Exploring Generalization Capabilities of Dynamic Neural Expansion

Project Report

Submitted by:

Arpitsinh Vaghela (AU1841034)

in partial fulfillment for the award of the degree

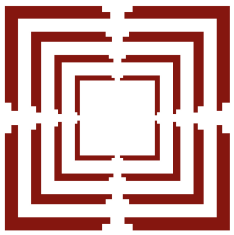
of

BACHELOR OF TECHNOLOGY

IN

INFORMATION AND COMMUNICATION TECHNOLOGY (ICT)

at



Ahmedabad University

School of Engineering and Applied Sciences(SEAS)

Ahmedabad, Gujarat, India

May, 2022

Declaration

I hereby declare that project entitled "***Exploring Generalization Capabilities of Dynamic Neural Expansion***" submitted for the B. Tech. (ICT) degree is my original work and the project has not formed the basis for the award of any other degree, diploma, fellowship or any other similar titles.

Arpitsinh Vaghela
Ahmedabad, Gujarat, India
22 May, 2022

Certificate

This is to certify that the project titled ***"Exploring Generalization Capabilities of Dynamic Neural Expansion"*** is the bona fide work carried out by ***Arpitsinh Vaghela***, a student of B Tech (ICT) of School of Engineering and Applied Sciences at Ahmedabad University during the academic year 2021-2022, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (Information and Communication Technology) and that the project has not formed the basis for the award previously of any other degree, diploma, fellowship or any other similar title.

23-05-2022

Prof. Mehul Raval
Associate Dean - Experiential Learning and Professor
School of Engineering and Applied Science, Ahmedabad University
Ahmedabad, Gujarat, India
23 May, 2022

Abstract

Deep learning is one of the fastest-growing topics with recent successes such as DALL-E (Text to Image Generation), GPT-3 (Language Modeling), and Yolov5 (Object Detection). However, in real-world workflows, one often has to train many different neural networks to reach an optimal model during the experimentation and design process. This process is tedious and time-consuming as each new model is trained from scratch. Searching for the best model for a given task is an active research area with diverse approaches such as neural architecture search, pruning, and evolutionary algorithms. These approaches are costly, requiring large search spaces or large architectures to start.

Another approach can be to start with a small network and learn an efficient architecture by incrementally increasing the size of a model. Expanding the size of a model has been used in settings such as continual learning, architecture search, optimization, and reinforcement learning. This work aims at exploring the generalization capabilities of this method.

Acknowledgement

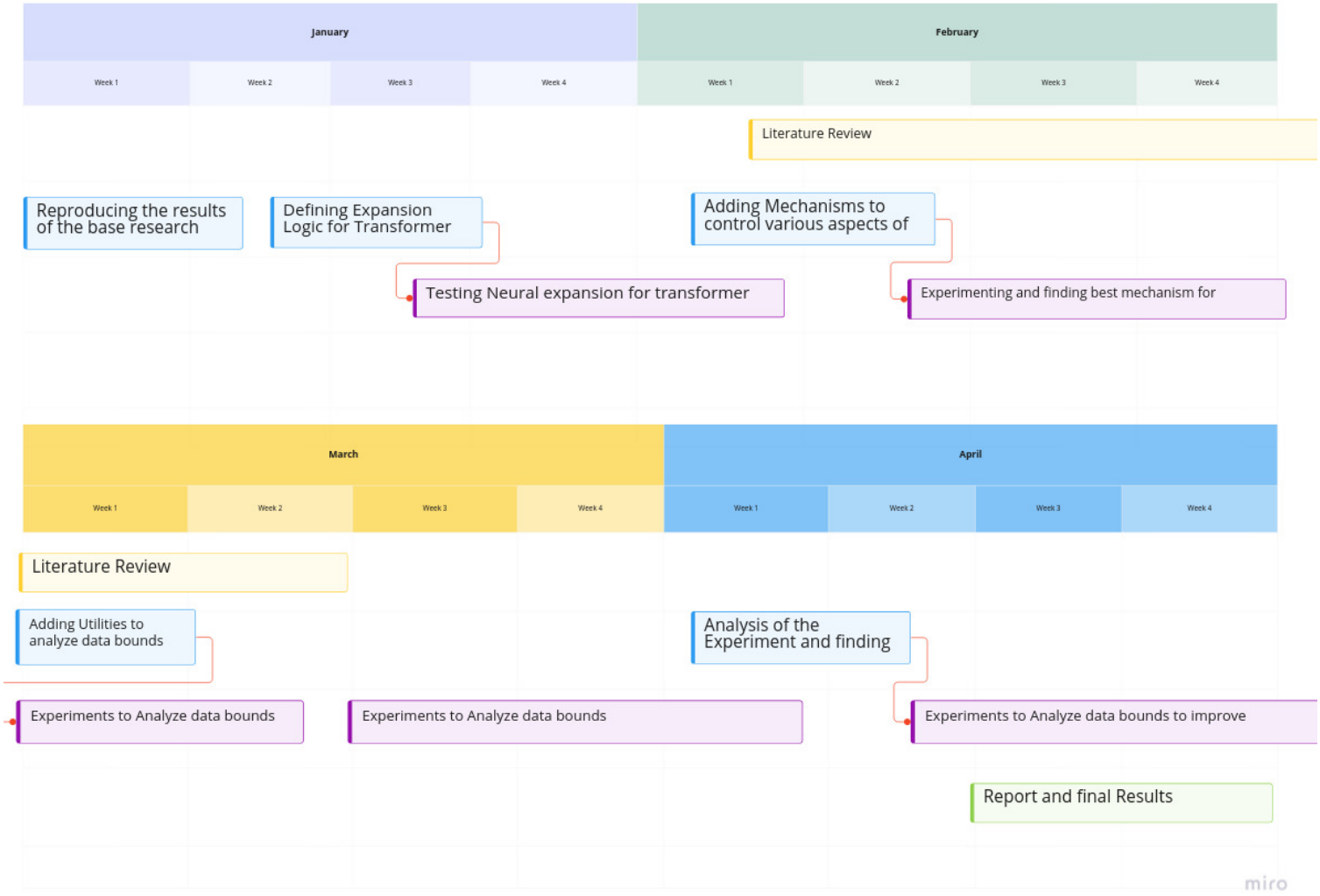
I want to thank **Prof. Mehul Raval** for his invaluable guidance throughout the project. I would like to thank my late grandfather for ensuring I get the best education enabling me to present this project today.

Contents

List of Figures

List of Tables

Ghantt Chart



Introduction

Deep learning is one of the fastest-growing topics with recent successes such as DALL-E (Text to Image Generation), GPT-3 (Language Modeling), and Yolov5 (Object Detection). However, in real-world workflows, one often has to train many different neural networks to reach an optimal model during the experimentation and design process. This process is tedious and time-consuming as each new model is trained from scratch. Searching for the best model for a given task is an active research area with diverse approaches such as neural architecture search, pruning, and evolutionary algorithms. These approaches are costly, requiring large search spaces or large architectures to start.

Another approach can be to start with a small network and learn an efficient architecture by incrementally increasing the size of a model. Expanding the size of a model has been used in settings such as continual learning, architecture search, optimization, and reinforcement learning. This work aims at exploring the generalization capabilities of this method.

1.1 Problem Definition

The generalization capabilities of neural networks have long been a source of interest to the machine learning community since it defies intuitions derived from classical learning theory. Several papers have attempted to understand the generalization phenomenon in deep learning models from a theoretical perspective. However, current bounds are still not tight enough to capture the generalization behavior accurately. Hence we try to reason about it through experimentation and observation. In this work, we study the generalization capabilities of neural expansion, i.e., incrementally increasing the model's size on small algorithmically generated datasets. In this setting, we can study questions about data efficiency, memorization, generalization, and learning speed in great detail.

1.2 Problem Overview

To understand the impact of network expansion on generalization, we train a model on small algorithmically generated datasets and transfer its weights to the following network, which is again trained on the same dataset. We conduct this experiment over various datasets with different train-test splits, i.e., to understand its data efficiency across various hyperparameters. We base our study on a similar study carried out in [1], providing us with a benchmark to compare against.

1.2.1 Algorithmically Generated Datasets

We consider datasets of binary operation of the form $a \circ b = c$ where a, b, c are discrete symbols and have no information about its internal structure, and $\circ$ represents a binary operation. Some of the binary operations considered are addition, composition of permutations, and bivariate polynomials. A Neural Network trained on a proper subset of all possible equations can fill the blanks in the binary operation table. An example can be seen in Figure 1.1. Since we use distinct abstract symbols for all distinct elements a, b, and c involved in the equations, the network is not aware of any internal structure of the elements. It has to learn about their properties only from interactions with other elements. For example, the network doesn't see numbers as a decimal or permutations notations.

★	a	b	c	d	e
a	a	d	?	c	d
b	c	d	d	a	c
c	?	e	d	b	d
d	a	?	?	b	c
e	b	b	c	?	a

Figure 1.1: An example of a small binary operation table. Image from [1]

The following are the binary operations that we have tried (for a prime number $p = 97$):

$$
\begin{aligned}
&x \circ y = x + y \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x - y \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x/y \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x^2 + y^2 \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x^2 + x.y + y^2 \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x^2 + x.y + y^2 + x \pmod{p} && \text{for } 0 \leq x, y < p \\
&x \circ y = x \cdot y && \text{for } x, y \in S_5
\end{aligned}
$$

We create a dataset of equations of the form $\langle x \rangle$ $\langle op \rangle$ $\langle y \rangle$ $\langle = \rangle$ $\langle x \circ y \rangle$ for each binary operation, where $\langle a \rangle$ stands for the token corresponding to element a. During training, we pick a fraction of available equations at random as the training set, with the rest of the equations as the validation set.

1.2.2 Network Architecture

We use a standard Decoder-Only Transformer with causal attention masking shown in figure 3.1 and calculate loss and accuracy only on the answer part of the equation. We start with a smaller network size that doesn't overfit on the given task. The model parameters are increased for N steps until it has 2 layers, 4 attention heads, and width 128, with a total of about $4 \cdot 10^5$ non-embedding parameters. This allows us to be consistent with [1], enabling us to compare the results obtained by us against their results, acting as a benchmark for this work.

1.2.3 Hardware Specification

Processor	Intel(R) Xeon(R) CPU E3-1245 v5 @ 3.50GHz
GPU	NVIDIA Quadro K5200
GPU Memory	8GB
OS	Ubuntu 21.04

Table 1.1: Hardware Specification of Remote System

1.2.4 Software Specification

Python version	3.9.5
CUDA version	11.3
PyTorch	1.10.1 (built from source)
pytorch-lightning	1.6.1
wandb	0.12.15
Git repository	`https://github.com/arpitvaghela/dyana.git`

Table 1.2: Software Specifications

Literature Survey

Earlier works on Network expansion suggest how to expand the network by splitting or adding. **Splitting** [2] duplicates existing neurons and adjusts outgoing weights so that the output in the next layer is unchanged. Or by **adding** with new weights initialized randomly [3] or using "network morphism" i.e., transformations that leave the function output unchanged as can be seen in [4]. In [5] the authors examine various complexity measures as a generalization measure carrying out a large-scale experiment and suggest that theoretical bounds are unable to express the generalization capabilities of Neural networks better than empirical measures.

In [1] they show that an over-parameterized network can generalize way beyond overfitting. They study this phenomenon on various small algorithmically generated datasets as they try to reason regarding the data efficiency of the phenomenon. In this work, we propose the simplest "network morphism" method for the Transformer Decoder-Only Architecture and explore the generalization capabilities of this method empirically through experiments using the algorithmically generated datasets used in [1].

Methodology

3.1 Transformer Architecture

The Transformer network is a model architecture relying entirely on an attention mechanism to draw global dependencies between input and output. Decoder Only Transformer consists of only decoder part of Transformer which uses stacked self-attention and point-wise, fully connected layers to map an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $z = (z_1, \ldots, z_n)$. Given z, a final feed-forward generates an output sequence $(y_1, \ldots, y_m)$ of symbols.

3.1.1 Decoder

The decoder is made up of a stack of N identical layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a fully connected feed-forward network. We employ a residual connection around each sub-layers, followed by layer normalization. The residual connections, all sub-layers in the model, and the embedding layers produce outputs of dimension d_{model}.

3.1.2 Attention

An attention function maps query and key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is a weighted sum of values, where the weight assigned to each value is a compatibility function of the query with the corresponding key a.shown in Figure 3.2

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{3.1}$$

3.1.3 Multi-Head Attention

In Multi-Head Attention, we project the queries, keys, and values h times using different learned linear projections to d_q, d_k, and d_v dimensions. We then perform

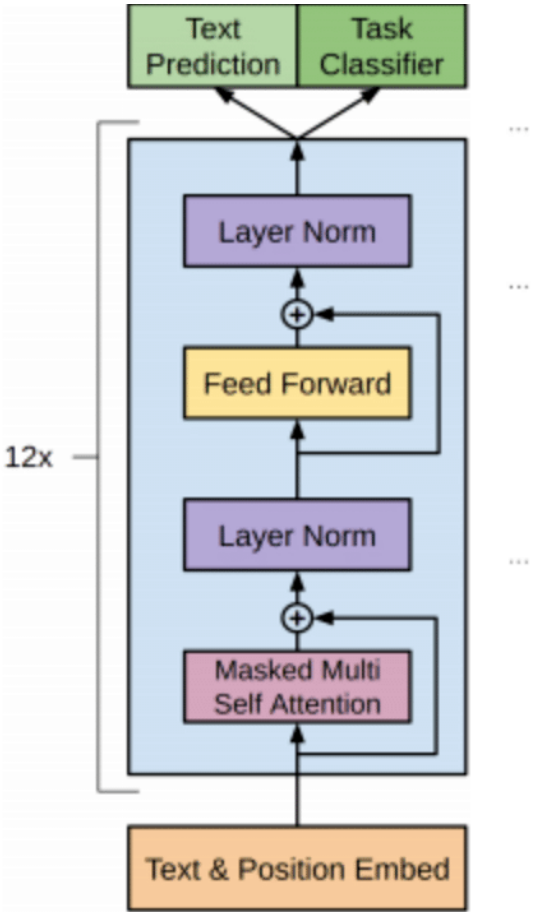


Figure 3.1: Decoder only Transformer Architecture

Source: Image from `https://www.researchgate.net/`

the attention function on these projected versions of queries, keys, and values in parallel, yielding output values of d_v-dimension. These are concatenated and projected again resulting in the final values, as shown in Figure 3.2.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.2}$$

Where the projections are parameter matrices
$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

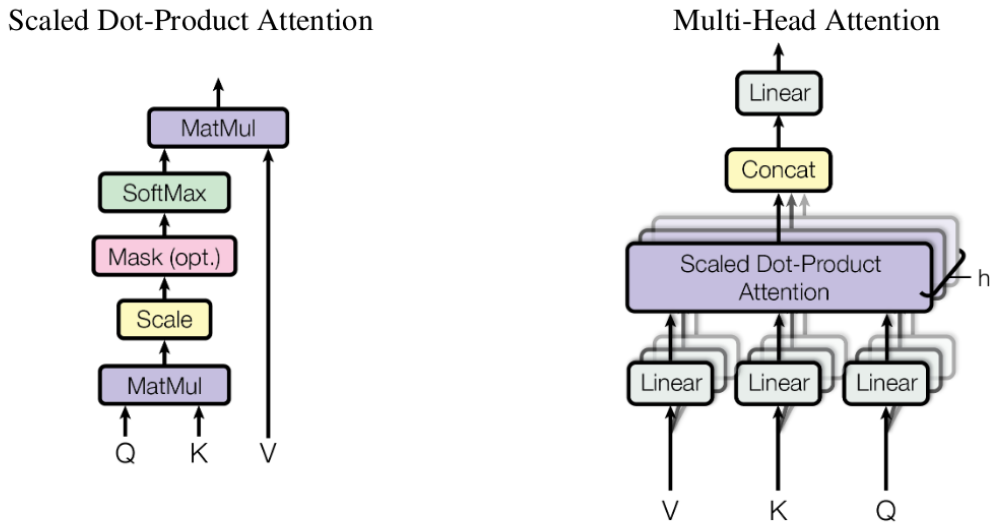


Figure 3.2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Source: Image from [6]

3.2 Expansion in Transformer Architecture

We transfer knowledge contained in one network to another network during training. The technique we used is based on *function preserving transformations* between networks as described in [2]. We initialize the student network to be a neural network representing the same function as the teacher but using a different number of parameters. Therefore, the student network will have the same output as the teacher network for a given input. We expect the student network to be always larger than the teacher network. We propose two methodologies.

Depth Expansion allows replacing a model that satisfies some properties with an equivalent deeper model, and

Width Expansion allows replacing a model with an equivalent model that is wider (has more units in each hidden layer).

We combine these two to get a student network with any desired additional depth and width.

3.2.1 Depth Expansion

When increasing the depth of the student network, we add extra blocks of decoder/s to the network. To ensure that this new network has the same output as that of the teacher network, we change the weights of the Multi-Head Attention and Feed Forward sub-layers to be a $\overrightarrow{0}$. We are ensuring the output of the block

is identical to the input after the residual connections.

3.2.2 Width Expansion

When increasing the width of the student network, we change the d_{model} of the network. Moreover, the parameters of the Multi-Head Attention sub-layer are influenced by the n_{head}, i.e., the number of heads in Multi-Head Attention. When increasing the width d_{model} of the network, we expect the output of the network to be an identical vector with zeros in the added dimensions.

In **Feed Forward Sub-layer** the dimension of the weight matrix changes from $(d_{model} * d_{model})$ to $(d_{model} + d_n * d_{model} + d_n)$, where the $d_{model_new} = d_{model} + d_n$. We update $W_{student}[d_{model} * d_{model}] = W_{teacher}$ and the rest to zero. This ensures that the output of Feed Forward sub layer is identical with zeros in the added dimensions.

In **Multi-Head Attention Sub-layer**, we update W_O as we did for the feed-forward sub layer. The output size of the projection weights (W^Q,W^K and W^V),i.e., $d_{key} = d_{model}/n_{head}$ may change with changes in d_{model} and n_{head}. To ensure the output after *concat* remains the same we update the projection weights such that $i * n_{head} + index = i' * n'_{head} + index'$ where i is the head index. If the required value doesn't exists we set the its value to zero in the student network. This ensures that the output of the Multi-Head sub layer is identical with zeros in the added dimensions.

3.3 Model Training and Optimization

We trained a standard decoder-only transformer [6] with causal attention masking and calculated loss and accuracy only on the answer part of the equation. For all experiments, we set the final set of expansion to be a transformer with 2 layers, width 128, and 4 attention heads, with a total of about $4 * 10^5$ non-embedding parameters. For most experiments, we used AdamW optimizer with learning rate 10^3, weight decay 1, $\beta_1 = 0.9$, $\beta_2 = 0.98$, and linear learning rate warm up over the first 10 updates, minibatch size 512 or half of training dataset size (whichever was smaller) and optimization budget of 10^5 gradient updates.

Experiments

4.1 Experiment 1

To experiment if "Neural Expansion leads to better Generalization", we start by training a smaller model on addition dataset with 20% train data split. This model was unable to reach $> 90\%$ validation accuracy after 10^5 training steps but was able to avoid overfitting. We then trained two models, larger than the earlier model. One with random initialization and other with knowledge from the earlier model according to section 3.2.

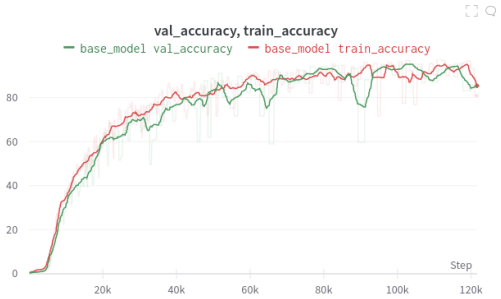


Figure 4.1: Base Model is unable to reach $> 90\%$ after 10^5 optimization steps.

	n_{layers}	d_{model}	n_{head}
Base Model	2	4	4
Model1	2	64	4
Model2	2	64	4

Figure 4.2: Network size of each model in Experiment 1

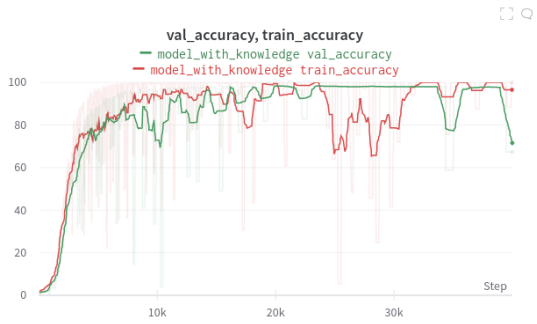


Figure 4.3: Model1 with knowledge from previous training generalizes better. It reaches a validation accuracy of $> 98\%$ within $20,000$ steps

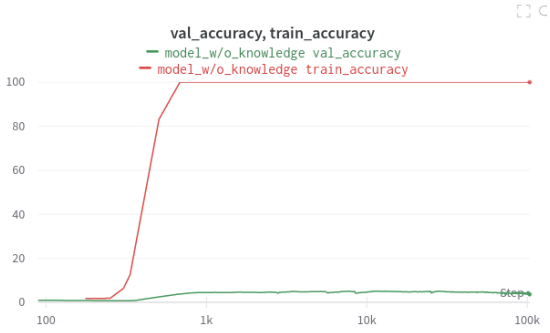


Figure 4.4: Model2 with random initialization tends to overfit on the task.

It was observed that the model with random initialization tends to overfit

within 1000 optimization steps and is unable to recover from the same within the budget of 10^5 optimization steps. Whereas the model initialized with knowledge from the smaller model is able to reach a validation accuracy of $> 98\%$ within $20,000$ steps and is able to prevent overfitting.

4.2 Experiment 2

To find the best optimization strategy over various training data ranges we trained on S_5 binary operation dataset using various optimization strategies and on range of train data percentages. We tried the following variants of optimization strategies to find the best optimization strategy:

- Adam optimizer
- Adam optimizer with full batch
- AdamW optimizer with weight decay 1
- AdamW optimizer with weight decay 1 towards the initialization instead of the origin

We start with a small network with $n_{layers} = 1$, $d_{model} = 8$ and $n_{head} = 2$. This network size was small enough to prevent overfitting even at the lowest train data percentage (10%). This network then undergoes 5 expansion as it trains with a training budget of 10^5 optimization steps distributed among these expansions as described in Table 4.1

Expansion	n_{layers}	d_{model}	n_{head}	Optimization steps
	1	8	2	5000
1	1	16	4	5000
2	2	16	4	5000
3	2	32	4	15000
4	2	64	4	20000
5	2	128	4	50000

Table 4.1: Network hyperparameters and Optimization steps for each expansion in Experiment 2

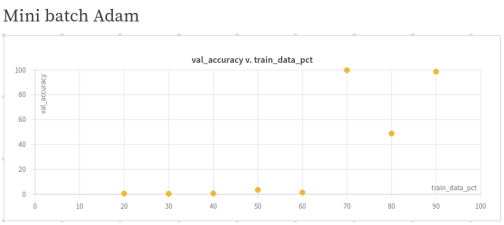


Figure 4.5: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained using expansion method and Adam with mini-batch

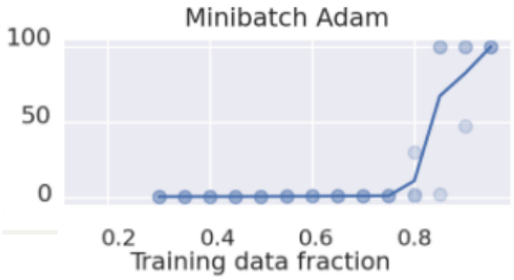


Figure 4.6: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained conventionally using Adam with mini-batch

Source: Image from [1]

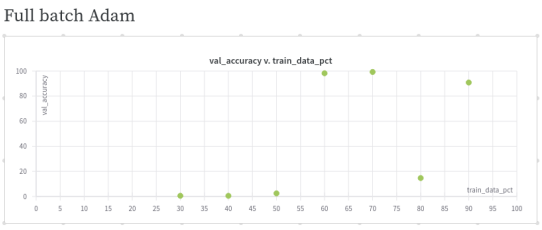


Figure 4.7: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained using expansion method and Adam with full-batch

Figure 4.8: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained conventionally using Adam with full-batch

Source: Image from [1]

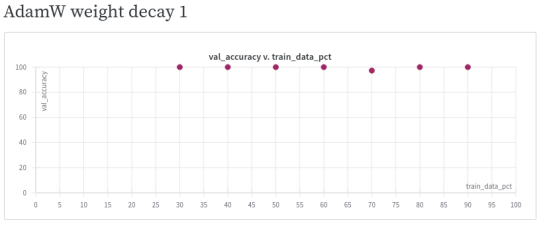


Figure 4.9: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained using expansion method and AdamW with weight decay 1

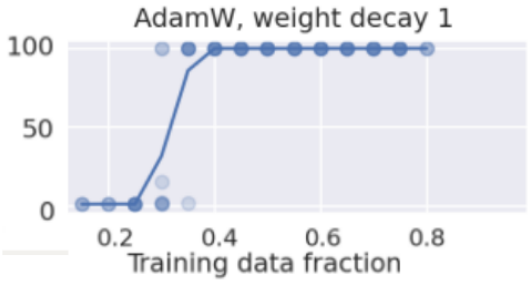


Figure 4.10: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained conventionally using AdamW with weight decay 1

Source: Image from [1]

We observe that this method performs better than [1] when using Adam optimizer with mini-batch size and full batch and are able to reach a validation accuracy $> 99\%$ with 10% lesser training data percent when using mini-batch

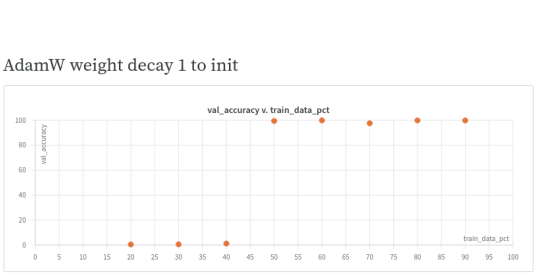


Figure 4.11: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained using expansion method and AdamW with weight decay 1 to initialization

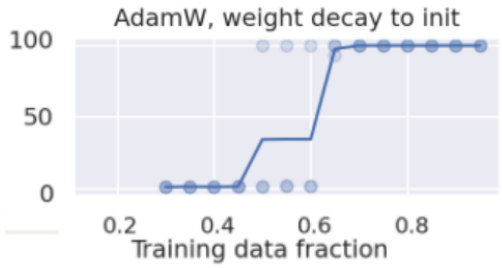


Figure 4.12: Train data percent vs Validation accuracy for binary operation $x \cdot y$ $x, y \in S_5$ trained conventionally using AdamW with weight decay 1 to initialization

Source: Image from [1]

Adam and 20% lesser training data percent when using full-batch Adam. When using AdamW optimizer we performer at par with [1]in both cases, weight decay to initialization and weight decay to zero. We observe the best performance when using AdamW and use the same for the experiments 3 and 4.

4.3 Experiment 3

In this next set of experiments we fixed the optimization function be AdamW with weight decay 1 and train across data range of binary operation datasets. The expansions used for the same are described in Table 4.2. The expansion steps are set such that the student network has twice the number of parameters as that of the teacher network.

Expansion	n_{layers}	d_{model}	n_{head}	Optimization steps
	1	8	2	5000
1	1	16	4	5000
2	2	16	4	5000
3	2	32	4	15000
4	2	64	4	20000
5	2	128	4	50000

Table 4.2: Network hyperparameters and Optimization steps for each expansion for Experiment 3

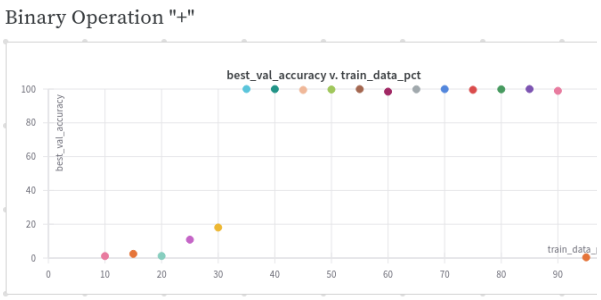


Figure 4.13: Train data percent vs Validation accuracy for binary operation $x+y \pmod{97}$ using expansion method

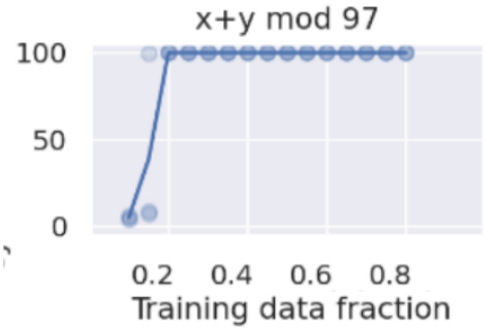


Figure 4.14: Train data percent vs Validation accuracy for binary operation $x+y \pmod{97}$ trained conventionally

Source: Image from [1]

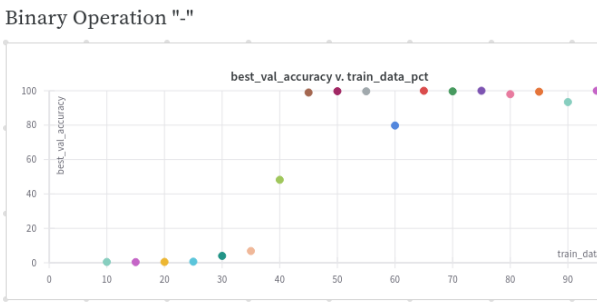


Figure 4.15: Train data percent vs Validation accuracy for binary operation $x-y \pmod{97}$ using expansion method

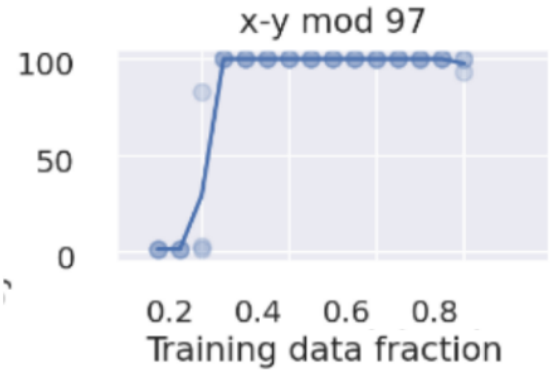


Figure 4.16: Train data percent vs Validation accuracy for binary operation $x-y \pmod{97}$ trained conventionally

Source: Image from [1]

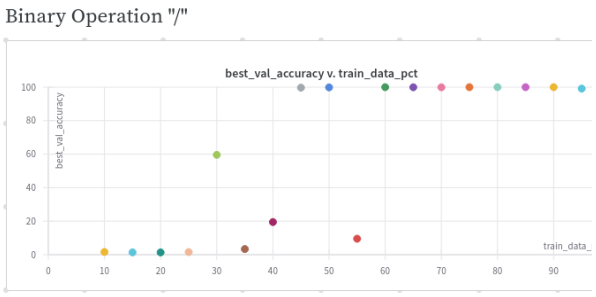


Figure 4.17: Train data percent vs Validation accuracy for binary operation $x/y \pmod{97}$ using expansion method

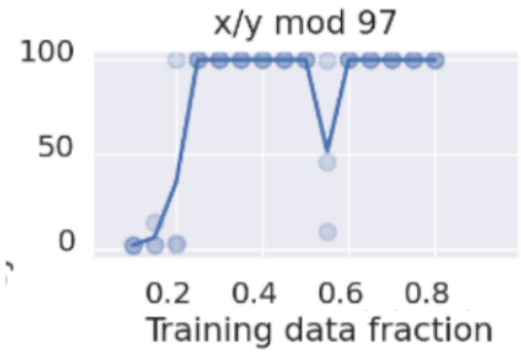


Figure 4.18: Train data percent vs Validation accuracy for binary operation $x/y \pmod{97}$ trained conventionally

Source: Image from [1]

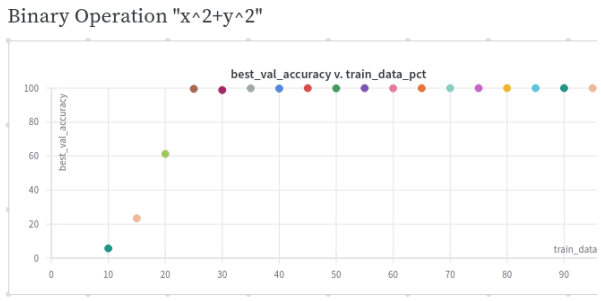


Figure 4.19: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 \pmod{97}$ using expansion method

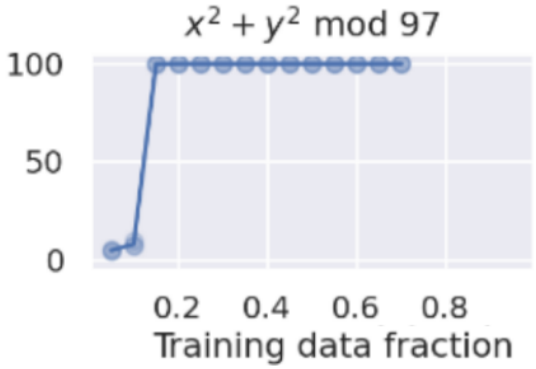


Figure 4.20: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 \pmod{97}$ trained conventionally

Source: Image from [1]

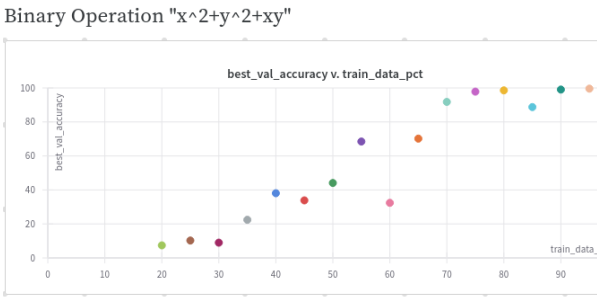


Figure 4.21: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y \pmod{97}$ using expansion method

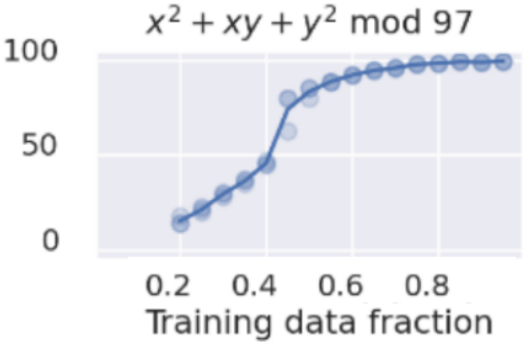


Figure 4.22: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y \pmod{97}$ trained conventionally

Source: Image from [1]

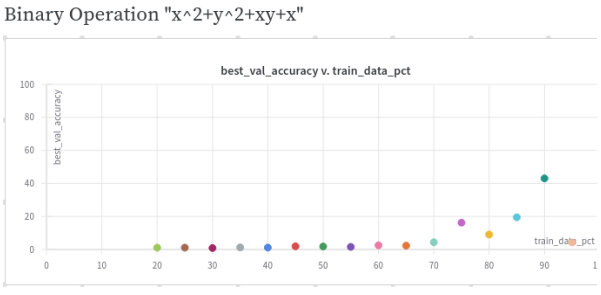


Figure 4.23: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y + x \pmod{97}$ using expansion method

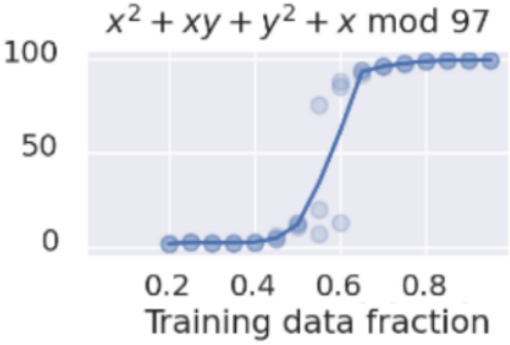


Figure 4.24: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y + x \pmod{97}$ trained conventionally

Source: Image from [1]

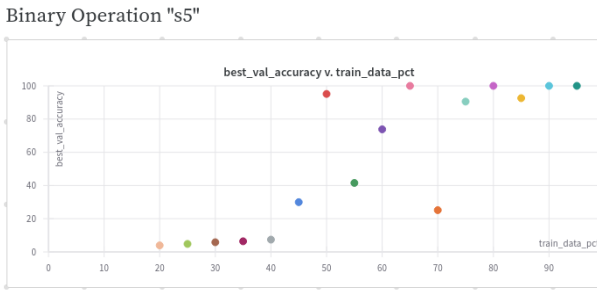


Figure 4.25: Train data percent vs Validation accuracy for binary operation $x \cdot y \; x, y \in S_5$ using expansion method

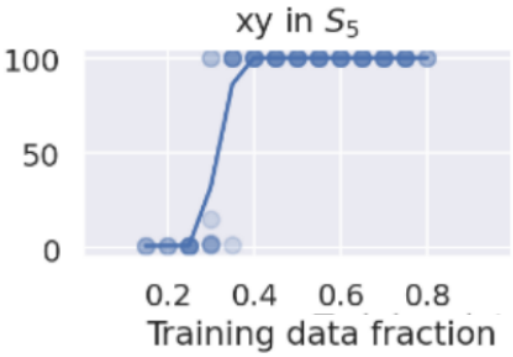


Figure 4.26: Train data percent vs Validation accuracy for binary operation $x \cdot y \; x, y \in S_5$ trained conventionally

We observe that we require $10\% \sim 15\%$ more training data to achieve a validation accuracy at par with the "Grokking" [1] results. However we are able to follow similar trend on validation accuracy improvement with increasing train data percentage. We are unable to perform well on $x^2 + y^2 + xy + x \pmod{97}$ dataset. We observe that much of the training curves still undergo overfitting and hence we decrease the number of parameters added during expansion in the next set of experiment.

4.4 Experiment 4

In this next set of experiments we aim at improving on the lower train data percentages. The expansions used for the same are described in Table 4.3. The expansion steps are set such that the number of parameters of the student network is atmost 16 more than that of the teacher network.

Expansion	n_{layers}	d_{model}	n_{head}	Optimization steps
	1	8	4	2000
1	2	8	4	2000
2	2	16	4	2000
3	2	32	4	2000
4	2	48	4	4000
5	2	64	4	4000
6	2	80	4	8000
7	2	96	4	8000
8	2	112	4	16000
9	2	128	4	50000

Table 4.3: Network hyperparameters and Optimization steps for each expansion of Experiment 4

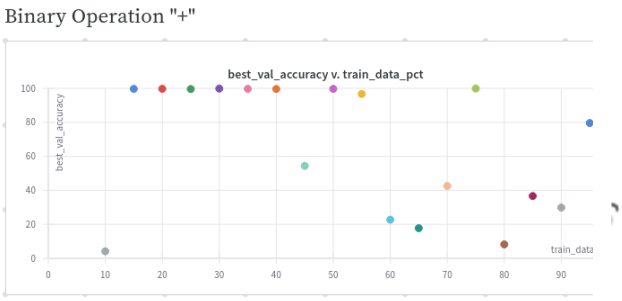


Figure 4.27: Train data percent vs Validation accuracy for binary operation $x+y \pmod{97}$ using expansion method

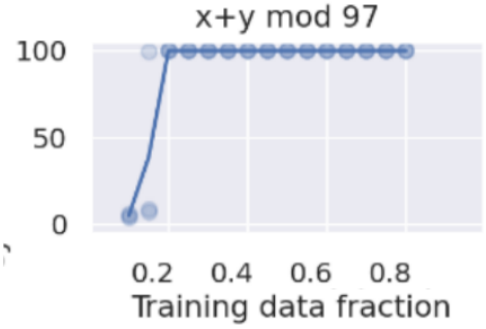


Figure 4.28: Train data percent vs Validation accuracy for binary operation $x+y \pmod{97}$ trained conventionally

Source: Image from [1]

Figure 4.29: Train data percent vs Validation accuracy for binary operation $x-y \pmod{97}$ using expansion method

Figure 4.30: Train data percent vs Validation accuracy for binary operation $x-y \pmod{97}$ trained conventionally

Source: Image from [1]

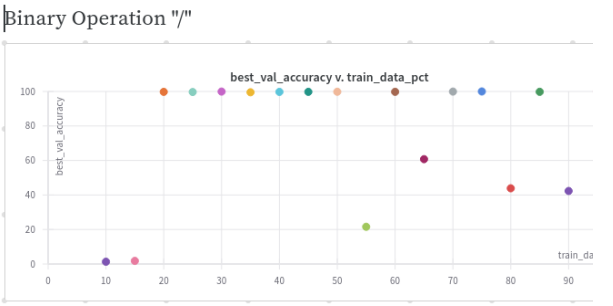


Figure 4.31: Train data percent vs Validation accuracy for binary operation $x/y \pmod{97}$ using expansion method

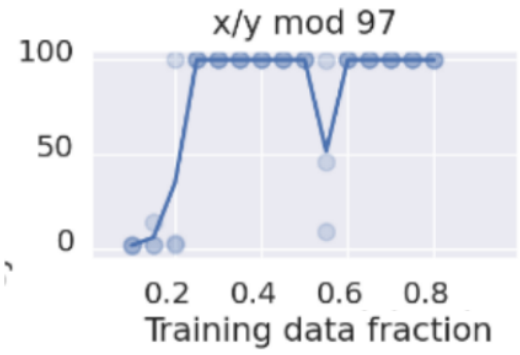


Figure 4.32: Train data percent vs Validation accuracy for binary operation $x/y \pmod{97}$ trained conventionally

Source: Image from [1]

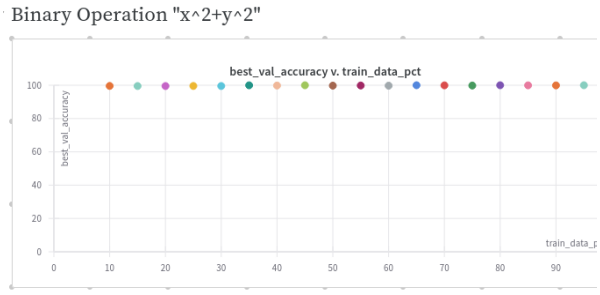


Figure 4.33: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 \pmod{97}$ using expansion method

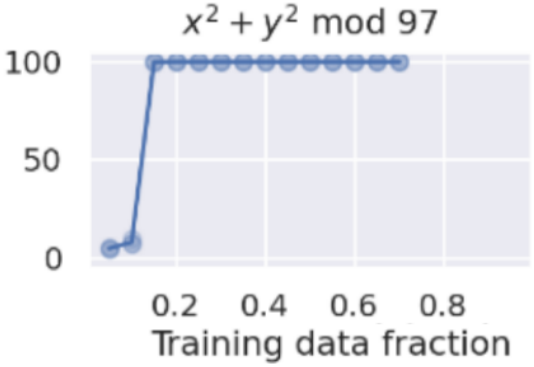


Figure 4.34: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 \pmod{97}$ trained conventionally

Source: Image from [1]

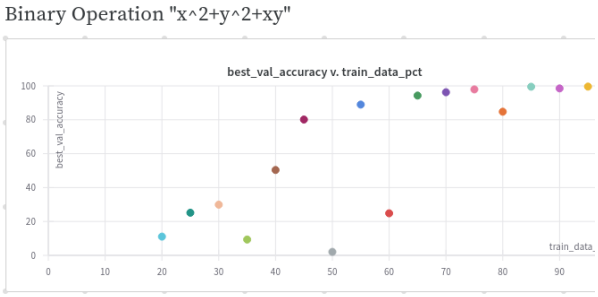


Figure 4.35: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y \pmod{97}$ using expansion method

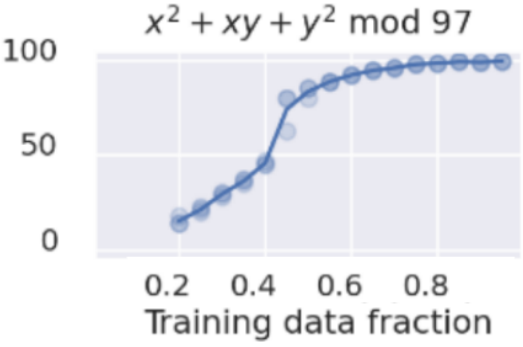


Figure 4.36: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y \pmod{97}$ trained conventionally

Source: Image from [1]

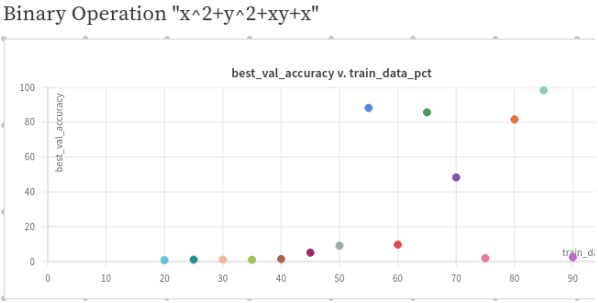


Figure 4.37: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y + x \pmod{97}$ using expansion method

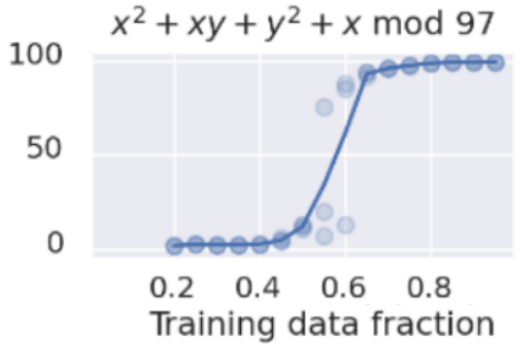


Figure 4.38: Train data percent vs Validation accuracy for binary operation $x^2 + y^2 + x.y + x \pmod{97}$ trained conventionally

Source: Image from [1]

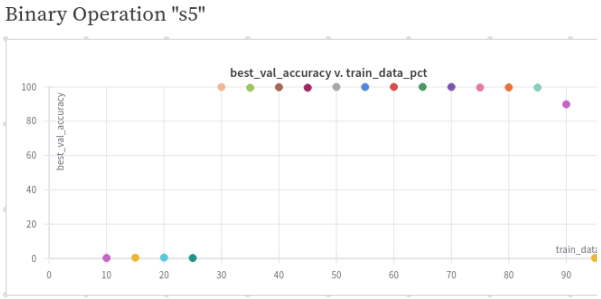


Figure 4.39: Train data percent vs Validation accuracy for binary operation $x \cdot y \; x, y \in S_5$ using expansion method

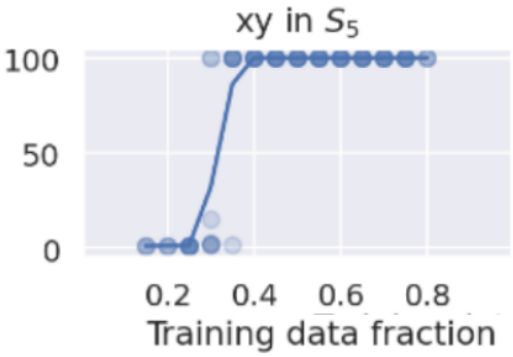


Figure 4.40: Train data percent vs Validation accuracy for binary operation $x \cdot y \; x, y \in S_5$ trained conventionally

We observe improvement on the previous results and are able to achieve $> 99\%$ validation accuracy at similar train data percentages or with 5% lesser training data as compared to [1]. We can see improvement in $x^2 + y^2 + xy + x$ binary operation dataset but we are not at par with [1]. We also observe drop in the performance at higher train data percentages this occurrence is explained in 4.5 and the training curve for the same can be seen in Figure 4.43.

4.5 Training Curves

We observe that overfitting still occurs at lower train data percent and the model reaches higher validation accuracy through "Grokking" phenomenon mentioned in [1]. Hence, model expansion doesn't hinder "Grokking" phenomenon as seen in Figure 4.41. At higher train data percentages overfitting doesn't occur and the model improves as it expands. We also observe that in some of the datasets at high train data percentage the training curve doesn't improve after a couple of expansions 4.43. This can be avoided using large expansion steps similar to ones in Experiment 3 (Table 4.2).

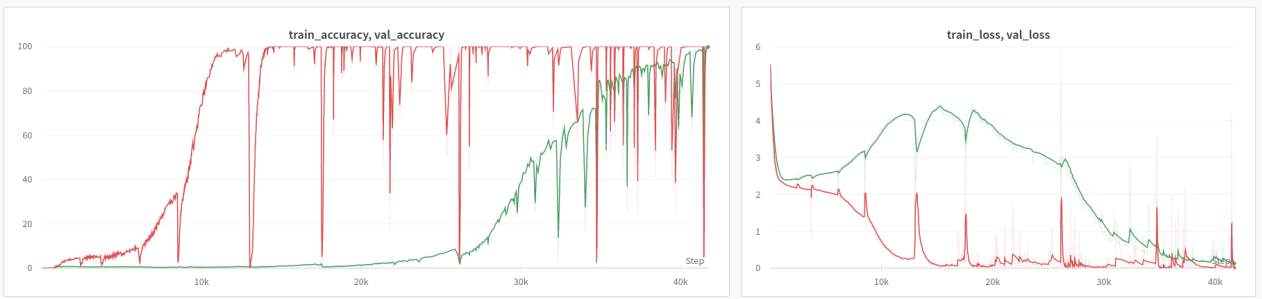


Figure 4.41: Training Curve for $x + y \pmod{97}$ binary operation dataset with 10% train data percentage with expansion as per Table 4.3 (Experiment 4)

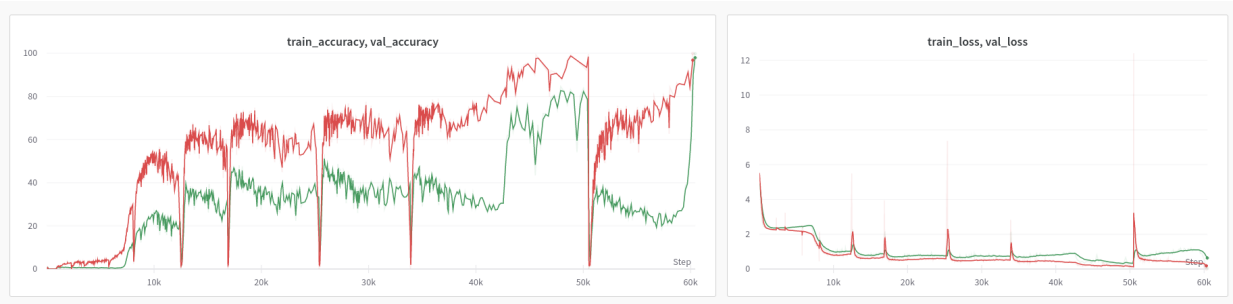


Figure 4.42: Training Curve for $x + y$ (mod 97) binary operation dataset with 30% train data percentage with expansion as per Table 4.3 (Experiment 4)

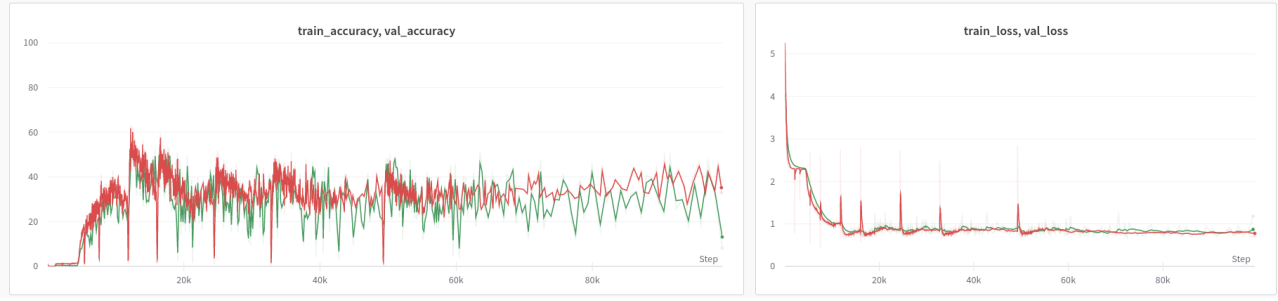


Figure 4.43: Training Curve for $x + y$ (mod 97) binary operation dataset with 80% train data percentage with expansion as per Table 4.3 (Experiment 4)

Discussion and Future Work

We are able to obtain results suggesting that neural expansion does help in generalization. We observe that smaller expansion sizes perform better but higher expansion size allows more flexibility which is required at higher train data percentages. More work is required to test if similar trends can be seen in real world datasets. Moreover a lot of tuning is required to make the method work, an algorithm ensuring that the expansion sizes are optimal, decreasing the amount of hyperparameter tuning that is currently required is needed.

References

[1] Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets.

[2] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[4] Utku Evci, Max Vladymyrov, Thomas Unterthiner, Bart van Merriënboer, and Fabian Pedregosa. GradMax: Growing neural networks using gradient information.

[5] Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. type: article.

[7] Johannes von Oswald, Dominic Zhao, Seijin Kobayashi, Simon Schug, Massimo Caccia, Nicolas Zucchet, and João Sacramento. Learning where to learn: Gradient sparsity in meta and continual learning.

[8] Xingkui Zhu, Shuchang Lyu, Xu Wang, and Qi Zhao. Tph-yolov5: Improved yolov5 based on transformer prediction head for object detection on drone-captured scenarios. *CoRR*, abs/2108.11539, 2021.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry,

Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[10] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *CoRR*, abs/2102.12092, 2021.

[11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization.

[12] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks.

[13] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism.

[14] Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks.

[15] Gintare Karolina Dziugaite, Alexandre Drouin, Brady Neal, Nitarshan Rajkumar, Ethan Caballero, Linbo Wang, Ioannis Mitliagkas, and Daniel M. Roy. In search of robust measures of generalization.

[16] Dustin Dannenhauer, Matthew Molineaux, Michael W. Floyd, Noah Reifsnyder, and David W. Aha. Self-directed learning of action models using exploratory planning.