_{t-1})$ if $x_t = \langle$ and $h_t = max(0, -a_{\mathcal{R}} + h_{t-1})$ if $x_t =\rangle$ for some $a_{\mathcal{R}} > 0$. In other words, the activation function is incremented by $a_{\mathcal{R}}$ everytime an opening bracket is read and decreased by $a_{\mathcal{R}}$ when a closing bracket is read, unless the value is 0, in which case, it remains 0. This describes exactly the computation of the counter in a semi-Dyck-1 counter, except that the increment and decrement are by 1 instead of $a_{\mathcal{R}}$.

Proof 8.4. (Proof of Theorem 8.1) The two last items are equivalent by Proposition 8.1.

We prove first that if $\mathcal{R}$ accepts the semi-Dyck-1 language, then it is CIC Compliant, i.e. it satisfies the three conditions from Definition 8.6. On input $x = \langle$, the output is $\max(0, a_{\mathcal{R}})$. Since x is not in the semi-Dyck-1 language, this output has to be positive, hence $a_{\mathcal{R}} > 0$. On input $x =\rangle$, the output is $\max(0, b_{\mathcal{R}})$. Since x is in the semi-Dyck-1 language, this output has to be 0, hence $b_{\mathcal{R}} \leq 0$. We prove now that $U = 1$, using Lemma 8.1.

- On input $\langle\langle$, the output is $a_{\mathcal{R}}(1 + U)$. This has to be positive, hence $U > -1$, since $a_{\mathcal{R}} > 0$.

- For all positive integers n, on input $\langle^n\rangle^n$, the output is $\max(0, (a_{\mathcal{R}}U^n + b_{\mathcal{R}})(1+U+\cdots+U^{n-1}))$ and has to be non-positive. Since $U > -1$, then $(1+U+\cdots+U^{n-1}) \geq 0$, hence $(a_{\mathcal{R}}U^n + b_{\mathcal{R}}) \leq 0$ for all n. Since $a_{\mathcal{R}} > 0$, $b_{\mathcal{R}} \leq 0$ and $\lim_{n\to\infty} U^n = \infty$ if $U > 1$, then necessarily $U \leq 1$.

- For all integers $n \geq 2$, on input '$\langle^n\rangle^{n-1}$', the output is $\alpha_n = a_{\mathcal{R}}U^{n-1}(1 + U + \cdots + U^{n-1}) + b_{\mathcal{R}}(1 + U + \cdots + U^{n-2})$ and has to be positive. If $-1 < U < 1$, then

$\lim_{n\to\infty}(1 + U + \cdots + U^{n-1}) = \lim_{n\to\infty}(1 + U + \cdots + U^{n-2}) = (1 - U)^{-1}$ and $\lim_{n\to\infty} U^{n-1} = 0$, hence $\lim_{n\to\infty} \alpha_n = b_{\mathcal{R}}(1 - U)^{-1} \leq 0$. This contradicts the fact that α_n is strictly positive for all n. Then we obtain that $U = 1$.

Finally, with $U = 1$, using input $x = \langle\rangle$, the output is $\max(0, a_{\mathcal{R}} + b_{\mathcal{R}})$ and is 0, so $a_{\mathcal{R}}+b_{\mathcal{R}} \leq 0$. For all positive integers n, on input $\langle^n\rangle^{n-1}$, the output is $na_{\mathcal{R}}+(n-1)b_{\mathcal{R}} > 0$, and hence $a_{\mathcal{R}} + ((n-1)/n)b_{\mathcal{R}} > 0$. Since $\lim_{n\to\infty} a_{\mathcal{R}} + ((n-1)/n)b_{\mathcal{R}} = a_{\mathcal{R}} + b_{\mathcal{R}}$, then $a_{\mathcal{R}} + b_{\mathcal{R}} \geq 0$. We finally get $a_{\mathcal{R}} = -b_{\mathcal{R}}$.

We finally prove that if $\mathcal{R}$ is CIC Compliant, it accepts the same language as a semi-Dyck-1 counter. This is immediate by Lemma 8.2. A string is accepted by $\mathcal{R}$ if and only if its output is 0 and if and only if the output of a semi-Dyck-1 counter is 0.

8.2 Experiments

Having identified the CICs that determine whether a ReLU RNN is Counting, we study the relationship between the exact and approximate fulfilment of the CICs and empirical behaviour of a ReLU RNN. We also investigate if training ReLU RNNs leads to them reaching or approximating the CICs and evaluate their counting behaviour empirically. Our general setup is similar to that of Gers and Schmidhuber (2001), Weiss et al. (2018a), and Suzgun et al. (2019a).

8.2.1 Datasets and Metrics

To test the counting behaviour of our models, we use the Dyck-1 Prediction Dataset (see Dataset 3) and the semi-Dyck-1 Dataset (see Dataset 4 defined below), with their characteristics shown in Table 8.1. All datasets consist of strings that are each a valid string in their respective languages as a whole.

Dataset 4. (semi-Dyck-1 Datasets) The semi-Dyck-1 datasets are created from the Dyck-1 prediction datasets by replacing every opening bracket with a closing one with probability 0.5. In order to generate strings of odd length, for half the strings we either

Table 8.1: The datasets in our experiments, with different string structures and lengths, each in two versions: Dyck-1 (D) and semi-Dyck-1 (S). We report for all datasets the size (number of strings), lengths of the strings, percentage of valid strings (i.e. count value of 0), the mean and maximal counter value per string averaged over each dataset.

			D valid	**counter**		**S valid**	**counter**	
Type	**size**	**lengths**	**%**	**mean**	**max**	**%**	**mean**	**max**
Training	10,000	2–50	5.5	4.1	8.5	65.3	0.6	2.8
Validation	5,000	2–50	5.2	4.3	8.9	65.3	0.6	2.9
Long	5,000	52–100	2.4	6.9	14.3	64.3	0.6	3.8
Zigzag	10	2,000	1.2	114.0	228.0	53.9	3.0	24.2
Very Long	100	1,000	0.2	27.2	56.1	63.2	0.7	7.4

add a closing bracket at a random position or remove a randomly chosen opening bracket, each with probability 0.5. In this way, the average string length is maintained.

- **Train:** 10,000 semi-Dyck-1 strings of lengths between 2 and 50 tokens.
- **Validation:** 5,000 semi-Dyck-1 strings of lengths between 2 and 50 tokens.
- **Long Test:** 5,000 semi-Dyck-1 strings of lengths between 52 and 100 tokens.
- **Very Long Test:** 100 semi-Dyck-1 strings of length 1,000 tokens.
- **Zigzag:** 10 semi-Dyck-1 strings of length 2000 tokens adapted from the Zigzag Dataset in Dataset 3.

In our datasets, there are two labels at every timestep, as introduced by Gers and Schmidhuber (2001); Suzgun et al. (2019a); Weiss et al. (2018a). For Dyck-1 the labels indicate which next tokens, $\langle$ or $\rangle$, would be possible in the language, i.e. the string at the current timestep concatenated with the indicated token would be a prefix for a valid Dyck-1 string. An opening bracket can occur at any point in a Dyck-1 string, therefore the corresponding label 1 is always 1, but a closing bracket cannot occur at a point when there are no excess opening brackets, therefore label 2, for a closing bracket, is 0 iff there are no excess opening brackets, i.e. the current string is a valid Dyck-1 string. Therefore, this task can also be viewed as a classification task for Dyck-1 validity. Label 1 is obviously redundant in this setting, but we include it to ensure comparability with the literature.

The semi-Dyck-1 datasets are created from the Dyck-1 datasets by replacing every opening bracket with a closing one with probability 0.5. In order to generate strings of odd length, for half the strings we either add a closing bracket at a random position or remove a randomly chosen opening bracket, each with probability 0.5. In this way, the average string length is maintained.

The labels are set to ensure compatibility with the previous experiments and the literature. At every timestep we set the label 1 to 1 and label 2 to 0 if the string up to this point is a valid semi-Dyck-1 string, analogous to the Dyck-1 datasets. For semi-Dyck-1, the interpretation of output neurons as indicating what next token is allowed in the language is not valid in our encoding (we can always add an opening or closing bracket and the string is still a prefix to a valid semi-Dyck-1 string).

This way of changing the datasets to contain strings that are in semi-Dyck-1 but not Dyck-1 has several side effects. It changes the overall proportion of closing brackets from 0.5 to just over 0.75. The class ratio between valid and invalid strings is much more balanced, as can be seen in Table 8.1. The counter values that are needed for processing the strings also changed to much lower values, especially for the Zigzag and the Very Long datasets. Higher values require the model to count more precisely as any deviations from $U = 1$ are multiplied by the counter values and deviations from $a = -b$ accumulate. The semi-Dyck-1 dataset is therefore easier to process and better model performance in tests can be expected.

8.2.2 Experimental Setup and Evaluation

For our experiments, we use a ReLU RNN with a single hidden neuron as shown in Figure 8.1. Output neuron 1 has always target value 1 and neuron 2 indicates whether the string up to the current time is valid in the respective language. Although output neuron 1 has no role in the classification, it is still included in the loss calculation and optimisation, for compatibility as mentioned above. The output neurons have a sigmoid activation function (see Definition 2.1). We experiment with Binary Cross Entropy (BCE) loss and Mean Squared Error (MSE) loss (see Definitions 2.8 and 2.6, respectively). The combination of a sigmoid activation with MSE loss is an unusual combination which

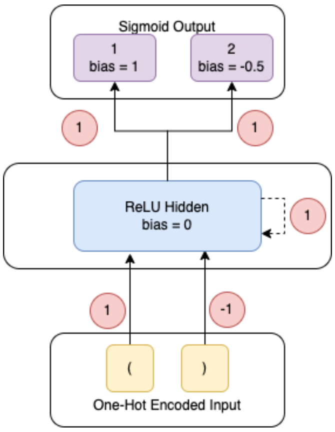


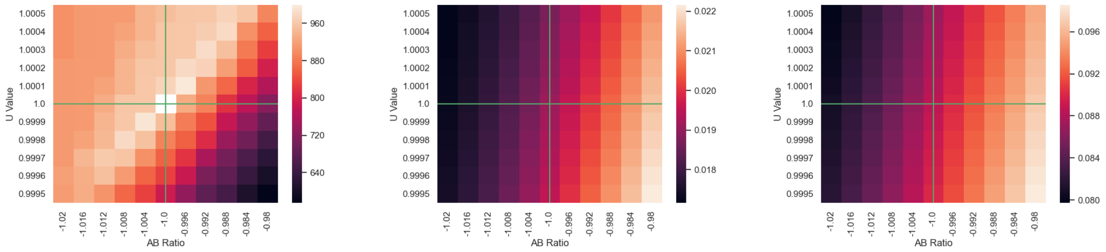
Figure 8.1: ReLU Model with a configuration satisfying the Counter Indicator Conditions.

does not have a probabilistic interpretation like the commonly used cross-entropy, but it is what was used by Gers and Schmidhuber (2001), Weiss et al. (2018a), and Suzgun et al. (2019a) and is retained here for comparability.

We report average Accuracy (number of strings classified correctly at every timestep) as a classification metric, as well as First Point of Failure (FPF), which reflects the exact counting capability better. The FPF is the first point at which a model fails when a string is processed. For each model the average FPF value over the Very Long dataset is reported. FPF tests the generalisation abilities of models for very long strings, where fully correct processing becomes increasingly rare for imperfect models, so that accuracy ceases to differentiate models well.

8.2.3 Validating the CIC Compliant Model

For the correct model configuration, we use input weight vector $[1, -1]$, which, together with a hidden neuron bias of 0, leads to $a = 1$ and $b = -1$, which satisfies the CICs 1 and 2. We also use a recurrent weight $U = 1$, satisfying the CIC 3. The output bias is 1 for output neuron 1 and -0.5 for output neuron 2. We use a threshold of 0.5 for the final classification when calculating accuracy and FPF. As shown in Table 8.3, the correct model achieves perfect results on all datasets.



(a) FPF (b) MSE (c) BCE

Figure 8.2: Heatmaps showing the FPF values on the Dyck-1 Very Long dataset and MSE and BCE loss on the Dyck-1 Validation dataset for models with a correct configuration and with deviations. The thin green lines represent the CIC values for the a/b ratio and U value, and the intersection between the green lines is the point of a correct model. For FPF, the value in the centre of graph (a) is undefined, as no failures occurred for the correct model. It can be seen that the lowest MSE and BCE loss values are not located at the position of the correct model configurations. For a larger version of these heatmaps see Appendix C.

8.2.4 Effect of Deviation from the Correct Model

We systematically vary the a and U values from correct weights to study the effect of the deviation from CIC values on the MSE validation loss, BCE validation loss and FPF. For the U deviations, we use increments of 0.0001 between 0.9995 and 1.0005 for the U weight. Similarly, we use increments of 0.004 between 0.98 and 1.02 for the a weight. For these model variations, the MSE loss and BCE loss calculated on the Dyck-1 Validation dataset, and the FPF on the Dyck-1 Very Long dataset are shown in Figure 8.2. We observe that the highest FPF value occurs where the correct model is located. However, within our test grid, the lowest MSE or BCE losses do not occur at the correct a/b ratio or U value. There are different values of a and b possible for any given a/b ratio value. If we vary the biases proportional to the value of b (or b) for a given a/b ratio the only change to the ReLU activation value is a multiplication by a constant factor, as is easy to show by induction. Thus the only change of the relative magnitudes of network outputs is caused by the sigmoid output activation (for MSE), which is a monotonic function. We have plotted additional heatmaps in Appendix C, illustrating the very minor changes of the loss for different values of b. Thus, minimising the MSE or BCE loss will likely not converge to a model that fulfils the CICs and counts correctly.

Table 8.2: Numbers of trained and converged models for Dyck-1 and semi-Dyck-1 experiments. The model names are as in Table 8.3.

	Dyck-1						Semi-Dyck-1					
	M/RI	**M/CI**	**M/CB**	**B/RI**	**B/CI**	**B/CB**	**M/RI**	**M/CI**	**M/CB**	**B/RI**	**B/CI**	**B/CB**
Trained	35	10	10	10	10	10	25	16	16	16	16	16
Conv	12	10	8	6	10	7	5	16	14	1	16	14

8.2.5 Training ReLUs to Count

We train models in different configurations: using MSE (M/) and BCE (B/) loss with Randomly Initialised (RI), and Correctly Initialised weights with and without Bias in the ReLU (CB and CI, respectively). We train models for 30 epochs, with the Adam optimiser (Kingma and Ba, 2014), using a learning rate of 0.01. We train models on the Dyck-1 dataset and on the semi-Dyck-1 dataset, and test each variant on both Dyck-1 and semi-Dyck-1 versions of the datasets.

We trained different models for different numbers of runs and in many runs the models did not converge to a loss value substantially below their initial state throughout the training. Therefore, we only include models that have converged in our results. The number of trained and converged models for each configuration is shown in Table 8.2. We select the models with the lowest validation loss for each run.

We report the accuracy for the converged models on all datasets and FPF values on the Very Long dataset in Table 8.3. All models show a large variation in the results. The largest differences can be observed between testing on the Dyck-1 and the semi-Dyck-1 datasets ((a) and (c) vs (b) and (d)), which was expected, given the difference in counter values discussed in section 8.2.1. The results differ also between the models trained on different datasets ((a) and (b) vs (c) and (d)), but unexpectedly the models trained on the less demanding semi-Dyck-1 dataset perform mostly better than the models trained on Dyck-1. However, fewer models converge when training on semi-Dyck-1 from random initialisation. Models trained with BCE loss perform broadly similar to the models trained with MSE loss.

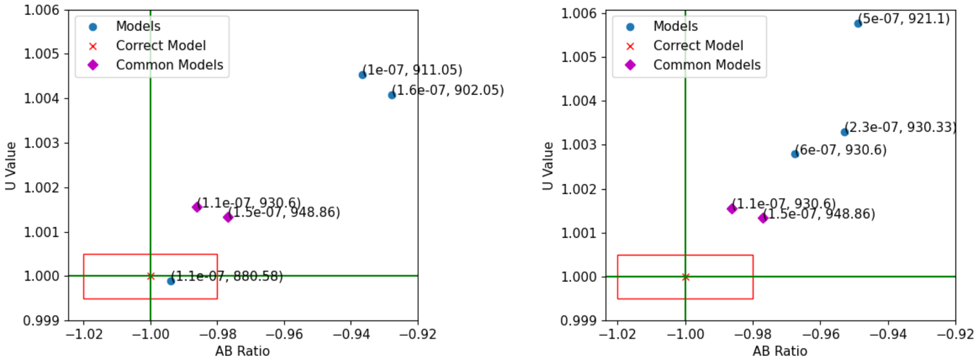
Models trained from correct initialisation (*/C*) do not retain the correct weight configuration and their performance after training is often worse than that of the correct models, especially with a trainable bias (*/CB). This is consistent with our observation

Table 8.3: Classification performance of various models trained and tested on Dyck-1 and semi-Dyck-1, and models with correct weights. Models are trained with MSE (M/) and BCE (B/) loss. Accuracy in percent and FPF values are given as mean (minimum/maximum) over runs after training models with Random Initialisation (RI), Correct Initialisation - without or with trainable ReLU bias (CI and CB). For B/RI trained on semi-Dyck-1, only one model converged, therefore there are no (min/max) values. An FPF value '-' indicates that the model did not fail on the Very Long dataset.

Mod	**Train**	**Validation**	**Long**	**Zigzag**	**V. Long**	**FPF**
		Models with correct weights, Testing: Dyck-1 (*), semi-Dyck-1 (**)				
(*)	100	100	100	100	100	-
(**)	100	100	100	100	100	-
		(a) Training: Dyck-1, Testing: Dyck-1				
M/RI	91.6 (48.7/100)	90.5 (43.1/100)	62.4 (3.74/100)	20.0 (0.0/40.0)	0.4 (0.0/4.0)	846.4 (528.2/979.3)
M/CI	95.2 (71.4/100)	94.4 (67.3/100)	76.1 (10.9/100)	25.0 (0.0/50.0)	1.2 (0.0/8.0)	905.3 (862.1/987.7)
M/CB	85.4 (39.4/100)	83.6 (33.2/100)	42.9 (0.9/98.1)	13.8 (0.0/30.0)	0.0 (0.0/0.0)	757.1 (432.0/911.9)
B/RI	97.7 (86.4/100)	97.3 (83.8/100)	83.9 (12.9/100)	23.3 (10.0/40.0)	8.3 (0.0/50.0)	848.4 (305.1/995.8)
B/CI	100 (100/100)	100 (100/100)	95.5 (60.3/100)	29.0 (10.0/60.0)	2.9 (0.0/29.0)	843.8 (553.0/992.4)
B/CB	91.9 (65.2/100)	90.4 (59.5/100)	72.2(3.9/100)	21.4 (10.0/40.0)	0.0 (0.0/0.0)	703.3 (257.4/959.9)
		(b) Training: Dyck-1, Testing: semi-Dyck-1				
M/RI	100 (100/100)	100 (99.9/100)	100 (99.6/100)	44.2 (20.0/90.0)	99.0 (88.0/100)	992.3 (907.6/-)
M/CI	100 (100/100)	100 (99.9/100)	100 (99.7/100)	53.0 (20.0/100)	98.0 (88.0/100)	984.9 (907.6/-)
M/CB	99.9 (99.4/100)	99.9 (99.2/100)	99.5 (97.0/100)	99.9 (99.2/100)	90.8 (56.0/100)	938.3 (723.2/-)
B/RI	100 (100/100)	100 (100/100)	100 (99.9/100)	66.7 (20.0/90.0)	99.3 (96.0/100)	995.1 (970.8/-)
B/CI	90.0 (0.0/100)	90.0 (0.0/100)	90.0 (0.0/100)	49.0 (0.0/100)	90.0 (0.0/100)	900.3 (2.6/-)
B/CB	100 (100/100)	100 (100/100)	99.8 (99.3/100)	48.6 (20.0/90.0)	96.0 (84.0/100)	974.4 (892.3/-)
		(c) Training: semi-Dyck-1, Testing: Dyck-1				
M/RI	99.4 (97.8/100)	99.2 (97.1/100)	73.3 (33.1/100)	22.0 (10.0/40.0)	3.2 (0.0/16.0)	724.7 (461.3/948.9)
M/CI	98.3 (87.5/100)	97.9 (85.6/100)	71.8 (25.8/100)	18.8 (10.0/70.0)	5.9 (0.0/94.0)	815.6 (436.2/999.5)
M/CB	100 (99.9/100)	99.9 (99.8/100)	90.3 (79.0/100)	25.0 (10.0/30.0)	6.8 (0.0/25.0)	893.3 (730.2/989.0)
B/RI	100	100	99.1	0.0	0.0	957.7
B/CI	90.8 (4.4/100)	90.2 (2.7/100)	68.9 (0.0/100)	16.4 (0.0/70.0)	7.1 (0.0/100)	788.2 (55.1/-)
B/CB	81.5 (0.0/100)	80.3 (0.0/100)	50.2 (0.0/100)	5.6 (0.0/30.0)	0.0 (0.0/0.0)	698.7 (0/985.0)
		(d) Training: semi-Dyck-1, Testing: semi-Dyck-1				
M/RI	100 (100/100)	100 (100/100)	100 (100/100)	46.0 (30.0/90.0)	100 (100/100)	- (-/-)
M/CI	100 (100/100)	100 (100/100)	100 (100/100)	42.5 (20.0/100)	100 (100/100)	- (-/-)
M/CB	100 (100/100)	100 (100/100)	100 (100/100)	50.0 (20.0/100)	99.7 (98.0/100)	995.8 (970.8/-)
B/RI	100	100	100	50.0	100	-
B/CI	100 (100/100)	100 (100/100)	100 (100/100)	68.1 (20.0/100)	99.7 (95.0/100)	996.0 (936.5/-)
B/CB	100 (100/100)	100 (100/100)	100 (100/100)	60.7 (30.0/100)	100 (100/100)	- (-/-)

that the correct weights are not at the minimum of the loss (Section 8.2.4) and with our findings in Chapter 5 (El-Naggar et al., 2022a) that training from correct weights with a sigmoid output activation function results in unlearning of correct weights. However, the models that are correctly initialised tend to converge better (see Table 8.2).

Overall, no trained models show perfect results in the tests, i.e. they do not learn the correct weights that satisfy the CICs. For the models trained on Dyck-1 strings, we show plots of the a/b and U values of the 5 M/RI models with the lowest validation loss and the 5 M/RI models with the highest FPF in Figure 8.3. More distribution plots can be found in Appendix D. The plots show that the all models deviate from the correct values, and the deviations of the a/b ratio and the U value tends to be positive.



(a) Average Validation Loss

(b) Average FPF

Figure 8.3: The best 5 M/RI models trained and tested on Dyck-1 selected (a) by average validation loss and (b) by FPF on the Very Long dataset. The red box represents the area corresponding to the heatmaps in Figure 8.2. The correct model position is at the intersection of the green lines. Two models present in both sets are marked with pink diamonds. For all models, the validation loss and FPF values are shown next to the markers.

8.3 Summary

We have formalised the counting mechanism of a ReLU RNN cell with the semi-Dyck-1 language and corresponding abstract counter machines that account for the absence of negative activation values in ReLUs. On this basis, we established three Counter Indicator Conditions (CICs) on the ReLU weights, which are necessary and sufficient for exhibiting correct counting behaviour. ReLU RNN cells that satisfy the CICs can count exactly, allowing for generalisation to strings of arbitrary length and arbitrarily great counter values (numeric representation permitting). We have empirically validated that a single-cell ReLU RNN that satisfies the CICs does indeed count correctly and accept the semi-Dyck-1 language, even on very long strings. However, our results also indicate that the mean squared error and the binary cross entropy as loss functions for training ReLU RNNs do not train the ReLU RNN to satisfy the CICs so that the trained models fail on very long strings.

Part III

Discrete Non-Negative Counter Module for RNNs

Chapter 9

Constructive Solution: Context

In Part I we find that standard RNNs do not learn exact counting with backpropagation precisely enough to generalise to arbitrarily long sequences. In Part II, we determine and prove Counter Indicator Conditions (CICs) on the weights of linear and ReLU RNNs and find that the models do not find the CICs in training. We also find that correctly initialised weights are unlearned during training and that the minimum of the loss function and CICs are not aligned. In this part, we ask the final overarching question stated in Chapter 1: can we design a discrete counter that can be integrated into RNNs and what is the effect of using it? We motivate the work presented in this part in section 9.1. Discrete elements cannot be readily integrated into NN models trained with backpropagation. Neural implementations of discrete structures have been proposed, and we discuss these implementations in section 9.2. We discuss alternative methods previously used to integrate discrete elements into NN models trained with backpropagation in section 9.3. The literature described in section 9.2 is directly related, and the other sections in this chapter cover background literature. In Chapter 10, we describe the design and implementation of the Discrete Non-Negative Counter (DNNC), which we equip with artificial gradients to allow for training with backpropagation. In Chapter 11 we integrate the DNNC in RNN models and use Dyck-1 acceptance tasks to evaluate the effect of the DNNC on model performance compared to the baseline in section 4.1.

9.1 Motivation

As stated previously, counting is a fundamental process that plays a role in numerous sequential tasks. As a result, there has been a renewed interest in integrating counting behaviour into RNN models in recent years.

In Part I, we empirically address counting behaviour in RNNs. The results in Part I show that RNNs do not learn counting behaviour with backpropagation precisely enough to generalise to arbitrarily long sequences, and therefore eventually fail. In Part II, we propose and prove two theorems where we define Counter Indicator Conditions (CICs) on the weights of single-cell linear and ReLU RNN that result in exact counting and subsequent generalisation to arbitrarily long sequences. When we train linear and ReLU RNNs with backpropagation, they do not find the CICs in training and even deviate from them due to a mismatch that we find between the loss functions and the CICs.

There have been previous attempts to integrate discrete behaviour in NN models. These include the neural implementations of discrete modules that we discuss in section 9.2, and alternative training methods, which we discuss in section 9.3. The results of these alternative solutions do not always guarantee that discrete behaviour is achieved.

Our results in Parts I and II show that counting behaviour is not learned by RNNs with backpropagation. The alternatives to integrating discrete behaviour that we discuss throughout this chapter do not guarantee the realisation of discrete behaviour. As a result, we focus here on developing a discrete solution to integrate exact discrete counting behaviour in RNNs while also allowing for training to occur with backpropagation.

9.2 Neural Implementations of Discrete Structures

Discrete modules are not differentiable and cannot be integrated into NN models as is when trained with backpropagation. A common method of integrating discrete behaviour into NN models is to implement fully differentiable neural modules that perform the function of discrete structures. The benefit of this approach is that these models can be trained with backpropagation, which is the most common training paradigm.

Neural adaptations of different discrete modules have been implemented with the aim of integrating discrete functionality into NNs. These include general purpose external memory like memory NNs (MemNNs) (Weston et al., 2014), Neural Network Pushdown Automata (NNPDA) (Sun et al., 1997, 2017), and neural stacks (Grefenstette et al., 2015; Joulin and Mikolov, 2015), and external memory that Neural Arithmetic Logic Units (NALUs) (Trask et al., 2018) and Differentiable Neural Computers (DNCs) (Graves et al., 2016).

NTMs (Graves et al., 2014), MemNNs (Weston et al., 2014), NNPDAs (Sun et al., 1997, 2017) and neural stacks (Grefenstette et al., 2015; Joulin and Mikolov, 2015) were all introduced to enhance NNs with external memory, with the aim of improving model generalisation. MemNNs (Weston et al., 2014) are NNs equipped with addressable memory that they can read from and write to. NTMs (Graves et al., 2014) are also designed to have NN controllers controlling large volumes of addressable memory, but are intended to behave like conventional Turing machines.

MemNNs (Weston et al., 2014) were introduced in order to increase and improve the long-term memory component of NN models. MemNNs are NN models equipped with memory that they can read from and write to. When tested on a question answering task, the MemNNs have outperformed their baseline models on the question answering task, but have not achieved perfect performance. They have also shown that the MemNNs that achieved the best performance have been trained with huge datasets. However, huge datasets are not always available, which restricts the use of the MemNNs.

Graves et al. (2014) introduce the Neural Turing Machine, which is inspired by the conventional Turing machine. The NTM is a fully differentiable implementation of a conventional discrete Turing machine. The NTM consists of a neural controller and a large amount of addressable memory. When tested, the NTM augmented models generalised more effectively to longer sequences than the standard LSTM baseline, but still did not generalise perfectly. Furthermore, in some instances, the controller has accessed the incorrect memory location.

Deleu and Dureau (2016) evaluate the ability of NTMs to learn the Dyck-1 language and generalise to longer sequences. They compare their results to an LSTM baseline and observe that the LSTM generalisation is limited, and the model does not generalise to arbitrarily long sequences. This is consistent with our results from Chapter 5. They report that NTM models generalise more effectively and for much longer than the standard LSTM baseline but still do not generalise to arbitrarily long sequences.

There have also been several implementations of fully differentiable neural stack models, where a neural controller is used to control a stack structure. These include the implementations by Sun et al. (1997), Joulin and Mikolov (2015) and Grefenstette et al. (2015), among others. Sun et al. (1997, 2017) develop the Neural Network Pushdown Automaton (NNPDA), where they combine an RNN controller with a continuous stack. The RNN controller is a higher order RNN which consists of different types of neurons corresponding to the input, current internal state, next internal state, and the stack action. The stack is equipped with continuous values representing the lengths of the elements in the stack. They use Real-Time Recurrent Learning (RTRL) (Williams and Zipser, 1989) to train the NNPDA. They then test their model on Dyck-1, 1^n0^n (which is a^nb^n with 1's and 0's instead), and palindromes. Their results show that a trained NNPDA generalises effectively to unseen data and to longer sequences. However, the second-order RNNs and RTRL training procedure are not standard approaches to NN model implementation and training.

Grefenstette et al. (2015) attempt to train a NN model with external memory to behave like different abstract data types (ADTs), specifically stacks, queues and dequeues. They connect the neural ADT to a neural controller and train both the neural memory structure to behave as an ADT, and the controller to learn to control the ADT. They find that the efficacy of the model learning varies based on the random initialisation. Furthermore, the values always remain stored in memory with varying degrees. This can result in a very large number of elements being stored in memory, which is inefficient.

Hao et al. (2018) evaluate the behaviour of RNN models augmented with the neural stack by Grefenstette et al. (2015) on a variety of simple computational tasks. They find that while it is possible for stack-augmented networks to learn to use the stack as a stack. Instead, they usually tend to not use it when other forms of memory (such as that of an LSTM) are available, or they the use stack as unstructured memory. They also find that the neural stacks are harder to train than standard RNNs.

Joulin and Mikolov (2015) equip RNNs with structured memory, specifically stacks and lists, and attempt to train the RNNs to control the memory structure. They then evaluate their models on a number of different algorithmic pattern learning tasks such as counting, binary addition and $a^n b^n$. Their results show that the performance of their RNN models improves when augmented with external memory. They find that the model generalisation to longer sequences improves when the controllers, which are continuous, are discretised at inference time.

Chen et al. (2020) develop a Neural-Symbolic Stack Machine (NeSS), where they combine a discrete stack with a NN controller. They evaluate their NeSS-augmented seq2seq models on the SCAN task and find that their models generalise perfectly to longer sequences on all splits of the SCAN task. However, they use an elaborate, non-standard training procedure which involves different training steps and is not straightforward. Their model is also tailor made for the SCAN task and complex and would therefore not be straightforward to interpret and adapt to different tasks.

Suzgun et al. (2019b) integrate neural stacks into Elman RNNs and LSTMs. They also extend the functionality of their neural stacks and develop a simplified version of the neural Turing machine, which they call the Baby-NTM. They then evaluate the performance of their stack-augmented and Baby-NTM-augmented models on Dyck-2, Dyck-3 and Dyck-6 and compare to their baseline models, which are standard Elman RNNs, LSTMs and the Stack-RNNs by Joulin and Mikolov (2015). Their results show that memory-augmented RNN models achieve more success at learning higher order Dyck languages than standard RNNs. However, there was a large variance in the performance of the models, where the minimum and maximum accuracies were not comparable.

9.3 Alternative Methods to Integrating Discrete Behaviour in RNNs

There have also been attempts to integrate discrete functions into NN models by using differentiable approximations at training, and using the actual discrete function during inference. These include the Estimate and Replace approaches used by Hadash et al.

(2018) and Jacovi et al. (2019). An Estimate and Replace approach allows for NN models to be trained with backpropagation, where part of the network is used to approximate a discrete function and then replaced with the exact function for inference.

Hadash et al. (2018) assign part of their model the role of a black-box function estimator, which is used as an interface for a discrete non-differentiable function into their NN models during training. The model is trained end-to-end, and at inference time, they replace their differentiable black-box estimator with the actual discrete function, and the NN model then interacts with the non-differentiable function. They find that their models require less training data when this Estimate and Replace approach is used, as opposed to their baseline which involves end-to-end model training without an estimator.

Jacovi et al. (2019) also use an estimate and replace approach. They use a neural estimator of a non-differentiable black-box function at training. During training, they use an additional loss functions to optimise their black-box estimator alongside the target loss that is traditionally used in end-to-end training. At inference, they replace their black-box estimator with the actual non-differentiable black-box function.

There has also been an interest in integrating symbolic logic into NNs, such as the work by Garcez et al. (2007), Tao et al. (2024), Manhaeve et al. (2018), and Bader (2009).

Garcez et al. (2007) use a binary counter to drive abducibles in an abductive logic program implemented using a neural network. The outputs of the neural network are used by a logic unit to increment the counter and progress through sets of abducibles until an abductive explanation is found for the hypotheses. (Manhaeve et al., 2018) introduce DeepProbLog which extends the probabilistic programming language ProbLog (De Raedt et al., 2007) to enable it to process neural predicates. Tao et al. (2024) theoretically examine neurosymbolic systems to determine the signals in a knowledge base that result in enhance or hinder learning, and minimising the inconsistency with the knowledge base. Bader (2009) studies the integration and effect of symbolic rules into NN models and propose a method to embed grammatical rules into a NN based part of speech tagger.

There has also been an interest in understanding the behaviour of Transformers on arithmetic tasks, such as the work by Shen et al. (2023). They examine the effect of position on model performance and how positional encodings can affect generalisation to longer sequences.

9.4 Summary

In this chapter, we review different approaches previously used to integrate counting and other discrete behaviour into RNN models. There have been several different neural implementations of discrete models. While these modules may allow for backpropagation to occur, they can be challenging to train and do not guarantee generalisation to arbitrarily long sequences. Other approaches include using alternative training approaches to backpropagation, and replacing the approximate differentiable modules used during training with discrete modules at inference. While the estimate and replace approach provides some improvement in generalisation over fully continuous models, the generalisation is still not perfect. Furthermore, backpropagation is considered the standard training regime and therefore it is important to find solutions to integrate counting behaviour into RNNs trained via backpropagation.

In this part, we propose a discrete non-negative counter module which we adapt for training with backpropagation. We design artificial gradients for the module in order to allow it to be integrated into NN models without impeding the backpropagation process.

Chapter 10

Discrete Non-Negative Counter Design

In this chapter, we design a Discrete Non-Negative Counter (DNNC) module with artificial gradients to integrate counting behaviour in NN models. We design the DNNC as a rudimentary stack with no memory, with the intention of extending this module to a fully functioning stack with memory in the future. We first describe the design of the DNNC and its operations for the forward pass. We then mathematically deduce the artificial gradients and detail the reasons for the gradients assigned. This module is then implemented in PyTorch.

10.1 Description of the Discrete Non-Negative Counter Module

We design a Discrete Non-Negative Counter (DNNC) module which emulates the behaviour of a traditional stack without storing any values. Discrete structures are non-differentiable, so we use artificial gradients are used to allow backpropagation to occur. The state of the DNNC is updated accordingly based on the inputs it receives. The DNNC is a recurrent structure, where the state is passed from one timestep to the next. This means it can be used to process bracket sequences, such as Dyck-1 sequences. The DNNC is shown in Figure 10.1.

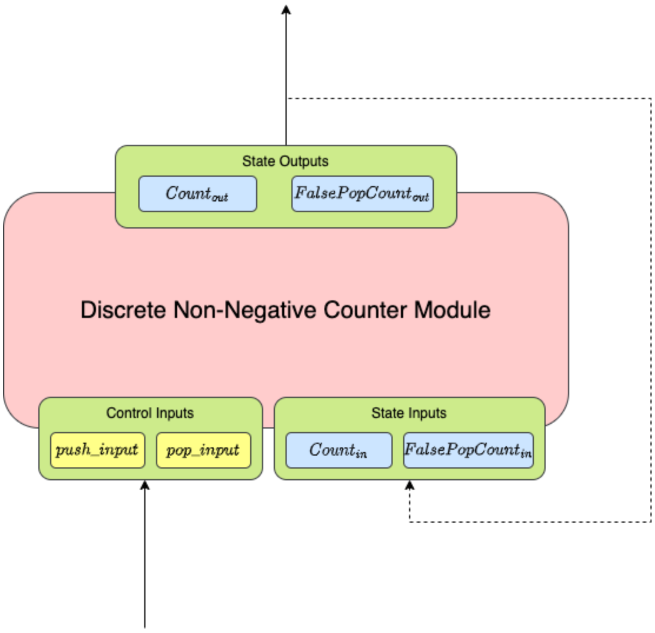


Figure 10.1: The basic design of the DNNC module, showing the inputs and outputs of the DNNC.

DNNC Control Inputs: The DNNC receives an input Tensor x of size 2. The first element of the tensor ($x(0)$) represents the push control signal, second element ($x(1)$) represents the pop control signal.

DNNC State: The DNNC State consists of the Count and false pop count. The Count is the number of elements present on the DNNC. The false pop count is the number of times a pop has been attempted when the Count was 0. The stack-counter state is updated based on the push and pop control signals and then returned as a single tensor containing 2 elements, where: $state(0) = Count$, and $state(1) = FalsePopCount$.

DNNC Outputs: The updated DNNC state is returned as a single tensor y and used as input to the following layer of a NN pipeline.

A custom activation function is used to update the DNNC state in the forward pass and assign the artificial gradients in the backward pass. The custom activation function of the stack-counter consists of a forward pass and a backward pass. In the forward pass, the DNNC state is updated and then returned. In the backward pass, the artificial gradients are applied based on the pre-op DNNC state.

10.2 The Forward Pass

The push and pop inputs that the DNNC receives (*push_input* and *pop_input*) are continuous values. They are represented in the DNNC by the discrete binary values *push* and *pop*. The values of *push* and *pop* are determined using a competitive input logic, where the continuous values *push_input* and *pop_input* are first compared to their respective thresholds ($threshold_{push}$ and $threshold_{pop}$), and to each other if needed. The values of *push* and *pop* are determined using the following competitive input logic. The forward pass consists of the following steps:

1. **The Competitive Input Logic:** The DNNC receives continuous inputs *push_input* and *pop_input*. They are represented using binary values *push* and *pop*. The competitive input logic determines the values of *push* and *pop* based on the values of the continuous inputs and how they compare to their their respective thresholds ($threshold_{push}$ and $threshold_{pop}$), and to each other if needed. The DNNC operation is determined based on the values of *push* and *pop*. This logic is defined in Equations 10.1 and 10.2.

2. **The Operational Logic:** The state of the DNNC is updated based on the operation that is determined by the input logic.

10.2.1 The Competitive Input Logic

The competitive input logic defined in Equations 10.1 and 10.2 is used to determine the operation that the DNNC will perform based on the continuous input values *push_input* and *pop_input*. Each input is compared to a dedicated threshold ($threshold_{push}$ and $threshold_{pop}$). If necessary, the input values are then compared to each other. The result of this competitive logic is reflected in the discrete binary values *push* and *pop*, which are then used to decide the DNNC operation. The resulting DNNC operations are shown in Equation 10.3.

$$push = \begin{cases} 0, & \text{if } \begin{cases} push\_input < threshold_{push} \\ push\_input \geq threshold_{push} \text{ and } push\_input < pop\_input \end{cases} \\ 1, & \text{if } push\_input \geq threshold_{push} \text{ and } push\_input \geq pop\_input \end{cases} \tag{10.1}$$

$$pop = \begin{cases} 0, & \text{if } \begin{cases} pop\_input < threshold_{pop} \\ pop\_input \geq threshold_{pop} \text{ and } pop\_input < push\_input \end{cases} \\ 1, & \text{if } pop\_input \geq threshold_{pop} \text{ and } pop\_input > push\_input \end{cases} \tag{10.2}$$

As a result of the binarisation, the possible operations are:

$$Operation = \begin{cases} \textbf{NoOp}, & \text{if } push = 0 \text{ and } pop = 0 \\ \textbf{Push}, & \text{if } push = 1 \text{ and } pop = 0 \\ \textbf{Pop}, & \text{if } push = 0 \text{ and } pop = 1 \end{cases} \tag{10.3}$$

	$push\_input \geq pop\_input$	$push\_input < pop\_input$	$push\_input \geq pop\_input$	$push\_input < pop\_input$
	$pop\_input < threshold_{pop}$		$pop\_input \geq threshold_{pop}$	
$push\_input < threshold_{push}$	$push = 0$ $pop = 0$	$push = 0$ $pop = 0$	$push = 0$ $pop = 1$	$push = 0$ $pop = 1$
$push\_input \geq threshold_{push}$	$push = 1$ $pop = 0$	$push = 1$ $pop = 0$	$push = 1$ $pop = 0$	$push = 0$ $pop = 1$

🟥 **Pop**
🟨 **NoOp**
🟩 **Push**

Table 10.1: Summary of input logic

10.2.2 The Operational Logic

Based on the values of *push* and *pop*, the DNNC state is updated. The following decision operational logic will be used to apply the operations above and make any required changes to the DNNC state.

The DNNC state consists of the following values:

- *Count*: A positive integer value which represents the number of elements on the

DNNC. The value of $Count \geq 0$.

- $FalsePopCount$: A positive integer value which represents the number of times a **Pop** operation has been attempted on an empty DNNC, i.e., when $Count = 0$. The value of $FalsePopCount \geq 0$.

The operational logic that updates the values of $Count$ and $FalsePopCount$ is described in Equations 10.4 and 10.5. The pre-operational DNNC state consists $Count_{in}$ and $FalsePopCount_{in}$, while the post-operational DNNC state consists of $Count_{out}$ and $FalsePopCount_{out}$.

$$Count_{out} = \begin{cases} Count_{in}, & \text{if } Operation = \textbf{NoOp} \\ Count_{in} + 1, & \text{if } Operation = \textbf{Push} \\ Count_{in} - 1, & \text{if } Operation = \textbf{Pop} \text{ and } Count_{in} > 0 \\ 0, & \text{if } Operation = \textbf{Pop} \text{ and } Count_{in} = 0 \end{cases} \tag{10.4}$$

$$FalsePopCount_{out} = \begin{cases} FalsePopCount_{in}, & \text{if } \begin{cases} Operation = \textbf{Push} \\ Operation = \textbf{NoOp} \\ Operation = \textbf{Pop} \text{ and } Count_{in} > 0 \end{cases} \\ FalsePopCount_{in} + 1, & \text{if } Operation = \textbf{Pop} \text{ and } Count_{in} = 0 \end{cases} \tag{10.5}$$

After the DNNC is updated, it is returned as output in a single tensor to the following layer of the model architecture. The operational logic is summarised in Table 10.2.

	$Count_{in} > 0$	$Count_{in} = 0$
Push	$Count_{in} + 1$ $FalsePopCount_{in}$	$Count_{in} + 1$ $FalsePopCount_{in}$
Pop	$Count_{in} - 1$ $FalsePopCount_{in}$	$Count_{in}$ $FalsePopCount_{in} + 1$
NoOp	$Count_{in}$ $FalsePopCount_{in}$	$Count_{in}$ $FalsePopCount_{in}$

Table 10.2: Summary of operational logic, where the updates made to the DNNC state are decided based on the pre-operational DNNC state and the input operation.

10.3 The Backward Pass

The backward pass is the stage at which the gradients are calculated and propagated to the previous layer. These gradients are derivatives of the output with respect to the input. To clarify, a derivative indicates the effect on the output $f(x)$ if an infinitesimally small change ϵ is made in the input x. Mathematically this if formally defined as:

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{10.6}$$

This discrete DNNC does not perform differentiable operations. In order to calculate gradients we do the following:

1. Assume differentiability, and assign artificial gradients based on the direction we want the values to be pulled towards. We assume that since they are linear, a connection can be made between valid points, and a decision is made when a non-differentiable point is encountered.

2. When looking at the derivatives of 2 points, $x + \epsilon$ and $x - \epsilon$, if the gradients are different, then we make a decision when we have a lack of differentiability based on the logic and the DNNC. If a point x is naturally differentiable, then the gradient at points $x+\epsilon$ and $x-\epsilon$ are the same, otherwise, that point is non-differentiable. In this case, we decide which gradient value to assign based on the desired behaviour of the DNNC.

The backward pass examines the pre-operation DNNC state ($Count_{in}$ and $FalsePopCount_{in}$), and assigns the according artificial gradients. It then applies the chain rule to the gradient from the output side using the artificial gradients and returns it to the previous layer so that backpropagation can continue. The following artificial gradients are created:

10.3.1 The Input Artificial Gradients

The artificial gradients are defined in Equations 10.7, 10.8, 10.9 and 10.10.

$$\frac{\partial Count_{out}}{\partial push} := 1 \tag{10.7}$$

$$\frac{\partial FalsePopCount_{out}}{\partial push} := 0 \tag{10.8}$$

$$\frac{\partial Count_{out}}{\partial pop} := \begin{cases} -1, & \text{if } Count_{in} > 0 \\ 0, & \text{if } Count_{in} = 0 \end{cases} \tag{10.9}$$

$$\frac{\partial FalsePopCount_{out}}{\partial pop} := \begin{cases} 0, & \text{if } Count_{in} > 0 \\ 1, & \text{if } Count_{in} = 0 \end{cases} \tag{10.10}$$

- In Equation 10.7, the value of $\frac{\partial Count_{out}}{\partial push}$ is set to 1. This is because a **Push** will always result in an increase in the value of $Count_{in}$, and hence a positive gradient is required.

- The value of $\frac{\partial FalsePopCount_{out}}{\partial push}$ in Equation 10.8 is set to 0 because a **Push** will have no effect on the value of $FalsePopCount_{in}$ This is why it is assigned a value of 0.

- In Equation 10.9, the value of $\frac{\partial Count_{out}}{\partial pop}$ is decided based on the pre-operational Count ($Count_{in}$).
  - If the DNNC is not empty ($Count_{in} > 0$), then a **Pop** will cause a decrease in $Count_{in}$. Therefore, a negative gradient is assigned.
  - If the DNNC is empty ($Count_{in} = 0$), then no change can happen to $Count_{in}$ in the event of a **Pop**. As a result, a 0 gradient is assigned.

- In Equation 10.10, the value of $\frac{\partial FalsePopCount_{out}}{\partial pop}$ also depends on the pre-operational Count, $Count_{in}$.
  - If the DNNC is not empty ($Count_{in} > 0$), no change will happen to $FalsePopCount_{in}$, in the event of a **Pop**. Therefore, a 0 gradient is assigned.
  - If the DNNC is empty ($Count_{in} = 0$), then $FalsePopCount_{in}$ will increase in the event of a **Pop**. Hence, a positive gradient is necessary.

After the artificial gradients are determined, the chain rule is applied to the gradients from the output side $grad\_output$, where $grad\_output[0] = \frac{\partial Loss}{\partial Count_{out}}$ and $grad\_output[1] = \frac{\partial Loss}{\partial FalsePopCount_{out}}$, to calculate the input gradients for the previous layer $grad\_input = \frac{\partial Loss}{\partial x}$.

$$\frac{\partial Loss}{\partial push} = \left[\frac{\partial Count_{out}}{\partial push} \times \frac{\partial Loss}{\partial Count_{out}}\right] + \left[\frac{\partial FalsePopCount_{out}}{\partial push} \times \frac{\partial Loss}{\partial FalsePopCount_{out}}\right] \tag{10.11}$$

$$\frac{\partial Loss}{\partial pop} = \left[\frac{\partial Count_{out}}{\partial pop} \times \frac{\partial Loss}{\partial Count_{out}}\right] + \left[\frac{\partial FalsePopCount_{out}}{\partial pop} \times \frac{\partial Loss}{\partial FalsePopCount_{out}}\right] \tag{10.12}$$

$$\frac{\partial Loss}{\partial x} = \left[\frac{\partial Loss}{\partial push}, \frac{\partial Loss}{\partial pop}\right] \text{ where } x = [push\_input, pop\_input] \tag{10.13}$$

When the chain rule is applied using the artificial gradients assigned in the backward pass, the following deductions can be made:

$$\frac{\partial Loss}{\partial push} = \frac{\partial Loss}{\partial Count_{out}} \tag{10.14}$$

$$\frac{\partial Loss}{\partial pop} = \begin{cases} -\frac{\partial Loss}{\partial Count_{out}}, & \text{if } Count_{in} > 0 \\ \frac{\partial Loss}{\partial FalsePopCount_{out}}, & \text{if } Count_{in} = 0 \end{cases} \tag{10.15}$$

$$\text{As a result: } \frac{\partial Loss}{\partial x} = \begin{cases} \left[\frac{\partial Loss}{\partial Count_{out}}, -\frac{\partial Loss}{\partial Count_{out}}\right], & \text{if } Count_{in} > 0 \\ \left[\frac{\partial Loss}{\partial Count_{out}}, \frac{\partial Loss}{\partial FalsePopCount_{out}}\right], & \text{if } Count_{in} = 0 \end{cases} \tag{10.16}$$

10.3.2 The State Artificial Gradients

In order for backpropagation through time to occur in a recurrent architecture, gradients between consecutive stack-counter states are calculated. Similarly to the input gradients, the state gradients are also artificial gradients, and are derived based on our logic. There are four state gradients:

- $\frac{\partial Count_{out}}{\partial Count_{in}}$: The derivative of the output Count wrt the input Count.
- $\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}}$: The derivative of the output false pop count wrt the input false pop count.
- $\frac{\partial Count_{out}}{\partial FalsePopCount_{in}}$: The derivative of the output Count wrt the input false pop count.
- $\frac{\partial FalsePopCount_{out}}{\partial Count_{in}}$: The derivative of the output false pop count wrt the input Count.

These gradients are each derived and then outlined in Tables 10.3, 10.4, 10.5 and 10.6, and summarised in Table 10.7. These gradients are also artificial gradients, which are integer values, similar to the input gradients.

Calculating Values of $\frac{\partial Count_{out}}{\partial Count_{in}}$

In order to assign the most appropriate gradients, we consider four cases:

1. **NoOp:** When a **NoOp** operation is executed, any change in the input Count ($Count_{in}$) will be reflected in the output Count ($Count_{out}$).

   - When $Count_{in} > 0$, the gradient will always be 1.
   - When $Count_{in} = 0$ and the target $Count \geq 0$ the gradient will be 1.
   - **Edge Case:** When $Count_{in} = 0$, and the target $Count$ is negative, this would result in a 0 gradient. This is an extremely rare (almost impossible) occurrence, and hence it is ignored. The possibility for an upward correction of the $Count$ is more important to focus on than the uncommon case where the target $Count$ is negative.

- $Count_{in} = 0$ is a turning point. We refer to equation 10.6 where a derivative is defined as $\lim_{\epsilon \to 0} \frac{f(x+\epsilon)-f(x)}{\epsilon}$. At $Count_{in} = 0$, inducing and infinitesimally small change ϵ in the positive direction will have a disconnect from the point where we have a change of ϵ in the negative direction. In short, the point $Count_{in} = 0$ is a non-differentiable point, where $x + \epsilon > 0$ and $x - \epsilon < 0$. This creates a non-differentiability, and a choice has to be made to prioritise one direction over the other. We prioritise the more common and important situation where a change in the positive direction $(x + \epsilon)$ is accounted for, and a change in the negative direction $(x - \epsilon)$ is ignored.

As a result of these assumptions, the value of $\frac{\partial Count_{out}}{\partial Count_{in}} = 1$ when a **NoOp** operation is executed.

2. **Push:** When a **Push** operation is executed, any change in $Count_{in}$ will be reflected in $Count_{out}$. Because the **Push** operation increments the Count, the value of $\frac{\partial Count_{out}}{\partial Count_{in}}$ will always be 1.

3. **Pop** from a non-empty DNNC:

   - $Count_{in} > 1$: Any change in the input Count will be reflected in the output Count. Therefore, a gradient value of 1 is assigned.
   - **Edge Case** $Count_{in} = 1$: We are faced with an edge case where if a **Pop** occurs with an input Count of 1. This is the switching point between an empty and non-empty DNNC, which makes it a point of of non-differentiability due to the resulting Count being 0, and a potential subsequent **Pop** having an input Count value of 0. If the target Count is greater than the actual output Count (i.e. greater than 0), then a positive gradient (1) is favourable to encourage an increase in the value of the Count. However, if the target Count is below 0 then this is a very rare case that ideally should not happen. It is not a meaningful situation in comparison to the more common and ideal situation where the target Count is 0 or greater. As a result, this particular situation can be ignored. The natural gradient below 1 $(1 - \epsilon)$ is 0, and the gradient above, $(1 + \epsilon)$ is equal to 1. At 1, there is a non-differentiability, and

a decision has to be made. In this case, we choose to assume that the value of the gradient will be equal to 1.

4. **Pop** from an empty DNNC: If a **Pop** operation occurs where the input Count is 0, then no change will occur and consequently, no change will be reflected in the output Count, resulting in a 0 gradient.

Case	Operation	$Count_{in}$	$\frac{\partial Count_{out}}{\partial Count_{in}}$
1	**NoOp**	Any	1
2	**Push**	Any	1
3	**Pop**	> 0	1
4	**Pop**	0	0

Table 10.3: The derivatives of the output Count wrt the input Count for each of the four cases.

Table 10.3 summarises the gradient values for each of the four mentioned cases.

Calculating Values of $\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}}$

The value of $FalsePopCount_{out} \geq FalsePopCount_{in}$, and any change to $FalsePopCount_{in}$ is always reflected in $FalsePopCount_{out}$. Even if a false pop operation does not occur at a particular time step, it is important to maintain the value of the false pop count and propagate it to previous time steps in order to indicate the number of false pops (if any) that occurred at any time step. As a result, the value of the gradient $\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}}$ will always be 1, regardless of the input stack-counter state.

Edge case: If the $FalsePopCount_{in}$ is 0 and the target $FalsePopCount = FalsePopCount_{in} - \epsilon$ is below 0, which should not happen, the resulting gradient in this case should be 0. However, since this is an extremely unusual case, which should not happen, it is ignored. If the target $FalsePopCount = FalsePopCount_{in} + \epsilon > 0$, the gradient will be positive. Conversely, if $FalsePopCount_{in} = 0$, the value of $FalsePopCount_{in} - \epsilon < 0$. This is a non-differentiable point, and we make a decision to prioritise the case where $FalsePopCount_{in} + \epsilon > 0$ and assign a gradient value of 1. The more common and important case of maintaining the $FalsePopCount$ and propagating this error signal to the previous time steps is prioritised and the edge case is ignored.

As shown in Table 10.4, the value of the gradient $\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}} = 1$.

Case	$FalsePopCount_{in}$	$\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}}$
1	0	1
2	> 0	1

Table 10.4: The state gradients of the output false pop count wrt the input false pop count

Calculating Values of $\frac{\partial Count_{out}}{\partial FalsePopCount_{in}}$

The value of the input false pop count has no effect on the output Count under all circumstances. Any changes to the value of $FalsePopCount_{in}$ do not affect the value of $Count_{out}$. Therefore, the value of this gradient is always going to be 0. This is an unconditional situation and $FalsePopCount_{in} + \epsilon = FalsePopCount_{in} - \epsilon = 0$ relative to $Count_{out}$.

Case	Operation	$Count_{in}$	$FalsePopCount_{in}$	$\frac{\partial Count_{out}}{\partial FalsePopCount_{in}}$
1	**NoOp**	Any	Any	0
2	**Push**	Any	Any	0
3	**Pop**	> 0	Any	0
4	**Pop**	0	Any	0

Table 10.5: This table outlines the possible values of the gradient of the output Count wrt the input false pop count

As shown in table 10.5, the gradient will always be 0.

Calculating Values of $\frac{\partial FalsePopCount_{out}}{\partial Count_{in}}$

In most cases, the value of $Count_{in}$ has no effect on the value of $FalsePopCount_{out}$. The only instance where the value of the input Count affects the output false pop count is when that input Count is 0 and a **Pop** operation is triggered. This is because when the input Count is 0, then the value of the output false pop count rises. Conversely, when the value of the input Count is higher, then the value of the false pop count is unchanged, it is not incremented like the input Count. In this case, there is also a discontinuity, where $FalsePopCount_{in} + \epsilon$ has a gradient of 0, and $FalsePopCount_{in} - \epsilon$ has a gradient of -1. We therefore find it sensible to assign this case a value of -1, because hypothetically, the false pop count increases as the Count decreases in the case of a pop from an empty DNNC. Since the Count cannot go below 0 and the false pop count is incremented, we follow the previous hypothetical statement and assign a gradient of -1. This is all outlined in table 10.6.

Case	Operation	$Count_{in}$	$\frac{\partial FalsePopCount_{out}}{\partial Count_{in}}$
1	**NoOp**	Any	0
2	**Push**	Any	0
3	**Pop**	> 0	0
4	**Pop**	0	-1

Table 10.6: The derivatives of the output False Pop count wrt the input Count

All the gradients deduced tables 10.3, 10.4, 10.5, and 10.6 are summarised in table 10.7.

Case	Operation	$Count_{in}$	$\frac{\partial Count_{out}}{\partial Count_{in}}$	$\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}}$	$\frac{\partial Count_{out}}{\partial FalsePopCount_{in}}$	$\frac{\partial FalsePopCount_{out}}{\partial Count_{in}}$
1	**NoOp**	Any	1	1	0	0
2	**Push**	Any	1	1	0	0
3	**Pop**	> 0	1	1	0	0
4	**Pop**	0	0	1	0	-1

Table 10.7: A summary of the state gradients

$$\frac{\partial Loss}{\partial Count_{in}} = \left[\frac{\partial Count_{out}}{\partial Count_{in}} \times \frac{\partial Loss}{\partial Count_{out}}\right] + \left[\frac{\partial FalsePopCount_{out}}{\partial Count_{in}} \times \frac{\partial Loss}{\partial FalsePop_{out}}\right] \tag{10.17}$$

$$\frac{\partial Loss}{\partial FalsePopCount_{in}} = \left[\frac{\partial FalsePopCount_{out}}{\partial FalsePopCount_{in}} \times \frac{\partial Loss}{\partial FalsePopCount_{out}}\right] + \left[\frac{\partial Count_{in}}{\partial FalsePopCount_{in}} \times \frac{\partial Loss}{\partial Count_{out}}\right] \tag{10.18}$$

After the four artificial state gradients are deduced, backpropagation through time can occur using the calculations in Equations 10.17 and 10.18.

We implement the DNNC module in PyTorch, and debug the module. We then design, implement, and execute unit tests in Python, which are outlined in Appendix F.

10.4 Summary

In this section we design a Discrete Non-Negative Counter (DNNC) module that can be used with NN models that learn via backpropagation. Through mathematical reasoning, we deduce the appropriate artificial gradients to allow for compatibility with backpropagation. The module is then implemented in PyTorch.

Chapter 11

Experiments Integrating the Discrete Non-Negative Counter Module into Neural Network Models

In Chapter 10, we design a Discrete Non-Negative Counter (DNNC) module, which we equip with artificial gradients to allow for learning to occur via backpropagation. The objective of this chapter is to determine the effect of our DNNC module on the performance of NN models. We compare the results of our models to the results in Chapter 4. The results of our experiments in Chapters 4, 7 and 8 have shown that RNNs configured to count exactly are subject to unlearning the correct weights and biases when tradtional training setups are used. This inspired us to design the discrete DNNC with artificial gradients, which is not trainable but still allows for backpropagation to happen.

In this chapter, we detail the experiments where we integrate the DNNC into NN models. The objectives of these experiments is to evaluate the ability of NN models to learn from data to operate this DNNC, and to analyse the effect of this DNNC on the model's ability to extrapolate and generalise to longer sequences. We gradually increase the difficulty of the learning task to determine the extent to which our models can learn to use our DNNC, and be able to identify any possible shortcomings with our DNNC and models.

After ensuring that the DNNC is working as intended, we conduct a number of experiments using the DNNC. These experiments are used to evaluate the effect of the DNNC on NN models, whether or not this DNNC can be used effectively when training the models using backpropagation, and whether or not the integration of this DNNC can improve the systematic generalisation of a model. A number of experiments are performed using a number of different models.

1. Integrating the DNNC into a Feed-Forward Network.
2. Dyck-1 Acceptance using the DNNC as a Hidden Layer in a Feed-Forward Network.
3. Dyck-1 Acceptance using the DNNC as a Recurrent Hidden Layer.
4. Dyck-1 Acceptance using DNNC-Augmented Recurrent Networks (Mixed Models).

The models are tested under a number of conditions and configurations. For each of the models, sigmoid and clipping output activation functions (see Definitions 2.1 and 2.5) are tested. We vary the degrees of freedom and information supplied to our models during backpropagation to push the learning boundaries and determine the effect of these changes on the models' performance. Different weight and bias initialisations are tested for the models which use the stack-counter as a hidden layer. We also test the effect of freezing the input or output layer on the performance and generalisation. More details of our experiment results can be found in Appendix G.

11.1 Experiment 1: Integrating the DNNC into a Feed-Forward Network

This experiment is the first experiment where we integrate our DNNC into a NN model. We start with a very simple model in this experiment. The objective of this experiment is to evaluate the ability of this simple NN model to learn to control this DNNC. The DNNC is tested in a feed-forward network to see if the network will learn weights when trained with the DNNC. The network is trained and tested and the results show that the network is capable of learning to use the DNNC, and correctly using the DNNC during

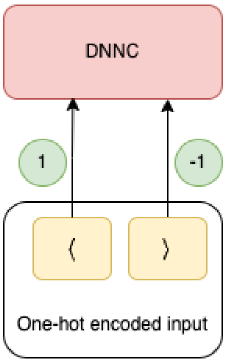


Figure 11.1: Architecture of the model used where the DNNC is integrated into a feed-forward network. This setup is used to evaluate the learning of the correct weights in a single timestep in order to produce the final DNNC state.

testing. We start with this simple task to investigate whether a randomly initialised fully connected layer will be able to learn how to control the DNNC. It is a relatively simple learning task, and a lot of information is provided to the network in the backward pass.

A feed-forward neural network with random weights is used for this experiment. For simplicity, and ease of tracing, the neural network pipeline used is very simple. The pipeline consists of:

1. A fully connected input layer with random weights and bias. This input layer consists of 2 neurons.
2. The DNNC.

The output of the model is the final DNNC state. The model used is shown in Figure 11.1.

11.1.1 Experimental Setup

In this experiment, we use Dataset 5, the Feed-Forward DNNC State Update Dataset.

Dataset 5. (Feed-Forward DNNC State Update Dataset) A balanced dataset of binary vectors representing **Push**, **Pop**, and **NoOp** operations is created for network inputs. Since the gradients are dependent on the DNNC state, specifically the Count, the dataset consists of 6 different possible cases:

1. **Push**

   (a) to a non-empty DNNC.

   (b) to an empty DNNC.

2. **Pop**

   (a) from a non-empty DNNC.

   (b) from an emtpy DNNC.

3. **NoOp**

   (a) on a non-empty DNNC.

   (b) on an empty DNNC.

The outputs in this dataset are the new DNNC state after the **Push**, **Pop**, or **NoOp** operations have been applied to the initial DNNC state. An empty DNNC is one where the initial count is 0, ($Count_{in} = 0$), and a non-empty DNNC has an initial count that is greater than 0 ($Count_{in} > 0$). The aim is to determine whether the correct weights can be learn to control the DNNC in a single timestep, as a first step to The training dataset consists of 6000 instances. The target values are also generated based on the binary vectors and the generated DNNC state. The test dataset consists of 2000 elements. The same 6 cases from training are included in the test set.

The DNNC state is initialised using the DNNC states generated in the dataset. The input binary vectors are then input to the model to trigger one of the DNNC operations **Push**, **Pop** or **NoOp**. The main research question that this experiment aims to answer is whether a simple NN can be trained to use the DNNC correctly. More specifically, can the network learn the correct weights and biases required to trigger the correct DNNC control signals? We run this experiment 10 times to fully evaluate the model. 2 different learning rates are tested.

The model is trained over 5 epochs, and the entire training set is used to train the model during each epoch. We use an Adam optimiser once with a learning rate of 0.001 and another time with a learning rate of 0.0003, and an MSE loss function. Feedback is given to the model after each element of the training data is passed through the model, which is at every time step. The training accuracy converges to 100% accuracy after the third or fourth epoch. For all runs and learning rates, the model consistently converges to 100% training accuracy.

The test data set consists of 2000 elements. The model achieves 100% accuracy on the test set. For all runs and learning rates tested, the model consistently achieves 100% accuracy on the test set.

11.1.2 Results

The results of training and testing the DNNC with a preceding input layer in a FF setting show that NNs are capable of learning and correctly using the DNNC. For training, the accuracy converges to 100% after 3 or 4 epochs. When tested, the model achieves 100% accuracy as well. This shows that a feed-forward neural network is capable of learning the correct gradients related to the DNNC, and also applying these gradients at inference time. For this experiment, and the model we used here, we can say that the model can be trained to correctly use the DNNC.

11.2 Experiment 2: Dyck-1 Acceptance using the DNNC in a Feed-Forward Network

In our previous experiment, we tested the ability of a very simple NN model to learn to use our DNNC module. For this experiment, we expand the model from the previous experiment and use a Dyck-1 acceptance task. The objective of this experiment is to gradually increase the difficulty of the learning task and push the boundary of the NN model to determine the extent to which the model can learn to successfully control the DNNC and systematically accept the Dyck-1 language. After determining in the previous experiment that a NN can learn to correctly use the DNNC, we increase the difficulty of the learning task slightly. Here, we emulate the last time step of a Dyck-1 sequence and perform a binary classification task, where a sequence is classified as valid or invalid in the Dyck-1 language.

The model is made up of a fully connected input layer consisting of 2 neurons,

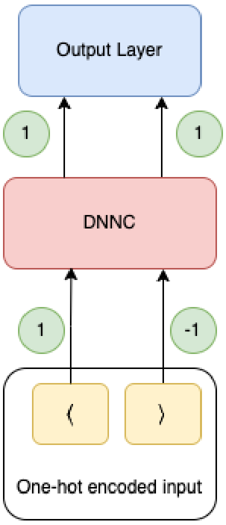


Figure 11.2: Architecture of the model used in the Dyck-1 acceptance task where the DNNC is used as a hidden layer in a feed-forward network. Examples of the correct weights are shown in the green circles.

followed by the DNNC, and finally a fully connected output layer consisting of one neuron. The input and output layers do not have biases. The model used is shown in Figure 11.2.

11.2.1 Experimental Setup

After determining in the previous experiment that a very simple model can learn to correctly control the DNNC, we increase the complexity of the model to push the boundary and make the learning task more difficult. We use two different output activation functions for this model.

1. **Sigmoid:** Here a sigmoid activation is used for the output layer.
2. **Clipping:** Here a clipping activation is used for the output layer. This bounds the output between 0 and 1.

Both configurations are trained under a variety of different conditions:

1. Correct/Random weight initialisation with no fixed weights.
2. Correct/Random initialisation with fixed input layer weights.

3. Correct/Random initialisation with fixed output layer weights.

Examples of the correct weights are shown in Figure 11.2. This experiment is designed to determine whether or not a NN model can be trained to use the DNNC as a hidden layer. Furthermore, because we observed some unlearning in our preliminary experiments, we investigate the unlearning (or lack thereof) that occurs as a result of using the DNNC in this setup. Additionally, we want to push the learning boundary and determine the extent to which NN models can learn to use the DNNC. As well as finding the breaking point of a model's learning, we want to find the cause, or the layer that causes the model to fail. Gradually making a learning task more difficult by increasing or decreasing the feedback information or degrees of freedom in training can help us achieve these goals. For each combination of configuration and condition, we run our experiment 10 times.

In this experiment we use Dataset 6, the Feed-Forward Dyck-1 Acceptance Dataset.

Dataset 6. (Feed-Forward DNNC Dyck-1 Acceptance Dataset) This dataset is a simulation of the Dyck-1 sequences of length 2. We have an initial DNNC state, which is made up of 2 values, one for the Count, and one for the false pop count. We also have a single bracket which we can assume is like the final bracket of a sequence. The input bracket is represented using a one-hot encoding, and the DNNC state is initialised before the input is passed to the network. The output labels are encoded as 0 for the valid class, and 1 for the invalid class. The values of the initial DNNC state values are either 0 or 1. The dataset is shown in Table 11.1. Because the dataset is not balanced, the minority class is oversampled. The resulting dataset contains 6 elements. Because this dataset represents a single timestep in a feed-forward network, the size of the dataset is very small. This can also provide some insight into whether the models can learn to control the DNNC from minimal data. The training set contains 4 of the elements in the dataset, and the test set contains the 2 remaining elements in the dataset. Both train and test sets are balanced.

We train our models over 10,000 epochs. An Adam optimiser is used with a learning rate of 0.001. The training dataset consists of 4 single bracket one-hot encodings and their corresponding initial DNNC states. The dataset is balanced, where 2 elements

Table 11.1: FF DNNC Dyck-1 Acceptance Dataset

Input Bracket	Initial Count	Initial False Pop Count	Class Label
$\langle$	1	0	Invalid
$\langle$	0	1	Invalid
$\rangle$	1	0	Valid
$\rangle$	0	1	Invalid

belong to the valid class and the other 2 belong to the invalid class. The DNNC state is initialised before passing the bracket through the model. Backpropagation is performed after every sequence passed through the model. The models we use have no biases, and as a result fewer degrees of freedom, but an additional classification layer. We sometimes freeze the weights in the input or output layer to try to get a better understanding of the model and how it learns. The model is able to achieve 100% training accuracy using both clipping and sigmoid activation functions. However, this is not consistently the case. Also, training from correct weights with the DNNC did not result in any unlearning with both the clipping and sigmoid activation functions.

The remainder of the dataset is used to test the model after training. The test dataset is balanced and consists of one valid element and one invalid element. The model is capable of achieving 100% accuracy using all variations of the model with all initialisations, however this is not consistent.

11.2.2 Results

The results of our experiments are shown in Table 11.2. The results show that no unlearning occurred when the model is trained from correct weights using both the clipping activation function and the sigmoid activation function. The model appears to have no trouble learning the output layer when a sigmoid activation is used, However, it seems to have more difficulty learning the correct weights for the input layer when a sigmoid activation is used for the output layer. When a clipping activation is used, the model seems to struggle more with learning the output layer, not the input layer, and freezing the input layer has a negative effect on the train and test accuracies. However, with a clipping activation and no frozen weights, the model seems more capable of learning the correct weights from the data than its sigmoid counterpart. When training from random weights, the use of a clipping configuration results in better performance than the sigmoid configuration on both train and test sets.

Table 11.2: DNNC as a hidden layer in a FF network Dyck-1 Acceptance Results using Adam optimiser and no biases

Output	**Weight**	**Fixed**		**Train**			**Test**		**Std**	
Activation	**Initialisation**	**Weights**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Train**	**Test**
Sigmoid	Random	None	75	100	80	50	100	60	10	20
Sigmoid	Random	Input	100	100	100	100	100	100	0	0
Sigmoid	Random	Output	50	100	60	50	100	60	20	20
Sigmoid	Correct	None	100	100	100	100	100	100	0	0
Sigmoid	Correct	Input	100	100	100	100	100	100	0	0
Sigmoid	Correct	Output	100	100	100	100	100	100	0	0
Clipping	Random	None	50	100	87.5	50	100	85	20.16	22.91
Clipping	Random	Input	50	100	75	50	100	70	25	24.49
Clipping	Random	Output	50	100	90	50	100	90	20	20
Clipping	Correct	None	100	100	100	100	100	100	0	0

11.3 Experiment 3: Dyck-1 Acceptance using the DNNC as a Recurrent Hidden Layer

Our previous experiments using the DNNC all used FF models. In this experiment, we incrementally increase the difficulty of the learning by activating the recurrent connection in the DNNC. We also use entire Dyck-1 sequences as opposed to our previous experiments where a single time step was used. The objective of this experiment is to gradually make the learning task more difficult in order to determine when the learning process starts to fail when the DNNC is integrated in a NN architecture. By doing so, we hope to be able to determine the shortcomings of our DNNC module. This experiment involves the use of the DNNC as a recurrent hidden layer. The model is tested on a Dyck-1 acceptance task, where the goal is to classify Dyck-1 sequences as valid or invalid. This is a slightly more complex model than the one in the previous experiment, because the recurrent connection in the DNNC is used. The learning task is also more difficult because full sequences spanning several time steps are passed through the model, as opposed to the previous 2 experiments where each dataset element required only a single time step. Additionally, the training in this experiment will require backpropagation through time, at the very end of a sequence, not after every time step. We increase the difficulty of the

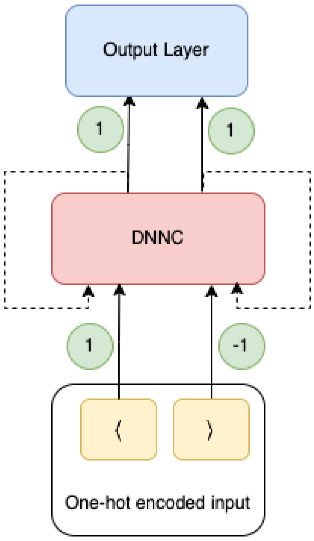


Figure 11.3: The architecture of the model where the DNNC is used as a recurrent hidden layer.

task and complexity of the model to determine the limit to which the model can learn to use the DNNC. We also want to test the effect of the DNNC on the systematicity of the model, and whether having the DNNC improves generalisation.

The model we use here consists of 3 layers, a fully connected input layer with 2 neurons, the recurrent DNNC layer and a fully connected output layer with one neuron. The model is shown in Figure 11.3, and the correct weights are also shown. The input layer has no activation function, whereas the output layer has either a sigmoid activation function or a clipping activation function. The model is mostly tested without biases in the input and output layers. However, we test some of the configurations with biases.

11.3.1 Experimental Setup

The complexity of the model is increased to determine the extent to which the model can learn the Dyck-1 language using the DNNC. We aim to find the best configuration for the DNNC model, and detect any potential shortcomings in the DNNC. We also want to determine if any unlearning happens when the model is correctly initialised. We use two output activation functions for this model.

1. **Sigmoid:** Here a sigmoid activation is used for the output layer.

2. **Clipping:** Here a clipping activation is used for the output layer. This bounds the output between 0 and 1.

Both configurations are trained under a variety of different conditions:

1. Correct/Random weight initialisation with no fixed weights.

2. Correct/Random initialisation with fixed input layer weights.

3. Correct/Random initialisation with fixed output layer weights.

Examples of the correct weights are shown in Figure 11.3. For both the sigmoid and clipping activations, the models are tested with and without biases when the weights are randomly initialised and no layers are frozen. Each configuration and condition combination is run 10 times.

For this experiment, we use Variant 1 of the Dyck-1 Validity Datasets (see Dataset 1). We train our models over 10,000 epochs. We use an Adam optimiser with a learning rate of 0.001 and the MSE loss function. 70% of our short dataset is used for training and the remainder is used for testing. The dataset used in training is balanced, and consists of Dyck-1 sequences of lengths 2 and 4 tokens. Before each sequence is passed to the model, the DNNC state is reset. Feedback is given to the model at the very end of the sequence, not after every time step.

The remaining 30% of the short dataset is used for testing. The dataset is balanced and consists of Dyck-1 sequences of lengths 2 and 4 tokens. The long dataset consisting of sequences of lengths 2 to 12 tokens is completely excluded from training and used to test the model's ability to generalise to longer sequences. This dataset is also balanced. Before each sequence is passed to the model, the DNNC state is reset.

11.3.2 Results

We test our models under a number of different conditions and configurations. The results for this experiment are shown in Tables 11.3 and 11.4. Our results show that under most conditions, it is possible for the model to learn the correct weights to operate the DNNC. When a traditional classification setup is used, the model seems to struggle to learn the input layer parameters from random weights. We also observe that unlearning happens when a sigmoid activation function is used and the model is trained from correct weights. On the other hand, when clipping is used and a model is trained from correct weights, unlearning does not occur. When a sigmoid is used with biases, it is evident that the model performs better when only one layer is trainable, not both. When clipping is used without biases and training from random weights, the model achieves better results when both input and output layers are trainable. When used with biases and training from random weights, we observe that the sigmoid configuration outperforms the clipping configuration. The clipping configuration without biases achieves better performance than the sigmoid configuration regardless of whether or not biases are used. Similar to the previous experiment, where the DNNC is used as a hidden layer in a FF setting and biases are not used, the model for this experiment which uses a clipping activation function and no biases outperforms the sigmoid configuration with and without biases. The best performance for both this experiment and the previous experiment are achieved when a clipping activation is used without biases. The recurrent model outperforms the FF model. The recurrent sigmoid configuration with biases achieves slightly better performance than the sigmoid configuration without biases. The use of the DNNC results in better generalisation when compared to the standard models from the baseline experiments in Chapter 4. The highest accuracy on the test and long test sets in the plain models was achieved by an LSTM with 3 hidden units. The highest accuracy on the long test set in this experiment was 85.17%, which is higher than the 80.07% result achieved by the plain LSTM on the long test set. The stack-counter has provided some generalisation benefits over the plain models.

11.4 Experiment 4: Dyck-1 Acceptance using DNNC Augmented Recurrent Networks (Mixed Models)

In our previous experiments, we test the DNNC as a hidden layer on its own. We test in with both feed-forward and recurrent settings. This experiment further complicates the learning task. The DNNC is used as part of a recurrent hidden layer. It is placed

Table 11.3: Recurrent DNNC Dyck-1 Acceptance Results - No Biases

Weight	**Output**	**Fixed**		**Train**			**Test**			**Long**	
Initialisation	**Activation**	**Weights**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
Random	Sigmoid	None	45.83	70.83	52.5	36.36	100	52.73	49.94	100	54.98
Random	Sigmoid	Input	100	100	100	100	100	100	100	100	100
Random	Sigmoid	Output	66.67	79.17	72.92	45.45	100	78.18	49.94	100	79.98
Correct	Sigmoid	None	62.5	62.5	62.5	36.36	36.36	36.36	49.94	49.94	49.94
Correct	Sigmoid	Input	100	100	100	100	100	100	100	100	100
Correct	Sigmoid	Output	70.83	70.83	70.83	100	100	100	100	100	100
Random	Clipping	None	50	100	87.5	54.55	100	88.18	50	100	85.17
Random	Clipping	Input	50	100	81.67	54.55	100	74.55	50	100	79.94
Random	Clipping	Output	50	100	80	36.36	100	74.55	49.94	100	79.98
Correct	Clipping	None	100	100	100	100	100	100	100	100	100

Table 11.4: Recurrent DNNC Dyck-1 Acceptance Results - With Biases

Weight	**Output**	**Fixed**		**Train**			**Test**			**Long**	
Initialisation	**Activation**	**Weights**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
Random	Sigmoid	None	54.17	100	77.92	45.45	100	61.82	50	100	60.53
Random	Clipping	None	29.12	75	54.17	45.45	63.64	54.55	46.69	51.87	49.53

alongside a single layer recurrent network (RNN, LSTM, or GRU). The objective of this experiment is to determine the extent to which NN models can learn to use the DNNC, and to identify shortcomings in our DNNC module and the models used. Similar to the previous experiment, where the DNNC is tested in a recurrent setting, we perform binary classification on Dyck-1 sequences.

The model used in this experiment is made up of 3 layers. The first layer is a fully connected input layer, followed by a recurrent hidden layer which is made up of a single layer RNN/LSTM/GRU alongside the DNNC. The final layer is the output layer which consists of one neuron with a sigmoid or clipping activation function. The DNNC has 2 inputs, and we vary the number of RNN/LSTM/GRU units in the hidden layer. We test the model with 2, 3, and 4 hidden RNN/LSTM/GRU units. As a result, the size of our input layer is either 4, 5, or 6 neurons. For all layers, biases are used in this experiment. The recurrent connection in the DNNC is used in this experiment. The architecture of the model used is shown in Figure 11.4.

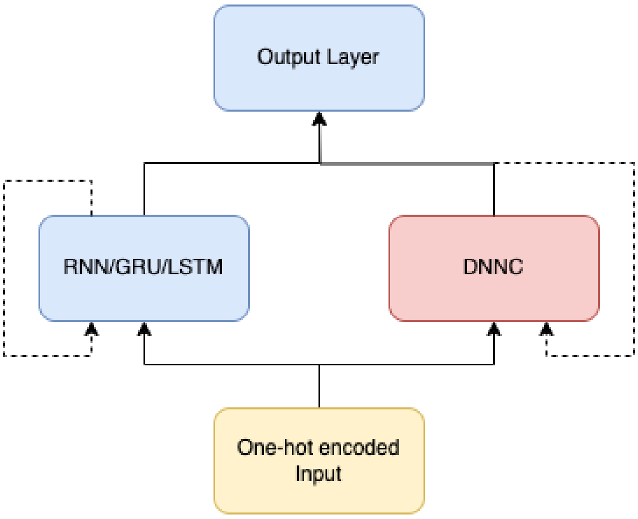


Figure 11.4: Architecture of the model used for Dyck-1 acceptance where the DNNC is used alongside a standard RNN in the hidden layer.

11.4.1 Experimental Setup

The complexity of the model is increased to determine the extent to which the model can systematically learn the Dyck-1 language using the DNNC. We aim to find the best configuration for the DNNC model, and detect any potential shortcomings in the DNNC. We also want to determine if any unlearning happens when the model is correctly initialised. We use two different output activation functions for this model.

1. **Sigmoid:** Here a sigmoid activation is used for the output layer.
2. **Clipping:** Here a clipping activation is used for the output layer. This bounds the output between 0 and 1.

The model is always trained from random weights. Each configuration of the model is run 10 times.

For this experiment, we use Variant 1 of the Dyck-1 Validity Datasets (see Dataset 1). We train our models over 10,000 epochs. We use an Adam optimiser with a learning rate of 0.001 and the MSE loss function. 70% of our short dataset is used for training and the remainder is used for testing. The dataset used in training is balanced, and consists of Dyck-1 sequences of lengths 2 and 4 tokens. Before each sequence is passed to the model, the stack state is reset. Feedback is given to the model at the very end of the sequence, not after every time step.

The remaining 30% of the short dataset is used for testing. The dataset is balanced and consists of Dyck-1 sequences of lengths 2 and 4 tokens. The long dataset consisting of sequences of lengths 2 to 12 tokens is completely excluded from training and used to test the model's ability to generalise to longer sequences. This dataset is also balanced. Before each sequence is passed to the model, the DNNC state is reset.

11.4.2 Results

This experiment involves a more complicated learning task than the previous experiments involving the use of the DNNC. Here, we incorporate the DNNC as part of a recurrent hidden layer, alongside a single layer recurrent network. The results of our experiments are summarised in Table 11.5.

Table 11.5: DNNC Augmented Recurrent Networks Dyck-1 Acceptance Results using Adam optimiser

	Output	Hidden	Train			Test			Long		
Model	**Activation**	**Units**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
RNN+DNNC	Sigmoid	2	95.83	100	97.92	72.73	100	83.64	50	90.67	69.49
GRU+DNNC	Sigmoid	2	91.67	100	98.75	90.91	100	94.55	50.88	100	76.24
LSTM+DNNC	Sigmoid	2	100	100	100	72.73	100	85.45	50	85.72	70.08
RNN+DNNC	Clipping	2	50	100	69.58	54.55	100	69.09	50	91.86	62.92
GRU+DNNC	Clipping	2	50	100	75	45.45	100	72.73	47.49	100	63.49
LSTM+DNNC	Clipping	2	50	100	90	54.55	100	81.82	50	100	71.65
RNN+DNNC	Sigmoid	3	95.83	100	98.75	63.64	90.91	81.82	47.82	80.74	62.13
GRU+DNNC	Sigmoid	3	100	100	100	81.82	100	91.82	45.63	89.09	63.39
LSTM+DNNC	Sigmoid	3	100	100	100	81.82	90.91	88.18	49.42	98.43	73.54
RNN+DNNC	Clipping	3	50	100	80	54.55	81.82	70.91	50	53.63	50.82
GRU+DNNC	Clipping	3	50	100	74.58	54.55	81.82	68.18	45.99	96.55	53.95
LSTM+DNNC	Clipping	3	50	100	70	45.45	90.91	65.45	49.97	90.09	58.74
RNN+DNNC	Sigmoid	4	95.83	100	99.17	72.73	100	84.55	51.18	73.6	67.3
GRU+DNNC	Sigmoid	4	100	100	100	81.82	100	90.91	53.72	81.66	67.59
LSTM+DNNC	Sigmoid	4	100	100	100	72.73	90.91	82.73	51.66	71.86	62.71
RNN+DNNC	Clipping	4	50	100	79.17	54.55	100	75.45	48.25	98.2	57.99
GRU+DNNC	Clipping	4	50	100	89.58	54.55	90.91	80	50	61.98	52.39
LSTM+DNNC	Clipping	4	50	100	85	54.55	90.91	77.27	50	88.35	56.81

All models in this experiment are capable of converging to 100% training accuracy, and most are capable of achieving 100% test accuracy in at least one run. The LSTM+DNNC models always converge to 100% training accuracy when a sigmoid activation is used. When a clipping activation is used, the LSTM+DNNC models do not always converge to 100% training accuracy. The highest accuracy achieved on the short and long test sets was using the GRU+DNNC model with 2 hidden units. In general, the sigmoid configuration seems to yield better results for all models in comparison to their counterparts which use a clipping activation function. The greatest improvement over the standard recurrent models in Chapter 4 seems to be evident in the GRU+DNNC models. Adding the DNNC to the GRU models seems to improve the performance on the short test set, and the generalisation on the longer test set. This is particularly evident when the sigmoid configuration is used.

11.5 Summary

This section describes the experiments where we integrate our discrete DNNC module into NN models. The use of our module has increased the systematicity of models. However, we would like to further evaluate the DNNC to gain a better understanding of the learning dynamics, and the effect of the DNNC.

Chapter 12

Conclusions

12.1 Summary

In this thesis, we address counting behaviour in RNNs. We address the 3 research questions defined in Chapter 1. We use 3 different approaches to answer the research questions: an empirical approach for the first question, a theoretical approach for the second question and a constructive approach for the third question. The empirical, theoretical and constructive approaches are detailed in Parts I, II and III, respectively.

12.1.1 Empirical Approach

In Part I, we empirically address the first research question: to what extent can RNNs be trained to count and generalise beyond the training data? Using a Dyck-1 acceptance task, we evaluate the effect of different hyperparameters on RNN learning and generalisation of counting behaviour. We find that in general, RNNs do not learn counting in training precisely enough to generalise to arbitrarily long sequences.

In Chapter 4, we investigate the effect of using different activation and loss functions on RNN learning of counting behaviour. We use a traditional classification configuration where a sigmoid output activation function is used in combination with a cross entropy loss function, and then we use a regression configuration, where the output is clipped between 0 and 1 and we combine this with mean squared error loss. We test the effect of using different output activation functions on the learning and generalisation of counting behaviour in LSTM, GRU and Elman RNNs with different hidden layer sizes. We find that the models fail to generalise to longer sequences and that the learning is generally more reliable and effective when the sigmoid activation function is used.

We test the classification and regression configurations on linear and ReLU RNNs initialised with random and correct weights. When we train the models from random weights, we find that they learn counting behaviour more effectively when the classification configuration is used compared to the regression configuration. However, when trained from correct weights, the classification configuration results in models unlearning the correct weights, while in the regression setup, the models retain the correct weights. We also train the models with training sequences of different lengths, and we find that the models train and generalise more effectively as the training sequences get longer. We also find that the unlearning of correct weights is less drastic when longer training sequences are used. To our knowledge, we are the first to investigate model unlearning of weights in training.

In Chapter 5, we focus on the learning and generalisation of counting behaviour in single-cell LSTM, GRU and ReLU RNN models. We train our models on datasets similar to those used by Suzgun et al. (2019a). We find that LSTMs learn counting behaviour more reliably than ReLU RNNs and GRUs. ReLU RNNs in general are difficult to train, and their learning of counting behaviour is unreliable. We use significantly longer sequences to test model generalisation, namely, the Very Long and Zigzag datasets. When we test our models on these datasets, we find that they mostly fail on all elements of these datasets.

We then investigate the behaviour of different models on the longer datasets. We introduce a new metric for evaluating generalisation to longer sequences, the First Point of Failure (FPF), which is the first point in the sequence where the model produces an incorrect label. We record the FPF for well trained LSTM, GRU and ReLU RNN models on the Zigzag dataset and we find that the models exhibit different failure modes. The LSTMs consistently count short and fail before the count reaches 0. The GRU models fail just after the models start a sequence of decrements. ReLU RNNs do not exhibit a systematic bias towards counting long or short.

We investigate the effect of loss on the generalisation to longer sequences. We correlate loss and FPF for our models and we find that lower loss can possibly indicate better generalisation, but this is not a reliable indicator, because we have observed models with higher loss and higher FPF at the same time. We then train LSTM and ReLU RNNs for longer and find that LSTMs do converge, but the decrease in the loss value gets smaller as the models converge. On the other hand, ReLU RNNs exhibit unpredictable and irregular behaviour when training. Therefore, we find that training for longer periods of time is not a practical approach for improving generalisation.

12.1.2 Theoretical Approach

In Part II, we address the second research question: under what conditions do RNNs count exactly? We focus on single-cell linear RNNs and single-cell ReLU RNNs.

In Chapter 7, we focus on linear RNNs. We formally define the Balanced Bracket (BB) language and the balanced bracket counter. We relate single-cell linear RNNs to the balaned bracket counter. We then propose a theorem where we define 2 Counter Indicator Conditions (CICs) on the weights of linear RNNs that lead to exact counting behaviour. We mathematically prove that the CICs are necessary and sufficient to achieve exact counting behaviour in linear RNNs.

We then train single-cell linear RNNs on training sequences of lengths 2, 4 and 8 tokens on both binary and ternary Dyck-1 classification tasks. We test our models on longer sequences of lengths 20 and 50 tokens to evaluate model generalisation to longer sequences. We find that in general the models do not learn exact counting and fail on longer sequences. The model performance decreases as test sequence length increases. We also find that the models train more effectively with longer sequences and therefore generalise more effectively to longer sequences.

We inspect the trained models and extract the weights and compare them to the CICs. We observe that the model weights approach the CICs but do not reach them exactly. We also observe that weights of models trained with longer sequences are closer to the CICs than the weights of models trained with shorter sequences.

In Chapter 8, we focus on single-cell ReLU RNNs. We formally define the semi-Dyck-1 language, a superset of Dyck-1 which can be fully accepted by a single-cell ReLU RNN and the corresponding semi-Dyck-1 counter. We propose a theorem which relates the a single-cell ReLU RNN to the semi-Dyck-1 counter and defines 3 Counter Indicator Conditions (CICs) on the weights of a ReLU RNN that result in exact counting behaviour. We then prove that the fulfillment of the CICs is equivalent to the ReLU RNN accepting the semi-Dyck-1 language and to accepting the language of the semi-Dyck-1 counter.

We train single-cell ReLU RNN models on both Dyck-1 and semi-Dyck-1 datasets. We train our models once from random weights, once from correct weights without biases in the ReLU layer, and once from correct weights with biases in the ReLU layer. We find that training from random weights is harder than training from correct weights. The models trained from correct weights without biases in the ReLU converge the most reliably. We test our models on longer sequences and we find that the models do not learn to count exactly and fail on longer sequences. We also find that the correctly initialised models unlearn the CICs and fail on longer sequences.

We investigate the effect of deviating from the CICs on Mean Squared Error (MSE) and Binary Cross Entropy (BCE) validation loss, and the First Point of Failure (FPF), which we define in Chapter 5. We find that the correct model which fulfills the CICs aligns with the highest FPF. However, the lowest MSE and BCE losses are not located at the point of the CICs. This explains the models unlearning the CICs when trained with MSE and BCE loss. While we do not have any hypotheses about the causes of the mismatch between the CICs and local minima of the loss functions, further investigations may help provide a better understanding of this.

12.1.3 Constructive Approach

Having proven the CICs in Part II, and showing in both Parts I and II that RNNs do not learn exact counting and fail to generalise systematically to arbitrarily long sequences, we propose a constructive solution in Part III, where we address the third research question: can we design a discrete counter that can be integrated into RNNs trained with backpropagation and what is the effect of using it? We propose a Discrete Non-Negative Counter (DNNC) module to integrate counting behaviour in RNNs.

In Chapter 10, we describe the design of the DNNC module. We equip the DNNC with artificial gradients to allow for training to occur with backpropagation. The artificial gradients are designed based on the intended behaviour of the DNNC. We implement the DNNC in PyTorch and design unit tests that ensure that it is functioning as intended in Appendix F.

After ensuring that the DNNC is functioning as intended, we integrate it into different models in Chapter 11. We first test the performance of the DNNC in a feed-forward setting. We then integrate our models into recurrent architectures where we use the DNNC either as the recurrent hidden layer in the model, or as part of a recurrent hidden layer alongside an LSTM, Elman or GRU. We test different activation functions in these experiments. We use the sigmoid activation function and the clipping activation function, where the output is clipped between 0 and 1. We use the Dyck-1 acceptance task as in Chapter 4. We train models from random and correct weight initialisations when using the DNNC as a recurrent hidden layer. We also test the models when fixing different weights. We find that models trained from correct weights with do not unlearn the correct weights when a clipping activation is used and all weights are trainable. When the sigmoid output activation is used, the models unlearn the correct weights, unless the input weights are fixed. We find that the model performance improves with the DNNC but we do not achieve perfect performance, as sometimes the models have difficulty learning to control the DNNC. As we have seen in Part II, we have found a misalignment between CICs and the minima of the loss functions, and therefore, implementing a continuous structure that behaves discretely seems to be difficult. By implementing the DNNC using artificial gradients, we hope to provide a discretely operating structure that can be used with NN models that train with backpropagation.

12.2 Future Work

There are several questions that emerge from the findings of this work that can be explored in the future. We discuss here the ideas for future work for the empirical, theoretical and constructive approaches we used throughout this thesis.

12.2.1 Empirical Approach

In our experiments, we find that RNNs do not learn exact counting behaviour and even unlearn exact counting behaviour when trained with backpropagation in the most common setups. There are a number of different questions and directions to explore to better understand this behaviour and develop solutions that result in RNNs learning counting behaviour that generalises to arbitrarily long sequences.

A possible approach to address this problem could be the use of different learning paradigms such as meta-learning or reinforcement learning for the learning of counting behaviour. Meta-learning is used by Lake (2019) and Chen et al. (2020) used reinforcement learning to improve performance on the SCAN task (Lake and Baroni, 2018). Another approach could be to perform more rigorous investigations to understand the reason for the unlearning.

The models we investigate here are all minimal models, which is not what is typically used in realistic tasks like language modelling, part-of-speech (POS) tagging, machine translation, and parsing. The models used in these tasks are larger models, specifically LSTMs often as seq2seq models (Sutskever et al., 2014) and more recently mostly Transformers (Vaswani et al., 2017). Investigating the learning of counting behaviour by sub-networks within these models would be a useful next research task. One possible avenue to for this could be to to investigate the effect of the sequence structure, e.g. counting depth and sequence length.

12.2.2 Theoretical Approach

To better understand counting behaviour in RNNs, we determine and formally prove Counter Indicator Conditions (CICs) on the weights of single-cell linear and ReLU RNNs that indicate whether the model counts exactly.

The CICs we have determined are for linear and ReLU RNNs, but not for LSTMs. Weiss et al. (2018a) show that LSTMs can be trained to count more reliably than simple RNNs, which is consistent with our own findings. However, the LSTMs do not learn to count exactly. Therefore, it would be interesting to develop CICs that indicate when single-cell LSTMs count exactly. We predict that the conditions for LSTMs to count would rely on saturating the sigmoid and tanh activation functions, as suggested by Weiss et al. (2018a). We would like to determine the exact conditions that would lead to these activation functions being saturated and resulting in exact counting.

Weiss et al. (2018a) identified counting neurons within a network visually by using graphs of the activation values. With the CICs we could more formally determine counting neurons, even within larger networks performing more complex tasks.

As mentioned earlier, single-cell RNNs are not commonly used models, but may be part of a much larger model, which are more commonly used in practice. Therefore, extending the CICs to larger models and developing a general set of CICs that hold for different size models is a possible direction to explore in the future.

Experimentally, we find a mismatch between CICs in ReLU RNNs and the minimum of the loss functions on our datasets. Developing a better understanding of the learning dynamics of ReLU RNNs could lead to understanding this mismatch. Training methods can then be developed that align the CICs with the minimum of the loss function. One possible approach could be to use Bayesian priors similar to Kopparti and Weyde (2020).

12.2.3 Constructive Approach

As a constructive approach to integrate counting behaviour in RNNs, we introduce the Discrete Non-Negative Counter (DNNC) module, which we equip with artificial gradients to allow for training with backpropagation. The DNNC improves counting behaviour in RNNs and there are several avenues for further development.

One possibility is to develop alternative options for the design of the artificial gradients, as we observe in our experiments that models do not learn to control the DNNC optimally. Another direction to explore is the learning and generalisation of models equipped with a fixed CIC-compliant cell and compare to models equipped with the DNNC. These models could be tested on both simple and more complex tasks that consist of a counting element, such as arithmetic problems. We can also develop alternative counting mechanisms for NNs by using the reparametrisation techniques used by variational autoencoders (Kingma and Welling, 2013). This would also allow for training with backpropagation.

The DNNC is designed to be used with FFNNs and RNNs, while in most modern NLP applications, Transformer models are used. The architecture of Transformers is not recurrent like RNNs, and they operate on a large window that is processed in parallel. A counting mechanism might still be beneficial, and help deal with problems like the lack of sensitivity to word order (Sinha et al., 2021). Developing a counting mechanism for Transformers could also improve the performance of Large Language Models (LLMs) on mathematical tasks, as it is known that arithmetic problems are generally difficult for LLMs (Lewkowycz et al., 2022). The current solution is an interface with Wolfram in ChatGPT/GPT-4 (OpenAI, 2023; Wolfram) to solve textual problems that contain arithmetic tasks. However this approach requires the generation of Wolfram Language code, which is less efficient than machine learning that provides counting as part of its internal capabilities.

12.3 Overall Reflections

We hope to have made a contribution to understanding and improving systematic behaviour in NNs. However, our finding that counting, framed as Dyck-1 acceptance, is still limited in RNNs, combined with a similar observation for regular languages by Weiss et al. (2018b), seems to indicate that there is still work to be done before even context-free grammars can be learned exactly (Chomsky and Schützenberger, 1959). Beyond these formal language tasks, integrating discrete logic into NNs (Garcez and Lamb, 2023) seems to be a field that could also benefit from improved learning of counting to improve the reasoning abilities of current AI models.

This work can also contribute to different NLP projects, such as those that focus on arithmetic operations. Also, evolving the DNNC into a fully functioning stack ADT can help merge the neural approaches with traditional parsers and hopefully result in more effective parsing.

Looking back, it is surprising that counting, which is a very simple task, is so difficult for NNs to learn, especially since NNs are the basis for the current state-of-the-art AI. The path towards developing the work presented in this thesis has been an interesting and insightful journey and an incredible learning experience. We are looking forward to continuing exploring this path and developing this research in the future.

Bibliography

D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. 79

S. Ayache, R. Eyraud, and N. Goudian. Explaining black boxes on sequential data using weighted automata. In *International Conference on Grammatical Inference*, pages 81–103. PMLR, 2019. 79

S. Bader. *Neural-symbolic integration.* PhD thesis, Dresden University of Technology, 2009. URL `https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-25468`. 115

Y. Bengio, P. Y. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181. URL `https://doi.org/10.1109/72.279181`. 31

J.-P. Bernardy. Can recurrent neural networks learn nested recursion? *Linguistic Issues in Language Technology*, 16, 2018. URL `https://aclanthology.org/2018.lilt-16.1`. 22, 46

S. Bhattamishra, K. Ahuja, and N. Goyal. On the practical ability of recurrent neural networks to recognize hierarchical languages. In D. Scott, N. Bel, and C. Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 1481–1494. International Committee on Computational Linguistics, 2020. doi: 10.18653/v1/2020.coling-main.129. URL `https://doi.org/10.18653/v1/2020.coling-main.129`. 46

M. Bodén and J. Wiles. Context-free and context-sensitive dynamics in recurrent neural networks. *Connection Science*, 12(3-4):197–210, 2000. 22, 45

M. Bodén and J. Wiles. On learning context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 13(2):491–493, 2002. 22

X. Chen, C. Liang, A. W. Yu, D. Song, and D. Zhou. Compositional generalization via neural-symbolic stack machines. *arXiv preprint arXiv:2008.06662*, 2020. 114, 151

Y. Chen, S. Gilroy, A. Maletti, J. May, and K. Knight. Recurrent neural networks as weighted language recognizers. In M. A. Walker, H. Ji, and A. Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 2261–2271. Association for Computational Linguistics, 2018. doi: 10.18653/v1/n18-1205. URL `https://doi.org/10.18653/v1/n18-1205`. 21, 41, 78

K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014. doi: 10.3115/v1/d14-1179. URL `https://doi.org/10.3115/v1/d14-1179`. 33

N. Chomsky. Three models for the description of language. *IRE Trans. Inf. Theory*, 2 (3):113–124, 1956. doi: 10.1109/TIT.1956.1056813. URL `https://doi.org/10.1109/TIT.1956.1056813`. 36, 43, 78

N. Chomsky. Aspects of the theory of syntax special technical report no. 11. Technical report, 1965. 42

N. Chomsky and M. P. Schützenberger. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 118–161. Elsevier, 1959. 42, 43, 153

J. Collins, J. Sohl-Dickstein, and D. Sussillo. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913*, 2016. 44

V. Crollen, X. Seron, and M.-P. Noël. Is finger-counting necessary for the development of arithmetic abilities? *Frontiers in Psychology*, 2:242, 2011. 22

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. 21, 29, 78

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007. 115

G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, et al. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022. 43

T. Deleu and J. Dureau. Learning operations on a stack with neural turing machines. *arXiv preprint arXiv:1612.00827*, 2016. 112

F. Domahs, K. Moeller, S. Huber, K. Willmes, and H.-C. Nuerk. Embodied numerosity: implicit hand-based representations influence symbolic number processing across cultures. *Cognition*, 116(2):251–266, 2010. 22

N. El-Naggar, P. Madhyastha, and T. Weyde. Experiments in learning Dyck-1 languages with recurrent neural networks. In A. Bundy and D. Mareschal, editors, *Proceedings of the 3rd Human-Like Computing Workshop (HLC 2022) co-located with the 2nd International Joint Conference on Learning and Reasoning (IJCLR 2022), Windsor, United Kingdom, September 28-30th, 2022*, volume 3227 of *CEUR Workshop Proceedings*, pages 24–28. CEUR-WS.org, 2022a. URL `http://ceur-ws.org/Vol-3227/El-Naggar.PP5.pdf`. 17, 25, 49, 58, 62, 90, 91, 107

N. El-Naggar, P. Madhyastha, and T. Weyde. Exploring the long-term generalization of counting behavior in RNNs. In *I Can't Believe It's Not Better Workshop: Understanding Deep Learning Through Empirical Falsification*, 2022b. 25, 66

N. El-Naggar, P. Madhyastha, and T. Weyde. Theoretical conditions and empirical failure of bracket counting on long sequences with linear recurrent networks. *EACL 2023*, page 143, 2023a. 25, 82

N. El-Naggar, A. Ryzhikov, L. Daviaud, P. Madhyastha, and T. Weyde. Formal and empirical studies of counting behaviour in relu rnns. In *International Conference on Grammatical Inference*, pages 199–222. PMLR, 2023b. 25, 94

J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. 43

J. L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine learning*, 7:195–225, 1991. 22, 43

P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Counter machines and counter languages. *Math. Syst. Theory*, 2(3):265–283, 1968. doi: 10.1007/BF01694011. URL `https://doi.org/10.1007/BF01694011`. 37, 80

J. A. Fodor and Z. W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988. 21, 65

K.-i. Funahashi and Y. Nakamura. Neural networks, approximation theory, and dynamical systems. *Notes on the Institute of Mathematical Analysis*, 804:18–37, 1992. 21, 29, 78

A. d. Garcez and L. C. Lamb. Neurosymbolic ai: The 3 rd wave. *Artificial Intelligence Review*, pages 1–20, 2023. 153

A. S. d. Garcez, D. M. Gabbay, O. Ray, and J. Woods. Abductive reasoning in neural-symbolic systems. *Topoi*, 26:37–49, 2007. 115

R. Gelman and C. R. Gallistel. Language and the origin of numerical concepts. *Science*, 306(5695):441–443, 2004. 21

F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000. 42

F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Trans. Neural Networks*, 12(6):1333–1340, 2001. doi: 10.1109/72.963769. URL `https://doi.org/10.1109/72.963769`. 22, 45, 46, 47, 101, 102, 104

A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 112

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016. 112

E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom. Learning to transduce with unbounded memory. In *Advances in neural information processing systems*, pages 1828–1836, 2015. 112, 113

G. Hadash, E. Kermany, B. Carmeli, O. Lavi, G. Kour, and A. Jacovi. Estimate and replace: A novel approach to integrating deep neural networks with existing applications. *arXiv preprint arXiv:1804.09028*, 2018. 114, 115

Y. Hao, W. Merrill, D. Angluin, R. Frank, N. Amsel, A. Benz, and S. Mendelsohn. Context-free transductions with neural stacks. In T. Linzen, G. Chrupala, and A. Alishahi, editors, *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018, Brussels, Belgium, November 1, 2018*, pages 306–315. Association for Computational Linguistics, 2018. doi: 10.18653/v1/w18-5433. URL `https://doi.org/10.18653/v1/w18-5433`. 113

J. Hewitt, M. Hahn, S. Ganguli, P. Liang, and C. D. Manning. Rnns can generate bounded hierarchical languages with optimal memory. *arXiv preprint arXiv:2010.07515*, 2020. 22, 43, 44

S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991. 31

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`. 31

S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001. 31

S. Hölldobler, Y. Kalinke, and H. Lehmann. Designing a counter: Another case study of dynamics and activation landscapes in recurrent networks. In *KI-97: Advances in Artificial Intelligence: 21st Annual German Conference on Artificial Intelligence Freiburg, Germany, September 9–12, 1997 Proceedings 21*, pages 313–324. Springer, 1997. 22, 44

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 29

A. Jacovi, G. Hadash, E. Kermany, B. Carmeli, O. Lavi, G. Kour, and J. Berant. Neural network gradient-based learning of black-box function interfaces. *arXiv preprint arXiv:1901.03995*, 2019. 115

A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28, 2015. 112, 113, 114

Y. Kalinke and H. Lehmann. Computation in recurrent neural networks: From counters to iterated function systems. In *Australian Joint Conference on Artificial Intelligence*, pages 179–190. Springer, 1998. 22

A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015. 46

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 68, 106

D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 152

R. Kopparti and T. Weyde. Weight priors for learning identity relations. *arXiv preprint arXiv:2003.03125*, 2020. 152

B. M. Lake. Compositional generalization through meta sequence-to-sequence learning. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9788–9798, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/f4d0e2e7fc057a58f7ca4a391f01940a-Abstract.html. 151

B. M. Lake and M. Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2879–2888. PMLR, 2018. URL http://proceedings.mlr.press/v80/lake18a.html. 21, 65, 151

N. G. Lan, M. Geyer, E. Chemla, and R. Katzir. Minimum description length recurrent neural networks. *Trans. Assoc. Comput. Linguistics*, 10:785–799, 2022. URL https://transacl.org/ojs/index.php/tacl/article/view/3649. 22, 47

M. Le Corre, G. Van de Walle, E. M. Brannon, and S. Carey. Re-visiting the competence/performance debate in the acquisition of the counting principles. *Cognitive psychology*, 52(2):130–169, 2006. 21

M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993. 29

A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022. 153

A. Mali, A. Ororbia, D. Kifer, and L. Giles. Investigating backpropagation alternatives when learning to dynamically count with recurrent neural networks. In *International Conference on Grammatical Inference*, pages 154–175. PMLR, 2021. 22, 47

R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018. 115

G. F. Marcus, S. Vijayan, S. B. Rao, and P. M. Vishton. Rule learning by seven-month-old infants. *Science*, 283(5398):77–80, 1999. 21, 65

W. Merrill. Sequential neural networks as automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 1–13, 2019. 22, 45, 80

W. Merrill. On the linguistic capacity of real-time counter automata. *CoRR*, abs/2004.06866, 2020. URL `https://arxiv.org/abs/2004.06866`. 27, 37, 67, 80, 83, 85, 95

W. Merrill, G. Weiss, Y. Goldberg, R. Schwartz, N. A. Smith, and E. Yahav. A formal hierarchy of RNN architectures. In D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 443–459. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.43. URL `https://doi.org/10.18653/v1/2020.acl-main.43`. 43

M. Minsky and S. A. Papert. Perceptrons. *Cambridge, MA: MIT Press*, 6:318–362, 1969. 78

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/arXiv.2303.08774. URL `https://doi.org/10.48550/arXiv.2303.08774`. 153

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR*

Workshop and Conference Proceedings, pages 1310–1318. JMLR.org, 2013. URL `http://proceedings.mlr.press/v28/pascanu13.html`. 31

H. Peng, R. Schwartz, S. Thomson, and N. A. Smith. Rational recurrences. In E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1203–1214. Association for Computational Linguistics, 2018. doi: 10.18653/v1/d18-1152. URL `https://doi.org/10.18653/v1/d18-1152`. 43, 80

G. Rabusseau, B. Balle, and J. Pineau. Multitask spectral learning of weighted automata. *Advances in neural information processing systems*, 30, 2017. 79

G. Rabusseau, T. Li, and D. Precup. Connecting weighted automata and recurrent neural networks through spectral learning. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1630–1639. PMLR, 2019. 79

P. Rodriguez. Simple recurrent networks learn context-free and context-sensitive languages by counting. *Neural computation*, 13(9):2093–2118, 2001. 22, 42

P. Rodriguez and J. Wiles. Recurrent neural networks can learn to implement symbol-sensitive counting. *Advances in Neural Information Processing Systems*, 10, 1997. 22, 45

P. Rodriguez, J. Wiles, and J. L. Elman. A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40, 1999. 45

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. 22, 42, 78

A. M. Schäfer and H. G. Zimmermann. Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 632–640. Springer, 2006. 21, 78

L. Sennhauser and R. C. Berwick. Evaluating the ability of lstms to learn context-free grammars. *arXiv preprint arXiv:1811.02611*, 2018. 22

R. Shen, S. Bubeck, R. Eldan, Y. T. Lee, Y. Li, and Y. Zhang. Positional description matters for transformers arithmetic. *arXiv preprint arXiv:2311.14737*, 2023. 116

H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 440–449, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 089791497X. doi: 10.1145/130385.130432. URL `https://doi.org/10.1145/130385.130432`. 21, 22, 29, 41, 78

K. Sinha, R. Jia, D. Hupkes, J. Pineau, A. Williams, and D. Kiela. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little. *arXiv preprint arXiv:2104.06644*, 2021. 153

N. Skachkova, T. A. Trost, and D. Klakow. Closing brackets with recurrent neural networks. In T. Linzen, G. Chrupala, and A. Alishahi, editors, *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018, Brussels, Belgium, November 1, 2018*, pages 232–239. Association for Computational Linguistics, 2018. doi: 10.18653/v1/w18-5425. URL `https://doi.org/10.18653/v1/w18-5425`. 46

A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022. 21

G.-Z. Sun, C. L. Giles, and H.-H. Chen. The neural network pushdown automaton: Architecture, dynamics and training. *International School on Neural Networks, Initiated by IIASS and EMFCSC*, pages 296–345, 1997. 112, 113

G.-Z. Sun, C. L. Giles, H.-H. Chen, and Y.-C. Lee. The neural network pushdown automaton: Model, stack and learning simulations. *arXiv preprint arXiv:1711.05738*, 2017. 112, 113

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014. 151

M. Suzgun, S. Gehrmann, Y. Belinkov, and S. M. Shieber. Lstm networks can perform dynamic counting. *arXiv preprint arXiv:1906.03648*, 2019a. 22, 46, 47, 51, 52, 66, 67, 68, 69, 84, 101, 102, 104, 147

M. Suzgun, S. Gehrmann, Y. Belinkov, and S. M. Shieber. Memory-augmented recurrent neural networks can learn generalized dyck languages. *arXiv preprint arXiv:1911.03329*, 2019b. 114

L. Tao, Y.-X. Huang, W.-Z. Dai, and Y. Jiang. Deciphering raw data in neuro-symbolic learning with provable guarantees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 15310–15318, 2024. 115

A. Trask, F. Hill, S. E. Reed, J. Rae, C. Dyer, and P. Blunsom. Neural arithmetic logic units. *Advances in neural information processing systems*, 31, 2018. 112

A. Trott, C. Xiong, and R. Socher. Interpretable counting for visual question answering. *arXiv preprint arXiv:1712.08697*, 2017. 22

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 21, 28, 43, 151

G. Weiss, Y. Goldberg, and E. Yahav. On the practical computational power of finite precision RNNs for language recognition. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, pages 740–745. Association for Computational Linguistics, 2018a. doi: 10.18653/v1/P18-2117. URL `https://aclanthology.org/P18-2117/`. 22, 44, 45, 46, 51, 69, 71, 72, 73, 80, 101, 102, 104, 151, 152

G. Weiss, Y. Goldberg, and E. Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pages 5247–5256. PMLR, 2018b. 79, 153

J. Weston, S. Chopra, and A. Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014. 112

J. Wiles and J. Elman. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the seventeenth annual conference of the cognitive science society*, number s 482, page 487. MIT Press Cambridge, MA, 1995. 22, 42, 45

R. J. Williams and D. Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection science*, 1(1):87–111, 1989. 113

S. Wolfram. Chatgpt gets its 'wolfram superpowers'! `https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/`. Accessed: 11/12/2023. 153

D. M. Yellin and G. Weiss. Synthesizing context-free grammars from recurrent neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–369. Springer, 2021. 79

Appendix A

Model Output and Loss Functions

Definition A.1 (Model Output Calculation)**.** The model's output layer performs the following calculation:

$$Y = \sigma(W_Y h_t + b_Y),$$

where Y is the output W_Y is the output weight, h_t is the output of the ReLU hidden layer, and b_Y is the output bias. σ is the logistic sigmoid function

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

where e is Euler's number.

Definition A.2 (Mean Squared Error MSE Loss (MSE))**.** Mean Squared Error loss is calculated as follows:

$$MSELoss = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

where n is the length of the string, Y_i is the predicted output value and $\hat{Y}_i$ is the target output value at timestep i.

Definition A.3 (Binary Cross Entropy Loss (BCE))**.** Binary Cross Entropy Loss is calculated using the following equation:

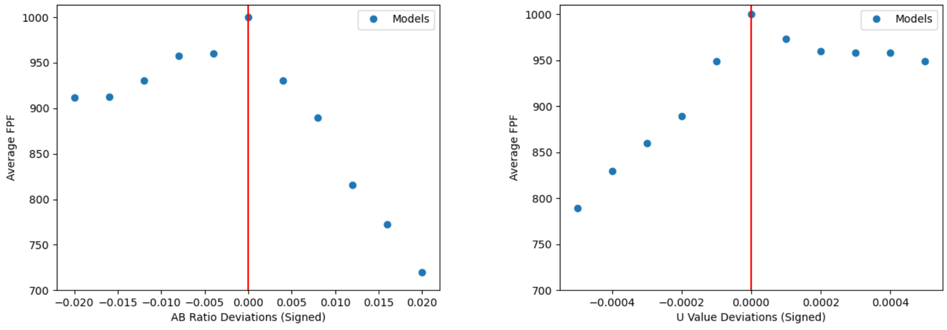
$$BCELoss = -\sum_{n=i}^{c} \hat{Y}_i \log(Y_i) + (1 - \hat{Y}_i) \log(1 - Y)$$

where Y is the predicted output, $\hat{Y}$ is the target output and c is the number of classes.

Appendix B

Deviation from CICs - FPF and Loss Plots

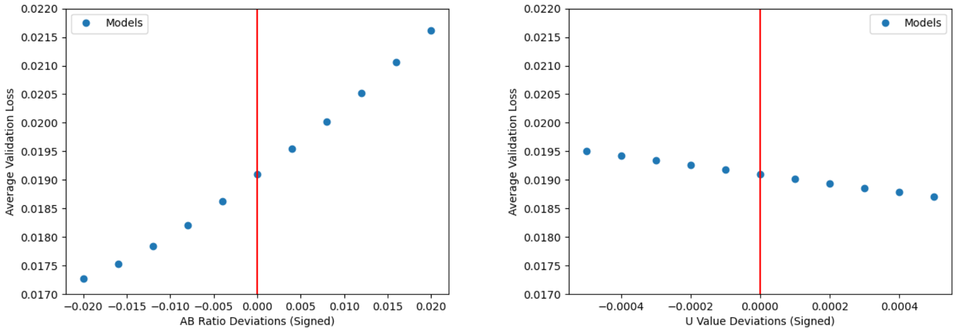
See Figure B.1 and Figure B.2.



(a) a/b Deviations and FPF

(b) U Deviations and FPF

Figure B.1: Effect of Deviations from the CICs on the First point of Failure (maximum is 1000).



(a) a/b Deviations and Validation Loss

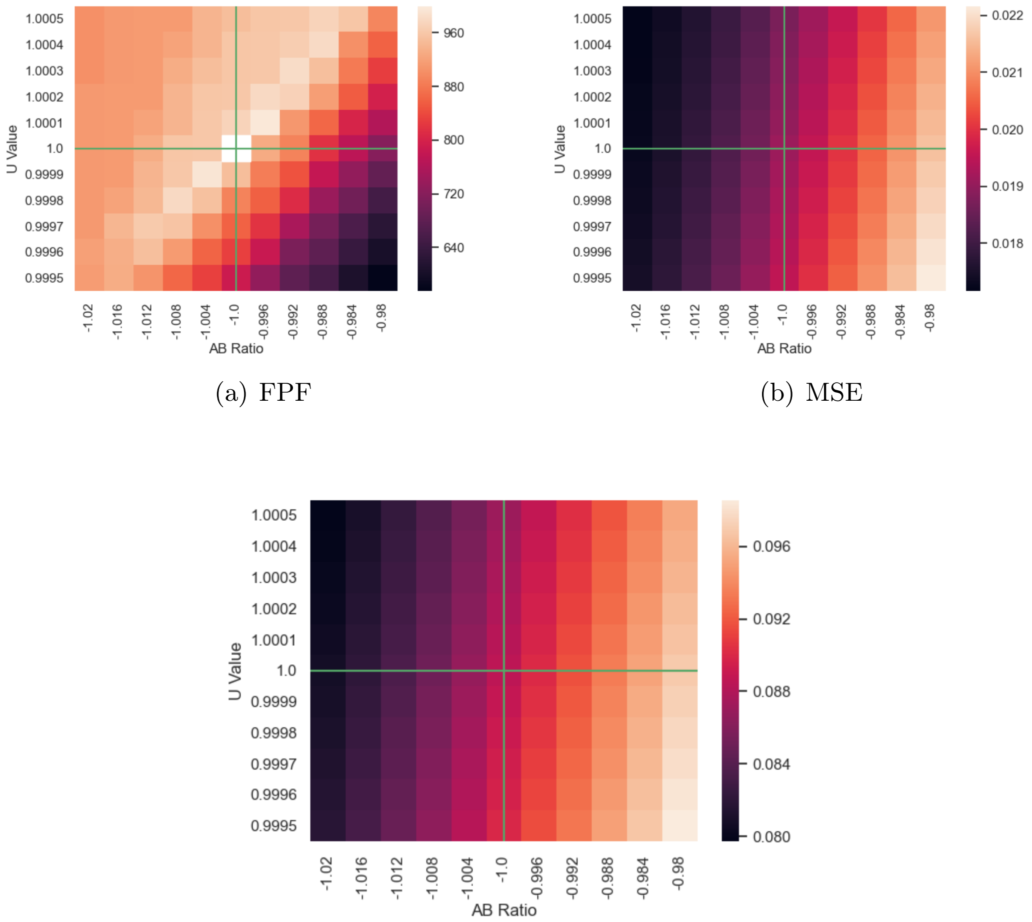
(b) U Deviations and Validation Loss

Figure B.2: Effect of Deviations in CICs on the Validation Loss

Appendix C

ReLU RNN Deviation from CICs - FPF and Loss Heatmaps.

See Figure C.1 and Figure C.2.



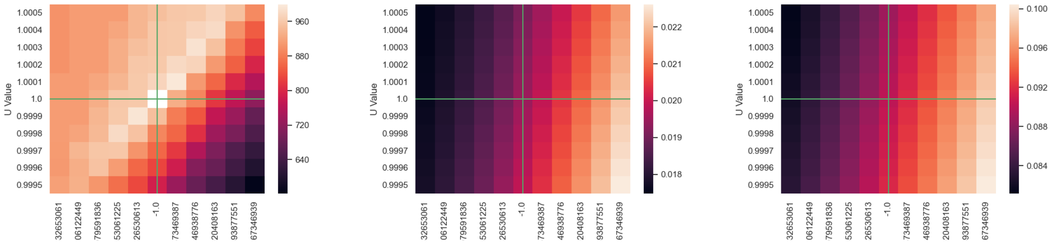
(a) FPF

(b) MSE

(c) BCE

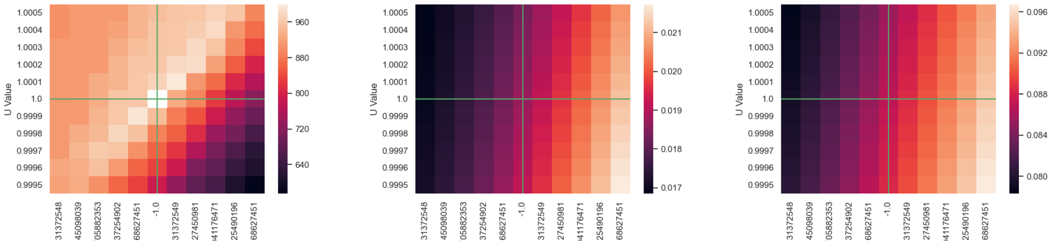
Figure C.1: Heatmaps showing the FPF values on the Dyck-1 Very Long dataset and MSE and BCE loss on the Dyck-1 Validation dataset for models with a correct configuration and with deviations. The thin green lines represent the CIC values for the a/b ratio and U value, and the intersection between the green lines is the point of a correct model. For FPF, the value in the centre of graph (a) is undefined, as no failures occurred for the correct model. It can be seen that the lowest MSE and BCE loss values are not located at the position of the correct model configurations.

b=0.98



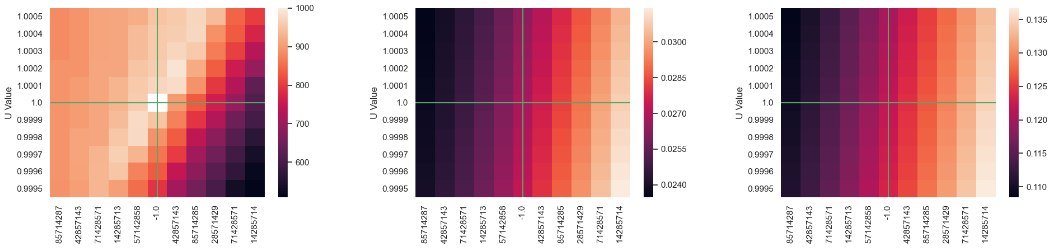
(a) FPF (b) MSE (c) BCE

b=1.02



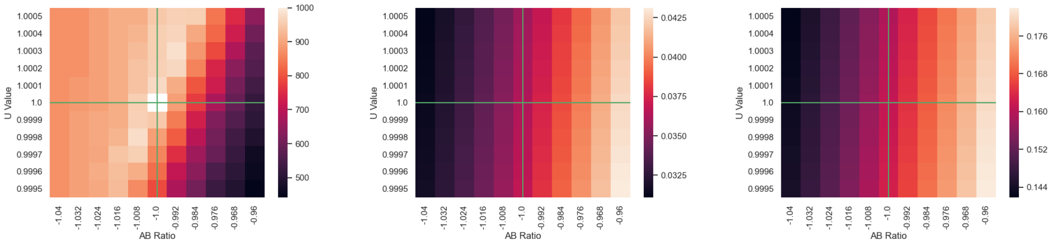
(d) FPF (e) MSE (f) BCE

b=0.7

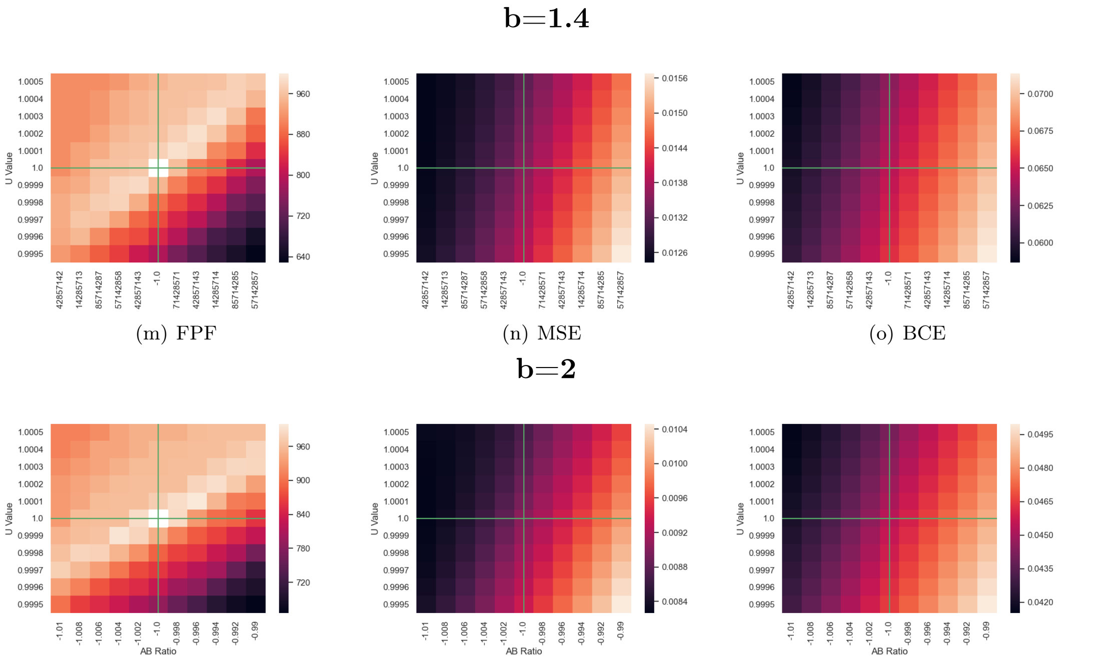


(g) FPF (h) MSE (i) BCE

b=0.5



(j) FPF (k) MSE (l) BCE



b=1.4

(m) FPF

(n) MSE

(o) BCE

b=2

(p) FPF

(q) MSE

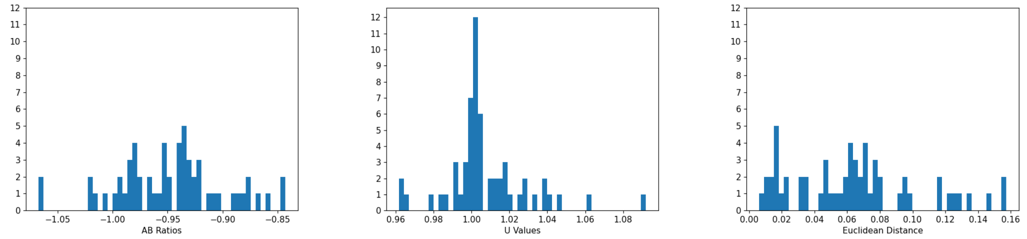
(r) BCE

Figure C.2: Heatmaps showing the FPF values on the Dyck-1 Very Long dataset and MSE and BCE loss on the Dyck-1 Validation dataset for models with a correct configuration and with deviations. The thin green lines represent the CIC values for the a/b ratio and U value, and the intersection between the green lines is the point of a correct model. It can be seen that the lowest MSE and BCE loss values are not located at the position of the correct model configurations.

Appendix D

Distribution of CICs in ReLU RNN Models Trained from Random Initialisation

We inspect the 12 models successfully trained on Dyck-1 from random initialisation with MSE loss and extract the U value and a/b ratio. We verify that $a > 0$ and plot the distribution of the a/b ratio and U and Euclidean distance between the observed $[a/b, U]$ and the correct $[-1, 1]$ in Figure D.1. The models do not reach the correct combination of values. The U value distribution has a clear peak at 1. The a/b ratio has a broader distribution with the mean not at -1, but slightly above.



(a) a/b Ratio

(b) Recurrent Weight U

(c) Euclidean Distance

Figure D.1: Distributions of the CICs over the 12 converged M/RI Dyck-1 models.

Appendix E

Testing ReLU RNNs on Dyck-1 Strings for semi-Dyck-1 Acceptance

The Dyck-1 datasets contain only valid Dyck-1 strings with balanced opening and closing brackets. The prefix sequences $(x_0, \ldots, x_k, \text{where}\, k < n)$ are also evaluated, where we have excess opening brackets or balanced brackets. The case of excess closing brackets does not occur in these datasets. Dyck-1 acceptance (where the excess closing brackets are invalid) or semi-Dyck-1 acceptance (where the results of excess closing brackets are valid) are therefore not fully covered by these datasets. However, in the case of a single-cell ReLU RNN $\mathcal{R}$, if b

R ≤ 0 and $h_t = 0$ for a balanced bracket string s with t tokens, then for $s_{t+1} =\rangle$, we have $h_{t+1} = 0$ by Definition 8.5. If $\mathcal{R}$ is a semi-Dyck-1 counter for excess open and for balanced bracket strings, it has to have $b \leq 0$, following the same argument as in the proof of Theorem 8.1. Therefore, it is sufficient to test excess opening and balanced bracket strings to test if a ReLU RNN accepts the semi-Dyck-1 language.

Appendix F

Testing the Implementation of the Discrete Non-Negative Counter Module

Before integrating the DNNC module into NN models, we have to ensure that it functions as intended, based on the design in Chapter 10. In this chapter, we describe the design of the unit tests for the forward and backward passes of the DNNC module. The unit tests are designed based on the logic defined in Chapter 10. We also ensure that all edge cases are covered by testing them individually. All unit tests are implemented in Python.

F.1 The Forward Pass

The testing of the forward pass consists of input logic tests and operational logic tests. The test cases for the input logic are based on the logic defined in section 10.2.1, and the test cases for the operational logic are created based on the logic defined in 10.2.2. To cover the largest range of possibilities, we create test cases for each of the three operations **Push** , **Pop**, and **NoOp**, then create smaller test cases for each operation based on the applicable logic.

F.1.1 Testing the Competitive Input Logic

The competitive input logic determines the operation that the DNNC will perform. The test sets are created based on the competitive input logic defined in equations 10.1, 10.2, and 10.3. In order to create test cases with high input coverage, the conditions that trigger each operation are tested separately using randomly generated numbers, and the output observed. Edge cases are determined and also tested separately. The values of $threshold_{push}$ and $threshold_{pop}$ are set to 0.5.

The following sets of test cases are used to test the input logic:

1. Test cases that trigger **Push** ($push = 1, pop = 0$):
   (a) $push\_input \geq threshold_{push}$ and $pop\_input \leq threshold_{pop}$.
   (b) $push\_input \geq threshold_{push}$ and $pop\_input \geq threshold_{pop}$ and $push\_input \geq pop\_input$.
2. Test cases that trigger **Pop** ($push = 0, pop = 1$):
   (a) $push\_input < threshold_{push}$ and $pop\_input \geq threshold_{pop}$.
   (b) $push\_input \geq threshold_{push}$ and $pop\_input \geq threshold_{pop}$ and $push\_input < pop\_input$.
3. Test cases that trigger **NoOp** ($push = pop = 0$):
   (a) $push\_input < threshold_{push}$ and $pop\_input < threshold_{pop}$.
4. Edge Cases:
   (a) $push\_input \geq threshold_{push}$ and $pop\_input \geq threshold_{pop}$ and $push\_input = pop\_input$. This should trigger **Push**.
   (b) $push\_input = threshold_{push}$ and $pop\_input = threshold_{pop}$ and $threshold_{push} = threshold_{pop}$.

      This should trigger **Push**.
   (c) $push\_input < threshold_{push}$ and $pop\_input = threshold_{pop}$.

      This should trigger **Pop**.

(d) $push\_input \geq threshold_{push}$ and $pop\_input < threshold_{pop}$ and $push\_input < pop\_input$.

This should trigger **Push**.

(e) $push\_input < threshold_{push}$ and $pop\_input \geq threshold_{pop}$ and $push\_input \geq pop\_input$.

This should trigger **Pop**.

For each test case, 20 instances were randomly generated and tested when applicable. For all test cases, the stack input logic produced the correct operations.

F.1.2 Testing Operational Logic

After thoroughly testing and checking the input logic, the operational logic is tested. Similar to the test cases for the competitive input logic, the test cases are divided by operation, **Push**, **Pop**, and **NoOp**. Each test case is tested separately using randomly generated numbers. For each operation, the test cases are divided once again based on the DNNC state. It does, however, play a part in deciding the behaviour of the **Pop** operation. This division of test cases is done to ensure the highest coverage of possible operation and DNNC state combinations. The test cases for the operational logic are derived from equations 10.4 and 10.5. We use the following test cases.

1. **Push**:

   (a) When $Count_t \geq 0$ and $FalsePopCount_t \geq 0$.

2. **Pop**:

   (a) When $Count_t > 0$ and $FalsePopCount_t \geq 0$.

   (b) When $Count_t = 0$ and $FalsePopCount_t \geq 0$.

3. **NoOp**:

   (a) When $Count_t \geq 0$ and $FalsePopCount_t \geq 0$.

For each set test case, 20 random instances were generated and used for testing. For all test cases, the results matched the expected outputs.

F.2 The Backward Pass

The test cases for the backward pass are derived from Equations 10.7, 10.8, 10.9 and 10.10 and Tables 10.3, 10.4, 10.5, 10.6, and 10.7. The test cases focus on the values of the resulting gradients. Similarly to forward pass test cases, the ones for the backward pass are divided by operation and then further divided based on the DNNC state. This is done with the aim of achieving the highest possible coverage in the test cases. They are then tested using randomly generated numbers. These test cases are:

1. **Push**:
   (a) When $Count_{in} > 0$.
   (b) When $Count_{in} = 0$.
2. **Pop**:
   (a) When $Count_{in} > 0$.
   (b) When $Count_{in} = 0$.
3. **NoOp**:
   (a) When $Count_{in} > 0$.
   (b) When $Count_{in} = 0$

In a similar manner to the testing of the input logic and operation logic, 20 random instances are generated for each test set. For all test cases, the results matched the expected outputs.

F.3 Designing and Implementing the Unit Tests

Based on the input and operational logic designed for the forward pass, and the backward pass logic defined in sections 10.2.1, 10.2.2 and 10.3, test cases are designed in sections F.1.1, F.1.2, and F.2 to evaluate the behaviour of the DNNC. Unit tests are created to test the DNNC using the test cases to ensure that it behaves according to the defined logic. Each of the defined test cases is tested in a dedicated test bench and randomly generated numbers are used. Sections F.3.1, F.3.2, and F.3.3 describe the unit tests for the input logic, operational logic, and backward pass, respectively. Each logic is implemented in its dedicated unit test and used to evaluate the results of the test cases. In all unit tests, the DNNC module behaved according to the defined logic.

F.3.1 The Input Logic Unit Test

The input logic determines the operation triggered by the DNNC based on the given inputs. To test each of the operations **Push**, **Pop**, and **NoOp** are being triggered correctly, each sub-case is tested in a dedicated unit test. By breaking down a larger unit test into smaller, more focused unit tests, this will provide higher coverage of the possible cases the DNNC can encounter. Random numbers are used to also increase this coverage. The input logic is implemented in the unit test to ensure that the inputs match the condition we are trying to evaluate, and to compare the resulting output to the expected output, which is also generated in the unit test. The thresholds $threshold_{push}$ and $threshold_{pop}$ are both set to 0.5 for all tests. The unit test for the input logic functions as follows:

1. **Generating input values:** The input values *push_input* and *pop_input* are randomly generated, where a value between 0 and 1 is generated for each. If an input for a test case is greater/smaller than its corresponding threshold when it should not be, the value of the threshold is subtracted/added to the input value to ensure the values match the inequalities defined in the input logic.

   For edge cases: If a value is assigned an explicit value such as that of a threshold, then that value is assigned as is and not randomly generated.

2. **Input to the DNNC module:** The generated input values are input to the DNNC module and the resulting output is compared to the expected value.

Where applicable, 20 random instances are generated to test each test case. This included all test cases except test case 4b, which is an edge case not containing any inequalities. The results of running this unit test show that implemented input logic in the DNNC matches the defined in the input logic for all test cases.

F.3.2 The Operational Logic Unit Test

The operational logic is built on the competitive input logic. Once we establish that the input logic of the DNNC is functioning as intended, the operational logic is tested. A unit test for the operation logic is created to implement the test cases defined in section F.1.2. It builds on the unit test defined for the input logic but also observes the state of the DNNC. Again, the logic is implemented in the unit test to ensure that the values match the condition we are trying to test, and to evaluate the resulting output. The expected outputs are generated in the unit test, and compared to the resulting output. The unit test for the operational logic behaves as follows:

1. **Generating Input Values:** The values for inputs *push_input* and *pop_input* are generated randomly, like they are generated for the input logic.

2. **Initialising the DNNC state:** The DNNC state is initialised based on the defined logic. Unless defined otherwise in the test cases, random integers are generated to initialise the DNNC state.

3. **Input to the DNNC module:** The input values are passed to the DNNC. The output is compared to the expected output.

Where applicable, 20 random instances are generated to test each test case. The results of running this test bench show that implemented operation logic in the DNNC matches the defined in the operation logic for all test cases.

F.3.3 The Backward Pass Unit Test

Before testing the backward pass, we make sure that both input and operational logic unit tests are successful. After testing the input and operational logic in the forward pass, and ensuring that they are working correctly, a unit test for the backward pass is created. This unit test builds on the input and operational logic unit tests detailed earlier. The unit test evaluates each operation and condition separately to ensure the highest possible coverage of input values and DNNC states. The expected output values and gradients are created in the unit test in order to evaluate the resulting output.

The logic for the backward pass is implemented in the unit test to achieve this. The backward pass unit test operates as follows:

1. **Generating Input Values:** The values for inputs *push_input* and *pop_input* are generated randomly. Each is assigned a random value between 0 and 1. The inputs are generated in the same way as in the input and operational logic unit tests.

2. **Initialising the DNNC state:** The DNNC state is initialised base on the defined logic. Unless defined otherwise in the test cases, random integers are generated to initialise the DNNC state.

3. **Calculating the loss and gradients:** The loss is calculated using the expected and resulting outputs and propagated backward to calculate the gradients. The resulting gradients are compared to the expected gradients.

For each test case, 20 random instances are generated. The results of the test bench show that the backward pass behaves as it should.

F.4 Implementation, Execution, and Results

Unit tests are implemented in Python for each of the test cases for the input logic, operational logic and backward pass of our stack module. The unit tests have been executed on our DNNC module. 20 random test input instances are generated for each test case, where applicable. If specific inputs are required for a particular test case or edge case, then we make sure to provide the adequate test inputs. All of our unit tests for the input logic, operational logic and backward pass produced 100% accuracy on the test inputs. This gives us confidence moving forward to incorporate our DNNC module in experiments, knowing that the implementation of our DNNC module is correct and consistent with our design.

F.5 Summary

In order to ensure that the DNNC module is working as intended according to the logic, test cases are created. The test cases for the input, operational and backward logic are each tested in dedicated unit tests. These unit tests are designed to cover as many possible DNNC inputs and states as possible. The results of the unit tests show that the DNNC behaves in accordance with the input logic, operation logic and backward logic.

Appendix G

Visualising the Results of the DNNC Experiments

G.1 Standard RNNs Experiment Results Visualisations

We visualise the runs for the plain recurrent models for the Dyck-1 Acceptance task.

G.1.1 Results of Dyck-1 Acceptance with Standard RNNs

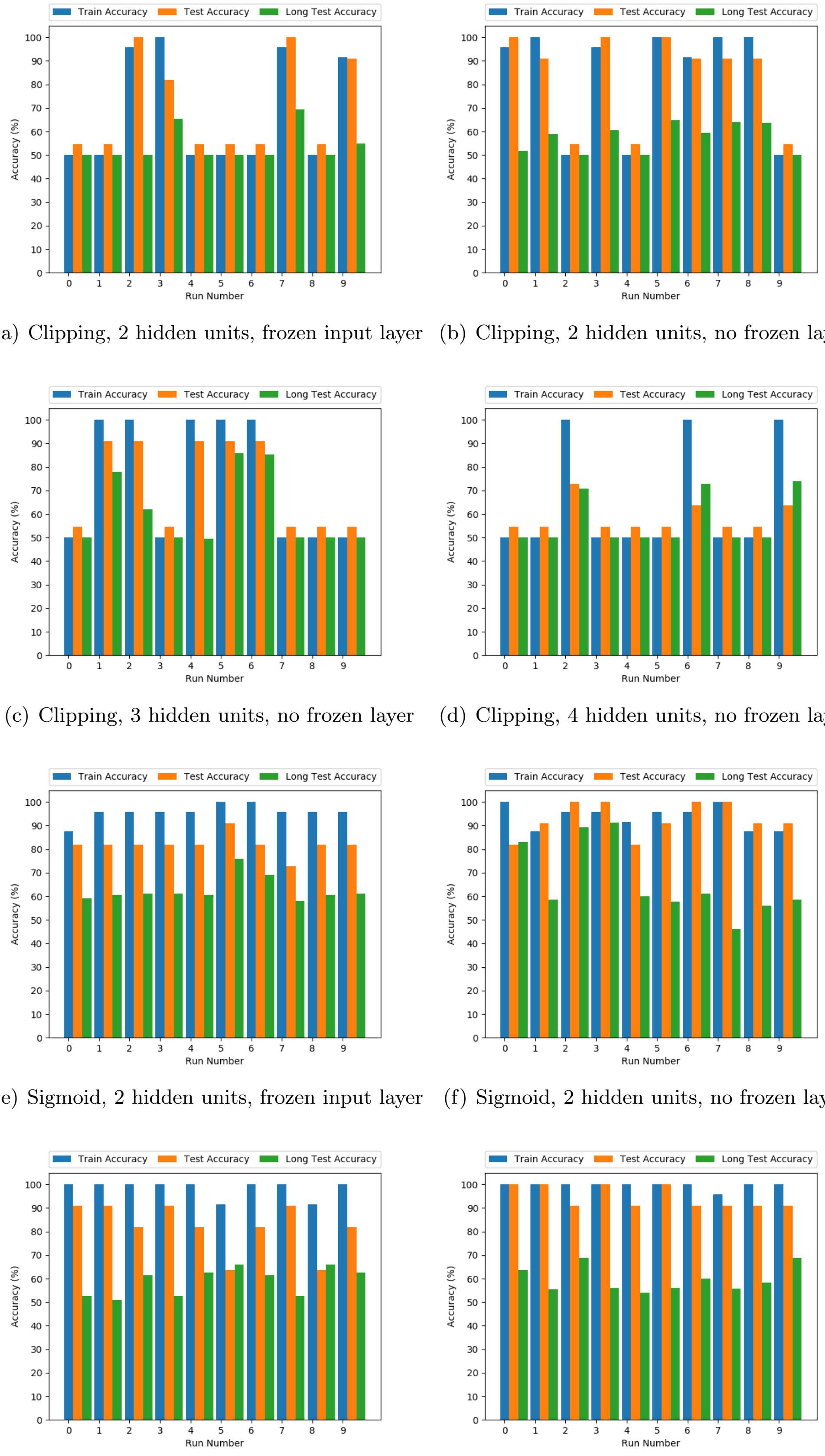
We visualise the results for each of the 10 experiment runs using Standard RNNs, Standard GRUs and Standard LSTMs.

Standard RNNs

Standard RNNs overall did not generalise well to longer sequences. When a sigmoid activation is used, a frozen input layer has minimal effect on training but reduces the test accuracy. When a clipping activation function is used, the freezing of the input layer results in lower accuracy for train and test sets than its fully trainable equivalent. The visualisations are shown in Figure G.1.

Standard GRUs

The Standard GRUs with a sigmoid activation for the most part are more consistent with their train and test accuracies than when a clipping activation is used. Generalisation to



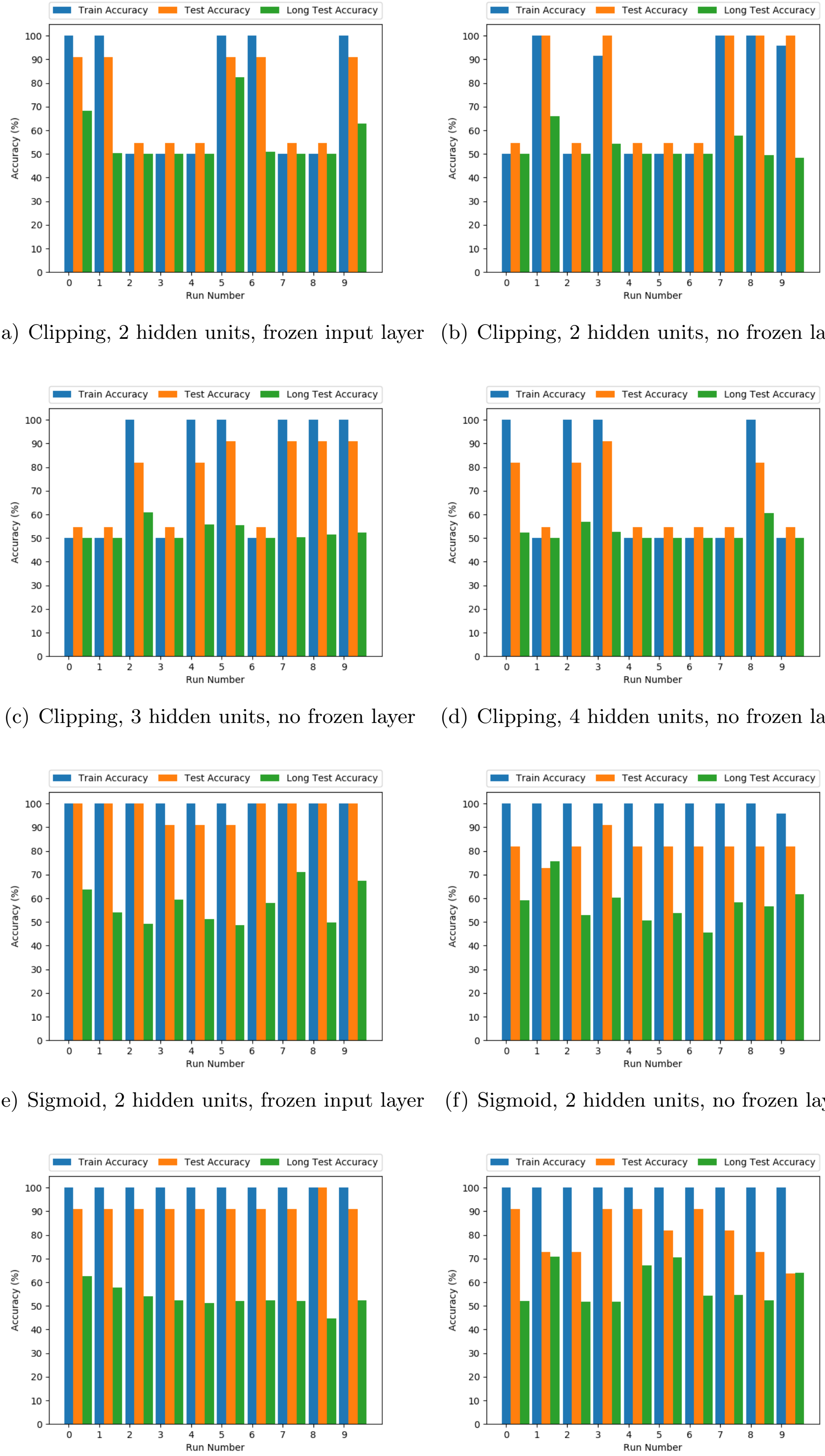
(a) Clipping, 2 hidden units, frozen input layer (b) Clipping, 2 hidden units, no frozen layer

(c) Clipping, 3 hidden units, no frozen layer (d) Clipping, 4 hidden units, no frozen layer

(e) Sigmoid, 2 hidden units, frozen input layer (f) Sigmoid, 2 hidden units, no frozen layer

(g) Sigmoid, 3 hidden units, no frozen layer (h) Sigmoid, 4 hidden units, no frozen layer

Figure G.1: The results across the 10 runs of Dyck-1 Acceptance with Standard RNNs using both Clipping and Sigmoid Activation Functions



(a) Clipping, 2 hidden units, frozen input layer

(b) Clipping, 2 hidden units, no frozen layer

(c) Clipping, 3 hidden units, no frozen layer

(d) Clipping, 4 hidden units, no frozen layer

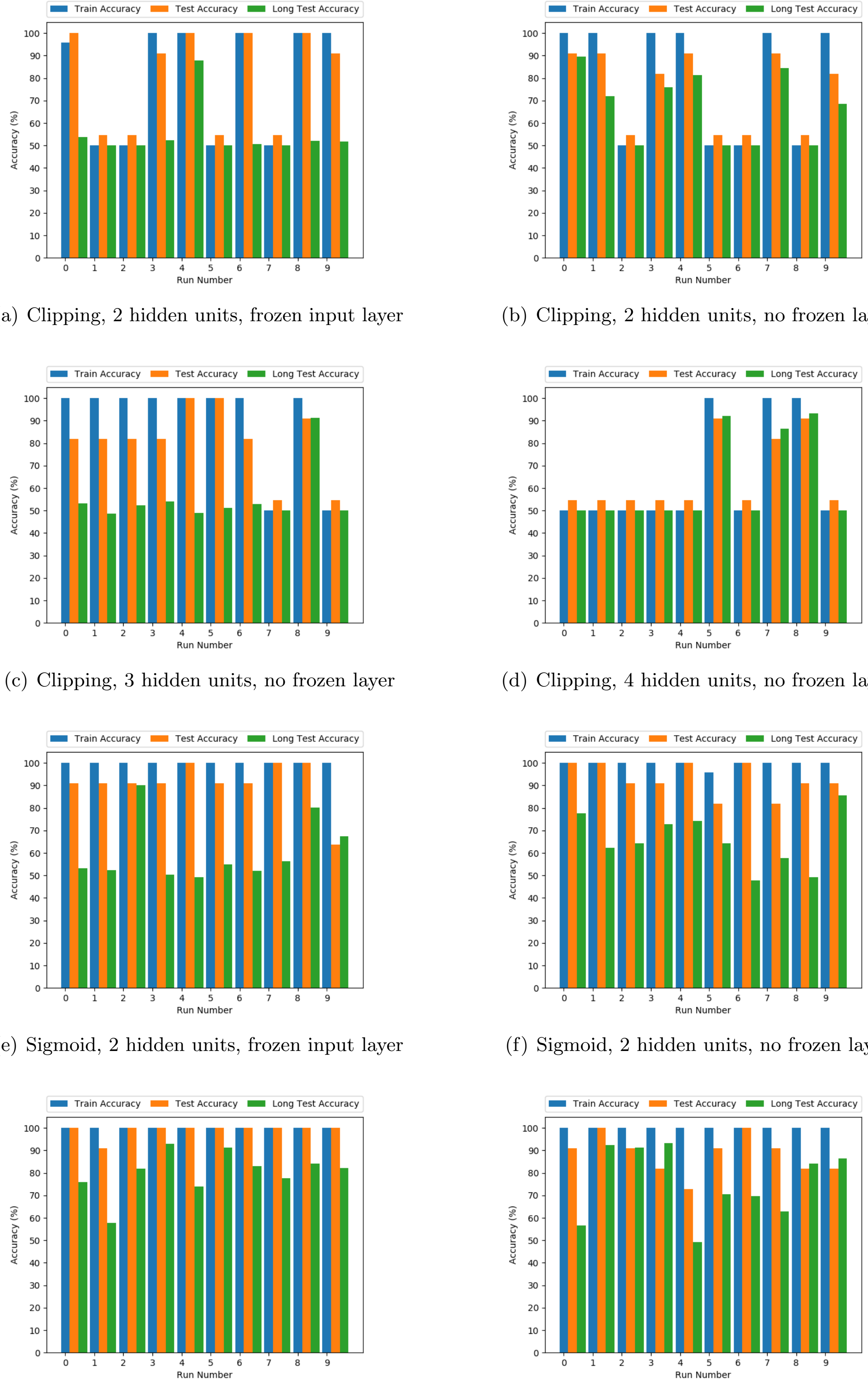
(e) Sigmoid, 2 hidden units, frozen input layer

(f) Sigmoid, 2 hidden units, no frozen layer

(g) Sigmoid, 3 hidden units, no frozen layer

(h) Sigmoid, 4 hidden units, no frozen layer

Figure G.2: The results across the 10 runs of Dyck-1 Acceptance with Standard GRUs using both Clipping and Sigmoid Activation Functions



(a) Clipping, 2 hidden units, frozen input layer

(b) Clipping, 2 hidden units, no frozen layer

(c) Clipping, 3 hidden units, no frozen layer

(d) Clipping, 4 hidden units, no frozen layer

(e) Sigmoid, 2 hidden units, frozen input layer

(f) Sigmoid, 2 hidden units, no frozen layer

(g) Sigmoid, 3 hidden units, no frozen layer

(h) Sigmoid, 4 hidden units, no frozen layer

Figure G.3: The results across the 10 runs of Dyck-1 Acceptance with Standard LSTMs using both Clipping and Sigmoid Activation Functions

longer sequences is not high. When a clipping activation function is used, freezing the input layer slightly improves the test accuracy but can slightly improve generalisation in some runs. The visualisations are shown in Figure G.2.

Standard LSTMs

Sigmoid activation results in higher accuracy across the board with Standard LSTM models. Train accuracy always converges to 100% when a sigmoid activation is used. The overall best performing Standard LSTM model uses a sigmoid activation and 3 hidden units. The visualisations are shown in Figure G.3.

G.2 DNNC Models Experiment Results Visualisations

We visualise the experiment runs for the models that use the symbolic DNNC for Dyck-1 Acceptance.

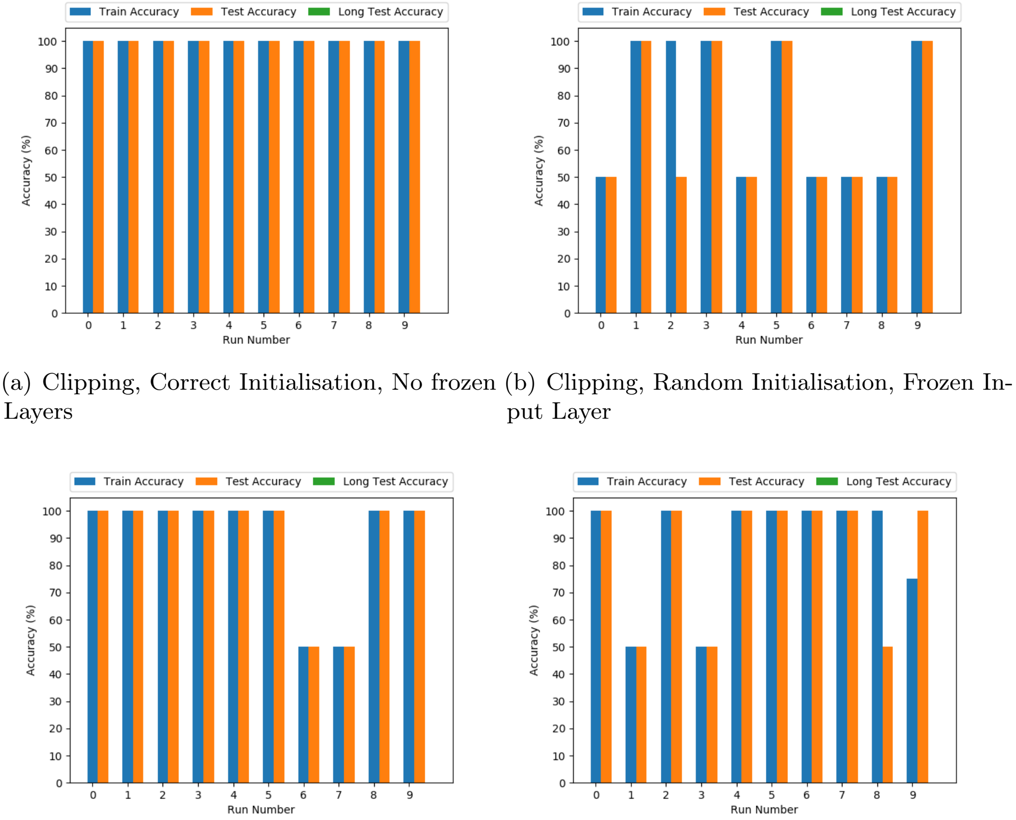
G.2.1 Results of Dyck-1 Acceptance using the DNNC as a Hidden Layer in a Feed-Forward Network

When a clipping activation function is used, no unlearning happens when the correct weights are initialised. It seems that the model has more difficulty learning the output layer than it does the input layer. When randomly initialised, the model more often than not can learn the correct input and output weights. The visualisations of these results are shown in Figure G.4.

On the other hand, when a sigmoid activation function is used, the model has more difficulty learning the input layer from data. When the input layer is frozen, the correct output layer weights can be learned from the data. When the correct initialisation is used, the model does not unlearn the correct weights under any condition. The visualisations of these results are shown in Figure **??**.

G.2.2 Results of Dyck-1 Acceptance using the DNNC as a Recurrent Hidden Layer

When a clipping activation function is used with correct initialisations, no unlearning happens. A sigmoid activation causes the model to unlearn the correct weights if no layers are frozen, however, if the output layer is frozen, training accuracy does not converge to 100% but short and long test accuracies are 100%. The models which use a clipping activation and no bias learn the input layer more easily than the output layer. Models that use a sigmoid activation and no bias learn the output layer more easily. It seems that biases sometimes help the models with a sigmoid activation learn the correct weights. On the other hand, biases seem to hinder the learning of the models when a clipping activation is used. The results for models which use a clipping activation function and no biases are shown in Figure G.6, the results for models using a sigmoid



(a) Clipping, Correct Initialisation, No frozen Layers

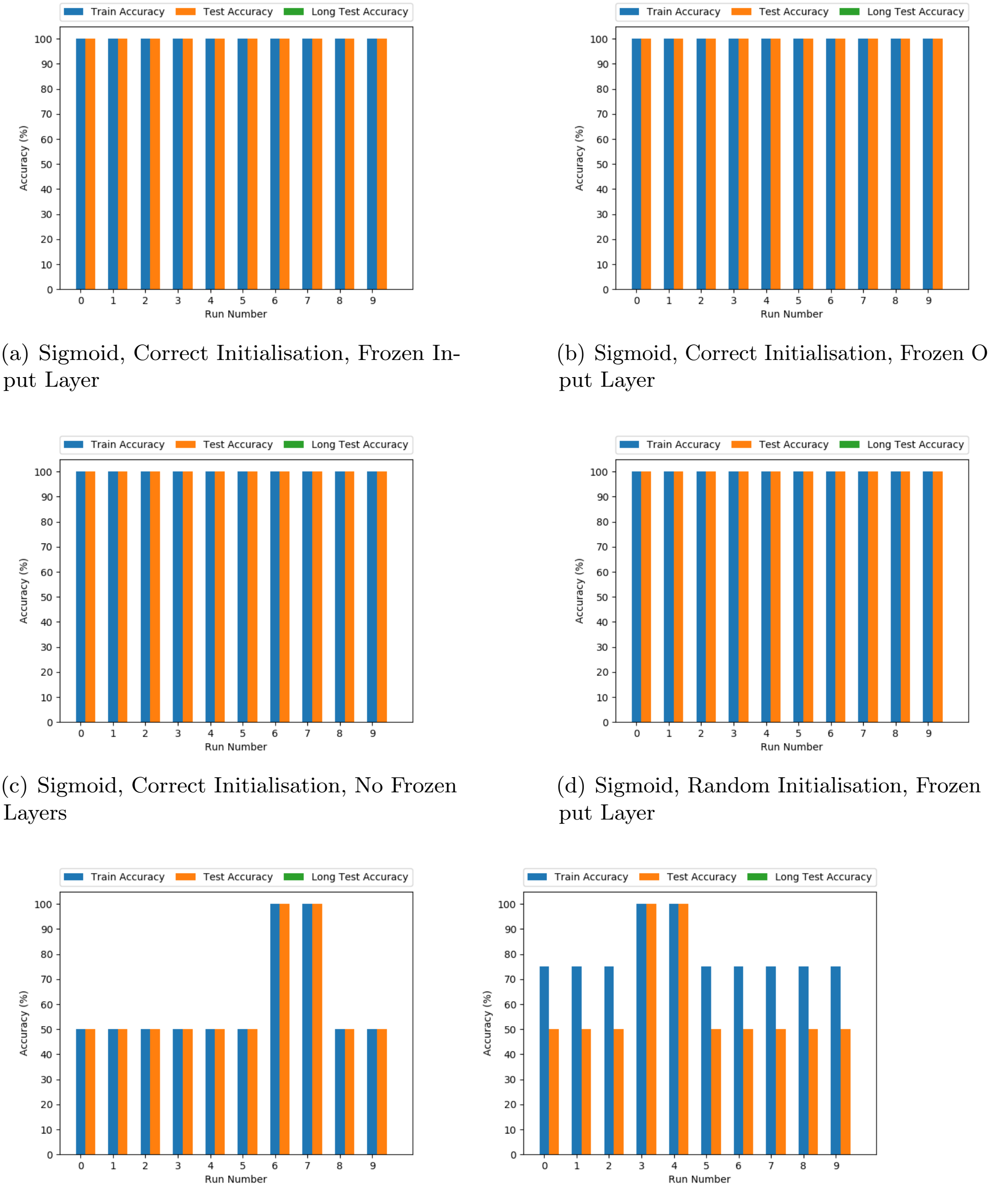
(b) Clipping, Random Initialisation, Frozen Input Layer

(c) Clipping, Random Initialisation, Frozen Output Layer

(d) Clipping, Random Initialisation, No Frozen Layers

Figure G.4: The results across 10 runs of Dyck-1 Acceptance with the DNNC as a hidden layer in a Feed-Forward Network, using a Clipping Activation Function

activation function and no biases are shown in Figure G.7, and the results of models that use biases are shown in Figure G.8.



(a) Sigmoid, Correct Initialisation, Frozen Input Layer

(b) Sigmoid, Correct Initialisation, Frozen Output Layer

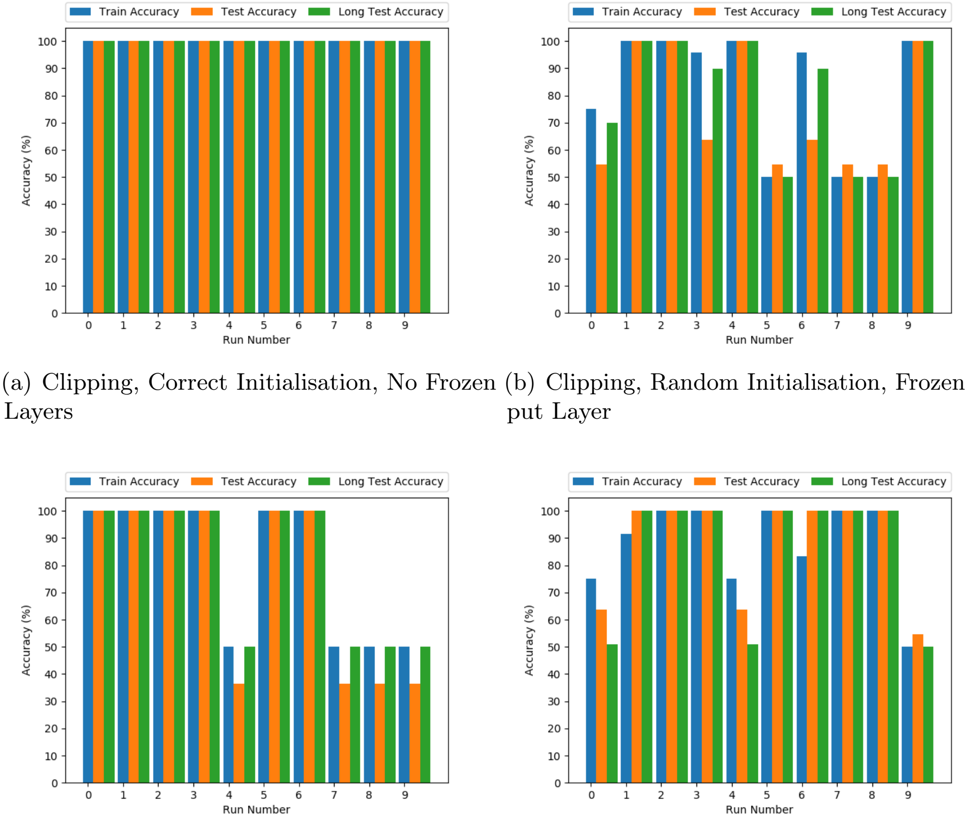
(c) Sigmoid, Correct Initialisation, No Frozen Layers

(d) Sigmoid, Random Initialisation, Frozen Input Layer

(e) Sigmoid, Random Initialisation, Frozen Output Layer

(f) Sigmoid, Random Initialisation, No Frozen Layers

Figure G.5: The results across 10 runs of Dyck-1 Acceptance with the DNNC as a hidden layer in a Feed-Forward Network, using a Sigmoid Activation Function



(a) Clipping, Correct Initialisation, No Frozen Layers

(b) Clipping, Random Initialisation, Frozen Input Layer

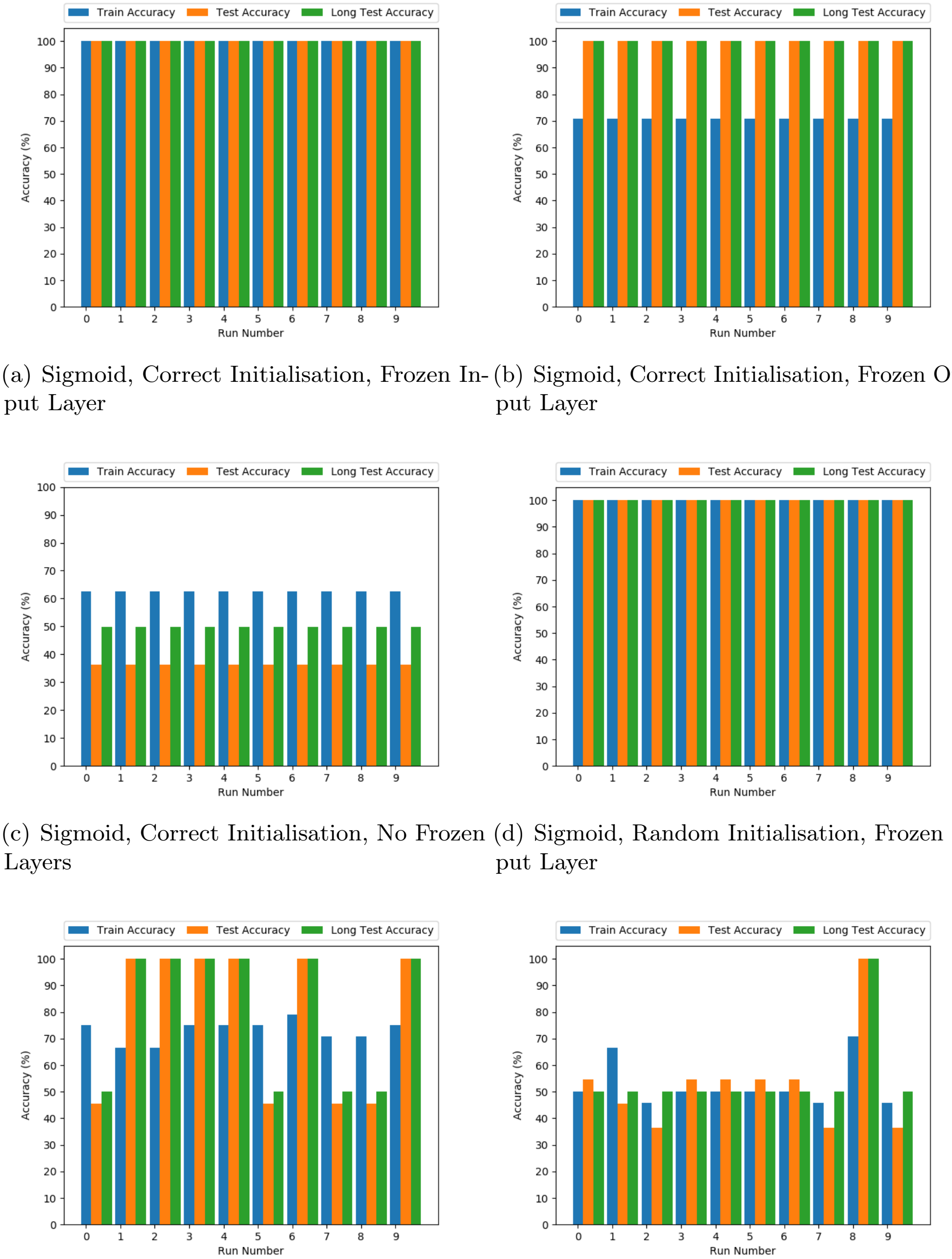
(c) Clipping, Random Initialisation, Frozen Output Layer

(d) Clipping, Random Initialisation, No Frozen Layers

Figure G.6: The results across 10 runs of Dyck-1 Acceptance with the DNNC as a recurrent hidden layer, using a Clipping Activation Function. Biases are not used here.

G.3 Results of Dyck-1 Acceptance using DNNC Augmented Recurrent Networks (Mixed Models)

For these mixed models, the sigmoid activation function produces higher train, test and long test accuracies when compared to the clipping equivalents. Out of the mixed models tested, the best generalisation overall is achieved by a Stack LSTM with a sigmoid activation function and 3 hidden units. Stack GRU models with sigmoid activation functions almost always converge to 100% training accuracy. The results for the Stack RNN, Stack GRU and Stack LSTM models can be found in Figures G.9, G.10, and G.11, respectively.



(a) Sigmoid, Correct Initialisation, Frozen Input Layer

(b) Sigmoid, Correct Initialisation, Frozen Output Layer

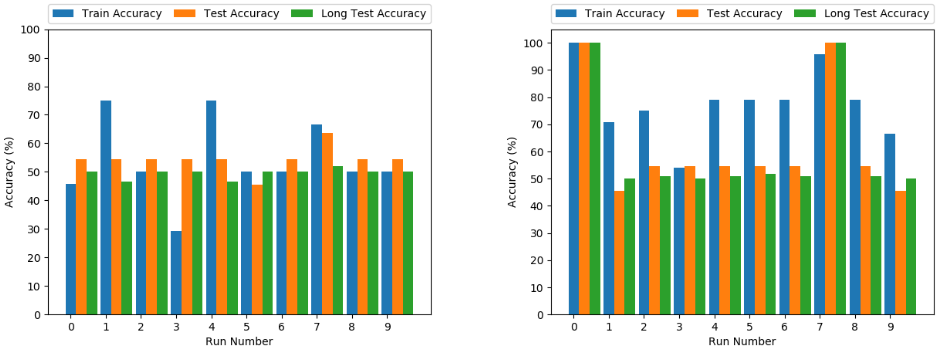
(c) Sigmoid, Correct Initialisation, No Frozen Layers

(d) Sigmoid, Random Initialisation, Frozen Input Layer

(e) Sigmoid, Random Initialisation, Frozen Output Layer

(f) Sigmoid, Random Initialisation, No Frozen Layers

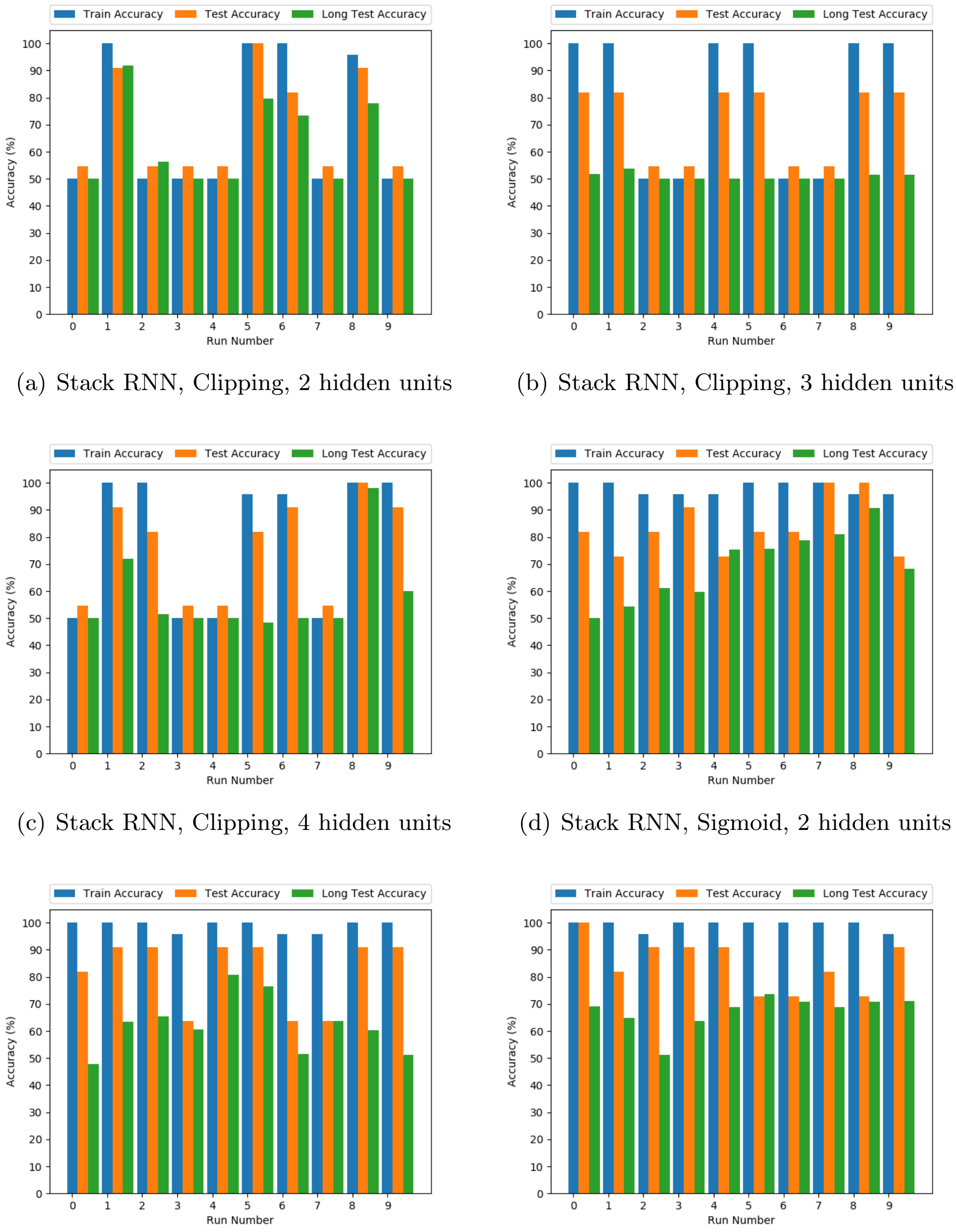
Figure G.7: The results across 10 runs of Dyck-1 Acceptance with the DNNC as a recurrent hidden layer, using a Sigmoid Activation Function. Biases are not used here.



(a) Clipping, Random Initialisation, No Frozen Layers, With Bias

(b) Sigmoid, Random Initialisation, No Frozen Layers, With Bias

Figure G.8: The results across 10 runs of Dyck-1 Acceptance with the DNNC as a recurrent hidden layer, using both Sigmoid and Clipping Activation Functions. Biases are used here.



(a) Stack RNN, Clipping, 2 hidden units

(b) Stack RNN, Clipping, 3 hidden units

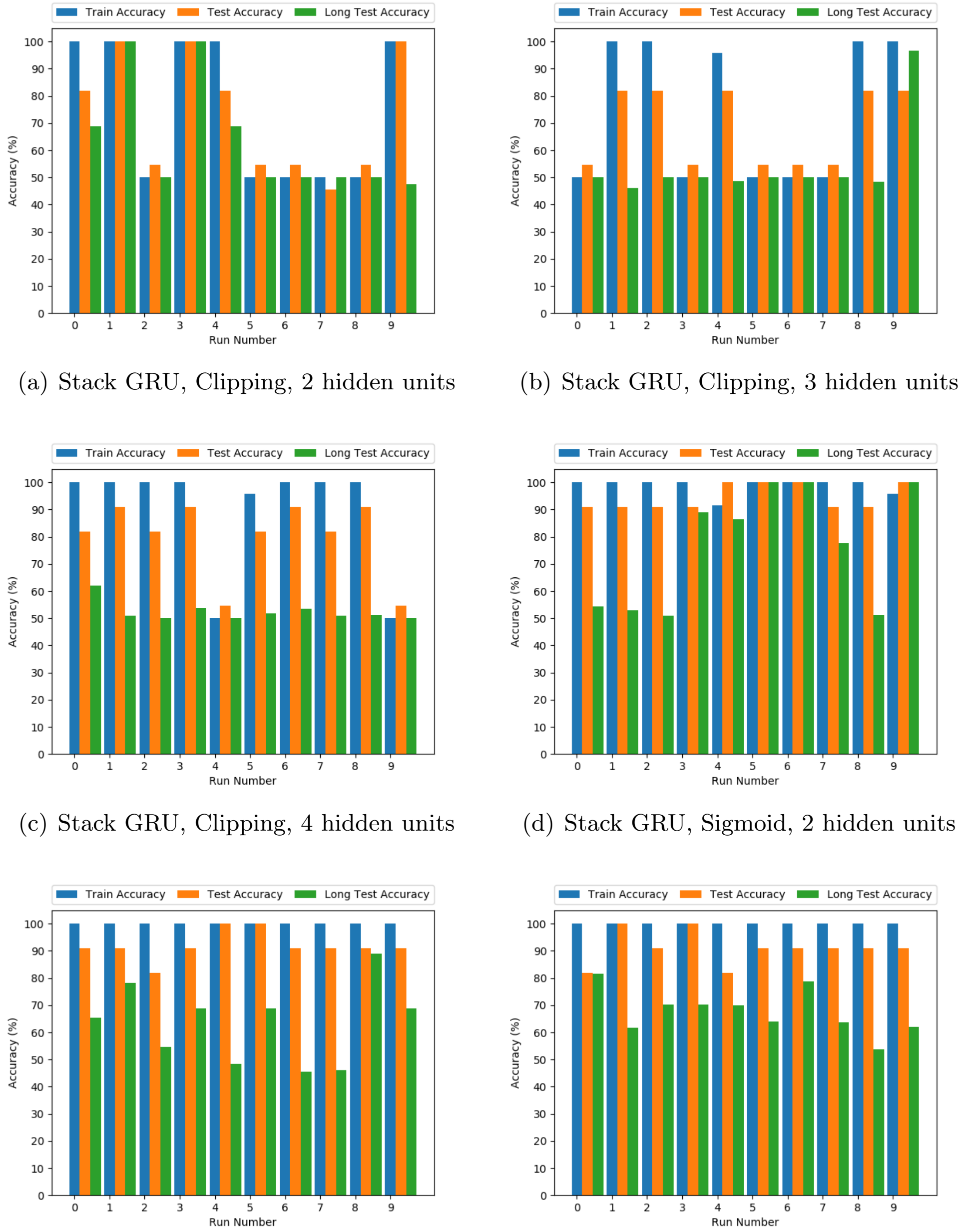
(c) Stack RNN, Clipping, 4 hidden units

(d) Stack RNN, Sigmoid, 2 hidden units

(e) Stack RNN, Sigmoid, 3 hidden units

(f) Stack RNN, Sigmoid, 4 hidden units

Figure G.9: The results across 10 runs of Dyck-1 Acceptance with the DNNC as part of a recurrent hidden layer alongside a standard RNN network, using both Sigmoid and Clipping Activation Functions.



(a) Stack GRU, Clipping, 2 hidden units

(b) Stack GRU, Clipping, 3 hidden units

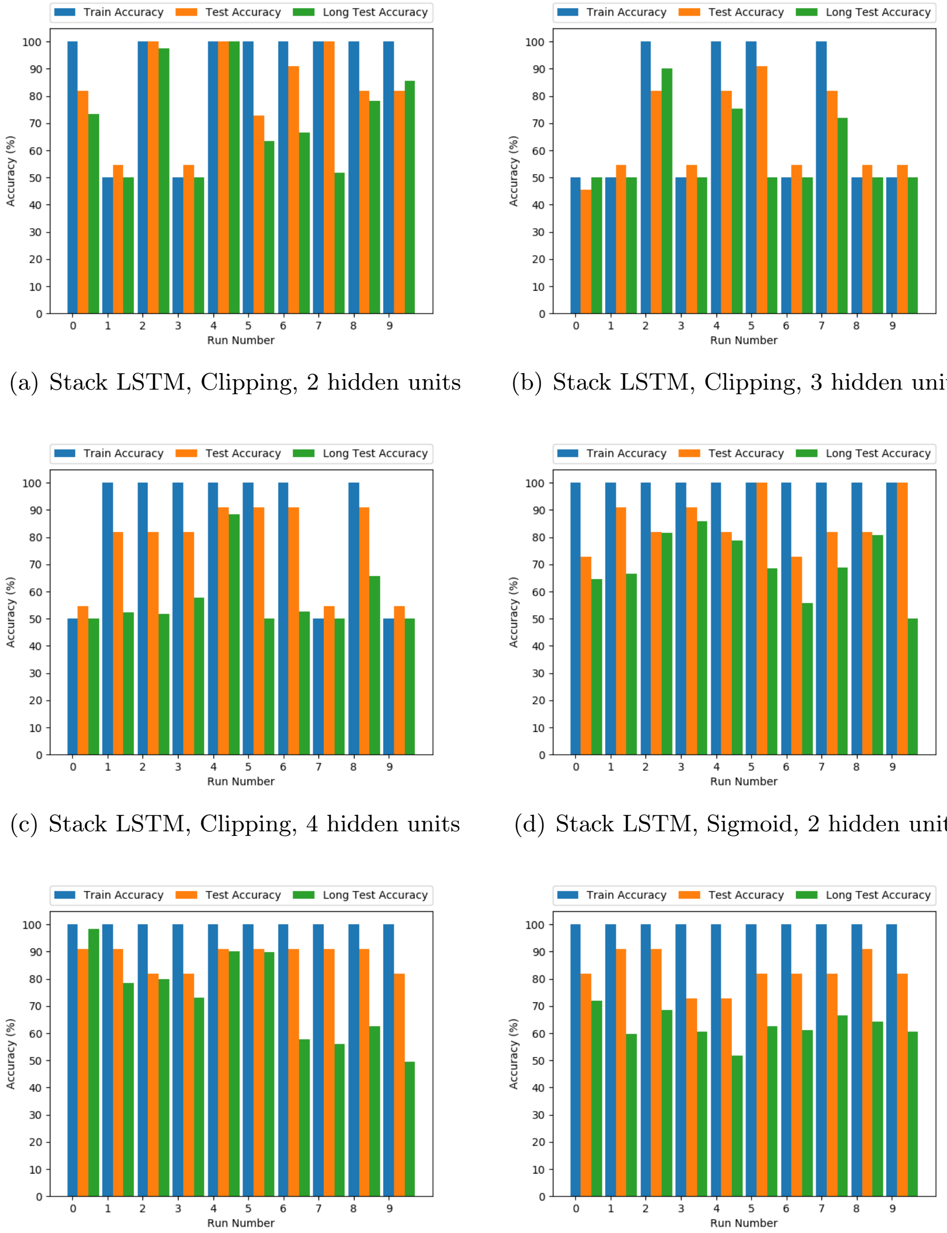
(c) Stack GRU, Clipping, 4 hidden units

(d) Stack GRU, Sigmoid, 2 hidden units

(e) Stack GRU, Sigmoid, 3 hidden units

(f) Stack GRU, Sigmoid, 4 hidden units

Figure G.10: The results across 10 runs of Dyck-1 Acceptance with the DNNC as part of a recurrent hidden layer alongside a standard GRU network, using both Sigmoid and Clipping Activation Functions.



(a) Stack LSTM, Clipping, 2 hidden units

(b) Stack LSTM, Clipping, 3 hidden units

(c) Stack LSTM, Clipping, 4 hidden units

(d) Stack LSTM, Sigmoid, 2 hidden units

(e) Stack LSTM, Sigmoid, 3 hidden units

(f) Stack LSTM, Sigmoid, 4 hidden units

Figure G.11: The results across 10 runs of Dyck-1 Acceptance with the DNNC as part of a recurrent hidden layer alongside a standard LSTM network, using both Sigmoid and Clipping Activation Functions.