`openretina`: Collaborative Retina Modelling Across Datasets and Species

Federico D'Agostino[*1], Thomas Zenkel[*2,3], Dominic Gonschorek[2,3], Larissa Höfling[&1], Thomas Euler[&2,3], and Matthias Bethge[&1]

[*]Equal contribution
[&]Equal contribution
[1]Tübingen AI Center, University of Tübingen, Germany
[2]Institute of Ophthalmic Research, University of Tübingen, Germany
[3]Werner Reichardt Centre for Integrative Neuroscience, University of Tübingen, Germany

Abstract

Studying the retina plays a crucial role in understanding how the visual world is translated into the brain's language. As a stand-alone neural circuit with easily controllable input, the retina provides a unique opportunity to develop a complete and quantitatively precise model of a computational module in the brain. However, decades of data and models remain fragmented across labs and approaches. To address this, we have launched an open-source retina modelling platform on a shared GitHub repository, aiming to provide a unified data and modelling framework across species, recording techniques, stimulus conditions, and use cases. Our initial release consists of a Python package, `openretina`, a modelling framework based on PyTorch, which we designed for optimal accessibility and extensibility. The package includes different variations on a basic "Core + Readout" model architecture, easily adaptable dataloaders, integration with modern deep learning libraries, and methods for performing *in-silico* experiments and analyses on the models. We illustrate the versatility of the package by providing dataloaders and pre-trained models for data from several laboratories and studies across species. With this starter pack in place, `openretina` can be used within minutes. Through step-by-step examples, we here provide retina researchers of diverse backgrounds a hands-on introduction to modelling, including using models as tools for visualising retinal computations, generating and testing hypotheses, and guiding experimental design.

Retina | Deep Learning | Computational Neuroscience

Code **Website** **Data**

1. Introduction

`openretina` is an accessible, modular, and extensible Python package for training and using artificial neural network (ANN) based computational models of the retina. Along with an initial collection of datasets and pre-trained models, this tutorial-style paper provides an easy entry point into retina modelling for researchers from diverse backgrounds.

This paper introduces `openretina` as a stand-alone Python package [1]. Yet, beyond that, it is also the first step in a broader effort: OpenRetina, an initiative to develop an open, collaborative platform for computational retina research. By providing standardised access to datasets, models, and tools, OpenRetina aims to make retinal system identification more accessible, reproducible, and scalable. The package presented here serves as an initial framework, designed to lower the barrier to entry and provide a common foundation for building and comparing models.

1.1. Background

Modelling the retina has been a long-standing challenge in computational and systems neuroscience. Research efforts towards understanding the complexities of retinal image processing have driven advancements in computational modelling techniques, which in turn have enhanced our understanding of the visual system.

Foundational work established mathematical descriptions of retinal function, such as spatio-temporal filtering models [2], which characterized how different cell types process visual information. Later, statistical models such as linear-nonlinear-Poisson (LNP) models [3] and generalized linear models [GLMs, 4] provided a more refined, probabilistic framework for predicting retinal responses. These approaches laid the groundwork for modern system identification techniques but remained inherently limited in their ability to capture the full nonlinear dynamics of retinal ganglion cell (RGC) responses.

A turning point has been the recent adoption of deep learning tools, particularly convolutional neural networks (CNNs). A lot of research in the retina [e.g. 5–12] and visual cortices [e.g. 13–17] has gone into employing ANNs to vastly improve the predictive accuracy, expressivity, and interpretability of system identification models. Once trained, these models can further be used to

synthesise a variety of stimuli to characterise neurons and neural populations [18–22], and to generate functional hypotheses *in silico* [23].

1.2. The package

`openretina` makes the toolbox of ANN models available to researchers from various backgrounds. By integrating dataset access, model training, and analysis in a unified framework, our package offers a streamlined approach to modelling retinal responses with state-of-the-art techniques (Fig. 1e). We take inspiration for our Python package from other neuroscience deep learning toolkits, such as `neuralpredictors` [24–26, on which we base part of our codebase], and `torch-deep-retina` [5, 10, which also provides data-driven models of the retina]. However, we differentiate ourselves by offering an end-to-end solution designed to facilitate model benchmarking and community-driven development, easy dataloading, and model interpretation.

First, `openretina` directly integrates multiple publicly available retinal datasets, making them immediately usable for training neural networks on a wide range of neural responses across organisms (Fig. 1a), recording modalities (Fig. 1b), and stimuli (Fig. 1c). This will allow researchers to develop more generalisable models, while facilitating cross-dataset comparisons. Additionally, the package includes pre-trained model checkpoints for each dataset, allowing users to quickly apply these models without any retraining.

Second, beyond model training, `openretina` includes visualisation tools for dataset exploration and *in-silico* analysis tools to probe and interpret learned representations. Users can visualise model weights, synthesise maximally exciting inputs that reveal feature selectivity, and extend these optimisation objectives to explore response properties in greater detail (Fig. 1d).

Ultimately, OpenRetina is intended to become a central hub for neural network modelling of retinal data, offering a growing collection of datasets and tools for model development that bridges across research laboratories. The `openretina` package is the first building block of this effort. We welcome collaboration and contributions, and are committed to providing an intuitive interface for both machine learning researchers and neuroscientists.

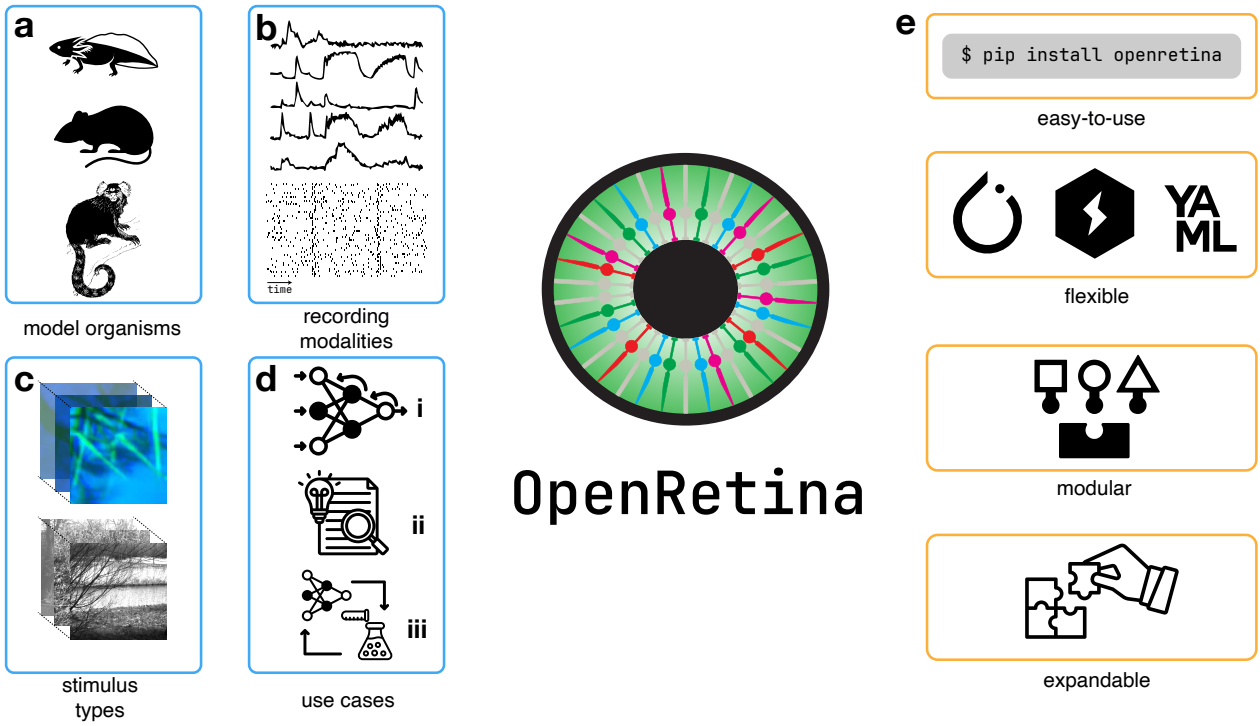

Fig. 1. `openretina` - a Python package for ANN-based computational models of the vertebrate retina. (a) `openretina` supports modelling a variety of experimental data across different model organisms, **(b)** recording modalities (e.g. electrophysiological or imaging recordings), and **(c)** stimuli (e.g. colour, greyscale, images, movies). **(d)** `openretina` allows researchers to use existing models or train their own (**d (i)**); to use models for *in-silico* testing of hypotheses about retinal computations (**d (ii)**), and guiding the design of new experiments (**d (iii)**). **(e)** The package is designed to be: easy to use, offering an easy entry point into retina modelling; flexible, using modern deep learning tools; modular, featuring model architectures that allow combining different components (core + readout); and expandable, allowing researchers to contribute new datasets and models and increasing the value of `openretina` as a resource for the retina research community.

1.3. Paper structure

This work provides a current overview of the capabilities of `openretina`, reflecting its present state while laying the foundation for future iterations and improvements.

We begin with the installation options for the package, together with a brief survey of its main capabilities (Sec. 2).

In Section 3 we then outline the process of training a retina model, covering everything from dataset integration (Sec. 3.1), model architecture (Sec. 3.2), and evaluation (Sec. 3.3).

Equipped with a trained model, we then explore methods for interpreting its learned representations and extracting insights into retinal computations (Section 4). Given `openretina`'s open and extensible nature, we also detail how researchers can contribute by adding new datasets and models to expand its utility (Sec. 5).

Finally, in Section 6, we conclude with a discussion on the broader implications of collaborative retina modelling and the future directions of this initiative.

2. Getting started

To get started with `openretina`, you need a Python installation of version 3.10 or higher. We recommend setting up a dedicated virtual environment or using a containerised setup to avoid dependency conflicts. Users can then choose one of the following two options to install our package.

Recommended option: If you would like to train your own models, modify configurations, and follow the accompanying Jupyter notebooks, we recommend cloning our GitHub repository and installing from there:

```
git clone "git@github.com:open-retina/open-retina.git"
cd open-retina
pip install -e .
```

This option ensures access to all training configuration files and example notebooks. It is the preferred approach for users who want full flexibility in working with the package.

Alternative option: If you only want to use the pre-trained models and intend to use `openretina` "as-is" for analysis and exploration rather than extending it, you can install the package directly from the Python package index (PyPI) by running the command:

```
pip install openretina
```

Both installation options will give you access to the core functionality of the package, and all code examples in this paper are designed to be executed regardless of the chosen option. However, users installing from PyPI will not have access to the training configuration files (see Section 3.4) or example notebooks provided in the GitHub repository.

Code examples

Downloading a model trained on the data described in Höfling *et al.* [23] and running a forward pass is as simple as follows:

```
import torch
from openretina.models import *
# Download model
model = load_core_readout_from_remote("hoefling_2024_base_low_res", "cpu")
# Run forward pass
responses = model.forward(torch.rand(model.stimulus_shape(time_steps=50)))
```

Listing 1. Example to download a pre-trained model and run a forward pass on a Gaussian random noise stimulus.

The first command downloads the model to a local directory on your computer[1] and initialises it on the CPU. The `model` is an object of type `CoreReadout` as described in Section 3.2. With the next command, we input a random video stimulus to the model and receive its predicted responses.

`openretina` also offers a command line interface to train models and create visualisations. Here is an example of how to explore its options and how to create visualisations of a pre-trained model.

[1] This directory defaults to `${HOME}/openretina_cache`, but can be changed by setting the environment variable `OPENRETINA_CACHE_DIRECTORY`.

```
openretina --help
# Download and visualize a pre-trained model
openretina visualize --model-path hoefling_2024_base_low_res --save-folder visualizations
# Download training data and train a new model (downloading data takes some time)
openretina train --config-path configs --config-name hoefling_2024_core_readout_low_res
```

Listing 2. Visualizing a pre-trained model and training a new model using `openretina`'s command line interface

The first command will show you a help message on how to use the command line tool. The `visualize` command will download the pre-trained model and save visualisations of its weights together with optimised stimuli that strongly activate each neuron of the model, both for the internal neurons and the output neurons of the model. The `train` command will download the training data and train a new model from scratch. It is recommended to do this on a machine with access to a GPU, which will then be used by default. You will find the logs, training results, and checkpoints in the specified output folder `exp`.

You can get an impression of the stimulus, the recorded retinal ganglion cell (RGC) responses, and model predictions by running our introductory notebook `openretina_demo.ipynb`. A representative output is shown in Figure 2.

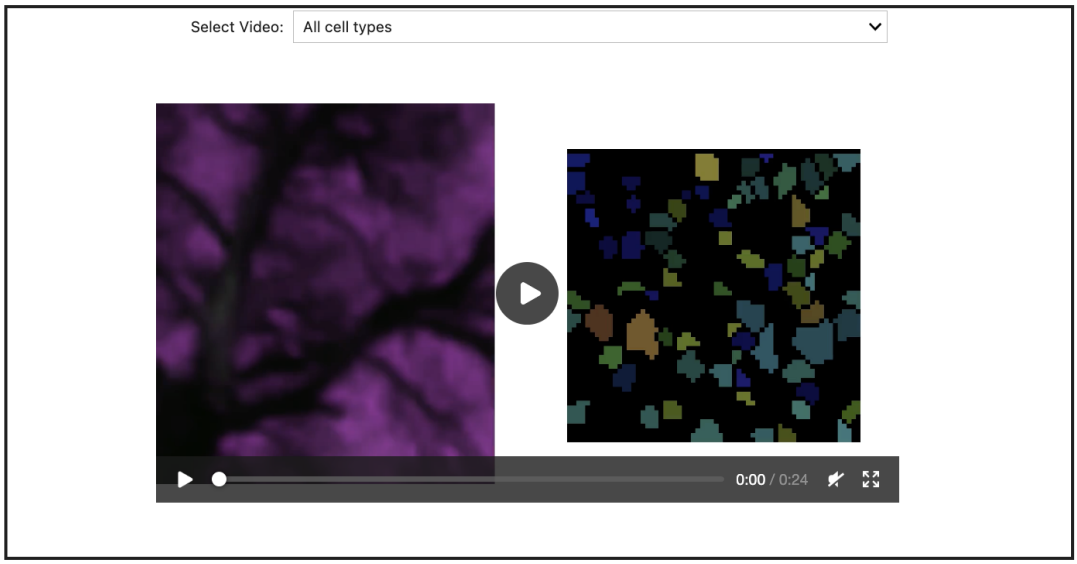

Fig. 2. Example screenshot of an interactive visualisation obtained with the introductory demo notebook. The interface displays the stimulus video alongside neural responses, shown from a Region of Interest (ROI) perspective. Each ROI represents a segmented retinal neuron, with activity recorded over time using two-photon (2P) calcium imaging. Neurons are colour-coded by cell type. The model was trained on data from Höfling *et al.* [23].

If you would like to start training a model from scratch on one of the existing datasets, or on your own dataset, proceed to Section 3. If you are interested in *in-silico* methods using our pre-trained models, go to Section 4.

3. Training a retina model

In this section, we provide a comprehensive guide to constructing and optimising system identification models of the retina using the `openretina` package. To streamline the process, the training infrastructure integrates PyTorch Lightning [27] which simplifies training and logging, and Hydra [28], which manages configuration options. While requiring a modest initial learning curve, these packages offer significant efficiency and flexibility for configuring and managing model runs.

In Section 3.1, we first introduce the structure of input data, covering supported visual stimuli (3.1.1) and neural response formats (3.1.2). We then outline the core components of our models in Section 3.2, explaining how they are structured and instantiated within the package. Sections 3.2.3 and 3.3 provide an overview of training and evaluating a model. Finally, in Section 3.4, we present a unified training script that integrates these steps, walking users through how to run and modify it for their own use cases.

We will guide the reader both through the package's more general functionalities and through a running example. Further demonstrations can be found as interactive notebooks on the project's GitHub repository. These include reproducing training of the base "Core + Readout" model (Fig. 3 a, right), which was trained on two-photon calcium imaging recordings of mouse RGC

responses to a dichromatic movie stimulus [23], as well as training a similar architecture on spiking data from salamander [10] and spiking data from mouse and marmoset [29].

Going through a single example will be sufficient to gain familiarity with the package's core functionality and workflow, yet exploring multiple datasets and model variations can provide a deeper understanding of its flexibility and potential applications.

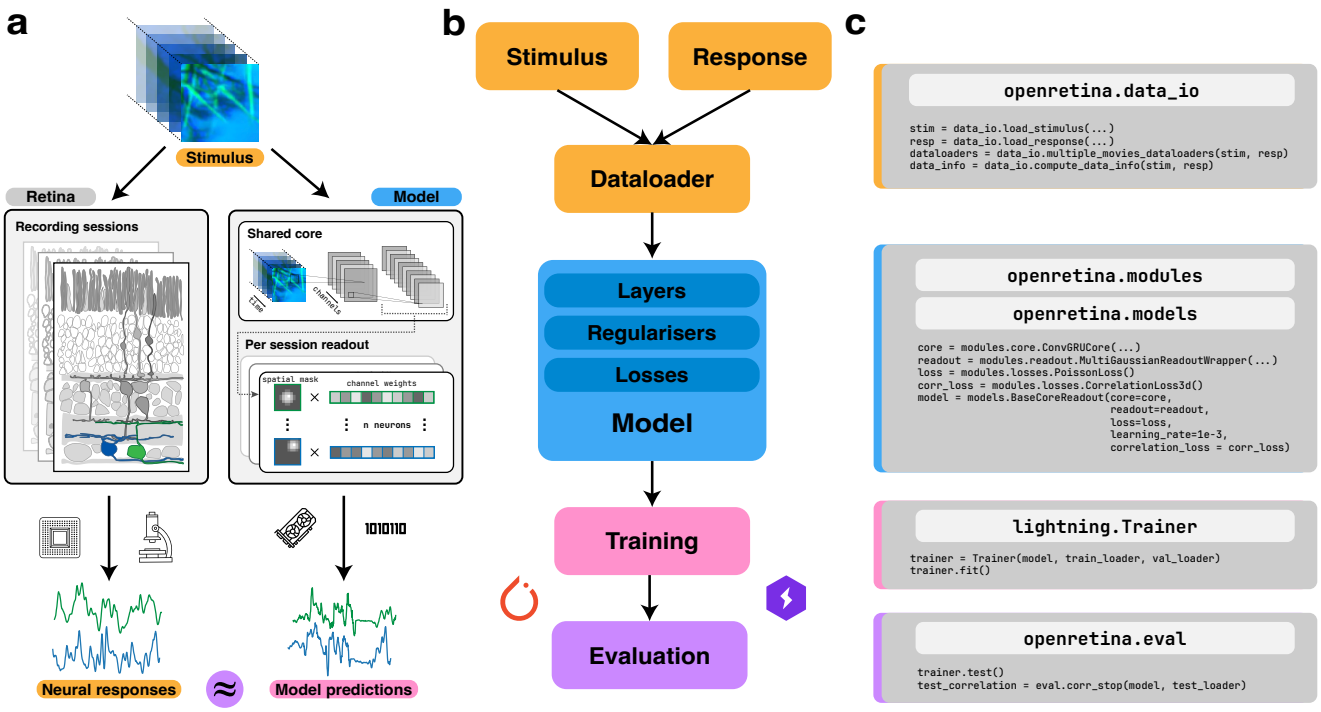

Fig. 3. Overview of modelling framework and training pipeline in `openretina`. (**a**) Typical retina experiments involve recordings of neural responses across multiple experimental sessions. Each session includes paired stimulus and response data, such as videos presented to the retina and the corresponding neural activity recorded via multi-electrode arrays (MEAs) or two-photon calcium imaging. Predictive models are then trained using the same stimulus-response pairs to reproduce retinal output. (**b**) Model training workflow. Input data to the model (in orange) for different sessions is processed, split into different folds for training and evaluation, and composed into dataloaders by functions in `openretina.data_io`. The core modelling stage (in blue) passes these inputs into a full model (`openretina.models`) composed of layers, regularisers, and loss functions (`openretina.modules`), with training managed by PyTorch and Lightning (in pink). After training, evaluation utilities (`openretina.eval`, in purple) provide metrics, perform comparisons to ground truth, and support testing on novel stimuli. (**c**) Mock training code examples for the different stages. For more detailed explanations and working examples, refer to the main text and package repository.

3.1. Data

`openretina` supports any dataset where both the visual stimuli (the inputs) and the retinal responses (the outputs) can be expressed as matching time series coming from experimental data. For training and running models, the data is then fed to a dataloader and expressed as a series of tuples $\{\text{input}_t, \text{output}_t\}$ (Fig. 3 b, top).

As a terminology note, throughout the paper, we use "session" or "experimental session" to refer to a single experimental recording on a specific portion of retinal tissue under controlled conditions. Each session contains both retinal activity and the corresponding visual stimulus presentation, ensuring a shared temporal reference for inputs and outputs.

Throughout running examples and explanations in this paper, we always treat the more general case of datasets containing multiple sessions.

In `openretina`, you have two options to integrate a new dataset:

1. Implement custom input/output functions for your native data structure and add the corresponding configuration to `openretina`.

2. Convert your data set into the `hdf5` [30] -based data structure that `openretina` natively supports.

We will first describe the basic input/output functionality in `openretina` and look at worked examples of custom data loading functions for a dataset. This is also explained in an interactive notebook on our GitHub repository at `notebooks/new_datasets_guide.ipynb`. We will then conclude our discussion on data by briefly describing the `hdf5`-based data structure we support in Section 3.1.4.

3.1.1. Loading stimuli

```
from openretina.data_io.base import MoviesTrainTestSplit
import numpy as np

example_stimulus = MoviesTrainTestSplit(
    train=np.random.rand(1, 200, 50, 50), # channels, train_time, height, width
    test=np.random.rand(1, 100, 50, 50),  # channels, test_time, height, width
    stim_id=None, # Optional identifier for the stimulus
    norm_mean=None, # Optional, will store the mean of the stimulus if it was normalised
    norm_std=None,  # Optional, will store the std of the stimulus if it was normalised
)
```

Listing 3. Example usage of the `MoviesTrainTestSplit` class for importing a random stimulus.

Input stimuli to the model are expected to be in the form of videos, represented as 4-dimensional tensors of shape `(color_channels x time x height x width)`. In typical experiments, stimuli are divided into training and testing sequences, with testing sequences often repeated more frequently to allow computing some estimates of response reliability. To accommodate this routine structure, our base class for input movies `openretina.data_io.base.MovieTrainTestSplit` is designed to load the training and testing stimuli for a certain experimental session. In the case of multiple sessions, regardless of experiment setup, the first step in loading the stimuli for training is creating a Python dictionary with the experimental sessions' names as *keys* and `MovieTrainTestSplit` objects as *values*.

The exact procedure to do so will depend on the format in which experimental data has been exported and on the arrangement of the stimuli used in the experiments. Here we provide a practical example.

Stimulus loading example: Maheswaranathan *et al.* [10]

```
from openretina.data_io.maheswaranathan_2023.stimuli import load_all_stimuli

# Data can be found at https://doi.org/10.25740/rk663dm5577 or on openretina HuggingFace
movies_dictionary = load_all_stimuli(
    base_data_path=..., # Path to the "ganglion_cell_data" folder in the archive
    stim_type="naturalscene", # "whitenoise" stimulus also present
    normalize_stimuli=True, # Always recommended for more stable training
)
```

Listing 4. Importing stimulus data from Maheswaranathan *et al.* [10].

In Listing 4, we provide a minimal example with a pre-defined function in `openretina` to load all stimuli present in this dataset. Note that for a new dataset a similar function should be coded by the user. Depending on how the dataset was exported this should be a relatively simple function to code up (e.g. `load_all_stimuli` is only around 40 lines of code in this case). The output dictionary created by this function will look as follows, with session names as *keys* and movie objects as *values*:

```
{   '15-10-07': MoviesTrainTestSplit(train=numpy.ndarray(shape=(1, 359802, 50, 50)),
                                     test=numpy.ndarray(shape=(1, 5996, 50, 50)),
                                     stim_id='naturalscene',
                                     random_sequences=None,
                                     norm_mean=51.49113464355469,
                                     norm_std=53.6270637512207),
    '15-11-21a': MoviesTrainTestSplit(train=numpy.ndarray(shape=(1, 359802, 50, 50)),
                                      test=numpy.ndarray(shape=(1, 5997, 50, 50)),
                                      stim_id='naturalscene',
                                      random_sequences=None,
                                      norm_mean=51.49113464355469,
                                      norm_std=53.6270637512207),
    '15-11-21b': MoviesTrainTestSplit(train=numpy.ndarray(shape=(1, 359796, 50, 50)),
                                      test=numpy.ndarray(shape=(1, 5996, 50, 50)),
                                      stim_id='naturalscene',
                                      random_sequences=None,
                                      norm_mean=51.49095916748047,
                                      norm_std=53.62714385986328)}
```

3.1.2. Loading neural responses

```
from openretina.data_io.base import ResponsesTrainTestSplit
import numpy as np

example_responses = ResponsesTrainTestSplit(
    train=np.random.rand(60, 200), # neurons, train_time
    test=np.random.rand(60, 100),  # neurons, test_time
    test_by_trial=np.random.rand(10, 60, 100), # trials, neurons, test_time
    stim_id=None, # Optional identifier for the stimulus
    session_kwargs={"cell_types": np.random.randint(1, 42, 60)}, # Optional extra information to store
    # regarding the responses. Can be any dictionary. Will be saved in the model.
)
```

Listing 5. Example usage of the `ResponsesTrainTestSplit` class for importing mock neural data.

Neural responses for each session are expected to be expressible in a 2-dimensional tensor of shape `n_neurons x time`, where `time` should match the temporal bins for the stimulus in that same session. Paralleling stimulus organisation, responses also have their base class `openretina.data_io.base.ResponsesTrainTestSplit`, holding training and testing responses to the respective stimuli. Also mirroring stimuli organisation, responses from multiple experimental sessions should be organised in a dictionary, with the experimental sessions' names as *keys* and `ResponsesTrainTestSplit` objects as *values*.

The exact procedure to extract the responses in this format will depend on experimental design and pre-processing pipelines.

Responses loading example: Maheswaranathan *et al.* [10]

```
from openretina.data_io.maheswaranathan_2023.responses import load_all_responses

# Data can be found at https://doi.org/10.25740/rk663dm5577 or on openretina HuggingFace
responses_dictionary = load_all_stimuli(
    base_data_path=..., # Path to the "ganglion_cell_data" folder in the archive, same as movies
    response_type="firing_rate_20ms", # 5ms and 10ms binning also present
    stim_type="naturalscene", # "whitenoise" stimulus also present
)
```

Listing 6. Importing neural responses from Maheswaranathan *et al.* [10].

Like for the stimuli, in Listing 6 we provide a minimal example with a pre-defined function in `openretina` to load all responses present in this dataset.

The output dictionary will have this structure:

```
{   '15-10-07': ResponsesTrainTestSplit(train=numpy.ndarray(shape=(9, 359802)),
                                        test=numpy.ndarray(shape=(9, 5997)),
                                        test_by_trial=None,
                                        stim_id='naturalscene',
                                        session_kwargs={}),
    '15-11-21a': ResponsesTrainTestSplit(train=numpy.ndarray(shape=(14, 359802)),
                                         test=numpy.ndarray(shape=(14, 5997)),
                                         test_by_trial=None,
                                         stim_id='naturalscene',
                                         session_kwargs={}),
    '15-11-21b': ResponsesTrainTestSplit(train=numpy.ndarray(shape=(27, 359796)),
                                         test=numpy.ndarray(shape=(27, 5996)),
                                         test_by_trial=None,
                                         stim_id='naturalscene',
                                         session_kwargs={})}
```

3.1.3. Composing complete dataloaders

With both stimuli and responses organised in matching dictionaries, the last step in data preprocessing before training is to compose dataloaders, which will manage the efficient batching and shuffling of data during training.
In `PyTorch`, a `DataLoader` is a utility that wraps datasets and handles the sampling, batching, and shuffling of data, enabling consistent data iteration during training and evaluation. In our case, dataloaders are particularly useful for splitting the stimulus sequences into smaller, manageable chunks while ensuring that these chunks remain correctly matched to the corresponding neural responses.
We provide an organised interface for creating these dataloaders, ensuring compatibility with diverse datasets and preprocessing requirements.

The core base utility for this task is the `multiple_movies_dataloader` function, found under `openretina.data_io.base_dataloader`. It takes as input the matching dictionaries of stimuli and responses, and outputs a dictionary containing dataloaders for training, validation, and testing organised by session.

Dataloading example: Maheswaranathan *et al.* [10]

```
from openretina.data_io.base_dataloader import multiple_movies_dataloaders

dataloaders = multiple_movies_dataloaders(
    neuron_data_dictionary=responses_dictionary,
    movies_dictionary=movies_dictionary,
    train_chunk_size=50,
    batch_size=64,
    seed=42,
    clip_length=90,
    num_val_clips=50,
    allow_over_boundaries=True,
)
```

Listing 7. Creating dataloaders from dictionaries of stimuli and responses.

The dictionary of dataloaders created in this way will have the following structure:

```
{   'test': {   '15-10-07': torch.utils.data.DataLoader,
                '15-11-21a': torch.utils.data.DataLoader,
                '15-11-21b': torch.utils.data.DataLoader},
    'train': {   '15-10-07': torch.utils.data.DataLoader,
                 '15-11-21a': torch.utils.data.DataLoader,
                 '15-11-21b': torch.utils.data.DataLoader},
    'validation': {   '15-10-07': torch.utils.data.DataLoader,
                      '15-11-21a': torch.utils.data.DataLoader,
                      '15-11-21b': torch.utils.data.DataLoader}}
```

Special case: Instantiating a single dataloader for one session, one stimulus.

```
import numpy as np
from openretina.data_io.base_dataloader import get_movie_dataloader

random_stimulus = np.random.rand(1, 300, 50, 50) # channels, time, h, w
random_responses = np.random.rand(300, 42) # time, neurons

single_dataloader = get_movie_dataloader(
    movie=random_stimulus,
    responses=random_responses,
    split="train",
    scene_length=300, # Assuming this is a single coherent clip
    chunk_size=50, # Tells the dataloader how long in time we want the samples to be
    batch_size=1, # How many samples are going to be in a "batch"
)
```

While the dictionary organisation we provide for multiple dataloaders is useful during training and testing, in certain scenarios (e.g. prototyping a new architecture, evaluating the model on held-out data), it might be useful to instantiate a single dataloader using a single pair of stimuli and responses for a single session. The function to do so is `get_movie_dataloader`, still under `openretina.data_io.base_dataloader`, which takes as inputs a 4-dimensional tensor of shape `channels x time x height x width`, and a `time x neurons` responses tensor. It then outputs a single `torch.utils.data.DataLoader` object, which samples and returns matching stimulus-response pairs. Details for the sampling must be provided through the function's hyperparameters.

3.1.4. HDF5-based data structure

In certain scenarios, for example when the user's data format is still under development and changing, it might be preferable and more straightforward to adjust the data pre-processing and exporting pipeline than writing custom data loading code. For such circumstances, `openretina` provides a template `hdf5` structure that users can follow to export their data in a format that is well-suited for modelling.

In this data structure, each recorded session is stored as a separate `hdf5` file, and each stimulus is stored as a NumPy [31] file within the `stimuli` subfolder. To help users match their data to this format, `openretina` has utilities to create example synthetic data in this structure:

```
# Print help message
openretina create-data --help
# Create data in the folder ‘./test_data‘
openretina create-data ./test_data --num-colors 3 --num-stimuli 4 --num-sessions 2
```

Listing 8. Creating artificial data for the pre-implemented `hdf5` dataloader.

This command generates two `hdf5` files, one for each session, in the specified folder `test_data` with responses corresponding to each of the four stimuli. Additionally, the `test_data/stimuli` subfolder contains the corresponding stimuli as NumPy files. The `hdf5` files also include an attribute called `session_info`, which stores metadata such as the cell type of each neuron in the session.

To correctly match the responses to the stimuli, both must be sampled at the same frequency and have an equal number of temporal frames (see Sec. 3.1). While the duration of each stimulus can vary, all stimuli should have the same number of colour channels, as well as identical height and width.

This data can now be read using the functions in `openretina.data_io.h5_dataset_reader`. It will also be readily compatible with existing training configurations (see Listing 14).

With all the options for data import and loading covered, we can now move to the models' side.

3.2. Models

Models in `openretina` adopt the widely used "Core + Readout" framework [see 7, 13, 14, 32–37, among others]. Simply put, this approach separates the model architecture into two distinct components. The Core, which is shared across all sessions and cells, processes the input stimulus extracting a rich feature representation of the visual input. Following this the Readout, which is experimental-session-specific, maps the Core's output feature space to individual neurons' firing rate predictions for a given stimulus.

All models present at the time of writing can be classified as "data-driven": All components of the models are trained from scratch on the stimulus-response pairs. [1]

3.2.1. Using a model

Now continuing our running example on how to go from data to a trained model, let us see how to instantiate such a model: we focus on the base Core + Readout model provided in `openretina.models.core_readout`, creatively named `CoreReadout`.

```
from openretina.models import CoreReadout
from openretina.data_io.base import compute_data_info

data_info = compute_data_info(responses_dictionary, movies_dictionary) # Same args as dataloaders
n_neurons_dict = data_info["n_neurons_dict"]

model = CoreReadout(
    in_shape=(1, 100, 50, 50),     # (channels, time, height, width). Time can be arbitrary
    hidden_channels=(16, 32),      # Number of filters for each convolutional layer
    temporal_kernel_sizes=(11, 5), # Size of spatial convolutional kernels
    spatial_kernel_sizes=(15, 11), # Size of temporal convolutional kernels
    n_neurons_dict=n_neurons_dict, # Used to initialise the readout networks
    data_info=data_info,           # Not required, but recommended to facilitate downstream tasks
)
```

Listing 9. Instantiating a simple "Core + Readout" model.

Some arguments are required when initialising a model:

- `in_shape`: A tuple specifying the expected shape of the input tensor. This should include dimensions for channels, time [2], height, and width. This parameter informs the model about the shape of input data, and is necessary to properly initialise the layers of the Core network.
- `hidden_channels`, `temporal_kernel_sizes`, `spatial_kernel_sizes`: These arguments all expect tuples of integers, and need to have all the same length. The length of the tuples will determine the number of convolutional layers the model will have (two in our example), and the arguments will determine the sizes of the temporal and spatial kernels, respectively.

[1] For completeness, the alternative are "task-driven" models, where the Core usually comes from a pre-trained Computer Vision model, which is then either kept fixed or fine-tuned. The Readout is still completely learned.

[2] Note that the time dimension can be set to an arbitrary value here, as we assume all models can be used on movie chunks of variable duration. All other parameters however will constrain directly model architectures.

- `n_neurons_dict`: A dictionary where *keys* represent session names and *values* indicate the number of neurons in each session. This parameter is used to initialise the Readout networks, ensuring they are appropriately sized for the datasets being used.

A convenient way to provide the model with `n_neurons_dict` is to extract all the information regarding the data fed to the model with `openretina.data_io.base.compute_data_info` (see Listing 9). This function takes as arguments the same matching dictionaries of stimuli and responses which are fed to the dataloaders, and outputs a dictionary with different arguments that describe the data. `data_info` can also be passed as an argument to the model during initialisation. This way, it will be saved as one of the model's attributes, and be accessible also, for example, once the model is trained and is loaded from a checkpoint.

In addition to the required parameters, users can specify additional hyperparameters for the model, mostly governing the strength of regularisation at different stages. In many cases, the default values provided by the `CoreReadout` class suffice for obtaining an initial good fit, though these parameters should be adjusted as needed to tailor the model to specific datasets or research questions. We refer the advanced user to the package documentation for further information on each model's hyperparameters.

3.2.2. Extending and customising models

Each Core + Readout model inherits from `openretina.models.core_readout.BaseCoreReadout`, a subclass of PyTorch Lightning's `LightningModule`.

Complete models within `openretina` are always expected to be instances of PyTorch Lightning's `LightningModule`. This design choice is motivated by the way PyTorch Lightning abstracts away much of the boilerplate code typically required for training, validation, and testing neural networks. For example, Lightning handles tasks such as checkpointing, gradient accumulation, distributed training, and early stopping, allowing to focus on model architecture and experimentation rather than implementation details. Practically, unlike standard PyTorch implementations, where components such as the loss function, optimiser, and training loop are defined externally, PyTorch Lightning integrates these elements directly within the model class itself. Training and evaluation procedures are then seamlessly handled by Lightning's Trainer class, simplifying the overall workflow.

In the context of `openretina`, this structure enables us to focus on defining key model components while maintaining a clean and extensible codebase.

Concretely, when building a new class inheriting from `BaseCoreReadout`, the following parameters will be required during initialisation:

- `core`: The Core module, responsible for processing the input stimulus and extracting a shared feature representation.
- `readout`: The Readout module, which maps the Core's features to neuron-specific firing rate predictions.
- `learning_rate`: A float specifying the learning rate for the optimiser. The default optimiser is Adam [38].
- `loss`: The training loss function (default is Poisson loss). Users can also provide custom loss functions.
- `correlation_loss`: The evaluation loss function (default is the time-averaged Pearson correlation coefficient). This provides a standardised metric for assessing prediction quality across neurons.

The package already provides dedicated submodules for different components, enabling users to customise and extend models with ease.

Model components are organised under `openretina.modules`, and include:

- `openretina.modules.core` for Core architectures.
- `openretina.modules.readout` for different Readout mechanisms.
- `openretina.modules.layers` which includes more specialised components used within cores or readouts, such as convolutional layer types and regularizers.
- `openretina.modules.losses` which contains the objectives used in training.

This modularity not only simplifies the process of building and testing models but also facilitates experimentation with new ideas in system identification.

While this approach provides a robust foundation, certain aspects, such as changing the optimiser or training schedule, are currently not configurable through model hyperparameters but should be changed by accessing class internals. To address this limitation, we plan to introduce more flexible support for these features in future releases of the package.

3.2.3. Training procedure

With the models initialised, training is now as easy as instantiating and calling PyTorch Lightning's `Trainer` class. The only small step we need to do before that is turning our dictionaries of dataloaders across sessions into a single iterator using classes in `openretina.data_io.cyclers`.

```
from openretina.data_io.cyclers import LongCycler, ShortCycler
from lightning import Trainer

train_loader = LongCycler(dataloaders["train"], shuffle=True)
val_loader = ShortCycler(dataloaders["validation"])

trainer = Trainer(max_epochs=50, ...) # Extra trainer hyperparameters would go here
trainer.fit(model, train_loader, val_loader)
```

Listing 10. Starting the training procedure.

Depending on the configuration passed, Lightning will handle a range of tasks automatically, including training loop management, validation at specified intervals, checkpointing, and logging metrics. We refer readers to Pytorch Lightning's documentation for additional information [3].

It is worth noting again that within `Lightning` (and `openretina`), loss functions and optimisers are part of the model class and definition (see again Section 3.2.2), which is what allows for simple training code.

3.3. Evaluation

Once training is complete, we can evaluate on the test set.

```
test_loader = ShortCycler(dataloaders["test"])

trainer.test(model, test_loader)
```

Listing 11. Evaluating the model on the test set.

This will put the model in evaluation mode, and report the loss metrics on the test set. The available loss functions are defined in the module `openretina.modules.losses`, and include the mean squared error loss, the correlation loss and the Poisson loss. The latter is used by default for training, while the correlation loss is used to report performance.

Additionally, we provide more evaluation metrics and oracles that are less suitable as loss functions in the module `openretina.eval`. These include , among others:

- The *Leave-One-Out* oracle, also known as the jackknife oracle, estimates the upper bound of model performance using repeated stimulus presentations. For each trial, it predicts the response by averaging the responses from all other repeats of the same stimulus. The predicted response is then compared to the left-out response using correlation. This method accounts for trial-to-trial variability but may underestimate the true oracle correlation when only few repeats are available. [see 39, for more details].

- The *Fraction of explainable variance explained (FEVE)*, i.e., the proportion of stimulus-driven, and hence explainable, variance captured by the model.This is implemented following Cadena *et al.* [14].

3.4. Wrapping up: our unified training script

`openretina` provides a unified training script that works seamlessly across all models, leveraging Hydra [28] for management of `.yaml` configuration files and PyTorch Lightning [27] for training. This script simplifies the process of setting up and running experiments, offering both flexibility and ease of use. It can be found at `openretina.cli.train.py`, and when the package is installed it can simply be called from the terminal with the command `openretina train`.

[3] https://lightning.ai/docs/pytorch/stable/

Config structure

The unified training script relies on a structured configuration system, as shown in Listing 12. The configuration files are organised into several subdirectories, each serving a specific purpose:

- `data_io` and `dataloader` contain dataset-specific configurations, such as file paths, preprocessing steps, and dataloader parameters. Users must add their dataset-specific configs here to use the unified script. We already have examples for the datasets used in Maheswaranathan *et al.* [10], Höfling *et al.* [23], and Karamanlis *et al.* [29].
- `trainer` includes default parameters for the PyTorch Lightning Trainer, such as maximum epochs, checkpointing, and distributed training options.
- `logger` provides configurations for different logging options (e.g., CSV, TensorBoard [40], WandB [41]) to track training progress and results.
- `model` contains default configurations for models, including hyperparameters, Core + Readout setups, and other architecture-related options.
- `training_callbacks` defines optional callbacks, such as early stopping, learning rate scheduling, and custom evaluation hooks.
- `quality_checks` contains configurations for further optional quality checks to apply after data import.

The key idea is that all these subfolders contain sensible defaults and optional configurations in different `.yaml` files, which can be overridden by an outer configuration file (i.e. to be placed in the root `config/` folder) that orchestrates the experiment.

```
configs/                                        # Main config folder
|-- data_io/                                    # Data input/output specifics
|   |-- hoefling_2024.yaml
|   |-- maheswaranathan_2023.yaml
|   `-- ...
|-- dataloader/                                 # Dataloader-specific hyperparams
|   |-- hoefling_2024.yaml
|   |-- maheswaranathan_2023.yaml
|   `-- ...
|-- hydra/                                      # Hydra-specific options
|-- logger/                                     # Run logging options
|-- model/                                      # Model configurations
|   |-- base_core_readout.yaml
|   |-- linear_nonlinear_poisson.yaml
|   `-- ...
|-- quality_checks/                             # Extra quality checks for data IO
|-- trainer/                                    # Trainer configurations
|-- training_callbacks/                         # Extra training functionality
|-- hoefling_2024_core_readout_high_res.yaml    # Example "outer config"
|-- maheswaranathan_2023_core_readout.yaml      # Example "outer config"
|-- karamanlis_2024_core_readout.yaml           # Example "outer config"
`-- ...
```

Listing 12. Config folder structure.

How to use the unified training script

To use the unified training script on a new dataset, first add dataset-specific configurations: Create and place the necessary configurations in the `data_io` and `dataloader` folders. This includes paths to stimulus movies and neural responses, preprocessing steps, and batch size settings. Secondly, create an "outer config" to be placed in the root `config/` folder: This is a YAML that specifies your experiment's overarching settings. This file will reference the nested configs in other folders and overwrite defaults where needed (e.g., model parameters, trainer settings).

Once these are in place, you can run the training script with Python, passing the outer config name (without the `.yaml` extension. Hydra will resolve the configuration hierarchy and apply the specified settings. You can further override specific parameters during runtime, i.e. when calling the script from the command line. For example, you can change the experiment name used during logging:

```
# Assuming we start from the openretina root project folder
openretina train --config-name your_outer_config_name \
    exp_name=my_first_openretina_run
```

Listing 13. Example command to run the unified training script.

Providing an additional example linking back to Section 3.1.4, you can train a Core + Readout model with the help of the `hdf5_core_readout` configuration as follows:

```
openretina train --config-path configs --config-name hdf5_core_readout \
    paths.data_dir="test_data" data_io.test_names="[random_noise_2, random_noise_3]" \
    data_io.color_channels=3 data_io.video_height=16 data_io.video_width=8
```

Listing 14. Training a Core + Readout model with the `hdf5` dataloader. This will train a small Core + Readout model using the artificially generated data as detailed in Subsection 3.1.4. When working with your own dataset, be sure to adjust the number of colour channels and the video dimensions accordingly and specify the names of the stimuli to be used for testing.

While it is not strictly necessary to use the unified script to train a model within `openretina`, doing so offers numerous advantages. By organising all configurations into a centralised structure, users gain greater control over all moving parts of a model run, from data loading to training and evaluation. This approach also encourages consistency and reproducibility, which are critical for collaborative research efforts.

We refer to Hydra's documentation [4] for a detailed guide on the usage of structured configs.

4. In-silico experiments

We now have access to a trained neural network that can replicate the input-output relationship of the retina. In this section, we describe different use cases that help us understand what the model learned, and we give examples of how this knowledge can lead to a better understanding of the computational processes of the retina. The relevant code is in the Python sub-module `openretina.insilico`.

4.1. Optimising stimuli

The output of the neural network contains the model's prediction of the biological neurons' responses to the stimulus (see Figure 3a). A natural question to ask is which stimulus activates each neuron the most by finding the maximally exciting input (MEI) [19, 20]. As shown in Listing 15, `openretina` supports this by defining an objective and then using the function `optimize_stimulus` to optimise a stimulus towards the given objective.

```
from functools import partial
import torch; import matplotlib.pyplot as plt
from openretina.models import *
from openretina.insilico import *
from openretina.utils import *

model = load_core_readout_from_remote("hoefling_2024_base_low_res", "cpu")
# Define objective that increases the activity of a single neuron between timestep 10 and 20.
objective = IncreaseObjective(
    model,
    neuron_indices=[10],
    data_key="session_2_ventral2_20201016",
    response_reducer=SliceMeanReducer(axis=0, start=10, length=10),
)
# Initialise random stimulus with a variance of 0.1
stimulus = torch.randn(model.stimulus_shape(time_steps=40), requires_grad=True)
stimulus.data = stimulus.data * 0.1
#  Optimise stimulus with gradient descent towards the objective.
optimize_stimulus(
    stimulus=stimulus,
    optimizer_init_fn=partial(torch.optim.SGD, lr=100.0),
    objective_object=objective,
    optimization_stopper=OptimizationStopper(max_iterations=10),
)
# Visualise stimulus
save_stimulus_to_mp4_video(stimulus[0], "mei.mp4")
fig, axes = plt.subplots(2, 2, figsize=(7 * 3, 12))
plot_stimulus_composition(stimulus[0], axes[0,0], axes[0,1], axes[1,0])
```

Listing 15. Python example to optimise a stimulus towards an objective and visualise the optimised stimulus.

In this case, the pre-implemented objective maximises the response of a single neuron: we initialise a random stimulus and optimise it toward our objective using gradient descent with a learning rate of 100.0 for 10 optimisation steps.

[4] https://hydra.cc/docs/intro/

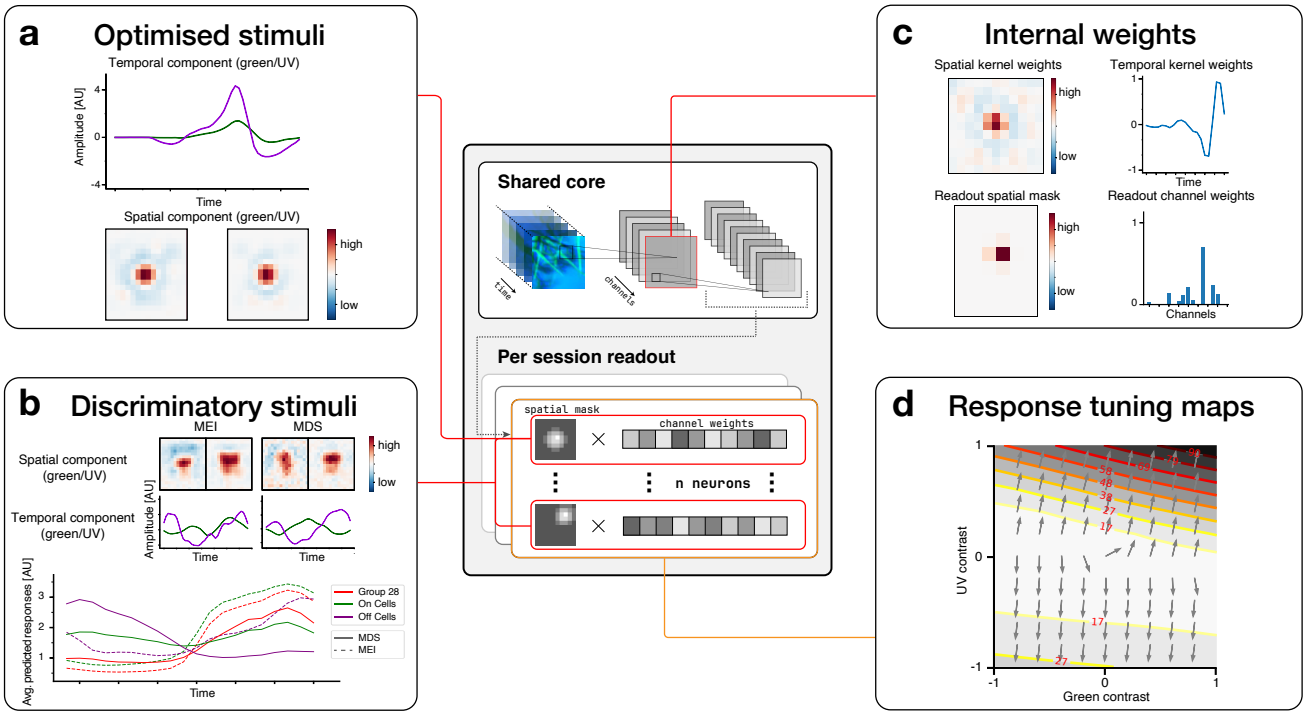

Fig. 4. `openretina` **provides tools for visualising model weights and for performing *in-silico* experiments.** (**a**) **Optimised stimuli:** The digital twin model allows us to optimise stimuli towards any objective, e.g. towards activating a particular neuron strongly. The panel shows the spatial and temporal components of an optimised MEI stimulus for a neuron. (**b**) **Discriminatory stimuli:** Discriminatory objective functions are used to optimise a stimulus towards activating certain neurons strongly while inhibiting other groups of neurons. The upper part of the panel shows both an MEI, optimised to activate all cells of group 28, and a most discriminatory stimulus (MDS), which was optimised to drive the activity of RGC cells of group 28, while keeping the activity of ON and OFF cells low. As seen by the average responses in the optimisation window (lower sub-plot), while for the MEI stimulus the average response of ON cells is higher than the response of group 28 cells, the MDS stimulus leads to a higher response of group 28 cells compared to other ON cells. (**c**) **Internal weights**: By zooming in and visualising individual model weights, we can better understand how the neural network reproduces the behaviour of the retina. The upper part of the panel shows a visualisation of the spatial and temporal weights of a three-dimensional convolutional layer. The lower part of the panel shows the readout mask, which determines which spatial location in the stimulus influences a modelled neuron, and the readout channel weights that determine the influence of each convolutional channel on the modelled neuron. (**d**) **Response tuning maps:** We can evaluate the model neurons along directions in stimulus space that we are interested in by changing stimulus parameters corresponding to those directions and examining model responses. In this example, we examined the directions along which green and UV temporal contrast selectivities, respectively, change from negative to positive.

The result is the stimulus that the model predicts to be maximally exciting for the given neuron. This optimised stimulus can then be saved as a video as shown in the example code, or decomposed into its temporal and spatial components (Fig. 4 a; for illustration see accompanying Jupyter notebook `notebooks/mei_example.ipynb`). For the decomposition, we followed Höfling *et al.* [23] and ran singular value decomposition on each colour channel individually. The visualisation then shows the first singular vector for the decomposed spatial and temporal dimensions, respectively.

MEIs can be used to gain insights about neurons' tuning properties. By analysing the MEIs of a trained model, Höfling *et al.* [23] was able to discover an RGC type selective to chromatic contrasts that allows the detection of changes in visual context. By additionally confirming these results experimentally, this provides a study how optimised stimuli can lead to a better understanding of the features a neuron is selective to.

4.2. Discriminatory stimuli

To give an example of a more complex optimisation objective in our repository, we provide the Jupyter notebook `notebooks/most_discriminative_stimulus.ipynb`. Here, we show how to generate stimuli that elicit maximally distinct responses in different cell types as visualized in Fig. 4 b. Burg, Zenkel et al. (2024) [22] used this approach to cluster neurons based on their functional responses. Since you can define arbitrarily complex objectives as long as they are expressible as differentiable functions, this provides a powerful interface for probing the neural network.

4.3. Zoom In: Visualisation and interpretation of model weights

While the use of optimised stimuli can give an indication of the features that a modelled neuron responds to, it does not indicate *how* the model computed a feature. Speaking in terms of retinal information processing, an MEI gives insights about the message that a retinal neuron sends to the brain [42–44], but it does not tell us about how the extraction of this message is implemented in retinal circuitry [10, 45, 46]. Towards such a mechanistic understanding, we can extend the approach of optimising stimuli to internal neurons of the model. To the extent that the architecture of the model corresponds to the architecture of the retinal circuit (e.g. in terms of the number of layers, or the number of cell types or convolutional channels, respectively), a transfer of insights gained about internal model neurons to biological retinal interneurons is warranted (see e.g. Maheswaranathan *et al.* [10] and Schröder *et al.* [47]). For the default Core + Readout architecture, we support weight visualisation for both the convolutional layer and the readout layer, as visualised in Fig. 4 c and illustrated in Listing 16:

```
from openretina.models import *
model = load_core_readout_from_remote("hoefling_2024_base_low_res", "cpu")

# Visualise a specific channel weight for the first convolutional layer
conv_layer_figure = model.core.plot_weight_visualization(layer=0, in_channel=1, out_channel=0)

# Visualise the readout weights of a neuron for a particular readout session
session_key = model.readout.readout_keys()[0]
readout_figure = model.readout[session_key].plot_weight_for_neuron(5)
```

Listing 16. Python example to plot convolutional and readout weights of the Core + Readout model

For convolutional layers, we visualise the weights of each channel of a convolutional layer separately. The spatiotemporal separable convolution layer defined in `STSeparableBatchConv3d` consists of a two-dimensional spatial weight and one-dimensional temporal weight. This temporal component is then multiplied by the spatial weight to receive a three-dimensional tensor for the convolution operation. The `conv_layer_figure` of Listing 16 plots both the spatial weight, the computed temporal component, and the sine and cosine temporal weights.

The readout layer defined in `SimpleSpatialXFeature3d` can be visualised as a two-dimensional Gaussian mask that defines from which position the activities are read out from the core. Similarly, the feature weights that indicate the importance of each channel of the core for the output of the current neuron, are visualised as an one-dimensional bar chart. The `readout_figure` of Listing 16 contains both of these visualisations. By combining the visualisations of the weights and the MEIs of the internal neurons, we get a sense how the neural network calculates its features.

Using `openretina`'s command line interface, you can create weight visualisations in combination with the MEIs for all internal and output neurons as follows:

```
# Explore the options of the visualise tool
openretina visualize --help
# Download and visualise a pretrained model
openretina visualize --model-path hoefling_2024_base_low_res --save-folder visualizations
# We also support the original ensemble model of the paper Hoefling, Elife, 2024.
openretina visualize --is-hoefling-ensemble-model --model-id 0 --save-folder vis_ensemble_0
```

Listing 17. Visualising a pre-trained model and training a new model using openretina's command line interface

Here, we first print the help string of the `visualize` command. We then plot weight and MEI visualisations for all neurons of the pre-trained model "hoefling_2024_base_low_res", which are written into the folder "visualizations". We also support to download and visualise the original ensemble model used in the paper by Höfling *et al.* [23]. The last command will download this model and visualise all weights and MEIs of the first model of the ensemble models.

4.4. Analysing tuning around the optimal stimulus

The optimal stimulus is the stimulus that sits at the maximum of the model neuron's tuning curve. In order to get a more comprehensive understanding of a cell's tuning, for example in terms of how selectively it responds to a stimulus, or how sensitive it is to changes in the stimulus along different dimensions of stimulus space, we can expand our estimate of the tuning curve around the maximum. How this can be done within `openretina` is demonstrated using the example of chromatic contrast tuning in transient Suppressed-by-Contrast RGCs in the mouse [see Fig. 4d, 23], as shown in the notebook available at `notebooks/hoefling_gradient_analysis.ipynb`.

5. Contributing to the package

Scientific progress is often driven by open collaboration, and `openretina` is built with this in mind. By providing an easy-to-use and well-documented interface, we aim to create a growing repository of tools for studying retinal computation, favouring a joint scientific effort. Presently, there are two primary ways to contribute: by adding support for new datasets or by expanding the set of available models.

To add a dataset, contributors should create a new submodule under `openretina.data_io`, named after the dataset or the publication it originates from. This directory should contain code to load the dataset and generate PyTorch-compatible dataloaders, following the conventions outlined in the data section.

For model contributions, new architectures should be placed under `openretina.models`, while individual components or modules can be added to `openretina.modules`. All models should be compatible with PyTorch Lightning and, where possible, inherit from existing base classes. We particularly welcome architectures that incorporate biologically inspired mechanisms or novel computational approaches relevant to retinal processing. In parallel, we are actively working on expanding the set of available models ourselves.

Contributions should be submitted via pull requests on GitHub. Detailed guidelines on formatting, compatibility checks, and required tests can be found on the `openretina` GitHub project page. We encourage researchers to join us in refining and extending this platform to accelerate progress in understanding retinal computation.

6. Discussion

With the release of `openretina`, we introduce an accessible Python package designed to train and use ANNs that replicate the input-output behaviour of the retina. By integrating open-source datasets of retinal recordings from mice [23], salamanders [10], and marmosets [29], `openretina` additionally provides the foundation for a platform for sharing and comparing data and models across research groups, recording modalities and species.

We envision OpenRetina—beyond the `openretina` package—as a collaborative effort between research groups, and we strongly encourage contributions from researchers and practitioners in the field. Our goal is to create an open, flexible platform that evolves through shared development, and we are open to discussions about design decisions with those interested in shaping the project. Our priority is to build a tool that benefits the field of retinal system identification as a whole.

6.1. Towards collaborative and comparative modelling in visual neuroscience

Platforms for sharing data and comparing models across research groups have gained traction in vision research across species and brain regions. For example, Brain-Score compares neurally mechanistic models of primate ventral stream visual function with respect to their neural and behavioural alignment [48]; the Sensorium Challenge evaluates ANNs of large-scale neuronal activity in the mouse early visual cortex [26]; and the Algonauts Project [49] ranks ANN models of human whole brain processing of multimodal input. These initiatives reflect a shift in computational cognitive and systems neuroscience towards collaborative and comparative modelling and benchmarking. OpenRetina presents a first step towards an analogous platform for retinal processing.

6.2. Current limitations and outlook

`openretina` provides a strong foundation for modelling retinal computation, yet several areas remain open for further development—each presenting an opportunity for collaboration and contribution.

Our focus in developing `openretina` so far has been on establishing a flexible and extensible framework, but going forward, we plan to integrate a broader set of models. The package currently focuses on deep neural networks, which have demonstrated excellent predictive performance, but mechanistic models remain essential for understanding the retina's computational principles [47, 50–52]. This is where the modular design of `openretina` comes into play. It makes it possible to extend the framework to support also circuit-based or biophysical models. This will then allow to directly compare data-driven and mechanistic approaches, and combine them in hybrid approaches.

Another important direction is addressing generalisation across datasets and drawing comparisons across species [53]. Most existing models are trained and evaluated on individual datasets, making it difficult to assess the extent to which learned representations reflect universal principles of retinal computation. Expanding `openretina` to facilitate systematic cross-datasets and cross-species comparisons will help uncover both shared and species-specific processing strategies in the retina.

Finally, a key challenge is the limited availability of open retinal datasets. For our initiative to reach its full potential as a platform for comparative modelling, more experimental data needs to be shared by the community. We encourage researchers to contribute datasets and collaborate on expanding the available data landscape.

As neuroscience moves toward more open and comparative approaches to modelling, `openretina` aims to be a step in this direction for retinal research. By bringing together datasets, models, and analysis tools in a shared framework, we seek to accelerate scientific discovery and provide a valuable resource for the broader field of computational and systems neuroscience.

Acknowledgments

We thank Jonathan Oesterle for valuable feedback on the manuscript. This work was supported by the German Research Foundation (DFG): SFB 1233 “Robust Vision: Inference Principles and Neural Mechanisms” (TP D2), project number: 276693517; EU 42/12-1, project number: 505379160. We thank the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for supporting Federico D’Agostino. MB acknowledges financial support by the Federal Ministry of Education and Research (BMBF), FKZ: 01IS24085B

References

1. D'Agostino, F., Zenkel, T. & Höfling, L. *openretina* version 1.0.0. 2025. doi:10.5281/zenodo.14988976. https://github.com/open-retina/open-retina/.
2. Enroth-Cugell, C. & Robson, J. G. The Contrast Sensitivity of Retinal Ganglion Cells of the Cat. *The Journal of Physiology* **187,** 517–552. doi:10.1113/jphysiol.1966.sp008107 (1966).
3. Chichilnisky, E. J. A Simple White Noise Analysis of Neuronal Light Responses. *Network (Bristol, England)* **12,** 199–213 (2001).
4. Pillow, J. W., Shlens, J., Paninski, L., Sher, A., Litke, A. M., Chichilnisky, E. J. & Simoncelli, E. P. Spatio-temporal correlations and visual signalling in a complete neuronal population. en. *Nature* **454,** 995–999 (2008).
5. McIntosh, L., Maheswaranathan, N., Nayebi, A., Ganguli, S. & Baccus, S. Deep learning models of the retinal response to natural scenes. *Advances in neural information processing systems* **29** (2016).
6. Batty, E., Merel, J., Brackbill, N., Heitman, A., Sher, A., Litke, A., Chichilnisky, E. J. & Paninski, L. *Multilayer Recurrent Network Models of Primate Retinal Ganglion Cell Responses* in *International Conference on Learning Representations* (2017).
7. Klindt, D., Ecker, A. S., Euler, T. & Bethge, M. *Neural system identification for large populations separating "what" and "where"* in *Advances in Neural Information Processing Systems* (eds Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S. & Garnett, R.) **30** (Curran Associates, Inc., 2017).
8. Ding, X., Lee, D., Grant, S., Stein, H., McIntosh, L., Maheswaranathan, N. & Baccus, S. *A mechanistically interpretable model of the retinal neural code for natural scenes with multiscale adaptive dynamics* in *2021 55th Asilomar Conference on Signals, Systems, and Computers* (2021), 287–291. doi:10.1109/IEEECONF53345.2021.9723187.
9. Qiu, Y., Klindt, D., Szatko, K. P., Gonschorek, D., Hoefling, L., Schubert, T., Busse, L., Bethge, M. & Euler, T. Efficient Coding of Natural Scenes Improves Neural System Identification. *PLOS Computational Biology* **19,** e1011037. doi:10.1371/journal.pcbi.1011037 (2023).
10. Maheswaranathan, N., McIntosh, L. T., Tanaka, H., Grant, S., Kastner, D. B., Melander, J. B., Nayebi, A., Brezovec, L. E., Wang, J. H., Ganguli, S. & Baccus, S. A. Interpreting the Retinal Neural Code for Natural Scenes: From Computations to Neurons. *Neuron* **111,** 2742–2755.e4. doi:10.1016/j.neuron.2023.06.007 (2023).
11. Sridhar, S., Vystrčilová, M., Khani, M. H., Karamanlis, D., Schreyer, H. M., Ramakrishna, V., Krüppel, S., Zapp, S. J., Mietsch, M., Ecker, A. & Gollisch, T. Modeling Spatial Contrast Sensitivity in Responses of Primate Retinal Ganglion Cells to Natural Movies. *bioRxiv,* 2024.03.05.583449. doi:10.1101/2024.03.05.583449 (2024).
12. Vystrčilová, M., Sridhar, S., Burg, M. F., Gollisch, T. & Ecker, A. S. Convolutional Neural Network Models of the Primate Retina Reveal Adaptation to Natural Stimulus Statistics. *bioRxiv,* 2024.03.06.583740. doi:10.1101/2024.03.06.583740 (2024).
13. Ecker, A. S., Sinz, F. H., Froudarakis, E., Fahey, P. G., Cadena, S. A., Walker, E. Y., Cobos, E., Reimer, J., Tolias, A. S. & Bethge, M. *A Rotation-Equivariant Convolutional Neural Network Model of Primary Visual Cortex* in *International Conference on Learning Representations* (2018).
14. Cadena, S. A., Denfield, G. H., Walker, E. Y., Gatys, L. A., Tolias, A. S., Bethge, M. & Ecker, A. S. Deep Convolutional Models Improve Predictions of Macaque V1 Responses to Natural Images. *PLOS Computational Biology.* doi:10.1371/JOURNAL.PCBI.1006897 (2019).
15. Burg, M. F., Cadena, S. A., Denfield, G. H., Walker, E. Y., Tolias, A. S., Bethge, M. & Ecker, A. S. Learning Divisive Normalization in Primary Visual Cortex. *PLOS Computational Biology* **17,** e1009028. doi:10.1371/journal.pcbi.1009028 (2021).
16. Cowley, B. R., Stan, P. L., Pillow, J. W. & Smith, M. A. Compact Deep Neural Network Models of Visual Cortex. *bioRxiv.* doi:10.1101/2023.11.22.568315 (2023).
17. Du, F., Núñez-Ochoa, M. A., Pachitariu, M. & Stringer, C. Towards a Simplified Model of Primary Visual Cortex. *bioRxiv,* 2024.06.30.601394. doi:10.1101/2024.06.30.601394 (2024).
18. Olah, C., Mordvintsev, A. & Schubert, L. Feature Visualization. *Distill* **2,** e7. doi:10.23915/distill.00007 (2017).
19. Walker, E. Y., Sinz, F. H., Cobos, E., Muhammad, T., Froudarakis, E., Fahey, P. G., Ecker, A. S., Reimer, J., Pitkow, X. & Tolias, A. S. Inception Loops Discover What Excites Neurons Most Using Deep Predictive Models. *Nature Neuroscience* **22,** 2060–2065. doi:10.1038/s41593-019-0517-x (2019).
20. Bashivan, P., Kar, K. & DiCarlo, J. J. Neural Population Control via Deep Image Synthesis. *Science* **364,** eaav9436. doi:10.1126/science.aav9436 (2019).
21. Ding, Z., Tran, D. T., Ponder, K., Cobos, E., Ding, Z., Fahey, P. G., Wang, E., Muhammad, T., Fu, J., Cadena, S. A., Papadopoulos, S., Patel, S., Franke, K., Reimer, J., Sinz, F. H., Ecker, A. S., Pitkow, X. & Tolias, A. S. Bipartite Invariance in Mouse Primary Visual Cortex. *bioRxiv.* doi:10.1101/2023.03.15.532836 (2023).

22. Burg, M. F., Zenkel, T., Vystrčilová, M., Oesterle, J., Höfling, L., Willeke, K. F., Lause, J., Müller, S., Fahey, P. G., Ding, Z., Restivo, K., Sridhar, S., Gollisch, T., Berens, P., Tolias, A. S., Euler, T., Bethge, M. & Ecker, A. S. *Most discriminative stimuli for functional cell type clustering* in *The Twelfth International Conference on Learning Representations* (2024).
23. Höfling, L., Szatko, K. P., Behrens, C., Deng, Y., Qiu, Y., Klindt, D. A., Jessen, Z., Schwartz, G. W., Bethge, M., Berens, P., Franke, K., Ecker, A. S. & Euler, T. A Chromatic Feature Detector in the Retina Signals Visual Context Changes. *eLife* **13,** e86860. doi:10.7554/eLife.86860 (2024).
24. Willeke, K., Sinz, F., Blessing, C., Walker, E. Y., Bashiri, M., kklurz, Jagadish, A. K., Burg, M., cgruner, Nix, A., PollyTur, Ustyuzhaninov, I., Cadena, S., M00mo, Pierzchlewicz, P. & Synicix. *Sinzlab/Neuralpredictors: Prerelease 2023* Zenodo. 2023. doi:10.5281/zenodo.7702134. 10.5281/zenodo.7702134.
25. Willeke, K. F., Fahey, P. G., Bashiri, M., Pede, L., Burg, M. F., Blessing, C., Cadena, S. A., Ding, Z., Lurz, K.-K., Ponder, K., *et al.* The sensorium competition on predicting large-scale mouse primary visual cortex activity. *arXiv preprint arXiv:2206.08666* (2022).
26. Turishcheva, P., Fahey, P. G., Vystrčilová, M., Hansel, L., Froebe, R., Ponder, K., Qiu, Y., Willeke, K. F., Bashiri, M., Wang, E., *et al.* The dynamic sensorium competition for predicting large-scale mouse visual cortex activity from videos. *arXiv preprint,* arXiv–2305 (2024).
27. Falcon, W. & The PyTorch Lightning team. *PyTorch Lightning* version 1.4. 2019. doi:10.5281/zenodo.3828935. https://github.com/Lightning-AI/lightning.
28. Yadan, O. *Hydra - A framework for elegantly configuring complex applications* Github. 2019. https://github.com/facebookresearch/hydra.
29. Karamanlis, D., Khani, M. H., Schreyer, H. M., Zapp, S. J., Mietsch, M. & Gollisch, T. Nonlinear Receptive Fields Evoke Redundant Retinal Coding of Natural Scenes. *Nature,* 1–8. doi:10.1038/s41586-024-08212-3 (2024).
30. The HDF Group. *Hierarchical Data Format, version 5* https://github.com/HDFGroup/hdf5.
31. Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., *et al.* Array programming with NumPy. *Nature* **585,** 357–362 (2020).
32. Sinz, F., Ecker, A. S., Fahey, P., Walker, E., Cobos, E., Froudarakis, E., Yatsenko, D., Pitkow, Z., Reimer, J. & Tolias, A. *Stimulus Domain Transfer in Recurrent Models for Large Scale Cortical Population Prediction on Video* in *Advances in Neural Information Processing Systems* **31** (Curran Associates, Inc., 2018).
33. Willeke, K. F., Restivo, K., Franke, K., Nix, A. F., Cadena, S. A., Shinn, T., Nealley, C., Rodriguez, G., Patel, S., Ecker, A. S., Sinz, F. H. & Tolias, A. S. Deep Learning-Driven Characterization of Single Cell Tuning in Primate Visual Area V4 Unveils Topological Organization. *bioRxiv,* 2023.05.12.540591. doi:10.1101/2023.05.12.540591 (2023).
34. Lurz, K.-K., Bashiri, M., Willeke, K., Jagadish, A., Wang, E., Walker, E. Y., Cadena, S. A., Muhammad, T., Cobos, E., Tolias, A. S., Ecker, A. S. & Sinz, F. H. *Generalization in data-driven models of primary visual cortex* in *International Conference on Learning Representations* (2021).
35. Turishcheva, P., Burg, M. F., Sinz, F. H. & Ecker, A. S. *Reproducibility of Predictive Networks for Mouse Visual Cortex* in *The Thirty-eighth Annual Conference on Neural Information Processing Systems* (2024).
36. Wang, E. Y., Fahey, P. G., Ponder, K., Ding, Z., Chang, A., Muhammad, T., Patel, S., Ding, Z., Tran, D., Fu, J., Papadopoulos, S., Franke, K., Ecker, A. S., Reimer, J., Pitkow, X., Sinz, F. H. & Tolias, A. S. Towards a Foundation Model of the Mouse Visual Cortex. *bioRxiv,* 2023.03.21.533548. doi:10.1101/2023.03.21.533548 (2023).
37. Ustyuzhaninov, I., Burg, M. F., Cadena, S. A., Fu, J., Muhammad, T., Ponder, K., Froudarakis, E., Ding, Z., Bethge, M., Tolias, A. S. & Ecker, A. S. Digital Twin Reveals Combinatorial Code of Non-Linear Computations in the Mouse Primary Visual Cortex. *bioRxiv,* 2022.02.10.479884. doi:10.1101/2022.02.10.479884 (2022).
38. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
39. Lurz, K.-K., Bashiri, M., Walker, E. Y. & Sinz, F. H. *Bayesian Oracle for Bounding Information Gain in Neural Encoding Models* in *The Eleventh International Conference on Learning Representations* (2022).
40. Developers, T. *TensorFlow* version v2.18.0. 2024. doi:10.5281/zenodo.13989084.
41. Biewald, L. *Experiment Tracking with Weights and Biases* 2020. https://www.wandb.com/.
42. Lettvin, J. Y., Maturana, H. R., McCulloch, W. S. & Pitts, W. H. What the Frog's Eye Tells the Frog's Brain. *Proceedings of the IRE* **47,** 1940–1951 (1959).
43. Szatko, K. P. & Franke, K. What the eye tells the brain: retinal feature extraction. *Neuroforum* **28,** 13–19 (2022).
44. Reinhard, K. & Münch, T. A. Visual properties of human retinal ganglion cells. *PLoS One* **16,** e0246952 (2021).
45. Karamanlis, D., Schreyer, H. M. & Gollisch, T. Retinal encoding of natural scenes. *Annual Review of Vision Science* **8,** 171–193 (2022).
46. Kerschensteiner, D. Feature detection by retinal ganglion cells. *Annual review of vision science* **8,** 135–169 (2022).
47. Schröder, C., Klindt, D., Strauss, S., Franke, K., Bethge, M., Euler, T. & Berens, P. *System Identification with Biophysical Constraints: A Circuit Model of the Inner Retina* in *Advances in Neural Information Processing Systems* **33** (2020).

48. Schrimpf, M., Kubilius, J., Hong, H., Majaj, N. J., Rajalingham, R., Issa, E. B., Kar, K., Bashivan, P., Prescott-Roy, J., Geiger, F., Schmidt, K., Yamins, D. L. K. & DiCarlo, J. J. Brain-Score: Which Artificial Neural Network for Object Recognition is most Brain-Like? *bioRxiv.* doi:10.1101/407007 (2018).
49. Gifford, A. T., Bersch, D., St-Laurent, M., Pinsard, B., Boyle, J., Bellec, L., Oliva, A., Roig, G. & Cichy, R. M. The Algonauts Project 2025 Challenge: How the Human Brain Makes Sense of Multimodal Movies. *arXiv preprint.* doi:10.48550/arXiv.2501.00504 (2025).
50. Lappalainen, J. K., Tschopp, F. D., Prakhya, S., McGill, M., Nern, A., Shinomiya, K., Takemura, S.-y., Gruntman, E., Macke, J. H. & Turaga, S. C. Connectome-Constrained Networks Predict Neural Activity across the Fly Visual System. *Nature* **634,** 1132–1140. doi:10.1038/s41586-024-07939-3 (2024).
51. Deistler, M., Kadhim, K. L., Pals, M., Beck, J., Huang, Z., Gloeckler, M., Lappalainen, J. K., Schröder, C., Berens, P., Gonçalves, P. J. & Macke, J. H. Differentiable Simulation Enables Large-Scale Training of Detailed Biophysical Models of Neural Dynamics. *bioRxiv.* doi:10.1101/2024.08.21.608979 (2024).
52. Idrees, S., Manookin, M. B., Rieke, F., Field, G. D. & Zylberberg, J. Biophysical Neural Adaptation Mechanisms Enable Artificial Neural Networks to Capture Dynamic Retinal Computation. *Nature Communications* **15,** 5957. doi:10.1038/s41467-024-50114-5 (2024).
53. Baden, T., Euler, T. & Berens, P. Understanding the retinal basis of vision across species. *Nature Reviews Neuroscience* **21,** 5–20. doi:10.1038/s41583-019-0242-1 (2020).