Rethinking Neural Computation

Anonymous submission

Abstract

Previous research into expressiveness of Neural architectures to perform arbitrary computation predominantly comes from the expectation these Neural architectures will learn which computations to perform in the training process (out of all arbitrary computations). We question the practicality of this expectation given recent theoretical and empirical results on challenges with Deep Learning. Instead we propose a simple Neural architecture, the Synthesizer, that simply outfits other Neural architectures with the ability to memorize and perform programs, thus extending any implicit Neural computational abilities with explicit computational abilities. Our theoretical and empirical analysis finds the Synthesizer is capable of learning and performing arbitrary programs in a practical manner while still maintaining standard Deep Learning abilities both theoretically and empirically. Preliminary experiments show how the Synthesizer has the novel ability to learn and perform "soft-programs" that merge computational and Deep Learning abilities and display a kind of "thinking fast and slow".

Introduction

This paper comes primarily as the intersection of two disparate lines of thought: 1) A theoretical question of whether Neural architectures and Deep Learning can practically be trained to perform *any* arbitrary computation, and 2) A thought experiment about how humans learn computational tasks and are able to merge these abilities with "intuition".

Neural Arbitrary Computation? To this first point, despite Recurrent Neural Networks (RNNs), LSTMs, and Transformers previously being shown to be Turing Complete (Siegelmann and Sontag 1994)(Chen et al. 2018)(Pérez, Marinković, and Barceló 2019a)(Pérez, Barceló, and Marinkovic 2021)(Pérez, Marinković, and Barceló 2019b). These results are targeted at showing the *expressiveness* of these architectures (i.e. there are not limitations to the ability of the models to perform arbitrary computations) and likewise do not/are not meant to show whether these Neural architectures can practically be trained with gradient descent to perform said arbitrary computations. Memory Assisted Neural Networks (MANNs) like Neural Turing Machines (Graves, Wayne, and Danihelka 2014), Differentiable Neural Computers (DNCs) (Graves et al. 2016), and Neural GPUs (Kaiser and Sutskever 2015) definitively have arbitrary memory manipulation (and thus computational) abilities, and are meant to address some of the practicality concerns of Neural architectures learning to perform arbitrary computation (by showing experimental results of the architectures learning basic kinds of computation), though no promises are guaranteed for these architecture to practically learn *any computation* especially since, again these models are expected to learn computation implicitly. We further add to this discussion drawing from the Statistical Query (Kearns 1998) and Learning Complexity literature as there are numerous examples of "reasonable" input-output distributions which are both trivial to compute, but need exponential time to learn using gradient descent regardless of Neural architecture. To give some examples, (Shamir 2016) considers inputs passed through a 2-3 layered NN (in a family of similar NNs), (Goel et al. 2020) and (Song et al. 2017) consider 1 layer NNs, (Daniely and Vardi 2021) considers a variety of easily computable distributions (ReLU NNs, DNF formulas, Boolean circuits, etc.) under cryptographic assumptions to name just a few (note that each paper considers different kinds of "reasonableness", i.e. what distribution the inputs are drawn from, activation function constraints, etc.). We contend in this paper that this inability to learn certain kinds of computation can simply be addressed by having a Neural Architecture that can explicitly learn any arbitrary computation.

Human Computational Abilities Thought Experiment Consider the computational ability for a human to count, say, a binary sequence, ex. "How many 1s are in the following sequence: 11101011?". How would a human learn to perform such a task? Would they be shown thousands or millions of binary sequences alongside the correct answer (say with some kind of reward/punishment scheme) and be expected to learn the concept of counting? We view this latter question as the paradigm of modern Deep Learning systems (including MANNs) which try to get Neural architectures to learn computation simply through being shown examples. A more realistic scenario would be that this counting task would be reformulated into teaching the computational concept of counting itself, i.e. *explicitly* teaching a human to enumerate each "1" in the sequence and to then total the number of "1s". Note that there is some nuance to this sce-

nario though, and that despite the human being "explicitly taught computation", that there is still an intuitive aspect to this whole process. Despite being taught to count, the human would still need to use intuitive cues to know when to use their counting ability. To illustrate this, imagine a child being presented with a binary sequence like "11101011" on a sheet of paper by a teacher. The child may be extremely confused (given their youth) as to what is expected of them and will need to "intuitively guess" that the teacher wants them to count the number of "1s" and thus start to counting aloud "one, two, three...", but note how if the teacher exclaims "No! Don't do that!" the child will have to use their intuitive natural language interpreting skills and know to stop. The point of this scenario is the describe how humans merge their computational and intuitive abilities seamlessly, and is what we hope to achieve with the Synthesizer learning and performing "soft-programs". Alternatively, one can also view this scenario as combining "thinking fast and slow" as coined by (Kahneman 2011) from the cognitive and psychological literature, where both humans and the Synthesizer delicately balance fast "intuitive" thinking (like deep learning) and slow methodical thinking (like computation). This is also the motivation for why this paper cares about creating Neural architectures that can learn to perform arbitrary computation (say, despite computers being able to perform arbitrary computation) while integrating said architecture with Deep Learning.

Other Related Work

Hybrid Neural Systems This field was developed greatly in the 1990s and has almost the *exact same premise* as this paper, i.e., trying to combine "connectionist" Neural Networks with "computationalist" symbolic processing structures (so that you would have a Neural architecture that could do both). Our Neural architecture does have a separate symbolic/logical processing system as Hybrid Neural Systems often do (Wermter and Sun 1998)(McGarry et al. 1999) and instead we show how Deep Learning can be viewed equivalently as learning to compute a "soft-instruction" and computation can be viewed as learning a connectionist representation of algorithms. To this extent we may have achieved the goal of Hybrid Neural Systems to merge computationalist and connectionist abilities.

Program Synthesis As stated, the Synthesizer will "memorize programs" though we draw a *sharp* distinction between our work and the Language Model Program Synthesis literature first preliminarily explored in (Wang and Komatsuzaki 2021) and (Brown et al. 2020) then explicitly explore in (Chen et al. 2021) and has been used in a variety of applications like to solve competitive programming problems (Li et al. 2022) or university-level mathematics (Drori et al. 2022). The Synthesizer has external memory which it will use (only at inference time though) to *perform* a program and not just list out the program in text form (as per Language Model Program Synthesis). This distinction is important because it gives the Synthesizer the ability learn and perform "soft-programs" i.e. where there are instructions/aspects to the program that are not valid instructions in any programming language and thus can't be generated by Language Modeling programs. However this literature is still important as it supports the concept that Neural architectures can understand natural language prompts and relate them to code. We also reiterate that the Synthesizer can replicate all classification-based Deep Learning including Language Model Program Synthesis as we will show.

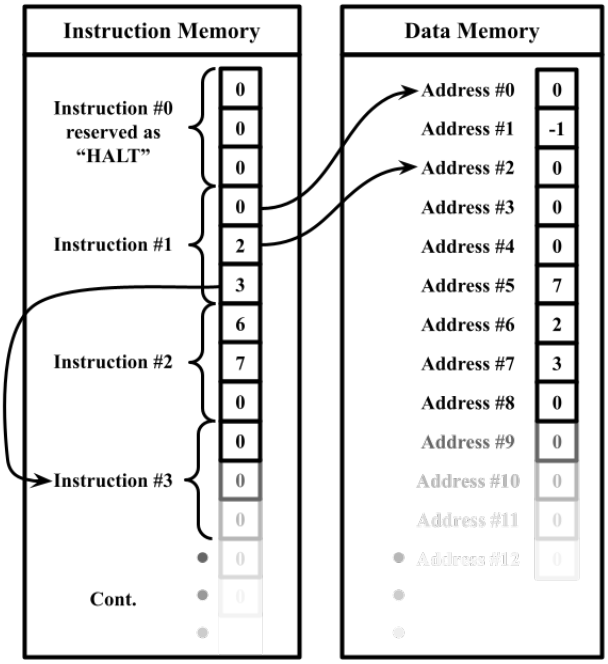

Figure 1: Subleq example and memory layout. "Instruction Memory" is where subleq instructions are stored, "Data Memory" stores numbers that will be manipulated during runtime of the Instructions. As shown the 0-th instruction is reserved as a "HALT" instruction that ends the program. Thus, at runtime, a subleq program will start with Instruction #1 which says to "subtract the data stored at Address #2 (i.e. $B*$) from the data stored at Address #0 (i.e. $A*$) and go to instruction #3 next if $B^* - A^* \leq 0$ (which it is in this case) otherwise just go to the next instruction in Instruction Memory (in this case instruction #2). Note that any "inputs" that would normally be passed to a program will simply just be stored in Data Memory. This also shows how subleq programs (in Instruction Memory) can be represented simply as a vector of only integers.

Subleq Background

In order to get a Neural architecture to "memorize and perform" arbitrary computation, we need to introduce a model of arbitrary computation. We choose the esoteric language "subleq" because it is an extraordinarily simple model of a computation as it only has one instruction, but is nonetheless Turing Complete (i.e. capable of expressing any arbitrary computation) even when Instruction Memory and Data Memory are separate (Nürnberg, Wiil, and Hicks 2004). A "subleq instruction" takes three arguments (A, B, C) where

Algorithm 1: Subleq

Input: Addresses (A, B, C)
Output: C_{t+1}
  $B^* = B^* - A^*$
  if $B^* \leq 0$ **then**
    Let $C_{t+1} = C$
  else
    Let $C_{t+1} = C_t + 1$
  end if
  if C == 0 **then**
    HALT
  end if
  (**global**) $C_t = C_{t+1}$ # note: C_t is initialized at 1
  return C_{t+1}

A and B are addresses in "Data Memory" and C references an instruction in "Instruction Memory" (see Figure 1 for an example and what this would look like).

We consider a variation of subleq shown in Algorithm 1 which functionalizes subleq (so that it can used in the Synthesizer) and has a HALT-ing capability to stop the program. This adaptations do not change core of how subleq represents "subtract and branch if $\leq$ 0" which makes it Turing Complete (as that is what makes subleq able to arbitrarily manipulate memory). Note in Algorithm 1, the * symbol means to retrieve the value at said address in "Data Memory". Also note that normal subleq instructions are perfectly capable of implementing more expressive instructions like JMP, ADD, MOV, BEQ, etc. (Nürnberg, Wiil, and Hicks 2004).

Warning We use subleq as it is a simple Turing Complete model of computation which makes analysis, experimentation, and understanding of our arguments very practical and reasonable since all instructions are just integer addresses. However, subleq is very clumsy language and **cannot** perform algorithms that generalize to any arbitrary input lengths (because the addresses in Instruction Memory are static/unchangeable). To resolve this matter, we see no reason why the theoretical analysis and experiments in this paper can not be trivially extended to have the Synthesizer implement some version of assembly which does have programs that generalize to arbitrary input lengths (i.e. instead of producing logits for to classify the tuple (A, B, C), the Synthesizer would produce logits to classify the tuple ($\langle$ $assembly\ instruction$ $\rangle$, arg1, arg2, arg3) where program memory would still be static).

We take this opportunity at the end of this subleq section to clarify and reiterate that the Synthesizer *should not be used to perform large & tedious computations* (ex. large matrix multiplications) as of course it would be much faster and more reliable to have a standard computer do so). The point of this paper to show a Neural architecture that has explicit arbitrary computational abilities in a way replicates human computational abilities to some extent.

Synthesizer Architecture

The Synthesizer is a simple extension of other Neural Architecture/Deep Learning systems simply by making said architectures produce the subleq instructions (A, B, C) for instruction number C_t (see Figure 2) that will be run at inference time. Note that because the Synthesizer "memorizes" subleq programs, at inference time there is no explicit "Instruction Memory" (however there is still "Data Memory"), the instruction numbers C, C_t, and C_{t-1} are just integers that serve as "placeholders" for the "Instruction Memory" that is implicitly represented in the weights and biases of the Synthesizer. To elaborate on this, in this section we will define the Synthesizer, describe the dataset format/how to train it, show a short proof that bounds the number of parameters needed to do "program memorization", and finally show a construction for how the Synthesizer can replicate any classification-based Deep Learning system.

Definitions

For those who are comfortable enough with the description given by Figure 2, feel free to skim this subsection and go straight to the subsection on training the Synthesizer.

Neural Network Model Definition We define the "Neural Network Model" inside the synthesizer loosely as any Neural Network-based Model that can map $\mathbb{R}^{|inputs|} \rightarrow \mathbb{R}^{|outputs|}$ for all $|input| \in \mathbb{N}$ and $|output| \in \mathbb{N}$. This would include Feedforward Neural Network (FNN), Convolutional Neural Network (CNN), or Recurrent Neural Networks (RNN), and simple variations/combinations of these structures or even a Transformer as will be explore in the experimental section. We leave this definition loose as we believe there is room for development in terms of what Neural Network model should be used in the Synthesizer.

Synthesizer and its Components Definitions

1. Let $nn_{(\ell,k,n)}$ be a Neural Network model (denoted as nn) that accepts an input vector $\mathbf{v_{in}} \in \mathbb{R}^{(1+\ell)}$ and outputs a vector $\mathbf{v_{out}} \in \mathbb{R}^{2k+n}$ where ℓ is maximum length of the "Program Description", k is maximum size of Data Memory, and n is the maximum number of instructions, (see Figure 1 for clarification on what Data Memory and instructions are).
2. Let $\mathbf{v_{in}} = concat([C_t], \mathbf{d})$ where $concat$ takes any vector inputs arguments and combines them into one vector, $[C_t]$ is a one-dimensional vector that represents instruction to run at time t, and $\mathbf{d} \in R^n$ and represents the "Program Description". In the worst case this can simply be the subleq Instruction Memory vector (see Figure 1), though in more practical implementations a tokenized natural language description should also work (as per Language Model Program Synthesis literature).
3. Let $split(\mathbf{v_{out}}, k, n)$ be a function that returns the tuple $(\mathbf{v_{out}}[0:k], \mathbf{v_{out}}[k:2k], \mathbf{v_{out}}[2k:2k+n])$, where $[x:y]$ means to return elements x through y (not including element y) of a vector. This tuple represent the "logits" for classifying the addresses of the subleq tuple (A, B, C) which we will call (A_l, B_l, C_l)

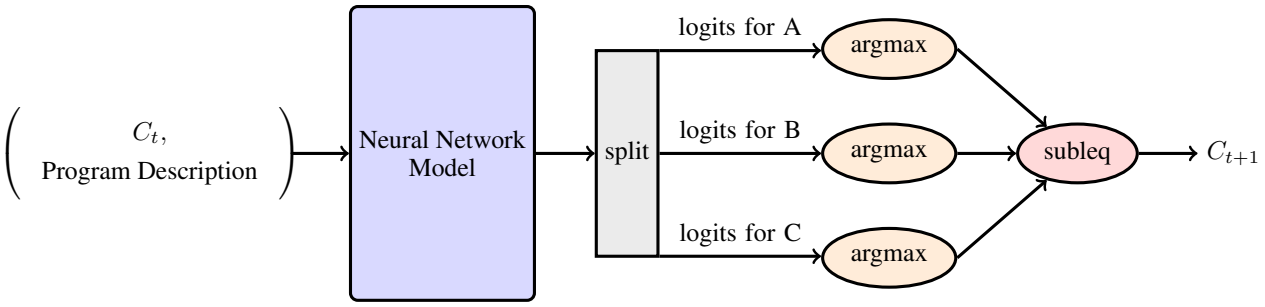

Figure 2: Synthesizer Architecture. It takes as input the instruction number C_t concatenated with a vectorized "Program Description" (for example, through tokenization) then predicts the operands of said instruction by producing a output vector that will be split into "logits" for each subleq arguments (A, B, C) (logits can simply be viewed as "scores" for each possible classification output). The argmaxes will produce the actual subleq arguments (A, B, C) to run and will pass them to the "subleq" operator (described in Algorithm 1) which accesses external "Data Memory" (see Figure 1). The output of the Synthesizer is the next instruction to run.

4. Let $argmax$ be function which takes as input the tuple (A_l, B_l, C_l) and argmaxes each element in the 3-tuple (i.e. outputs the index of the max logit in the logits vector) to produce a classification prediction for the subleq parameters (A, B, C)
5. Let subleq be a function defined by Algorithm 1. Note that as per Algorithm 1, the subleq function will reference and alter external memory. This can be implemented by simply maintaining an array for Data Memory that also contains the inputs to the program.
6. Let a Synthesizer be denoted by $S_{(nn,\ell,k,n)}$ where the Neural Network Model inside the Synthesizer is $nn_{(\ell,k,n)}$. Thus, we can define a Synthesizer as

$$Synthesizer(\mathbf{v_{in}}) =$$

$$subleq(argmax(split(nn_{(\ell,k,n)}(\mathbf{v_{in}}), k, n)))$$

Training the Synthesizer

Dataset Consider a set of subleq programs $P^{n,k}$ with $< n$ instructions and use $< k$ sized Data Memory where $|P^{n,k}| = N$ (i.e. there are N total programs). Let $P_i^{n,k}$ be the i-th program in $P^{n,k}$ and $P_{i,j}^{n,k}$ be the j-th instruction in $P_i^{n,k}$. Let $(A, B, C)_{ij}$ be the parameters for the j-th subleq instruction of the i-th program in $P_{n,k}$. Finally, let d_i be the vectorized "Program Description" for program $P_i^{n,k}$. Inputs of the dataset are defined as $x_{i,j} = concat([j], d_i)$ with corresponding output $y_{ij} = (A, B, C)_{i,j}$ for all i on the range $[0, N)$ and for all Instruction #s $j \in P_i^{n,k}$. The maximum size of this dataset is thus $N * n$.

Training During training, neither the $argmax$ nor $subleq$ steps need to be computed as the only "learn-able" parts of Synthesizer that produce logits for (A, B, C) from the input. Thus to train the Synthesizer, simply input $x_{i,j}$ to the Synthesizer and take the logits $(A_{logits}, B_{logits}, C_{logits})$ after the $split$ operation (see Definitions subsection) and compute the Cross-Entropy Loss for each of the logits and sum them, i.e. $loss = CE(A_l, A) + CE(B_l) + CE(C_l)$ where $CE(logits, label) = -ln\frac{exp(logits[label])}{\sum_{l \in logits} exp(l)}$.

Inference Note that inference time differs from training as at inference time the logits will be passed to the $argmax$ and $subleq$ parts of the so that the subleq program can will run.

Program Memorization Bound

Because the ability of the Synthesizer to explicitly learn any arbitrary computation relies on "program memorization" we show a simple proof that makes use of known NN memorization capabilities to ensure that said "program memorization" is realizable from a theoretical perspective and show a construction for how this can be done with an upper bound of a polynomial number of parameters.

Theorem 1 $\forall P^{n,k}, i, j \, \exists \, \ell, d_i, nn \mid S_{nn,\ell,k,n}(concat([j], d_i))$ *which correctly predicts* (A, B, C) *for* $P_{i,j}^{n,k}$ *using either only* $O(k * n + n * N)$ *parameters or* $O(k * N + n * N)$ *parameters.*

We can prove this with a simple construction that particularly makes use of results from (Yun, Sra, and Jadbabaie 2018) which shows a construction for Neural Networks that memorize classification datasets with 100% accuracy.

1. Let $d_i = [i]$ and likewise let $\ell = 1$
2. All inputs in the dataset $x_{i,j} = concat([j], d_i) = concat([j], [i]) = [j, i]$ are thus unique
3. If the logits produced for a classification task are the z dimensional one-hot vectors of the label (where z is the maximum class label), then the logits will be 100% accurate for classifying the label as the argmax used to determine the predicted label will cancel the one-hot encoding of the label
4. (Yun, Sra, and Jadbabaie 2018) shows a construction for a ReLU Feedforward network with 4-layer that maps arbitrary inputs to one-hot vectors of dimension z with hidden dimension sizes of $h_1 h_2 \geq 4(|dataset|)$ and

$h_3 > 4z$ where h_1 is the size of the first hidden activation, h_2 is the size of the second hidden activation, and h_3 is the size of the third hidden activation layer (this is stated explicitly in Proposition 3.2, though the construction is later in the paper) so long as inputs are unique (already shown by step 2)

5. Let nn_A be the network described in step 4 which maps $[j, i]$ to one-hot vectors of the label A for $P_{i,j}^{n,k}$ which has hidden sizes of $h_1 = 2N$, $h_2 = 2n$ as in our case $|dataset| < n * N$ and with these sizes, $h_1 h_2 = 4(|dataset|)$ as per step 4. Continuing, let $h_3 = k$ the maximum size of Data Memory which A is addressing.
6. Do the same for B, i.e. let nn_B be the network that maps $[j, i]$ to one-hot vectors of the label B for $P_{i,j}^{n,k}$ with the same hidden sizes as the previous step (since B also just addresses Data Memory)
7. Let nn_C be the network that maps $[j, i]$ to one-hot vectors of the label C for $P_{i,j}^{n,k}$ which has (reversed) hidden sizes of $h_1 = 2n$, $h_2 = 2N$ (makes for better complexity) and $h_3 = n$ since C addresses Instruction #s

Generalizing Deep Classification Architectures

Theorem 2 *The Synthesizer can perform similarly to known classification Deep Learning systems by using the same Neural Architecture inside the Synthesizer and then learning a family of one instruction programs that rely on the input. This assumes that extra input dimensionality and output dimensionality does not particularly hurt performance while training.*

Soft-Instruction Construction

1. Consider a State-of-the-Art (SOTA) Deep learning Classifier architecture. We could make the Neural Network model in the Synthesizer be this SOTA architecture.
2. Now, let us consider a family of programs that only have one instruction but have an extended Program Description. The program description would be "Classify this input: {classification input}" where the "classification input" would be the input the input the SOTA architecture takes i.e., "text", "image", "signal", etc. The single instruction to learn would be

   $$subleq(\&(-classification\ number),\ \&output,\ 0)$$

   where "&" just means "the address of". The classification numbers are just the same classification numbers of the SOTA model. At a higher level, the instruction to learn is to just to write the right class number to an output location in memory.
3. Train the Synthesizer to output the correct classification number for each labeled input (i.e., mainly just picking operand A correctly as picking operands B/C correctly is trivial as they are always the same).

Note how the classification problem the Synthesizer learns is the same as the original classification problem. Furthermore, the Synthesizer is also using the same SOTA model to do the processing. Thus, the only difference is there may be some extra dimensionality to the input and output that is either trivial or can be learned to be ignored. To elaborate, the input would contain the program description and instruction number which are ignorable since they would always be the same. Then, the output has two other trivial classification to make (for operands B and C which are also always the same) as well as some extra output dimensionality for predicting A as each classification needs to be able to address the entire external memory (though most addresses would not be considered). Note we will show this exact construction and show that the extra input dimensionality does not harm performance with experiments.

ResNets and Transformers ResNet is a classifier model that fits directly into the construction. Transformers can be purely classification based in the case of BERT (Devlin et al. 2018) (and can thus be fit into the construction). In the case of Language Modeling, we could do still do next token classification just without the parallelism of the original Transformer model given by the matrix multiplication of the multi-head attention (Vaswani et al. 2017).

Regression Since the Synthesizer is classifier based and produces discrete outputs, it fundamentally cannot perform regression. However the Synthesizer can predict binned regression values (ex. if one is performing regression to predicting y between the range [-1, 1] one can instead consider discretely predicting if $y \geq 0$ or $y < 0$).

Experiments

Experiment 1: Arbitrary Program Memorization

In this experiment we hope to compliment Theorem 1 and its theoretical analysis with a practical implementation to show that Synthesizer can learn arbitrary computation through memorizing programs. Instead of using a Feedforward NN as the theoretical analysis does, we instead use a more modern and state-of-the-art Transformer, specifically the BERT (Devlin et al. 2018) architecture. This has two intentions, to show that not just Feedforward NNs can memorize programs and also because it leads into our "soft-programs" experiments which also use BERT NLP abilities. Given this BERT architecture we also use the BERT tokenizer to tokenize C_t as well as Program Descriptions. We use a BERT model with embedding and hidden sizes of 256 where the intermediate size is 1024 (for the feedforward layer).

Experimental Setup To show arbitrary program learning abilities we generate random subleq programs of length 256 and vary the number of the programs for the Synthesizer to memorize. For this experiment, the program description is simply a "special token" (i.e. extending the embedding to represent another token). Results are shown in Table 1, and while random is not the same as arbitrary, we assert that if the Synthesizer can memorize programs that are completely without pattern (because they're random) that there is no particular reason to believe that the Synthesizer could not memorize any arbitrary program (given sufficient parameters of course). Exact specifications for this experiment can be found in the Appendix.

Program Length	# of Programs	# of Steps until 100% Accuracy	Batch Size
256	16	1030 ± 74	204
256	32	1366 ± 95	409
256	64	1907 ± 153	819
256	128	2991 ± 72	1632

Table 1: Basic experiment details of training a synthesizer to memorize random programs. Each experiment was run 5 times and shown is the mean number of steps needed to reach 100% accuracy on predicting the correct addresses for each (A_{ij}, B_{ij}, C_{ij}) as well as $\pm$ the standard deviation (both are rounded to the nearest integer). Batch sizes were arbitrary set as $\lfloor$(Program Length*Program Inputs)/20$\rfloor$ because preliminary experiments generally found larger batch sizes lead to faster convergence for program memorization.

Experiment 2: Soft-Instruction Experiment

As Theorem 2 which decribes how the Synthesizer can generalize/replicate classification Deep Learning using a "Soft-Instruction" we show just that in this experiment. We generate variation of the sst2 dataset (Socher et al. 2013) with the same formulation as the construction for Theorem 2 (i.e. a program description of "Classify this input: {sst2 input text} in this case we're classifying the binary task of sentiment analysis. We again use a BERT model of the same specification of Experiment 1 (note that we do not pretrain our BERT models as the original BERT model does with masking (Devlin et al. 2018) as this would not fit with the theme/scope of this paper). To show, the input dimensionality did not hurt results we compare the Synthesizer vs a BERT model of the same specifications that is directly trained for sequence classification. For a fair comparison, only the accuracy of correctly classifying argument A (which is the argument that differs based on sentiment) are shown in Figure 3, and exact experimental specifications are listed in the Appendix.

Experiment 3: Soft-Imperative Experiment

This experiment is the most tricky to explain, but is potentially the most important. We explore whether or not computational and Deep Learning abilities can be performed at the same time where the process is somewhere in between perform computation and utilizing Deep Learning abilities. To do this, we consider a variation of both the Arbitrary Program Learning experiment and the Soft-Instruction experiment and wonder whether every instruction in a program could be turned into a soft-instruction. Thus we task the model with the prompt "As long as the input text is postive run {special program token from Experiment 1} [SEP] {sst2 input text}". To generate the dataset, we take the sst2 dataset and randomly combine each sst2 sample with a random program token and random instruction number (though we only consider 10 different random programs of length 10), when the text is negative sentiment, the Synthesizer is tasked with producing "subleq(0,0,0)" to halt the execution (similar to the child-teacher thought experiment in the introduction) whereas when the text is positive the Synthesizer is tasked with correctly producing subleq instruction for the random program at the instruction number. To make sure the Synthesizer unambiguously classified between the two, the random programs generated would never reference data or instruction memory 0 (as those would be same arguments as subleq(0,0,0) meant for negative sentiment texts). Results are plotted in Figure 3, and surprisingly the Synthesizer is able to just about match the performance of the base Bert for Sequence Classification model, which suggests the errors primarily came from not classifying the text correctly as opposed to not being able to perform the program correctly.

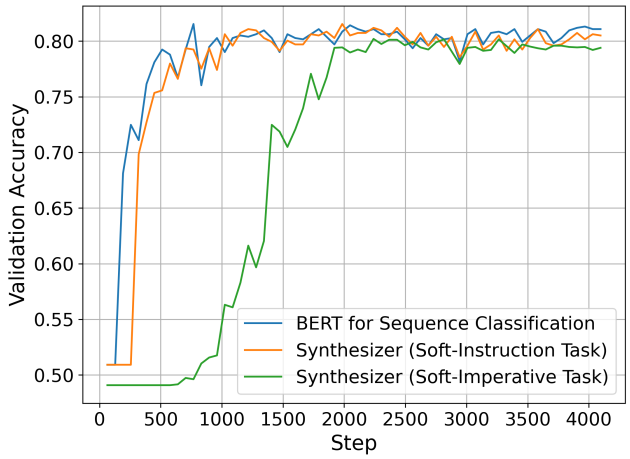

Figure 3: Validation accuracies of the Soft-Instruction and Soft-Imperative task trained for 2 epochs compared to just training a BERT model of the same size. After repeating the experiments 5 times best validation accuracies $\pm$ the standard devation were as follows: BERT for Sequence Classification achieved 81.9% $\pm$ .5%, Synthesizer (Soft-Instruction Task) achieved 81.9% $\pm$ .5%, Synthesizer (Soft-Imperative Task) achieved 80.7% $\pm$ 1.3%.

Limitations & Future Work

Specific architecture No system calls are currently allowed including even basic ones like print, memory allocation or read/write from disk. Though, the focus of this paper is on expressiveness of the Synthesizer. As stated, subleq is a somewhat clumsy language that acts as a simple model of computation for analysis, but may not be right for practical applications. A Reduced Instruction Set Computer (RISC), Complex Instruction Set Computer (CISC), or a kind of Assembly may be more suitable for future work.

Limited Datasets While there is an abundance of programming task data ((Chen et al. 2021) used public Github repositories) and also an abundance of natural language/vision data, there are not particularly many tasks that require both at the same time. It is also difficult to consider what "programming" looks like when allowed to make use of deep learning within computation.

Societal Impact

This work is largely theoretical in nature and presents abstract ideas and thus most ethical/impact considerations would be speculative. Though, we do hope that future work that merges computation with deep learning using similar ideas seriously considers the risks of wrong/biased/misaligned arbitrary code execution that is also empowered by possible biased and misaligned deep learning biases.

References

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, Y.; Gilroy, S.; Maletti, A.; May, J.; and Knight, K. 2018. Recurrent Neural Networks as Weighted Language Recognizers. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2261–2271. New Orleans, Louisiana: Association for Computational Linguistics.

Daniely, A.; and Vardi, G. 2021. From Local Pseudorandom Generators to Hardness of Learning. In Belkin, M.; and Kpotufe, S., eds., *Proceedings of Thirty Fourth Conference on Learning Theory*, volume 134 of *Proceedings of Machine Learning Research*, 1358–1394. PMLR.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Drori, I.; Zhang, S.; Shuttleworth, R.; Tang, L.; Lu, A.; Ke, E.; Liu, K.; Chen, L.; Tran, S.; Cheng, N.; Wang, R.; Singh, N.; Patti, T. L.; Lynch, J.; Shporer, A.; Verma, N.; Wu, E.; and Strang, G. 2022. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32): e2123433119.

Goel, S.; Gollakota, A.; Jin, Z.; Karmalkar, S.; and Klivans, A. 2020. Superpolynomial Lower Bounds for Learning One-Layer Neural Networks using Gradient Descent. In III, H. D.; and Singh, A., eds., *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, 3587–3596. PMLR.

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

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

Kahneman, D. 2011. *Thinking, fast and slow*. New York: Farrar, Straus and Giroux. ISBN 9780374275631 0374275637.

Kaiser, Ł.; and Sutskever, I. 2015. Neural GPUs Learn Algorithms.

Kearns, M. 1998. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6): 983–1006.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; Hubert, T.; Choy, P.; d'Autume, C. d. M.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Robson, E. S.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-Level Code Generation with AlphaCode.

McGarry, K.; Wermter, S.; MacIntyre, J.; and St Peter's Campus, S. P. W. 1999. Hybrid neural systems: from simple coupling to fully integrated neural networks. *Neural Computing Surveys*, 2(1): 62–93.

Nürnberg, P. J.; Wiil, U. K.; and Hicks, D. L. 2004. A Grand Unified Theory for Structural Computing. In Hicks, D. L., ed., *Metainformatics*, 1–16. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-24647-3.

Pérez, J.; Barceló, P.; and Marinkovic, J. 2021. Attention is Turing-Complete. *Journal of Machine Learning Research*, 22(75): 1–35.

Pérez, J.; Marinković, J.; and Barceló, P. 2019a. On the Turing Completeness of Modern Neural Network Architectures.

Pérez, J.; Marinković, J.; and Barceló, P. 2019b. On the Turing Completeness of Modern Neural Network Architectures.

Shamir, O. 2016. Distribution-Specific Hardness of Learning Neural Networks.

Siegelmann, H. T.; and Sontag, E. D. 1994. Analog computation via neural networks. *Theoretical Computer Science*, 131(2): 331–360.

Socher, R.; Perelygin, A.; Wu, J.; Chuang, J.; Manning, C. D.; Ng, A.; and Potts, C. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1631–1642. Seattle, Washington, USA: Association for Computational Linguistics.

Song, L.; Vempala, S.; Wilmes, J.; and Xie, B. 2017. On the complexity of learning neural networks. *Advances in neural information processing systems*, 30.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, B.; and Komatsuzaki, A. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

Wermter, S.; and Sun, R. 1998. An overview of hybrid neural systems. In *International Workshop on Hybrid Neural Systems*, 1–13. Springer.

Yun, C.; Sra, S.; and Jadbabaie, A. 2018. Small ReLU networks are powerful memorizers: a tight analysis of memorization capacity.