Penzai + Treescope: A Toolkit for Interpreting, Visualizing, and Editing Models As Data

Daniel D. Johnson [1 2]

Abstract

Much of today's machine learning research involves interpreting, modifying or visualizing models after they are trained. I present *Penzai*, a neural network library designed to simplify model manipulation by representing models as simple data structures, and *Treescope*, an interactive pretty-printer and array visualizer that can visualize both model inputs/outputs and the models themselves. Penzai models are directly structured as compositions of modular operations, and expose the model forward pass in the structure of the model object itself, while also using named axes to ensure each operation is semantically meaningful. Users can insert new logic and extract intermediate values by directly transforming the model object using Penzai's tree-editing selector system, and get immediate feedback by visualizing the modified model with Treescope. I describe the motivation and main features of Penzai and Treescope, and discuss how treating the model as data enables a variety of analyses and interventions to be implemented as data-structure transformations, without requiring model designers to add explicit hooks.

1. Introduction

Due to the increasing capabilities of large language models and other foundation models, and the similarly increasing cost to training them, much research with large models has shifted to after their initial training run. This includes interpreting the "circuits" inside models (e.g. Wang et al., 2022), probing internal representations (e.g. Luo et al., 2021), or fine-tuning models using parameter-efficient adapters to control their behavior (e.g. Hu et al., 2021). Conducting this research often involves either *visualizing* model components, *inserting* new logic to intervene on activations, *replacing* individual model components, or some combination of these.

Unfortunately, the original model representation is often not well-suited for making modifications, as most model codebases are designed for efficient training and inference. And working around these limitations often comes at the expense of readability or missing functionality. As one example, TransformerLens (Nanda & Bloom, 2022) supports a wide set of interventions using a "hooked" reimplementation of common transformer variants, but does not support efficient sampling or multiple-device computation. This kind of analysis has historically been even more complex when using JAX (Bradbury et al., 2018), because hook-based interfaces use global state that is difficult to combine with JAX's purely functional transformations.

Similarly, the original model representation is not usually designed to enable easy visualization. Existing tools, such as the Language Interpretability Tool (Tenney et al., 2020) or the Transformer Debugger neuron viewer (Mossing et al., 2024), have been introduced to help researchers understand model behavior, but tend to support visualizations of only model outputs or specific intermediate values (such as attention patterns) and do not support visualizing the model structure itself. General-purpose plotting libraries like Matplotlib (Hunter, 2007) or Plotly (Plotly Technologies Inc., 2015) tend to prioritize tabular data and are not well-suited to visualizing data involving multidimensional arrays.

This paper describes Penzai, a JAX library focused on making it easier to manipulate complex neural network models and their activations, and Treescope, a Python pretty-printer designed to visualize models and multidimensional-array data. These libraries aim to simplify research into pretrained models by first *treating models as simple data structures*, which are designed to be modular and can be directly manipulated by the user in order to change their behavior, and then *providing general tools for visualizing and editing those data structures* in an interactive setting. This allows Penzai and Treescope to support a "what-you-see-is-what-you-get" research workflow: model interventions can be visualized simply by pretty-printing the modified model, and there is always a direct correspondence between the model's internal structure, the structure of its pretty-printed visualization,

[1]Google DeepMind [2]University of Toronto, Department of Computer Science, Ontario, Canada. Correspondence to: Daniel D. Johnson <ddjohnson@cs.toronto.edu>.

Proceedings of the Mechanistic Interpretability Workshop at the 41^{st} International Conference on Machine Learning, Vienna, Austria. 2024.

and its runtime behavior.

2. Previous Model-Manipulation Strategies

A number of libraries have proposed interfaces for intervening on model activations while a model runs. TransformerLens (Nanda & Bloom, 2022) includes a transformer implementation with **hook points**, and enables users to add hook functions that can read or modify activations. This supports a wide variety of transformations, but also requires users to manage global state of hooks, and does not support efficient sampling or multi-device computation. `pyvene` (Wu et al., 2024) similarly allows modifying internal model activations with new logic, but represents interventions using a **intervention schema** instead of global hooks. NNSight (Fiotto-Kaufman, 2024) uses a **tracing context** to provide an interface where activations of PyTorch models can be extracted and modified based on their location in the model; these modifications are then converted to a graph and evaluated on remote workers. A common difficulty with many of these approaches is that activations must first be located before they can be modified; each transformer implementation and modification library comes with its own conventions and syntax for accessing them.

A different form of model modification is **hot-swapping**, where parts of a model are replaced entirely with other parts. This strategy is sometimes used to add new parameters to pretrained models for parameter-efficient fine-tuning. For instance, the PEFT (Mangrulkar et al., 2022) and Levanter (Stanford CRFM, 2023) libraries replace internal linear layers with low-rank adapter layers to implement LoRA fine-tuning (Hu et al., 2021), although the details of this are mostly abstracted away from the user.

3. Penzai: Treating the Forward Pass as Data

How can we make it as easy as possible for new users to inspect and modify the behavior of models that they did not train? The central idea in Penzai is to focus on decomposing the model object into self-explanatory pieces, and allow the user to directly modify and recombine these pieces in new ways. This removes the need for hooks or tracing, and makes it easy to tell how to change the behavior of a model: simply modify the model object itself.

3.1. Combinators and Primitive Layers

Concretely, Penzai provides a library of simple neural network components that can be combined to implement more complex models. Many of these components are *combinators*, which use child layers to implement more complex behaviors. This includes standard combinators such as `Sequential` (also provided by libraries like PyTorch and Keras), but also more advanced combinators such as `BranchAndMultiplyTogether`, which multiplies the results of running its children in parallel, and `Attention`, which routes intermediates between query, key, and value heads. Importantly, these combinators make minimal assumptions about their children. For instance, the `Attention` combinator has no parameters and is just responsible for routing queries, keys, values, and outputs, not for computing dot products or attention masks; those are implemented by child layers. This means that the same `Attention` combinator is compatible with many common attention variants, such as rotary positional embeddings (Su et al., 2024) or grouped-query attention (Ainslie et al., 2023), without requiring a complex implementation. In addition to these combinators, Penzai also provides components for primitive operations such as `Linear`, `AddBias`, `ApplyCausalAttentionMask`, and `Softmax`. All primitive operations are written in terms of named axes instead of making assumptions about axis positions, to ensure their behavior does not require memorizing array ordering conventions.

To keep model structures as simple as possible, Penzai avoids storing configuration data (such as activation functions or causal decoding flags) as attributes on the model object, and models do not use conditional branching at runtime. Instead, each model's sublayers are specialized for a single configuration, and stores only the information that it needs, using fully-documented and type-annotated attributes. By convention, each layer accepts a single "main input", usually the activations from the previous layer, along with a set of "side input" keyword arguments that are shared across all layers and provide context such as token positions or attention masks. This convention makes it possible to compose layers together in a generic way, with minimal model-specific routing logic.

Users can customize the behavior of models or change their configuration by directly hot-swap different implementations, e.g. replacing each `Attention` combinator with a `KVCachingAttention` combinator to enable fast autoregressive decoding. This means model implementations in Penzai have a one-to-one correspondence between the Python structure of the model object and the computations that run during its forward pass. Users are free to change how their model works without using hooks or intervention schemas, by instead directly inserting a new primitive into the model at the relevant place, and using pretty-printing to identify which locations they need to modify. An example Transformer block in Penzai is shown in Figure 1.

3.2. Lightweight Named Axes System

Giving axes names makes it easier to identify the purpose of each axis, prevents users from needing to memorize specific ordering conventions, and simplifies visualizing array data.

Figure 1. A partially-expanded Treescope rendering of a Transformer block from Penzai's implementation of the Gemma 7B model (Gemma Team, 2024), showing the `pz.nn.Attention` combinator and some of the primitive sublayers it contains.

Unfortunately, named axis implementations often introduce additional implementation burden due to changing the semantics of each individual operation. To give the benefits of named axes without requiring changes to the array API, Penzai includes a lightweight named axis system based on *locally-positional semantics* and inspired by Chiang et al. (2021). Users interact with this system by constructing positional *views* of a subset of axes in an array, and then applying ordinary JAX operations to these views, which are then "lifted" across all other axes automatically. Concretely, each axis of a Penzai `NamedArray` has either a position or a name (but not both). Individual named axes can be converted to positional views using `.untag(...)`, and JAX functions can be vectorized over remaining named axes using `pz.nx.nmap`; later, the positional view axes can be re-bound to names using `.tag(...)`. Thus, computations like "take a softmax over axis `foo`" can be expressed as

```
pz.nx.nmap(jax.nn.softmax)(
  array.untag("foo"), axis=0
).tag("foo")
```

without requiring a named-axis-specific implementation of `softmax`. This means Penzai's named axis system is compatible with the full JAX (and, thus, Numpy) array APIs.

3.3. Models Are JAX Pytrees, Plus Mutable State

Similar to Equinox (Kidger & Garcia, 2021), each of Penzai's layer classes is immutable and is registered as a JAX pytree[1], which makes it possible to manipulate using common JAX utilities. However, mutability can be useful for implementing parameter sharing, per-layer state, and extraction of intermediate activations. To support these use cases, Penzai layers are allowed to store *mutable variables* as attributes, which come in two forms: `Parameter`s, which are updated by optimizers, and `StateVariable`s, which are usually updated during each layer's forward pass. Both types of variable can be freely shared between layers.[2]

To allow them to be safely used with JAX's function transformations, variables can be "frozen" before applying a function transformation, which turns them into ordinary immutable JAX pytrees. These frozen variables can then be temporarily "unfrozen" (which creates new mutable copies) inside the transformed function, then frozen again before returning from the function, resulting in a purely-functional view of the model's behavior. Frozen parameters can also be directly embedded into the model once they no longer need to be updated.

3.4. Selectors Enable Flexible Tree Modifications

To help users modify Penzai models, Penzai includes a powerful tree-rewriting utility, `pz.select`, which "selects"

[1] https://jax.readthedocs.io/en/latest/pytrees.html

[2] Penzai's original "V1" neural network system did not store mutable variables in the model, and instead expressed parameter sharing and state using a "data-effect" system, which rewrote the model structure when the model was called. This paper describes the newer "V2" design, which directly supports mutable state.

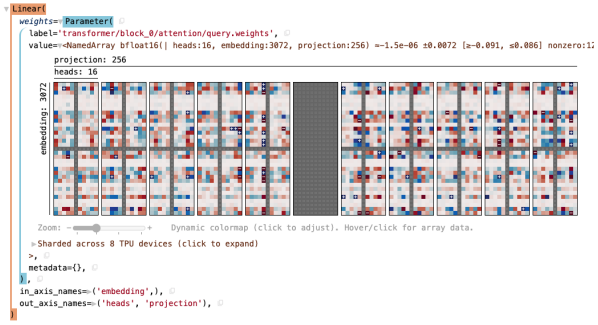

Figure 2. When pretty-printing a Penzai `Linear` layer, Treescope renders an inline faceted visualization of the parameter array.

parts of data structures by type or position. For instance,

```
pz.select(model)
  .at_instances_of(pz.nn.Attention)
  .at_instances_of(pz.nn.Softmax)
  .insert_after(some_new_layer)
```

will return a copy of the model with new logic (`some_new_layer`) after each attention pattern computation, making it possible to retrieve or modify their values. It is similarly possible to e.g. swap out pretrained `Linear` layers for low-rank finetuning layers using code like

```
pz.select(model)
  .at_instances_of(pz.nn.Linear)
  .apply(loraify)
```

`pz.select` uses the JAX pytree registry to support modifying any JAX-compatible object. Because Penzai models are JAX pytrees, `pz.select` can be used to perform arbitrary modifications to Penzai models. And because Penzai models intentionally expose the model forward pass using combinators, and decompose operations into independent, semantically meaningful chunks, users are free to intervene at arbitrary points and insert logic similar to hook-based approaches. Direct model editing also supports a wider set of transformations than hook points, such as replacing individual model components with linear approximations.

4. Treescope: Automatic Visualization of Models and Array Data

Exploratory research with neural networks often involves manipulating deeply-nested data structures containing multidimensional arrays, which can be difficult to summarize and visualize. When using Penzai's model-editing tools to change the structure of a complex model, visualizing the modified structure can also be very useful to confirm that the correct modification was made.

The Treescope pretty-printer simplifies this process by automatically producing rich interactive visualizations of deeply-nested machine learning data structures, including both model objects and their inputs and outputs. Treescope directly integrates with Jupyter IPython notebook environments (Kluyver et al., 2016), and includes an automatic array visualizer, which renders faceted summaries of multi-dimensional array shapes and values and directly embeds them into the pretty-printed output. To prevent outputs from becoming too large, Treescope automatically truncates the array contents to show only a small slice (similar to the ordinary `repr`), but also adds a summary of the distribution of values across the entire array. An example rendering of a `Linear` layer with Treescope, including an inline array visualization, is shown in Figure 2.

Users can click Treescope renderings to fold or unfold individual components, allowing them to "drill down" into components of interest.[3] Treescope also inserts "copy path" buttons at every level of the printed tree, which show how to extract the clicked component from the original object for further manipulation. And in "roundtrip mode" (toggled by pressing the `r` key), Treescope adds fully-qualified names to all classes, making it possible to directly execute the pretty-printed code to rebuild supported data structures.

Treescope and Penzai were designed together, and Penzai's model components are specifically implemented to be easy to visualize and explore in Treescope. In particular, Penzai models support *fully-roundtrippable pretty-printing*: the pretty-printed output of a Penzai model is enough to rebuild the model architecture even after modifications are made by the user. However, Treescope also supports visualizing models and data structures from other neural network libraries. In particular, Treescope supports rendering arbitrary JAX pytrees and models built using the JAX libraries Equinox and `flax.nnx` (even when model weights are sharded across multiple accelerators), and recently added partial support for rendering PyTorch modules and tensors as well. Treescope also includes an extension system, which makes it possible for users to add rich visualization support for new types, or to replace Treescope's automatic array visualization with different types of inline figure. Ultimately, the goal is to allow Treescope's visualization system to be used in combination with other interpretability tools, without having to first rewrite existing models using Penzai.

5. Using Penzai and Treescope for Interpretability Research

5.1. Transformer Implementation

As a starting point for research into interpreting and controlling model behaviors, Penzai includes a generic Transformer (Vaswani et al., 2017) implementation, which uses

[3] This interface was inspired by similar interfaces in the JavaScript console in Google Chrome and Firefox.

Figure 3. A modified Transformer block, where the feed-forward layer has been replaced with a `LinearizeAndAdjust` combinator (which computes a linear approximation of its target layer) and a `RewireComputationPaths` operation (which copies activations across a named "worlds" batch axis).

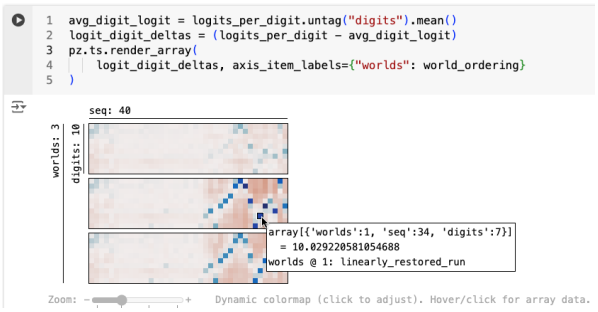

Figure 4. A visualization of a rank-3 array of logit differences using Treescope (from the intervention in Figure 3), with a mouse tooltip giving more information about a specific array element.

Penzai's combinators to directly mirror the transformer forward pass in the the model's structure. This implementation supports loading a variety of pretrained models, including Gemma 2B and 7B (Gemma Team, 2024), Llama 1, 2, and 3 (Touvron et al., 2023a;b), Mistral 7B (Jiang et al., 2023), and the Pythia scaling suite (Biderman et al., 2023). Following Penzai's design conventions, each of these architecture variants corresponds to a different specialization of the same `TransformerLM` base class (and common components `TransformerBlock`, `Attention`, and `TransformerFeedForward`). Architectural differences are encoded by using different sublayer arrangements, and each model can be freely reconfigured by the user. Capturing or intervening on intermediates, fine-tuning, low-rank adaptation, and sampling are all supported, and can even be combined with each other. Additionally, due to JAX's simple APIs for multi-device computation, all of these modifications can be seamlessly distributed across multiple accelerator devices.

5.2. Utilities for Common Operations

Penzai includes a collection of extra utilities in `penzai.toolshed`, including tools for patching individual activations between counterfactual model inputs (see Zhang & Nanda, 2023). Unlike hook-based workflows, these tools work by inserting new layers into the model, resulting in a copy of the model that includes the given intervention, and avoiding the need to manage global state. Directly editing the model structure also enables interventions that cannot be expressed easily with hooks, such as linearizing parts of a model for easier analysis (shown in Figures 3 and 4).

`penzai.toolshed` also includes utilities for basic training, low-rank finetuning, shape annotation, and multi-device sharding. Each utility has a self-contained implementation as a model transformation, and can either be used as-is or taken as a starting point for more complex workflows.

5.3. Example: Finding Induction Heads In Gemma 7B

Penzai includes a tutorial notebook walking users through the process of analyzing the Gemma 7B open-weights model (Gemma Team, 2024) in a Colab notebook[4], starting with exploring the model's structure and predictions on simple examples, then visualizing attention patterns throughout the model to identify candidate induction heads, and finally ablating and patching them to confirm that they are responsible for the copying behavior, while using Treescope to get quick interactive feedback throughout the process.

Because it uses JAX as a backend, exploration in Penzai can immediately benefit from many of JAX's features. Each step in the notebook is seamlessly parallelized across multiple TPU devices. Additionally, the effects of individual MLP layers can be decomposed into linear and nonlinear components using JAX's linearization transform `jax.jvp`.

6. Discussion

By representing models as compositions of simple building blocks, and providing powerful tools for both visualizing and editing these data structures, Penzai and Treescope can support a wide variety of use cases without restricting expressivity. These libraries are under continued development, and I hope they can serve as a foundation for new research on interpreting machine learning models, understanding their training dynamics, and steering their behaviors.

Acknowledgements

Building Penzai and Treescope would not have been possible without the help of Dougal Maclaurin, who supported the open-source release and gave useful feedback throughout the development process. I would also like to thank Danny Tarlow, Hugo Larochelle, David Duvenaud, and Chris Maddison for their advice and encouragement.

[4] https://penzai.readthedocs.io/en/stable/notebooks/induction_heads.html

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. GQA: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

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

Chiang, D., Rush, A. M., and Barak, B. Named tensor notation. *arXiv preprint arXiv:2102.13196*, 2021.

Fiotto-Kaufman, J. nnsight: The package for interpreting and manipulating the internals of deep learned models. , 2024. URL https://github.com/JadenFiotto-Kaufman/nnsight.

Gemma Team. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Kidger, P. and Garcia, C. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *arXiv preprint arXiv:2111.00254*, 2021.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., et al. Jupyter notebooks-a publishing format for reproducible computational workflows. *Elpub*, 2016:87–90, 2016.

Luo, S., Ivison, H., Han, S. C., and Poon, J. Local interpretations for explainable natural language processing: A survey. *ACM Computing Surveys*, 2021.

Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., and Bossan, B. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

Mossing, D., Bills, S., Tillman, H., Dupré la Tour, T., Cammarata, N., Gao, L., Achiam, J., Yeh, C., Leike, J., Wu, J., and Saunders, W. Transformer debugger. https://github.com/openai/transformer-debugger, 2024.

Nanda, N. and Bloom, J. Transformerlens. https://github.com/TransformerLensOrg/TransformerLens, 2022.

Plotly Technologies Inc. Collaborative data science, 2015. URL https://plot.ly.

Shazeer, N. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Stanford CRFM. Levanter: Legible, Scalable, Reproducible Foundation Models with Named Tensors and Jax , 2023. URL https://github.com/stanford-crfm/levanter.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., et al. The language interpretability tool: Extensible, interactive visualizations and analysis for nlp models. *arXiv preprint arXiv:2008.05122*, 2020.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

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

Wang, K., Variengien, A., Conmy, A., Shlegeris, B., and Steinhardt, J. Interpretability in the wild: a circuit for

indirect object identification in gpt-2 small. *arXiv preprint arXiv:2211.00593*, 2022.

Wu, Z., Geiger, A., Arora, A., Huang, J., Wang, Z., Goodman, N. D., Manning, C. D., and Potts, C. pyvene: A library for understanding and improving PyTorch models via interventions. *arXiv preprint arXiv:2403.07809*, 2024.

Zhang, B. and Sennrich, R. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Zhang, F. and Nanda, N. Towards best practices of activation patching in language models: Metrics and methods. *arXiv preprint arXiv:2309.16042*, 2023.

A. Additional Penzai and Treescope Visualizations

Figure 5. The Gemma 7B open-weights model (Gemma Team, 2024), loaded using Penzai's transformer implementation and visualized using Treescope. The mouse cursor is hovering over a "copy path" button, which copies the location of the selected object to the clipboard when clicked.

Figure 6. Comparison of Transformer block structures for GPT-NeoX/Pythia (Biderman et al. 2023, left) and Llama (Touvron et al. 2023a, right) architectures. GPT-NeoX runs the attention and feedforward parts in parallel, and uses LayerNorm (Ba et al., 2016) and a standard MLP network with biases. In contrast, Llama uses two separate residual blocks, and uses RMSNorm (Zhang & Sennrich, 2019) and a gated feedforward network (Shazeer, 2020) with no learned bias terms. These differences can be concisely expressed using different arrangements of Penzai combinators. (Any class with a "`# Sequential`" annotation simply runs its children in order, without custom logic. Subclasses of `Sequential` are often used to improve readability and allow manipulation by `pz.select`.)

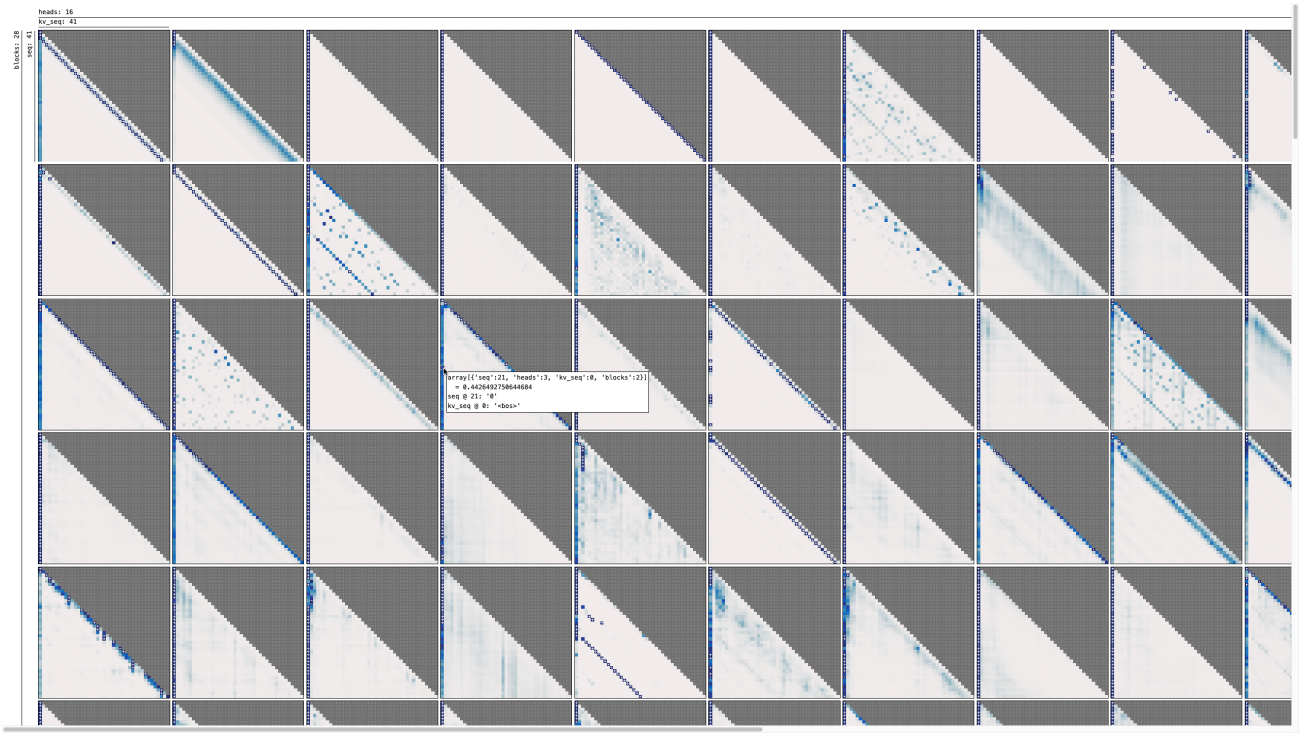

Figure 7. A faceted rendering of attention patterns on an example sequence, produced by Treescope. The hover tooltip gives information about the specific tokens being attended to by the head under the mouse cursor.

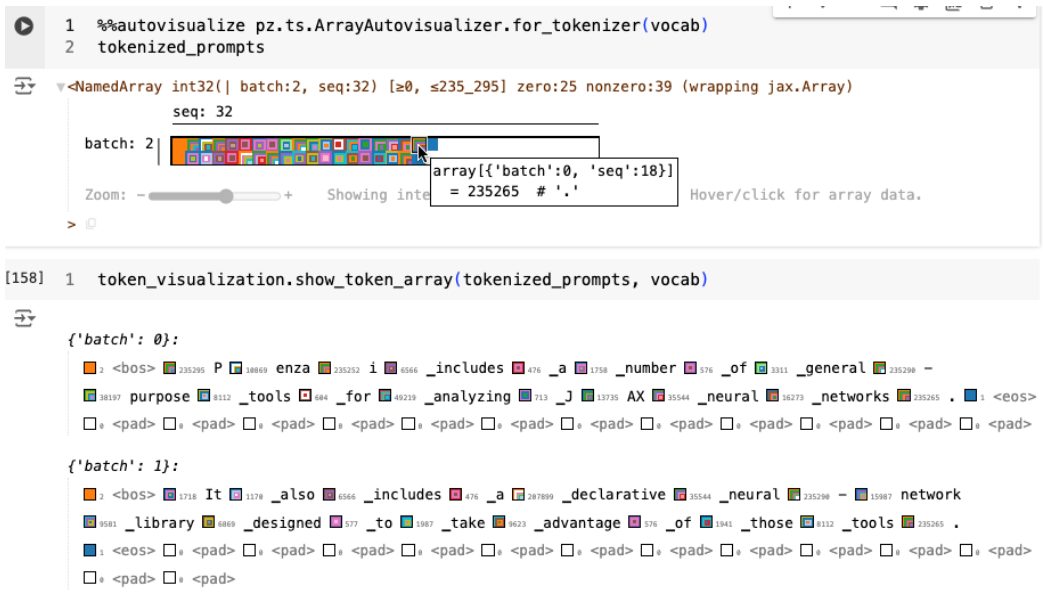

Figure 8. Two renderings of a batch of token sequences. The first uses Treescope's default array visualizer for discrete data, which maps each token ID to a unique "digitbox" pattern, where each color corresponds to one digit of the value. The second interleaves these with the token values, using a token-visualization helper function. Control tokens and padding are easily recognizable across the two sequences due to having single-digit token IDs, which map to solid-color box renderings.