PsychRNN

Release 1.0.0-alpha

Daniel B. Ehrlich*, Jasmine T. Stone*, David Brandfonbrener, Alex

Jul 13, 2020

CONTENTS:

This package is intended to help cognitive scientists easily translate task designs from human or primate behavioral experiments into a form capable of being used as training data for a recurrent neural network.

We have isolated the front-end task design, in which users can intuitively describe the conditional logic of their task from the backend where gradient descent based optimization occurs. This is intended to facilitate researchers who might otherwise not have an easy implementation available to design and test hypothesis regarding the behavior of recurrent neural networks in different task environements.

Start with *Hello World* to get a quick sense of what PsychRNN does. Then go through the *Simple Example* to get a feel for how to customize PsychRNN. The rest of *Getting Started* will help guide you through using available features, defining your own task, and even defining your own model.

Release announcments are posted on the psychrnn mailing list and on GitHub.

Code is written and upkept by: Daniel B. Ehrlich, Jasmine T. Stone, David Brandfonbrener, and Alex Atanasov.

Contact: psychrnn@gmail.com

CHAPTER

ONE

INSTALLATION GUIDE

1.1 System requirements

- python = 2.7 or python >= 3.4
- numpy
- tensorflow >= 1.13.1
- For notebook demos, jupyter
- For notebook demos, ipython
- For plotting features, matplotlib

PsychRNN was developed to work with both Python 2.7 and 3.4+ using TensorFlow 1.13.1+. It is currently being tested on Python 2.7 and 3.4-3.8 with TensorFlow 1.13.1-2.2.

Note: TensorFlow 2.2 does not support Python < 3.5. Only TensorFlow 1.13.1-1.14 are compatible with Python 3.4. Python 3.8 is only supported by TensorFlow 2.2.

1.2 Installation

Normally, you can install with:

```
pip install psychrnn=1.0.0-alpha
```

Alternatively, you can download and extract the source files from the GitHub release. Within the downloaded PsychRNN-v1.0.0-alpha folder, run:

```
python setup.py install
```

If you're concerned about clashing dependencies, PsychRNN can be installed in a new `conda` environment:

```
conda create -n psychrnn python=3.6
conda activate psychrnn
pip install psychrnn=1.0.0-alpha
```

[THIS OPTION IS NOT RECOMMENDED FOR MOST USERS] To get the most recent (not necessarily stable) version from the github repo, clone the repository and install:

```
git clone https://github.com/murraylab/PsychRNN.git
cd PsychRNN
python setup.py install
```

1.3 Contributing

Please report bugs to https://github.com/murraylab/psychrnn/issues. This includes any problems with the documentation. Fixes (in the form of pull requests) for bugs are greatly appreciated.

Feature requests are welcome but may or may not be accepted due to limited resources. If you implement the feature yourself we are open to accepting it in PsychRNN. If you implement a new feature in PsychRNN, please do the following before submitting a pull request on GitHub:

- Make sure your code is clean and well commented
- If appropriate, update the official documentation in the `docs/` directory
- Write unit tests and optionally integration tests for your new feature in the `tests/` folder.
- Ensure all existing tests pass (`pytest` returns without error)

For all other questions or comments, contact psychrnn@gmail.com.

CHAPTER

TWO

API DOCUMENTATION

2.1 Backend

2.1.1 Base RNN Object

Classes

RNN(params)	The base recurrent neural network class.

class psychrnn.backend.rnn.**RNN**(*params*)

Bases: abc.ABC

The base recurrent neural network class.

Note: The base RNN class is not itself a functioning RNN. forward_pass must be implemented to define a functioning RNN.

Methods

build()	Build the TensorFlow network and start a TensorFlow session.
destruct()	Close the TensorFlow session and reset the global default graph.
forward_pass()	Run the RNN on a batch of task inputs.
get_effective_W_in()	Get the input weights used in the network, after masking by connectivity and dale_ratio.
get_effective_W_out()	Get the output weights used in the network, after masking by connectivity, and dale_ratio.
get_effective_W_rec()	Get the recurrent weights used in the network, after masking by connectivity and dale_ratio.
get_weights()	Get weights used in the network.
save(save_path)	Save the weights returned by `get_weights()` to `save_path`
test(trial_batch)	Test the network on a certain task input.
train(trial_batch_generator[, train_params])	Train the network.
train_curric(train_params)	Wrapper function for training with curriculum to streamline curriculum learning.

Parameters

- **params** (*dict*) – The RNN parameters. Use your tasks's `get_task_params()` function to start building this dictionary. Optionally use a different network's `get_weights()` function to initialize the network with preexisting weights.
- **Dictionary Keys:**
  - **name** (*str*) – Unique name used to determine variable scope. Having different variable scopes allows multiple distinct models to be instantiated in the same TensorFlow environment. See TensorFlow's variable_scope for more details.
  - **N_in** (*int*) – The number of network inputs.
  - **N_rec** (*int*) – The number of recurrent units in the network.
  - **N_out** (*int*) – The number of network outputs.
  - **N_steps** (*int*): The number of simulation timesteps in a trial.
  - **dt** (*float*) – The simulation timestep.
  - **tau** (*float*) – The intrinsic time constant of neural state decay.
  - **N_batch** (*int*) – The number of trials per training update.
  - **rec_noise** (*float, optional*) – How much recurrent noise to add each time the new state of the network is calculated. Default: 0.0.
  - **transfer_function** (*function, optional*) – Transfer function to use for the network. Default: tf.nn.relu.
  - **load_weights_path** (*str, optional*) – When given a path, loads weights from file in that path. Default: None
  - **initializer** (*`WeightInitializer` or child object, optional*) – Initializer to use for the network. Default: `WeightInitializer` (`params`) if `params` includes `W_rec` or `load_weights_path` as a key, `GaussianSpectralRadius` (`params`) otherwise.
  - **W_in_train** (*bool, optional*) – True if input weights, W_in, are trainable. Default: True
  - **W_rec_train** (*bool, optional*) – True if recurrent weights, W_rec, are trainable. Default: True
  - **W_out_train** (*bool, optional*) – True if output weights, W_out, are trainable. Default: True
  - **b_rec_train** (*bool, optional*) – True if recurrent bias, b_rec, is trainable. Default: True
  - **b_out_train** (*bool, optional*) – True if output bias, b_out, is trainable. Default: True
  - **init_state_train** (*bool, optional*) – True if the inital state for the network, init_state, is trainable. Default: True
  - **loss_function** (*str, optional*) – Which loss function to use. See `psychrnn.backend.loss_functions.LossFunction` for details. Defaults to `"mean_squared_error"`.

  Other Dictionary Keys

  - Any dictionary keys used by the regularizer will be passed onwards to `psychrnn.backend.regularizations.Regularizer`. See `Regularizer` for key names and details.

- Any dictionary keys used for the loss function will be passed onwards to *psychrnn.backend.loss_functions.LossFunction*. See *LossFunction* for key names and details.
- If `initializer` is not set, any dictionary keys used by the initializer will be pased onwards to *WeightInitializer* if `load_weights_path` is set or `W_rec` is passed in. Otherwise all keys will be passed to *GaussianSpectralRadius*
- If `initializer` is not set and `load_weights_path` is not set, the dictionary entries returned previously by *get_weights()* can be passed in to initialize the network. See *WeightInitializer* for a list and explanation of possible parameters. At a minimum, `W_rec` must be included as a key to make use of this option.
- If `initializer` is not set and `load_weights_path` is not set, the following keys can be used to set biological connectivity constraints:
  - **input_connectivity** (*ndarray(dtype=float, shape=(* `N_rec`, `N_in` *)), optional*) – Connectivity mask for the input layer. 1 where connected, 0 where unconnected. Default: np.ones((`N_rec`, `N_in`)).
  - **rec_connectivity** (*ndarray(dtype=float, shape=(* `N_rec`, `N_rec` *)), optional*) – Connectivity mask for the recurrent layer. 1 where connected, 0 where unconnected. Default: np.ones((`N_rec`, `N_rec`)).
  - **output_connectivity** (*ndarray(dtype=float, shape=(* `N_out`, `N_rec` *)), optional*) – Connectivity mask for the output layer. 1 where connected, 0 where unconnected. Default: np.ones((`N_out`, `N_rec`)).
  - **autapses** (*bool, optional*) – If False, self connections are not allowed in N_rec, and diagonal of `rec_connectivity` will be set to 0. Default: True.
  - **dale_ratio** (float, optional) – Dale's ratio, used to construct Dale_rec and Dale_out. 0 <= dale_ratio <=1 if dale_ratio should be used. `dale_ratio * N_rec` recurrent units will be excitatory, the rest will be inhibitory. Default: None

Inferred Parameters:

- **alpha** (*float*) – The number of unit time constants per simulation timestep.

build()
Build the TensorFlow network and start a TensorFlow session.

destruct()
Close the TensorFlow session and reset the global default graph.

abstract forward_pass()
Run the RNN on a batch of task inputs.

Note: This is an abstract function that must be defined in a child class.

Returns

- **predictions** (*ndarray(dtype=float, shape=(*`N_batch`, `N_steps`, `N_out` *))*) – Network output on inputs found in self.x within the tf network.

- **states** (*ndarray(dtype=float, shape=(*`N_batch`, `N_steps`, `N_rec` *))*) – State variable values over the course of the trials found in self.x within the tf network.

Return type tuple

get_effective_W_in()
Get the input weights used in the network, after masking by connectivity and dale_ratio.

Returns tf.Tensor(dtype=float, shape=(`N_rec`, `N_in`))

get_effective_W_out()
Get the output weights used in the network, after masking by connectivity, and dale_ratio.

Returns tf.Tensor(dtype=float, shape=(`N_out`, `N_rec`))

get_effective_W_rec()
Get the recurrent weights used in the network, after masking by connectivity and dale_ratio.

Returns tf.Tensor(dtype=float, shape=(`N_rec`, `N_rec`))

get_weights()
Get weights used in the network.

Allows for rebuilding or tweaking different weights to do experiments / analyses.

Returns

Dictionary of rnn weights including the following keys:

Dictionary Keys

- **init_state** (*ndarray(dtype=float, shape=(1, :attr:`N_rec` *))*) – Initial state of the network's recurrent units.
- **W_in** (*ndarray(dtype=float, shape=(:attr:`N_rec`. :attr:`N_in` *))*) – Input weights.
- **W_rec** (*ndarray(dtype=float, shape=(:attr:`N_rec`, :attr:`N_rec` *))*) – Recurrent weights.
- **W_out** (*ndarray(dtype=float, shape=(:attr:`N_out`, :attr:`N_rec` *))*) – Output weights.
- **b_rec** (*ndarray(dtype=float, shape=(:attr:`N_rec`, *))*) – Recurrent bias.
- **b_out** (*ndarray(dtype=float, shape=(:attr:`N_out`, *))*) – Output bias.
- **Dale_rec** (*ndarray(dtype=float, shape=(:attr:`N_rec`, :attr:`N_rec`))*) – Diagonal matrix with ones and negative ones on the diagonal. If `dale_ratio` is not `None`, indicates whether a recurrent unit is excitatory(1) or inhibitory(-1).
- **Dale_out** (*ndarray(dtype=float, shape=(:attr:`N_rec`, :attr:`N_rec`))*) – Diagonal matrix with ones and zeroes on the diagonal. If `dale_ratio` is not `None`, indicates whether a recurrent unit is excitatory(1) or inhibitory(0). Inhibitory neurons do not contribute to the output.
- **input_connectivity** (*ndarray(dtype=float, shape=(:attr:`N_rec`, :attr:`N_in`))*) – Connectivity mask for the input layer. 1 where connected, 0 where unconnected.

- **rec_connectivity** (*ndarray(dtype=float, shape=(:attr:`N_rec`, :attr:`N_rec`))**) – Connectivity mask for the recurrent layer. 1 where connected, 0 where unconnected.
- **output_connectivity** (*ndarray(dtype=float, shape=(:attr:`N_out`, :attr:`N_rec`))**) – Connectivity mask for the output layer. 1 where connected, 0 where unconnected.
- **dale_ratio** (*float*) – Dale's ratio, used to construct Dale_rec and Dale_out. Either `None` if dale's law was not applied, or 0 <= dale_ratio <=1 if dale_ratio was applied.

Note: Keys returned may be different / include other keys depending on the implementation of *RNN* used. A different set of keys will be included e.g. if the *LSTM* implementation is used. The set of keys above is accurate and meaningful for the *Basic* and *BasicScan* implementations.

Return type dict

save(*save_path*)

Save the weights returned by *get_weights()* to `save_path`

Parameters **save_path** (*str*) – Path for where to save the network weights.

test(*trial_batch*)

Test the network on a certain task input.

Parameters **trial_batch** ((*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Task stimulus to run the network on. Stimulus from *psychrnn.tasks.task.Task.get_trial_batch()*, or from next(*psychrnn.tasks.task.Task.batch_generator()*).

Returns

- **outputs** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Output time series of the network for each trial in the batch.
- **states** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_rec` *))*) – Activity of recurrent units during each trial.

Return type tuple

train(*trial_batch_generator*, *train_params={}*)

Train the network.

Parameters

- **trial_batch_generator** (*Task* object or *Generator[tuple, None, None]*) – the task to train on, or the task to train on's batch_generator. If a task is passed in, task.:func:*batch_generator* () will be called to get the generator for the task to train on.
- **train_params** (*dict, optional*) – Dictionary of training parameters containing the following possible keys:

  Dictionary Keys

  - **learning_rate** (*float, optional*) – Sets learning rate if use default optimizer Default: .001

- **training_iters** (*int, optional*) – Number of iterations to train for Default: 50000.
- **loss_epoch** (*int, optional*) – Compute and record loss every 'loss_epoch' epochs. Default: 10.
- **verbosity** (*bool, optional*) – If true, prints information as training progresses. Default: True.
- **save_weights_path** (*str, optional*) – Where to save the model after training. Default: None
- **save_training_weights_epoch** (*int, optional*) – Save training weights every 'save_training_weights_epoch' epochs. Weights only actually saved if `training_weights_path` is set. Default: 100.
- **training_weights_path** (*str, optional*) – What directory to save training weights into as training progresses. Default: None.
- **curriculum** (*~psychrnn.backend.curriculum.Curriculum object, optional*) – Curriculum to train on. If a curriculum object is provided, it overrides the trial_batch_generator argument. Default: None.
- **optimizer** (tf.compat.v1.train.Optimizer *object, optional*) – What optimizer to use to compute gradients. Default: tf.train.AdamOptimizer (learning_rate=:data:*train_params`['learning_rate']*).
- **clip_grads** (*bool, optional*) – If true, clip gradients by norm 1. Default: True
- **fixed_weights** (*dict, optional*) – By default all weights are allowed to train unless `fixed_weights` or `W_rec_train`, `W_in_train`, or `W_out_train` are set. Default: None. Dictionary of weights to fix (not allow to train) with the following optional keys:

  Fixed Weights Dictionary Keys (in case of *Basic* and *BasicScan* implementations)

  - **W_in** (*ndarray(dtype=bool, shape=(:attr:`N_rec`, :attr:`N_in` *)), optional*) – True for input weights that should be fixed during training.
  - **W_rec** (*ndarray(dtype=bool, shape=(:attr:`N_rec`, :attr:`N_rec` *)), optional*) – True for recurrent weights that should be fixed during training.
  - **W_out** (*ndarray(dtype=bool, shape=(:attr:`N_out`, :attr:`N_rec` *)), optional*) – True for output weights that should be fixed during training.

  Note In general, any key in the dictionary output by *get_weights()* can have a key in the fixed_weights matrix, however fixed_weights will only meaningfully apply to trainable matrices.

- **performance_cutoff** (*float*) – If `performance_measure` is not `None`, training stops as soon as performance_measure surpases the performance_cutoff. Default: None.
- **performance_measure** (*function*) – Function to calculate the performance of the network using custom criteria. Default: None.

Arguments

* **trial_batch** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Task stimuli for `N_batch` trials.
* **trial_y** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Target output for the network on `N_batch` trials given the `trial_batch`.
* **output_mask** (*ndarray(dtype=bool, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Output mask for `N_batch` trials. True when the network should aim to match the target output, False when the target output can be ignored.
* **output** (*ndarray(dtype=bool, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Output to compute the accuracy of. `output` as returned by *psychrnn.backend.rnn.RNN.test()*.
* **epoch** (*int*): Current training epoch (e.g. perhaps the performance_measure is calculated differently early on vs late in training)
* **losses** (*list of float*): List of losses from the beginning of training until the current epoch.
* **verbosity** (*bool*): Passed in from `train_params`.

Returns *float*

Performance, greater when the performance is better.

Returns

- **losses** (*list of float*) – List of losses, computed every `loss_epoch` epochs during training.
- **training_time** (*float*) – Time spent training.
- **initialization_time** (*float*) – Time spent initializing the network and preparing to train.

Return type tuple

train_curric(*train_params*)

Wrapper function for training with curriculum to streamline curriculum learning.

Parameters **train_params** (*dict, optional*) – See *train()* for details.

Returns See *train()* for details.

Return type tuple

2.1.2 Implemented RNN Models

Basic (Vanilla) RNNs

Classes

Basic(params)	The basic continuous time recurrent neural network model.
BasicScan(params)	The basic continuous time recurrent neural network model implemented with tf.scan .

class psychrnn.backend.models.basic.**Basic**(*params*)

Bases: *psychrnn.backend.rnn.RNN*

The basic continuous time recurrent neural network model.

Basic implementation of *psychrnn.backend.rnn.RNN* with a simple RNN, enabling biological constraints.

Parameters **params** (*dict*) – See *psychrnn.backend.rnn.RNN* for details.

Methods

forward_pass()	Run the RNN on a batch of task inputs.
output_timestep(state)	Returns the output node activity for a given timestep.
recurrent_timestep(rnn_in, state)	Recurrent time step.

forward_pass()

Run the RNN on a batch of task inputs.

Iterates over timesteps, running the *recurrent_timestep()* and *output_timestep()*

Implements *psychrnn.backend.rnn.RNN.forward_pass()*.

Returns

- **predictions** (*tf.Tensor(*N_batch, N_steps, N_out *)*) – Network output on inputs found in self.x within the tf network.
- **states** (*tf.Tensor(*N_batch, N_steps, N_rec *)*) – State variable values over the course of the trials found in self.x within the tf network.

Return type tuple

output_timestep(*state*)

Returns the output node activity for a given timestep.

Parameters **state** (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*) – State of network at a given timepoint for each trial in the batch.

Returns Output of the network at a given timepoint for each trial in the batch.

Return type output (*tf.Tensor(dtype=float, shape=(* N_batch , N_out *)*)

recurrent_timestep(*rnn_in*, *state*)

Recurrent time step.

Given input and previous state, outputs the next state of the network.

Parameters

- **rnn_in** (*tf.Tensor(dtype=float, shape=(?,* N_in *))*) – Input to the rnn at a certain time point.
- **state** (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*) – State of network at previous time point.

Returns New state of the network.

Return type new_state (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*)

class psychrnn.backend.models.basic.**BasicScan**(*params*)

Bases: *psychrnn.backend.models.basic.Basic*

The basic continuous time recurrent neural network model implemented with tf.scan .

Produces the same results as *Basic*, with possible differences in execution time.

Parameters **params** (*dict*) – See *psychrnn.backend.rnn.RNN* for details.

Methods

forward_pass()	Run the RNN on a batch of task inputs.
output_timestep(dummy, state)	Wrapper function for *psychrnn.backend.models.basic.Basic.output_timestep()*.
recurrent_timestep(state, rnn_in)	Wrapper function for *psychrnn.backend.models.basic.Basic.recurrent_timestep()*.

forward_pass()

Run the RNN on a batch of task inputs.

Iterates over timesteps, running the *recurrent_timestep()* and *output_timestep()*

Implements *psychrnn.backend.rnn.RNN.forward_pass()*.

Returns

- **predictions** (*tf.Tensor(*N_batch, N_steps, N_out *)*) – Network output on inputs found in self.x within the tf network.
- **states** (*tf.Tensor(*N_batch, N_steps, N_rec *)*) – State variable values over the course of the trials found in self.x within the tf network.

Return type tuple

output_timestep(*dummy*, *state*)

Wrapper function for *psychrnn.backend.models.basic.Basic.output_timestep()*.

Includes additional dummy argument to facilitate tf.scan.

Parameters

- **dummy** – Dummy variable provided by tf.scan. Not actually used by the function.
- **state** (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*) – State of network at a given timepoint for each trial in the batch.

Returns Output of the network at a given timepoint for each trial in the batch.

Return type output (*tf.Tensor(dtype=float, shape=(* N_batch , N_out *))*)

recurrent_timestep(*state*, *rnn_in*)

Wrapper function for *psychrnn.backend.models.basic.Basic.recurrent_timestep()*.

Parameters

- **state** (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*) – State of network at previous time point.
- **rnn_in** (*tf.Tensor(dtype=float, shape=(?,* N_in *))*) – Input to the rnn at a certain time point.

Returns New state of the network.

Return type new_state (*tf.Tensor(dtype=float, shape=(* N_batch , N_rec *))*)

LSTM

Classes

LSTM(params)	LSTM (Long Short Term Memory) recurrent network model

class psychrnn.backend.models.lstm.**LSTM**(*params*)

Bases: *psychrnn.backend.rnn.RNN*

LSTM (Long Short Term Memory) recurrent network model

LSTM implementation of *psychrnn.backend.rnn.RNN*. Because LSTM is structured differently from the basic RNN, biological constraints such as dale's, autapses, and connectivity are not enabled.

Parameters **params** (*dict*) – See *psychrnn.backend.rnn.RNN* for details.

Methods

forward_pass()	Run the LSTM on a batch of task inputs.
output_timestep(hidden)	Returns the output node activity for a given timestep.
recurrent_timestep(rnn_in, hidden, cell)	Recurrent time step.

forward_pass()

Run the LSTM on a batch of task inputs.

Iterates over timesteps, running the *recurrent_timestep()* and *output_timestep()*

Implements *psychrnn.backend.rnn.RNN.forward_pass()*.

Returns

- **predictions** (*tf.Tensor(*`N_batch`, `N_steps`, `N_out` *)*) – Network output on inputs found in self.x within the tf network.
- **hidden** (*tf.Tensor(*`N_batch`, `N_steps`, `N_rec` *)*) – Hidden unit values over the course of the trials found in self.x within the tf network.

Return type tuple

output_timestep(*hidden*)

Returns the output node activity for a given timestep.

Parameters **hidden** (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_rec` *)*) – Hidden units of network at a given timepoint for each trial in the batch.

Returns Output of the network at a given timepoint for each trial in the batch.

Return type output (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_out` *)*)

recurrent_timestep(*rnn_in*, *hidden*, *cell*)

Recurrent time step.

Given input and previous state, outputs the next state of the network.

Parameters

- **rnn_in** (*tf.Tensor(dtype=float, shape=(?,* `N_in` *)*) – Input to the rnn at a certain time point.
- **hidden** (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_rec` *)*) – Hidden units state of network at previous time point.
- **cell** (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_rec` *)*) – Cell state of the network at previous time point.

Returns

- **new_hidden** (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_rec` *)*) – New hidden unit state of the network.

- **new_cell** (*tf.Tensor(dtype=float, shape=(* `N_batch` , `N_rec` *))*) – New cell state of the network.

Return type tuple

2.1.3 Backend Modules

Initializations

Classes

AlphaIdentity(**kwargs)	Generate recurrent weights $w(i,i) = alpha$, $w(i,j) = 0$ where $i \neq j$.
GaussianSpectralRadius(**kwargs)	Generate random gaussian weights with specified spectral radius.
WeightInitializer(**kwargs)	Base Weight Initialization class.

class psychrnn.backend.initializations.**AlphaIdentity**(***kwargs*)

Bases: *psychrnn.backend.initializations.WeightInitializer*

Generate recurrent weights $w(i,i) = alpha$, $w(i,j) = 0$ where $i \neq j$.

If Dale is set, balances the alpha excitatory and inhibitory weights using *balance_dale_ratio()*, so w(i,i) will not be exactly equal to alpha.

Keyword Arguments **alpha** (*float*) – The value of alpha to set w(i,i) to in W_rec.

Other Keyword Args: See *WeightInitializer* for details.

class psychrnn.backend.initializations.**GaussianSpectralRadius**(***kwargs*)

Bases: *psychrnn.backend.initializations.WeightInitializer*

Generate random gaussian weights with specified spectral radius.

If Dale is set, balances the random gaussian weights between excitatory and inhibitory using `balance_dale_ratio()` before applying the specified spectral radius.

Keyword Arguments **spec_rad** (*float, optional*) – The spectral radius to initialize W_rec with. Default: 1.1.

Other Keyword Args: See *WeightInitializer* for details.

class psychrnn.backend.initializations.**WeightInitializer**(***kwargs*)

Bases: `object`

Base Weight Initialization class.

Initializes biological constraints and network weights, optionally loading weights from a file or from passed in arrays.

Keyword Arguments

- **N_in** (*int*) – The number of network inputs.
- **N_rec** (*int*) – The number of recurrent units in the network.
- **N_out** (*int*) – The number of network outputs.
- **load_weights_path** (*str, optional*) – Path to load weights from using np.load. Weights saved at that path should be in the form saved out by *psychrnn.backend.rnn.RNN.save()* Default: None.
- **input_connectivity** (ndarray(dtype=float, shape=(`N_rec`, `N_in`)), optional) – Connectivity mask for the input layer. 1 where connected, 0 where unconnected. Default: np.ones((`N_rec`, `N_in`)).

- **rec_connectivity** (ndarray(dtype=float, shape=(N_rec, N_rec)), optional) – Connectivity mask for the recurrent layer. 1 where connected, 0 where unconnected. Default: np.ones((N_rec, N_rec)).
- **output_connectivity** (ndarray(dtype=float, shape=(N_out, N_rec)), optional) – Connectivity mask for the output layer. 1 where connected, 0 where unconnected. Default: np.ones((N_out, N_rec)).
- **autapses** (*bool, optional*) – If False, self connections are not allowed in N_rec, and diagonal of rec_connectivity will be set to 0. Default: True.
- **dale_ratio** (*float, optional*) – Dale's ratio, used to construct Dale_rec and Dale_out. 0 <= dale_ratio <=1 if dale_ratio should be used. dale_ratio * N_rec recurrent units will be excitatory, the rest will be inhibitory. Default: None
- **which_rand_init** (*str, optional*) – Which random initialization to use for W_in and W_out. Will also be used for W_rec if which_rand_W_rec_init is not passed in. Options: *'const_unif'*, *'const_gauss'*, *'glorot_unif'*, *'glorot_gauss'*. Default: *'glorot_gauss'*.
- **which_rand_W_rec_init** (*str, optional*) – Which random initialization to use for W_rec. Options: *'const_unif'*, *'const_gauss'*, *'glorot_unif'*, *'glorot_gauss'*. Default: which_rand_init.
- **init_minval** (*float, optional*) – Used by *const_unif_init()* as minval if 'const_unif' is passed in for which_rand_init or which_rand_W_rec_init. Default: -.1.
- **init_maxval** (*float, optional*) – Used by *const_unif_init()* as maxval if 'const_unif' is passed in for which_rand_init or which_rand_W_rec_init. Default: .1.
- **W_in** (ndarray(dtype=float, shape=(N_rec, N_in)), optional) – Input weights. Default: Initialized using the function indicated by which_rand_init
- **W_rec** (ndarray(dtype=float, shape=(N_rec, N_rec)), optional) – Recurrent weights. Default: Initialized using the function indicated by which_rand_W_rec_init
- **W_out** (ndarray(dtype=float, shape=(N_out, N_rec)), optional) – Output weights. Defualt: Initialized using the function indicated by which_rand_init
- **b_rec** (ndarray(dtype=float, shape=(N_rec,)), optional) – Recurrent bias. Default: np.zeros(N_rec)
- **b_out** (ndarray(dtype=float, shape=(N_out,)), optional) – Output bias. Default: np.zeros(N_out)
- **Dale_rec** (ndarray(dtype=float, shape=(N_rec, N_rec)), optional) – Diagonal matrix with ones and negative ones on the diagonal. If dale_ratio is not None, indicates whether a recurrent unit is excitatory(1) or inhibitory(-1). Default: constructed based on dale_ratio
- **Dale_out** (ndarray(dtype=float, shape=(N_rec, N_rec)), optional) – Diagonal matrix with ones and zeroes on the diagonal. If dale_ratio is not None, indicates whether a recurrent unit is excitatory(1) or inhibitory(0). Inhibitory neurons do not contribute to the output. Default: constructed based on dale_ratio
- **init_state** (ndarray(dtype=float, shape=(1, N_rec)), optional) – Initial state of the network's recurrent units. Default: .1 + .01 * np.random.randn(N_rec).

Methods

balance_dale_ratio()	If dale_ratio is not None, balances initializations['W_rec'] 's excitatory and inhibitory weights so the network will train.
const_gauss_init(connectivity)	Initialize ndarray of shape connectivity with values from a normal distribution.

continues on next page

Table 9 – continued from previous page

const_unif_init(connectivity)	Initialize ndarray of shape `connectivity` with values uniform distribution with minimum `init_minval` and maximum `init_maxval` as set in *WeightInitializer*.
get(tensor_name)	Get `tensor_name` from `initializations` as a Tensor.
get_dale_ratio()	Returns the dale_ratio.
get_rand_init_func(which_rand_init)	Maps initialization function names (strings) to generating functions.
glorot_gauss_init(connectivity)	Initialize ndarray of shape `connectivity` with values from a glorot normal distribution.
glorot_unif_init(connectivity)	Initialize ndarray of shape `connectivity` with values from a glorot uniform distribution.
save(save_path)	Save `initializations` to `save_path`.

initializations
Dictionary containing entries for `input_connectivity`, `rec_connectivity`, `output_connectivity`, `dale_ratio`, `Dale_rec`, `Dale_out`, `W_in`, `W_rec`, `W_out`, `b_rec`, `b_out`, and `init_state`.
Type dict

balance_dale_ratio()
If dale_ratio is not None, balances *initializations['W_rec']* 's excitatory and inhibitory weights so the network will train.

const_gauss_init(*connectivity*)
Initialize ndarray of shape `connectivity` with values from a normal distribution.
Parameters **connectivity** (*ndarray*) – 1 where connected, 0 where unconnected.
Returns ndarray(dtype=float, shape=connectivity.shape)

const_unif_init(*connectivity*)
Initialize ndarray of shape `connectivity` with values uniform distribution with minimum `init_minval` and maximum `init_maxval` as set in *WeightInitializer*.
Parameters **connectivity** (*ndarray*) – 1 where connected, 0 where unconnected.
Returns ndarray(dtype=float, shape=connectivity.shape)

get(*tensor_name*)
Get `tensor_name` from *initializations* as a Tensor.
Parameters **tensor_name** (*str*) – The name of the tensor to get. See *initializations* for options.
Returns Tensor object

get_dale_ratio()
Returns the dale_ratio.

$0 \leq dale\_ratio \leq 1$ if dale_ratio should be used, dale_ratio = None otherwise. `dale_ratio * N_rec` recurrent units will be excitatory, the rest will be inhibitory.
Returns Dale ratio, None if no dale ratio is set.
Return type float

get_rand_init_func(*which_rand_init*)
Maps initialization function names (strings) to generating functions.
Parameters **which_rand_init** (*str*) – Maps to `[which_rand_init]_init`. Options are *'const_unif'*, *'const_gauss'*, *'glorot_unif'*, *'glorot_gauss'*.
Returns `self.[which_rand_init]_init`
Return type function

glorot_gauss_init(*connectivity*)
Initialize ndarray of shape `connectivity` with values from a glorot normal distribution.

Draws samples from a normal distribution centered on 0 with *stddev = sqrt(2 / (fan_in + fan_out))* where *fan_in* is the number of input units and *fan_out* is the number of output units. Respects the `connectivity` matrix.

Parameters **connectivity** (*ndarray*) – 1 where connected, 0 where unconnected.
Returns ndarray(dtype=float, shape=connectivity.shape)

glorot_unif_init(*connectivity*)
Initialize ndarray of shape `connectivity` with values from a glorot uniform distribution.

Draws samples from a uniform distribution within [-limit, limit] where *limit* is *sqrt(6 / (fan_in + fan_out))* where *fan_in* is the number of input units and *fan_out* is the number of output units. Respects the `connectivity` matrix.

Parameters **connectivity** (*ndarray*) – 1 where connected, 0 where unconnected.
Returns ndarray(dtype=float, shape=connectivity.shape)

save(*save_path*)
Save *initializations* to `save_path`.

Parameters **save_path** (*str*) – File path for saving the initializations. The .npz extension will be appended if not already provided.

Loss Functions

Classes

LossFunction(params)	Set the loss function for the *RNN* model.

class psychrnn.backend.loss_functions.**LossFunction**(*params*)
Bases: `object`

Set the loss function for the *RNN* model.

Parameters **params** (*dict*) – Dictionary of parameters including the following keys:

Dictionary Keys

- **loss_function** (*str*) – String indicating what loss function to use. If `params["loss_function"]` is not *mean_squared_error* or *binary_cross_entropy*, `params[params["loss_function"]]` defines the custom loss function. Default: "mean_squared_error".
- **params["loss_function"]** (*function, optional*) – Defines the custom loss function. Must have the same signature as *mean_squared_error()* and *binary_cross_entropy()*.

Methods

binary_cross_entropy(predictions, y, output_mask)	Binary cross-entropy.
mean_squared_error(predictions, y, output_mask)	Mean squared error.
set_model_loss(model)	Returns the model loss, calculated as indicated by `type` (inferred from `params["loss_function"]`.

binary_cross_entropy(*predictions*, *y*, *output_mask*)
Binary cross-entropy.

Binary label values are assumed to be 0 and 1.

```
loss = mean(output_mask * -(y * log(predictions) + (1-y)*
log(1-predictions)))
```

Parameters
- **predictions** (*tf.Tensor(dtype=float, shape =(*N_batch, N_steps, N_out *))*) – Network output.
- **y** (*tf.Tensor(dtype=float, shape =(*?, N_steps, N_out *))*) – Target output.
- **output_mask** (*tf.Tensor(dtype=float, shape =(*?, N_steps, N_out *))*) – Output mask for N_batch trials. True when the network should aim to match the target output, False when the target output can be ignored.

Returns Binary cross-entropy.
Return type tf.Tensor(dtype=float)

mean_squared_error(*predictions*, *y*, *output_mask*)
Mean squared error.

```
loss = mean(square(output_mask * (predictions - y)))
```

Parameters
- **predictions** (*tf.Tensor(dtype=float, shape =(*N_batch, N_steps, N_out *))*) – Network output.
- **y** (*tf.Tensor(dtype=float, shape =(*?, N_steps, N_out *))*) – Target output.
- **output_mask** (*tf.Tensor(dtype=float, shape =(*?, N_steps, N_out *))*) – Output mask for N_batch trials. True when the network should aim to match the target output, False when the target output can be ignored.

Returns Mean squared error.
Return type tf.Tensor(dtype=float)

set_model_loss(*model*)
Returns the model loss, calculated as indicated by type (inferred from params["loss_function"].

'mean_squared_error' indicates *mean_squared_error()*, 'binary_cross_entropy' indicates *binary_cross_entropy()*. If type is not one of the above options, custom_loss_function is used. The custom loss function would have been passed in to params as params[type].

Parameters **model** (*RNN* object) – Model for which to calculate the regularization.
Returns Model loss.
Return type tf.Tensor(dtype=float)

Regularizations

Classes

Regularizer(params)	Regularizer Class

class psychrnn.backend.regularizations.**Regularizer**(*params*)
Bases: object

Regularizer Class

Class that aggregates all types of regularization used.

Parameters **params** (*dict*) – The regularization parameters containing the following optional keys:

Dictionary Keys

- **L1_in** (*float, optional*) – Parameter for weighting the L1 input weights regularization. Default: 0.
- **L1_rec** (*float, optional*) – Parameter for weighting the L1 recurrent weights regularization. Default: 0.
- **L1_out** (*float, optional*) – Parameter for weighting the L1 output weights regularization. Default: 0.
- **L2_in** (*float, optional*) – Parameter for weighting the L2 input weights regularization. Default: 0.
- **L2_rec** (*float, optional*) – Parameter for weighting the L2 recurrent weights regularization. Default: 0.
- **L2_out** (*float, optional*) – Parameter for weighting the L2 output weights regularization. Default: 0.
- **L2_firing_rate** (*float, optional*) – Parameter for weighting the L2 regularization of the relu thresholded states. Default: 0.
- **custom_regularization** (*function, optional*) – Custom regularization function. Default: None.

Args:

- **model** (*RNN object*) – Model for which to calculate the regularization.
- **params** (*dict*) – Regularization parameters. All params passed to the *Regularizer* will be passed here.

Returns: tf.Tensor(dtype=float)– The custom regularization to add when calculating the loss.

Methods

L1_weight_reg(model)	L1 regularization
L2_firing_rate_reg(model)	L2 regularization of the firing rate.
L2_weight_reg(model)	L2 regularization
set_model_regularization(model)	Given model, calculate the regularization by adding all regualarization terms (scaled with the parameters to be either zero or nonzero).

L1_weight_reg(*model*)

L1 regularization

$regularization = L1\_in * ||W\_in||_1 + L1\_rec * ||W\_rec||_1 + L1\_out * ||W\_out||_1$

Parameters **model** (*RNN* object) – Model for which to calculate the regularization.
Returns The L1 regularization to add when calculating the loss.
Return type tf.Tensor(dtype=float)

L2_firing_rate_reg(*model*)

L2 regularization of the firing rate.

$regularization = L2\_firing\_rate * ||relu(states)||_2^2$

Parameters **model** (*RNN* object) – Model for which to calculate the regularization.
Returns The L2 firing rate regularization to add when calculating the loss.
Return type tf.Tensor(dtype=float)

L2_weight_reg(*model*)

L2 regularization

$regularization = L2\_in * ||W\_in||_2^2 + L2\_rec * ||W\_rec||_2^2 + L2\_out * ||W\_out||_2^2$

Parameters **model** (*RNN* object) – Model for which to calculate the regularization.

Returns The L2 regularization to add when calculating the loss.
Return type tf.Tensor(dtype=float)

set_model_regularization(*model*)
Given model, calculate the regularization by adding all regualarization terms (scaled with the parameters to be either zero or nonzero).

The following regularizations are added: *L1_weight_reg()*, *L2_weight_reg()*, and *L2_firing_rate_reg()*.
Parameters **model** (*RNN* object) – Model for which to calculate the regularization.
Returns The regularization to add when calculating the loss.
Return type tf.Tensor(dtype=float)

Curriculum

Classes

Curriculum(tasks, **kwargs)	Curriculum object.

Functions

default_metric(curriculum_params, ...)	Default metric to use to evaluate performance when using Curriculum learning.

class psychrnn.backend.curriculum.**Curriculum**(*tasks*, ***kwargs*)
Bases: object

Curriculum object.

Allows training on a sequence of tasks when Curriculum is passed into *train()*.
Parameters
- **tasks** (list of *Task* objects) – List of tasks to use in the curriculum.
- **metric** (*function, optional*) – Function for calculating whether the stage advances and what the metric value is at each metric_epoch. Default: *default_metric()*.
Arguments
– **curriculum_params** (*dict*) – Dictionary of the *Curriculum* object parameters, containing the following keys:
Dictionary Keys
– **stop_training** (*bool*) – True if the network has finished training and completed all stages.
– **stage** (*int*) – Current training stage (initial stage is 0).
– **metric_values** (*list of [float, int]*) – List of metric values and the stage at which each metric value was computed.
– **tasks** (*list of :class:`psychrnn.tasks.task.Task` objects*) – List of tasks in the curriculum.

- **metric** (*function*) – What metric function to use. *default_metric()* is an example of one in terms of inputs and outputs taken.
- **accuracies** (*list of functions*) – Accuracy function to use at each stage.
- **thresholds** (*list of float*) – Thresholds for each stage that accuracy must reach to move to the next stage.
- **metric_epoch** (*int*) – Calculate the metric and test if the model should advance to the next stage every `metric_epoch` training epochs.
- **output_file** (*str*) – Optional path for saving out themetric value and stage. If the .npz filename extension is not included, it will be appended.

- **input_data** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Task inputs.
- **correct_output** (*ndarray(dtype=float, shape = (*`N_batch`, `N_steps`, `N_out` *))*) – Correct (target) task output given input_data.
- **output_mask** (*ndarray(dtype=float, shape = (*`N_batch`, `N_steps`, `N_out` *))*) – Output mask for the task. True when the network should aim to match the target output, False when the target output can be ignored.
- **output** (*ndarray(dtype=float, shape = (*`N_batch`, `N_steps`, `N_out` *))*) – The network's output given input_data.
- **epoch** (*int*) – The epoch number in training.
- **losses** (*list of float*) – List of losses, computed during training.
- **verbosity** (*bool*) – Whether to print information as training progresses.

Returns *tuple*

- **advance** (*bool*) – True if the the stage should be advanced. False otherwise.
- **metric_value** (*float*) – Value of the computed metric.

- **accuracies** (*list of functions, optional*) – Optional list of functions to use to calculate network performance for the purposes of advancing tasks. Used by *default_metric()* to compute accuracy. Default: `[tasks[i].accuracy_function for i in range(len(tasks))]`.
- **thresholds** (*list of float, optional*) – Optional list of thresholds. If metric = default_metric, accuracies must reach the threshold for a given stage in order to advance to the next stage. Default: `[.9 for i in range(len(tasks))]`
- **metric_epoch** (*int*) – Calculate the metric and test if the model should advance to the next stage every `metric_epoch` training epochs. Default: 10

- **output_file** (*str*) – Optional path for saving out the metric value and stage. If the .npz filename extension is not included, it will be appended. Default: None.

Methods

get_generator_function()	Return the generator function for the current task.
metric_test(input_data, correct_output, ...)	Evaluates whether to advance the stage to the next task or not.

get_generator_function()
Return the generator function for the current task.
Returns Task batch generator for the task at the current stage.
Return type *psychrnn.tasks.task.Task.batch_generator()* function

metric_test(*input_data*, *correct_output*, *output_mask*, *test_output*, *epoch*, *losses*, *verbosity=False*)
Evaluates whether to advance the stage to the next task or not.
Parameters
- **input_data** (ndarray(dtype=float, shape =(N_batch, N_steps, N_out))) – Task inputs.
- **correct_output** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – Correct (target) task output given input_data.
- **output_mask** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – Output mask for the task. True when the network should aim to match the target output, False when the target output can be ignored.
- **test_output** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – The network's output given input_data.
- **epoch** (**int**) – The epoch number in training.
- **losses** (**list of float**) – List of losses, computed during training.
- **verbosity** (**bool, optional**) – Whether to print information as metric is computed and stages advanced. Default: False

Returns True if stage advances, False otherwise.

psychrnn.backend.curriculum.**default_metric**(*curriculum_params*, *input_data*, *correct_output*, *output_mask*, *output*, *epoch*, *losses*, *verbosity*)
Default metric to use to evaluate performance when using Curriculum learning.

Advance is true if accuracy >= threshold, False otherwise.
Parameters
- **curriculum_params** (*dict*) – Dictionary of the *Curriculum* object parameters, containing the following keys:
  Dictionary Keys
  - **stop_training** (*bool*) – True if the network has finished training and completed all stages.
  - **stage** (*int*) – Current training stage (initial stage is 0).
  - **metric_values** (*list of [float, int]*) – List of metric values and the stage at which each metric value was computed.
  - **tasks** (*list of :class:`psychrnn.tasks.task.Task` objects*) – List of tasks in the curriculum.
  - **metric** (*function*) – What metric function to use. *default_metric()* is an example of one in terms of inputs and outputs taken.

- **accuracies** (*list of functions with the signature of* psychrnn.tasks.task.Task.accuracy_function()) – Accuracy function to use at each stage.
- **thresholds** (*list of float*) – Thresholds for each stage that accuracy must reach to move to the next stage.
- **metric_epoch** (*int*) – Calculate the metric / test if advance to the next stage every metric_epoch training epochs.
- **output_file** (*str*) – Optional path for where to save out metric value and stage.

- **input_data** (ndarray(dtype=float, shape =(N_batch, N_steps, N_out))) – Task inputs.
- **correct_output** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – Correct (target) task output given input_data.
- **output_mask** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – Output mask for the task. True when the network should aim to match the target output, False when the target output can be ignored.
- **output** (ndarray(dtype=float, shape = (N_batch, N_steps, N_out))) – The network's output given input_data.
- **epoch** (**int**) – The epoch number in training.
- **losses** (**list of float**) – List of losses, computed during training.
- **verbosity** (**bool**) – Whether to print information as training progresses. If True, prints accuracy every time it is computed.

Returns

- **advance** (*bool*) – True if the accuracy is >= the threshold for the current stage. False otherwise.
- **metric_value** (*float*) – Value of the computed accuracy.

Return type tuple

Simulation

Simulators implement the forward running of RNN models in NumPy, outside of the TensorFlow framework.

Classes

BasicSimulator([rnn_model, params, ...])	Simulator implementation for psychrnn.backend.models.basic.Basic and for psychrnn.backend.models.basic.BasicScan.
LSTMSimulator([rnn_model, params, ...])	Simulator implementation for psychrnn.backend.models.lstm.LSTM and for psychrnn.backend.models.lstm.LSTM.
Simulator([rnn_model, params, weights_path, ...])	The simulator class.

Functions

relu(x)	NumPy implementation of tf.nn.relu
sigmoid(x)	NumPy implementation of tf.nn.sigmoid

class psychrnn.backend.simulation.**BasicSimulator**(*rnn_model=None*, *params=None*, *weights_path=None*, *weights=None*, *transfer_function=<function relu>*)

Bases: *psychrnn.backend.simulation.Simulator*

Simulator implementation for *psychrnn.backend.models.basic.Basic* and for *psychrnn.backend.models.basic.BasicScan*.

See *Simulator* for arguments. **Methods**

rnn_step(state, rnn_in, t_connectivity)	Given input and previous state, outputs the next state and output of the network as a NumPy implementation of *psychrnn.backend.models.basic.Basic.recurrent_timestep* and of *psychrnn.backend.models.basic.Basic.output_timestep*.
run_trials(trial_input[, t_connectivity])	Test the network on a certain task input, optionally including ablation terms.

rnn_step(*state*, *rnn_in*, *t_connectivity*)

Given input and previous state, outputs the next state and output of the network as a NumPy implementation of *psychrnn.backend.models.basic.Basic.recurrent_timestep* and of *psychrnn.backend.models.basic.Basic.output_timestep*.

Additionally takes in `t_connectivity`. If `t_connectivity` is all ones, *rnn_step()*'s output will match that of *psychrnn.backend.models.basic.Basic.recurrent_timestep* and *psychrnn.backend.models.basic.Basic.output_timestep*. Otherwise `W_rec` is multiplied by `t_connectivity` elementwise, ablating / perturbing the recurrent connectivity.

Parameters

- **state** (ndarray(dtype=float, shape=(`N_batch` , `N_rec`))) – State of network at previous time point.
- **rnn_in** (ndarray(dtype=float, shape=(`N_batch` , `N_in`))) – State of network at previous time point.
- **t_connectivity** (ndarray(dtype=float, shape=(`N_rec` , `N_rec`))) – Matrix for ablating / perturbing W_rec.

Returns

- **new_output** (*ndarray(dtype=float, shape=(*`N_batch`, `N_out` *))*) – Output of the network at a given timepoint for each trial in the batch.
- **new_state** (*ndarray(dtype=float, shape=(*`N_batch`, `N_rec` *))*) – New state of the network for each trial in the batch.

Return type tuple

run_trials(*trial_input*, *t_connectivity=None*)

Test the network on a certain task input, optionally including ablation terms.

A NumPy implementation of *test()* with additional options for ablation.

N_batch here is flexible and will be inferred from trial_input.

Repeatedly calls *rnn_step()* to build output and states over the entire timecourse of the `trial_batch`

Parameters

- **trial_batch** ((*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Task stimulus to run the network on. Stimulus from *psychrnn.tasks.task.Task.get_trial_batch()*, or from next(*psychrnn.tasks.task.Task.batch_generator()*). To run the network autonomously

without input, set input to an array of zeroes. N_steps will still indicate for how many steps to run the network.
- **t_connectivity** ((*ndarray(dtype=float, shape =(*N_steps, N_rec, N_rec *))*) – Matrix for ablating / perturbing W_rec. Passed step by step to *rnn_step()*.

Returns
- **outputs** (*ndarray(dtype=float, shape =(*N_batch, N_steps, N_out *))*) – Output time series of the network for each trial in the batch.
- **states** (*ndarray(dtype=float, shape =(*N_batch, N_steps, N_rec *))*) – Activity of recurrent units during each trial.

Return type tuple

class psychrnn.backend.simulation.**LSTMSimulator**(*rnn_model=None*, *params=None*, *weights_path=None*, *weights=None*)

Bases: *psychrnn.backend.simulation.Simulator*

Simulator implementation for *psychrnn.backend.models.lstm.LSTM* and for *psychrnn.backend.models.lstm.LSTM*.

See *Simulator* for arguments.

The contents of weights / np.load(weights_path) must now include the following additional keys:

Dictionary Keys
- **init_hidden** (*ndarray(dtype=float, shape=(*N_batch , N_rec *))*) – Initial state of the cell state.
- **init_hidden** (*ndarray(dtype=float, shape=(*N_batch , N_rec *))*) – Initial state of the hidden state.
- **W_f** (*ndarray(dtype=float, shape=(*N_rec + N_in, N_rec *))*) – f term weights
- **W_i** (*ndarray(dtype=float, shape=(*N_rec + N_in, N_rec *))*) – i term weights
- **W_c** (*ndarray(dtype=float, shape=(*N_rec + N_in, N_rec *))*) – c term weights
- **W_o** (*ndarray(dtype=float, shape=(*N_rec + N_in, N_rec *))*) – o term weights
- **b_f** (*ndarray(dtype=float, shape=(*N_rec, *))*) – f term bias.
- **b_i** (*ndarray(dtype=float, shape=(*N_rec, *))*) – i term bias.
- **b_c** (*ndarray(dtype=float, shape=(*N_rec, *))*) – c term bias.
- **b_o** (*ndarray(dtype=float, shape=(*N_rec, *))*) – o term bias.

Methods

rnn_step(hidden, cell, rnn_in)	Given input and previous state, outputs the next state and output of the network as a NumPy implementation of *psychrnn.backend.models.lstm.LSTM.recurrent_timestep* and of *psychrnn.backend.models.lstm.LSTM.output_timestep*.
run_trials(trial_input)	Test the network on a certain task input, optionally including ablation terms.

rnn_step(*hidden*, *cell*, *rnn_in*)

Given input and previous state, outputs the next state and output of the network as a NumPy implementation of *psychrnn.backend.models.lstm.LSTM.recurrent_timestep* and of *psychrnn.backend.models.lstm.LSTM.output_timestep*.

Parameters
- **hidden** (ndarray(dtype=float, shape=(N_batch , N_rec))) – Hidden units state of network at previous time point.
- **cell** (ndarray(dtype=float, shape=(N_batch , N_rec))) – Cell state of the network at previous time point.
- **rnn_in** (ndarray(dtype=float, shape=(N_batch , N_in))) – State of network at

previous time point.

Returns

- **new_output** (*ndarray(dtype=float, shape=(*`N_batch`, `N_out` *))*) – Output of the network at a given timepoint for each trial in the batch.
- **new_hidden** (*ndarray(dtype=float, shape=(* `N_batch` , `N_rec` *))*) – New hidden unit state of the network.
- **new_cell** (*ndarray(dtype=float, shape=(* `N_batch` , `N_rec` *))*) – New cell state of the network.

Return type tuple

`run_trials`(*trial_input*)

Test the network on a certain task input, optionally including ablation terms.

A NumPy implementation of *`test()`* with additional options for ablation.

N_batch here is flexible and will be inferred from trial_input.

Repeatedly calls *`rnn_step()`* to build output and states over the entire timecourse of the `trial_batch`

Parameters **trial_batch** ((*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Task stimulus to run the network on. Stimulus from *`psychrnn.tasks.task.Task.get_trial_batch()`*, or from next(*`psychrnn.tasks.task.Task.batch_generator()`*). To run the network autonomously without input, set input to an array of zeroes. N_steps will still indicate for how many steps to run the network.

Returns

- **outputs** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*) – Output time series of the network for each trial in the batch.
- **states** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_rec` *))*) – Activity of recurrent units during each trial.

Return type tuple

class psychrnn.backend.simulation.**Simulator**(*rnn_model=None*, *params=None*, *weights_path=None*, *weights=None*, *transfer_function=<function relu>*)

Bases: `abc.ABC`

The simulator class.

Note: The base Simulator class is not itself a functioning Simulator. run_trials and rnn_step must be implemented to define a functioning Simulator

---

Methods

`rnn_step`(state, rnn_in, t_connectivity)	Given input and previous state, outputs the next state and output of the network.
`run_trials`(trial_input[, t_connectivity])	Test the network on a certain task input, optionally including ablation terms.

Parameters

- **rnn_model** (*`psychrnn.backend.rnn.RNN`* object, optional) – Uses the *`psychrnn.backend.rnn.RNN`* object to set `alpha` and `rec_noise`. Also used to initialize weights if `weights` and `weights_path` are not passed in. Default: None.
- **weights_path** (*str, optional*) – Where to load weights from. Take precedence over rnn_model weights. Default: `rnn_model.get_weights()`. np.load(`weights_path`) should return something of the form `weights`.

- **transfer_function** (*function, optonal*) – Function that takes an ndarray as input and outputs an ndarray of the same shape with the transfer / activation function applied. NumPy implementation of a TensorFlow transfer function. Default: `relu()`.
- **weights** (*dict, optional*) – Takes precedence over both weights_path and rnn_model. Default: np.load(`weights_path`). Dictionary containing the following keys:

  Dictionary Keys

  - **init_state** (*ndarray(dtype=float, shape=(1,* `N_rec` *))*) – Initial state of the network's recurrent units.
  - **W_in** (*ndarray(dtype=float, shape=(*`N_rec`*.* `N_in` *))*) – Input weights.
  - **W_rec** (*ndarray(dtype=float, shape=(*`N_rec`*,* `N_rec` *))*) – Recurrent weights.
  - **W_out** (*ndarray(dtype=float, shape=(*`N_out`*,* `N_rec` *))*) – Output weights.
  - **b_rec** (*ndarray(dtype=float, shape=(*`N_rec`*,))*) – Recurrent bias.
  - **b_out** (*ndarray(dtype=float, shape=(*`N_out`*,))*) – Output bias.
- **params** (*dict, optional*) –

  Dictionary Keys

  - **rec_noise** (*float, optional*) – Amount of recurrent noise to add to the network. Default: 0
  - **alpha** (*float, optional*) – The number of unit time constants per simulation timestep. Defaut: (1.0* dt) / tau
  - **dt** (*float, optional*) – The simulation timestep. Used to calculate alpha if alpha is not passed in. Required if alpha is not in params and rnn_model is None.
  - **tau** (*float*) – The intrinsic time constant of neural state decay. Used to calculate alpha if alpha is not passed in. Required if alpha is not in params and rnn_model is None.

abstract rnn_step(*state*, *rnn_in*, *t_connectivity*)

Given input and previous state, outputs the next state and output of the network.

Note: This is an abstract function that must be defined in a child class.

Parameters

- **state** (ndarray(dtype=float, shape=(`N_batch` , `N_rec`))) – State of network at previous time point.
- **rnn_in** (ndarray(dtype=float, shape=(`N_batch` , `N_in`))) – State of network at previous time point.
- **t_connectivity** (ndarray(dtype=float, shape=(`N_rec` , `N_rec`))) – Matrix for ablating / perturbing W_rec.

Returns

- **new_output** (*ndarray(dtype=float, shape=(*`N_batch`*,* `N_out` *))*) – Output of the network at a given timepoint for each trial in the batch.
- **new_state** (*ndarray(dtype=float, shape=(*`N_batch`*,* `N_rec` *))*) – New state of the network for each trial in the batch.

Return type tuple

abstract run_trials(*trial_input*, *t_connectivity=None*)

Test the network on a certain task input, optionally including ablation terms.

A NumPy implementation of *test()* with additional options for ablation.

N_batch here is flexible and will be inferred from trial_input.

Parameters

- **trial_batch** ((*ndarray(dtype=float, shape =(*N_batch, N_steps, N_out *))*) – Task stimulus to run the network on. Stimulus from *psychrnn.tasks.task.Task.get_trial_batch()*, or from next(*psychrnn.tasks.task.Task.batch_generator()*). If you want the network to run autonomously, without input, set input to an array of zeroes, N_steps will still indicate how long to run the network.
- **t_connectivity** ((*ndarray(dtype=float, shape =(*N_steps, N_rec, N_rec *))*) – Matrix for ablating / perturbing W_rec. Passed step by step to rnn_step.

Returns

- **outputs** (*ndarray(dtype=float, shape =(*N_batch, N_steps, N_out *))*) – Output time series of the network for each trial in the batch.
- **states** (*ndarray(dtype=float, shape =(*N_batch, N_steps, N_rec *))*) – Activity of recurrent units during each trial.

Return type tuple

psychrnn.backend.simulation.**relu**(*x*)

NumPy implementation of tf.nn.relu

Parameters **x** (*ndarray*) – array for which relu is computed.

Returns np.maximum(x,0)

Return type ndarray

psychrnn.backend.simulation.**sigmoid**(*x*)

NumPy implementation of tf.nn.sigmoid

Parameters **x** (*ndarray*) – array for which sigmoid is computed.

Returns 1/(1 + np.exp(-x))

Return type ndarray

2.2 Tasks

2.2.1 Base Task Object

Classes

Task(N_in, N_out, dt, tau, T, N_batch)	The base task class.

class psychrnn.tasks.task.**Task**(*N_in*, *N_out*, *dt*, *tau*, *T*, *N_batch*)

Bases: abc.ABC

The base task class.

The base task class provides the structure that users can use to define a new task. This structure is used by example tasks *PerceptualDiscrimination*, *MatchToCategory*, and *DelayedDiscrimination*.

Note: The base task class is not itself a functioning task. The generate_trial_params and trial_function must be defined to define a new, functioning, task.

Methods

accuracy_function(correct_output, ...)	Function to calculate accuracy (not loss) as it would be measured experimentally.
batch_generator()	Generates a batch of trials.
generate_trial(params)	Loop to generate a single trial.
generate_trial_params(batch, trial)	Define parameters for each trial.
get_task_params()	Get dictionary of task parameters.
get_trial_batch()	Get a batch of trials.
trial_function(time, params)	Compute the trial properties at `time`.

Parameters
- **N_in** (*int*) – The number of network inputs.
- **N_out** (*int*) – The number of network outputs.
- **dt** (*float*) – The simulation timestep.
- **tau** (*float*) – The intrinsic time constant of neural state decay.
- **T** (*float*) – The trial length.
- **N_batch** (*int*) – The number of trials per training update.

Inferred Parameters:
- **alpha** (*float*) – The number of unit time constants per simulation timestep.
- **N_steps** (*int*): The number of simulation timesteps in a trial.

accuracy_function(*correct_output*, *test_output*, *output_mask*)

Function to calculate accuracy (not loss) as it would be measured experimentally.

Output should range from 0 to 1. This function is used by *Curriculum* as part of it's *default_metric()*.

Parameters
- **correct_output** (ndarray(dtype=float, shape =(`N_batch`, `N_steps`, `N_out`))) – Correct batch output. `y_data` as returned by *batch_generator()*.
- **test_output** (ndarray(dtype=float, shape =(`N_batch`, `N_steps`, `N_out`))) – Output to compute the accuracy of. `output` as returned by *psychrnn.backend.rnn.RNN.test()*.
- **output_mask** (ndarray(dtype=bool, shape =(`N_batch`, `N_steps`, `N_out`))) – Mask. `mask` as returned by func:*batch_generator*.

Returns 0 <= accuracy <=1

Return type float

> **Warning:** This function is abstract and may optionally be implemented in a child Task object.

Example

See *PerceptualDiscrimination*, *MatchToCategory*, and *DelayedDiscrimination* for example implementations.

batch_generator()

Generates a batch of trials.

Returns

Return type Generator[tuple, None, None]

Yields *tuple* –
- **stimulus** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Task stimuli for `N_batch` trials.

- **target_output** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Target output for the network on `N_batch` trials given the `stimulus`.
- **output_mask** (*ndarray(dtype=bool, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Output mask for `N_batch` trials. True when the network should aim to match the target output, False when the target output can be ignored.
- **trial_params** (*ndarray(dtype=dict, shape =(*`N_batch` *,))*): Array of dictionaries containing the trial parameters produced by *generate_trial_params()* for each trial in `N_batch`.

generate_trial(*params*)

Loop to generate a single trial.

Parameters **params** (*dict*) – Dictionary of trial parameters generated by *generate_trial_params()*.

Returns

- **x_trial** (*ndarray(dtype=float, shape=(*`N_steps`, `N_in` *))*) – Trial input given `params`.
- **y_trial** (*ndarray(dtype=float, shape=(*`N_steps`, `N_out` *))*) – Correct trial output given `params`.
- **mask_trial** (*ndarray(dtype=bool, shape=(*`N_steps`, `N_out` *))*) – True during steps where the network should train to match `y`, False where the network should ignore `y` during training.

Return type tuple

abstract generate_trial_params(*batch*, *trial*)

Define parameters for each trial.

Using a combination of randomness, presets, and task attributes, define the necessary trial parameters.

Parameters

- **batch** (*int*) – The batch number for this trial.
- **trial** (*int*) – The trial number of the trial within the batch data:*batch*.

Returns Dictionary of trial parameters.

Return type dict

Warning: This function is abstract and must be implemented in a child Task object.

Example

See *PerceptualDiscrimination*, *MatchToCategory*, and *DelayedDiscrimination* for example implementations.

get_task_params()

Get dictionary of task parameters.

Note: N_in, N_out, N_batch, dt, tau and N_steps must all be passed to the network model as parameters – this function is the recommended way to begin building the network_params that will be passed into the RNN model.

Returns

Dictionary of *Task* attributes including the following keys:

Dictionary Keys

- **N_batch** (*int*) – The number of trials per training update.
- **N_in** (*int*) – The number of network inputs.

- **N_out** (*int*) – The number of network outputs.
- **dt** (*float*) – The simulation timestep.
- **tau** (*float*) – The unit time constant.
- **T** (*float*) – The trial length.
- **alpha** (*float*) – The number of unit time constants per simulation timestep.
- **N_steps** (*int*): The number of simulation timesteps in a trial.

Note: The dictionary will also include any other attributes defined in your task definition.

Return type dict

get_trial_batch()
Get a batch of trials.

Wrapper for `next(self.batch_generator())`.

Returns
- **stimulus** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_in` *))*): Task stimuli for `N_batch` trials.
- **target_output** (*ndarray(dtype=float, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Target output for the network on `N_batch` trials given the `stimulus`.
- **output_mask** (*ndarray(dtype=bool, shape =(*`N_batch`, `N_steps`, `N_out` *))*): Output mask for `N_batch` trials. True when the network should aim to match the target output, False when the target output can be ignored.
- **trial_params** (*ndarray(dtype=dict, shape =(*`N_batch` *,))*): Array of dictionaries containing the trial parameters produced by *generate_trial_params()* for each trial in `N_batch`.

Return type tuple

abstract trial_function(*time*, *params*)
Compute the trial properties at `time`.

Based on the :data:'params' compute the trial stimulus (x_t), correct output (y_t), and mask (mask_t) at `time`.

Parameters
- **time** (*int*) – The time within the trial (0 <= `time` < `T`).
- **params** (*dict*) – The trial params produced by *generate_trial_params()*

Returns
- **x_t** (*ndarray(dtype=float, shape=(*`N_in` *,))*) – Trial input at `time` given `params`.
- **y_t** (*ndarray(dtype=float, shape=(*`N_out` *,))*) – Correct trial output at `time` given `params`.
- **mask_t** (*ndarray(dtype=bool, shape=(*`N_out` *,))*) – True if the network should train to match the y_t, False if the network should ignore y_t when training.

Return type tuple

Warning: This function is abstract and must be implemented in a child Task object.

Example

See *PerceptualDiscrimination*, *MatchToCategory*, and *DelayedDiscrimination* for example implementations.

2.2.2 Implemented Example Tasks

Delayed Discrimination Task

Classes

DelayedDiscrimination(dt, tau, T, N_batch[, ...])	Delayed discrimination task.

class psychrnn.tasks.delayed_discrim.**DelayedDiscrimination**(*dt*, *tau*, *T*, *N_batch*, *onset_time=None*, *stim_duration_1=None*, *delay_duration=None*, *stim_duration_2=None*, *decision_duration=None*)

Bases: *psychrnn.tasks.task.Task*

Delayed discrimination task.

Following a fore period, the network receives an input, followed by a delay. After the delay the network receives a second input. The second input channel receives noisy input that is inversely ordered compared to the input received by the first input channel. The network must respond by activating the output node that corresponds to the input channel with the greater input as the first stimulus.

Takes two channels of noisy input (`N_in` = 2). Two channel output (`N_out` = 2) with a one hot encoding (high value is 1, low value is .2).

Loosely based on Romo, R., Brody, C. D., Hernández, A., & Lemus, L. (1999). Neuronal correlates of parametric working memory in the prefrontal cortex. Nature, 399(6735), 470.

Parameters

- **dt** (*float*) – The simulation timestep.
- **tau** (*float*) – The intrinsic time constant of neural state decay.
- **T** (*float*) – The trial length.
- **N_batch** (*int*) – The number of trials per training update.
- **onset_time** (*float, optional*) – Stimulus onset time in terms of trial length `T`.
- **stim_duration_1** (*float, optional*) – Stimulus 1 duration in terms of trial length `T`.
- **delay_duration** (*float, optional*) – Delay duration in terms of trial length `T`.
- **stim_duration_2** (*float, optional*) – Stimulus 2 duration in terms of trial length `T`.
- **decision_duration** (*float, optional*) – Decision duration in terms of trial length `T`.

Methods

accuracy_function(correct_output, ...)	Calculates the accuracy of `test_output`.
generate_trial_params(batch, trial)	Define parameters for each trial.
trial_function(t, params)	Compute the trial properties at `time`.

accuracy_function(*correct_output*, *test_output*, *output_mask*)

Calculates the accuracy of `test_output`.

Implements *accuracy_function()*.

Takes the channel-wise mean of the masked output for each trial. Whichever channel has a greater mean is considered to be the network's "choice".

Returns 0 <= accuracy <= 1. Accuracy is equal to the ratio of trials in which the network made the correct choice as defined above.

Return type float

generate_trial_params(*batch*, *trial*)

Define parameters for each trial.

Implements *generate_trial_params()*.

Parameters

- **batch** (*int*) – The batch number that this trial is part of.
- **trial** (*int*) – The trial number of the trial within the batch *batch*.

Returns

Dictionary of trial parameters including the following keys:

Dictionary Keys

- **stimulus_1** (*float*) – Start time for stimulus one. `onset_time`.
- **delay** (*float*) – Start time for the delay. `onset_time` + `stimulus_duration_1`.
- **stimulus_2** (*float*) – Start time in for stimulus one. `onset_time` + `stimulus_duration_1` + `delay_duration`.
- **decision** (*float*) – Start time in for decision period. `onset_time` + `stimulus_duration_1` + `delay_duration` + `stimulus_duration_2`.
- **end** (*float*) – End of decision period. `onset_time` + `stimulus_duration_1` + `delay_duration` + `stimulus_duration_2` + `decision_duration`.
- **stim_noise** (*float*) – Scales the stimlus noise. Set to .1.
- **f1** (*int*) – Frequency of first stimulus.
- **f2** (*int*) – Frequency of second stimulus.
- **choice** (*str*) – Indicates whether `f1` is '>' or '<' `f2`.

Return type dict

trial_function(*t*, *params*)

Compute the trial properties at `time`.

Implements *trial_function()*.

Based on the `params` compute the trial stimulus (x_t), correct output (y_t), and mask (mask_t) at `time`.

Parameters

- **time** (*int*) – The time within the trial (0 <= `time` < `T`).
- **params** (*dict*) – The trial params produced by *generate_trial_params()*.

Returns

- **x_t** (*ndarray(dtype=float, shape=(*`N_in` *,))*) – Trial input at `time` given `params`. First channel contains `f1` during the first stimulus period, and `f2` during the second stimulus period, scaled to be between .4 and 1.2. Second channel

contains the frequencies but reverse scaled – high frequencies correspond to low values and vice versa. Both channels have baseline noise.
- **y_t** (*ndarray(dtype=float, shape=(*`N_out` *,))*) – Correct trial output at `time` given `params`. The correct output is encoded using one-hot encoding during the decision period.
- **mask_t** (*ndarray(dtype=bool, shape=(*`N_out` *,))*) – True if the network should train to match the y_t, False if the network should ignore y_t when training. The mask is True for during the decision period and False otherwise.

Return type tuple

Match to Category Task

Classes

MatchToCategory(dt, tau, T, N_batch[, N_in, ...])	Multidirectional decision-making task.

class psychrnn.tasks.match_to_category.**MatchToCategory**(*dt*, *tau*, *T*, *N_batch*, *N_in=16*, *N_out=2*)

Bases: *psychrnn.tasks.task.Task*

Multidirectional decision-making task.

On each trial the network receives input from units representing different locations on a ring. Each input unit magnitude represents closeness to the angle of input. The network must determine which side of arbitrary category boundaries the input belongs to and respond accordingly.

Takes `N_in` channels of noisy input arranged in a ring with gaussian signal around the ring centered at 0 at the stimulus angle. `N_out` channel output arranged as slices of a ring with a one hot encoding towards the correct category output based on the angular location of the gaussian input bump.

Loosely based on Freedman, David J., and John A. Assad. "Experience-dependent representation of visual categories in parietal cortex." Nature 443.7107 (2006): 85-88.

Parameters
- **dt** (*float*) – The simulation timestep.
- **tau** (*float*) – The intrinsic time constant of neural state decay.
- **T** (*float*) – The trial length.
- **N_batch** (*int*) – The number of trials per training update.
- **N_in** (*int, optional*) – The number of network inputs. Defaults to 16.
- **N_out** (*int, optional*) – The number of network outputs. Defaults to 2.

Methods

accuracy_function(correct_output, ...)	Calculates the accuracy of `test_output`.
generate_trial_params(batch, trial)	Define parameters for each trial.
trial_function(t, params)	Compute the trial properties at `time`.

accuracy_function(*correct_output*, *test_output*, *output_mask*)

Calculates the accuracy of `test_output`.

Implements *accuracy_function()*.

Takes the channel-wise mean of the masked output for each trial. Whichever channel has a greater mean is considered to be the network's "choice".

Returns 0 <= accuracy <= 1. Accuracy is equal to the ratio of trials in which the network made the correct choice as defined above.

Return type float

generate_trial_params(*batch*, *trial*)

Define parameters for each trial.

Implements *generate_trial_params()*.

Parameters

- **batch** (*int*) – The batch number that this trial is part of.
- **trial** (*int*) – The trial number of the trial within the batch *batch*.

Returns

Dictionary of trial parameters including the following keys:

Dictionary Keys

- **angle** (*float*) – Angle at which to center the gaussian. Randomly selected.
- **category** (*int*) – Index of the N_out category channel that contains the `angle`.
- **onset_time** (*float*) – Stimulus onset time. Set to 200.
- **input_dur** (*float*) – Stimulus duration. Set to 1000.
- **output_dur** (*float*) – Output duration. The time given to make a choice. Set to 800.
- **stim_noise** (*float*) – Scales the stimlus noise. Set to .1.

Return type dict

trial_function(*t*, *params*)

Compute the trial properties at `time`.

Implements *trial_function()*.

Based on the `params` compute the trial stimulus (x_t), correct output (y_t), and mask (mask_t) at `time`.

Parameters

- **time** (*int*) – The time within the trial (0 <= `time` < `T`).
- **params** (*dict*) – The trial params produced by *generate_trial_params()*.

Returns

- **x_t** (*ndarray(dtype=float, shape=(*`N_in` *,)))* – Trial input at `time` given `params`. For `params['onset_time'] < time < params['onset_time'] + params['input_dur']`, gaussian pdf with mean = angle and scale = 1 is added to each input channel based on the channel's angle.
- **y_t** (*ndarray(dtype=float, shape=(*`N_out` *,)))* – Correct trial output at `time` given `params`. 1 in the `params['category']` output channel during the output period defined by `params['output_dur']`, 0 otherwise.
- **mask_t** (*ndarray(dtype=bool, shape=(*`N_out` *,)))* – True if the network should train to match the y_t, False if the network should ignore y_t when training. True during the output period, False otherwise.

Return type tuple

Perceptual Discrimination Task

Classes

PerceptualDiscrimination(dt, tau, T, N_batch)	Two alternative forced choice (2AFC) binary discrimination task.

class psychrnn.tasks.perceptual_discrimination.**PerceptualDiscrimination**(*dt*, *tau*, *T*, *N_batch*, *coherence=None*, *direction=None*)

Bases: *psychrnn.tasks.task.Task*

Two alternative forced choice (2AFC) binary discrimination task.

On each trial the network receives two simultaneous noisy inputs into each of two input channels. The network must determine which channel has the higher mean input and respond by driving the corresponding output unit to 1.

Takes two channels of noisy input (`N_in` = 2). Two channel output (`N_out` = 2) with a one hot encoding (high value is 1, low value is .2) towards the higher mean channel.

Loosely based on Britten, Kenneth H., et al. "The analysis of visual motion: a comparison of neuronal and psychophysical performance." Journal of Neuroscience 12.12 (1992): 4745-4765

Parameters

- **dt** (*float*) – The simulation timestep.
- **tau** (*float*) – The intrinsic time constant of neural state decay.
- **T** (*float*) – The trial length.
- **N_batch** (*int*) – The number of trials per training update.
- **coherence** (*float, optional*) – Amount by which the means of the two channels will differ. By default None.
- **direction** (*int, optional*) – Either 0 or 1, indicates which input channel will have higher mean input. By default None.

Methods

accuracy_function(correct_output, ...)	Calculates the accuracy of `test_output`.
generate_trial_params(batch, trial)	Define parameters for each trial.
trial_function(t, params)	Compute the trial properties at `time`.

accuracy_function(*correct_output*, *test_output*, *output_mask*)

Calculates the accuracy of `test_output`.

Implements *accuracy_function()*.

Takes the channel-wise mean of the masked output for each trial. Whichever channel has a greater mean is considered to be the network's "choice".

Returns 0 <= accuracy <= 1. Accuracy is equal to the ratio of trials in which the network made the correct choice as defined above.

Return type float

generate_trial_params(*batch*, *trial*)
Define parameters for each trial.

Implements *generate_trial_params()*.

Parameters
- **batch** (*int*) – The batch number that this trial is part of.
- **trial** (*int*) – The trial number of the trial within the batch *batch*.

Returns

Dictionary of trial parameters including the following keys:

Dictionary Keys

- **coherence** (*float*) – Amount by which the means of the two channels will differ. `self.coherence` if not None, otherwise `np.random.exponential(scale=1/5)`.
- **direction** (*int*) – Either 0 or 1, indicates which input channel will have higher mean input. `self.direction` if not None, otherwise `np.random.choice([0, 1])`.
- **stim_noise** (*float*) – Scales the stimlus noise. Set to .1.
- **onset_time** (*float*) – Stimulus onset time. `np.random.random() * self.T / 2.0`.
- **stim_duration** (*float*) – Stimulus duration. `np.random.random() * self.T / 4.0 + self.T / 8.0`.

Return type dict

trial_function(*t*, *params*)
Compute the trial properties at `time`.

Implements *trial_function()*.

Based on the `params` compute the trial stimulus (x_t), correct output (y_t), and mask (mask_t) at `time`.

Parameters
- **time** (*int*) – The time within the trial (0 <= `time` < `T`).
- **params** (*dict*) – The trial params produced by *generate_trial_params()*.

Returns
- **x_t** (*ndarray(dtype=float, shape=(*`N_in` *,))*) – Trial input at `time` given `params`. For `params['onset_time'] < time < params['onset_time'] + params['stim_duration']` , 1 is added to the noise in both channels, and `params['coherence']` is also added in the channel corresponding to `params[dir]`.
- **y_t** (*ndarray(dtype=float, shape=(*`N_out` *,))*) – Correct trial output at `time` given `params`. From `time > params['onset_time'] + params[stim_duration] + 20` onwards, the correct output is encoded using one-hot encoding. Until then, y_t is 0 in both channels.
- **mask_t** (*ndarray(dtype=bool, shape=(*`N_out` *,))*) – True if the network should train to match the y_t, False if the network should ignore y_t when training. The mask is True for `time > params['onset_time'] + params['stim_duration']` and False otherwise.

Return type tuple

CHAPTER

THREE

GETTING STARTED

Each guide below includes a link to a Colab notebook that will let you experiment with each example on your own in the browser.

3.1 Hello World!

A popular 2-alternative forced choice perceptual discrimination task is the random dot motion (RDM) task. In RDM, the subject observes dots moving in different directions. The RDM task is a forced choice task – although dots can move in any direction, their are two directions in which the movement of the coherent dots could be. The subject must make a choice towards one of the two directions at the end of the stimulus period (Britten et al., 1992).

To make it possible for an RNN to complete this task, we model this task as two simultaneous noisy inputs into each of two input channels, representing the two directions. The network must determine which channel has the higher mean input and respond by driving the corresponding output unit to 1, and the other output unit to .2. We've included this example task in PsychRNN as *PerceptualDiscrimination*.

To get started, let's train a basic model in PsychRNN on this 2-alternative forced choice perceptual discrimination task and test how it does on task input. For simplicity, we will use the model defaults.

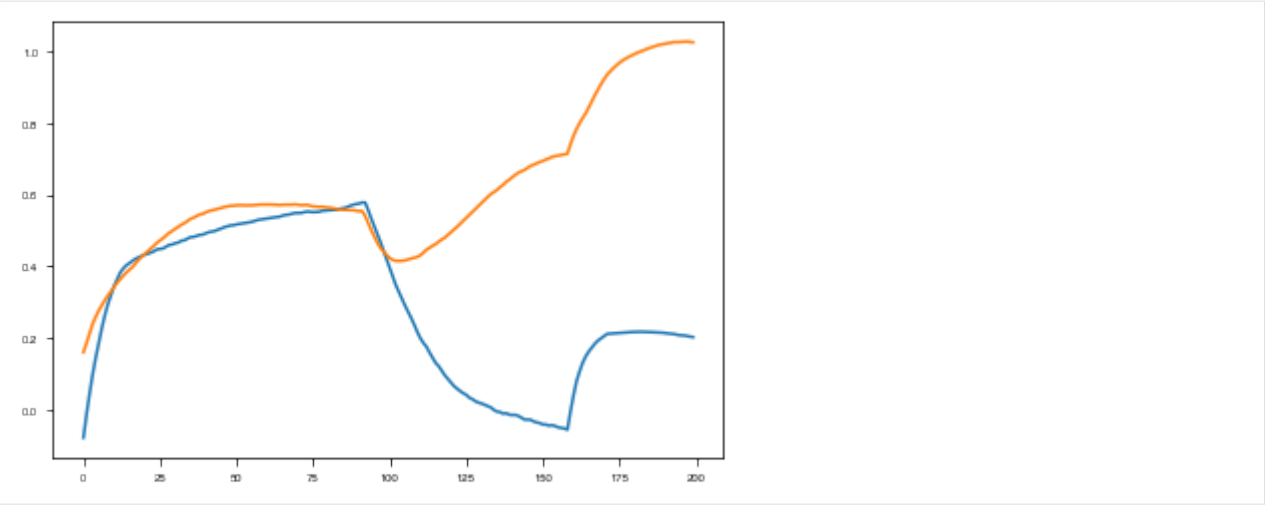
[2]:
```
from matplotlib import pyplot as plt
%matplotlib inline

# ---------------------- Import the package --------------------------
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination
from psychrnn.backend.models.basic import Basic

# ---------------------- Set up a basic model ------------------------
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
network_params = pd.get_task_params() # get the params passed in and defined in pd
network_params['name'] = 'model' # name the model uniquely if running mult models in␣
↪unison
network_params['N_rec'] = 50 # set the number of recurrent units in the model
model = Basic(network_params) # instantiate a basic vanilla RNN

# ---------------------- Train a basic model -------------------------
model.train(pd) # train model to perform pd task

# ---------------------- Test the trained model ----------------------
x,target_output,mask, trial_params = pd.get_trial_batch() # get pd task inputs and␣
↪outputs
model_output, model_state = model.test(x) # run the model on input x

# ---------------------- Plot the results ----------------------------
```

(continues on next page)

(continued from previous page)

```
plt.plot(model_output[0][0,:,:])

# ---------------------- Teardown the model -------------------------
model.destruct()
```

```
Iter 1280, Minibatch Loss= 0.173182
Iter 2560, Minibatch Loss= 0.110828
Iter 3840, Minibatch Loss= 0.089823
Iter 5120, Minibatch Loss= 0.090613
Iter 6400, Minibatch Loss= 0.082815
Iter 7680, Minibatch Loss= 0.084474
Iter 8960, Minibatch Loss= 0.084676
Iter 10240, Minibatch Loss= 0.082200
Iter 11520, Minibatch Loss= 0.076985
Iter 12800, Minibatch Loss= 0.080215
Iter 14080, Minibatch Loss= 0.078905
Iter 15360, Minibatch Loss= 0.074752
Iter 16640, Minibatch Loss= 0.071335
Iter 17920, Minibatch Loss= 0.062578
Iter 19200, Minibatch Loss= 0.040781
Iter 20480, Minibatch Loss= 0.033980
Iter 21760, Minibatch Loss= 0.043546
Iter 23040, Minibatch Loss= 0.025923
Iter 24320, Minibatch Loss= 0.022121
Iter 25600, Minibatch Loss= 0.018697
Iter 26880, Minibatch Loss= 0.018280
Iter 28160, Minibatch Loss= 0.021877
Iter 29440, Minibatch Loss= 0.016974
Iter 30720, Minibatch Loss= 0.020424
Iter 32000, Minibatch Loss= 0.022463
Iter 33280, Minibatch Loss= 0.018284
Iter 34560, Minibatch Loss= 0.029344
Iter 35840, Minibatch Loss= 0.014679
Iter 37120, Minibatch Loss= 0.024408
Iter 38400, Minibatch Loss= 0.016357
Iter 39680, Minibatch Loss= 0.023122
Iter 40960, Minibatch Loss= 0.019468
Iter 42240, Minibatch Loss= 0.018826
Iter 43520, Minibatch Loss= 0.013565
Iter 44800, Minibatch Loss= 0.015086
Iter 46080, Minibatch Loss= 0.018692
Iter 47360, Minibatch Loss= 0.012970
Iter 48640, Minibatch Loss= 0.018514
Iter 49920, Minibatch Loss= 0.016651
Optimization finished!
```



Congratulations! You've successfully trained and tested your first model! Continue to *Simple Example* to learn how to define more useful models.

3.2 Simple Example

This example walks through the steps and options involved in setting up and training a recurrent neural network on a cognitive task.

Most users will want to *define their own tasks*, but for the purposes of getting familiar with the package features, we will use one of the *built-in tasks*, the 2-alternative forced choice *Perceptual Discrimination* task.

This example will use the *Basic* implementation of *RNN*. If you are new to RNNs, we recommend you stick with the Basic implementation. PsychRNN also includes *BasicScan* and *LSTM* implementations of RNN. If you want to use a different architecture, you can *define a new model*, but that should not be necessary for most use cases.

[2]:
```
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination
from psychrnn.backend.models.basic import Basic

import tensorflow as tf
from matplotlib import pyplot as plt
%matplotlib inline
```

3.2.1 Initialize Task

First we define some global parameters that we will use when setting up the task and the model:

[3]:
```
dt = 10 # The simulation timestep.
tau = 100 # The intrinsic time constant of neural state decay.
T = 2000 # The trial length.
N_batch = 50 # The number of trials per training update.
N_rec = 50 # The number of recurrent units in the network.
name = 'basicModel' #  Unique name used to determine variable scope for internal use.
```

[4]:
```
pd = PerceptualDiscrimination(dt = dt, tau = tau, T = T, N_batch = N_batch) #␣
↪Initialize the task object
```

3.2.2 Initialize Model

When we initialize the model, we pass in a dictionary of parameters that will determine how the network is set up.

Set Up Network Parameters

PerceptualDiscrimination.get_task_params() puts the passed in parameters and other generated parameters into a dictionary we can then use to initialize our Basic RNN model.

[5]:
```
network_params = pd.get_task_params()
print(network_params)
```

```
{'N_batch': 50, 'N_in': 2, 'N_out': 2, 'dt': 10, 'tau': 100, 'T': 2000, 'alpha': 0.1,
↪'N_steps': 200, 'coherence': None, 'direction': None, 'lo': 0.2, 'hi': 1.0}
```

We add in a few params that any *RNN* needs but that the *Task* doesn't generate for us.

[6]:
```
network_params['name'] = name # Unique name used to determine variable scope.
network_params['N_rec'] = N_rec # The number of recurrent units in the network.
```

There are some other optional parameters we can add in. Additional parameter options like those for *biological constraints*, *loading weights*, and *other features* are also available:

[7]:
```
network_params['rec_noise'] = 0.0 # Noise into each recurrent unit. Default: 0.0
network_params['W_in_train'] = True # Indicates whether W_in is trainable. Default:␣
↪True
network_params['W_rec_train'] = True # Indicates whether W_rec is trainable. Default:␣
↪True
network_params['W_out_train'] = True # Indicates whether W_out is trainable. Default:␣
↪True
network_params['b_rec_train'] = True # Indicates whether b_rec is trainable. Default:␣
↪True
network_params['b_out_train'] = True # Indicates whether b_out is trainable. Default:␣
↪True
network_params['init_state_train'] = True # Indicates whether init_state is trainable.
↪ Default: True

network_params['transfer_function'] = tf.nn.relu # Transfer function to use for the␣
↪network. Default: tf.nn.relu.
network_params['loss_function'] = "mean_squared_error"# String indicating what loss␣
↪function to use. If not `mean_squared_error` or `binary_cross_entropy`, params[
↪"loss_function"] defines the custom loss function. Default: "mean_squared_error".

network_params['load_weights_path'] = None # When given a path, loads weights from␣
↪file in that path. Default: None
# network_params['initializer'] = # Initializer to use for the network. Default:␣
↪WeightInitializer (network_params) if network_params includes W_rec or load_weights_
↪path as a key, GaussianSpectralRadius (network_params) otherwise.
```

Initialization Parameters

When `network_params['initializer']` is not set, the following optional parameters will be passed to the initializer. See *WeightInitializer* for more details. If `network_params['W_rec']` and `network_params['load_weights_path']` are not set, these parameters will be passed to the *GaussianSpectralRadius Initializer*. Not all optional parameters are shown here. See *Biological Constraints* and *Loading Model with Weights* for more options.

[8]:
```
network_params['which_rand_init'] = 'glorot_gauss' # Which random initialization to␣
↪use for W_in and W_out. Will also be used for W_rec if which_rand_W_rec_init is not␣
↪passed in. Options: 'const_unif', 'const_gauss', 'glorot_unif', 'glorot_gauss'.␣
↪Default: 'glorot_gauss'.

network_params['which_rand_W_rec_init'] = network_params['which_rand_init'] # 'Which␣
↪random initialization to use for W_rec. Options: 'const_unif', 'const_gauss',
↪'glorot_unif', 'glorot_gauss'. Default: which_rand_init.

network_params['init_minval'] = -.1 # Used by const_unif_init() as minval if 'const_
↪unif' is passed in for which_rand_init or which_rand_W_rec_init. Default: -.1.

network_params['init_maxval'] = .1 # Used by const_unif_init() as maxval if 'const_
↪unif' is passed in for which_rand_init or which_rand_W_rec_init. Default: .1.
```

Regularization Parameters

Parameters for regularizing the loss are passed in through network_params as well. By default, there is no regularization. Below are options for regularizations to include. See *Regularizer* for details.

[9]:
```
network_params['L1_in'] = 0 # Parameter for weighting the L1 input weights␣
↪regularization. Default: 0.

network_params['L1_rec'] = 0 # Parameter for weighting the L1 recurrent weights␣
↪regularization. Default: 0.

network_params['L1_out'] = 0 # Parameter for weighting the L1 output weights␣
↪regularization. Default: 0.

network_params['L2_in'] = 0 # Parameter for weighting the L2 input weights␣
↪regularization. Default: 0.

network_params['L2_rec'] = 0 # Parameter for weighting the L2 recurrent weights␣
↪regularization. Default: 0.

network_params['L2_out'] = 0 # Parameter for weighting the L2 output weights␣
↪regularization. Default: 0.

network_params['L2_firing_rate'] = 0 # Parameter for weighting the L2 regularization␣
↪of the relu thresholded states. Default: 0.

network_params['custom_regularization'] = None # Custom regularization function.␣
↪Default: None.
```

Instantiate Model

[10]:
```
basicModel = Basic(network_params)
```

3.2.3 Train Model

Set Up Training Parameters

Set the training parameters for our model. All of the parameters below are optional.

[11]:
```
train_params = {}
train_params['save_weights_path'] =  None # Where to save the model after training.␣
↪Default: None
train_params['training_iters'] = 100000 # number of iterations to train for Default:␣
↪50000
train_params['learning_rate'] = .001 # Sets learning rate if use default optimizer␣
↪Default: .001
train_params['loss_epoch'] = 10 # Compute and record loss every 'loss_epoch' epochs.␣
↪Default: 10
train_params['verbosity'] = False # If true, prints information as training␣
↪progresses. Default: True
train_params['save_training_weights_epoch'] = 100 # save training weights every 'save_
↪training_weights_epoch' epochs. Default: 100
train_params['training_weights_path'] = None # where to save training weights as␣
↪training progresses. Default: None
train_params['optimizer'] = tf.compat.v1.train.AdamOptimizer(learning_rate=train_
↪params['learning_rate']) # What optimizer to use to compute gradients. Default: tf.
↪train.AdamOptimizer(learning_rate=train_params['learning_rate'])
train_params['clip_grads'] = True # If true, clip gradients by norm 1. Default: True
```

Example usage of the optional fixed_weights parameter is available in the *Biological Constraints* tutorial

[12]:
```
train_params['fixed_weights'] = None # Dictionary of weights to fix (not allow to␣
↪train). Default: None
```

Example usage of the optional performance_cutoff and performance_measure parameters is available in *Curriculum Learning* tutorial.

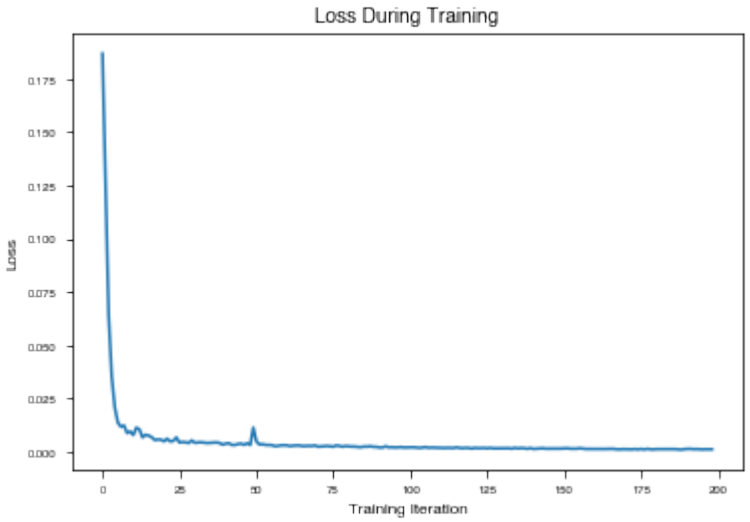
[13]:
```
train_params['performance_cutoff'] = None # If performance_measure is not None,␣
↪training stops as soon as performance_measure surpases the performance_cutoff.␣
↪Default: None.
train_params['performance_measure'] = None # Function to calculate the performance of␣
↪the network using custom criteria. Default: None.]
```

Train Model on Task using Training Parameters

[14]:
```
losses, initialTime, trainTime = basicModel.train(pd, train_params)
```

[15]:
```
plt.plot(losses)
plt.ylabel("Loss")
plt.xlabel("Training Iteration")
plt.title("Loss During Training")
```

[15]:
```
Text(0.5, 1.0, 'Loss During Training')
```



3.2.4 Test Model

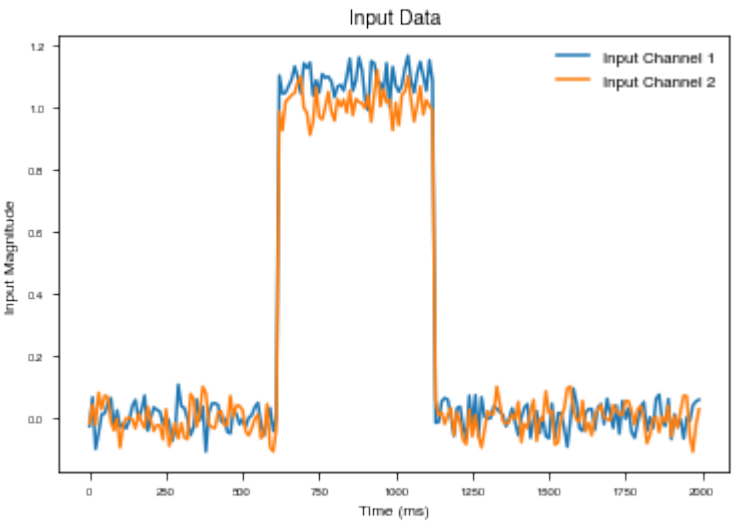
Get a batch of trials from the task to test the network on.

[16]:
```
x,y,m, _ = pd.get_trial_batch()
```

Plot the x value of the trial – for the PerceptualDiscrimination, this includes two input neurons with different coherence.

[17]:
```
plt.plot(range(0, len(x[0,:,:])*dt,dt), x[0,:,:])
plt.ylabel("Input Magnitude")
plt.xlabel("Time (ms)")
plt.title("Input Data")
plt.legend(["Input Channel 1", "Input Channel 2"])
```

[17]:
```
<matplotlib.legend.Legend at 0x7fcb537509b0>
```

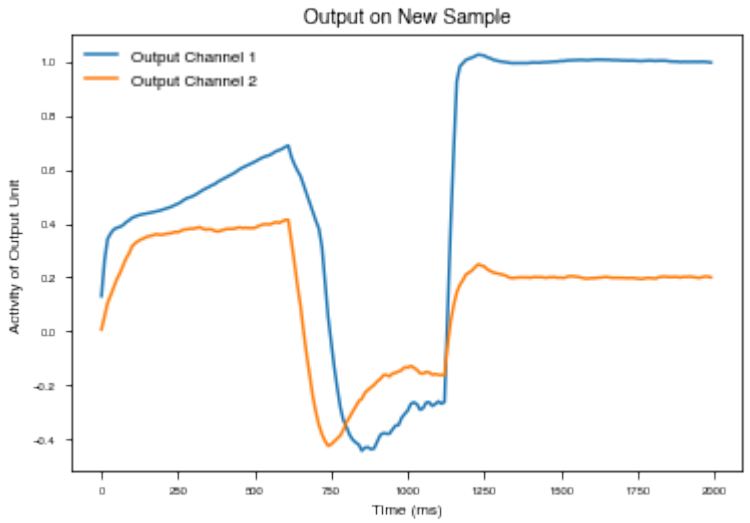


Run the trained model on this trial (not included in the training set).

[18]:
```
output, state_var = basicModel.test(x)
```

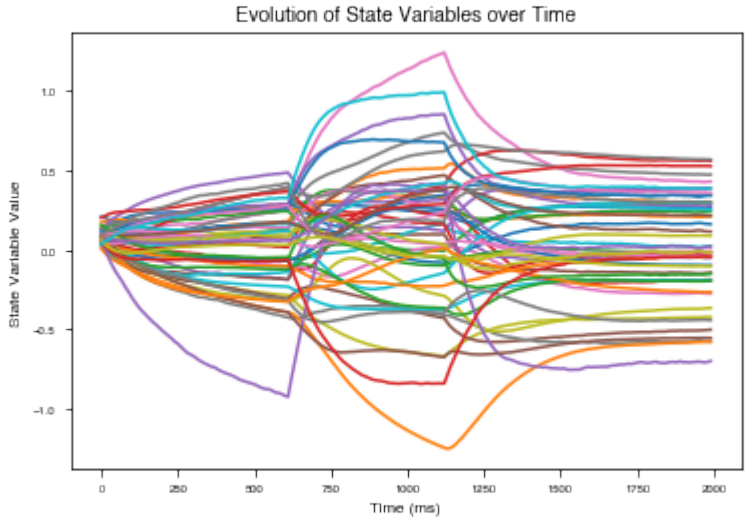
[19]:
```
plt.plot(range(0, len(output[0,:,:])*dt,dt),output[0,:,:])
plt.ylabel("Activity of Output Unit")
plt.xlabel("Time (ms)")
plt.title("Output on New Sample")
plt.legend(["Output Channel 1", "Output Channel 2"])
```

[19]:
```
<matplotlib.legend.Legend at 0x7fcb53704198>
```



[20]:
```
plt.plot(range(0, len(state_var[0,:,:])*dt,dt),state_var[0,:,:])
plt.ylabel("State Variable Value")
plt.xlabel("Time (ms)")
plt.title("Evolution of State Variables over Time")
```

[20]:
```
Text(0.5, 1.0, 'Evolution of State Variables over Time')
```



3.2.5 Get & Save Model Weights

We can get the weights used by the model in dictionary form using *get_weights*, or we can save the weights directly to a file using *save*.

[21]:
```
weights = basicModel.get_weights()

print(weights.keys())
```

```
dict_keys(['init_state', 'W_in', 'W_rec', 'W_out', 'b_rec', 'b_out', 'Dale_rec',
↪'Dale_out', 'input_connectivity', 'rec_connectivity', 'output_connectivity', 'init_
↪state/Adam', 'init_state/Adam_1', 'W_in/Adam', 'W_in/Adam_1', 'W_rec/Adam', 'W_rec/
↪Adam_1', 'W_out/Adam', 'W_out/Adam_1', 'b_rec/Adam', 'b_rec/Adam_1', 'b_out/Adam',
↪'b_out/Adam_1', 'dale_ratio'])
```

[22]:
```
basicModel.save("./weights/saved_weights")
```

3.2.6 Cleanup

Clean up the model to clear out the tensorflow namespace

[23]:
```
basicModel.destruct()
```

3.3 Biological Constraints

The default RNN network has all to all connectivity, and allows units to have both excitatory and inhibitory connections. However, this does not reflect the biology we know. PsychRNN includes a framework for easily specifying biological constraints on the model.

This example will introduce the different options for biological constraints included in PsychRNN: - Dale Ratio - Autapses - Connectivity - Fixed Weights

[2]:
```
import numpy as np

from matplotlib import pyplot as plt
from matplotlib.colors import Normalize
%matplotlib inline

# ---------------------- Import the package ---------------------------
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination
from psychrnn.backend.models.basic import Basic

# ---------------------- Set up a basic model ---------------------------
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
network_params = pd.get_task_params() # get the params passed in and defined in pd
network_params['name'] = 'model' # name the model uniquely if running mult models in␣
↪unison
network_params['N_rec'] = 50 # set the number of recurrent units in the model

# --------------------- Set up variables that will be useful later -------
N_in = network_params['N_in']
N_rec = network_params['N_rec']
N_out = network_params['N_out']
```

This function will plot the colormap of the weights

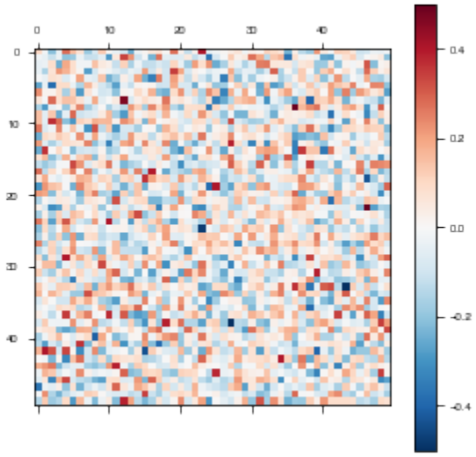
[3]:
```
def plot_weights(weights, title=""):
    cmap = plt.set_cmap('RdBu_r')
    img = plt.matshow(weights, norm=Normalize(vmin=-.5, vmax=.5))
    plt.title(title)

    plt.colorbar()
```

3.3.1 Biologically Unconstrained

[4]:
```
basicModel = Basic(network_params) # instantiate a basic vanilla RNN we will compare␣
↪to later on
```

[5]:
```
weights = basicModel.get_weights()
plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[6]:
```
basicModel.destruct()
```

3.3.2 Dale Ratio

Dale's Principle states that a neuron releases the same set of neurotransmitters at each of its synapses (Eccles et al., 1954). Since neurotransmitters tend to be either excitatory or inhibitory, theorists have taken this to mean that each neuron has exclusively either excitatory or inhibitory synapses (Song et al., 2016; Rajan and Abbott, 2006).

To set the dale ratio, simply set `network_params['dale_ratio']` equal to the proportion of total recurrent neurons that should be excitatory. The remainder will be inhibitory.

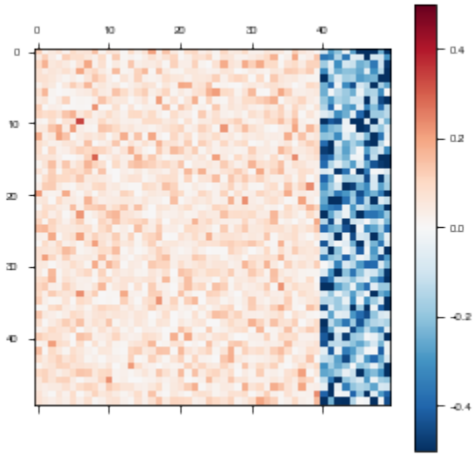
The dale ratio can be combined with any other parameter settings except for `network_params['initializer']`, in which case the dale ratio needs to be passed directly into the *initializer* being used. Dale ratio is not enforced if *LSTM* is used as the RNN imlementation.

Once the model is instantiated it can be trained and tested as demonstrated in *Simple Example*

[7]:
```
dale_network_params = network_params.copy()
dale_network_params['name'] = 'dales_model'
dale_network_params['dale_ratio'] = .8
daleModel = Basic(dale_network_params)
```

[8]:
```
weights = daleModel.get_weights()
plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[9]:
```
daleModel.destruct()
```

3.3.3 Autapses

To disallow autapses (self connections) or not, simply set `network_params['autapses'] = False`.

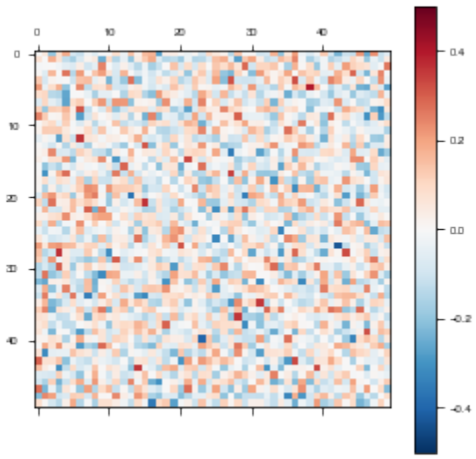
The autapses parameter can be combined with any other parameter settings except for `network_params['initializer']`, in which case the boolean for autapses needs to be passed directly into the *initializer* being used. Autapses are not enforced if *LSTM* is used as the RNN imlementation.

Once the model is instantiated it can be trained and tested as demonstrated in *Simple Example*

[10]:
```
autapses_network_params = network_params.copy()
autapses_network_params['name'] = 'autapses_model'
autapses_network_params['autapses'] = False
autapsesModel = Basic(autapses_network_params)
```

[11]:
```
weights = autapsesModel.get_weights()
plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



Notice the white line on the diagonal (self-connections) above, where the weights are 0.

[12]:
```
autapsesModel.destruct()
```

3.3.4 Connectivity

The brain is not all-to-all connected, so it can be useful to restrict and structure the connectivity of our RNNs.

The input_connectivity, recurrent_connectivity, and output_connectivity parameters allow us to do just that. Any subset of them can be combined with any other parameter settings except for `network_params['initializer']`, in which case the connectivity matrices need to be passed directly into the *initializer* being used. Connectivity is not enforced if *LSTM* is used as the RNN imlementation.

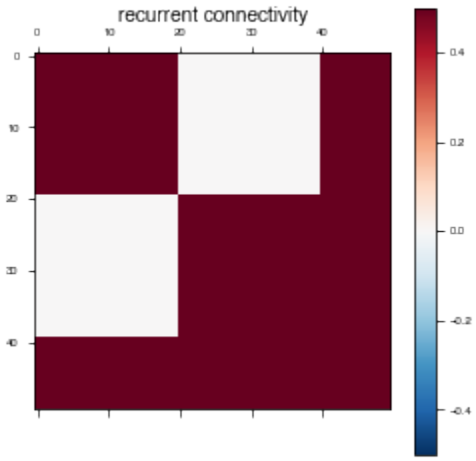
Once the model is instantiated it can be trained and tested as demonstrated in *Simple Example*

[13]:
```
modular_network_params = network_params.copy()
modular_network_params['name'] = 'modular_model'

# Set connectivity matrices to the default -- fully connected
input_connectivity = np.ones((N_rec, N_in))
rec_connectivity = np.ones((N_rec, N_rec))
output_connectivity = np.ones((N_out, N_rec))

# Specify certain connections to disallow. This can be done with input and output␣
↪connectivity matrices as well
rec_connectivity[2*(N_rec//5):4*(N_rec//5),:2*(N_rec//5)] = 0
rec_connectivity[:2*(N_rec//5),2*(N_rec//5):4*(N_rec//5)] = 0
```

Plot the recurrent connectivity matrix

[14]:
```
plot_weights(rec_connectivity, "recurrent connectivity")
```

```
<Figure size 432x288 with 0 Axes>
```

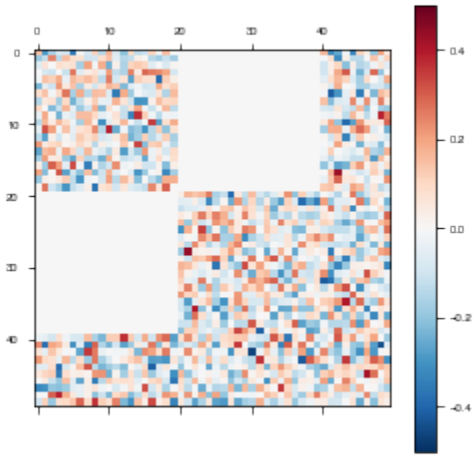


Specify the connectivity matrices in `network_params`.

[15]:
```
modular_network_params['input_connectivity'] = input_connectivity
modular_network_params['rec_connectivity'] = rec_connectivity
modular_network_params['output_connectivity'] = output_connectivity
modularModel = Basic(modular_network_params)
```

[16]:
```
weights = modularModel.get_weights()

plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[17]:
```
modularModel.destruct()
```

3.3.5 Fixed Weights

Some parts of the brain we may assume to be less plastic than others. Alternatively, we may want to specify particular weights within the model and train the rest of them around those.

The fixed_weights parameter for the *train()* fucntion allows us to do this.

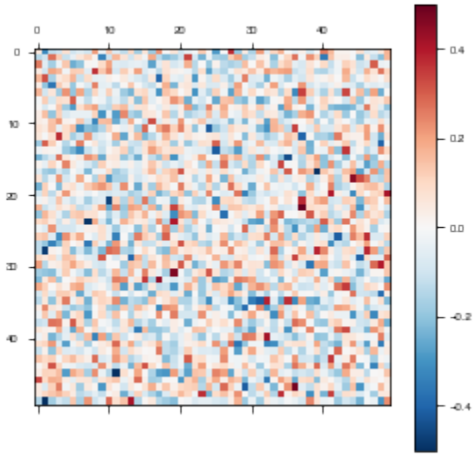
Instantiate the model

[18]:
```
fixed_network_params = network_params.copy()
fixed_network_params['name'] = 'fixed_model'
fixedModel = Basic(fixed_network_params) # instantiate a basic vanilla RNN we will␣
↪compare to later on
```

Plot the model weights before training

[19]:
```
weights = fixedModel.get_weights()

plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[20]:
```
# Set fixed weight matrices to the default -- fully trainable
W_in_fixed = np.zeros((N_rec,N_in))
W_rec_fixed = np.zeros((N_rec,N_rec))
W_out_fixed = np.zeros((N_out, N_rec))

# Specify certain weights to fix.
W_rec_fixed[N_rec//5*4:, :4*N_rec//5] = 1
W_rec_fixed[:4*N_rec//5, N_rec//5*4:] = 1


# Specify the fixed weights parameters in train_params
train_params = {}
train_params['fixed_weights'] = {
    'W_in': W_in_fixed,
    'W_rec': W_rec_fixed,
    'W_out': W_out_fixed
}
```

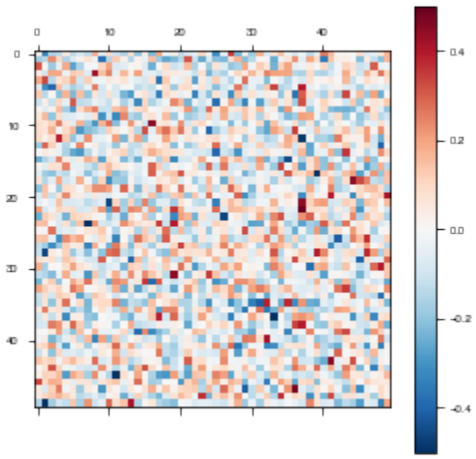
[21]:
```
losses, initialTime, trainTime = fixedModel.train(pd, train_params)
```

```
Iter 1280, Minibatch Loss= 0.177185
Iter 2560, Minibatch Loss= 0.107636
Iter 3840, Minibatch Loss= 0.099301
Iter 5120, Minibatch Loss= 0.085224
Iter 6400, Minibatch Loss= 0.082593
Iter 7680, Minibatch Loss= 0.079836
Iter 8960, Minibatch Loss= 0.080765
Iter 10240, Minibatch Loss= 0.079680
Iter 11520, Minibatch Loss= 0.072564
Iter 12800, Minibatch Loss= 0.067365
Iter 14080, Minibatch Loss= 0.040751
Iter 15360, Minibatch Loss= 0.052333
Iter 16640, Minibatch Loss= 0.046463
Iter 17920, Minibatch Loss= 0.031513
Iter 19200, Minibatch Loss= 0.033700
Iter 20480, Minibatch Loss= 0.033375
Iter 21760, Minibatch Loss= 0.035751
Iter 23040, Minibatch Loss= 0.041844
Iter 24320, Minibatch Loss= 0.038133
Iter 25600, Minibatch Loss= 0.023348
Iter 26880, Minibatch Loss= 0.027589
Iter 28160, Minibatch Loss= 0.019354
Iter 29440, Minibatch Loss= 0.022398
Iter 30720, Minibatch Loss= 0.020543
Iter 32000, Minibatch Loss= 0.013847
Iter 33280, Minibatch Loss= 0.017195
Iter 34560, Minibatch Loss= 0.019519
Iter 35840, Minibatch Loss= 0.020920
Iter 37120, Minibatch Loss= 0.016392
Iter 38400, Minibatch Loss= 0.019325
Iter 39680, Minibatch Loss= 0.015266
Iter 40960, Minibatch Loss= 0.031248
Iter 42240, Minibatch Loss= 0.023118
Iter 43520, Minibatch Loss= 0.015399
Iter 44800, Minibatch Loss= 0.018544
Iter 46080, Minibatch Loss= 0.021445
Iter 47360, Minibatch Loss= 0.012260
Iter 48640, Minibatch Loss= 0.017937
Iter 49920, Minibatch Loss= 0.020652
Optimization finished!
```

Plot the weights after training:

[22]:
```
weights = fixedModel.get_weights()

plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[23]:
```
fixedModel.destruct()
```

Unfortunately, it's hard to see visually whether the weights actually stayed fixed or not. To make it more apparent, we will set all of the fixed weights to the same value, the average of their previous value.

[24]:
```
weights['W_rec'][N_rec//5*4:, :4*N_rec//5] = np.mean(weights['W_rec'][N_rec//5*4:, :
↪4*N_rec//5])
weights['W_rec'][:4*N_rec//5, N_rec//5*4:] = np.mean(weights['W_rec'][:4*N_rec//5, N_
↪rec//5*4:])
```

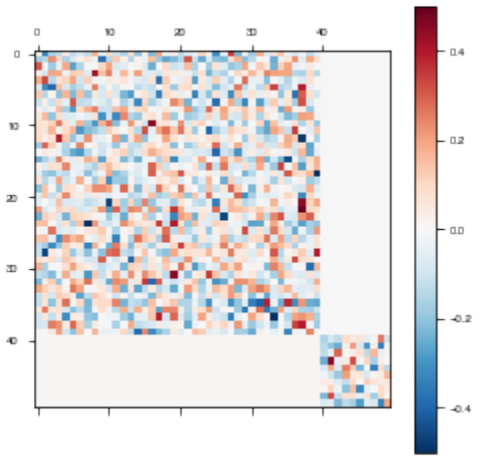
Now we make a new model loading the weights `weights`

[25]:
```
fixed_network_params = network_params.copy()
fixed_network_params['name'] = 'fixed_model_clearer'
for key, value in weights.items():
    fixed_network_params[key] = value
fixedModelClearer = Basic(fixed_network_params) # instantiate an RNN loading the␣
↪revised weights from the previous model
```

Plot the model weights before training

[26]:
```
weights = fixedModelClearer.get_weights()

plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[27]:
```
losses, initialTime, trainTime = fixedModelClearer.train(pd, train_params)
```

```
Iter 1280, Minibatch Loss= 0.050554
Iter 2560, Minibatch Loss= 0.024552
Iter 3840, Minibatch Loss= 0.021128
Iter 5120, Minibatch Loss= 0.028251
Iter 6400, Minibatch Loss= 0.019927
Iter 7680, Minibatch Loss= 0.016723
Iter 8960, Minibatch Loss= 0.013385
Iter 10240, Minibatch Loss= 0.016600
Iter 11520, Minibatch Loss= 0.020957
Iter 12800, Minibatch Loss= 0.012375
Iter 14080, Minibatch Loss= 0.019829
Iter 15360, Minibatch Loss= 0.020301
Iter 16640, Minibatch Loss= 0.019600
Iter 17920, Minibatch Loss= 0.017423
Iter 19200, Minibatch Loss= 0.010484
Iter 20480, Minibatch Loss= 0.014385
Iter 21760, Minibatch Loss= 0.017793
Iter 23040, Minibatch Loss= 0.009582
Iter 24320, Minibatch Loss= 0.014552
Iter 25600, Minibatch Loss= 0.010809
Iter 26880, Minibatch Loss= 0.012337
Iter 28160, Minibatch Loss= 0.017401
Iter 29440, Minibatch Loss= 0.012895
Iter 30720, Minibatch Loss= 0.016758
Iter 32000, Minibatch Loss= 0.011036
Iter 33280, Minibatch Loss= 0.007268
Iter 34560, Minibatch Loss= 0.008717
Iter 35840, Minibatch Loss= 0.014370
Iter 37120, Minibatch Loss= 0.012818
Iter 38400, Minibatch Loss= 0.021543
Iter 39680, Minibatch Loss= 0.011174
Iter 40960, Minibatch Loss= 0.010043
Iter 42240, Minibatch Loss= 0.015098
Iter 43520, Minibatch Loss= 0.012391
Iter 44800, Minibatch Loss= 0.011706
Iter 46080, Minibatch Loss= 0.015107
Iter 47360, Minibatch Loss= 0.012814
Iter 48640, Minibatch Loss= 0.009676
```

(continues on next page)

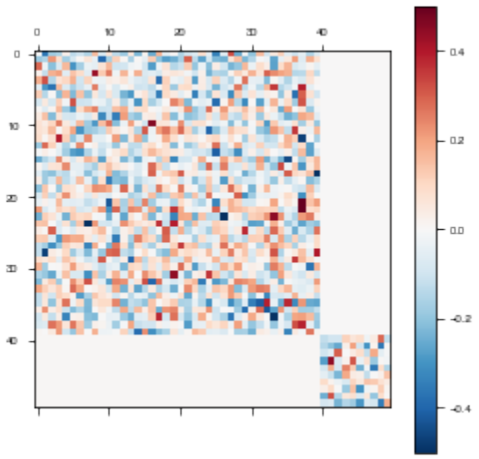
(continued from previous page)

```
Iter 49920, Minibatch Loss= 0.009720
Optimization finished!
```

Plot the model weights after training. Now it is clear that the weights haven't changed.

[28]:
```
weights = fixedModelClearer.get_weights()

plot_weights(weights['W_rec'])
```

```
<Figure size 432x288 with 0 Axes>
```



[29]:
```
fixedModelClearer.destruct()
```

3.4 Curriculum Learning

[2]:
```
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination
from psychrnn.backend.models.basic import Basic
from psychrnn.backend.curriculum import Curriculum, default_metric
import numpy as np

from matplotlib import pyplot as plt
%matplotlib inline
```

3.4.1 Instantiate Curriculum Object

We generate a list of tasks that constitute our curriculum. We will train on these tasks one after another. In this example, we train the network on tasks with higher coherence, slowly decreasing to lower coherence.

[3]:
```
pds = [PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 50, coherence␣
↪= .7 - i/5) for i in range(4)]
```

Set optional parameters for the curriculum object. More information about these parameters is available *here*.

[4]:
```
metric = default_metric # Function for calculating whether the stage advances and␣
↪what the metric value is at each metric_epoch. Default: default_metric().
accuracies = [pds[i].accuracy_function for i in range(len(pds))] # optional list of␣
↪functions to use to calculate network performance for the purposes of advancing␣
↪tasks. Used by default_metric() to compute accuracy. Default: [tasks[i].accuracy_␣
↪function for i in range(len(tasks))].
thresholds = [.9 for i in range(len(pds))] # Optional list of thresholds. If metric =␣
↪default_metric, accuracies must reach the threshold for a given stage in order to␣
↪advance to the next stage. Default: [.9 for i in range(len(tasks))]
metric_epoch = 1 # calculate the metric / test if advance to the next stage every␣
↪metric_epoch training epochs.
output_file = None # Optional path to save out metric value and stage to. Default:␣
↪None.
```

Initialize a curriculum object with information about the tasks we want to train on.

[5]:
```
curriculum = Curriculum(pds, output_file=output_file, metric_epoch=metric_epoch,␣
↪thresholds=thresholds, accuracies=accuracies, metric=metric)
```

3.4.2 Initialize Models

We add in a few params that Basic(RNN) needs but that PerceptualDiscrimination doesn't generate for us.

[6]:
```
network_params = pds[0].get_task_params()
network_params['name'] = 'curriculumModel' #Used to scope out a namespace for global␣
↪variables.
network_params['N_rec'] = 50
```

Instantiate two models. *curriculumModel* that will be trained on the series of tasks, pds, defined above. *basicModel* will be trained only on the final task with lowest coherence.

[7]:
```
curriculumModel = Basic(network_params)
network_params['name'] = 'basicModel'
basicModel = Basic(network_params)
```

3.4.3 Train Models

Set the training parameters for our model to include curriculum. The other training parameters shown in *Simple Example* can also be included.

[8]:
```
train_params = {}
train_params['curriculum'] = curriculum
```

We will train the curriculum model using *train_curric()* which is a wrapper for *train* that does't require a task to be passed in outside of the curriculum entry in train_params.

[9]:
```
curric_losses, initialTime, trainTime = curriculumModel.train_curric(train_params)
```

```
Accuracy: 0.6
Accuracy: 0.62
Accuracy: 0.48
Accuracy: 0.48
Accuracy: 0.44
Accuracy: 0.44
```

(continues on next page)

(continued from previous page)

```
Accuracy: 0.42
Accuracy: 0.5
Accuracy: 0.5
Iter 500, Minibatch Loss= 0.180899
Accuracy: 0.62
Accuracy: 0.46
Accuracy: 0.52
Accuracy: 0.46
Accuracy: 0.52
Accuracy: 0.6
Accuracy: 0.38
Accuracy: 0.6
Accuracy: 0.6
Accuracy: 0.4
Iter 1000, Minibatch Loss= 0.114158
Accuracy: 0.48
Accuracy: 0.44
Accuracy: 0.5
Accuracy: 0.5
Accuracy: 0.58
Accuracy: 0.98
Stage 1
Accuracy: 1.0
Stage 2
Accuracy: 1.0
Stage 3
Accuracy: 0.92
Stage 4
Optimization finished!
```

Set training parameters for the non-curriculum model. We use performance_measure and cutoff so that the model trains until it 90% accurate on the hardest task, just like the curriculum model does. This will give us a more fair comparison when we look at losses and training time

[10]:
```
def performance_measure(trial_batch, trial_y, output_mask, output, epoch, losses,␣
↪verbosity):
    return pds[len(pds)-1].accuracy_function(trial_y, output, output_mask)

train_params['curriculum'] = None
train_params['performance_measure'] = performance_measure
train_params['performance_cutoff'] = .9
```

Train the non-curriculum model.

[11]:
```
basic_losses, initialTime, trainTime= basicModel.train(pds[len(pds)-1], train_params)
```

```
performance: 0.54
performance: 0.6
performance: 0.42
performance: 0.54
performance: 0.26
performance: 0.24
performance: 0.58
performance: 0.42
performance: 0.52
Iter 500, Minibatch Loss= 0.102338
performance: 0.56
```

(continues on next page)

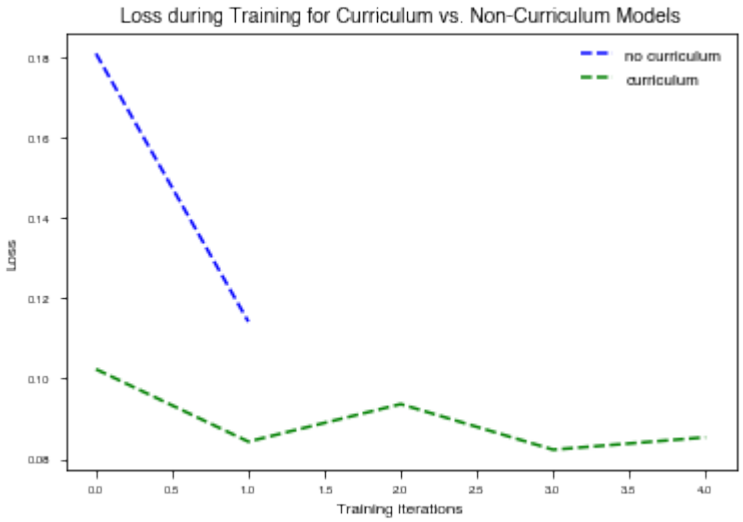
(continued from previous page)

```
performance: 0.56
performance: 0.46
performance: 0.48
performance: 0.56
performance: 0.54
performance: 0.52
performance: 0.5
performance: 0.54
performance: 0.56
Iter 1000, Minibatch Loss= 0.084302
performance: 0.4
performance: 0.48
performance: 0.52
performance: 0.44
performance: 0.46
performance: 0.5
performance: 0.64
performance: 0.38
performance: 0.52
performance: 0.56
Iter 1500, Minibatch Loss= 0.093645
performance: 0.44
performance: 0.5
performance: 0.46
performance: 0.5
performance: 0.4
performance: 0.5
performance: 0.46
performance: 0.6
performance: 0.6
performance: 0.56
Iter 2000, Minibatch Loss= 0.082302
performance: 0.58
performance: 0.4
performance: 0.46
performance: 0.5
performance: 0.46
performance: 0.54
performance: 0.62
performance: 0.46
performance: 0.42
performance: 0.56
Iter 2500, Minibatch Loss= 0.085385
performance: 0.56
performance: 0.44
performance: 0.5
performance: 0.52
performance: 0.36
performance: 0.42
performance: 0.56
performance: 0.7
performance: 0.96
Optimization finished!
```

Plot Losses

Plot the losses from curriculum and non curriculum training.

[12]:
```
plt.plot( curric_losses, 'b--', label = 'no curriculum')
plt.plot(basic_losses, 'g--', label='curriculum')
plt.legend()
plt.title("Loss during Training for Curriculum vs. Non-Curriculum Models")
plt.ylabel('Loss')
plt.xlabel('Training iterations')
plt.show()
```



3.4.4 Cleanup

[13]:
```
basicModel.destruct()
```

[14]:
```
curriculumModel.destruct()
```

3.5 Accessing and Modifying Weights

In *Simple Example*, we saved weights to `./weights/saved_weights`. Here we will load those weights, and modify them by silencing a few recurrent units.

[2]:
```
import numpy as np
weights = dict(np.load('./weights/saved_weights.npz', allow_pickle = True))
weights['W_rec'][:10, :10] = 0
```

Here are all the different weights you have access to for modifying. The ones that don't end in `Adam` or `Adam_1` will be read in when loading a model from weights.

[3]:
```
print(weights.keys())
```

```
dict_keys(['init_state', 'W_in', 'W_rec', 'W_out', 'b_rec', 'b_out', 'Dale_rec',
↪'Dale_out', 'input_connectivity', 'rec_connectivity', 'output_connectivity', 'init_
↪state/Adam', 'init_state/Adam_1', 'W_in/Adam', 'W_in/Adam_1', 'W_rec/Adam', 'W_rec/
↪Adam_1', 'W_out/Adam', 'W_out/Adam_1', 'b_rec/Adam', 'b_rec/Adam_1', 'b_out/Adam',
↪'b_out/Adam_1', 'dale_ratio'])
```

(continues on next page)

(continued from previous page)

Save the modified weights at `'./weights/modified_saved_weights.npz'`.

[4]:
```
np.savez('./weights/modified_saved_weights.npz', **weights)
```

3.6 Loading Model with Weights

[5]:
```
from psychrnn.backend.models.basic import Basic
```

[6]:
```
network_params = {'N_batch': 50,
                  'N_in': 2,
                  'N_out': 2,
                  'dt': 10,
                  'tau': 100,
                  'T': 2000,
                  'N_steps': 200,
                  'N_rec': 50
                 }
```

3.6.1 Load from File

Set network parameters.

[7]:
```
file_network_params = network_params.copy()
file_network_params['name'] = 'file'
file_network_params['load_weights_path'] = './weights/modified_saved_weights.npz'
```

Instantiate model.

[8]:
```
fileModel = Basic(file_network_params)
```

Verify that the W_rec weights are modified as expected.

[9]:
```
print(fileModel.get_weights()['W_rec'][:10,:10])
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

[10]:
```
fileModel.destruct()
```

3.6.2 Load from Weights Dictionary

Set network parameters.

[11]:
```
dict_network_params = network_params.copy()
dict_network_params['name'] = 'dict'
dict_network_params.update(weights)
type(dict_network_params['dale_ratio']) == np.ndarray and dict_network_params['dale_
↪ratio'].item() is None
```

[11]:
```
True
```

Instantiate model.

[12]:
```
dictModel = Basic(dict_network_params)
```

Verify that the W_rec weights are modified as expected.

[13]:
```
print(dictModel.get_weights()['W_rec'][:10,:10])
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

[14]:
```
dictModel.destruct()
```

3.7 Simulation in NumPy

Simulator has NumPy implementations of the included models. Once the model is trained, experiments can be done entirely in NumPy without any reliance on TensorFlow, giving full control to researchers.

There may be some floating point error differences between NumPy and TensorFlow implementations – these grow the more timepoints the model is run on, but shouldn't cause major issues.

Here we will demonstrate training a simple model in tensorflow, and then loading it and simulating it in NumPy.

The Simulator can be loaded either directly from a model, from saved weights in a file, or from a dictionary of weights. All options will be shown below.

[2]:
```
from psychrnn.backend.models.basic import Basic
from psychrnn.backend.simulation import BasicSimulator
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination

import numpy as np

from matplotlib import pyplot as plt
%matplotlib inline
```

3.7.1 Load from Model

To load from a model we first need to have a model. Here we instantiate a basic model from the weights saved out by *Simple Example*.

[3]:
```
network_params = {'N_batch': 50,
                  'N_in': 2,
                  'N_out': 2,
                  'dt': 10,
                  'tau': 100,
                  'T': 2000,
                  'N_steps': 200,
                  'N_rec': 50,
                  'name': 'Basic',
                  'load_weights_path': './weights/saved_weights.npz'
                 }
tf_model = Basic(network_params)
```

Instantiate the simulator from the model. Because the model was originally trained as a *Basic* model, we will use *BasicSimulator* to simulate the model.

[4]:
```
simulator = BasicSimulator(rnn_model = tf_model)
```

Instantiate task to run the simulator on:

[5]:
```
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
```

Simulate Model

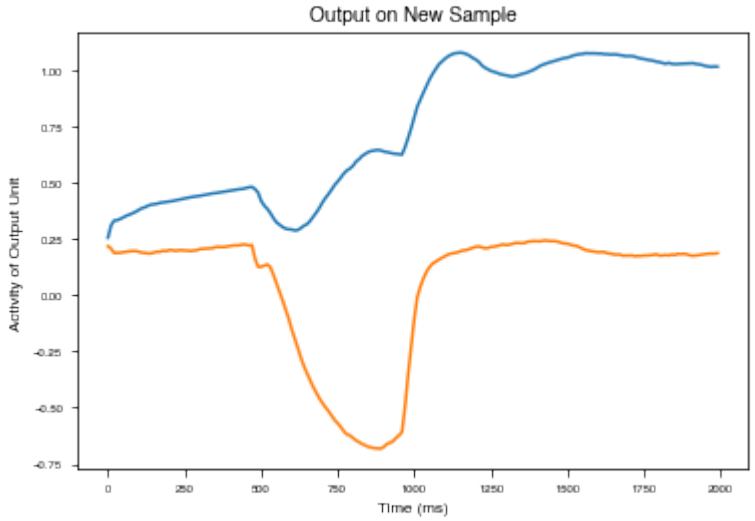
Simulate *tf_model.test()* using the simulator's NumPy impelmentation, *simulator.run_trials(x)*.

[6]:
```
x, y, mask, _ = pd.get_trial_batch()
outputs, states = simulator.run_trials(x)
```

We can plot the results form the simulated model much as we could plot the results from the model in *Simple Example*.

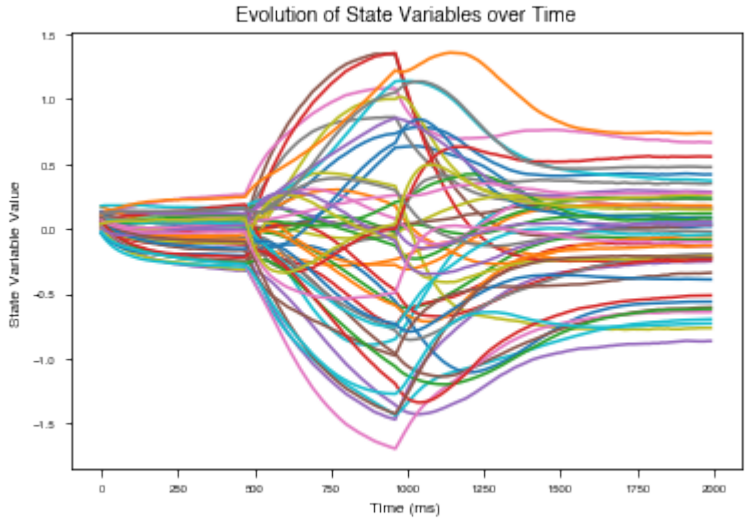
[7]:
```
plt.plot(range(0, len(outputs[0,:,:])*10,10),outputs[0,:,:])
plt.ylabel("Activity of Output Unit")
plt.xlabel("Time (ms)")
plt.title("Output on New Sample")
```

[7]:
```
Text(0.5, 1.0, 'Output on New Sample')
```



[8]:
```
plt.plot(range(0, len(states[0,:,:])*10,10),states[0,:,:])
plt.ylabel("State Variable Value")
plt.xlabel("Time (ms)")
plt.title("Evolution of State Variables over Time")
```

[8]:
```
Text(0.5, 1.0, 'Evolution of State Variables over Time')
```



[9]:
```
tf_model.destruct()
```

3.7.2 Load from File

Instantiate the simulator from the weights saved to file. Because the model was originally trained as a *Basic* model, we will use *BasicSimulator* to simulate the model.

[10]:
```
simulator = BasicSimulator(weights_path='./weights/saved_weights.npz', params = {'dt':
↪ 10, 'tau': 100})
```

Instantiate task to run the simulator on:

[11]:
```
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
```

Simulate Model

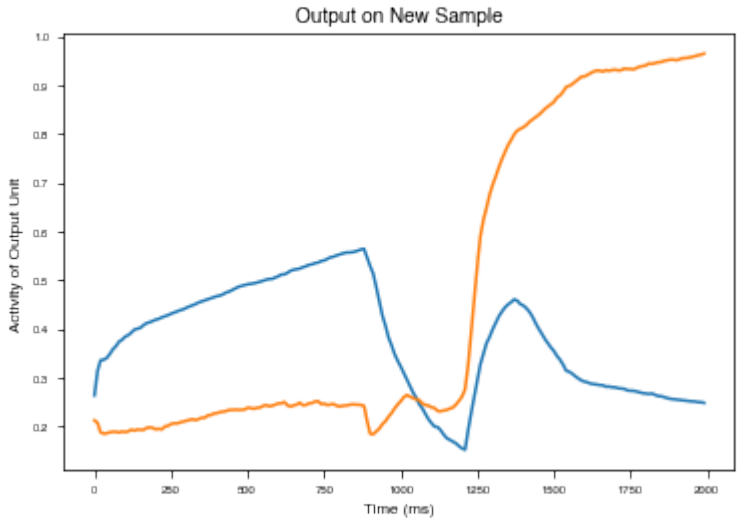
Simulate *tf_model.test()* using the simulator's NumPy impelmentation, *simulator.run_trials(x)*.

[12]:
```
x, y, mask, _ = pd.get_trial_batch()
outputs, states = simulator.run_trials(x)
```

We can plot the results form the simulated model much as we could plot the results from the model in *Simple Example*.

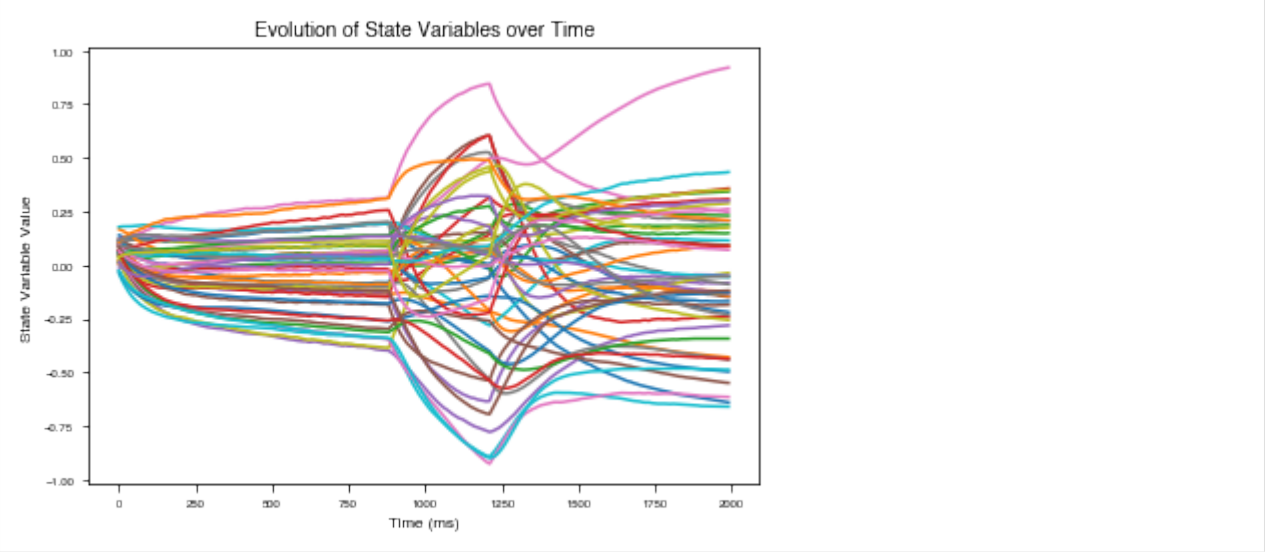
[13]:
```
plt.plot(range(0, len(outputs[0,:,:])*10,10),outputs[0,:,:])
plt.ylabel("Activity of Output Unit")
plt.xlabel("Time (ms)")
plt.title("Output on New Sample")
```

[13]:
```
Text(0.5, 1.0, 'Output on New Sample')
```



[14]:
```
plt.plot(range(0, len(states[0,:,:])*10,10),states[0,:,:])
plt.ylabel("State Variable Value")
plt.xlabel("Time (ms)")
plt.title("Evolution of State Variables over Time")
```

[14]:
```
Text(0.5, 1.0, 'Evolution of State Variables over Time')
```



3.7.3 Load from Dictionary

Instantiate the simulator from a dictionary of weights. Because the model was originally trained as a *Basic* model, we will use *BasicSimulator* to simulate the model.

[15]:
```
weights = dict(np.load('./weights/saved_weights.npz', allow_pickle = True))
simulator = BasicSimulator(weights = weights , params = {'dt': 10, 'tau': 100})
```

Instantiate task to run the simulator on:

[16]:
```
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
```

Simulate Model

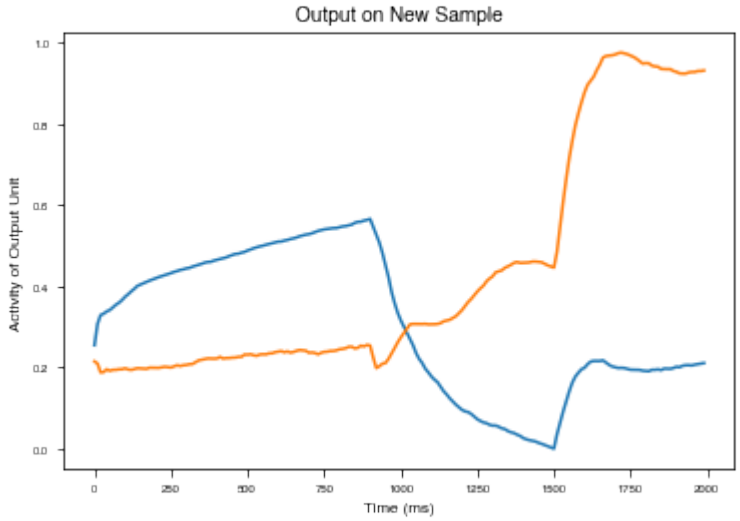
Simulate *tf_model.test()* using the simulator's NumPy impelmentation, *simulator.run_trials(x)*.

[17]:
```
x, y, mask, _ = pd.get_trial_batch()
outputs, states = simulator.run_trials(x)
```

We can plot the results form the simulated model much as we could plot the results from the model in *Simple Example*.

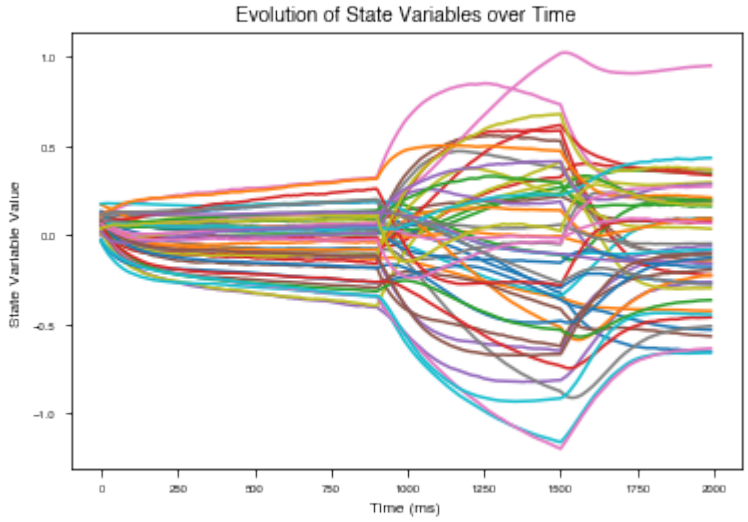
[18]:
```
plt.plot(range(0, len(outputs[0,:,:])*10,10),outputs[0,:,:])
plt.ylabel("Activity of Output Unit")
plt.xlabel("Time (ms)")
plt.title("Output on New Sample")
```

[18]:
```
Text(0.5, 1.0, 'Output on New Sample')
```



[19]:
```
plt.plot(range(0, len(states[0,:,:])*10,10),states[0,:,:])
plt.ylabel("State Variable Value")
plt.xlabel("Time (ms)")
plt.title("Evolution of State Variables over Time")
```

[19]:
```
Text(0.5, 1.0, 'Evolution of State Variables over Time')
```



[1]:
```
# THIS CELL SETS STUFF UP FOR DEMO / COLLAB. THIS CELL CAN BE IGNORED.

#-----------------------------------GET RID OF TF DEPRECATION WARNINGS-------------
↪------------------------#
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

#---------------------------------INSTALL PSYCHRNN IF IN A COLAB NOTEBOOK------------
↪------------------------#
```

(continues on next page)

(continued from previous page)

```
# Installs the correct branch / release version based on the URL. If no branch is␣
↪provided, loads from master.
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False

if IN_COLAB:
    import json
    import re
    import ipykernel
    import requests
    from requests.compat import urljoin

    from notebook.notebookapp import list_running_servers
    kernel_id = re.search('kernel-(.*).json',
                          ipykernel.connect.get_connection_file()).group(1)
    servers = list_running_servers()
    for ss in servers:
        response = requests.get(urljoin(ss['url'], 'api/sessions'),
                                params={'token': ss.get('token', '')})
        for nn in json.loads(response.text):
            if nn['kernel']['id'] == kernel_id:
                relative_path = nn['notebook']['path'].split('%2F')
                if 'blob' in relative_path:
                  blob = relative_path[relative_path.index('blob') + 1]
                  !pip install git+https://github.com/murraylab/PsychRNN@$blob
                else:
                  !pip install git+https://github.com/murraylab/PsychRNN
```

3.8 Define New Task

Below is a sample implentation of a simple perceptual discrimination task. The newly defined task implements and inherits from *Task*. Other examples of tasks are available *here*.

This task will involve two inputs and two outputs. The network must indicate which of the two inputs (directions) has larger signal, and the mean difference in magnitude of the two inputs will be indicated by coherence.

To define the task, `generate_trial_params` and `trial_function` are the two key functions that must be defined.

In this simple task, `generate_trial_params` assigns both the direction and the coherence of the trial randomly.

`trial_function` is given a time point indicating what time in the trial it is currently at as well as the output from `generate_trial_params`. The function intializes the input, `x_t`, with noise, the output, `y_t`, with zeros, and the mask, `mask_t` with ones. During the stimulus period, `x_t` has input signal added to it. During the response period, `y_t[direction]` is set to 1 to indicate the correct direction of the stimulus. Before and during the stimlus period, the mask is set to 0 so that when training, the network knows not to care about it's outputs before the response period.

[2]:
```
from psychrnn.tasks.task import Task
import numpy as np
```

(continues on next page)

(continued from previous page)

```
class SimplePD(Task):
    def __init__(self, dt, tau, T, N_batch):
        super(SimplePDM, self).__init__(2, 2, dt, tau, T, N_batch)

    def generate_trial_params(self, batch, trial):
        """Define parameters for each trial.

        Using a combination of randomness, presets, and task attributes, define the␣
↪necessary trial parameters.

        Args:
            batch (int): The batch number that this trial is part of.
            trial (int): The trial number of the trial within the batch.

        Returns:
            dict: Dictionary of trial parameters.

        """

        # ----------------------------------
        # Define parameters of a trial
        # ----------------------------------
        params = dict()
        params['coherence'] = np.random.exponential(scale=1/5)
        params['direction'] = np.random.choice([0, 1])

        return params

    def trial_function(self, time, params):
        """ Compute the trial properties at the given time.

        Based on the params compute the trial stimulus (x_t), correct output (y_t),␣
↪and mask (mask_t) at the given time.

        Args:
            time (int): The time within the trial (0 <= time < T).
            params (dict): The trial params produced generate_trial_params()

        Returns:
            tuple:

            x_t (ndarray(dtype=float, shape=(N_in,))): Trial input at time given␣
↪params.
            y_t (ndarray(dtype=float, shape=(N_out,))): Correct trial output at time␣
↪given params.
            mask_t (ndarray(dtype=bool, shape=(N_out,))): True if the network should␣
↪train to match the y_t, False if the network should ignore y_t when training.

        """
        stim_noise = 0.1
        onset = self.T/4.0
        stim_dur = self.T/2.0

        # ----------------------------------
        # Initialize with noise
        # ----------------------------------
        x_t = np.sqrt(2*self.alpha*stim_noise*stim_noise)*np.random.randn(self.N_in)
```

(continues on next page)

(continued from previous page)

```
        y_t = np.zeros(self.N_out)
        mask_t = np.ones(self.N_out)

        # ----------------------------------
        # Retrieve parameters
        # ----------------------------------
        coh = params['coherence']
        direction = params['direction']

        # ----------------------------------
        # Compute values
        # ----------------------------------
        if onset < time < onset + stim_dur:
            x_t[direction] += 1 + coh
            x_t[(direction + 1) % 2] += 1

        if time > onset + stim_dur + 20:
            y_t[direction] = 1.

        if time < onset + stim_dur:
            mask_t = np.zeros(self.N_out)

        return x_t, y_t, mask_t
```

Now that the task is defined, we can instantiate it and use it to build a model:

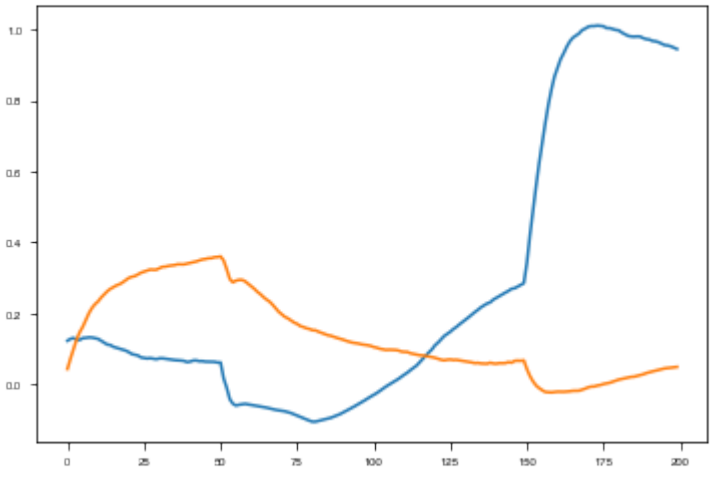
[3]:
```
from matplotlib import pyplot as plt
%matplotlib inline

from psychrnn.backend.models.basic import Basic

# ---------------------- Set up a basic model --------------------------
pd = SimplePD(dt = 10, tau = 100, T = 2000, N_batch = 128)
network_params = pd.get_task_params() # get the params passed in and defined in pd
network_params['name'] = 'model' # name the model uniquely if running mult models in␣
↪unison
network_params['N_rec'] = 50 # set the number of recurrent units in the model
model = Basic(network_params) # instantiate a basic vanilla RNN

# ---------------------- Train a basic model --------------------------
model.train(pd) # train model to perform pd task

# ---------------------- Test the trained model --------------------------
x,target_output,mask, trial_params = pd.get_trial_batch() # get pd task inputs and␣
↪outputs
model_output, model_state = model.test(x) # run the model on input x

# ---------------------- Plot the results --------------------------
plt.plot(model_output[0][0,:,:])

# ---------------------- Teardown the model ------------------------
model.destruct()
```

```
Iter 1280, Minibatch Loss= 0.103770
Iter 2560, Minibatch Loss= 0.069724
Iter 3840, Minibatch Loss= 0.062085
Iter 5120, Minibatch Loss= 0.059507
```

(continues on next page)

(continued from previous page)

```
Iter 6400, Minibatch Loss= 0.055375
Iter 7680, Minibatch Loss= 0.049892
Iter 8960, Minibatch Loss= 0.037656
Iter 10240, Minibatch Loss= 0.023892
Iter 11520, Minibatch Loss= 0.015843
Iter 12800, Minibatch Loss= 0.012522
Iter 14080, Minibatch Loss= 0.011632
Iter 15360, Minibatch Loss= 0.013904
Iter 16640, Minibatch Loss= 0.011842
Iter 17920, Minibatch Loss= 0.009156
Iter 19200, Minibatch Loss= 0.009582
Iter 20480, Minibatch Loss= 0.009885
Iter 21760, Minibatch Loss= 0.007577
Iter 23040, Minibatch Loss= 0.009727
Iter 24320, Minibatch Loss= 0.005300
Iter 25600, Minibatch Loss= 0.008526
Iter 26880, Minibatch Loss= 0.009385
Iter 28160, Minibatch Loss= 0.008682
Iter 29440, Minibatch Loss= 0.005043
Iter 30720, Minibatch Loss= 0.010335
Iter 32000, Minibatch Loss= 0.005916
Iter 33280, Minibatch Loss= 0.007762
Iter 34560, Minibatch Loss= 0.007408
Iter 35840, Minibatch Loss= 0.005352
Iter 37120, Minibatch Loss= 0.005865
Iter 38400, Minibatch Loss= 0.010364
Iter 39680, Minibatch Loss= 0.007844
Iter 40960, Minibatch Loss= 0.006617
Iter 42240, Minibatch Loss= 0.004358
Iter 43520, Minibatch Loss= 0.005139
Iter 44800, Minibatch Loss= 0.006852
Iter 46080, Minibatch Loss= 0.006175
Iter 47360, Minibatch Loss= 0.006748
Iter 48640, Minibatch Loss= 0.006968
Iter 49920, Minibatch Loss= 0.006449
Optimization finished!
```



3.9 Define New Model

Defining a new model requires some familiarity with tensorflow. In this case, we add a feedforward input layer to the *Basic RNN*. Because we have added a new weight matrix, we need to modify our initialization. We then must define what a *forward_pass* for the model looks like.

3.9.1 init

We define the feedforward weight matrix, W_in_first to have `'N_feedforward_out'` outputs and `'N_in'` inputs. We must thus change W_in and input_connectivity to have `'N_feedforward_out'` inputs. Optionally, users can pass in keys for `'N_feedforward_out'` and `'W_in_first_train'` in the params dictionary.

We call `super(RNN, self).__init__` so that all the initialization work done by *RNN* carries over.

All modified or added matrices are put into `initializer.initializations` and initialized in our variable scope.

3.9.2 forward_pass

`forward_pass()` iterates through the network, one timepoint at a time. For each timepoint the output and state is recorded added to an array that will be returned. The output and state are calculated using `recurrent_timestep()` and `output_timestep()`.

output_timestep

`output_timestep()` takes the state and calculates the output. `output_timestep()` is the same as in *Basic RNN*.

recurrent_timestep

`recurrent_timestep()` takes the state and input and calculates the next state. This is where the feedforward layer is added, as processed input. The remainder of the function is the same as in *Basic RNN*

[2]:
```
from __future__ import division

from psychrnn.backend.rnn import RNN
import tensorflow as tf
tf.compat.v1.disable_eager_execution()

import numpy as np

class Basic_with_Feedforward(RNN):
    """ The basic recurrent neural network model.

    Basic implementation of :class:`psychrnn.backend.rnn.RNN` with a simple RNN.
    Input goes through a feedforward layer before being passed to the recurrent part␣
↪of the RNN.
    Biological constraints are enabled.

    Args:
        params (dict): See :class:`psychrnn.backend.rnn.RNN` for details.
            Additional Dictionary Keys:
```

(continues on next page)

(continued from previous page)

```
            N_feedforward_out (int, optional): Number of outputs from the␣
↪feedforward input layer. Default: 32
            W_in_first_train (bool, optional): True if feedforward weights, W_in_
↪first, are trainable. Default: True

    """

    def __init__(self, params):

        self.N_feedforward_out = params.get('N_feedforward_out', 32)
        self.W_in_first_train = params.get('W_in_first_train', True)

        super(Basic_with_Feedforward, self).__init__(params)

        self.initializer.initializations['W_in_first'] = params.get('W_in_first',␣
↪self.initializer.rand_init(np.ones((self.N_feedforward_out, self.N_in))))

        self.initializer.initializations['feedforward_input_connectivity'] = params.
↪get('feedforward_input_connectivity', np.ones(( self.N_rec, self.N_feedforward_
↪out)))

        self.initializer.initializations['feedforward_W_in'] = params.get(
↪'feedforward_W_in', self.initializer.rand_init(np.ones(( self.N_rec, self.N_
↪feedforward_out))))

        with tf.compat.v1.variable_scope(self.name) as scope:
            # Input weight matrix:
            self.W_in_first = \
                tf.compat.v1.get_variable('W_in_first', [self.N_feedforward_out, self.
↪N_in],
                                initializer=self.initializer.get('W_in_first'),
                                trainable=self.W_in_first_train)

            self.input_connectivity = tf.compat.v1.get_variable('feedforward_input_
↪connectivity', [self.N_rec, self.N_feedforward_out],
                                                    initializer=self.initializer.
↪get('feedforward_input_connectivity'),
                                                    trainable=False)

            self.W_in = \
                tf.compat.v1.get_variable('feedforward_W_in', [self.N_rec, self.N_
↪feedforward_out],
                                initializer=self.initializer.get('feedforward_W_in'),
                                trainable=self.W_in_train)

    def recurrent_timestep(self, rnn_in, state):
        """ Recurrent time step.

        Given input and previous state, outputs the next state of the network.

        Arguments:
            rnn_in (*tf.Tensor(dtype=float, shape=(?*, :attr:`N_in` *))*): Input to␣
↪the rnn at a certain time point.
            state (*tf.Tensor(dtype=float, shape=(* :attr:`N_batch` , :attr:`N_rec`␣
↪*))*): State of network at previous time point.

        Returns:
```

(continues on next page)

(continued from previous page)

```
            new_state (*tf.Tensor(dtype=float, shape=(* :attr:`N_batch` , :attr:`N_
↪rec` *))*): New state of the network.

        """
        processed_input = self.transfer_function(tf.matmul(rnn_in, self.W_in_first,␣
↪transpose_b=True, name="3"))

        new_state = ((1-self.alpha) * state) \
                    + self.alpha * (
                        tf.matmul(
                            self.transfer_function(state),
                            self.get_effective_W_rec(),
                            transpose_b=True, name="1")
                        + tf.matmul(
                            processed_input,
                            self.get_effective_W_in(),
                            transpose_b=True, name="2")
                        + self.b_rec)\
                    + tf.sqrt(2.0 * self.alpha * self.rec_noise * self.rec_noise)\
                      * tf.random.normal(tf.shape(input=state), mean=0.0, stddev=1.0)

        return new_state

    def output_timestep(self, state):
        """ Output timestep.

        Given the state, what is the output of the network?

        Arguments:
            state (*tf.Tensor(dtype=float, shape=(* :attr:`N_batch` , :attr:`N_rec`␣
↪*))*): State of network at a given timepoint for each trial in the batch.

        Returns:
            output (*tf.Tensor(dtype=float, shape=(* :attr:`N_batch` , :attr:`N_out`␣
↪*))*): Output of the network at a given timepoint for each trial in the batch.

        """

        output = tf.matmul(self.transfer_function(state),
                                self.get_effective_W_out(), transpose_b=True, name="3
↪") \
                    + self.b_out

        return output

    def forward_pass(self):

        """ Run the RNN on a batch of task inputs.

        Iterates over timesteps, running the :func:`recurrent_timestep` and :func:
↪`output_timestep`

        Implements :func:`psychrnn.backend.rnn.RNN.forward_pass`.

        Returns:
            tuple:
            * **predictions** (*tf.Tensor(*:attr:`N_batch`, :attr:`N_steps`, :attr:`N_
↪out` *))*) -- Network output on inputs found in self.x within the tf network.
```

(continues on next page)

(continued from previous page)

```
            * **states** (*tf.Tensor(*:attr:`N_batch`, :attr:`N_steps`, :attr:`N_rec`␣
↪*))*) -- State variable values over the course of the trials found in self.x within␣
↪the tf network.

        """

        rnn_inputs = tf.unstack(self.x, axis=1)
        state = self.init_state
        rnn_outputs = []
        rnn_states = []
        for rnn_input in rnn_inputs:
            state = self.recurrent_timestep(rnn_input, state)
            output = self.output_timestep(state)
            rnn_outputs.append(output)
            rnn_states.append(state)
        return tf.transpose(a=rnn_outputs, perm=[1, 0, 2]), tf.transpose(a=rnn_states,
↪ perm=[1, 0, 2])
```

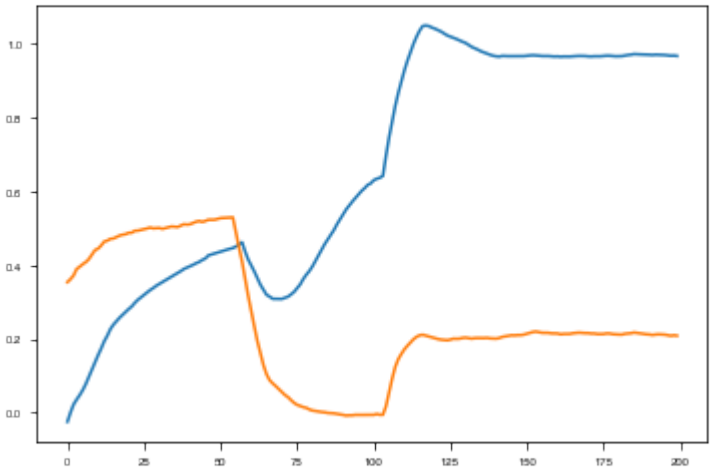
[3]:
```
from matplotlib import pyplot as plt
%matplotlib inline

# ---------------------- Import the package --------------------------
from psychrnn.tasks.perceptual_discrimination import PerceptualDiscrimination
from psychrnn.backend.models.basic import Basic

# ---------------------- Set up a basic model --------------------------
pd = PerceptualDiscrimination(dt = 10, tau = 100, T = 2000, N_batch = 128)
network_params = pd.get_task_params() # get the params passed in and defined in pd
network_params['name'] = 'model' # name the model uniquely if running mult models in␣
↪unison
network_params['N_rec'] = 50 # set the number of recurrent units in the model
model = Basic_with_Feedforward(network_params) # instantiate a basic vanilla RNN

# ---------------------- Train a basic model --------------------------
model.train(pd) # train model to perform pd task

# ---------------------- Test the trained model --------------------------
x,target_output,mask, trial_params = pd.get_trial_batch() # get pd task inputs and␣
↪outputs
model_output, model_state = model.test(x) # run the model on input x

# ---------------------- Plot the results --------------------------
plt.plot(model.test(x)[0][0,:,:])

# ---------------------- Teardown the model --------------------------
model.destruct()
```

```
Iter 1280, Minibatch Loss= 0.128628
Iter 2560, Minibatch Loss= 0.095538
Iter 3840, Minibatch Loss= 0.089444
Iter 5120, Minibatch Loss= 0.085808
Iter 6400, Minibatch Loss= 0.082073
Iter 7680, Minibatch Loss= 0.073270
Iter 8960, Minibatch Loss= 0.059777
Iter 10240, Minibatch Loss= 0.035830
Iter 11520, Minibatch Loss= 0.025013
Iter 12800, Minibatch Loss= 0.031894
```

(continues on next page)

(continued from previous page)

```
Iter 14080, Minibatch Loss= 0.072593
Iter 15360, Minibatch Loss= 0.038075
Iter 16640, Minibatch Loss= 0.027692
Iter 17920, Minibatch Loss= 0.025687
Iter 19200, Minibatch Loss= 0.031069
Iter 20480, Minibatch Loss= 0.020855
Iter 21760, Minibatch Loss= 0.012131
Iter 23040, Minibatch Loss= 0.015565
Iter 24320, Minibatch Loss= 0.012561
Iter 25600, Minibatch Loss= 0.012876
Iter 26880, Minibatch Loss= 0.018462
Iter 28160, Minibatch Loss= 0.015606
Iter 29440, Minibatch Loss= 0.009202
Iter 30720, Minibatch Loss= 0.016933
Iter 32000, Minibatch Loss= 0.011563
Iter 33280, Minibatch Loss= 0.014158
Iter 34560, Minibatch Loss= 0.014100
Iter 35840, Minibatch Loss= 0.026791
Iter 37120, Minibatch Loss= 0.024815
Iter 38400, Minibatch Loss= 0.024076
Iter 39680, Minibatch Loss= 0.015183
Iter 40960, Minibatch Loss= 0.014680
Iter 42240, Minibatch Loss= 0.010198
Iter 43520, Minibatch Loss= 0.007630
Iter 44800, Minibatch Loss= 0.011275
Iter 46080, Minibatch Loss= 0.010074
Iter 47360, Minibatch Loss= 0.013527
Iter 48640, Minibatch Loss= 0.011785
Iter 49920, Minibatch Loss= 0.019069
Optimization finished!
```



3.10 Further Extensibility – Initializations, Loss Functions, and Regularizations

If you wish to modify weight initializations you must define an initialization class describing your preferred initial weight patterns that inherits from *WeightInitializer* or one of it's child classes.

If you wish to modify loss functions or regularizations you must define a function and pass that function into the RNN as part of the RNN's params. See *LossFunction* and *Regularizer* respectively for details.

PYTHON MODULE INDEX

p

INDEX

I

L

M

O

P

R

S

T

W