
Jasmine Stone
CPSC 490 Advisor: Steven Zucker
Advisor: John Murray

May 2, 2019

Abstract

Modeling using recurrent neural networks (RNNs) is quickly becoming a popular strategy to probe the mechanisms behind cognitive tasks. RNNs are able to accomplish many popular cognitive tasks but are vastly more manipulable and observable than other classic model organisms such as monkeys and humans. Modeling is a fast, inexpensive way to investigate the abilities and limitations of proposed mechanisms behind cognitive tasks, and can lead to deeper understanding of current proposed mechanisms, as well as propose new mechanistic hypotheses. These insights can in turn guide experimental focus to better distinguish between current proposals.

Here we introduce an accessible and extensible Python package for training RNNs on a variety of cognitive tasks. While our package already includes implementations of multiple cognitive tasks, we also provide an accessible framework (requiring only knowledge of Python and NumPy) for easily defining new cognitive tasks and for customizing existing tasks. The deep learning details are handled by the TensorFlow-based backend and are extensible for projects that require additional customization. We provide multiple initializations, loss functions and regularizations as well as a framework to easily add more.

Our package focuses on facilitating specification of neurobiological constraints on synaptic architecture and on learning. We include optional implementations Dale’s Law, neurobiologically plausible sparse connectivity, regional connectivity differences and individual unit time-constants.

The design of PsychRNN further enable novel investigations. Task modularity makes it easy to investigate how parametric variations in task demands determine network solutions. Our package additionally can return intermediate network solutions so researchers can investigate how network solutions develop over the course of training. PsychRNN allows researchers to implement task shaping (also known as curriculum learning) – training is adjusted in closed loop based on performance. Experimentalists regularly train animals to perform tasks by shaping tasks – PsychRNN allows investigation of how shaping trajectory choice could affect the observed results. PsychRNN is available at [https://github.com/dbehrlich/PsychRNN](https://github.com/dbehrlich/PsychRNN).
1 Introduction

Neural networks trained to perform a given cognitive task successfully simulate neurons and predict neuronal responses in cortex for animal models performing the same task (Yamins et al., 2014; Yamins and DiCarlo, 2016). Multiple research groups have used RNNs in this way to model motor and sensory cortex successfully—they have found that the network dynamics of the RNN align with what we know of neuronal responses (Goudar and Buonomano, 2018; Sussillo et al., 2015). By analyzing the structure of the RNN state variable for a given task, researchers are able to better predict and explain previously not-well-understood features of neuronal responses (Goudar and Buonomano, 2018; Mante et al., 2013).

Barak (2017) formalized using reverse engineering of RNN solutions to inform our understanding of the brain. Modeling cognitive tasks using RNNs is currently an area of intense interest, as evidenced by the numerous papers concerning this subject (Kuroki and Isomura, 2018; Beiran and Ostojic, 2019; Wang et al., 2018; van Gerven, 2017).

However, there are limited resources for people who want to model these cognitive tasks using RNNs (Song et al., 2016). For experimental labs, getting started with machine learning frameworks often requires a dedicated graduate or postdoc student to learn and be in charge of projects relating to machine learning, a very time-intensive and expensive commitment. Even for theory labs, building a framework for modeling cognitive tasks is time intensive. And, if theorists choose to code each experiment separately rather than developing a framework, much code is duplicated, and the resulting code files are often hard to understand for others. We thus introduce a framework, PsychRNN, designed to be easy to use and flexible for theorists and experimentalists alike. PsychRNN reduces the time and effort needed to start modeling cognitive tasks using RNNs and thus makes incorporating this research into normal research workflows very easy and convenient. Here we introduce the features of PsychRNN, compare it to alternatives, and describe the overall package structure.

2 Key Features

Here we describe the key features and advantages of PsychRNN.

2.1 Intuitive Task Definitions

PsychRNN’s framework for designing tasks allows experimentalists to implement tasks in terms similar to those they are familiar with from their animal models. Rather than creating distant abstractions, researchers can work with tasks that are intuitively similar to those they already use. For the purposes of this paper, we demonstrate the PsychRNN framework using two categorically different tasks, Vibrotactile Delayed Discrimination (VDD) and Random Dot Motion (RDM) that are both standards in the neuroscience literature.

Note that PsychRNN’s framework allows for the definition of any time-varying task, but that we use these widely understood standard tasks to demonstrate the capabilities of our package. Additionally, the design decisions described below for VDD and RDM are only
one possible set of design decisions—other design decisions, including those using different numbers of input and output channels, are also possible using PsychRNN.

2.1.1 VDD

The VDD task is the parametric working memory task used by Romo et al. (1999). In the VDD task, the animal feels a first frequency stimulus and must store a trace of that frequency value for comparison (Romo et al., 1999). Then, after a delay, the animal feels a second frequency stimulus and must indicate whether the second stimulus frequency is higher or lower than the first (Romo et al., 1999).

In the implementation of the VDD task included with PsychRNN, two channels are used. On any given trial, one channel has a higher first than second stimulus frequency, whereas the other channel has a lower first than second stimulus frequency. It varies randomly by trial which channel has the higher second frequency. Frequency pairs are chosen at random from a predefined list. Decisions must be made in a short interval following the conclusion of the second stimulus. These design decisions are similar to those made by Song et al. (2016). PsychRNN’s VDD task implementation is illustrated in Figure 1.

2.1.2 RDM

The RDM task is a perceptual decision making task (Britten et al., 1992). In RDM experiments with animals, animals are shown a pattern of random dots in which some percentage of the dots are moving in the same direction while the remainder of the dots move randomly (Britten et al., 1992). The percentage of dots moving in the same direction is called the percent coherence (Britten et al., 1992). The stimuli of moving dots is shown for a certain amount of time, after which the animal must make a saccade, or eye movement, to one of two targets to indicate the perceived direction of motion of the dots (Britten et al., 1992). 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 (Britten et al., 1992).

The RDM task implementation includes two channels, each representing a direction of motion. The float value of a given input channel at a given time point represents the instantaneous coherence of the motion of dots in the direction of represented by that channel. Since there is noise in the signal, integration over time is required at low coherence levels to determine the direction of motion, since at any given time point the evidence is non-conclusive. After the stimulus period is over, a decision is made—the channel of goal output representing the correct decision direction is set to one while the channel of goal output representing the incorrect decision remains at zero. This RDM task implementation is illustrated in Figure 2.

2.2 Abstracted Neural Network Training and Simulation Backend

To get started with PsychRNN, one needs to select or define a task to work with. Once the task is defined using only Python and NumPy (or a predefined task is selected), training the
Figure 1: **VDD Task Input and Goal Output.** Sample input and goal output produced by the VDD task included in PsychRNN. (a) Input produced by the VDD task. Both channels are shown—one channel has a higher first than second stimulus frequency, the other has a higher second than first stimulus frequency. (b) Correct (goal) output for the input shown in a. The goal output decision is made immediately after the second stimulus concludes, and last for a short duration. The channel with a lower first than second stimuli has a low goal output value, and the channel with a higher first than second stimuli has a high goal output value.
Figure 2: **RDM Task Input and Goal Output.** Sample input and goal output produced by the RDM task included in PsychRNN. (a) Input produced by the RDM task. Both channels are shown—the orange channel has slightly higher coherence, however, at any given point in time either channel could have higher coherence. (b) Correct (goal) output for the input shown in a. The output decision is made immediately after the second stimulus concludes, and lasts for the remainder of the trial.
network is easy and doesn’t require any knowledge of TensorFlow, or other machine learning packages (see Code Sample 1 for a minimal example for how to train a network on any given task using PsychRNN.) Once the network is trained, the output and state variables of the network on any given trial can be extracted (see Code Sample 1 line 18).

As described in Section 2.1, the task definition defines the input and goal output for task trials. Here I introduce the other feature the task definition includes—the output mask. Some tasks are more constrained than others. In some tasks, fixation or hold patterns must be maintained until the response period. In other tasks, behavior is free until the response period. Without an output mask, the output unit of the neural network would be optimized to match the goal output for the whole time course. In the case of the RDM, this means both channels would be constrained to be zero until the stimulus period ended, at which time the channel representing the correction would be optimized to jump to 1. In the version of RDM implemented here, we include an output mask that is zero until the decision period starts, at which point it becomes one. This means that the network is optimized to match the goal output in the decision period but not before. The output mask is implemented as a multiplicative factor when calculating the loss—any losses from time points where the output mask is zero are zeroed out and don’t count towards the total loss.

When the network is trained, the weights are optimized for having the network, given the task input, output the goal output wherever the mask is nonzero. So, once the network is trained, researchers may be interested in seeing what the output from the network is on the task (see Figure 3). The output of the network can be thought of as representing the animal behavior—that is, the observable decision that the animal makes. In the case of the RDM task, this is a saccade. Normally when neuroscience researchers experiment using these cognitive tasks with animal models, they simultaneously record from neuronal populations. These state variables can be thought of as representing these neuronal populations (see Figure 4). The state variables are the values of each RNN unit over the course of a trial before any nonlinear activation function is applied. Thus, researchers can validate the network’s training by checking that the model’s performance follows similar trends to that of animal models, and analyses can be done on the state variables to try to understand what different neuronal populations are doing.

Synaptic weight matrices, and all the details of the trained model can also be accessed and saved if researchers want to investigate other properties of the network.

2.3 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. Biological constraints affect the dynamics of the resulting system and the state variables that we see, and so are of direct interest both to experimentalists (especially those interested in specific biological constraints, or features that arise from the structural constraints of the brain) and to theorists, many of whom are interested in the dynamics of the brain.

Dale’s Principle states that a neuron releases the same set of neurotransmitters at each
Figure 3: **Network Output on RDM Task.** Sample output of an RNN network trained on the RDM task. (a) The input and goal output as specified by the task, and the network output when passed the input in the top panel. The output mask, specified by the task definition, is shown overlaid on the plot of network output to illustrate where the network was optimized to as closely match the goal output as possible. Because the output mask is zero during the stimulus period, that is, the RNN is not penalized or rewarded for whatever output occurs during that period, the network is unconstrained during that period. (b) Plot of the percent of decisions the network makes towards direction zero at varying coherence levels. Negative coherences levels indicate a task input of abs(coherence) in direction one. This curve is the same shape as those produced by animals trained on the RDM task. This plot validates that the network successfully learned the task and that the simplified task representation still capturing the essence of the task.
Figure 4: Analysis of State Variables from RNN Trained on RDM Task. The network outputs a state variable trace over time for each recurrent unit in the network. In the network shown here, there are fifty units, and so fifty state variables. These state variables can be thought of as neuronal populations. (a) State variable traces output from trials at a given coherence. Each subplot represents a different coherence value, labeled at the top of each subplot. Only trials where the network output the correct response are included (so direction zero for positive coherence, and direction one for negative coherence). One thousand trials were performed at each coherence level—state variable values were averaged across these one thousand trials at each coherence level. The same colors are used for a given state variable (or neuronal population) in each subplot. Note that the grey line diverges most sharply in negative coherence cases, whereas the orange line diverges most sharply in positive coherence cases. Note also that the larger the value of abs(coherence), the more divergent those lines are. (b) Plot of the first two principle components of the state variables at different coherences. PCA vectors were found by concatenating the data shown in each subplot in a to make a big matrix that was [number of state variables x [number of coherences * number of time steps in a trial]] and then demeaning and performing PCA. Observe the differing trajectories for positive and negative coherences, and the separation of trajectories within positive or negative coherences by strength of coherence.
import psychrnn
from psychrnn.tasks import rdm as rd
from psychrnn.backend.models.basic import Basic
import tensorflow as tf

rdm = rd.RDM(dt = 10, tau = 100, T = 2000, N_batch = 128)
gen = rdm.batch_generator()

params = rdm.dict_
params ['name'] = 'model'
params ['N_rec'] = 50

model = Basic(params)
model.build()
model.train(gen)

x, goal_output, mask = next(gen)
model_output, model_state = model.test(x)

model.destruct()

Code Sample 1: Training Model and Extracting Output and State Variables
This code sample is a minimal example of using PsychRNN. In this sample, all relevant modules are imported (lines 1-4), an RDM task object is initialized and parameters are set (lines 6-11), the basic RNN model is initialized, built, and trained (lines 13-15), output and state variables are extracted (lines 17-18), and the RNN model is destructed to clean up memory (line 20). N_rec denotes the number of recurrent units to use. In this example, as in Figures 3 and 4, fifty recurrent units are used. This code sample shows how to extract all data from the task and network used in Figures 3 and 4. In line 17 the task input (x), goal output, and output mask are all extracted. In line 18, the network output and state variables are both extracted from the network for the trials given by input x.
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). We thus include an optional parameter to be passed to the RNN model when it is initialized called **dale’s ratio**. When **dale’s ratio** is passed into the network, that ratio of recurrent units are made to have only positive synapses, or connections, with other neurons or recurrent units and one minus that ratio of the recurrent units are made to have only negative synapses, or connections with other neurons or recurrent units. This enables researchers interested in how the constraints of neurons affect the dynamics of the network to ask those questions. Additionally, researchers who study, for example, inhibitory interneurons, would otherwise see no relevant correlate in the RNN for what they are studying. With the implementation of Dale’s law, the role of inhibitory populations can be easily investigated using PsychRNN.

Additionally, the brain is not fully connected—instead there are different regions or areas which are densely connected within a given region, and sparsely connected across areas. We include a framework for specifying the connectivity matrix for the input, output, and recurrent layers, allowing researchers to specify which connections can exist (have nonzero weights) and which cannot. Using block-like matrices can then produce connectivity patterns that can resemble different brain regions. These constraints also produce different dynamics which are of interest to researchers. Connectivity patterns are implemented as part of the initialization. Weights matrices are multiplied element wise with the connectivity matrices. The connectivity matrices are by default all ones, allowing all-to-all connectivity. Defining alternate connectivity matrices in the initialization passed in to the model thus forces sparser, structured connectivity in the network.

We can represent most cognitive tasks using a variety of different neural networks, so, for many researchers, the interesting question is (Barak, 2017): What network architectures and biological constraints lead to network states that most closely match what we record from neurons in animals? And what insight can we glean from differences we observe between biology and neural networks? Thus, the ability to enforce biological constraints is essential to make this package relevant and useful for researchers, and is a current topic of interest (Barak, 2017).

2.4 Modularity

In PsychRNN, tasks are defined as modular objects. This allows researchers to change task parameters easily and iteratively without having to hardcode values. Since it is easy to try different task parameters, researchers can iterate over different parameters to see how those parameters affect network results.

For example, with the VDD task, the duration of stimulus one and stimulus two can be varied individually (see Figure 5) with very little effort (see Code Sample 2). Researchers can also experiment with the delay duration, onset time, decision duration, or any other parameters they decide to include in their task definition. For the purposes of this paper, stimulus one and stimulus two are varied because they are the easiest to see. Varying the stimulus duration during training is crucial (and done automatically in the implementation
Figure 5: **Modularity of Task Definition** One channel of input generated by VDD task with varied stimulus one duration and stimulus two duration. Stimulus one duration is varied across columns, and stimulus two duration is varied across rows. In PsychRNN, varying the parameters of the task as illustrated above is easy (see Code Sample 2). In previous packages for modeling of cognitive tasks using recurrent neural networks, tasks were not nearly so modular, and varying parameters as seen above would have been much more cumbersome.

included with the PsychRNN release) to force the network to learn not to integrate area under the curve (a possible solution if stimulus duration is not varied) but to obtain some version of the first frequency value for comparison with the second.

PsychRNN’s backend is modular as well—regularizations, loss functions, initializations, and models (e.g. RNNs vs. long short-term memory networks (LSTMs) etc) are all defined separately. So researchers can mix and match different versions of these components or define new versions of these components without worrying about any other component being affected. This modularity lowers the barrier-to-entry – a researcher who wants to create new regularizations need not be deeply familiar with the loss functions, initializations, or models, etc.

### 2.5 Curriculum Learning

The idea behind curriculum learning stems from the observation that people learn best in stages—for example, when learning to play chess, people often learn how to first play a
game with only pawns, and then slowly learn to play more complicated games, for example with pawns and rooks, and then with pawns, bishops and rooks, etc. Eventually, after a number of progressively harder sorts of games involving chess pieces, people learn to play a full chess game with all of the pieces. Bengio et al. (2009) showed that machines also learn best in stages this way—neural networks trained in stages are trained faster, that is with fewer iterations. Additionally, neural networks can reach better local minima, that is reach a lower asymptotic loss, when trained in stages in this way (Bengio et al., 2009). Additionally, since curriculum learning can result in faster training (see Figure 6), some tasks that were previously not feasible within current computational limits are now feasible using curriculum learning. Furthermore, the order in which tasks in the curriculum are presented, and the tasks chosen as part of the curriculum affects performance (Kirkpatrick et al., 2017; Pentina et al., 2015).

Because PsychRNN tasks are modular and object oriented, as described in Section 2.4, we were able to design an intuitive framework for curriculum learning that is easy to use even without any machine learning knowledge, or knowledge of TensorFlow. Curriculum learning is implemented by passing a curriculum object to the RNN model when training is executed. Although very flexible and customizable, the simplest form of the curriculum object can be instantiated solely with the list of tasks that one wants to train on sequentially (See Code Sample 3).

In the past, neuroscience researchers haven’t really experimented with curriculum learning in machines. However, since PsychRNN includes an easy-to-use implementation of curriculum learning (see Code Sample 3), neuroscience researchers can now experiment with curriculum learning without much startup cost at all.

Curriculum learning is perhaps especially interesting to our main audience as a way to think about how to speed up training of animals, and how training protocols optimized for RNNs could potentially reveal task factorizations useful for animal training.
Figure 6: **Training Performance with Curriculum Learning** Loss (mean square error) and accuracy across training iterations on a network trained with curriculum learning, and a network trained without curriculum learning. Observe that the network trained using curriculum learning took far fewer iterations to reach 90 percent accuracy than the network trained without curriculum learning. The network trained without curriculum learning was trained solely on stimulus with coherence = .1. The network trained with curriculum learning was trained with the curriculum shown in Code Sample 3, with coherence decreasing from .7 to .5 to .3 to .1 as performance improved. When the network reached 90 percent accuracy on stimuli with coherence = .1, training was stopped.
<table>
<thead>
<tr>
<th>Implemented Features</th>
<th>PsychRNN</th>
<th>PyCog</th>
<th>TensorFlow</th>
<th>Keras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Citation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Key Advantage</strong></td>
<td>Curriculum Learning, supported backend</td>
<td>Biological Constraints on RNNs</td>
<td>Flexibility</td>
<td>Works with multiple backends</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Python</td>
<td>Python</td>
<td>Python, Partial C</td>
<td>Python</td>
</tr>
<tr>
<td><strong>LSTM</strong></td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Modular Task Definition</strong></td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Object Oriented RNN and Task Framework</strong></td>
<td>Yes</td>
<td>Partially</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Dale’s Principle</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Connectivity Pattern Easily Specifiable</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Curriculum Learning Implemented</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Backend</strong></td>
<td>TensorFlow</td>
<td>Theano</td>
<td>TensorFlow</td>
<td>TensorFlow, Theano, CNTK</td>
</tr>
<tr>
<td><strong>Backend Currently Supported</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Most</td>
</tr>
<tr>
<td><strong>Backend supports automatic differentiation</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>GPU support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Optimizer</strong></td>
<td>TensorFlow built in options with implemented regularizers</td>
<td>SGD with implemented regularizers</td>
<td>built in options</td>
<td>built in options</td>
</tr>
</tbody>
</table>
Code Sample 3: **Curriculum Learning**  
The code sample above trains an RNN on a sequence of RDM's with decreasing coherence. The network starts by learning to perform the RDM task with high coherence. Once the network reaches 90 percent accuracy on a given task, the next task begins. This continues until the network has reached 90 percent accuracy on the final task—in this case, the lowest coherence RDM. Curriculum learning is done by making a list of tasks that form the curriculum (line 5), and then passing that list in to the Curriculum class to form a curriculum object (line 7). That curriculum object is then included in the training parameters dictionary (line 8), and when the network is passed those training parameters for training, the network will be trained using the curriculum, or sequence of tasks defined in line 5.

### 3 Package Structure

The PsychRNN package is organized into two main portions: the Task Object and the Backend, as illustrated in Figure 7. Anyone using PsychRNN, experimentalists and theorists alike, will want to be able to define the tasks they are interested in. Therefore, the task definition is entirely separate from the Backend so that the only tools one needs to be familiar with to define a new task are Python and NumPy. The Backend is approachable and customizable, but customizing it may require knowledge of TensorFlow depending on what exactly is being customized. We expect that experimentalists with less experience with TensorFlow and machine learning will not need to customize the backend too much. We have included customizations that we expect people without machine learning experience to be interested in, such as biological constraints and curriculum learning have been abstracted out so that no knowledge of TensorFlow is required. For those who do know TensorFlow, new models are easy to define, as are new regularizations, loss functions and initializations. For those with experience with TensorFlow, PsychRNN provides a modular framework within which to try out new models and regularizations in an easy and convenient way, and enables one to get started training RNNs on cognitive tasks quickly and easily.

#### 3.1 Task

Up until this point, we have only concretely discussed the RDM and VDD tasks included in the PsychRNN release. Here we explain how to define your own new task using only
In order to train a network on a task, one must define a task object or use one of the tasks included in the PsychRNN release and instantiate a backend. Doing this requires only knowledge of Python and NumPy. When a network is trained, measures of performance indicate the speed at which the network is learning at regular epochs (time points or iterations) throughout training. We include two measures of performance: loss and accuracy. Loss is a metric defined by the loss function chosen (by default, loss is the mean square error over the goal output and network output). Loss is a typical metric used in machine learning, and a lower loss value corresponds to better performance. Optimization of the network weights is performed on the loss. Accuracy is an optional metric that is defined in the task definition. We use accuracy to be a more biologically relevant measure of performance. On a given trial, the accuracy value is either one (success) or zero (failure). In contrast, loss on a given trial is a positive real-numbered value. Accuracy is calculated over multiple trials to obtain a ratio of trials correct to trials incorrect. As discussed in Section 2.2, once the network is trained, the synaptic weight matrix can be saved out, and state variables and network output can be generated for any given trial.

Figure 7: **Overall Package Structure** In order to train a network on a task, one must define a task object or use one of the tasks included in the PsychRNN release and instantiate a backend. Doing this requires only knowledge of Python and NumPy. When a network is trained, measures of performance indicate the speed at which the network is learning at regular epochs (time points or iterations) throughout training. We include two measures of performance: loss and accuracy. Loss is a metric defined by the loss function chosen (by default, loss is the mean square error over the goal output and network output). Loss is a typical metric used in machine learning, and a lower loss value corresponds to better performance. Optimization of the network weights is performed on the loss. Accuracy is an optional metric that is defined in the task definition. We use accuracy to be a more biologically relevant measure of performance. On a given trial, the accuracy value is either one (success) or zero (failure). In contrast, loss on a given trial is a positive real-numbered value. Accuracy is calculated over multiple trials to obtain a ratio of trials correct to trials incorrect. As discussed in Section 2.2, once the network is trained, the synaptic weight matrix can be saved out, and state variables and network output can be generated for any given trial.
Figure 8: **Task Structure** Defining a new task requires defining two functions. One, `trial_function` describes the task input and output. The other, `generate_trial_params` defines the parameters for a given trial. Only Python and NumPy knowledge is necessary to define a new task.

Python and NumPy (See Figure 8). Two functions need to be defined to create a new task: `generate_trial_params` and `trial_function`. An accuracy function to accompany the task can optionally be defined. `generate_trial_params` creates trial specific parameters for the task (e.g. coherence and correct decision direction for the RDM task). It takes two inputs: the batch number and the trial number. Neither of these variables are used in the task definitions included with PsychRNN, but these variables can be used, for example, to balance trials within a batch, or to modify the parameters generated by batch number. `generate_trial_params` returns a dictionary, `params`, containing all the parameters necessary for input to `trial_function`. `trial_function` specifies the input, goal output, and output mask at a given time $t$ given the parameters generated by `generate_trial_params`. An example task definition is shown in Code Sample.
class SimpleRDM(Task):
    def __init__(self, dt, tau, T, N_batch):
        super(SimpleRDM, self).__init__(2, 2, dt, tau, T, N_batch)
    def generate_trial_params(self, batch, trial):
        # 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, t, params):
        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)
        y_t = np.zeros(self.N_out)
        mask_t = np.ones(self.N_out)

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

        # Compute values
        if onset < t < onset + stim_dur:
            x_t[dir] += 1 + coh
            x_t[(dir + 1) % 2] += 1
        if t > onset + stim_dur + 20:
            y_t[dir] = 1.
        if t < onset + stim_dur:
            mask_t = np.zeros(self.N_out)
        return x_t, y_t, mask_t

Code Sample 4: Example Task Definition The Code Sample above defines a simple RDM task. `generate_trial_params` selects the coherence and direction on a trial by trial basis. `trial_function` sets the input, goal output and output mask depending on the time in the trial and the parameters passed in.
Figure 9: **Backend Structure** The figure above illustrates the details of the backend. First, the model, or network architecture, is selected. Currently, versions of a normal RNN and LSTM are implemented — more models or architectures can be defined given a solid understanding of TensorFlow. Then, that model is instantiated with a dictionary of parameters. That dictionary of parameters must include the number of recurrent units, but it may also include specifications of loss functions, initializations, regularizations, or biological constraints. These, with the exception of the biological constraints described in Section 2.3, are described below. When the network is trained, training specifications, such as the optimizer or a curriculum can be specified. This creates a trained neural network, parameterized as one chooses. If any parameter is not set, a default is used.

### 3.2 Backend

The backend includes all of the neural network details that are very customizable, but not necessary to get started with PsychRNN. The TensorFlow details are abstracted away by the backend so that researchers are free to work with or without an understanding of TensorFlow. Additionally, since the backend is internally modular, different components of the backend can be swapped in and out interchangeably. In this section, modular components of the backend are described so that researchers who want to get into more depth with PsychRNN know what tools are available to them. Figure 9 illustrates the components of the backend.

#### 3.2.1 Models

Recurrent neural networks are a large class of neural network architectures that enable processing of an input spaced over time. In the PsychRNN release, we include a basic RNN (what we’ve been referring to as an RNN throughout the rest of this paper), and an LSTM model. The basic RNN model is governed by the following equations

\[
\tau dx = (-x + W_{rec}r + b_{rec} + W_{in}u)dt + \sigma_{rec}\sqrt{2\tau}d\xi \\
r = f(x)
\]
\[ z = W_{out}r + b_{out} \]

We also include an implementation of an LSTM, a network that enables longer term memory than is easily attainable with the basic RNN (Hochreiter and Schmidhuber 1997).

Other models can also be defined by users, but require a strong understanding of TensorFlow.

### 3.2.2 Regularizers

Regularizers help prevent the network from overfitting to the data. We include options for L1 regularization, L2 regularization, L2 firing rate regularization, and the regularization introduced in Sussillo et al. (2015). Other regularizations can be added easily to the Regularizer class with some TensorFlow knowledge. By default, no regularizations are used. If nonzero values for the variables used by a given regularizer are set in the parameters used to instantiate a neural network model, that regularizer will then be used.

### 3.2.3 Initializations

The weights that define a neural network are typically defined randomly. However, with RNNs, big performance improvements can be made with some simple initialization tricks (V. Le et al., 2015). Since these initializations can be crucial for getting the network to train consistently, we have included a few of the initializations currently used in the field (Glorot and Bengio, 2010). By default, the recurrent weights are initialized with a gaussian spectral radius of 1.1 (Sussillo and Abbott, 2009). We also include an initialization called Alpha Identity (which can be selected by passing the initializer into the network model when it is instantiated) introduced in V. Le et al. (2015) that initializes the recurrent weights as an identity matrix scaled by alpha. Each of these initializations make the network much more likely to learn any given task properly.

PsychRNN includes a WeightInitialization class that initializes all network weights randomly, all biases as zero, and connectivity masks as all to all. Any new initializations must inherit this class and can override any variety of initializations defined in the base class WeightInitialization.

### 3.2.4 Loss Functions

The RNN is optimized to minimize the loss, so the choice of loss function can be crucial for determining exactly what it is the network is learning. By default, the loss function is mean square error. Our backend also includes an option for using binary_cross_entropy as the loss function. Other loss functions can be easily defined with some TensorFlow knowledge and added to the LossFunction class. Loss functions take in the network output (predictions), the goal output (y) and the output mask and return a float calculated using the TensorFlow graph.

20
3.2.5 Curriculum Learning

In section 2.5, we discussed the simplest use of curriculum learning. However, the curriculum class included in PsychRNN is very flexible and allows for much customization. By default, accuracy, as defined within a task, is used to measure the performance of the task. When the performance surpasses .9, the network starts training with the next task. However, it is possible that one might want to advance the training stage at a different threshold than .9, or advance the stage at different thresholds depending at which stage the network is at within the curriculum. The curriculum object thus includes an optional input array, thresholds for specifying the cutoff for the performance. Additionally, there are many reasonable ways to determine when to advance the curriculum stage other than accuracy, for example using loss, number of iterations, or some other measure. We include an optional metric function that can be passed into the curriculum class to define a custom measure of performance.

The implementation of curriculum learning included in PsychRNN is thus both easy to use and get started with and flexible and customizable. Customizing the Curriculum as described above requires only knowledge of Python and NumPy.

4 Conclusion

PsychRNN lowers the barrier-to-entry for theorists and experimentalists alike who are interested in using RNNs to model cognitive tasks. PsychRNN provides advantages over other alternatives (see Table 1), such as PyCog, by increasing modularity of task definition and providing a framework for curriculum learning (Song et al., 2016). This increased task modularity allows researchers to easily investigate how different task parameters affect performance and results. Curriculum learning has not been previously included in any framework of this sort, and thus provides novel direction for inquiry.

Curriculum learning, as it applies to neuroscience, is an interesting area for exploration. The animal models that experimentalists use are trained in stages, with a curriculum of a sort. For example, a monkey trained on a perceptual task may first be trained to hold fixation briefly, and then trained to hold fixation for increasingly long time periods. Eventually, when the animal has learned to fixate, other targets are added when the fixation point disappears that the animal must motion towards (Berger et al., 2018). The animal is eventually exposed to the goal task, but first, extensive training must be done in progressive stages. While some training protocols are standardized (eg Brunton et al. (2013); Berger et al. (2018)), most are not, leading to variation between research labs, and even within labs, between different animals. Curriculum learning with RNNs could allow researchers to prototype different training regimes with RNNs, a much faster and cheaper method than training animals. Then, the most successful training protocols with RNNs could be used with animals to try to speed up the training of animals or to improve the animal’s performance. This has not been tried before, so we don’t know whether the curriculum that work with RNNs will also be the most effective curriculum for animal models, but we think this is a promising area for exploration, and so have included the tools to make this easy in PsychRNN.
While animals with short lifespans, such as rats or mice, are often only trained to perform one task within their lifetime, more expensive, longer living animals, such as non-human primates (NHP) are often trained and tested on many different tasks sequentially. Publications often do not report what tasks animals have previously been trained to perform or the training protocols used with the animals (although a few recent papers, e.g. [Brunton et al. 2013; Berger et al. 2018] have included their training protocols). However, it is possible, even likely, that the way in which animals are trained, and what tasks they have performed previously affects either their behavior or their neuronal solutions to the tasks. This area is currently only speculation since tools for asking these questions have not existed previously. The introduction of PsychRNN, with its curriculum learning framework, allows researchers to begin to investigate these questions.

5 Acknowledgments

This thesis would not have been possible without the support and guidance of my CPSC 490 Advisor, Prof. Steven Zucker—thank you. Thank you especially to Daniel Ehrlich who spearheaded the PsychRNN project back when it was Sisyphus, and who has managed and overseen the codebase as it has matured and developed. Thank you also to Daniel for being an incredible mentor throughout this whole process, and for answering all my questions and reading and re-reading my work. Thank you to David Brandfonbrener and Alex Atanasov who started the PsychRNN codebase with Daniel when they were undergraduates here—my work would not have been at all possible had they not already done so much. Thank you as well to John Murray, my advisor, without whom none of this would have been possible.

References


