A Method details

A.1 Categorical attention

As described in Section 3.2, we implement categorical attention by associating each attention head with a boolean predicate matrix, $\mathbf{W}_{\text{predicate}} \in \{0,1\}^{k \times k}$, where k is the variable cardinality, with the constraint that each row $\mathbf{W}_{\text{predicate},i}$ sums to one. The self-attention pattern is then defined by a score matrix $\mathbf{S} \in \{0,1\}^{N \times N}$, with $\mathbf{S} = \mathbf{x}\mathbf{W}_Q \mathbf{W}_{\text{predicate}} (\mathbf{x}\mathbf{W}_K)^\top$. To ensure that each query token attends to a single key token, we use hard attention, defining the attention pattern at position i as $\mathbf{A}_i = \text{One-hot}\left(\arg\max_j \mathbf{S}_{i,j}\right)$.

Defaulting to the beginning of sequence. We implement hard attention so that, in the event that there is no matching key for a query, the model defaults to attend to the first token in the sequence. Let $\mathbf{S} \in \{0,1\}^{N \times N}$ denote the score matrix for a sequence with length N. We define a modified score matrix $\bar{\mathbf{S}}$ such that, for each row i,

$$\bar{\mathbf{S}}_{i,j} = \begin{cases} \mathbf{S}_{i,j} + (1 - \max_j \mathbf{S}_{i,j}) & \text{if } j = 1 \\ \mathbf{S}_{i,j} & \text{otherwise.} \end{cases}$$

Breaking ties. Furthermore, we implement the attention mechanism so that, in the event that there is more than one matching key, the model attends to the closest match. Given the score matrix $\mathbf{S} \in \{0,1\}^{N \times N}$, we define the modified score matrix $\bar{\mathbf{S}}$ such that, for each row i, $\bar{\mathbf{S}}_{i,j} = \mathbf{S}_{i,j} \times b_{i-j}$, where $b_{i-j} \in [0,1]$ is a bias associated with the offset between position i and j. For most experiments, we fix the bias to decrease from 1, when $|i - j| = 1$, to $1/N$, when $|i - j| = N$, with $b_0 = 1/N$ to bias the query at position i against attending to itself. This is similar to types of relative positional bias that have been proposed for regular Transformer-based language models [Press et al., 2022].

A.2 Additional modules

Numerical attention. We implement limited support for numerical variables, designed to ensure that all variables within the program are integers within a bounded range, which allows us to discretize the program by enumerating all possible inputs. First, we include numerical attention heads, which read categorical variables as key and query, and numerical variables as value. The numerical attention head computes attention scores $\mathbf{S} \in \{0,1\}^{M \times N}$ using the `select` operator. Unlike in categorical attention, each query can attend to more than one key, and queries can attend to nothing if there is no matching key. Given attention scores $\mathbf{S} \in \{0,1\}^{M \times N}$ and value variable `var`, the output for the i^{th} token is defined as $\sum_{j=1}^{N} \mathbf{S}_{i,j} \texttt{var}[j]$. At the input layer, there is a single numerical variable, `ones`, which is frozen and equal to 1 for all positions; attention heads that read `ones` as the value variable are equivalent to the `selector_width` primitive in RASP. At higher layers, numerical attention heads can read the output of lower-layer attention heads as values. Figure 8 depicts a numerical attention head from a program for the Double Histogram task.

Numerical attention does not directly correspond to attention in a standard Transformer, because it computes a weighted sum rather than a weighted average. But a numerical attention head can be implemented in a standard Transformer by composing an attention head with a feed-forward layer, using the beginning-of-sequence token to allow the model to attend to nothing. This is how the `selector_width` primitive is implemented in Tracr [Lindner et al., 2023].

Feed-forward layers. In RASP, feed-forward layers are used to implement arbitrary element-wise operations, and they play an important role in many human-written RASP programs. In our Transformer Programs, we restrict the capacity of the feed-forward layers to ensure that they can be decompiled. Each feed-forward layer reads ℓ input variables, which are designated in advance to be either numerical or categorical variables, and outputs one new categorical variable. We convert feed-forward layers to programs by enumerating all possible inputs and forming a lookup-table. For all experiments, we set $\ell = 2$. For categorical attention, given variable cardinality k, there are k^2 possible inputs. An example of a feed-forward layer is given in Figure 9, which depicts a circuit in a program for the Dyck-2 task.

For numerical attention, we can bound the input range by determining the cardinality of the numerical attention heads. For each numerical attention head, the minimum output value is equal to 0 and the maximum output value is equal to the maximum input length multiplied by the cardinality of the input variable (that is, the value variable). For example, if an attention head reads `ones` as the value

```python
def num_predicate_0_1(q_token, k_token):
    if q_token in {"0"}:
        return k_token == "0"
    elif q_token in {"1"}:
        return k_token == "1"
    elif q_token in {"2"}:
        return k_token == "2"
    elif q_token in {"3"}:
        return k_token == "3"
    elif q_token in {"4"}:
        return k_token == "4"
    elif q_token in {"5"}:
        return k_token == "5"
    elif q_token in {"<s>"}:
        return k_token == "<pad>"

num_attn_0_1_pattern = select(
    tokens, tokens, num_predicate_0_1)
num_attn_0_1_outputs = aggregate_sum(
    num_attn_0_1_pattern, ones)
```

```python
def num_mlp_0_1(num_attn_0_1_output):
    key = num_attn_0_1_output
    if key in {0, 1}:
        return 4
    return 0

num_mlp_0_1_outputs = [
    num_mlp_0_1(k0)
    for k0 in num_attn_0_1_outputs]
```

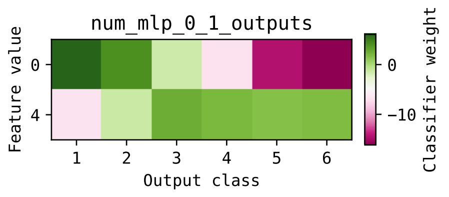


Figure 8: An example of a numerical attention head and MLP in a program for the Double Histogram task. For this task, the model must output, for each position, the number of unique tokens with the same histogram value. In this example, an attention head (*left*) calculates the histogram for each position. An MLP (*top right*) reads the histogram values and outputs a value of `0` if the histogram value is greater than one, and `4` otherwise. Inspecting the corresponding classifier weights (*bottom right*), we see that an output value of `0`—meaning a histogram count greater than `1`—increases the likelihood that the double-histogram value is `1` or `2`, and decreases the likelihood of larger values. Because the input length is limited to 8, this reflects the fact that if one number appears many times, it is unlikely that another number appears the same number of times. An output of `4` (meaning a histogram count of `1`) increases the likelihood that the double-histogram is greater than `1`. Note that we configure all MLPs to read two input variables, but some MLPs learn to read the same variable for both inputs, as in this example. This allows us to compress the corresponding function.

variable, the maximum output value is equal to the maximum input length. If an attention head reads the output of a first-layer attention head as value, the maximum output value is equal to the square of the maximum input length.

A.3 Extracting programs

In this section, we provide more details about our procedure for converting our trained models into Python programs. While there are many possible ways to express the discretized model as Python code, we choose a mapping that facilitates analysis with a standard Python debugger. We use several simple strategies to improve the readability of the code, by annotating variable types, compressing statements, and removing unreachable branches. Illustrative programs are depicted in Figures 8 and 9.

Attention. Each attention head is represented by a `predicate` function, which takes as input a key and query and outputs a value in $\{0, 1\}$. In Transformer Programs, all keys and queries are categorical variables with cardinality k, so the `predicate` function can be defined by enumerating the possible query values, which we do with a series of `if` statements. Additionally, if multiple query values are mapped to the same key, we condense the `predicate` by combining these queries into a single branch. This is illustrated in Figure 9. In the first attention head (*left*), each query position attends to the previous position (with the exception of the first token), so we cannot apply any compression. In the second attention head (*bottom right*), fifteen out of the sixteen possible query values are mapped to a single key value, so we combine them into a single branch.

MLPs. We convert feed-forward modules to functions by enumerating all possible inputs and forming a key/value lookup-table. For all experiments, we set each MLP to read $\ell = 2$ input variables. In some cases, the MLP learns to read the same variable for both input variables, allowing us to reduce the length of the function. We further reduce the length of these functions by identifying the MLP output value associated with greatest number of keys, and returning this value as the default value without explicitly evaluating the corresponding condition. These two forms of compression are illustrated in Figure 8 (*top right*). This MLP reads a single numerical input variable and outputs a value of `4` if the input is `0` or `1` and a value of `0` otherwise.

```python
# First attention head: copy previous token.
def predicate_0_0(q_position, k_position):
    if q_position in {0, 13}:
        return k_position == 12
    elif q_position in {1}:
        return k_position == 0
    elif q_position in {2}:
        return k_position == 1
    elif q_position in {3}:
        return k_position == 2
    elif q_position in {4}:
        return k_position == 3
    elif q_position in {5}:
        return k_position == 4
    elif q_position in {6}:
        return k_position == 5
    elif q_position in {7}:
        return k_position == 6
    elif q_position in {8}:
        return k_position == 7
    elif q_position in {9}:
        return k_position == 8
    elif q_position in {10}:
        return k_position == 9
    elif q_position in {11}:
        return k_position == 10
    elif q_position in {12}:
        return k_position == 11
    elif q_position in {14}:
        return k_position == 13
    elif q_position in {15}:
        return k_position == 14
attn_0_0_pattern = select_closest(positions, positions,
                                  predicate_0_0)
attn_0_0_outputs = aggregate(attn_0_0_pattern, tokens)
```

```python
# MLP: reads current token and previous token
# Outputs 13 if it sees "(}" or "{)".
def mlp_0_0(token, attn_0_0_output):
    key = (token, attn_0_0_output)
    if key in {(")", ")"),
               (")", "}"),
               ("{", ")"),
               ("}", ")"),
               ("}", "}")}:
        return 4
    elif key in {(")", "{"),
                 ("}", "(")}:
        return 13
    elif key in {("(", ")"),
                 ("(", "}"),
                 (")", "("),
                 ("{", "}"),
                 ("}", "{")}:
        return 0
    return 7
mlp_0_0_outputs = [
    mlp_0_0(k0, k1) for k0, k1 in
    zip(tokens, attn_0_0_outputs)
]

# 2nd layer attention: check for "(}" or "{)"
def predicate_1_2(position, mlp_0_0_output):
    if position in {0, 1, 2, 4, 5, 6, 7, 8, 9,
                    10, 11, 12, 13, 14, 15}:
        return mlp_0_0_output == 13
    elif position in {3}:
        return mlp_0_0_output == 4
attn_1_2_pattern = select_closest(
    mlp_0_0_outputs, positions, predicate_1_2)
attn_1_2_outputs = aggregate(
    attn_1_2_pattern, mlp_0_0_outputs)
```

Figure 9: An example of a circuit in a program for Dyck-2. For this task, the inputs consist of strings from the vocabulary $\{$`(`, `)`, `{`, `}`$\}$. At each position i, the model must output `T` if the string up to position i is a valid string in Dyck-2; `P` if it is the prefix of a valid string; and `F` otherwise. A string is valid if every parenthesis is balanced by a parenthesis of the same type. This circuit recognizes an invalid pattern, where a left parenthesis (`(` or `{`) is immediately followed by a right parenthesis of the wrong type (`}` or `)`, respectively). First, an attention head (*left*) copies the `tokens` variable from the previous position. Second, an MLP (*top right*) reads the output of this attention head, along with the `tokens` variable, and classifies the input into one of four categories—in particular, returning `13` if it sees the pattern `(}` or `{)`. In the second layer, another attention head (*bottom right*) looks for `13`s, propagating this information to later positions.

Annotating variable types. In our Transformer model, all categorical variables are encoded using one-hot embeddings, which are represented as integers in the corresponding program. To improve readability, we replace these integers with symbolic values where appropriate. At the input layer, we represent the values of the `tokens` variable as strings rather than integer indices. At subsequent layers, we determine the appropriate type by following the computation graph. For example, in Figure 9, the `tokens` variable takes on values in $\{$`(`, `)`, `{`, `}`$\}$; the first attention head reads `tokens` as the value variable, so we can automatically determine that `attention_0_0_outputs` variable takes on values of the same type; finally, the MLP reads `tokens` and `attention_0_0_outputs` as input variables, so we define the `mlp_0_0` function in terms of the `token` value type.

B Experiment details

In this section we describe additional implementation details for the experiments in Section 4. We will publish all code needed to reproduce the experiments following the review period.

B.1 In-context learning

Data. For our in-context learning experiment (Section 4.1), we sample 20,000 sequences of length 10. The first token is a beginning-of-sequence token, and the remainder of the sequence is formed by randomly sampling a many-to-one mapping between letter types (either `a`, `b`, `c`, `d`) and numbers (`0`, `1`, `2`, `3`), and then alternating letters and numbers until reaching the target length. The model is trained to predict the number following each letter, or a special `unk` token if the letter has not appeared earlier in the sequence. We use an autoregressive mask so that at each position i, the model can attend only to keys with positions $j \leq i$.

Training. We train each model for 250 epochs with a batch size of 512, a learning rate of 0.05, and annealing the Gumbel temperature geometrically from 3.0 to 0.01, decreasing the temperature at each training step. We take one Gumbel sample per step. The model has two layers with one categorical attention head per layer, with a variable cardinality of 10. We train five models with different random initializations and pick the one with the lowest test loss. We implement all models in PyTorch [Paszke et al., 2019] and use the Adam optimizer [Kingma and Ba, 2014].

B.2 RASP tasks

Data. For each RASP task, we sample 20,000 inputs without replacement and partition them into train, validation, and test sets containing 16,000/2,000/2,000 instances respectively. With the exception of the Dyck tasks, we sample inputs by sampling tokens uniformly from the vocabulary. For the Dyck tasks, we follow Weiss et al. [2021] and sample with a bias towards strings with a valid prefix. Specifically, given maximum length of N, with probability 0.5, we sample N tokens uniformly from the vocabulary; otherwise, we sample a valid Dyck string s with length $|s| \leq N$, and append $N - |s|$ randomly sampled tokens to the end to obtain a string with length N. For all tasks, we obtain 20,000 samples without replacement by sampling strings uniformly until the set of unique strings has the intended size. We prepend all inputs with a beginning of sequence token `bos`. For the `sort` and `reverse` tasks, we also append an end-of-sequence token, `eos`. Following Weiss et al. [2021], we report the token-level accuracy.

Training. As above, we train the model for 250 epochs with a batch size of 512, and a learning rate of 0.05. We anneal the Gumbel temperature geometrically from 3.0 to 0.01, decreasing the temperature at each training step, and taking one Gumbel sample per step. These hyperparameters were chosen after initial experiments on the RASP tasks. We do a grid search for number of layers (2, 3), number of attention heads (4, 8), and number of MLPs per layer (2, 4). The attention heads are evenly divided between categorical and numerical heads. Similarly, the MLPs are evenly divided between MLPs with two categorical inputs, and MLPs with two numerical inputs. We train models with five random initializations, pick the model with the best accuracy on a validation set, and report the accuracy on a held-out test set. Each model takes between five and fifteen minutes to train on an Nvidia RTX 2080 GPU, depending on the number of layers.

B.3 Named entity recognition

Data. We train on data from the CoNLL-2003 shared task [Sang and De Meulder, 2003], using the distribution from HuggingFace Datasets [Lhoest et al., 2021]. This data is pre-tokenized and we filter the dataset to sentences with a maximum length of 30 tokens and add special beginning- and end-of-sequence tokens. Following Collobert et al. [2011], we preprocess the data by replacing all contiguous sequences of numbers with a special number symbol, so, for example, the string “19.99” is replaced with “#.#”. We use the standard train/validation/test split and evaluate the results using a Python implementation of the standard CoNLL evaluation script [Nakayama, 2018].

Training. For both the standard Transformer and Transformer Programs, we use a batch size of 32 and perform a grid search over the number of layers (1, 2) and the number of attention heads (4, 8). For the standard Transformer, we train for up to 100 epochs, taking a checkpoint after each epoch and picking the checkpoint with the highest performance on the validation set. For the Transformer Program, we search for the number of training epochs (30, 50, 100); as above, we anneal the temperature geometrically from 3.0 to 0.01, and we report results after discretizing the model at the end of training.

C Additional results

C.1 Additional RASP results

In this section, we provide additional code examples and ablations on the RASP tasks.

Longer sequences. In Table 3, we show the accuracy of Transformer Programs trained on RASP tasks with a longer maximum sequence length and a larger input vocabulary. Because we use one-hot encodings for the token and position embeddings, these values determine the cardinality, k, of the

Dataset	$\lvert\mathcal{V}\rvert$=8, N=8	$\lvert\mathcal{V}\rvert$=8, N=16	$\lvert\mathcal{V}\rvert$=16, N=16
Reverse	99.78	72.21	60.27
Hist	100.00	100.0	100.0
2-Hist	99.92	97.72	95.70
Sort	99.97	99.48	91.57
Most Freq	77.34	75.73	53.08

Table 3: RASP accuracy after increasing the size of the input vocabularies ($\lvert\mathcal{V}\rvert$) and maximum sequence length (N). The cardinality of categorical variables, k, is set to the maximum of $\lvert\mathcal{V}\rvert$ and N. On most tasks, performance degrades moderately when trained on longer sequences, and degrades more when we also increase the vocabulary size. The largest drop is on the Reverse task.

categorical variables used in the program. In Table 3, we increase the vocabulary size and maximum sequence length from eight to sixteen. All other experiment details are the same as in Appendix B.2. Performance degrades on longer sequences, underscoring some of the optimization challenges in learning Transformer Programs for larger scales.

Dataset	**L**	**H**	**M**	**Acc.**
Reverse	3	8	2	99.90
Hist	3	8	4	83.20
2-Hist	2	8	2	52.51
Sort	3	2	2	99.99
Most Freq	3	8	4	68.47
Dyck-1	3	8	4	99.27
Dyck-2	3	8	2	99.00

Table 4: Accuracy of Transformer Programs on RASP tasks using only categorical variables, with the number of layers (L), attention heads (H), and MLPs (M) used in the best-performing model.

No numerical variables. In our main experiments, we train Transformer Programs with an equal number of categorical attention heads and numerical attention heads, and categorical MLPs and numerical MLPs. In Table 4, we compare results on RASP tasks with only categorical variables. The experiment setting is otherwise the same as in Appendix B.2, but all attention heads and MLPs are constrained to read only categorical variables. Not surprisingly, performance degrades on three tasks that are primarily based on counting: Histograms, Double Histograms, and Most Frequent. However, the Transformer Programs still achieve good performance on the other four tasks. These include the balanced parenthesis languages (Dyck-1 and Dyck-2), which are most naturally solved by keeping a count of unmatched parentheses at each position.

C.2 Analyzing the generated code

Here, we provide some additional analysis of the generated code, focusing on the RASP tasks. Complete, generated programs will be included in our code release.

Program length. Table 5 shows the number of lines in our best-performing program for each RASP task, before and after applying the compression strategies described in Appendix A.3. The program length includes the basic library functions used to We use an automated Python code formatter,[2] which applies a number of standard style conventions and, in particular, enforces a maximum line length of 88 characters.

What information do the attention heads read? Because each attention head reads a fixed set of named variables, we can characterize how information flows through the programs by examining which variables are read by each head. In Figure 10, we summarize this information for RASP programs. At the first layer, the majority of categorical attention heads read `positions` as key and query variables and `tokens` as the value. At higher layers, `positions` remains the most common key variable, but the models are more likely to read the outputs of lower-layer attention heads as the value variable. Numerical attention heads are less likely to read `positions` and more likely to

[2] https://github.com/psf/black

Dataset	Full	Pruned
Reverse	1893	713
Hist	324	160
2-Hist	1309	423
Sort	1503	635
Most Freq	1880	666
Dyck-1	9975	892
Dyck-2	5406	733

Table 5: The number of lines in best programs for each RASP task before and after applying a set of simple pruning strategies based on static analysis of the code.

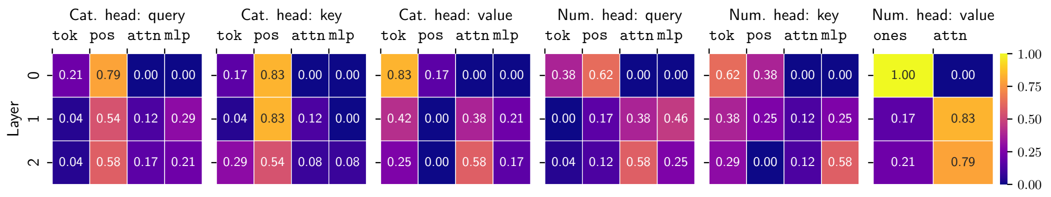


Figure 10: For each of the RASP tasks, we learn a Transformer Program with three layers and eight heads per-layer, divided evenly between categorical and numerical attention heads, and summarize the types of variables that are read at different layers. For each layer, we list the key, query, and value variables read by attention heads at that layer, and calculate the proportion of heads that read the `tokens` variable; `positions`; `ones` (for numerical attention); the output of a previous attention head (`attn`); or the output of a previous MLP (`mlp`). We aggregate over RASP programs and compare categorical attention heads (*left*) and numerical attention heads (*right*).

read `tokens`, `attn`, and `mlp` outputs. Both kinds of attention successfully learn to compose modules, with higher-layer modules reading the outputs of modules at lower layers.

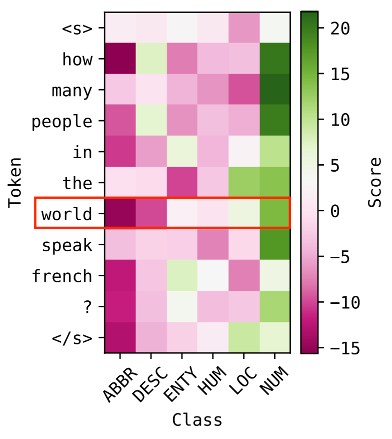
C.3 Text classification

Model	TREC	MR	Subj	AG
Bag of words	74.80	77.8	92.6	89.6
Standard Transformer	87.72	78.9	93.0	91.4
Transformer Program	85.20	77.4	92.9	90.8

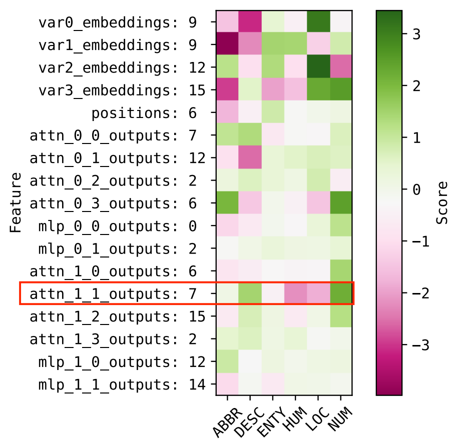
Table 6: Classification accuracy on question classification (**TREC**); sentiment analysis (**MR**); subjectivity classification (**Subj**); and topic classification (**AG news**). The Bag of words baseline is a multinomial Naive Bayes model trained on unigram features.

Next, we train Transformer Programs for three standard text classification datasets: classifying questions as one of six topic [TREC; Voorhees and Tice, 2000]; classifying movie reviews as positive or negative [MR; Pang and Lee, 2005]; classifying sentences as objective or subjective [Subj; Pang and Lee, 2004]; and classifying sentences from news articles as one of four topics [AG News; Zhang et al., 2015]. We filter the datasets to sentences with at most 64 words and use a vocabulary of the 10,000 most common words, replacing the remaining words with an *unknown* token, and fixing the variable cardinality at 64. As above, we compare the Transformer Program with a standard Transformer. For both models, we use the Transformer to extract token embeddings, obtain a sentence embedding by averaging the token embeddings, and train a linear classifier on the sentence embedding. We hold out 10% of the training data, pick the model that performs best on this held-out set, and report the accuracy on the standard test split, averaging the results over five random seeds. We initialize both models with GloVe embeddings and use grid search to select the model dimension, number of layers, and number of heads, and training hyper-parameters (learning rate and number of training epochs), as described in Appendix B.3.

Accuracy. Table 6 compares the accuracy for Transformer Programs, standard Transformers, and a bag-of-words baseline. The Transformer Programs are competitive with thee standard Transformer



(a) Classification scores for each token embedding.

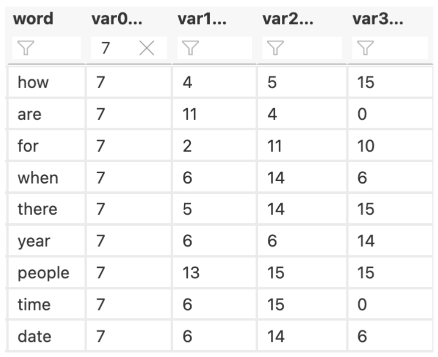


(b) Feature-level scores for the "world" token.

```
# attn_1_1: Copy var0 from early positions.
def predicate_1_1(var2_embedding, position):
    if var2_embedding in {0, 4, 5, 6, 10, 12, 13}:
        return position == 1
    elif var2_embedding in {1, 7, 8, 9, 11, 14}:
        return position == 4
    elif var2_embedding in {2, 3}:
        return position == 5
    elif var2_embedding in {15}:
        return position == 2

attn_1_1_pattern = select_closest(
    positions, var2_embeddings, predicate_1_1)
attn_1_1_outputs = aggregate(
    attn_1_1_pattern, var0_embeddings)
```

(c) The code for computing the `attn_1_1_outputs` feature.



word	var0...	var1...	var2...	var3...
	7			
how	7	4	5	15
are	7	11	4	0
for	7	2	11	10
when	7	6	14	6
there	7	5	14	15
year	7	6	6	14
people	7	13	15	15
time	7	6	15	0
date	7	6	14	6

(d) A subset of the embedding table, filtering to words with `var0_embedding` values of 7.

Figure 11: Visualizing the predictions for an example from the TREC question classification dataset. We classify sequences by pooling the final-layer token embeddings, so we can visualize the classification scores for each token embedding (Figure 11a). In this example, the first three tokens ("how", "many", and "people") have the highest scores in favor of the `NUM` class, but most other tokens have high scores for this class as well. To see why, we can inspect the feature-level scores for individual token embeddings. In Figure 11b, we display the categorical features of the "world" token along with the corresponding classifier weights. For this token, the input features favor the `LOC` label, but attention features increase the score for `NUM`. Figure 11c displays the subset of the program that calculates one of these attention features (`attn_1_1_output = 7`). This attention head generally copies the `var0_embeddings` variable from first or fourth position—positions that can be expected to be informative for question classification. In Figure 11d, we display the most frequent words that have `var0_embedding` values of 7: they include question words like "how" and "when" and nouns like "year", "time", and "date", which are more likely to occur in numerical questions.

on all four datasets, performing slightly worse on the movie review dataset and TREC. This could be because the standard Transformer can more effectively leverage pre-trained word embeddings, or because it is easier to regularize.

Interpretability. We illustrate the interpretability of these programs in Figure 11 by inspecting the features for an example from the TREC dataset.