Inverse Tracr: Mapping Neural Network Weights to Code

Master's Thesis

William Baker

Supervisors: Langosco Langosco
Herbie Bradley
David Krueger

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
MPhil in Machine Learning and Machine Intelligence

Churchill College September 2023

Declaration

I, William Baker of Churchill College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. This dissertation contains fewer than 15,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

In all sections we rely on the Tracr library developed by Lindner et al. [2023]. Our experiments make extensive use of the JAX machine learning framework. As well as Python v3.10, Numpy, Matplotlib, chex, dm-haiku, networkx, pandas, tensorboard, tqdm, kaleido, plotly, torch, flax, optax, jax-smi, transformers, cloudpickle

Total Words: 14964

Code can be found here: https://1drv.ms/u/s!AsshhlQM3x93sxl_EjtwH8qpuMvf?e=bYHSea

William Baker
September 2023

I would like to thank Lauro, Herbie, and David for their masterful guidance, without whom my research would not have been possible.

Abstract

How do you explain a Transformer? Most commonly, people trace activations through the model and learn how model behavior correlates to its inputs. This is a time-consuming process with the potential for misleading outcomes. Our proposal aims to describe the actual computation of a Transformer in code, in a manner less susceptible to misleading alignment.

Recent advancements have granted us the capability to express an algorithm in code and then compile it into a *Transformer Program* - a transformer that precisely implements the same algorithm as the code. With this capacity to compile code into transformers, we can learn how to invert this process. In this paper, we investigate how we can learn to invert transformer programs into the code they derive from, achieved through the use of *meta-models* - models that take the parameters of other models as input. Furthermore, we outline a series of steps aimed at advancing towards the ability to decompile an arbitrary transformer into code, accomplished through the distillation of transformer programs.

We learn to invert three categories of transformer programs using meta-models. The first category comprises of standard transformer programs that are output directly by the compiler; these are sparse and do not faithfully represent how traditional transformers handle information. Next, we compress these transformer programs, resulting in models that are less sparse and present a greater challenge in terms of distinguishing the underlying computation. Finally, we embark on the process of distilling transformer programs from scratch, leading to the creation of models that closely replicate the computational process performed by conventionally trained transformers.

Table of contents

Glossary

meta-model A model that operates by taking the parameters of other models as its input, making predictions based on the behaviors of those underlying models.

MHA Multi-head attention - a mechanism in Transformer models that concurrently captures diverse relationships within input data.

MLP Multi-layer perceptron - a class of artificial neural networks characterized by multiple layers of interconnected nodes

model zoo A dataset of models who's parameters are used to train a meta-model

RASP Restricted Access Sequence Processing Language - A programming language developed by Weiss et al. [2021] that can be implemented by Transformers

Tracr A compiler for RASP that compiles RASP code into a transformer program that exactly implements the algorithm described by RASP

transformer A machine learning model consisting of a number of alternating MHA and MLP blocks.

transformer program A transformer that implements a RASP program

Chapter 1

Introduction

The emphasis on the explainability of machine learning models is increasingly gaining prominence within the field of machine learning. This trend is particularly notable due to the persistent superiority of machine learning models over classical methods in various domains. However, there exists a significant imperative to explain the rationale behind these models. This desire stems from both ethical considerations, prioritizing the transparency and accountability of automated decision-making systems, and strategic reasons, such as fostering user trust and aiding human-machine collaboration.

In response to this demand, the field of Explainable AI (XAI) has emerged. XAI attempts to bridge the gap between the inscrutable *'black box'* nature of advanced machine learning techniques and the need for interpretability within the decision-making processes. Most research focuses on some form of gradient weighted class activation mapping which passes samples through a model and analyses which areas of the model contributed most to the classification, such as LIME [Ribeiro et al., 2016], SHAP [Lundberg and Lee, 2017] and Grad-CAM [Selvaraju et al., 2016].

However, alongside Weiss et al. [2021], we believe that the decision-making process within a transformer model most closely follows that of a program. Current research is also pointing this way with Nanda et al. [2023] investigating how transformers implement algorithms in Fourier space. Our goal is to investigate the possibility of being able to extract the underlying program of a machine learning model and hence better understand its algorithmic procedure. If we can generate models that implement known programs we can test our ability to extract said program.

In Weiss et al. [2021]'s recent publication the RASP, restricted access sequence processing language, was conceived. A programming language where every possible program is guaranteed to have a corresponding transformer that can implement that program exactly. The thought-provoking research conducted by Lindner et al. [2023] built on the ideas presented by Weiss et al. [2021], providing a deterministic compiler called Tracr that can convert RASP code into a transformer. We call these transformers that implement a RASP program a '*transformer program*' following the convention from Friedman et al. [2023].

Tracr served as the inspiration for our study as it naturally prompted the idea of reversing the process, and instead taking a transformer program and attempting to generate the RASP program that it implements. The transformer programs generated by Tracr poorly represent how traditional transformers process data. To address this, we sought a method with the potential to generalize towards inverting an arbitrary transformer. Much of the practice of machine learning specializes in how to train models that generalize well to new data.

Aligning with this principle, we took a machine learning approach to tackle the inverse Tracr problem. Our machine learning model takes a transformer program as input and predicts the corresponding implemented program, employing a form of hyper representation learning known as meta-modeling — where a meta-model is any model that takes the parameters of another model as input.

Training this meta-model required novel solutions to various distinct challenges.

- Program Generator - Tracr lacks the direct capability to compile an arbitrary number of transformer programs. To do so we need to first generate a dataset of RASP programs, where each RASP program is sufficiently complex to teach the meta-model the nuances of the computation within a transformer program, but not so complex as to take a long time to generate or compile with Tracr, since we'll need to generate lots of them. The transformer programs then form our model zoo, with which we'll train our meta-model. (Section 4)

- Determining appropriate representations to input the parameters into our meta-model. (Section 5.1)

- Learning how to deal with permutations of programs, as any given program has many different ways in which it can be implemented and ordered. Thus, we devised a novel method for ordering programs based on instruction depth within a graphical representation. (Section 5.1.1)

- Established measures to quantify the distance between a program prediction to the desired target, ensuring these metrics were differentiable to enable machine learning training via error backpropagation. (Section 5.1.1)

- We extended the innovative concept of compressing transformer programs (Chapter 6) from Lindner et al. [2023] and developed effective methods for distilling transformer programs into more conventional transformer models. (Section 7)

- Scaling up this process into a dataset containing hundreds of thousands of samples presented logistical challenges and quality assurance. (Section 5.3 and 6.1)

- We also developed tools to visualize arbitrary Tracr transformer programs to aid in our understanding, which are used throughout section 3.2.1.

Chapter 2

Problem Specification

In this chapter, we'll explore in greater detail the problem of inverting Tracr and our motivations in pursuing this task. Given a black box transformer model, how do you interpret the computation it performs? One might suggest visualizing the correlation between activations over a range of inputs, or perhaps even trace weights through the network. Nevertheless, these approaches become impractical for expansive models and are susceptible to misleading outcomes.

According to the findings by Weiss et al. [2021], they deduced that the computational nature of a transformer closely resembles that of a program. Their structure consists of a number of steps, each with an input and an output. Building upon this notion, they specify a language that respects the types of computation that a transformer performs as much as possible. Taking this idea a step further, Lindner et al. [2023] expanded upon it and built a compiler for RASP that can deterministically translate a RASP program into a transformer program that performs the exact same computation as described by the RASP program. This implies that we are presently capable of writing programs in RASP and constructing Transformers that enact these programs, coining the term "**transformer programs**."

We believe that the ultimate way of understanding a Transformer is to decompile them into the underlying program that they implement, as this provides an exact description of what the transformer is doing in a format that people are familiar with interpreting. There are a few ways we could go about doing so, first, we could create a deterministic map between a Tracr transformer program and RASP code, however, such a deterministic mapping would only function when given a transformer compiled by Tracr, which would be of little use when trying to interpret a truly opaque model trained in the conventional manner.

Instead, we propose training a meta-model to approximate transformer programs into their corresponding RASP program, by using such an approximate method it is much easier to generalize towards more natural models. To create a meta-model that can map from Transformer programs to RASP programs we needed to construct a dataset of pairs of RASP programs and transformer programs. We had to devise a way of creating hundreds of thousands of unique RASP programs while ensuring that once compiled, they contain sufficient variability to cover as much of the space of transformer programs as possible. If our RASP programs were too simple we would not be fairly assessing our capability to map from an arbitrary transformer program to a RASP program, since some RASP programs would lie outside the sample space of our RASP program dataset.

The simplest way we found of solving this problem was to randomly sample programs. By defining the sampling space for each possible operation within RASP to be as large as possible, any program that can be expressed in RASP could be sampled. Given this way of automatically creating RASP programs, compiling them into transformer programs is trivial thanks to the Tracr compiler. We call the meta-model that we trained on this dataset the "*Standard Inverse Tracr Meta-model*", we were able to achieve up to 73% accuracy at predicting tokens in a program correctly. This alone does not complete the path towards interpreting a *natural* model trained through gradient descent, as Transformer programs compiled directly from RASP code have a very unnatural structure consisting of sparse, one-hot encoded weights.

Lindner et al. [2023] does offer inspiration on how to *compress* a transformer program to more closely match the structure of natural models. To understand this we first have to note how a transformer program stores information between layers. It uses a *residual stream* this is a vector that persists across layers, onto which each layer writes. It functions similarly to a hidden state in a recurrent neural network. Lindner et al. [2023] proposes that we compress this residual stream, every time we wish to read from it we decompress the stream, and whenever we wish to write to it we compress the data we wish to add, before adding it to the residual stream. One can train a compression matrix $W_{emb}^{\top}$ that can be added before each computational block and a decompression matrix W_{emb} added after each block. In practice this causes the residual stream to be much more information dense, containing an analogue encoding of variables rather than a one-hot encoding, more akin to a conventional transformer. This compression matrix can then be used to compress the parameters of the model, giving a new model with fewer parameters that performs the exact same computation as the original. Another meta-model, that we call the "*Compressed Inverse Tracr Meta-model*" is then trained on a dataset of these compressed models, moving us one step closer towards being able to map from a natural model to a rasp program.

By a *conventional transformer* we mean a transformer that has been trained using gradient descent that optimizes all of its parameters to perform some function. These models have no constraints when being trained other than to conform to a certain architecture. Typically their structure is not interpretable and consists of computation distributed randomly across layers and across parameters. We can match this concept while maintaining a transformer programs link to RASP code, by distilling Tracr transformer programs from scratch. Rather than only training W_{emb} we train all the parameters of the model using a teacher to show the model what overall computation to perform. The teacher is a standard Tracr transformer program and the student is a randomly initialized compressed transformer. The loss function used for distillation then minimizes the difference between the outputs of each computational block in the student vs teacher, yielding a natural model that performs the same computation as a standard transformer program. It's worth noting that this is more constrained than the *conventional transformer* we outlined earlier, but dramatically

decreases the constraints compared to a compressed model. Again, we can train another meta-model to map from these natural transformer programs to RASP programs. We call this meta-model a "*Natural Inverse Tracr Meta-model*". This scheme enables us to test the viability of decompiling almost any transformer model into code, as natural models themselves have been trained in a very similar manner.

In summary, we have 3 datasets with which to train our meta-models:

- a dataset of standard transformer programs and RASP programs, were all the parameters in the transformer program are compiled from Tracr
- a dataset of compressed transformer programs and RASP programs, were most the parameters in the transformer program are compiled from Tracr, but we train our own compression matrix
- a dataset of natural transformer programs and RASP programs, where all the parameters are trained from scratch using a standard transformer program as a teacher.

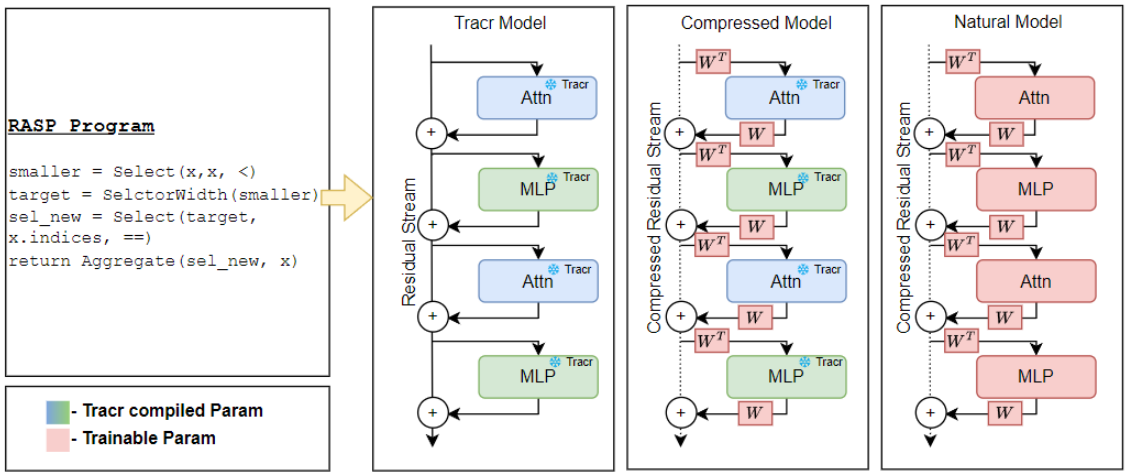


Figure 2.1: The three categories of transformer program used to train meta-models,
Tracr Model - directly compiled by Tracr from RASP code and is input directly into the meta-model for training.
Compressed Model - uses the same Attention and MLP blocks from the Tracr model, but uses a compression matrix W to compress and write to the residual stream, and a decompression matrix $W^\top$ to read from the residual stream. W is trained to maintain the same computation as the uncompressed version.
Natural Model has the same architecture as the compressed model, but all of its parameters are trained from scratch to replicate the computation in the original Tracr model.
Each of these model types will be produced at scale to build three model zoos with which we'll train meta-models. **Note:** A trainable parameter is a parameter that is optimized through a machine learning framework

Chapter 3

Related work

3.1 RASP - Thinking like Transformers

At its heart, our research heavily depends on the work from Lindner et al. [2023] enabling us to generate a dataset of RASP programs and their corresponding transformer programs. Before we dive into the technical aspects of Tracr, it is worth reviewing the programming language upon which it is based, RASP [Weiss et al., 2021].

The novel research by Weiss et al. [2021] presents a programming language where every possible program has a corresponding implementation as a transformer program consisting of a sequence of multi-head attention (MHA) and multi-layer perceptron (MLP) blocks. Admittedly, it is not as powerful as to solve the reverse problem of being able to represent an arbitrary transformer with RASP code, but it does provide strong foundations on which we can build.

RASP uses two types of variables, s-op's and selectors, an s-op is any arbitrary vector which can be either numeric or categorical. The name s-op is an abbreviation for *sequence operation* as RASP is a lazy functional programming language, operating over functions rather than explicit sequences. A selector (Fig 3.1) is a confusion matrix given 2 s-op's and some predicate function that maps from the pair of s-ops to a given value. Selectors are only returned by the select operator. When compiled these select operators form the first half of an attention head, where the two s-op's provide the keys and queries.

The value matrix is determined by whichever operator is applied to the selector, either *aggregate* or *selector width*. Selector width (Fig 3.1) is the simpler of the two and performs a summation of each column in the selection matrix, resulting in a new s-op where each value corresponds to the predicate for the corresponding element in the key vector applied to every element in the query vector.

The aggregate operator (Fig 3.1) performs a similar aggregation, but takes an additional s-op *'s'* which acts as a weight to be applied to the keys, then rather than taking the sum for each key applied to every query, the mean over queries is taken. In essence, selector width behaves similarly to a histogram while aggregate behaves similarly to a weighted average. It's worth noting that the summation and weighted average require a 2-layer MLP after the attention head to perform the remaining computation and map the outputs to the desired location in the residual stream.

The exact form of attention employed by RASP is defined as follows:

$$Attention(Q,K,V) = softmax\left(\frac{Q \cdot K^{\top}}{\sqrt{d_k}}\right)V \tag{3.1}$$

A useful summary of how the **K**ey, **Q**uery, and **V**alue matrix interact with the data is as follows:

1. $Q = x \cdot W_q$ - here W_k specifies what information we wish to gather from x which will be stored in Q
2. $K = x \cdot W_k$ - where W_k designates the accessible information in x to be gathered into K
3. $V = v \cdot W_v$ - V filters what information is contained within the softmax term based on what's useful to us for the given sample x

The term $Q \cdot K^{\top}$ computes a similarity score, indicating the relevance of each element to the query element. A scaling is applied to the similarity scores of factor $\sqrt{d_k}$ where d_k is the dimension of the Key projections, which helps normalize the outputs by the number of parameters. Applying the softmax term gives attention weights, which determine the importance of each element in the input sequence with respect to the Query element.

Every operation in RASP that requires attention uses a single attention head, multi-head attention layers are formed when multiple attention operations occur at the same depth in the computational tree, as we can simply run the attention heads in parallel.

Lindner et al. [2023] extend RASP with 2 further operators adding simple mapping utilities, namely *map* and *sequence map*, which use The MLP architecture, described in equation 3.2, to implement an arbitrary lambda over one or two s-op's respectively, "simply because MLPs can approximate any function with accuracy depending on the width and depth of the MLP, Hornik et al. [1989]".

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{3.2}$$

While we're on the topic it's also worth noting that transformers as a whole are provably universal approximations provided a fixed sequence length [Yun et al., 2019]. A major limitation of Transformers and by extension the RASP programming language when compared to other programming languages, is their inability for input dependent loops. You may also question the computational efficiency of RASP programs implemented using a transformer architecture but at the very least they can perform a sort with $O(n^2)$ complexity[Weiss et al., 2021] which is somewhat reassuring, although still slower than $O(n \log n)$.

To summarize, extended RASP contains the following 5 key operators, the computational blocks required for each RASP operator can be seen in figure 3.1 while examples of how each operator work can be seen in figures 3.2 and 3.1.

- **Select** - Applies a predicate on the pairwise product of 2 sequences, resulting in a confusion matrix
- **Selector Width** - Given a confusion matrix computes a sequence corresponding to the sum of each column in the confusion matrix
- **Aggregate** - Given a confusion matrix and a sequence, first multiplies each row of the confusion matrix element wise with the sequence. Producing a weighted confusion matrix, then computes the average of each row within this matrix.
- **Map** - Applies an arbitrary lambda to a sequence - can be arbitrary since the input domain is known
- **Sequence Map** - Applies an arbitrary lambda that takes two parameters to a pair of sequences

	Select	Aggregate	Selector Width	Map	Sequence Map
Attention-Head	✓	✓	✓		
MLP		✓	✓	✓	✓

Table 3.1: Computational blocks required for each RASP operation

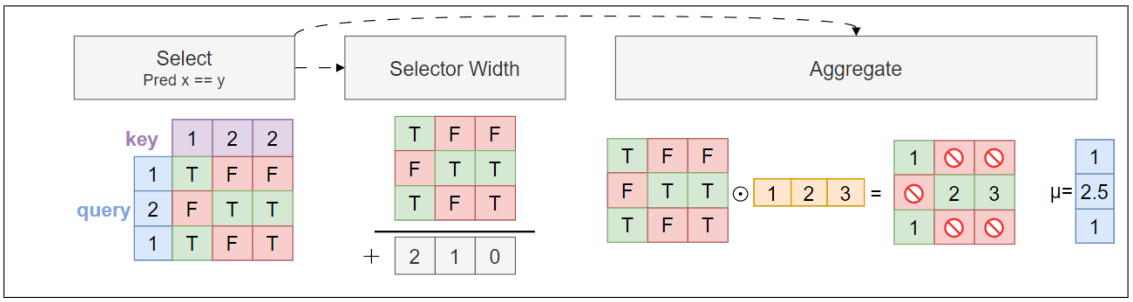


Figure 3.1: Select, Selector Width and Aggregate RASP Operators. The **Select** operation performs some predicate over 2 variables, using an attention head, here is an example of the equality predicate given two sequences '122' and '121'. The **Selector Width** operation computes the sum of columns of a selection matrix, here is an example applied to the confusion matrix we just generated. The **Aggregate** operation computes the weighted average of rows. Here the weights '123' are used, but anything could be used. The averages of the rows are then 1, 2.5 and 1.

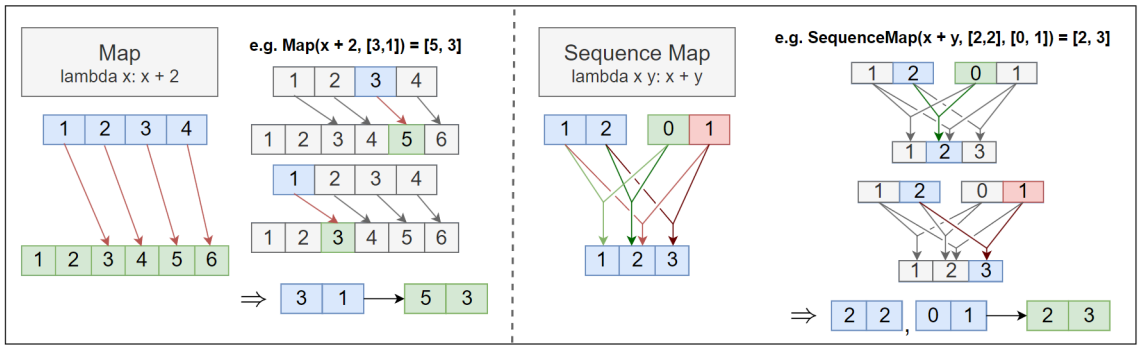


Figure 3.2: Map and Sequence Map RASP Operators. In each case, the first diagram expresses the mapping between inputs and outputs that the layer uses to memorize the lambda. The second diagram is an example of this mapping being used for a sequence of inputs.

3.2 Tracr - Compiled Transformers... for Interpretability

The main contribution from Lindner et al. [2023] is the ability to deterministically compile RASP code into a transformer. The first concept they introduce is the residual stream (courtesy of Elhage et al. [2021]). This is a vector of constant size from which each block reads from and writes to. This enables residual connections across layers, which corresponds to the current state of the program. In other words, the residual stream allows us to store variables in memory for use later in the program. To aid in understanding the computation within a Tracr transformer program, Tracr encapsulates this residual stream within *named basis directions* which act as labels for every index in the residual stream. Blocks then read from a subset of these basis directions and write to another subset.

In order to compile the blocks that perform the computation within a Tracr transformer program, a number of steps are taken:

1. Convert the RASP program into a computational graph, where each node is a RASP operation.

2. Infer the domain and range of each RASP operation in the program.

3. Translate each node into a CRAFT block that implements the operation with an attention head and/or MLP.

4. Compile CRAFT blocks into a CRAFT model - combining attention heads and MLP's.

5. Translate the CRAFT model into a Jax model.

3.2.1 Worked Example - Histogram

To best explain these steps, let's consider a simple example where we wish to compute a histogram over the input tokens, so for each input token, we output the frequency of that token in the input e.g. "hello" -> $[1, 1, 2, 2, 1]$. If we wished to implement this in python the code would look something like this:

```python
tokens = list('hello')
def hist(tokens: str):
    same_tok = np.zeros((5,5))
    for i, xi in enumerate(tokens):
      for j, xj in enumerate(tokens):
        if xi == xj:
          same_tok[i][j] = 1
    return np.sum(same_tok, axis=1)
# e.g. hist('hello') = [1,1,2,2,1]
#      hist('aab') = [2, 2, 1]
#      hist('abbcccdddd') = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

Figure 3.3: Python Histogram Program that computes the frequency of tokens present in the input

The corresponding RASP program is much simpler:

```python
def hist(tokens):
    same_tok = Select(tokens, tokens, Comparison.EQ).named("same_tok")
    return SelectorWidth(same_tok).named("hist")
# e.g. hist(list('aab')): same_tok = [[1, 1, 0],   =>  hist = [2, 2, 1]
#                                     [1, 1, 0],
#                                     [0, 0, 1]]
```

Figure 3.4: RASP Histogram Program that performs the same algorithm as the python implementation in Figure 3.3. we first compute a confusion matrix of the pairwise equality product over the tokens, then by summing each column in this matrix the frequency of each token is obtained.

The computational graph or '*RASP model*' is formed by assigning a node to each RASP operator and a directed edge to each RASP operator from it's operands in the program. In

our example this graph is very simple as the select operation has a single unique operand, tokens, as does the selector width operator, which depends only on the select operation.

$$tokens \longrightarrow same\_tok \longrightarrow hist \tag{3.3}$$

Next, the basis directions for each node in the computational graph are inferred, each operator is applied to every element in its input space and the range of the function is stored and used as the domain for later operations. By propagating the space of possible values through the program, we can assign an element in the residual stream to correspond to the binary encoding of each value in the domain and range of every operation. Each of these axes is assigned a name corresponding to the named operator, e.g. `tokens`, `same_tok` or `hist`, as well as the corresponding value for that axis. For instance the basis directions for the tokens s-op are `tokens=h`, `tokens=e`, `tokens=l` and `tokens=o`. The complete set of named basis directions in the residual stream follow in figure 3.5.

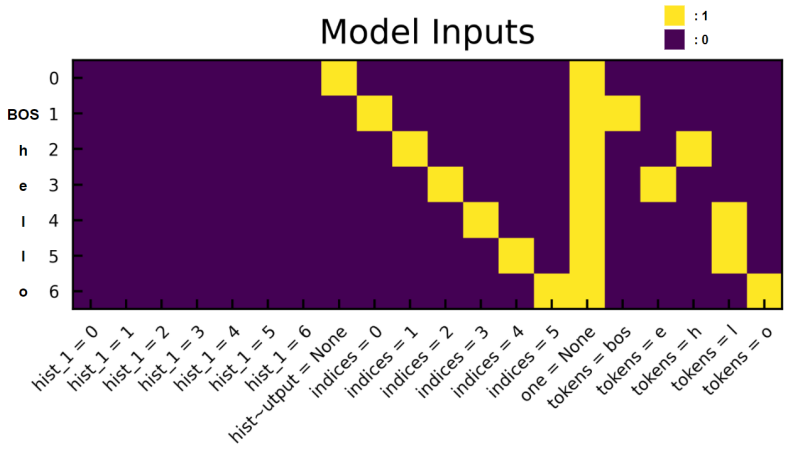


Figure 3.5: Initial state of the residual stream encoding the inputs "hello" for the histogram program. The input space with named basis directions is labeled on the x-axis. In the y-axis are the input time steps. The 'indices' directions are a onehot encoding of the index of each input timestep, the 'one' direction is unit functioning similarly to inputting a ones axis to an MLP to remove the need for explicit bias terms. The 'tokens' directions encode which token is input at each timestep, `[blank, BOS, h, e, l, l, o]`. The remaining directions will be written to during program execution and store the outputs of the program.

Afterward, each node in the computational graph is compiled into an attention head and/or MLP using an intermediate representation called CRAFT. CRAFT allows us to represent the variable sizes of attention heads and MLP's exactly as well as maintain named basis directions at the inputs and outputs of every block. The details of how the computational graph is compiled into a CRAFT model are beyond the scope of this literature review but in summary the CRAFT compiler describes how to map each operator

applied to a given input type (numerical or categorical) to the parameters of an attention head and/or MLP. For simple operations like Map or Sequence Map, these compiled parameters need only map values between inputs and outputs (Figure 3.2) since the domain and range of each operation have been established earlier and the mapping function is known.

The first operator in our example program is the select operation, which outputs a confusion matrix. Given that the two inputs Q and K are the same ("hello") and the equality predicate is being used, the confusion matrix from applying the attention head with parameters W_{QK} will be $Q \times W_{QK} \times K^{\top}$. In Figure 3.6 the diagonal is 1 for $x >= 2$ as each token is the same as itself, while $(4,5)$ and $(5,4)$ are also 1, from the two occurrences of 'l' in the input. At this point the Select operation has concluded and we move on to the selector width operation. The SoftMax activation is applied and the $W_{OV} \times V$ matrix selects the 'ones' column as the output. You can see how in this context the SoftMax operation actually computes the sum of original confusion matrix in the rows axis.

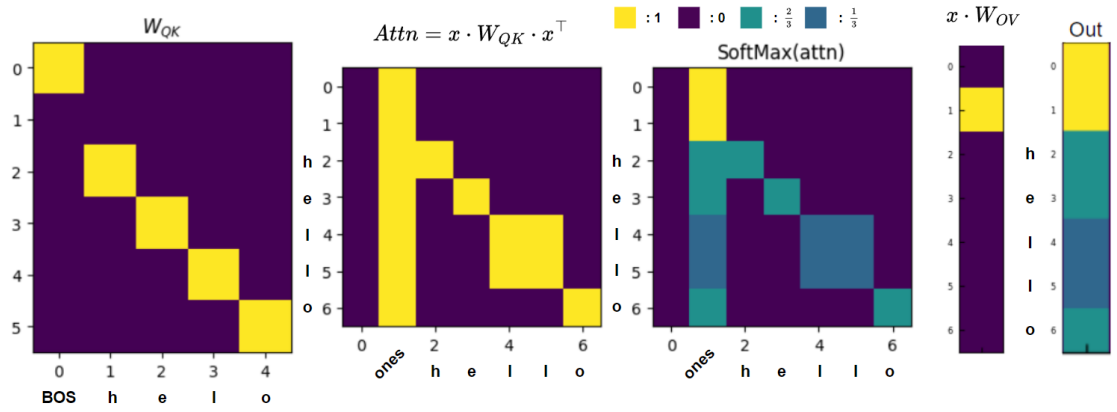


Figure 3.6: Select operation, using an attention head to compute the predicate 'equals' between the tokens, resulting in a confusion matrix. The Query, Key parameter is applied to the token inputs giving the 2nd figure. Applying softmax causes the ones column to act as an inverted accumulator, where $\frac{1}{2}$ corresponds to a token frequency of 1, and $\frac{1}{3}$ corresponds to a token frequency of 2. The Value parameter times the inputs, causes just the ones column with the inverse accumulated outputs to be kept

The outputs now contain a scalar encoding of the histogram values over our input tokens, however, we wish for them to be one hot encoded, which is the job of the MLP.

The first MLP layer matrix has a bar of 100's and below that a scale that exponentially decreases from -15 to -75. The result is the same scale multiplied by the attention outputs, such that the two rows corresponding to 'l' are 2x the rows corresponding to 'h', 'e' and 'o'.

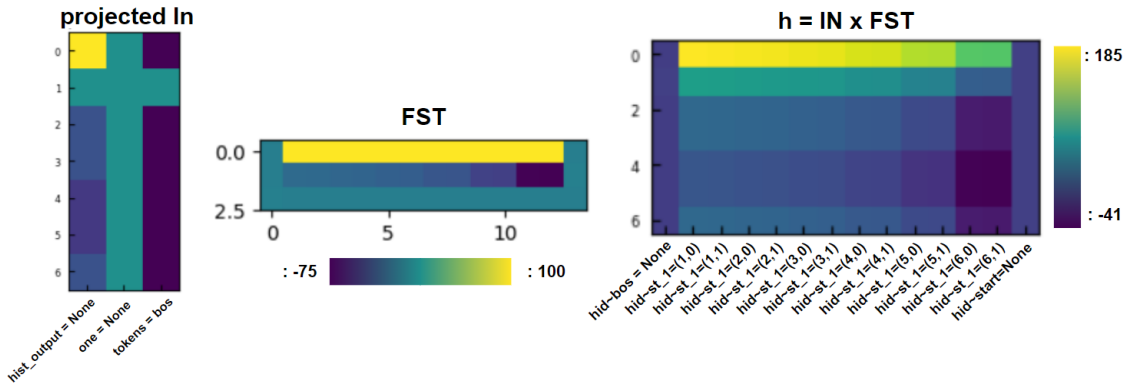


Figure 3.7: Inputs to first MLP layer on left, the first layer applies a gradient, resulting in the gradients in the ouptuts h

Next, we apply a ReLU activation to the outputs and then multiply by the second layer parameters, whose alternating checkerboard pattern cause the signals from incorrect indices to cancel leaving just a one-hot encoding of the frequencies of the input tokens.

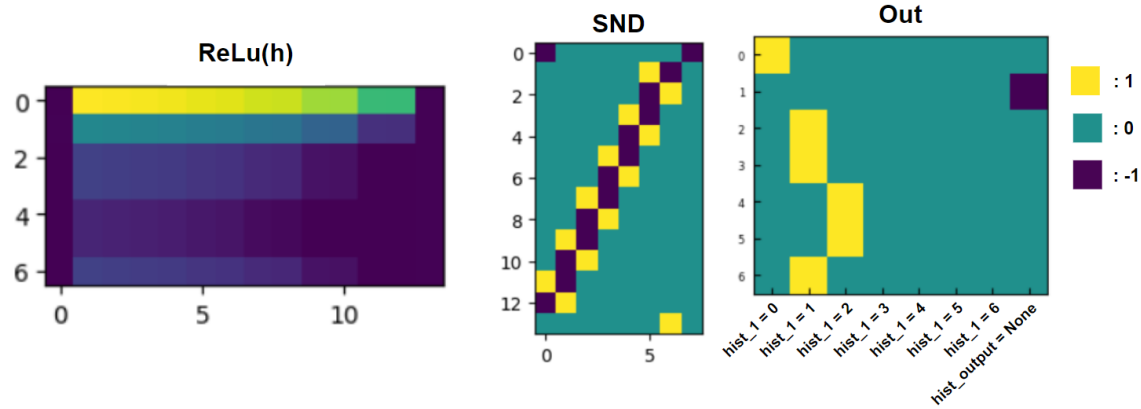


Figure 3.8: First ReLU is applied to the outputs of the first layer, then the alternating checkerboard pattern causes signals in the gradient to cancel out, resulting in the one-hot encoding of the token frequencies on the right

To conclude we have just worked through how to convert a simple RASP program into a functioning transformer program that implements the same program, as well as studied the parameters that are compiled to achieve this. The final contribution from Tracr that we use is another compiler step that compiles the CRAFT parameters we've seen up to this point into a JAX transformer, however, this is a relatively trivial case of copying the parameters and padding with zeros such that every key, query and value weight matrix are the same shape.

3.2.2 Tracr Compilation

An integral part of the CRAFT compiler within Tracr is a method of combining attention heads and MLP's into the same block. For instance, consider Figure 4.2, the program branches into two independent computations, $Select \rightarrow SelectorWidth$ and $Select \rightarrow Aggregate$, both of which require an attention head and a 2 layer MLP. Due to their independence they can be run in parallel. To run the attention heads in parallel is trivial, the heads can simply be combined into a multi-head attention layer with 2 distinct attention heads, resulting in a W_{QK} and W_{OV} matrix of double the width. MLP layers are bit more nuanced, however thanks to the construction of the residual stream each MLP writes to mutually exclusive regions of the residual stream. Consequently, if an extra projection matrix is introduced to their outputs to ensure alignment with the correct residual stream section, the MLP parameters can be concatenated. This projection matrix can be multiplied into the second layer's parameters resulting in a single, two layer MLP that implements both the selector width and aggregate operations and writes the result to the correct regions of the residual stream.

3.2.3 Deterministic Decompilation

During our investigation Friedman et al. [2023] published their research on deterministically decompiling Tracr into Python. This is a very interesting approach, enabling them to use Python development tools and automatic analysis to decode the operations performed by a transformer program. Their approach focuses on the constraints present within Tracr compiled transformer programs to enable deterministic decompilation. Much as we decoded the operations performed within a compiled transformer program manually in Section 3.2.1, they consider each block individually to isolate its computation. Mapping operations are the simplest to decode, since they are implemented as maps from the input to output space. The parameters can be decompiled into a simple dictionary that performs the same operations as a Map or Sequence Map operation.

Select operations are a bit more technical, but since the only variable that affects the functionality of an attention head is the selection predicate, the mapping operation performed by the predicate can be decoded in a similar manner to Map operations; resulting in a dictionary that can represent an arbitrary predicate. The remainder of a select operation is then hard-coded as a subroutine. Similarly, Selector Width operations do not vary in functionality so are also hard coded. Finally, they just needed to decompile the aggregate operation, however much like the Selector Width operation, its functionality does not vary either, only its inputs do. Once the inputs are decoded by considering the residual stream they can be passed through a hard-coded aggregate block.

In conclusion, their decompiler is very useful to interpret the computation of a Tracr transformer program with python code. But does not offer any mechanism to generalize to less synthetic models.

Chapter 4

Program Generation

To train our *"Standard Inverse Tracr Meta-model"* we need a large number of example transformer programs and their corresponding RASP programs to learn from. Such a dataset of labeled models is called a model zoo. Since Tracr provides the ability to compile these transformer programs from RASP we just need to generate a dataset of RASP programs so we can compile them into transformer programs. In this section, due to the lack of existing research on how to efficiently generate random programs, we devised a simple Monte Carlo program sampling algorithm.

We can restrict every line of our program to contain a single RASP function call with some number of arguments and store the result in a unique variable. To generate the program itself we can repeatedly sample RASP operations and randomly assign arguments with matching types that are currently within scope or generate constants when appropriate. An overview of the algorithm is detailed in Figure 4.1.

```
initial_scope = {tokens, indices}
operations = []
for n in range(0, n-1):
    op = sample_rasp_operator(scope, RASP_OPS) #Sample a new function
     ↪  to add to the program
    operations.append(op)

def sample_rasp_operator(scope, RASP_OPS):
    op = sample(RASP_OPS)
    switch op:
        case Map:
            lam = sample(Categoric_Lambda | Numeric_Lambda)
            if lam is categoric:
                return Map(var(SOp), gen_const(CAT_OR_NUM), lam)
            elif lam is Numeric:
                return Map(var(SOp), gen_const(NUM) + noise(), lam)
        case SequenceMap:
            lam = sample(Numeric_Lambda)
            v1, v2 = vars(2, SAME_TYPE)
            return SequenceMap(v1, v2, lam)
        case Select:
            pred = sample(Predicate)
            v1, v2 = vars(2, SAME_TYPE)
            return Select(v1, v2, pred)
        case Aggregate:
            v1 = var(SELECT)
            v2 = var(Numeric)
            return Aggregate(v1, v2)
        case SelectorWidth:
            return SelectorWidth(var(SELECT))

```

Figure 4.1: Simplified RASP Program Generation Algorithm. `Var(X)` samples a variable of type X from the current scope. `vars(2, SAME_TYPE)`
samples two variables of the same categoric/numeric type within the current scope.

As the program grows, the number of (return) variables does so too, and the space of possible RASP programs grows exponentially. Our program generation operates with 2 main data types, numerical (ints & floats) and categorical (vocabulary of strings). This is due to the different behavior of lambdas depending on the input type, for instance, 1 + 1 = 2 yet '1' + '1' = '11'. We also faced the problem of how to sample constants such that the output of lambdas would have some variety, we denote this as the entropy of the program. For instance with inputs in the space $[1,2,3,4,5]$ a lambda $x < 6$ would have infinite entropy as the entire output space would be *True*. To mitigate this we sample constants from the same space as the inputs, for instance $x < 3$ retains much more information. To increase the search space we also occasionally add noise to numeric constants. This is a very crude mechanism to maintain the entropy, given more time we would have liked to track the domain of every variable during program construction such that operations can be chosen more appropriately. For instance after just a single operation that adds 3 to the domain $[1,2,3,4,5]$, the lambda $x < 6$ seems far more reasonable with minimal entropy. Indeed we do find that, especially for longer programs our crude generation algorithm tends to produce programs with very high entropy at the output. However for our purposes this does not cause many problems as we are primarily studying the program that the parameters of a transformer program implements at a line by line level, rather than treating the model as a black box with some input being mapped to some output.

We sample from a small number of lambdas (Table 4.1) for basic mathematical operations that get used by the Map and Sequence Map operations, we tried to minimize this variation as it is known to be difficult to decode the operations that an MLP block is performing Nanda et al. [2023].

RASP OP	Categoric Lambda	Numeric Lambda	Predicate
Map	$x < y$	x + y	EQ
Sequence Map	$x <= y$	x * y	FALSE
Select	$x > y$	x - y	TRUE
Aggregate	$x >= y$	x or y	GEQ
Selector Width	$x! = y$	x and y	GT
	$x == y$		LEQ
	not x		LT
			NEQ

Table 4.1: Relevant primitives that the program generator samples from

We also preferentially sample parameters from operations that are later in the program, where sampling weights follow the recursive geometric sequence $w_n = w_{n-1} * 2 - w_{n-2} + 1$.

After a number of operations have been sampled you are left with a directed acyclic graph where any given leaf can be chosen as the return variable. You can think of the program as a kind of tree with 2 roots, the tokens and indices (Figure 4.2). However, only ancestors of the chosen return variable influence the program, any non-ancestral nodes will be spurious and independent of the output of the program. Accordingly, once these nodes have been pruned the true length of the program is frequently shorter than the number of operations that we originally sampled.

In order to generate programs of reasonable length we have to minimize the number of non-ancestral nodes in the computational graph. First, the aforementioned preferential sampling of parameters goes some way to help constrain a program, by introducing a bias towards adding nodes to the deepest nodes (measured from the root). Additionally, we also post hoc compute the longest path from the root within the computational graph and choose the terminal node in this path as the return variable. This still results in programs that are on average 30% shorter than the specified target program length, but since our Monte Carlo algorithm is computationally cheap, we simply rejection sample to ensure a program of adequate length is generated.

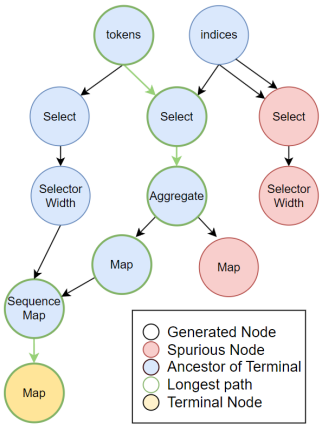


Figure 4.2: Pruning process and terminal node selection after sampling operations

In practice, we had to make the process of generating RASP programs as efficient as possible as a large number of these programs would be required to train the meta-model ($\sim$800,000). After implementing our algorithm we found that the Monte Carlo program generation algorithm we devised earlier indeed functioned as expected being able to construct a program in $\sim$10ms $\pm$ 2ms (including rejection sampling). However, compiling this model into a CRAFT model was far less reliable, for a program with 30 instructions it takes around 200ms on average but up to 15s, or even not terminating in some cases, and in others running out of memory. We also noted that the upper quartile of compilation times fits the trend of $t = 0.2 + (\frac{L}{18})^4$, where L is the number of instructions in the program. Accordingly, we wrote a simple timeout script that terminates compilation if the time taken exceeds that of 60% of previous programs.

We chose this strategy as it is independent of the chosen program length and machine performance, as we know that the timeout issue happens 40% of the time, and we'll be generating many programs in secession (We initialize with the timeout formula). We also had to protect compilation against running out of memory, thankfully this is very rare and when this issue occurs the compiler attempts to allocate an absurd amount of memory in one go without ever allocating it and causing the machine to run out of memory for other processes, so we could safely catch the exception and try again. Since we're able to generate a sample on average in 0.2 seconds, when multi-threaded on a machine with 12 threads, we were able to achieve 55 samples per second on average. This is not fast enough to generate synchronously during training but fast enough to compile a dataset of 800,000 programs in under a day.

4.1 Example programs

```
def example_program_1(tokens, indices):
    v1 = Select(PRED_NEQ, indices, tokens)
    v2 = SelectorWidth(v1)
    v3 = Select(PRED_LT, v2, v2)
    v4 = SelectorWidth(v3)
    v5 = Aggregate(v3, v4)
    v6 = SequenceMap(LAM_ADD, v2, v5)
    return Map(LAM_LE, v6)
```

Figure 4.3: A randomly sampled program generated using our algorithm, containing 2 attention heads and 2 map operations requiring MLP's

```
def example_program_2(tokens, indices):
    v1 = Map(LAM_SUB, indices)
    v2 = SequenceMap(LAM_SUB, tokens, tokens)
    v3 = SequenceMap(LAM_MUL, v1, v1)
    v4 = Map(LAM_OR, v2)
    v5 = Select(PRED_TRUE, indices, v2)
    v6 = Aggregate(indices, v5)
    v7 = Select(PRED_LT, v3, v6)
    return Aggregate(v4, v7)
```

Figure 4.4: Another randomly sampled program generated using our algorithm, containing 2 attention heads and 4 map operations requiring MLPs

Chapter 5

Meta Model

In this chapter, we will discuss how we trained our standard inverse Tracr meta-model to be able to decompile standard transformer programs. This involves establishing representations for transformer programs and RASP programs compatible with our meta-model. As well as developing a differentiable loss function for evaluating the quality of our predictions. Finally, we optimize the architecture and training hyperparameters to maximize the performance of the meta-model on our model zoo. We were able to predict 73.04% of tokens correctly but struggled at reconstructing whole programs correctly.

5.1 Input and Output Representation

Now that we have a RASP program and its corresponding transformer program we face the problem of how best to encode the transformer program for input to our meta-model and how to represent the RASP program as the output of the model.

5.1.1 RASP program representation

We'll first consider how to represent the RASP program. A typical program consists of an ordered list of operations and operands, however, under this encoding, the same program can be written many ways, depending on the order of operations and for some functions the ordering of parameters. To compensate for this we can represent the program as a computational graph, where operands are nodes in this graph and operations are edges. Now a program can be represented as a set of operations and operands and an ID denoting the return variable. For instance the following program:

```
1 def hist(tokens):
2     same_tok = Select(tokens, tokens, PRED_EQ)
3     return Aggregate(sel)
```

has a graphical representation:

$$tokens \longrightarrow same\_tok \longrightarrow hist \tag{5.1}$$

which can be written in the form of operation, argument(s), return variable. Where variables act as directed edges connecting nodes in the graph.

```
1    Select, ==, tokens, tokens, v1
2    Aggregate, v1, v2
```

Figure 5.1: Textual Graph Representation

Line level loss function

Given this graphical representation, we must devise a differentiable loss function to measure the difference between this computational graph and another target computational graph. First, we must establish a distance function between any two nodes of the graph, thanks to the textual representation in figure 5.1 we can treat the input as text, with each operation, argument or return variable being a token. However, this would be inefficient as the one-hot encoding of each token would be the number of operations and possible arguments combined equaling around $53 + L$ possible tokens (where L is the max program length, since there is 1 variable per operation), so up to $5 \times (53 + L)$ binary digits in total. Additionally, this representation would have variable length. We can simplify the problem greatly by padding the arguments with NA, as the most number of arguments in a program is 3.

```
1 Select, ==, tokens, tokens, v1
2 Aggregate, v1, NA, NA, v2
```

Now we have a fixed length representation where the first token is always an operation, the next 3 are either a lambda, predicate or variable, and the final token is always a variable. This has the added benefit of greatly reducing the search space as we need only consider the 5 possible operations in our first token, the $48 + L$ possible arguments in our next 3 tokens and L tokens in our return argument. Meaning that we've reduced the search space from $265 + 5L$ to $149 + 3L$, nearly half. Following convention we can use the cross entropy loss to compute the difference between any two lines, since we have 5 tokens, we have to apply the cross entropy loss 5 times, we then take the sum of the result. To summarize, when comparing any two lines, y_p and y_t of a program, we encode each line into a binary string with $149 + 3L$ digits and split into 5 segments, then compute the cross entropy loss between segment 1 in y_p and segment 1 in y_t plus the cross entropy loss of segment 2 in y_p and segment 2 in y_t... etc, up to the fifth and final segment.

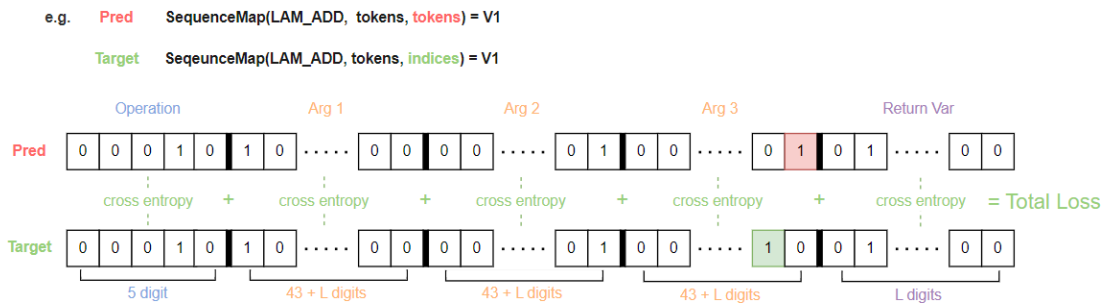


Figure 5.2: Segmented cross entropy loss between program lines - Above there are example predictions and targets where we are incorrectly predicting arg3 as tokens, below are their encoded representations with 5 segments corresponding to one hot encodings of the operation, 3 arguments and return variable. The cross entropy loss is then taken between each of these segments to measure the distance between the predicted and target sequences.

The issue with this representation is that our arguments could have any ordering which would result in false negatives when the prediction is actually correct but the wrong permutation was predicted. Rather than considering all the possible permutations of predictions we instead enforce an ordering over the parameters, Lambdas > Predicates > Variables, and sort each subcategory alphabetically. This gives us a single unique representation for any line in a program. The only caveat is that this representation requires our model to learn the ordering, nevertheless, in our experiments, we could not find a single case where the meta-model failed to predict tokens that followed this ordering.

Handling program permutations

The second, and equally important aspect, is ensuring the loss function is permutation invariant. Thanks to our graphical representation, any ordering of the program is valid, even given a standard program representation, if the computational graph branches as any point, there will be multiple possible representations of the same graph. Ideally, we wish to find the minimal cost bijective mapping between a predicted program and a target program. However, there is very little literature on such a permutational invariant loss function, but a brute force method would be to consider all possible bijective mappings and for each, compute the sum of the distances between corresponding lines, then return the total distance of the bijective mapping that has the minimal loss. However, this approach is highly computationally expensive, as there exist 15! mappings between two programs of length 15. To solve this we again turned to a canonical ordering over programs such that any given program has a unique representation.

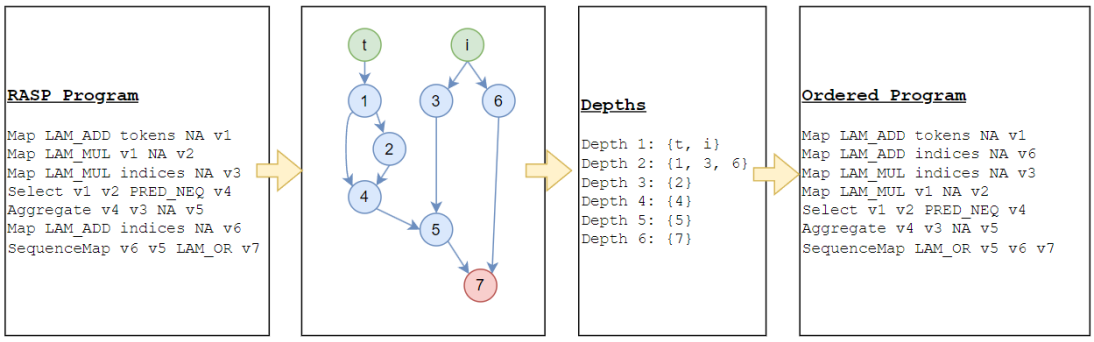


Figure 5.3: Computational Depth Program Ordering - given a program we construct the computational graph, compute the depths of each node in the graph w.r.t. the tokens and indices, allowing us to order the program by computational depth, breaking ties alphabetically

The question now is how to generate a canonical ordering of a program? An arbitrary sorting would suffice but small mistakes would get amplified, for instance if a program had the operations `Aggregate > Map > Selector Width` but the first operation was guessed incorrectly as a `Sequence Map`, under an alphabetical ordering the new program would be `Map > Selector Width > Sequence Map`, and it would appear that the model predicted every single token incorrectly when in reality it just guessed one token incorrectly. To mitigate this issue we propose an ordering dependent on the depth of the operation within the computational tree, then for nodes of the same depth, sort using an alphabetical ordering.

Under this ordering we found that small errors in predictions were indeed unlikely to cause knock on effects later in the program, despite tie breaker scenario relying on an alphabetical ordering. Again the meta-model must be able to learn a how to represent a program that follows this ordering, but given that the alternative is intractable we thought it was a worthy compromise, and in practice we found that, again the model made no errors in the ordering of it's output program in the samples we looked at, proving that the meta-model was able to discern the computational depth of an operation within a program.

As an initial check that our program encoding could be learned by a transformer we trained a program to program auto-encoder using a high degree of compression in the hidden state, after training, the model was able to reproduce the program with 94% accuracy suggesting that our program representation was being learned effectively by the transformer autoencoder.

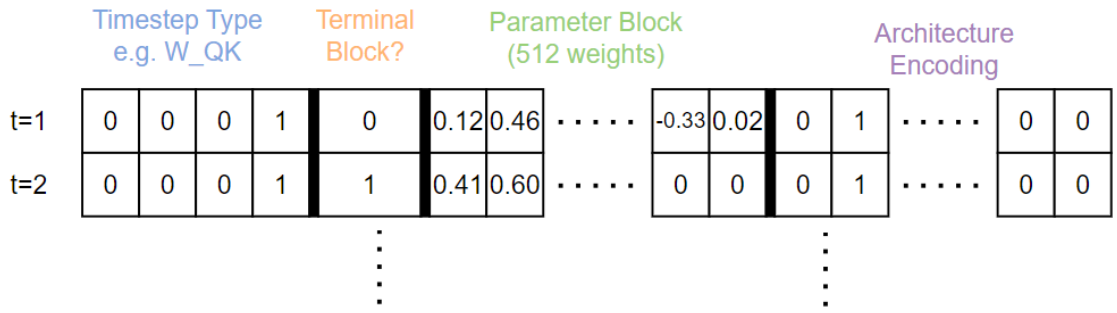


Figure 5.4: Parameter Encoding - How we encode the parameters of the model, each timestep (vertically) describes one block of up to 512 parameters in the model. Each timestep consists of 4 sections. The timestep type, is a fixed length one hot encoding, denoting the type of the parameter described within the parameter block. The parameter block contains the raw weights within the parameter being described, if this contains more than 512 weights it overflows into the next timestep. Only the final parameter block for a given parameter will have terminal block set to 1, so the terminal block indicates whether we have finished with the current parameter. The Architecture encoding is a concatenated string of one hot block type encodings. Since there are only two block types Attention and MLP, each index corresponds to a single block, 0 represents an attention block, 1 represents an MLP block. This varies in length according to the number of blocks in the model being described.

5.1.2 Transformer Representation

Now that we have established the output representation, we must also establish the input representation, that is how to represent the parameters of a Transformer as an input to a meta-model. We drew inspiration from existing works on meta-modeling tasks and chose to flatten input parameters to form "*layer-by-layer tokenizations*" of transformer programs that we will input to our meta-model [Peebles et al., 2022], [Schürholt et al., 2021]. In order to limit the input dimension of our model we defined the maximum token length to be 512 parameter weights, if any given parameter exceeds this length limit the remainder is overflowed to another token until the entire parameter has been tokenised. In order to differentiate different parameters, a terminal digit was added to the end of this sequence, set to 1 if the parameter is the terminal parameter in the sequence. In previous research model zoo's consist solely of the same form of architecture, either MLP's or CNN's, however transformer programs consist of a sequence of multi-head attention blocks and MLP blocks. Accordingly, we added a further token to distinguish the layer type. Finally we also experimented with adding an architecture summary to every timestep, this consisted of a tokens representing the two block types MLP's and MHA's concatenated together, forming a string of variable length that encoded the high level architecture.

5.2 Meta Model Training

In this section we'll discuss the meta-model, the model that learns how to map from the transformer program representations we've been generating, to the RASP program that they implement.

First we must establish the best suited architecture for our problem. We need to be able to interpret a variable length sequence of inputs, this immediately rules out MLP's (Multi layer perceptrons) and CNN's (Convolutional Neural Networks) leaving recurrent neural networks and transformer models. Since transformer models show the most promise in the current landscape of the field we chose to continue with transformers. Nonetheless, we can have input layers that contain MLP's or CNN's. Since the input data is image like, consisting of a matrix of correlated values, using a CNN would seem to be appropriate at first glance. Yet in reality the parameters of a transformer program, although correlated are not **spatially correlated** meaning that the filter's used by a CNN would be inefficient at extracting the relationships within our input. We also decided against an MLP input layer as typically transformer models are used without them [Vaswani et al., 2017][Radford et al., 2019]Brown et al. [2020].

Since transformer models function like a fully connected graph neural network, they do not have an embedded representation of the timestep of each input since they are called on all input timesteps simultaneously. Accordingly we had to add a positional encoding to ensure that the ordering of our inputs is preserved since we input parameters into the model in the order that they are executed by the transformer program.

At the time of writing 3 main transformer architectures were available to us with support for JAX[2], GPT2[14], GPTJ[20] and GPTNeo[1] however all of which have nearly identical architectures with the only exception being that GPTJ uses a custom rotary positional encoding applied to the key and query heads. We found that all performed equivalently on our dataset, and the only important metric was the number of parameters in the model. Due to this we chose to solely use GPT2 as it is the best known of the three.

Since GPT2 is a large language model it is typically used combined with a text embedding layer that encodes text input tokens. Since our input data is numerical we discarded these embedding layers and solely used the attention blocks for each architecture. Similarly we trained each model from scratch since there are no pre-trained models suited to our task.

Modern LLMs have been optimized for language modeling so typically use causal masking. Causal masking simulates a transformer being ran one input timestep at a time, such that at every output timestep the model does not have access to future input time-steps. This is particularly useful for textual entailment tasks where you don't want the model to be able to cheat and have access to future targets. If we are to use causal masking in our

modeling we would have to ensure that the predictions are output at some timestep greater than the inputs upon which it depends. Since the decoded program depends on the entire input transformer program, in order to allow the model to consider the entire input before outputting the program, the output must be predicted at a timestep later than the final input timestep. An alternative is to simply disable causal masking within the model and allow every output timestep to depend on every input timestep, however in order to minimize the changes we made to the default LLM architecture we chose to keep causal masking and offset our prediction timesteps from out input timesteps, so we can 'see' the entire program before making the first prediction; as described in figure 5.5. This allows us to more easily compare different architecture sizes without fine-tuning hyperparameters but does increase the computational cost by a constant of L.

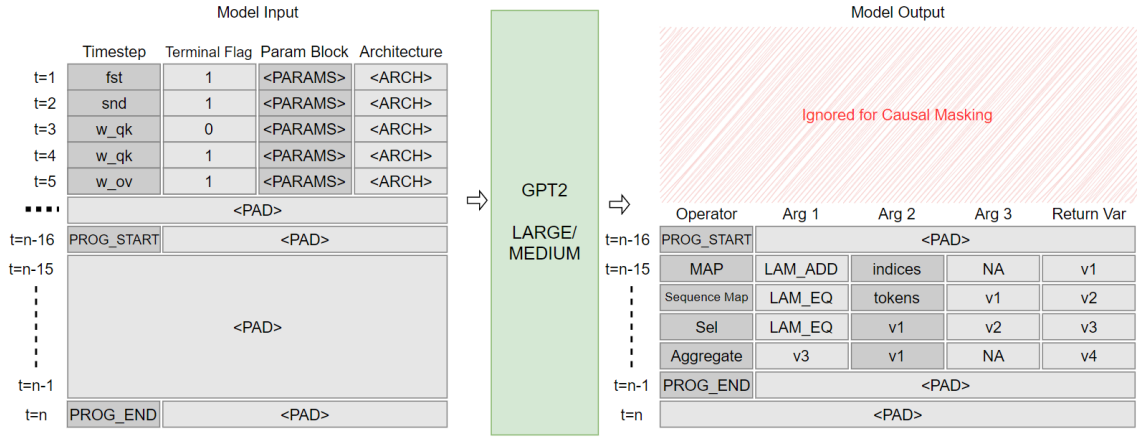


Figure 5.5: Causal Masking input and output format - On the left we have the first 5+ timesteps encoded according to Figure 5.4, after some padding that accounts for varying sequence length within batches, we allocate 17 timesteps for the program. On the output side, the first $n-17$ predictions corresponding to the input data and padding are discarded, then the final 17 timesteps are populated with the program prediction according to the format in Figure 5.2

Also note in figure 5.5 the program start prompt in the input as well as the padding that proceeds it. These tokens were added due to the variable number of time steps within a batch. As a result, we designated the last 15 time steps for RASP program prediction, with padding space allocated before them. If the program start flag was not added to the input the model would struggle to determine the length of padding within the input.

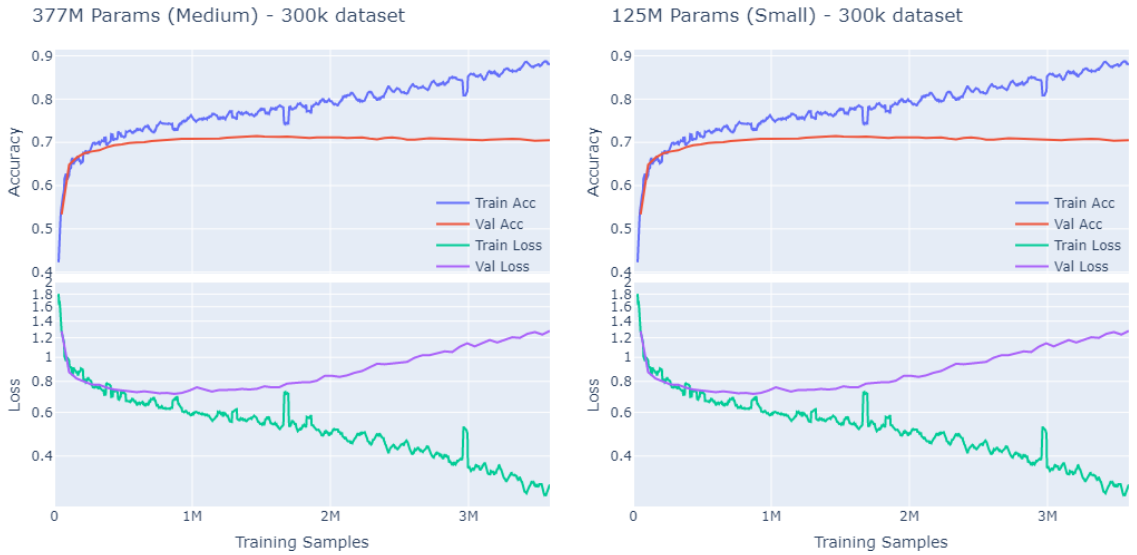
Similarly, we also sandwiched targets between program start and program end tags, since Vaswani et al. [2017] found that the beginning of sequence tokens help improve performance. The end of program token in the output is also required as programs have variable length.

5.3 Dataset Summary

Alongside the development of the meta-model, we also iterated the size and configuration of the dataset. In all our tests we used a program length of 15, and a validation split that was 10% the size of the training dataset; using this validation split to test generalization performance. Initially, we believed that 300k samples would be enough to train the meta-model, and we only generated categorical inputs as this made program generation simpler since we only had to store strings as the input vocabulary of the transformer program. As we progressed through development we realized that 300k samples would not suffice so generated an additional 600k samples giving us a very large dataset with 900k samples. Despite this the dataset was still slightly overfitting we decided to increase the size of the program search space by enabling numerical inputs to the transformer program.

5.4 Architecture Size Selection

At this point, we began training models to decompile Tracr transformer programs back into their generating RASP code. Given our 300k sample standard dataset, we had to establish the correct size of the model to use, we iterated through 3 different sizes of models; small, medium, and large. We began with GPT Medium a 377M parameter architecture, however, this overfit, with the validation loss diverging from the training loss after 1M samples (Figure 5.6a). Accordingly, we decreased the model size to small with 125M parameters, which also overfit in a similar manner (Figure 5.6b).



(a) GPT Medium + small dataset

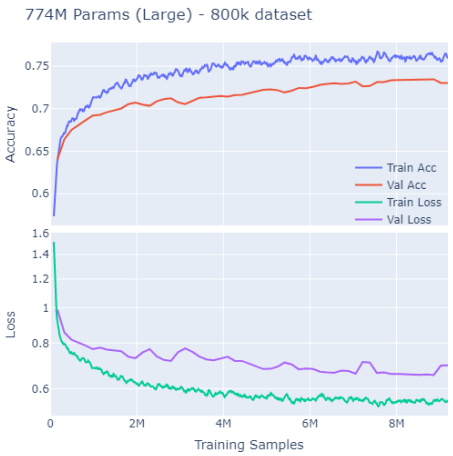
(b) GPT Small + small dataset

Figure 5.6: Training curves for small/medium architectures on the small transformer program dataset, in both cases the models are overfitting with characteristic divergence between the training and validation loss, because the training dataset is too small

At this point, we increased the size of the dataset to 900k samples, which increased training time to the extent that we needed to use an A100 GPU for reasonable training times.

Since A100's have 80GB of VRAM this also enabled us to experiment with GPT Large with 774M parameters. After training this larger model on the larger dataset it only overfit slightly (Figure 5.7a) so we decided to enable numerical inputs to the transformer dataset and retrained the same model. On this new large dataset with greater input variability, GPT large did not overfit and the performance was very reasonable Figure 5.7b. Achieving 73% (token level) accuracy on the validation split.

(a) GPT Large + dataset only categorical inputs



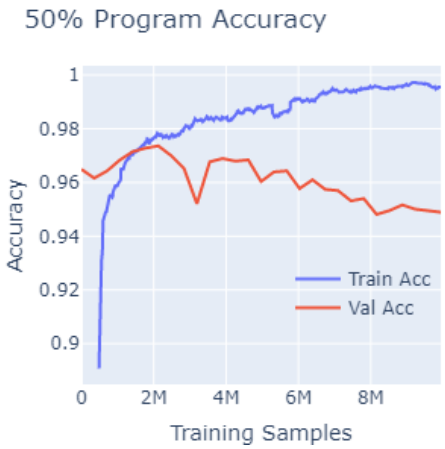
(b) GPT Large + dataset with Numeric inputs

Figure 5.7: Training curves for GPT Large architectures on the large transformer program dataset, both showing strong generalization performance to the validation set, however adding numeric inputs enabled stronger correlation between the training and validation loss, reducing overfit

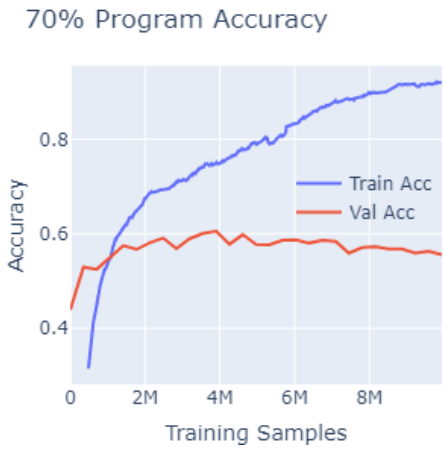
5.5 Standard Meta Model Results

During training we tracked a variety of metrics to measure the performance of our models, first we logged the loss and token level accuracy, that is the percentage of tokens predicted correctly in the output as shown in Figure 5.7b. The accuracy measure of our best model on our final dataset is 73.04% (token level accuracy). We also tracked the *whole program accuracy* (Figure 5.8d), that is the percent of the time the program was predicted entirely correctly. We recorded this on both the train and validation split throughout training. Our *whole program accuracy* is much lower than expected, just 0.3%. Interestingly the whole program accuracy on the training dataset was also low, just 10%, this suggests that our model is not fitting the dataset perfectly, potentially because the

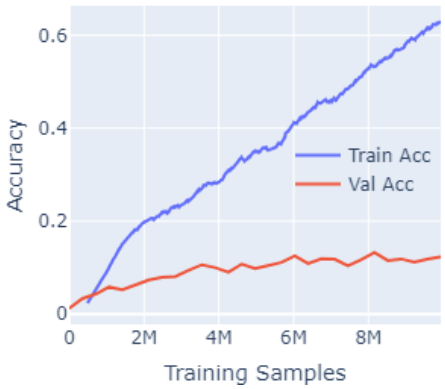
Due to the poor *whole program accuracy*, we also logged the frequency with which we guessed 50%, 60%, 70%, 80% and 90% of the tokens correctly to better gauge the consistency of our predictions. Programs were guessed under 70% correctly almost all the time on the training dataset (Figure 5.8a) and consistently generalized to being 50% correct on 96% of the validation dataset. This suggests that some features are easy to generalize but others present a much greater challenge. The 90% program level accuracy chart (Figure 5.8c) suggests that the model had not yet converged on the training dataset as the training accuracy was still strongly increasing during training. However, all of the validation accuracy trends had converged, suggesting that the model was in fact overfitting on the data.



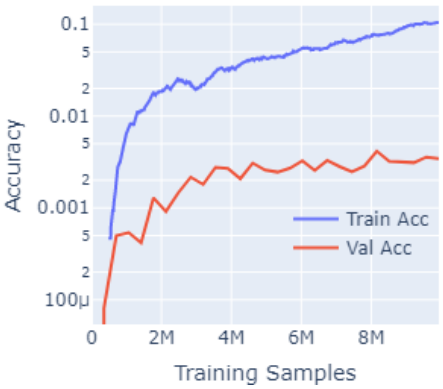
(a) 50% program accuracy



(b) 70% program accuracy



(c) 90% program accuracy



(d) 100% program accuracy

Figure 5.8: Program level accuracy of GPT Large on our Standard Transformer program dataset.
The 100% graph shows the percentage of training and validation samples that were predicted perfectly with 100% accuracy throughout training.
The remaining 50%, 70% and 90% graphs show the percentage of training and validation samples that were predicted with over 50%, 70% and 90% accuracy respectively.

In order to decrease model overfitting on our data we tested two techniques, the first was to augment our data with some noise, we found that adding Gaussian noise with 0.4 times the standard deviation of the input data worked well. We also increased the input dropout percentage from 5% to 30%, which causes the model to be less reliant on any single feature as inputs get randomly removed during training.

These additions did help reduce model overfitting, however, did not lead to a significant increase in performance suggesting that either a larger model is required or a larger dataset.

Figure 5.9: Training metrics of GPT2 Large with noise added to the inputs during training equal to 0.4 times the standard deviation of the input data. As well as dropout increased to 30% in order to try and decrease overfit. However, results are no better than the model trained without these augmentations in Figure 5.7b

In summary, the best model we tested achieved a token level accuracy of 73.04%, while just predicting 0.4% of programs perfectly (5.8d). The model did however, correctly predict 70% of the tokens in a program correctly 60% of the time (5.8b). This allows us to conclude that in the simplest scenario meta-models perform reasonably at inverting Tracr transformer programs but are far from optimal, only being able to reliably extract high-level features; this is more evident in our in-depth analysis of the specific outputs of the meta-model in the next section.

5.5.1 Output Visualization

To give a more quantitative view on exactly what the model is struggling with we also visualized the target vs. predicted program tokens 4 times every epoch.

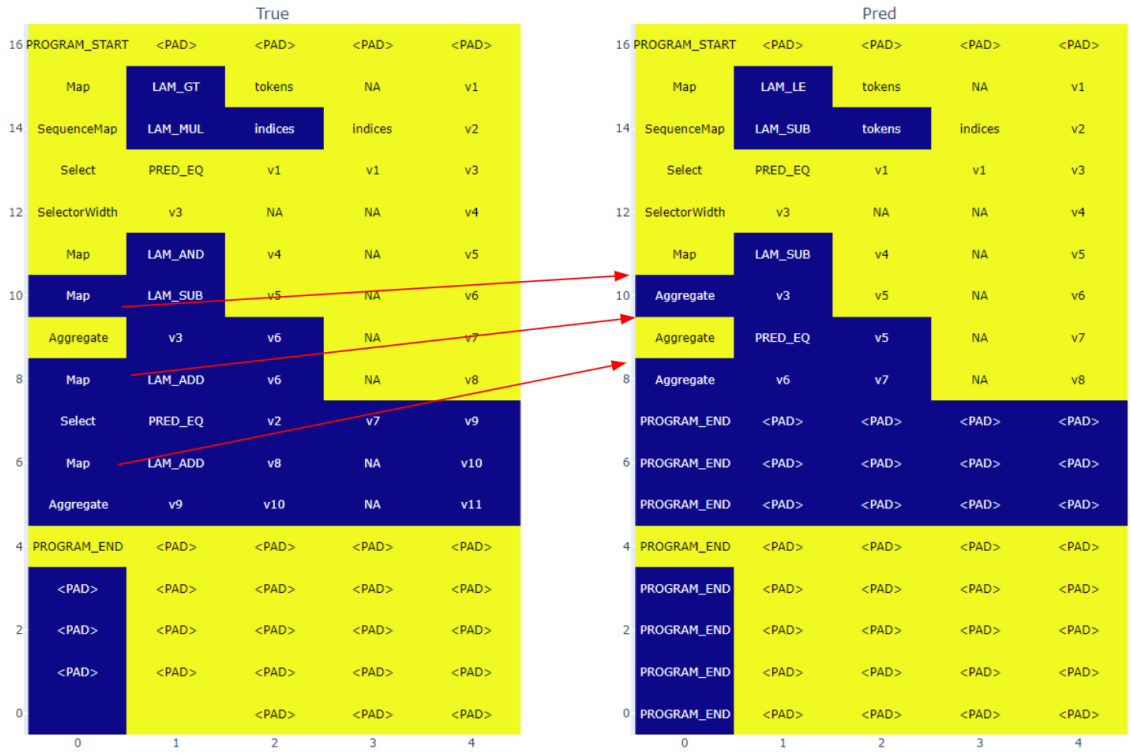


Figure 5.10: Example prediction from trained Tracr Transformer Program meta-model decoder

Figure 5.10 shows a stereotypical prediction that we saw in our trained models. On the left we have the target output, the true program that was used to generate the transformer program parameters that were input into the model. On the right we have the predictions that were made by the model. Cells are colored in yellow when the prediction is correct, and blue when incorrect. In this example the program has missed 3 Map operations that occur at the same depth as aggregate operations. This is a very common occurrence and represents one of the most difficult traits of transformer programs to decode. That is when MLP layers get combined, it is very hard to distinguish which Map operation took place. In this example we have 2 maps that get combined into the same MLP in lines 9 and 10, consequently, the model only predicted one Map operation as it was unable to recognize that there were two map operations applied in series. In position 9 we have an aggregate operation, if you can recall, an aggregate operation consists of both an attention head and an MLP block. In position 8, a map operation is applied in parallel with this aggregation operation, as a consequence, the MLP block for the aggregate operation gets concatenated to the MLP block for the Map operation, and hence disguises the presence of the Map operation. The same thing occurs with the map and aggregate commands in lines 5 and 6.

This theme of struggling to decode Map operations in a compiled program continued throughout all our testing, leading us to conclude that the map operations are in fact the hardest features to decode. Accordingly, their parameters are similarly challenging to decode, we use 7 basic lambdas (Table 4.1) in all of our map operations however logically they are very similar, accordingly these features were also hard to predict correctly even when the meta-model correctly identified a map block. We have summarized the mistakes we found in a sample of predictions from the meta-model in Table 5.1.

It's also worth noting that our ordering of programs established in section 5.1.1 only allows for small errors in the output to be accommodated without cascading errors across the entire program. However, for large errors such as missing an entire line in the program, all subsequent lines will be classified as incorrect by the loss function. This is not optimal but was the only alternative we could find to the intractable task of considering every possible mapping from lines in the predicted program to lines in the target program.

Token Accuracy	Problem
98%	Wrong variables used
95%	Mistook Lambdas, Wrong variables used
95%	Mistook Lambdas
95%	Wrong variables used
90%	Mistook Lambdas, Incorrect Argument Ordering for a select operation
80%	Mistook Lambdas, Predicted Map rather than Sequence Map
50%	Mistook Lambdas, Missed Map, Mistook `Select, Map` for `Aggregate`
50%	Predicted one extra map, Predicted 4 Maps as 1
60%	Missed Map
50%	Extra Map, mistook Aggregate x4 for 2 Maps and 2 Aggregates
40%	Predicted Map rather than Sequence Map, Missed a Select
35%	Predicted Map rather than Select, Extra Selector, Extra Map, Missed Sequence Map, Missed Selector Width

Table 5.1: Descriptions of mistakes made by the meta-model on a sample of outputs

Chapter 6

Compressed Models

So far we have only been considering the parameters of Transformer programs that have been output by the Tracr compiler directly, the next step in our progression towards a less synthetic transformer is what we're calling a 'compressed transformer program'. As established in chapter 3.2.1 a transformer program consists of a number of multi-head attention and MLP blocks, each of which reads from and writes to a residual stream. As proposed by Lindner et al. [2023] a compressed transformer takes the MHA and MLP blocks compiled by Tracr and places a decompression layer before each block and a compression layer after each block. The effect of this is a compressed residual stream that contains only the information needed for the transformer program in the most compact representation possible. Lindner et al. [2023] also propose that the simplest possible compression and decompression layer suffice, namely a single matrix W_{emb} to perform the decompression, and its transpose $W_{emb}^{\top}$ to perform the compression. These compression and decompression blocks must be trained such that they retain only the important information that is required for the transformer program to function. Lindner et al. [2023] suggests that this can be achieved through distillation from a standard Tracr transformer program to a compressed transformer program. So we compile a RASP program into a Tracr transformer and call this transformer the teacher model. Then add a compression and decompression matrix before an after each MHA/MLP block to form the student model. The two models can then be called on the same input and a loss function (Equation 6.1) that measures the distance between the uncompressed outputs can be used to minimize the error between the computation performed by the teacher model and the student model. We freeze all the parameters of the student model except the matrix W_{emb}. Once the model has converged we are left with 2 models with the same parameters except for the students extra compression matrix W_{emb}. Whereby applying the compression and decompression has very little impact on the computation of the model except for reducing the size of the residual stream.

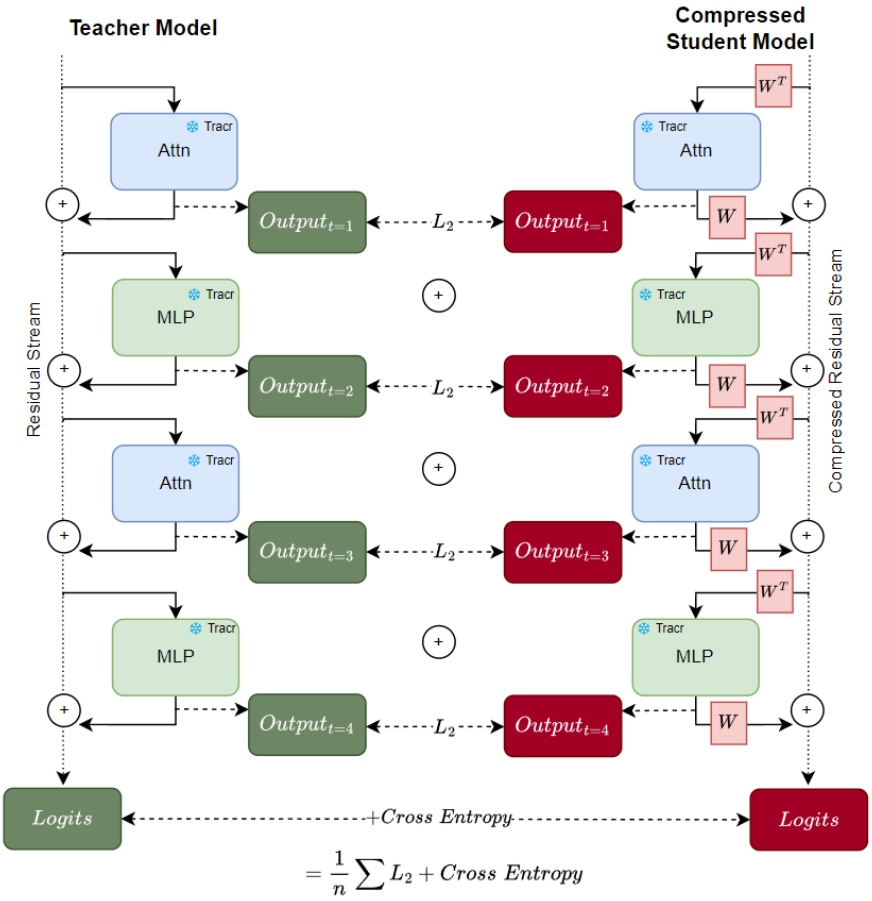


Figure 6.1: Tracr proposed compression loss function that consists of layerwise L2 output losses plus a cross entropy loss term between the output logits

Unfortunately, the code that Lindner et al. [2023] used to distill the compressed transformer programs is closed source so we had to implement our own solution from scratch. Since Tracr already compiles transformer programs into Jax, we decided to proceed with implementing our distillation in Jax as well. This came with it's own challenges as Jax has only gained popularity in recent years and there are currently no examples of how to perform distillation within Jax.

Initially, we used the loss function provided by Lindner et al. [2023] which stores the intermediate outputs from every block of the teacher and student after calling both on the same sample, then measures the distance between these outputs. Since the blocks always operate within the uncompressed residual space the outputs from the blocks in the standard and compressed model will always have the same shape. Accordingly, we can minimize this distance and minimize the difference in the computation performed by a compressed model compared to the original standard model.

$$\begin{aligned} \mathscr{L}(W,x) &= \mathscr{L}_{out}(w,x) + \mathscr{L}_{layer}(w,x) \\ \mathscr{L}_{out} &= loss(f(x), \hat{f}_w(x)) \\ \mathscr{L}_{layer} &= \sum_{layer\ i} (h_i(x) - \hat{h}_{w,i}(x))^2 \end{aligned} \tag{6.1}$$

We distilled a number of transformer programs from randomly generated programs using a compression factor of 2, meaning that the compressed residual stream was half that of the original residual stream. Initially, we disabled the L_{out} term, as we though that mimicking the computation of the teacher alone should suffice since the output decoder is shared between the teacher and student. However, we found that the accuracy of these models was very low, producing the same outputs as the teacher model just 70% of the time on average, with a very high variance frequently producing distilled models with 0% accuracy. Since we'll be needing to create a model zoo of these distilled models, reliability is imperative. Accordingly we tried a couple of different loss functions, namely L1 and Cross entropy loss (Equation 6.2). However we found that neither had a significant impact on performance or reliability (Table 6.1).

$$\begin{aligned} \text{L1 loss} \qquad & \mathscr{L}_{layer} = \sum_{layeri} (h_i(x) - \hat{h}_{w,i}(x)) \\ \text{Cross Entropy loss} \qquad & \mathscr{L}_{layer} = \sum_{i} h_i(x) \cdot \log \hat{h}_{w,i}(x) \end{aligned} \tag{6.2}$$

	L2	L1	Cross Entropy
Acc	**70.5**(45.2)	67.8(43.4)	71.7(43.0)
Loss	0.00065(0.0011)	0.0034(0.0044)	−14.6(28.5)

Table 6.1: Accuracy and Loss Measures for distilled compressed models with various loss functions to minimize the error between layer outputs

To help debug this issue we fixed the program to be the 'sort unique' example provided by Tracr as this is the example that Lindner et al. [2023] used to test the effectiveness of model compression. We then set the compression level to 1 so that no compression was taking place and initialized W_{emb} to be the identity matrix + noise. This presents the easiest possible problem to learn as the optimization problem is being initialized very close to the optimal solution, maximizing the probability that the nearest local optimum is the global optima, a.k.a. the identity matrix. During training, the compressed student

model converged towards the solution but not fully. As the noise was not being fully removed. Figure 6.2 shows how W_{emb} is still noisy after training. We also visualize $W_{emb}^{\top}W_{emb}$ as kindly advised by David Lindner, which shows us the overall result of applying decompression followed by compression on the residual stream. Similarly in Figure 6.2, $W_{emb}^{\top}W_{emb}$ is also noisy.

We believe that the reason the model is not converging is due to the sparse nature of the inputs. This makes model updates very inefficient as only a small percentage of the model gets activated at each timestep resulting in the majority of gradients being zero. To activate more of the network in each training step we decided to remove the token embedding layer that maps from a string of text to one hot encoding, and replace it with randomly generated embeddings. In doing so we actually adversely affected the accuracy and loss (Table 6.2) however the compression matrix W_{emb} converges to the correct global optima as show in in Figure 6.2.

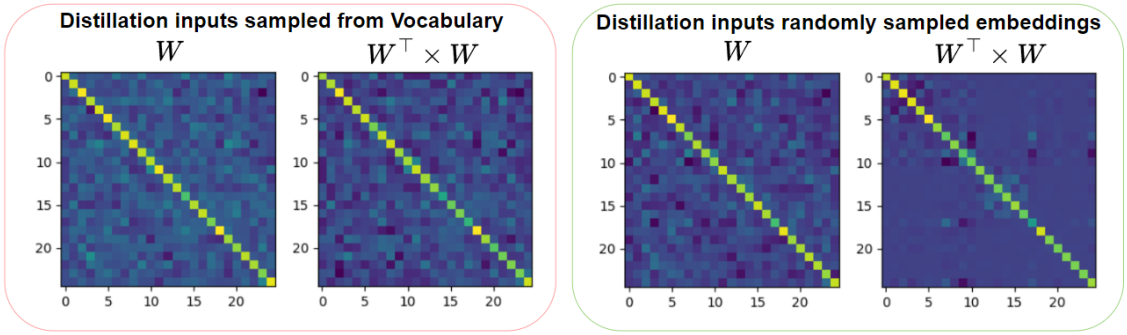


Figure 6.2: Affect of sampling from vocabulary for distillation vs removing embedding layer and generating random embeddings

	Vocabulary Samples	Random Samples
Acc	**70.5**(45.2)	61.1(43.1)
Loss	0.00065(0.0011)	0.032(0.094)

Table 6.2: Accuracy and Loss Measures for distilled compressed models with various loss functions
Values formatted by "mean (standard deviation)"

In order to increase the accuracy of the program we can add the $\mathscr{L}_{out}$ term to the loss function that measures the distance between the logits at the output of the model, as suggested by Tracr's loss function. The intuition being that currently the loss function only compares the outputs of the model during computation, so retains the parts of the residual stream that are most relevant to this computation. However the loss function

does not currently have any regard for the decoded outputs of the model. Adding a cross entropy loss term between the logits of the teacher and student model proved to remedy this issue. When sampling from the vocabulary, the accuracy increased to nearly 100% and was far more reliable, having a variance of 4.4% rather than 45.2%. We can also verify our prediction that the compression matrix W_{emb} is not retaining output indices by visualizing the decompression+compression operation $W_{emb}^{\top} W_{emb}$ as we did previously. As predicted Figure 6.3 clearly shows a set of indices from 6 to 10 that were previously discarded by the decompression and compression now being retained. However when sampling randomly from the embeddings the boost in accuracy at the outputs cannot be seen, possibly due to the unnatural input when a random embedding is provided as opposed to a one-hot encoded input.

	Vocab Sampling		Random Sampling	
	L2 + C.E.	L2 × C.E.	L2 + C.E.	L2 × C.E.
Acc	97.8(4.4)	93.5(14.5)	63.9(38.1)	56.3(48.3)
Loss	0.00178(0.0026)	0.000172(0.00051)	0.057(0.13)	0.0081(0.015)

Table 6.3: Effect of combining a loss term to minimize the output logit error when distilling compressed models
Values formatted by "mean (standard deviation)", C.E. - Cross entropy

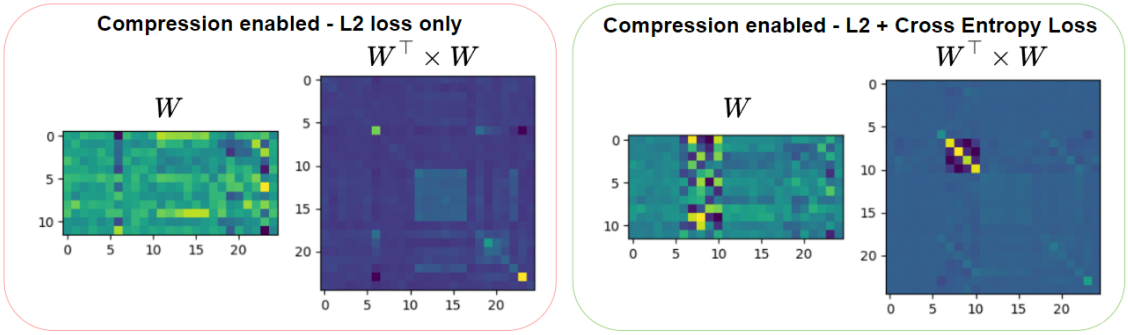


Figure 6.3: Interesting effect of adding the Cross entropy loss between the output logits on the information retained when compressing the residual stream - on the left, very little of the residual stream is maintained when using purely a layer-wise loss. When the output loss term is added the compression matrix retains indices 5 to 10, which happen to be those indices that correspond to the outputs of the program. Demonstrating how the output loss term affects the information retained in the residual stream.

Ideally we would like to balance the best of both worlds, and converge to an optimal compression matrix by generating samples from randomly from the embedding space. And have high accuracy with respect to the teacher by generating samples from the vocabulary

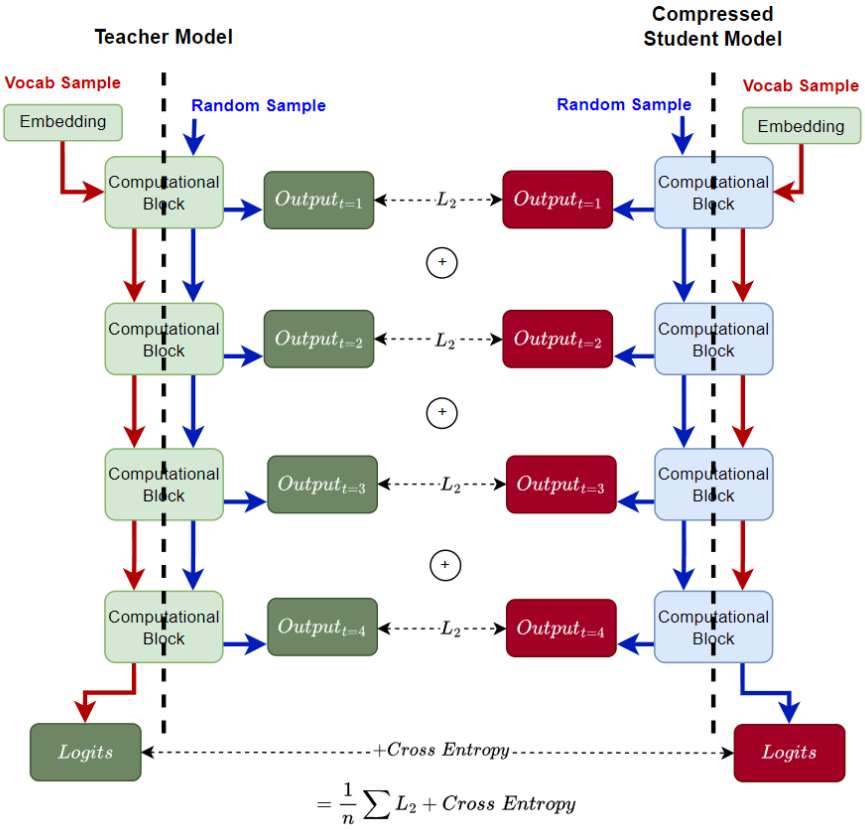


Figure 6.4: Combined Random + Vocab Loss function to train an optimal compression matrix that also maximizes accuracy

and using a cross entropy loss term between the logits. Thankfully, this is possible by simply combining both scenarios into a single update step (Figure 6.4). At every training timestep we generate 2 samples; one is randomly sampled from the embedding space and another is randomly sampled from the vocabulary. The teacher and the student are called on both inputs, then the Tracr loss function (Equation 6.1) is computed between the outputs of the teacher and student for the random sample. While the logit loss function is computed between the logits of the teacher and student for the vocabulary sample. These two loss terms are then added together and then passed to the optimizer to backpropagate the error. In our testing this has shown the best performance overall with $87\% \pm 20$ accuracy and a loss of 0.31 ± 0.59, these scores might not look amazing initially, but over 60% of our samples had 100% accuracy, and those that did not, maintained a high loss of over 0.2 throughout training. Since our main goal is to generate a model zoo quickly, we can actually discard these 'high loss' samples very early on in training, greatly improving computational efficiency.

6.1 Automated Training

Now that we have outlined the best way to train a compressed transformer program we need to create a model zoo that is suitable for training our meta-model. In order for the model to learn the actual relation between the parameters and RASP program, we need to maximize the quality of the dataset. This means we need to be able to guarantee that every distilled model matches the outputs of its teacher, we arbitrarily decided that anything over 90% accurate would suffice. Of equal importance we also need to guarantee that the computation performed by the student model is the same as the teacher. A large problem with the nature of the programs we generate is that they frequently output the same value for the entire input space, making the mapping function very easy to learn if we only considered the accuracy of the model. Accordingly we also reject any models that have a loss above 0.05 as they do not mimic the computation of the teacher closely enough. It's also essential to maximize the number of distilled models we can train within a given time frame and computational constraints. Consequently, significant effort was invested in optimizing the training process to minimize training duration and reduce computational inefficiencies. Through the process of optimization we managed to decrease the training time from 20 minutes to just 4 minutes on average, the key mechanisms through which we achieved this are as follows (in order of importance):

1. **cosine annealing** - drastically reduces training time, due to the low number of parameters and simple loss landscape the main limiting factor is the schedulers ability to reduce the learning rate proportionally to the error. We also tried a more aggressive scheduler that exponentially reduced learning rate, but this was more prone to sub-optimal solutions.

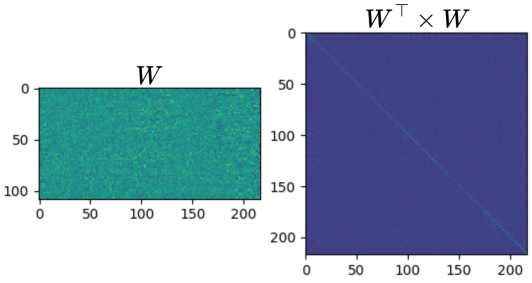
2. **scheduler warm up period** - initially the loss can be very high in the order of 10^4, by using a warm up scheduler we greatly reduced the explosion of gradients during this period.

3. **Norm Gradient Clipping** - in rare cases gradients explode, using norm gradient clipping greatly reduces the likelihood of this. Conversely using value clipping was found to increase the likelihood of exploding gradients, perhaps because negative features are allowed to persist for longer.

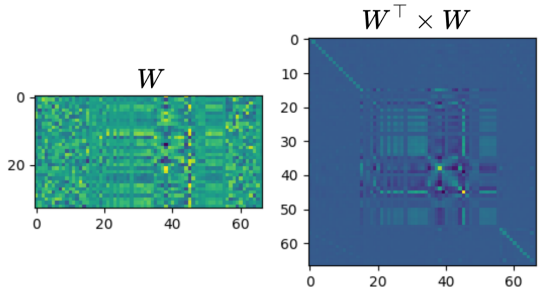
4. **CRAFT timeouts** - as mentioned in section 4 CRAFT often takes a long time to compile, we used the same strategy as before to limit compilation time to under 0.2 seconds.

5. **CRAFT to JAX bug** - due to the unconventional nature of CRAFT transformer programs the shape of the value matrix can sometimes be greater than that of the key/query matrix. Currently, this event crashes compilation, but we took the time to fix the bug in JAX's source code making compilation more reliable. We're also collaborating with Lindner et al. [2023] to resolve the bug in their repository.

We also experimented with using GPU acceleration but due to the small number of parameters in a Tracr transformer program this proved to be inefficient even when using the maxim possible batch size. And would increase the monetary cost of generating the dataset due to the greater cost of GPU acceleration.



(a) Typical Compressed Transformer Program Compression matrix approximately performing the identity mapping between decompression and compression



(b) Interesting pattern formed by decompression and compression operation

Figure 6.5: Exemplar compression matrix and decompression+compression operator $W^\top W$

6.2 Weight Compression

In this section we demonstrate how to convert from a standard compressed transformer (Algorithm 1) that explicitly decompresses the residual stream at the beginning of each block, into a compressed transformer with compressed parameters (Algorithm 2) that integrates the decompression and compression into the parameters of MHA and MLP layers.

Given a trained compressed transformer program it would be ideal if we could represent the parameters in the exact same format as the uncompressed Tracr model. Thankfully the Multi-Head attention blocks apply the key, query and value parameters directly to the input (Alg 1, ln 8), since the decompression matrix is also a linear operation the two matrices can be multiplied without changing the functionality of the block (Alg 2, ln 7). It does this while still allowing us to remove the decompression step before we call the block on the residual stream (Alg 1, ln 7). Additionally, the linear projection layer, present

at the end of each attention block can be used to integrate the compression matrix into the attention block (Alg 2, ln 11), allowing us to remove the compression step after the attention block (Alg 1, ln 14). Furthermore, the same logic can be applied to the MLP layer, by multiplying the first layer weights with the decompression matrix and the second layer weights and bias by the compression matrix (Alg 2, ln 13 and 14 respectively). The following algorithms prove this mathematically by considering the two representations, one as before with decompression before and compression after a block, and another where the compression has been moved inside the block and combined with the parameters.

Algorithm 1 Standard Compressed Transformer

1: $Attention(Q,K,V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$
2: $FFN(x,W_1,b_1,W_2,b_2) = \max(0, xW_1+b_1)W_2+b_2$
3: $SplitHeads(x) = ...$ ▷ Reshape operation
4: $MergeHeads(x) = ...$ ▷ Reshape operation
5: $r = xW_{emb}^\top$ ▷ Compress the input into the residual stream
6: **for** $n = 1 \rightarrow N_{Layers}$ **do**
7: $x = rW_{emb}$ ▷ Read from residual stream
8: $Q = xW_q,\ K = xW_k,\ V = xW_v$
9: $Q = SplitHeads(Q), K = SplitHeads(K), V = SplitHeads(V)$
10: $a = Attention(Q,K,V)$
11: $a = MergeHeads(a)$
12: $y = aW_p$ ▷ Project outputs residual stream indices
13: $r_d = yW_{emb}^\top$ ▷ Compress the outputs
14: $r = r + r_d$ ▷ Write to residual stream

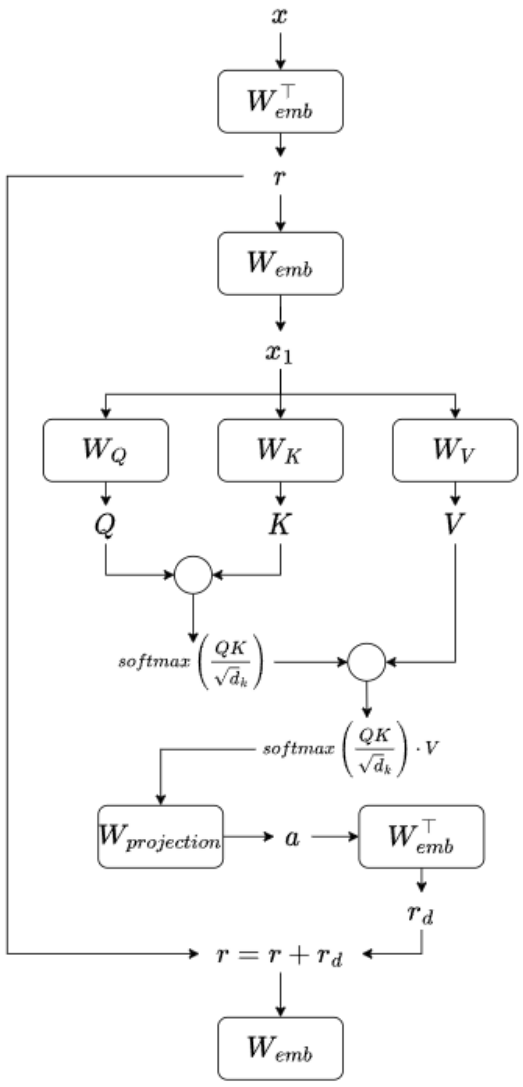
15: $x = rW_{emb}$ ▷ Read from residual stream
16: $f = xW_f + B_f$ ▷ MLP first layer
17: $f = max(f,0)$ ▷ ReLU Activation
18: $s = fW_s + B_s$ ▷ MLP Second layer
19: $r_d = sW_{emb}^\top$ ▷ Compress the outputs
20: $r = r + r_d$ ▷ Write to residual stream
21: **end for**

Algorithm 2 Compressed Transformer, with compression inside parameters

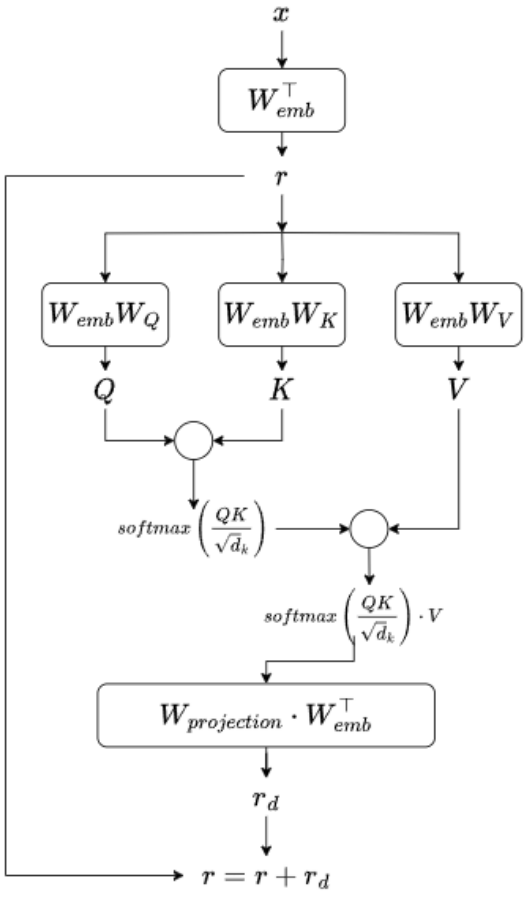
$Attention(Q,K,V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$
$FFN(x,W_1,b_1,W_2,b_2) = \max(0,xW_1+b_1)W_2+b_2$
$SplitHeads(x) = ...$
$MergeHeads(x) = ...$
$r = xW_{emb}^\top$ ▷ Compress the input into the residual stream
for $n = 1 \rightarrow N_{Layers}$ **do**
    $Q = x(W_{emb}W_q), K = x(W_{emb}W_k), V = x(W_{emb}W_v)$ ▷ W_{emb} combined into params
    $Q = SplitHeads(Q), K = SplitHeads(K), V = SplitHeads(V)$
    $a = Attention(Q,K,V)$
    $a = MergeHeads(a)$
    $y = aW_pW_{emb}^\top$ ▷ compression combined into params
    $r = r+y$ ▷ Write to residual stream

    $f = x(W_{emb}W_f)+(W_{emb}B_f)$ ▷ decompression combined into params
    $f = max(f,0)$ ▷ ReLU Activation
    $s = fW_sW_{emb}^\top + B_sW_{emb}^\top$ ▷ compression combined into params
    $r = r+s$ ▷ Write to residual stream
end for

We have even experimentally verified this result and implemented the Jax compressed transformer architecture in numpy[Harris et al., 2020] and then compressed the parameters and compared the outputs, verifying that both representations are indeed equivalent. Annoyingly the initial compression matrix applied to the inputs cannot be combined into any blocks as the inputs need to be compressed into the residual stream, however this could easy be considered a simple question of input data representation, rather than a component of the functionality of the compressed model.



(a) The computation performed by an attention head with compression outside the parameters, being applied before and after each block such that the residual stream is compressed



(b) The computation performed by an attention head with compression integrated into the attention head parameters such that the block implicitly reads and writes directly to the compressed space of the residual stream

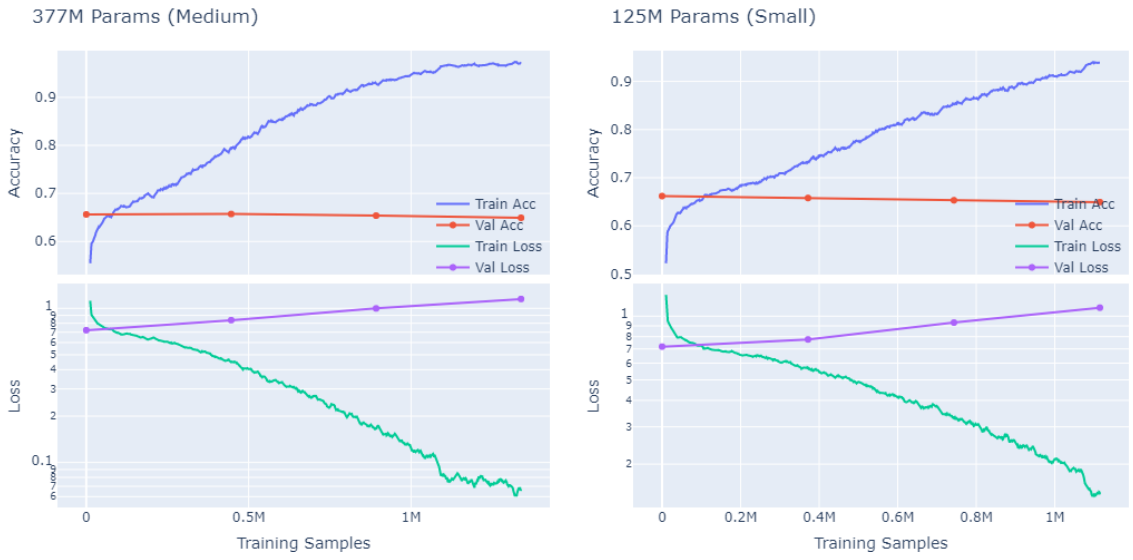
Figure 6.6: Uncompressed vs Compressed parameters of an attention head

By encoding compressed transformer programs in this manner we can train the meta-model with the exact same data structures as we used to train the meta-model to decode standard Tracr Transformer programs. This greatly simplifies the training workflow as we can reuse much of the same code to train a meta-model to decode compressed and standard transformer programs, where the only change required is the data loader to switch to the relevant dataset.

Overall we generated a dataset of 150,000 of these Compressed Transformer programs using around 20,000 CPU hours to do so (1 CPU core for one hour). Accordingly we averaged a train time of just 8 minutes per sample, a vast improvement over the 2 hours per sample that Lindner et al. [2023] achieved in their brief dive into compressed transformer programs. In order to add some noise to the samples without changing the functionality of the program we trained each model for an extra 3 epochs after converging, giving us an extra 3 models for every single distillation totaling 600,000 samples overall. However, in retrospect, these extra samples did not provide enough variation to be useful when training.

6.3 Compressed Model Results

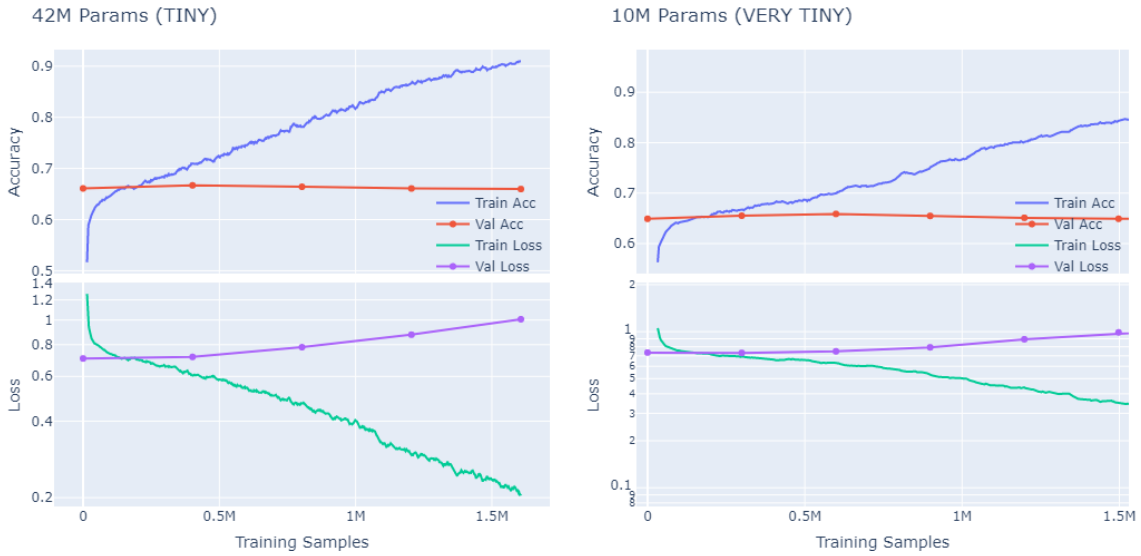
When training the compressed meta-model to decompile compressed transformer programs into RASP code, we found that overfitting was a much greater problem than it was for our standard model in Section 5.5. Due to the reduced size of the dataset we chose a smaller architecture than we used for our standard meta-model, GPT2 Medium, with 377M parameters. However, this overfitted immediately (Figure 6.7a) with the validation loss increasing since initialization. To attempt to reduce the amount the model overfit on the data we decreased the architecture size by a factor of 3 to GPT2 small with 125M parameters. However, this overfit just as much with the validation loss diverging from the training loss similarly to before (Figure 6.7b).



(a) GPT2 Medium (b) GPT2 Small

Figure 6.7: GPT2 Medium and Small both overfitting on the compressed meta-model dataset. GPT2 Medium has 24 layers each with 16 attention heads operating on a 1024 element hidden vector, giving 377M parameters. GPT2 Small has 12 layers with 6 attention heads and 768 element hidden vector, giving 125M parameters. Despite 3 times fewer parameters, GPT2 small overfits

In order to test how fewer parameters would be required to prevent the model from overfitting we continued to decrease the size of the model. First, to a custom architecture we're calling GPT2 Tiny with 3 times fewer parameters than GPT2 small, just 42 million. Despite the very small model, it showed the exact same trend of overfitting as GPT2 medium and small. Accordingly, we decreased the architecture once more to have just 10 million parameters, which did finally reduce the degree to which the model overfit the data, while also keeping a similar validation accuracy as before.

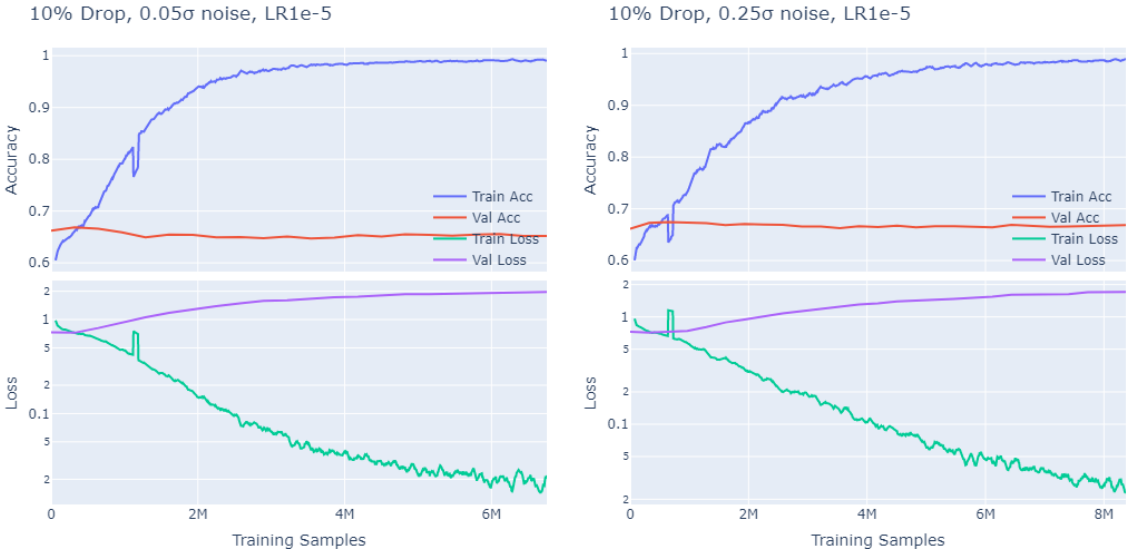


(a) GPT2 Tiny (b) GPT2 Very Tiny

Figure 6.8: GPT2 Tiny and Very Tiny overfitting on the compressed model dataset. Tiny has 8 layers each with 4 attention heads and a 512-element hidden vector. Very Tiny has a minuscule 4 layers each with 2 attention heads but keeps a 512-element hidden vector

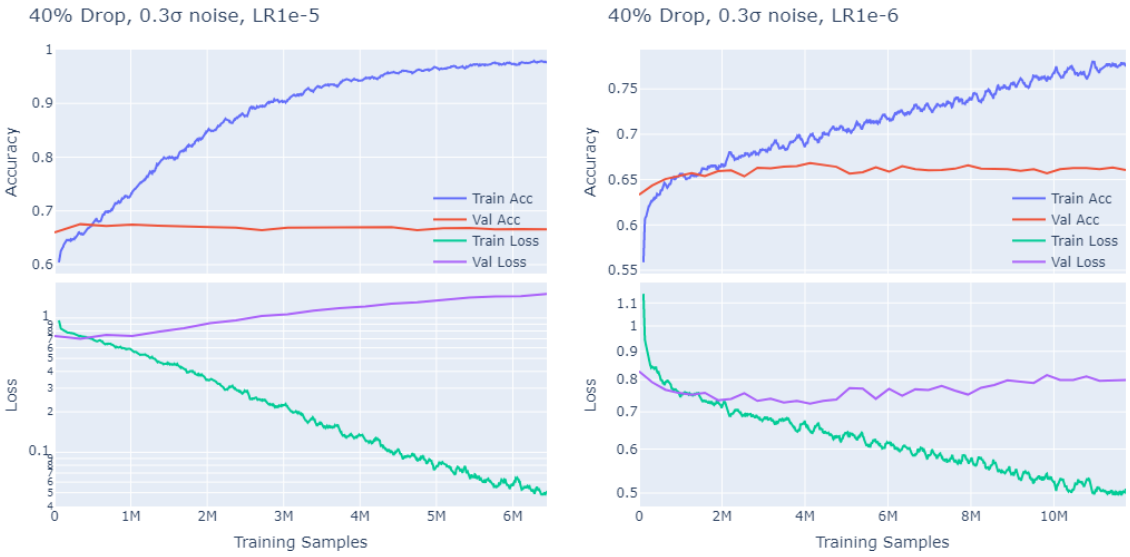
We believe that the reason the compressed meta-model is overfitting much more than the standard model is due to the added noise in the parameters of compressed transformer models, due to the compression matrix being trained non-deterministically. This noise makes every single sample in the dataset unique, whereas before many features were shared between samples. We believe that these unique features provide a mechanism to identify individual training samples and memorize their class rather than learning the inverse mapping from compressed transformer models to RASP code. In order to saturate the models ability to memorize samples ideally we would be able to generate more samples, increasing the dataset from 150,000 to around 1 million samples. However, the current dataset was already very computationally expensive, requiring 25,000 CPU hours taking over a week to execute. Due to time as well as financial constraints increasing the size of the dataset further was not practical.

The alternative available to us, was to add noise to the dataset increasing it's effective size, as well as using dropout to randomly mask input features potentially masking out those features that the model is overfitting on. We added this noise to GPT2 Medium as we know from our standard meta-model that this model has sufficient complexity to fit the standard dataset at the very least.



(a) 10% Dropout 0.05σ noise (b) 10% Dropout 0.25σ noise

Figure 6.9: Despite adding noise and increasing dropout the model still overfits



(a) 30% Dropout 0.4σ noise, LR1e-5 (b) 30% Dropout 0.4σ noise, LR1-6

Figure 6.10: Increasing the noise and dropout to extremely large values still does not prevent overfitting, however combining this with a decrease in the learning rate (on the right) does allow the model to converge without overfitting

We also assessed the program level accuracy similar to how we did in section 5.5, we have expressed the accuracy metrics for the best model we trained from Figure 6.10b.

Token Level accuracy	Program Level Accuracy					
	100%	90%	80%	70%	60%	50%
66.38	0.3	5.5	16	38	63	87

Table 6.4: Summary of the Token Level and Program Level Accuracy of the Compressed meta-model trained using dropout and noise with a low learning rate. Token level accuracy is the average percentage of tokens correctly predicted across the validation set. Program level accuracy at level X% is the percentage of predictions on the validation set that were predicted with above X% accuracy.

To conclude we are not sure whether our lower accuracy is purely due to compression adding noise to samples and making it easier to overfit the data. Or if there is some artifact in our dataset which we have not considered due to the complexity of our data generation pipeline.

Chapter 7

Natural Models

What would happen if we took the compressed models described in the previous chapter and randomly initialized the Tracr compiled parameters and trained them too? You would learn what we're calling a *Natural* transformer program, that is a transformer program for which all the parameters have been learned rather than compiled. They are as close to a conventional transformer, such as a GPT-style language model, as we can possibly get. The brilliant thing about maintaining the same architecture as the compressed transformer model is that it allows us to train them using the exact same method. So we can distill the behavior of a Tracr transformer program to a natural transformer program. This allows us to attempt the inverse problem of mapping from a natural transformer program back to the original RASP program as we have done for all the models so far.

Better still, we found that the exact same training configuration that we used to train compressed models worked very well when training all the parameters of a natural model. The only difference being that we use the Xavier initialization[Kumar, 2017] to initialize all of the parameters of the model and did not freeze any of the parameters as we did before. Otherwise, the loss function and all hyper-parameters remained the same.

Accordingly, we generated another dataset of 71,000 samples using 5000 CPU hours, achieving 10 minutes per sample. We did not allocate as many resources to training natural models as we did for training compressed models since we were not satisfied with the performance of our meta-model on compressed models (6.3), and natural models are theoretically much harder to learn.

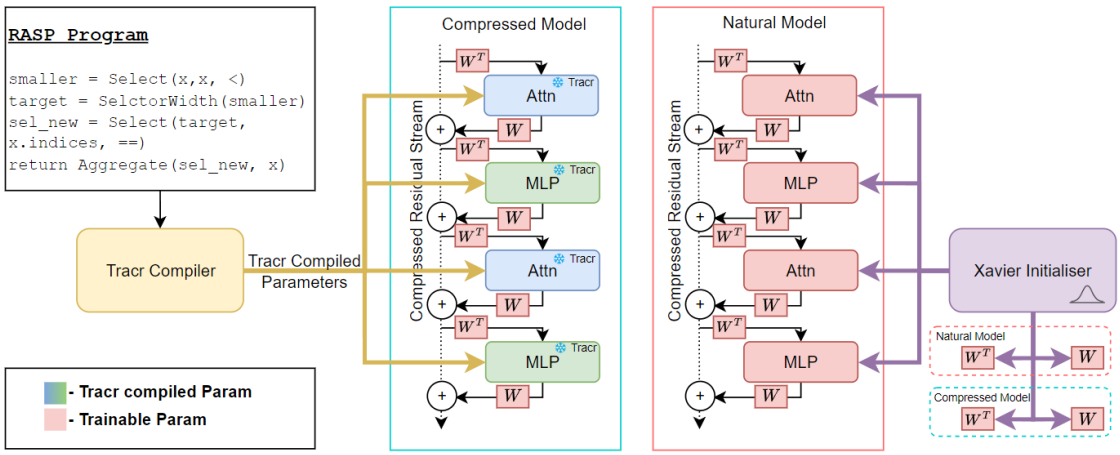


Figure 7.1: Compressed model initialization from Tracr parameters vs Random Xavier initialization of Natural model

7.1 Natural Model Results

We kept the same GPT2 Medium architecture combined with 0.4σ input noise, 30% input dropout, and low learning rate as we used for our *compressed meta-model*. The training curves for this meta-model have been plotted in Figure 7.2. As we have done previously, we also recorded the whole program accuracy and have presented the results in table 7.1. As expected the Natural meta-model performs worse than both the standard and compressed natural model, with just 60.18% token accuracy. Although considering that these models are trained entirely from scratch with very few constraints we thought this was an impressive result. It shows that the computational model used by transformers has basic components that emerge repeatedly from random initializations on related but different tasks.

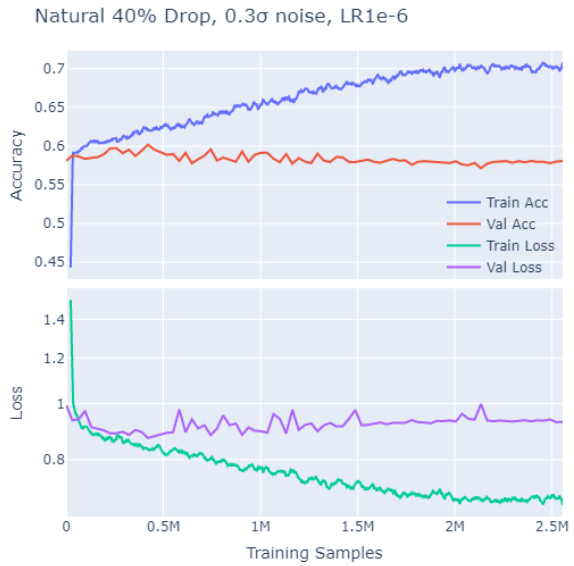


Figure 7.2: Natural meta-model results - GPT2 Medium architecture, 0.4σ input noise, 30% input dropout, LR 1e-6. Showing similar overfitting as the compressed meta-model but this is expected due to the small dataset size

Token Level accuracy	Program Level Accuracy					
	100%	90%	80%	70%	60%	50%
60.18	0.0	0.2	1.0	8.4	43	88.9

Table 7.1: Summary of the Token Level and Program Level Accuracy of the Natural Meta-model. Token level accuracy is the average percentage of tokens correctly predicted across the validation set. Program level accuracy at level X% is the percentage of predictions on the validation set that were predicted with above X% accuracy.

Chapter 8

Future Work & Discussion

8.1 Parameter Permutation

There are inherent invariances in nearly all machine learning models, whereby many distinct models exist that perform the exact same functional computation. Transformer models are no exception, ideally, our meta-model would be agnostic to these parameter invariances, a popular method would be to permute the parameters of our samples to create new samples. We could consider the research by Navon et al. [2023], Schürholt et al. [2021], and Roeder et al. [2021] to provide us with one such method of permuting parameters. They study MLP models and permute neurons between layers such that a neuron in layer n writes to a different index in the hidden state, but then in layer $n+1$ the parameters are adjusted to compensate for this alteration. By applying such a transformation the overall function that the model implements is unaffected; but the parameters have been rearranged, enabling us to greatly increase the size of the model zoo. Unfortunately, no such method of permuting the parameters of a Transformer model has been developed. We did spend some time on the subject and believe that a similar strategy can be employed in a transformer, however, the structure of an attention head adds great complexity to the problem and we did not have the time to implement this feature. If our work was to be continued we believe that parameter permutations such as this would greatly improve the performance of our meta-model.

8.2 Autoregressive Modeling

Another mechanism to improve performance is auto-regressive Modeling, where a model call is made for every timestep of the output. So for a program of length L the model would be called L times, each time it generates a prediction for the next timestep. Auto-regressive modeling is a very popular technique in modern LLM's as it allows the outputs of the model to be input back into the model for future predictions, typically increasing the coherence of the output prediction. However, it does come at a very high computational

cost, being L times more expensive per model call. Although we believe that given the restricted size of the dataset in the compressed and natural model case, the additional computational expense will be worth-wile, especially given that it would likely increase the validity of programs. Unfortunately, due to complications in training the compressed and natural models we did not have enough time to test auto-regressive modeling.

8.3 Program Generation

As mentioned in Section 4, our program generation algorithm is rather simple and frequently outputs programs that have the same output for the entire domain of inputs. In future we would like to improve this algorithm to minimize the entropy of the outputs, one way of doing so would be by tracking the domain of every variable during construction (similar to how Tracr does so during compilation) and to use this information to sample lambdas in such a way that minimizes the entropy of their outputs.

Additionally, our current program generation strategy creates an arbitrary computational tree that frequently includes spurious nodes that are independent of the output. These superfluous nodes are pruned during the construction process. As a result, programs that are generated are not guaranteed to be of the desired length and we had to rejection sample in order to guarantee our length criteria were met. In future work, we believe it would be worthwhile to constrain program generation such that it is guaranteed to output a program of a certain length.

8.4 Output Sampling

We also realized that the search space could be reduced for the meta-model by only sampling valid tokens from the output logits. Typically the outputs of the meta-model produce a valid program, however, in some cases, variable types are invalid for a given operation. Since we can check the validity of programs without supervision we could analyze outputs for errors, and whenever one is found iterate through the next most likely token prediction according to the output logits until the error has been removed. This would take some time to code, but would offer a very computationally cheap boost to performance.

8.5 Fine Tuning from Pre-Trained Standard Meta-model

A key issue with our findings is the cost of generating compressed and natural transformer programs in order to train our meta-model. Since the task of interpreting a standard vs a compressed or natural meta-model is very similar; it stands to reason that we could use the higher performance standard meta-model parameters to initialize compressed or natural models. Doing so would decrease the amount of data required to train the meta-model as much of the knowledge of decompilation would already be instilled within the model.

Chapter 9

Conclusion

In this paper, we have explored the RASP programming language and how it can be used to generate transformer programs. Building on this, we've constructed a model zoo consisting of these transformer programs by developing an algorithm to reliably generate programs in RASP. Next, we used this model zoo to train a meta-model to be able to decompile a transformer program into RASP code with good success, predicting 73.04% of tokens correctly. While achieving a perfect 100% reconstruction accuracy remained a challenge, we made significant strides.

We then shifted our focus to generating a model repository that aligns more with traditional transformers. This involved compressing our transformer programs, tackling many challenges related to the large amounts of computation required to do so. As anticipated, decompiling these compressed transformer programs proved tougher, resulting in us achieving 66.38% accuracy, highlighting the complexity of decompilation.

Lastly, we tackled the task of training transformer programs from scratch, while still maintaining the same computation as the source RASP program. Even with the intricate nature of these models, we managed to achieve a respectable accuracy rate of 60.18% at inverting the natural transformer programs.

In conclusion, our research leaves us optimistic about the potential for our findings to contribute to the development of improved techniques for decompiling transformer models into code.

Task	Token Level accuracy	Program Level Accuracy					
		100%	90%	80%	70%	60%	50%
Standard	73.04	0.4	13.27	32.61	58.41	81.1	95.4
Compressed	66.38	0.3	5.5	16	38	63	87
Natural	60.18	0.0	0.2	1.0	8.4	43	88.9

Table 9.1: Summary of the Token Level and Program Level Accuracy of each of the Meta models we have trained. Token level accuracy is the average percentage of tokens correctly predicted across the validation set. Program level accuracy at level X% is the percentage of predictions on the validation set that were predicted with above X% accuracy.

Bibliography

S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, Mar. 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

N. Elhage, N. Nanda, C. Olsson, T. Henighan, N. Joseph, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, N. DasSarma, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

D. Friedman, A. Wettig, and D. Chen. Learning transformer programs, 2023. URL https://arxiv.org/abs/2306.01128.

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https:

//doi.org/10.1016/0893-6080(89)90020-8. URL https://www.sciencedirect.com/science/article/pii/0893608089900208.

S. K. Kumar. On weight initialization in deep neural networks. *CoRR*, abs/1704.08863, 2017. URL http://arxiv.org/abs/1704.08863.

D. Lindner, J. Kramár, S. Farquhar, M. Rahtz, T. McGrath, and V. Mikulik. Tracr: Compiled transformers as a laboratory for interpretability, 2023. URL https://arxiv.org/abs/2301.05062.

S. M. Lundberg and S. Lee. A unified approach to interpreting model predictions. *CoRR*, abs/1705.07874, 2017. URL http://arxiv.org/abs/1705.07874.

N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. Progress measures for grokking via mechanistic interpretability, 2023.

A. Navon, A. Shamsian, I. Achituve, E. Fetaya, G. Chechik, and H. Maron. Equivariant architectures for learning in deep weight spaces, 2023.

W. Peebles, I. Radosavovic, T. Brooks, A. A. Efros, and J. Malik. Learning to learn with generative models of neural network checkpoints, 2022.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016. URL http://arxiv.org/abs/1602.04938.

G. Roeder, L. Metz, and D. Kingma. On linear identifiability of learned representations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9030–9039. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/roeder21a.html.

K. Schürholt, D. Kostadinov, and D. Borth. Self-supervised representation learning on neural network weights for model characteristic prediction. *CoRR*, abs/2110.15288, 2021. URL https://arxiv.org/abs/2110.15288.

R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *CoRR*, abs/1610.02391, 2016. URL http://arxiv.org/abs/1610.02391.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

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

G. Weiss, Y. Goldberg, and E. Yahav. Thinking like transformers. *CoRR*, abs/2106.06981, 2021. URL https://arxiv.org/abs/2106.06981.

C. Yun, S. Bhojanapalli, A. S. Rawat, S. J. Reddi, and S. Kumar. Are transformers universal approximators of sequence-to-sequence functions? *CoRR*, abs/1912.10077, 2019. URL http://arxiv.org/abs/1912.10077.