DynVision: A Toolbox for Biologically Plausible Recurrent Convolutional Networks

Robin Gutzen
Center for Data Science
New York University
New York, USA
robin.gutzen@nyu.edu

Grace W Lindsay
Center for Data Science and Department of Psychology
New York University
New York, USA
gm3239@nyu.edu

December 1, 2025

Abstract

Convolutional Neural Networks (CNNs) have demonstrated remarkable success in image recognition and exhibit conceptual similarities to the primate ventral visual pathway. Adding recurrence opens the door to exploring temporal dynamics and investigating mechanisms underlying recognition robustness, attentional modulation, and rhythmic perception phenomena. However, modeling spatiotemporal dynamics of biological vision using CNN-based architectures remains challenging. Incorporating functionally beneficial recurrence, capturing biologically plausible temporal phenomena such as adaptation and subadditive temporal summation, and maintaining topographic organization aligned with cortical structure require significant computational considerations. Although recent advances have incorporated neurobiological constraints, the field lacks accessible tools for efficiently integrating, testing, and comparing these approaches. Here, we introduce DynVision, a modular toolbox for constructing and evaluating recurrent convolutional neural networks (RCNNs) with biologically inspired dynamics. Our approach facilitates the incorporation of key visual cortex properties, including realistic recurrent architectures, activity evolution governed by dynamical systems equations, and structured connectivity reflecting cortical arrangements, while maintaining computational efficiency. We demonstrate the framework's utility through systematic analysis of emergent neural dynamics, highlighting how different biologically motivated modifications shape scientifically-relevant response patterns. Code can be found at: `https://github.com/Lindsay-Lab/DynVision/`

1 Introduction

The primate visual system is characterized by abundant recurrent connections [?]. In the ventral visual stream, for example, lateral recurrent connections exist amongst neurons within a visual cortical region and feedback connections go from higher areas like V4 back to lower ones such as V1. These connections are believed to play a crucial role in the system's remarkable ability to integrate information over time and recognize objects under diverse viewing conditions and with varying degrees of occlusion or noise.

For these reasons, many researchers have attempted to integrate these connections into one of the leading classes of visual system models: convolutional neural networks [?]. Although many past studies on recurrent convolutional neural networks (RCNNs) have found benefits of adding recurrence [? ? ?], others have had mixed results or found that recurrence does not serve the same function as in the brain [? ? ?].

These studies predominantly use discrete-time recurrent models, usually unrolled for only a handful of time steps. This coarse-grain approximation to recurrence cannot capture the full complexity of visual dynamics, and it may make it difficult for the network to learn the right parameters that replicate biology. The discrete-time approach is also at odds with more traditional methods in computational neuroscience, which treat neural circuits as continuous dynamical systems governed by differential equations.

Using continuous-time models in a machine learning setting poses certain engineering challenges. Specifically, neural dynamics are best captured using a small simulation step size (on the order of one millisecond), which requires tens to hundreds of time steps to mimic the visual system's response. This leads to a large computational graph when training with back-propagation through time. The implementation of these networks must therefore be made as efficient as possible to make exploration of relevant scientific questions with them feasible. Here, inspired by recent work that combines continuous-time differential equations with deep convolutional neural networks [**? ? ?**], we build a toolbox to make the use of such models easier and more widely available.

Our toolbox, DynVision, implements numerical ODE solvers to build RCNNs with more precise and realistic temporal dynamics. It allows for many different forms of recurrent connections, with options that are inspired by biology and that can help with efficiency. Users can also control a variety of parameters that govern dynamics within and across regions. Synthesizing approaches from the literature, we also include retina-inspired pre-processing, optional gain modulation, and activity regularization. We hope that providing this toolbox will help the community iterate on models of dynamical visual processing faster and more broadly.

2 Toolbox Design and Implementation

Here, we provide a description of the overarching philosophy and core functionalities of the DynVision toolbox. For a more detailed account of all functionalities, consult the DynVision software documentation. Code can be found at: `https://github.com/Lindsay-Lab/DynVision/`.

2.1 Design Approach

Abstraction Level On the one end of the computational neuroscience modeling spectrum there are detailed neural networks involving morphologies, cell-types, and spikes that focus on simulating realistic neural activity (e.g. NEST, Neuron, BRIAN). On the other end, there are abstract network models often based on machine learning methodologies that focus on learning and optimized behavioral performance such as classification (e.g. VGG, ResNet, Transformers). Complementary to the modeling efforts on either extreme, NeuroAI models (for lack of a better term) focus on information processing with biologically plausible dynamics at a level of abstraction that enables both computational ability and meaningful comparison with neural data and behavioral data. DynVision supports this modeling niche wherein biological realism and performance optimization are both required for the purposes of studying how biological features give rise to behavior.

Biological Analogies Components of an artificial neural network can be mapped to biology in a variety of ways and with varying levels of specificity. Multi-layer CNNs, for example, parallel the hierarchical compositionality of the ventral visual stream (V1 -> V2 -> V4 -> IT), with early layers detecting simple features through small receptive fields analogous to V1 simple with additional pooling operations representing the function of complex cells [**?**]. Deeper layers progressively combine these features into increasingly abstract representations and larger receptive fields. Weight sharing and spatial pooling in CNN models realize a translational invariance that support the learning of generalizable visual features and the recognition of objects regardless of their precise position. The convolutional tensor operations maintain a spatial correspondence of the visual field, representing the retinotopic organization found in the visual cortex.

On the smaller scale, we here interpret each entry in the 3D activation tensor (`[n_channels, y_dim, x_dim]`) as a unit representative of a single neuron or a population of neurons, with the activation value corresponding to an average spiking rate. This interpretation establishes direct correspondences between network components and cortical structures: convolutional operations represent connections between pyramidal neuron populations within or between cortical layers, pooling operations mimic the behavior of complex cells in[**?**], while non-linearities model the mechanism of action potential generation, and the dynamical systems formulation captures membrane potential dynamics.

The hidden states hold several timesteps of activity; accessing different timestep slots in the hidden states can represent the conduction delays that can vary across connection types (see Figure 1). Recurrent connections communicate delayed activity within a cortical area, potentially mediated by interneuron populations. Skip and feedback connections represent inter-areal communication between hierarchical cortical regions (e.g., V4->V1 feedback). These biological analogies actively inform architectural choices, as cortical morphology and connectivity paradigms constrain the network design space. Building off these analogies, we believe that more biologically plausible recurrent convolutional networks may capture aspects of human visual behavior and its computational efficiency.

In the following sections we provide an overview of the options our toolbox provides to users, motivated by what is needed in order to efficiently explore biologically-inspired dynamics.

2.2 Temporal Dynamics

Dynamical System Network Description In biological neural networks, activity evolves over time and space in a structured and continuous manner, based on past input and intrinsic network properties. DynVision implements numerical ODE solvers to build RCNNs with more precise and realistic temporal dynamics. This is often modeled within a dynamical system formulation as the differential equation in Figure 2 (1) [**? ?**]. The toolbox incorporates this dynamical systems formulation by accordingly evolving the activity over time using a stepwise numerical differential equation solver (Euler method, Figure 2 (2)), which evolves a layer's activity on a timescale τ for the next time step t based on the layer's activity of the previous timestep $t-dt$ and any input to this layer arriving at that timestep.

Note that some details of this formalism can modified by reordering the order of operations that are executed for each layer (with the `layer_operations` list parameter; see 2.3.4). This can include placing certain inputs outside of the activation function or moving the bias outside of the temporal evolution step.

Heterogeneous Delays Different neural connections are governed by different temporal delays, reflecting conduction velocities and the spatial distance between the neurons. Our toolbox allows for temporal unrolling with separate time delays for different types of connections between network layers. Specifically, at time t a layer can receive input from its own past activity r_l at $t-\Delta_{RC}$ via the laterally recurrent connections J_{RC}, feedforward input from the preceding layer r_{l-1} at $t-\Delta_{FF}$ via the connectivity J_{FF}, and recurrent feedback connections from a succeeding layer $r_{>l}$ via J_{FB}. By varying these parameters, users can systematically evaluate the influence of different temporal delays on network dynamics.

Unrolling of Time This system of hetereogenous delays naturally allows for unrolling in "biological" time, but it can just as well unrolling in "engineering" time by simply setting the feedforward delay $\Delta_{FF}=0$, effectively treating the network as an instantaneous system (Figure 3). Unrolling in engineering time is still possible if the network has skip or feedback connections. To switch from biological to engineering time, the skip connections need to be subtracted by the feedforward delay between source and target layer: for example, a skip delay $\Delta_{SK}=2\Delta_{FF}+1$ becomes $\Delta_{SK}=1$ if the skip connection spans two layers (assuming Δ_{SK} remains ≥ 0). Feedback connection delays need to be extended by the feedforward delay between target and source layer, for example, a feedback delay $\Delta_{FB}=\Delta_{FF}$ becomes $\Delta_{FB}=2\Delta_{FF}$ if the connection goes back one layer.

To make sure an input with a given number of timesteps is completely processed by the classifier, the simulation time is automatically extended by the number of residual timesteps it takes for the first input signal to reach the final layer. When simulating the network in engineering time, the residual timesteps are per definition zero.

Simulating a model in engineering time is slightly more computationally efficient. In an example 3.5h training run using the DyRCNNx4 with full recurrence, ($\Delta_{FF}=(0|10)$, $\Delta_{RC}=6$, $\Delta_{SK}=(2|22)$, $\Delta_{FB}=(30|10)$ on cifar100 with 30 timesteps), the time per epoch decreases by $\sim 29\%$ and memory demand decreased from 2.39 GB to 2.13 GB for unrolling in engineering time compared to biological time. Note: engineering time reduces the number of simulation steps but it can increase the length of the hidden state due to larger Δ_{FB} leading to larger memory demands.

Temporally Varied Input Presentation For modeling temporal model dynamics the input has to presented over multiple timesteps. The input can be presented as static image over time, combined with a null inputs for certain timesteps, or potentially undergo more complex time-dependent transformations (e.g., by varying the contrast).

The toolbox provides several options to extend static input images over time. By setting the `data_timesteps > 1` config parameter, the dataloaders create an extended view on the data along the time dimension without duplicating the data. This functionality is implemented with the pytorch dataloaders and the ffcv dataloaders. Alternatively, by setting the `data_timesteps = 1` and instead setting the `n_timesteps > 1`, the dataloader load only single input images that are only extend in the forward pass during the model run.

The time extension at runtime can be varied by setting the `data_presentation_pattern` to any boolean string such as `01110`, indicating a temporal sequence of null input (`0`) and input image (`1`). The pattern string is always proportionally stretched or compressed so that its length equals `n_timesteps` and thus each value corresponds to one timestep. The same functionality is also available by using the time extension with pytorch dataloaders with the option to also realize more complex temporal transformations.

Overwriting and parameterizing the iterator of the standard pytorch dataloader offers different testing scenarios analogous to common experimental protocols. For example, our 'StimulusDuration' loader presents an image after a delay δ_i for δ_s duration followed by silent outro period δ_o; the 'StimulusInterval' behaves similarly besides also presenting a second identical stimulus with a δ_d delay after the first; the 'StimulusContrast' loader adds to the StimulusDuration setup by multiplying the image input with a scalar to vary its contrast.

2.3 Network Connectivity

2.3.1 Recurrent Connection Types

DynVision contains several options for recurrent connectivity patterns. These options represent different beliefs about the spatial and feature extent of lateral connections. Specifically:

self-recurrence connects a unit only to itself. Input is determined by multiplying the feedforward activation tensor with a scalar weight ([**?**], mimicking connections within cortical columns Figure 4).

full recurrence applies a standard kernel convolution so that a unit is influenced by a nearby spatial region across all channels, capturing broad lateral connectivity across feature maps. This was introduced by [**?**] and is frequently used in the literature (e.g. [**?**], Figure 4).

depthpointwise recurrence instead of a full convolution, uses a depthwise (spatial dimension) and then a pointwise (feature dimension) convolution (see Figure 4). This is also known as a depthwise separable convolution and has been used by the computer vision community ([**?**]).While not commonly used in neuroscientific models, it does map onto the structure of lateral recurrence in the visual system observed, for example, in orientation pinwheels and topographic maps. Specifically, visual neurons tend to get input from other neurons that 1.) represent the same spatial location but have different feature preferences (depthwise) and 2.) have the same preferred features but represent different locations in space (pointwise) [**?**]. Depthwise separable convolutions are desirable from an engineering perspective as they reduce the number of parameters compared to a full convolution.

pointdepthwise recurrence Compared to the depthpointwise recurrence the order of the two partial convolutions is inverted. This follows from the idea that neurons representing similar locations in space are arranged closer together in cortical space so that a signal propagation across features (pointwise) would precede the signal exchange across locations (depthwise) (Figure 4).

local recurrence aims to capture the 2-D topology of visual cortices. Here, all units in a layer are systematically arranged on a 2-D grid (inspired by cortical organization such as orientation pinwheels in V1 ([**?**], see Figure 5). This mapping requires that the shape information of the input tensor be provided as `dim_y` and `dim_x`. A convolution with kernel size > 1 is applied to this grid. The result is that input to each unit is a combination of cortically-local feature and space information (Figure 5. This approach is inspired by LLCNNs [**? ?**] but also incorporates the spatial retinotopy in the 2-D map. Our approach has similarities to other topographic mappings like All-TNN [**?**] and TDANN [**?**], which also map to a combined feature-spatial plane, but have an explicit spatial loss instead of doing a convolution with weight-sharing on this plane. This mapping to a 2-D topology requires that the number of channels is a square. If this is not the case, the mapping extends the number of channels to the next square number by duplicating the last channels as with a reflecting boundary. The supplemented channels are removed again in the mapping back to the 3-D tensor.

localdepthwise recurrence The local recurrence can be extended with an additional depthwise convolution mimicking patchy long-range connections in the visual cortex [**?**] between units with different feature preferences but the same receptive field.

Standardly, the connectivity matrices J are implemented via kernel convolutions, which reduces the number of parameters to learn (biologically, this corresponds to an assumption that local connectivity statistics are similar for neurons across the cortical sheet).

In our toolbox, the module `RecurrentConnectedConv2d` combines a feedforward 2D convolution (`torch.nn.Conv2d`) operation with a recurrency operation of choice (`recurrence_type`) and handles the storage and integration of corresponding hidden states with a variable time delay (`t_recurrence`). The module can extended with a second feedforward convolution by setting a `mid_channels` attributes, as in several popular architectures [**?**].

2.3.2 Skip and Feedback Connections

Besides lateral recurrent connections that link units within one layer across time steps, skip and feedback connections link units across layers and time steps. Anatomical studies reveal that feedback projections from higher cortical areas like V4 and IT back to V1 are as numerous as feedforward connections, suggesting a fundamental role in visual computation [**? ?**].

Skip and feedback connections describe a similar process: a copy of a layer's output is diverted from the immediate feedforward path to be integrated with to a downstream layer (skip) or an upstream layer (feedback). Thus, we provide a generalized module implementation that covers both cases, and use the same integration strategies as available for the recurrent connection (e.g. additive or multiplicative). Similar to lateral recurrent connections, the shapes of the source and target layers may not necessarily match, in which case a convolution with a 1x1 kernel is applied to adjust the number of channels, and an up or downsampling operation adjusts the spatial dimensions.

To avoid requiring the user to determine the right layer shapes and corresponding transformations in advance, the toolbox offers an `auto_adapt` option, so that the connection can be defined by referencing a source and target layer, and the correct transformation is created upon the first forward pass. This facilitates architecture exploration because connections between layers and orders of operation can easily be swapped without causing errors.

2.3.3 Integration of (Recurrent) Connections

Some studies suggest that recurrence may be best modeled as a gain modulation (e.g.[**?**]). Therefore, in DynVision, the way the recurrent signal is integrated with bottom-up input can be set by `integration_strategy` to either 'additive' ($x' = x + h$) or 'multiplicative' ($x' = x * (1 + torch.tanh(h))$) or a custom callable function. For test runs the recurrence can be disabled by setting `feedforward_only = True`.

Where the recurrent input is integrated with regards to the convolutions can be selected with the argument `recurrence_target` to either 'input', 'output', or 'middle' (corresponding to prior to any convolutions, between convolution operators, or after all convolutions; see 1). In case there is a mismatch in the spatial dimensions of the recurrent activity tensor with the feedforward activity tensor, an additional up or downsampling is applied via an interpolation or a stride in the convolution operation.

2.3.4 Order of Layer Operations

The model base classes provide a structure to name and execute the operations of each layer in a flexible order. The activations of each layer are computed by calling its layer operations in the order defined by the `layer_operations` attribute. If a given operation is not defined for a layer, it is skipped.

For example, the order of operations used the DyRCNN models presented in this paper is (unless otherwise noted):

```
layer_operations:
    - "rconv"        # apply recurrent convolutional module
    - "addext"       # add external input
    - "addskip"      # add activity from skip connections
    - "addfeedback"  # add activity from feedback connections
    - "tstep"        # apply dynamical systems ode solver step
    - "nonlin"       # apply nonlinearity
    - "delay"        # set and get delayed activations
    - "record"       # record activations in storage buffer
    - "pool"         # apply pooling
```

The naming of the layers and operations is chosen by the user, except for the reserved operation names: `delay`, which writes the current activity into the hidden states and retrieves a past activity with the correct delay; `tstep`, which accesses the most recent hidden state); and `record`, which stores the current activity in the model's storage buffer.

Any custom model only needs to define its layer names (e.g. `self.layer_names = ['V1', 'V2', 'V4', 'IT']`) and modules in a `_define_architecture()`, following the naming convention `self.<operation>_<layer_name> = <module>`. For layer-unspecific operations, e.g. a non-linearity, the `_<layer_name>` can be omitted.

2.4 Loss Functions

The toolbox supports the application of multiple loss functions to be selected in the configs. The default is `loss: ['CrossEntropyLoss', 'EnergyLoss']`. These represent task accuracy and neural activity sparseness as follows:

Category Loss Per default we use we use the widely used cross-entropy-loss function as category loss. The category loss automatically ignores any model output for the residual timesteps, for which model input hasn't yet reached the classifier. Additionally, there is a `loss_reaction_time` setting that further removes the initial n timesteps (after any residual timesteps) from consideration for the category loss as desired.

Energy Loss Activity regularization promotes more stable network activity and is biologically motivated by mechanisms of homeostatic plasticity [**?**]. In the toolbox, we add an effective activity regularization via an energy loss function [inspired by **?**] that penalizes large unit responses, thus mimicking the metabolic constraints of spike generation in the brain. This loss function accesses the activations of any convolutional or linear module from the computational graph without the need for any additional storing of activation tensors. The energy loss is defined as the p-norm (default is $p = 1$) of the activation tensor r_m of module m divided by its number of units N_m and averaged across all modules M (Eq. 4).

$$\mathcal{L}_{energy} = \frac{1}{M} \sum_{m=1}^{M} \left(\frac{1}{N_m} \sum_{n=1}^{N_m} |r_{m,n}|^p \right)^{\frac{1}{p}} \tag{4}$$

Adding a weight decay further adds weight regularization to promotes network stability and more efficient and sparse coding, further modeling metabolic constraints.

Any loss function-specific configurations are set in the confis as dicts with `loss_config: [<loss_name>: <loss_configs>]`. A weighting of the individual loss contributions is coordinated with a `weight` scalar value in the loss configs. The available loss functions can be extended by adding a corresponding implementation or wrapper class for a pytorch loss function to the 'dynvision/losses/' folder. The BaseLoss parent class provides consistent input processing, shape validation, and reduction over the batch and time dimension.

2.5 Additional Biological Components

Supralinearity Individual neurons often exhibit nonlinear amplification, where strong inputs lead to disproportionately large activations. The influence of recurrence is necessary to stabilize this explosive activity [**?**]. The toolbox allows users to apply a supralinear activation function inspired by models of visual cortex [**? ?**] to encourage networks to make effective use of recurrent connections for regularizing feedforward activity.

Input Preprocessing Having a dedicated input layer to preprocess the input stimuli is a common component of vision models. The toolbox includes two biologically inspired versions of this: 1.) a simple adaption factor (fixed or learnable) that reduces the input values to the network over time so that it is necessarily more dependent on recurrent connections to retain information, 2.) a layer representing the retina and LGN and their representational bottleneck, as described in [**?**].

2.6 Reference Models

To facilitate rapid experimentation and benchmarking, the toolbox integrates implementations of several well-established neural network architectures from the literature, including AlexNet [**?**], the ResNet family [**?**], CORNet-RT [**?**], and CordsNet [**?**]. In addition to these models, users can easily instantiate the showcase models introduced in this paper, such as DyRCNNx2, DyRCNNx4, DyRCNNx8. The framework also supports the automatic loading of pre-trained weights, with mechanisms in-place to correctly match parameters even when model configurations have been modified (e.g., additional layers or biologically inspired features added). Notably, models are designed to be data context-aware, meaning they are automatically initialized with the appropriate number of output classes based on the provided dataset. The toolbox further enables selective training of only non-pre-trained parameters or full model fine-tuning, offering flexibility for transfer learning applications.

2.7 Software Design

DynVision is designed with scientific exploration in mind. We follow best principles for scientific software engineering to make the toolbox transparent and easy to use.

Modularity The fundamental advantage of using modeling approaches in neuroscience is that contrary to actual brains models allow for precise control and monitoring of each and every variable and therefore make it possible to probe their respective influence on model behavior. Such probing investigations may take the form of parameter sweeps, changing the order of operations, adding/removing/replacing individual operations, and inserting targeted perturbations. In order to effectively explore these options, the modeling environment must make the interfaces to the variables of interest explicit and make the components interoperable so that the model runs seamlessly in each configuration. The DynVision toolbox employs this paradigm of modularity by constructing core model and training capabilities by combining specialized base classes via multiple inheritance, defining the model's architecture as a sequence of layers, and each layer as a sequence of operations. This two-tier organization allows for flexible arrangement of computational components. Layer operations are built from interchangeable components including recurrence types, connectivity patterns, dynamics solvers, and signal delays which can be reordered and swapped without changing the overall model handling. This modularity enables researchers to conduct controlled experiments where specific components are varied while others are held constant, facilitating systematic exploration of the design space.

Configuration-driven Modeling Our toolbox aims to separate the many scientific modeling decisions from the technical implementation details as well as possible by designing a code framework that is generalized and flexible enough to allow for many variations while remaining computationally efficient. In support of this, model configuration choices are defined in a YAML-based config schema which is integrated into the workflow via a rigorous parameter handling pipeline. The advantages are i) a lower coding barrier for developing and testing complex but efficient models, ii) reproducibilty, reusability, and systematic parameter space space exploration, iii) making architectural choices, like the order of layer operation or type of time-unrolling, explicit and visible, iv) integrating technical advances independent from a model design.

Interoperability To better integrate technical advances, the toolbox relies on: PyTorch [**?**] for performative tensor operations and machine learning utilities; PyTorch lightning [**?**] to seamlessly organize state-of-the-art training, validation, and testing procedures and allocate them on GPU or CPU resources; ffcv dataloader [**?**] to optimize the computational bottleneck of data loading and transfer; the snakemake workflow manager [**?**] to facilitate automatization, scalability, and portability of workflows; yaml config scripts and Pydantic[1] parameter handling to provide a high-level, human-readable interface; and a cookiecutter-based structure to aid with a clear folder organization. This strong reliance on existing tools and developer communities aids in maintainability and further promotes well-defined interfaces for adding new components, models, optimization, and analysis methods

2.7.1 Base Classes

DynVision employs a modular base class architecture that separates concerns into specialized components, each addressing distinct aspects of neural network modeling. This design leverages multiple inheritance and mixin classes to achieve both flexibility and maintainability. A `BaseModel` class integrates all the components while researchers can also compose models with a subset of base classes or add additional parent classes with a specific functionality while ensuring consistent interfaces across the toolbox.

TemporalBase The `TempralBase` class robustly handles the processing of temporally extended input. It implements the utility to effectively manage the time dimension of signal tensors (`[batch_size, n_timesteps, n_channels, dim_y, dim_x]`), including temporal unrolling, and residual timesteps. The `forward()` function processes the input sequentially across time, where at each timestep, the `_forward()` function executes layer-specific operations in a configurable order (see Section 2.3.4).

LightningBase The `LightningBase` class integrates PyTorch Lightning to address much of the engineering overhead in training models in PyTorch, including automatic mixed-precision training, automatic logging, checkpointing, and hardware acceleration across single GPUs to multi-node clusters, and hyperparameter optimization utilites. This class specifically provides further abstraction by implementing a Lightning model tailored for training recurrent convolutional networks with all relevant configuration parameters set to reasonable defaults and editable the human-readable config files.

StorageBuffer The `StorageBuffer` component addresses the unique memory requirements of storing model output, activations, and hidden states across timesteps. It selectively records the model in- and out- puts and layer activations in training, validation, and testing context with fixed or cyclic recording strategies, variable buffer sizes, and optional CPU offloading strategies to capture population dynamics without exhausting GPU memory. This capability is crucial for comparing model responses against electrophysiological recordings, as it preserves the full temporal evolution of neural activations across the model over extended stimulus presentations. Further, the class implements utility functions for managing and exporting the recordings as tensor, dict, or pandas.DataFrame that combines labels, predictions, batch and sample indices, timesteps, and classifier unit activations.

DtypeDeviceCoordinator While most of the device and dtype coordination is already handled efficiently by PyTorch Lightning, the `DtypeDeviceCoordinator` system further ensures dtype and device consistency across child and parent

[1] `https://docs.pydantic.dev/latest/`

class (that may not be derived from a LightningModule), variables in storage buffers, and between dataloader and model trainer.

Monitoring The `Monitoring` class provides structured logging and debugging of the model, training, and system parameters, including detailed memory usage, including automatic detection of common numerical issues (NaN gradients, parameter drift), systematic tracking of parameter evolution, and real-time visualization of network dynamics and weight distributions. The logging capabilities integrate with the Lightning logging mechanism and its interface to external monitoring tools such as Weights&Biases [2].

2.7.2 Parameter Handling

Hierarchical Configuration The toolbox implements a comprehensive parameter management framework that separates configuration from code implementation, enhancing reproducibility and systematic exploration of biological parameter spaces. The system employs human-readable YAML configuration files organized hierarchically across specialized domains: `config_defaults.yaml` establishes baseline model parameters and system settings, `config_data.yaml` defines dataset specifications `config_experiments.yaml` specifies data selection and parameter sweeps protocols for model testing scenarios. The `config_workflow.yaml` file sits at the top of the hierarchy and defines any parameters that are relevant for the current application overwriting duplicate default settings in other config files. Further, `config_modes.yaml` provide whole parameter presets to rapidly switch between configuration contexts such as (`local` for local test runs, `debug` for verbose logging, `large_dataset` for memory-optimized processing, and `distributed` for multi-device cluster execution. This hierarchical organization enables researchers to override specific parameters at appropriate abstraction levels while maintaining consistent defaults.

Parameter Expansions and Overwrites The Snakemake workflow system extends this configuration framework through dynamic parameter expansion and command-line integration. Configuration parameters can be selectively overridden via command-line arguments (`snakemake -config model_name=DyRCNNx4 lr=0.001`), enabling rapid prototyping without modifying configuration files. Parameter lists are automatically expanded into Cartesian products for systematic exploration (`recurrence_type: [full, self, depthwise]` generates separate jobs for each architecture variant), while runtime configurations are automatically archived alongside experimental outputs to ensure complete reproducibility.

Parameter Parsing The script-level parameter handling employs type-safe Pydantic classes that provide robust validation, automatic type conversion, and biological constraint checking. Specialized parameter classes (`ModelParams`, `DataParams`, `TrainerParams`) encapsulate domain-specific validation logic and computed properties, while composite classes (`InitParams`, `TrainingParams`, `TestingParams`) combine these components for script-specific processing. The system supports flexible parameter aliasing (`lr` → `learning_rate`, `rctype` → `recurrence_type`) and implements context-aware validation that enforces system and biological feasibility constraints and cross-parameter consistency checks, e.g., time delays are integer multiples of time resolution, or batch size and learning rate scaling for multi-device training and gradient accumulation.

2.7.3 Data Handling

The toolbox addresses data management challenges in a configuration-driven pipeline that supports the computational demands of temporal modeling while maintaining experimental flexibility.

The toolbox implements a symbolic link-based organization system in the snakemake workflow scripts that eliminates storage redundancy while enabling flexible experimental configurations. Raw datasets are automatically partitioned into training, and testing subsets.

Data subsets can be further subsampled into groups containing only a selection of classes defined by specifying the class indices in `config_data.yaml`. The system automatically handles label index transformations to maintain consistency with the full dataset, ensuring that models trained on complete datasets can be seamlessly evaluated on class subsets. Besides making testing faster, this functionality is particularly to vary the testing difficulty or investigate category-specific neural responses.

For model training, the architecture is validated to be consistent with the training dataset and automatically adapts the classifier layers to the number of presented classes. This streamlines transfer learning and fine-tuning models on different datasets.

[2] https://wandb.ai

Recognizing that temporal neural network models require processing sequences of images over many timesteps, the toolbox integrates FFCV (Fast Forward Computer Vision) [**?**] data loading, achieving speedup compared to standard PyTorch DataLoaders. The accelerated preprocessing pipeline includes GPU-optimized transformations (incl. expanding for expanding timesteps) and direct memory mapping, reducing CPU-GPU transfer overhead that would otherwise dominate training time for models requiring hundreds of temporal integration steps.

2.7.4 Workflow Management

The toolbox employs Snakemake-based [**?**] workflow orchestration to integrate the complete model development and evaluation lifecycle into reproducible, scalable computational pipelines. The rule-based dependency system automatically manages the sequential stages from raw data acquisition through dataset organization, model initialization, training, analysis, and visualization ensuring that each computational step executes only when its prerequisites are satisfied and input data has changed. This approach addresses critical reproducibility challenges by maintaining explicit dependency tracking, assigning parameter selections to specific steps, enabling automatic parallelization of independent tasks, and providing deterministic execution paths that can be precisely replicated across different computing environments.

The workflow architecture seamlessly scales from individual laptops to high-performance computing clusters through adaptive resource management and centralized path configuration, accommodating the substantial memory and computational demands of biologically plausible temporal models that require extended simulation periods with fine-grained timesteps and substantial hidden state storage.

3 Demonstrations and Results

We built this toolbox to generalize biological RCNN network models and make them flexibly editable for systematic exploration. To demonstrate its capabilities, we focus on our primary working model, the DyRCNNx8, inspired by existing architectures in the literature. We systematically explore the modeling parameter space to identify optimal configurations, then validate the biological plausibility of the resulting models by comparing against neural data and behavioral observations. Having established that our DyRCNN models exhibit biologically realistic dynamics, we demonstrate the framework's generality by showing it can precisely recreate established models from the literature (CORnet-RT and CordsNet). Finally, we provide comprehensive computational benchmarking showing that the toolbox delivers significant performance advantages, and detail how specific modeling choices affect training speed, GPU memory usage, and model complexity.

3.1 Model Architecture

The version of the DyRCNNx8 model analyzed throughout this section contains variants of lateral recurrence and skip connections. Impacts of feedback connections will be explored in future work.

The architecture and order of operations is defined as follows:

[V1] **Conv2d**(*in=3, out=64, k=5×5, s=2*) → **ReLU** → **Conv2d**(*in=64, out=64, k=5×5, s=1*) → **+Recurrence** ($\Delta_{RC} = 6ms$) → **Euler Step** ($dt = 2ms, \tau = 9ms$) → **ReLU** → **Delay** ($\Delta_{FF} = 0ms$) → **MaxPool**(*3×3, s=2*)

[V2] **Conv2d**(*in=64, out=144, k=3×3, s=2*) → **ReLU** → **Conv2d**(*in=144, out=144, k=3×3, s=1*) → **+Recurrence** ($\Delta_{RC} = 6ms$) → **Euler Step** ($dt = 2ms, \tau = 9ms$) → **ReLU** → **Delay** ($\Delta_{FF} = 0ms$) → **MaxPool**(*3×3, s=2*)

[V4] **Conv2d**(*in=144, out=256, k=3×3, s=2*) → **ReLU** → **Conv2d**(*in=256, out=256, k=3×3, s=1*) → **+Recurrence** ($\Delta_{RC} = 6ms$) → **+Skip**(*V1*, $\Delta_{SK} = 0ms$) → **Euler Step** ($dt = 2ms, \tau = 9ms$) → **ReLU** → **Delay** ($\Delta_{FF} = 0ms$)

[IT] **Conv2d**(*in=256, out=529, k=3×3, s=2*) → **ReLU** → **Conv2d**(*in=529, out=529, k=3×3, s=1*) → **+Recurrence** ($\Delta_{RC} = 6ms$) → **+Skip**(*V2*, $\Delta_{SK} = 0ms$) → **Euler Step** ($dt = 2ms, \tau = 9ms$) → **ReLU** → **Delay** ($\Delta_{FF} = 0ms$)

[Classifier] **AdaptiveAvgPool2d**(*1*) → **Flatten** → **Linear**(*529 → 10*)

Further parameters and training details can be seen in Table 1.

Table 1: Default Training Configuration

Parameter	Value	Description
Temporal Dynamics		
Time steps	20	Number of simulation timesteps
Resolution (dt)	2 ms	Integration time step
Time constant (τ)	5 ms	Neural time constant
Feedforward delay (Δ_{FF})	30 ms	Feedforward connection delay
Skip connection delay (Δ_{SK})	0 ms	Skip connection delay
Recurrent delay (Δ_{RC})	6 ms	Recurrent connection delay
Integration strategy	Additive	Activity integration method
Dynamics solver	Euler	Numerical ODE solver
Recurrence target	Output	Recurrence applied to layer output
Architecture Configuration		
Skip connections	Enabled	V1$\rightarrow$V4, V2$\rightarrow$IT
Feedback connections	Disabled	No top-down connections
Normalization	None	No BatchNorm/LayerNorm
Training Setup		
Training epochs	200	Total training duration
Datasets	ImageNette	224px, 10 classes
Batch size	256	Training batch size
Initial learning rate	0.0008	Base learning rate
LR scheduler	CosineAnnealingLR	$T_{max} = 250$
Recurrent LR factor	0.2	LR scaling for recurrent weights
Optimizer	Adam	Optimization algorithm
Weight decay	0.0005	L2 regularization
Weight Initialization		
Feedforward weights	Truncated Normal	mean=0.0, std=0.004
Recurrent weights	Uniform	range=[-0.001, 0]
Bias initialization	Zero	All biases set to 0
Loss Function		
Primary loss	Cross-entropy	Classification loss
Auxiliary loss	Energy loss (0.2×)	Biological constraint
Loss reaction time	4 ms	No temporal offset for loss
Optimization		
Gradient accumulation	4 batches	Effective batch size: 1024
Precision	bf16-mixed	Mixed precision training
Gradient clipping	Norm, value=1.0	Gradient norm clipping

3.2 Model Training

3.2.1 Energy Loss and Temporal PPresentation Pattern

A critical consideration when training biologically plausible recurrent networks is balancing task performance with biological constraints. The energy loss function (Eq. 4) implements a metabolic constraint that promotes sparse, efficient neural coding. However, the relative weighting between the cross-entropy classification loss and the energy regularization loss affects both training dynamics and the final network behavior.

To establish appropriate training parameters for subsequent experiments, we systematically explored different energy loss weights during training. Figure 8 shows training and validation accuracy curves, the evolution of loss components, and most importantly, the temporal dynamics of network activity at different stages of training (early, middle, and late epochs).

These results demonstrate that [DESCRIBE KEY FINDINGS - e.g., optimal energy loss weight balances accuracy and biological plausibility, dynamics evolve from unstable early training to stable late training, choice of weight affects final temporal response patterns]. Based on these findings, we selected an energy loss weight of 0.4 and trained models for 300 epochs for all subsequent analyses, ensuring stable dynamics while maintaining high classification performance.

With these training parameters established, we built and trained DyRCNNx8 models with each of our six main forms of lateral recurrence available (self, full, depthpointwise, pointdepthwise, local, and localdepthwise).

As can be seen in Figure 9, models with all forms of recurrence are able to train to high accuracy and show stable output dynamics. The model with **full** recurrence has the most parameters and also highest performance by a small margin.

3.2.2 Equivalence of Engineering and Biological Time Unrolling

A unique feature of DynVision is the ability to train networks using either engineering time (where $\Delta_{FF} = 0$) or biological time (where $\Delta_{FF} > 0$) unrolling. As described in Section 2.2, these two approaches should produce mathematically equivalent dynamics when delays are properly adjusted. We validate this theoretical prediction empirically by training a model in engineering time and testing it in both engineering and biological time configurations.

As shown in Figure 7, the model produces identical temporal dynamics regardless of the unrolling scheme, confirming that researchers can choose the computationally more efficient engineering time for training while maintaining the ability to interpret results in biological time. This flexibility is particularly valuable when comparing model predictions to neural recordings where real-world propagation delays matter.

3.3 Evaluation of Modeling Choices

A main motivation for providing this toolbox is our belief that the research community needs to explore the space of RCNNs more thoroughly in order to understand the impact of different parameter regimes on model properties. Here, we systematically examine how specific modeling choices affect learned dynamics, ultimately justifying the parameter selections used for our DyRCNN models in subsequent biological validation experiments.

3.3.1 Loss Reaction Time

The evaluation of the training loss can be restricted to a subset of timesteps. The exact number and placement of these timesteps varies across studies, including models that use all timesteps [**?**], the last timestep [**?**], or a number of timesteps in between [**?**]. It is therefore important to understand the impacts of this parameter choice.

In Figure **??**, we show dynamics across layers when different timespans of the output are used for the loss function. These choices have clear qualitative impacts on dynamics, particularly at later layers. When the entire output time is used for the loss ($lossrt = 0$), activity ramps up quickly and then settles into a lower stable state. Using only a small number of timepoints at the end of stimulus presentation ($lossrt = 98$) leads to unstable dynamics. Therefore, we can see that while using fewer timesteps reduces computational costs, it also reduces stability and biological plausibility. Note: these results are based on the DyRCNNx4 model.

The results suggest that it is helpful to compute the loss for most of the timesteps in order to promote stability and realistic dynamics. This is further motivated biologially as object classification in biological time is an continously ongoing task that is usually not limited to a certain timespan within the object presentation. **?**] further argue that using all timesteps encourages the model to achieve a correct classification as quickly as possible and therefore produce better interpretabile reaction times. However, we also see that this incentive to classify quickly, when overtrained, can lead to strong overshoot and oscillatory activity that can make the model response more unstable when presented with temporally different inputs. Therefore, we find a small loss reaction time that excludes only the first few timesteps after image presentation (in engineering time) to be a well balanced modelling choice.

3.3.2 Initial Idle Period during Training

Initializing hidden states in recurrent models is realized in various ways. CorNet initializes hidden states with zeros, whereas the hidden states of our DyRCNN model produce None (i.e. no recurrent signal integration) when they are not yet populated. CordsNet addresses this issue with preceeding its forward pass with a number of idle timesteps, where only a null input is provided, to allow self-generated activity to stabilize and populate the hidden states. We systematically tested whether this design choice benefits our DyRCNN architecture. As null input we choose for the idle timeteps we choose zero tensors representing a neutral contrast-free input, which corresponds to the average pixel value, since the input images are normalized.

As shown in Figure 10, for our model there is no benefit for training with initial idle timesteps. This is likely because the energy loss regularization provides sufficient constraint on activity levels from the start of training, eliminating the need for an idle stabilization period. This simplifies the training protocol while maintaining stable dynamics.

3.3.3 Target of Recurrent Connections

Where exactly the recurrent input is incorporated into the feedforward computations is a modeling choice that usually does not receive much explicit attention. Yet, we show here that different choices can lead to qualitatively different dynamics.

In Figure 11, activity dynamics across layers are shown for recurrence fed into the input, middle, and output of a layer. The 'input' setting integrates the recurrent activations with the feedforward activations before they are fed into the convolution(s), the 'output' setting integrates the activations after the convolution(s), and if a `mid_channel` value is provided and there are two convolutions in a layer, the 'middle' setting integrates the recurrence in between the convolutions. In case of 'output' and 'middle' targets, the recurrent activation tensor is upsampled along the spatial dimensions, using a nearest neighbors algorithm, to compensate for the change in tensor shape due to the strides of the convolutions. Feeding recurrent inputs into the output of the layer's computations leads to particularly distinct dynamics. This highlights the need for tools that allow us to thoroughly explore the parameter space of RCNNs for the purposes of biological modeling.

3.3.4 Temporal Parameters

The continuous-time dynamics framework provides several temporal parameters that control the timescale and delays of neural activity evolution. We systematically explored the effects of:

Time Constant τ [TODO: Add content about time constant effects on dynamics]

Recurrence Delay Δ_{RC} [TODO: Add content about recurrence delay effects]

Skip Connection Delay Δ_{SK} [TODO: Add content about skip connection delay effects]

3.3.5 Summary of Parameter Selection

Through this systematic exploration of the modeling parameter space, we established the configuration used for all DyRCNN models in the biological validation experiments: energy loss weight of 0.2, no initial idle period, output-targeted recurrence integration, 200 training epochs, and the temporal parameters specified in Table 1. These choices balance biological plausibility with computational efficiency while producing stable, interpretable dynamics. Having justified these design decisions, we now validate that the resulting models exhibit biologically realistic properties.

3.4 Validation of Biological Plausibility

Having systematically explored the modeling parameter space and established our training protocol, we now validate that the resulting DyRCNN models exhibit key properties expected from biologically plausible neural systems. These analyses demonstrate that our carefully selected parameters produce models with realistic dynamics that match both neural data and behavioral observations.

3.4.1 Stability and Response to Null Input

Biological neural networks maintain stable baseline activity and respond appropriately to the absence of input. To verify these properties in our models, we examined network dynamics both during prolonged null input presentation and following stimulus offset.

Figure 13 demonstrates that trained models maintain stable activity levels without exhibiting runaway excitation or complete silence, both of which would indicate biologically implausible dynamics. Furthermore, network activity returns to stable baseline levels following stimulus offset, matching the behavior observed in cortical recordings. These properties emerge naturally from the combination of recurrent connections, temporal dynamics, and energy regularization, without requiring explicit normalization operations.

3.4.2 Comparison to Neural Data

In support of our goal of building and evaluating models based on their fidelity to biological neural dynamics, we analyze model dynamics using methods inspired by neural data analysis. Specifically, building off **?**], we test how the models capture temporal normalization characteristic of observed visual responses. This includes sublinear summation of responses to stimuli of increasing duration or contrast, contrast-dependent reaction time, and adaptation to repeated stimuli.

As shown in Figures 14, 15, and 16, the form of recurrence used impacts neural dynamics. In Figure 14, how activity changes as stimulus duration increases is shown for layer V2 of the model. Comparing to ECoG findings from V1 and other cortical regions reported in **?**], all recurrence types show a brain-like rise and plateau of activity in response to increasing duration. However, it appears that recurrent connections implemented as separable convolutions provide the best qualitative match to data.

In Figure 15, dynamics in response to increasing stimulus contrast are shown. Here again, separable convolutions provide the best qualitative match to data by capturing the decrease in time of peak response as a function of increasing contrast.

Finally, in Figure 16, responses of the model to two stimuli with increasing duration between them are shown. Interestingly, here only the model with full recurrence shows the same qualitative pattern seen in the data of increasing peak activity ratio as a function of increasing inter-stimulus interval. The other recurrence types show a decrease with increasing interval.

Groen et al. model the neural characteristics of adaptation, sublinear summation, and contrast-dependent reaction time with a delayed normalization (DN) model (panel C in their Figures 7, 8, 9 respectively). Notably, our model architecture does not include any normalization operation; instead, any normalization effect relies on the recurrent connections. This demonstrates that continuous-time recurrent dynamics can naturally give rise to key temporal phenomena observed in cortex without requiring explicit divisive normalization operations.

3.4.3 Robustness to Noise and Human Behavior

A hallmark of biological vision systems is their robustness to noisy or degraded input. Recurrent connections have been proposed as a mechanism that could enhance robustness by enabling temporal integration and contextual modulation. To test whether our models exhibit this property, and to link model behavior to human psychophysics, we evaluated model performance on images corrupted with varying levels of noise.

Figure 17 demonstrates [TODO: describe key findings about noise robustness, how recurrence helps, and connections to human perception]. This analysis provides a bridge between the biologically realistic temporal dynamics validated against neural data and functionally relevant behavioral outcomes, demonstrating that biological plausibility can enhance model performance on challenging visual tasks.

3.5 Framework Generality: Reference Model Reimplementation

Having systematically explored the DyRCNN parameter space and validated the biological plausibility of the resulting models, we now demonstrate the framework's generality and flexibility. We show that DynVision can precisely recreate established models from the literature, validating the toolbox's correctness while revealing implicit modeling choices that were previously hidden or underspecified in the original implementations.

3.5.1 CORnet-RT: Performance Validation

CORnet-RT [**?**] represents a successful approach to biologically-inspired recurrent vision models, optimized for Brain-Score performance. We reimplemented CORnet-RT within DynVision and trained it on ImageNette (a 10-class subset of ImageNet) to compare against the original implementation.

Our reimplementation achieved identical classification accuracy while delivering a 52% speedup: the original code required 13.51s per epoch while the DynVision implementation required only 8.86s per epoch on an NVIDIA A100 GPU (averaged over the first 80 epochs). Notably, our implementation computes the loss for all 5 timesteps rather than only the last timestep, performs more extensive logging, and applies more comprehensive data augmentations—demonstrating that the performance gains stem from systematic optimization rather than reduced functionality.

This successful reproduction validates the toolbox's correctness and foreshadows the computational efficiency gains detailed in the subsequent benchmarking section.

3.5.2 CordsNet: Revealing Hidden Implementation Details

CordsNet [**?**] implements continuous-time dynamics with an approach philosophically similar to DynVision, making it an important reference point. Reimplementing CordsNet within our framework revealed several implicit design choices in the original implementation:

Initial Idle Period The original CordsNet uses 100 timesteps of idle time with null input before stimulus presentation. Our systematic testing (Figure 10) found this unnecessary for biologically plausible dynamics when using energy loss regularization.

Loss Computation Window CordsNet computes the loss only over the last 70 of 100 timesteps. Our parameter space exploration (Section 3.2) showed that different loss reaction time choices profoundly affect learned dynamics.

Training on ImageNet Note that both CORnet-RT and CordsNet were originally trained on full ImageNet, while our DyRCNN analyses focus on ImageNette for computational efficiency. This dataset difference means direct performance comparisons would not be fair; instead, these reimplementations serve to validate framework correctness and demonstrate generality.

3.5.3 Framework Flexibility

The successful reimplementation of models with different architectural philosophies (CORnet's discrete-time approach vs CordsNet's continuous-time dynamics) demonstrates that DynVision provides a flexible substrate for exploring diverse biological RCNN architectures. The explicit parameterization system makes previously hidden modeling choices visible and systematically explorable, as demonstrated throughout our DyRCNN analyses.

Having validated both the biological plausibility of our models and the generality of the framework, we now turn to comprehensive computational benchmarking.

3.6 Computational Performance Benchmarking

Beyond the 52% speedup demonstrated with CORnet-RT, we provide comprehensive computational performance analysis showing how different modeling choices affect training speed, GPU memory usage, and model complexity. This analysis demonstrates that biological plausibility does not require sacrificing computational efficiency.

[speed and memory for different dataloaders] [speed and memory for shuffle pattern] [speed/memory/params for feedback, skip, loss rt, timesteps, idle, rctype, rctarget, bio/engineering unrolling ...]

4 Discussion

Here we share with the research community DynVision, a toolbox for easy and efficient exploration of RCNN models for the purposes of building biologically-realistic networks. We believe this is a timely contribution as existing RCNN research has turned up mixed results across studies with varying parameter and evaluation settings. We are also building on a budding trend of evolving RCNNs in continuous time using dynamical systems solvers.

While the main goal of this paper is to highlight the features of this toolbox, we also provide results for an example architecture. These results show that qualitative difference in dynamic patterns are dependent on parameter choices such as recurrent architecture, order of layer operations, and training loss time steps.

In future work we plan to conduct more thorough explorations of the recurrent parameter space. We also plan to add support for more bio-inspired features such as separation of excitiatory and inhibitory cells and unsupervised loss functions. We will also continue to increase the computational efficiency of model training and execution, including via exploration of alternative learning algorithms such a truncated backpropagation through time.

[interpretation of time resolution]

[put this somewhere here: In the cortex, inhibitory feedback connections have the role of maintaining computational stability through recurrent normalization. Stability in CNN models is often achieved by applying normalization operations that are less biologically-plausible (e.g. batch normalization). However, it has been shown that recurrent connections alone can also implement divisive normalization that realizes network stability [**?**]. Furthermore, recurrent connections (with heterogenous delays) promise to drive temporal dynamics that can modulate visual behavior over time to make object recognition more versatile and precise (e.g. through attentional gain control) [**?**].]

4.1 Code and Data Availability

5 Acknowledgments

We thank Tanish Vardhineni and Dan Maron for help with beta-testing the toolbox, and Wayne Soo for helpful discussions about training recurrent networks.

6 Supplementary

rcnn_architecture.png

Figure 1: **Architecture schematic of the DyRCNN model family, native to the DynVision toolbox.** The left hand side shows the signal flow and processing within one network layer. The layer operations are shown as circular pictograms or the merging or splitting of arrows. The layer units are represented by their 3D activation tensors. The order of layer operations (vertical labels) can be flexibly arranged as described in 2.3.4. Recurrent activity can be integrated with the 'input' (shown here), 'middle', or 'output' activations of the convolutions. Pointers to specific timestep slots of the hidden state indicate the respective delays (2.2), here $\Delta_{FF} = 2$, $\Delta_{RC} = 2$, $\Delta_{FB} = 2$, $\Delta_{SK} = 4$. The right hand side shows our four-layer network, the layer parameters, and the connections between layers. Note that the pooling operation is only applied in V1 and V2. The tuple parameters indicate the individual values for the two convolutions in the respective layer, when they differ from each other.

Dynamical Timescale Network Activations External Input Bias

$$\tau \frac{dr}{dt} = -r(t) + \Phi[I(t) + Jr(t)] + B \tag{1}$$

Time Dimension Activation Function Network Connectivity Time Step/Delay

$$r_l(t) = r_l(t-dt) + \frac{dt}{\tau}\left(-r_l(t-dt) + \Phi\left[I_l(t-dt) + \sum_{s=0}^{t-dt} J_l r(s)\right] + B_l\right) \tag{2}$$

$$\sum_{s=0}^{t-dt} J_l r(s) = J_{FF} \cdot r_{l-1}(t-\Delta_{FF}) + J_{RC} \cdot r_l(t-\Delta_{RC}) + J_{FB} \cdot r_{>l}(t-\Delta_{FB}) + J_{SK} \cdot r_{<l-1}(t-\Delta_{SK}) \tag{3}$$

Figure 2: **Dynamical systems formulation of the neural network activity with heterogenous connectivity and delays. 1)** Differential equation describing the evolution of network activity over time given the connectivity, and input, with a timescale given by τ. **2)** A numerical stepwise approach to approximately solving the differential equation via the Euler method for the activity r_l in layer l for the subsequent time step t. The contributions by feedforward (FF), lateral recurrent (RC), feedback (FB), and skip (SK) connections are expanded in equation **3)**. Notably, delays Δ need to be integer multiples of dt.

rcnn_unrolling_diagram.png

Figure 3: **Network with recurrent, skip, and feedback connections unrolled in engineering versus biological time.** In biological time (*right side*), the network has feedforward connections with delay $\Delta_{FF} = 2$, recurrent connections with $\Delta_{RC} = 2$, skip connections over two layers with $\Delta_{SK} = 5$, and feedback connections over one layer with $\Delta_{FB} = 2$. Note that each node has equivalent in- and out- put connections, but only a few are shown here for illustration purposes. By unrolling the network instead in engineering time (*left side*), the delays become $\Delta_{FF} = 0$, $\Delta_{RC} = 2$, $\Delta_{SK} = 1$, $\Delta_{FB} = 4$, but the signal flow through the network and time remains identical.

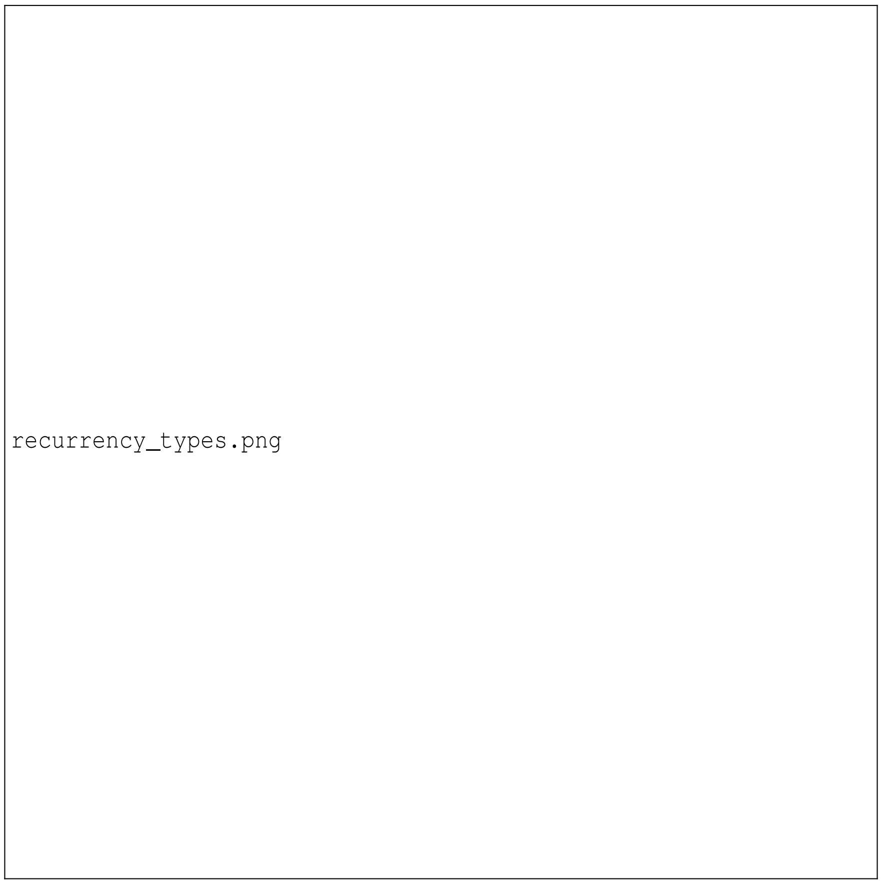

Figure 4: Four types of kernel convolutions used to effectively realize recurrence: self, full, depthwise, pointwise. For each convolution type, one 3D tensor (Height x Width x Channels) representing a model layer output is shown for two time steps Δt_{rc} apart. Each recurrency type uses a specific kind of kernel convolution to add recurrent connections to the layer.

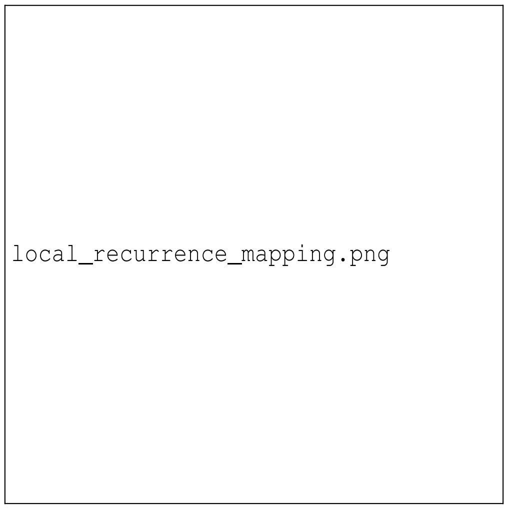

Figure 5: Local recurrence is computed by mapping the 3-D tensor of unit activations onto a 2-D grid so that all features (colors) of one spatial location are arranged close to each other, neighbored by the features of the adjacent spatial locations in a mirrored arrangement. This way, at the borders, the same features at adjacent spatial locations meet and a smooth topographic map can emerge.

rctarget_diagram.png

Figure 6: **Illustration recurrent network connectivity in our model and in cortical columns. A) B)** Taken from **?**].

training_energyloss_pattern=1.png

Figure 7: **Training DyRCNNx8 model with different energy loss weights.** *A)* Training and validation accuracy across 200 epochs for different energy loss weights (0.0, 0.1, 0.2, 0.5, 1.0). *B)* Evolution of cross-entropy and energy loss components during training. *C)* Temporal dynamics of layer activations at epoch 20 (early training), epoch 100 (mid-training), and epoch 200 (late training), showing the evolution from unstable to stable response patterns. *D)* Final test accuracy and mean energy consumption for each weight setting, demonstrating the trade-off between task performance and biological constraint satisfaction.

training_energyloss_pattern=1011.png

Figure 8: **Training DyRCNNx8 model with different energy loss weights.** *A)* Training and validation accuracy across 200 epochs for different energy loss weights (0.0, 0.1, 0.2, 0.5, 1.0). *B)* Evolution of cross-entropy and energy loss components during training. *C)* Temporal dynamics of layer activations at epoch 20 (early training), epoch 100 (mid-training), and epoch 200 (late training), showing the evolution from unstable to stable response patterns. *D)* Final test accuracy and mean energy consumption for each weight setting, demonstrating the trade-off between task performance and biological constraint satisfaction.

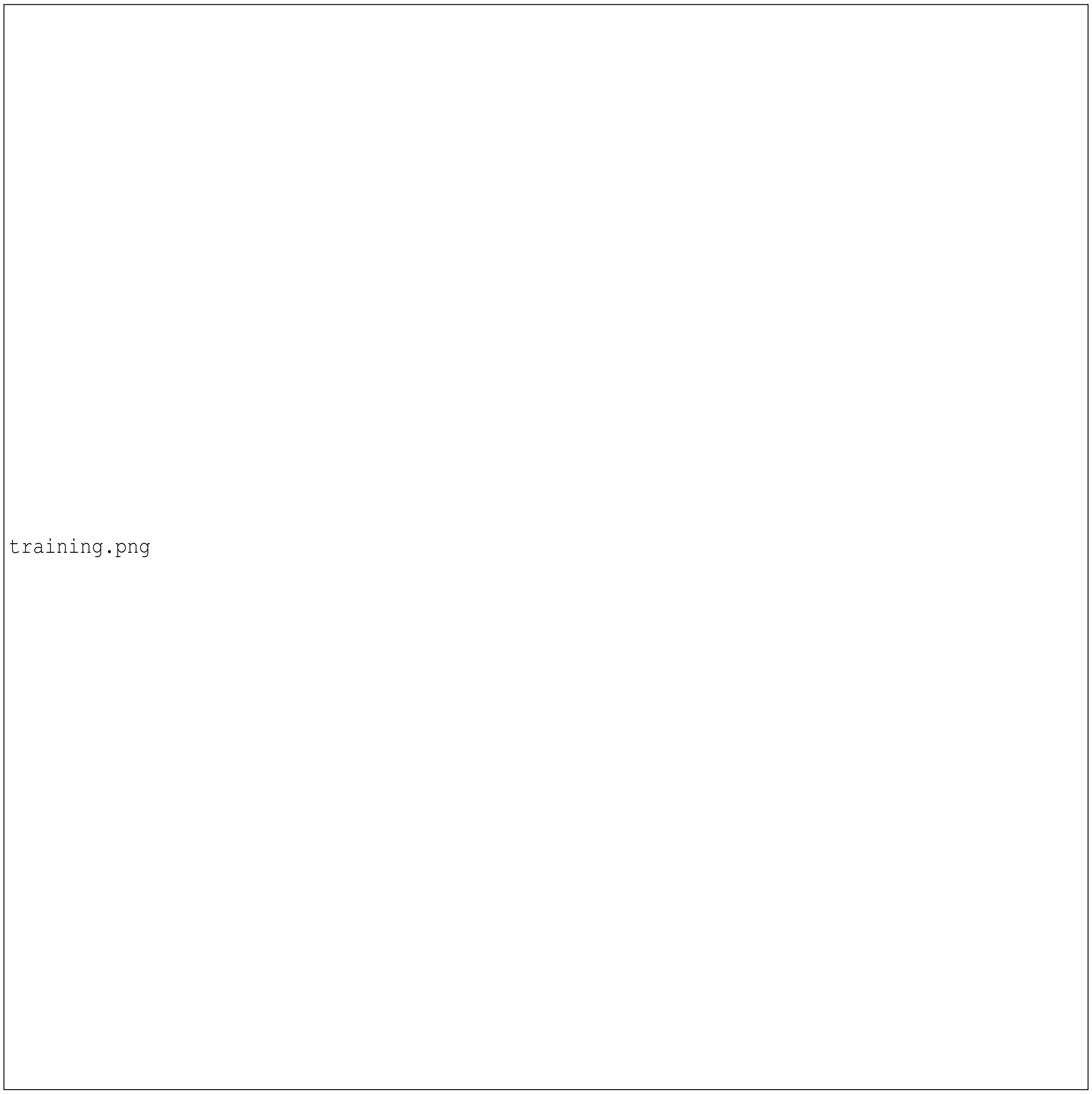

Figure 9: **Training the DyRCNNx8 model with different recurrence types.** All models were trained with 20 timesteps, $dt = 2$, $\tau = 9$, $\Delta_{FF} = 0$, $\Delta_{RC} = 6$, skip connections, and energy loss weight of 0.2 on imagenette for 200 epochs. *A)* Training (solid) and validation (dotted) accuracy for the model versions with different recurrence types. *B)* The model was trained with a combined loss: cross-entropy loss measuring classification performance and energy loss measuring mean unit activations. *C)* Testing the accuracy of the models on 50 timesteps with input presentation for the first 30 timesteps. Dashed lines show the confidence, the softmax value of the target classifier unit. Values are for 256 randomly chosen images. *D)* Comparison of the models' number of parameters, training speed, and performance.

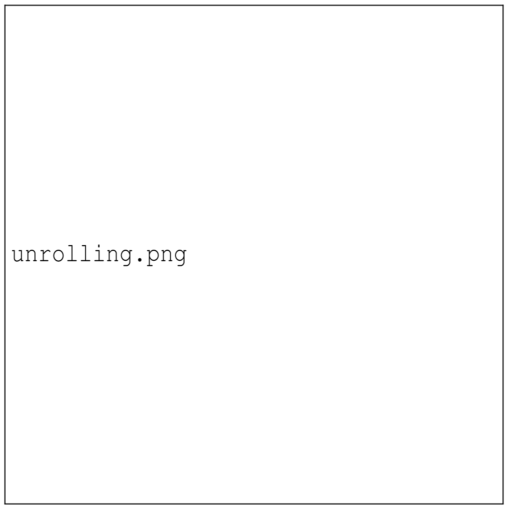

Figure 10: **Equivalence of engineering and biological time unrolling.** This DyRCNNx8 model was trained with full recurrence, skip (V1→V4; V2→IT), and feedback connections (IT→V2; V4→V1) on input with 40 timesteps unrolled in engineering time ($\Delta_{FF} = 0$ ms, $\Delta_{SK} = 0$ ms, $\Delta_{FB} = 34$ ms, $\Delta_{RC} = 6$ ms, $\tau = 9$ ms, $dt = 2$ ms), and then tested in both engineering time and biological time ($\Delta_{FF} = 10$ ms, $\Delta_{SK} = 20$ ms, $\Delta_{FB} = 14$ ms, rest identical) on a single image presented for 35 timesteps followed by 35 timesteps of null input (indicated by the grey step function). Each configuration yields equivalent model responses, validating the mathematical framework and demonstrating computational flexibility.

timesteps_response_tripytch.png

Figure 11: **Effect of initial idle timesteps on training dynamics.** [TODO: Add detailed caption describing comparison of models trained with and without idle periods, showing no benefit for DyRCNN architecture due to energy loss regularization.]

connection_response_tripytch.png

Figure 12: **Impact of recurrence target location on temporal dynamics.** [TODO: Add detailed caption comparing input, middle, and output recurrence integration strategies.]

timeparams_response_tripytch.png

Figure 13: **Effects of temporal parameters on network dynamics.** [TODO: Add detailed caption showing how different values of τ, Δ_{RC}, and Δ_{SK} affect temporal response patterns across layers.]

rctarget_stability_responses.png

Figure 14: **Models exhibit stable dynamics and appropriate null responses.** [TODO: Add detailed caption describing stability metrics, baseline activity levels, and response to null input across different layers and recurrence types.]

duration_rctype_dynamics_V1.png

Figure 15: **Temporal RCNN dynamics for different recurrence types in response to input with varied duration.** Testing input is a static image for 2 to 40 timesteps followed by a null input (zero tensor) indicated by the gray step-function in the subplots. The model (DyRCNNx8) was trained on imagenette for 20 timesteps and is tested on 50 timesteps. *A)* Power (average unit activation) per layer over time. *B)* Layer V2 power for inputs of different duration. *C)* Taken from [**?**], showing experimental findings from human visual cortex ECoG recordings illustrating the neural characteristic of sublinear summation. *D)* An analogous effect can be seen in all our model variants.

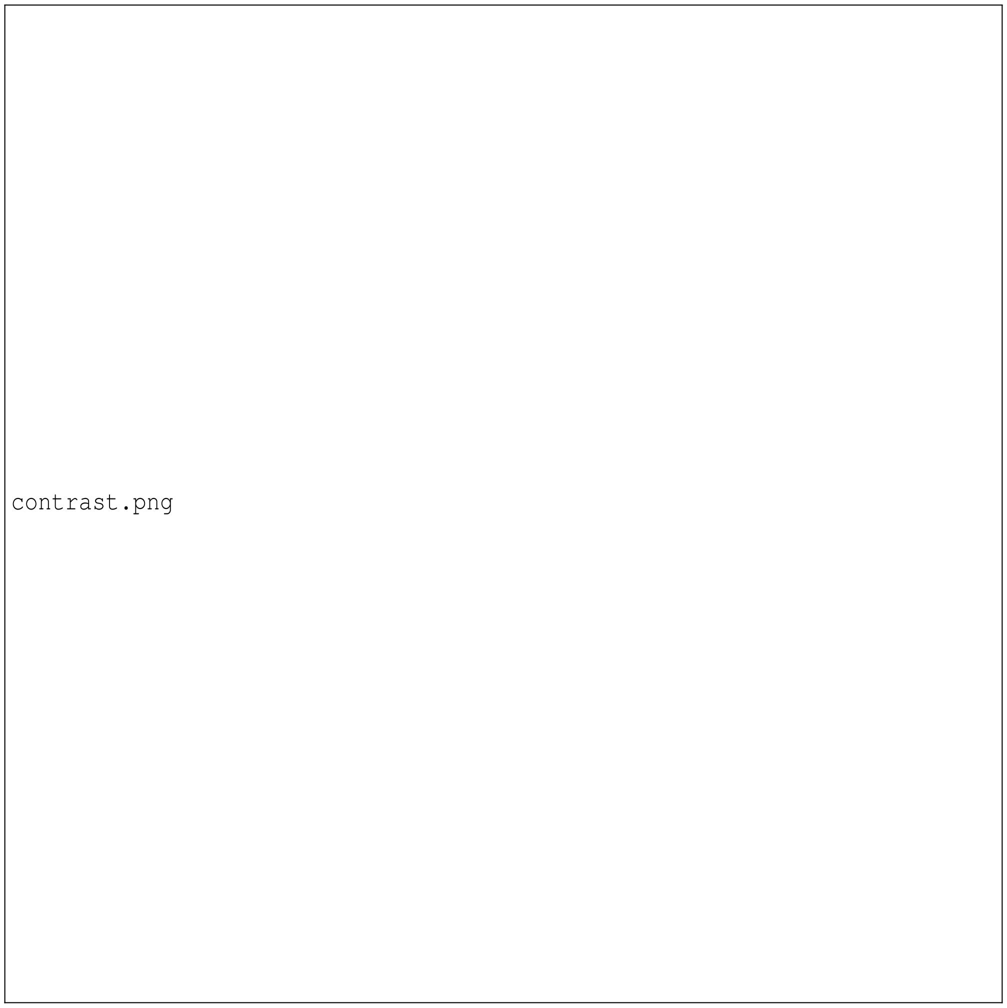

Figure 16: **Temporal RCNN dynamics for different recurrence types in response to input with varied contrast.** Testing input is a static image for 30 timesteps with a contrast factor between 0.2 to 1, followed by a null input (zero tensor) indicated by the gray step-function in the subplots. The model was trained on imagenette for 20 timesteps and is tested on 50 timesteps. *A)* Power (average unit activation) per layer over time. *B)* Layer V2 power for inputs with different contrast. *C)* Taken from [**?**], showing experimental findings from human visual cortex ECoG illustrating the neural characteristic of contrast-dependent reaction time. *D)* This effect is also visible in our models except for the full-recurrency version.

interval_rctype_dynamics_V1.png

Figure 17: **Temporal RCNN dynamics for different recurrence types in response to repeated input with varied delay interval.** Testing input is the same static image shown for 10 timesteps each, intercepted by an interval of null input (zero tensor) varied between 1-16 timesteps. The model was trained on imagenette for 20 timesteps and is tested on 50 timesteps. *A)* Power (average unit activation) per layer over time. *B)* Layer V2 power for inputs with different intervals. *C)* Taken from [**?**], showing experimental findings from human visual cortex ECoG illustrating the characteristic neural adaptation to repeated stimuli. *D)* An analogous effect can only be observed for the full-recurrent model; the other models show a contrary effect.

noiserobustness.png

Figure 18: **Recurrent models show noise robustness linked to human behavior.** [TODO: Add detailed caption describing noise robustness analysis, comparison with feedforward baselines, relationship between recurrence type and noise tolerance, and correlation with human psychophysical data if available.]