Beyond Drift Diffusion Models: Fitting a broad class of decision and RL models with HDDM

Alexander Fengler[1,2], Krishn Bera[1,2], Mads L. Pedersen[1,3], Michael J. Frank[1,2]

1 Department of Cognitive, Linguistic and Psychological Sciences, Brown University
2 Carney Institute for Brain Science, Brown University
3 Department of Psychology, University of Oslo, Norway

*** alexander_fengler@brown.edu**

Abstract

Computational modeling has become an central aspect of research in the cognitive neurosciences. As the field matures, it is increasingly important to move beyond standard models to quantitatively assess models with richer dynamics that may better reflect underlying cognitive and neural processes. For example, sequential sampling models (SSMs) are a general class of models of decision making intended to capture processes jointly giving rise to reaction time distributions and choice data in n-alternative paradigms. A number of model variations are of theoretical interest, but empirical data analysis has historically been tied to a small subset for which likelihood functions are analytically tractable. Advances in methods designed for likelihood-free inference have recently made it computationally feasible to consider a much larger spectrum of sequential sampling models. In addition, recent work has motivated the combination of SSMs with reinforcement learning (RL) models, which had historically been considered in separate literatures. Here we provide a significant addition to the widely used HDDM python toolbox and include a tutorial for how users can easily fit and assess a (user extensible) wide variety of SSMs, and how they can be combined with RL models. The extension comes batteries included, including model visualization tools, posterior predictive checks, and ability to link trial-wise neural signals with model parameters via hierarchical Bayesian regression.

1 Introduction

The drift diffusion model (DDM, also called Ratcliff Diffusion Model) [39, 41], and more generally the framework of sequential sampling models (SSMs) [19, 41, 56] have become a mainstay of the cognitive scientist's model arsenal.

SSMs are used to model neurocognitive processes that jointly give rise to choice and reaction time data in a multitude of domains, spanning from perceptual discrimination to memory retrieval to preference-based choice [22, 23, 39, 42, 47] across species [8, 16, 60]. Moreover, researchers are often interested in the underlying neural dynamics that give rise to such choice processes. As such, many studies include additional measurements such as EEG, fMRI or eyetracking signals as covariates, which act as latent variables and connect to model parameters (e.g. via a regression model) to drive trial specific parameter valuations [8, 12, 13, 38, 60]. See 1 for an illustration of the DDM and some canonical experimental paradigms.

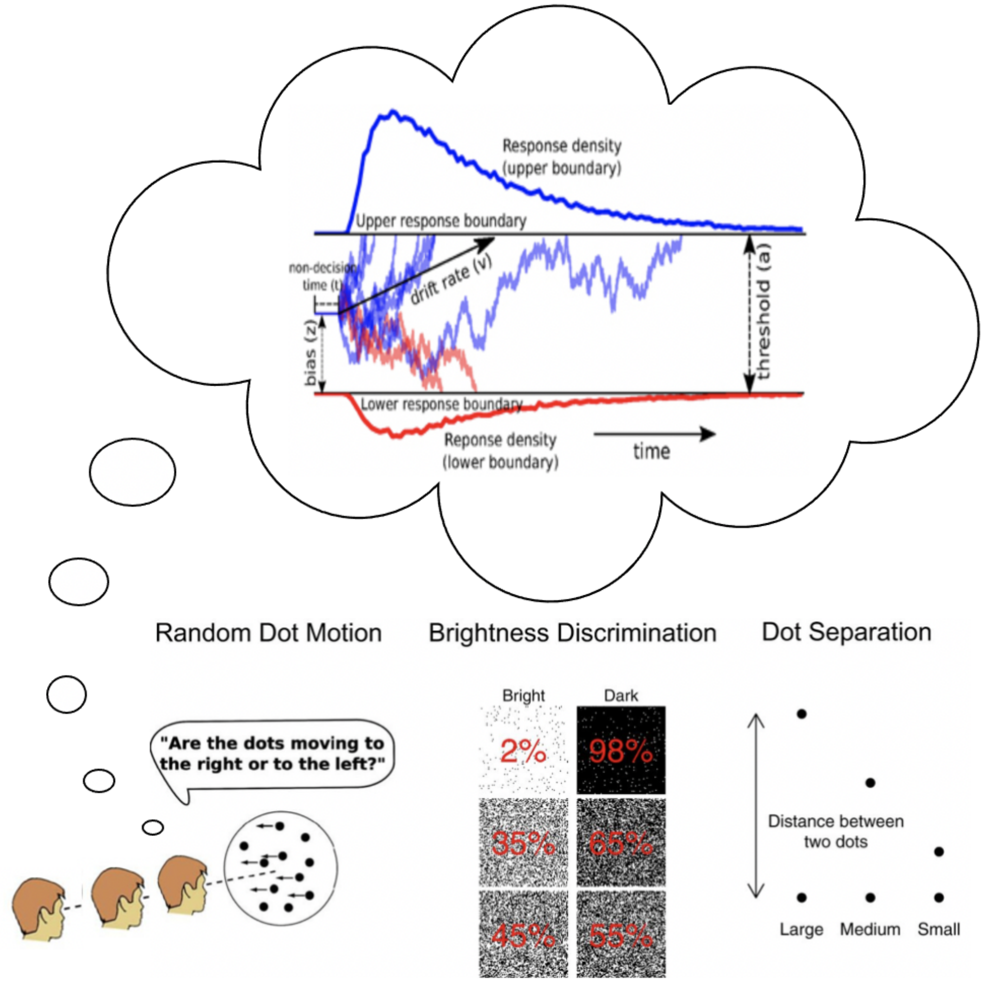

Figure 1. Drift Diffusion Model and some example applications.

The widespread interest and continuous use of SSMs across the research community has spurred the development of several software packages targeting the estimation of such models [10, 55]. For a hierarchical Bayesian approach to parameter estimation, the HDDM toolbox in python [58] is widely used and the backbone of hundreds of results published in peer reviewed journals.

HDDM allows users to conveniently specify and estimate DDM parameters for a wide range of experimental designs, including the incorporation of trial-by-trial covariates via regression models targeting specific parameters. As an example, one may use this framework to estimate whether trial by trial drift rate in a DDM covary with neural activity in a given region and time point, pupil dilation or eye gaze position. Moreover, by using hierarchical Bayesian estimation, HDDM optimizes the inference about such parameters at the individual subject and group levels.

Nevertheless, until now, HDDM and other such toolboxes have been largely limited to fitting the 2-alternative choice DDM (albeit allowing for the full DDM with inter-trial parameter variabilit). The widespread interest in SSMs has however also spurred theoretical and empirical investigations into various alternative model variants. Notable examples are, amongst others, race models with more than 2 decision options, the leaky competing accumulator model [54], SSMs with dynamic decision boundaries [6, 7, 40] and more recently SSMs based on levy flights rather than basic

Gaussian diffusions [59]. Moreover, SSMs naturally extend to n-choice paradigms.

A similar state of affairs is observed for another class of cognitive models which aim to simultaneously model the dynamics of a feedback-based learning across trials as well as the within-trial decision process. One way to achieve this is by replacing the choice rule in a reinforcement learning (RL) process a cognitive process models such as SSMs. While recent studies [10, 11, 36, 37], moved into this direction, they have again been limited to an application of the basic DDM.

Despite the great interest in these classes of models, tractable inference and therefore widespread adoption of such models has been hampered by the lack of easy to compute likelihood functions (including essentially all of the examples provided above). In particular, while many interesting models are straightforward to simulate, often researchers want to go the other way: from observed data to infer the most likely parameters. For all but the simplest models, such likelihood functions are analytically intractable, and hence previous approaches required computationally costly simulations and/or lacked flexibility in applying such methods to different scenarios [4, 45, 52, 53]. We recently developed a novel approach using artificial neural networks which can, given sufficient training data, approximate likelihoods for a large class of SSM variants, thereby amortizing the cost and enabling rapid efficient and flexible inference [9]. We dubbed such networks LANs, for *likelihood approximation networks*.

The core idea behind computation amortization is to run an expensive process only once, so that the fruits of this labor can later be reused and shared with the rest of the community. Profiting from the computational labor incurred in other research groups enables researchers to consider a larger bank of generative models and to sharpen conclusions that may be drawn from their experimental data. The benefit is three-fold. Experimenters will be able to adjudicate between a rising number of competing models (theoretical accounts), capture richer dynamics informed by neural activity, and at the same time new proposed models by theoreticians can find wider adoption and be tested against data much sooner.

Just as streamlining the analysis of simple SSMs (via e.g. the HDDM toolbox and others) allowed an increase in adoption, streamlining the production and inference pipeline for amortized likelihoods, we hope, will drive the embrace of SSMs variations in the modeling and experimental community by making a much larger class of models ready to be fit to experimental data.

Here we develop an extension to the widely used HDDM toolbox, which generalizes it to allow for flexible simulation and estimation of a large class o SSMs by reusing amortized likelihood functions.

Specifically, this extension incorporates,

- LAN [9] based likelihoods for a variety of SSMs (batteries included)
- LAN driven extension of the Reinforcement Learning (RL) - DDM capabilities, which allows RL learning rules to be applied to all included SSMs
- New plots which focus on visual communications of results across models
- An easy interface for users to import and incorporate their own models and likelihoods into HDDM

This paper is formulated as a tutorial to support application of the HDDM LAN extension for data analysis problems involving SSMs.

The rest of the paper is organized as follows. In section 2 we start by providing some basic overview of the capabilities of HDDM. Section 3 gives a brief overview of LANs [9]. Section 4 constitutes a tutorial with a detailed introduction on how to use these new features in HDDM. We conclude in section 5 embedding the new features into a broader agenda. Lastly we mention limitations and preview future developments in section 6.

2 HDDM: The basics

The HDDM python package [58], was designed to make hierarchical Bayesian inference for drift diffusion models simple for end-users with some programming experience in python. The toolbox has been widely used for this purpose by the research community and the feature set evolves to accommodate new use-cases. This section serves as a minimal introduction to HDDM to render the present tutorial self-contained. To get a deeper introduction to HDDM itself, please refer to the original paper [58], an extension paper specifically concerning reinforcement learning capabilities [36], and the documentation of the package. Here we concern ourselves with a very basic workflow that uses the HDDM package for inference.

Data HDDM expects a data set, provided as a `pandas DataFrame` [27] with three basic columns. A `'subj_idx'` column which identifies the subject, a `'response'` column which specifies the choice taken in a given trial (usually coded as 1 for *correct* choices and 0 for *incorrect* choices and a `'rt'` column which store the trial wise reaction times (in seconds). Other columns can be added, for example to be used as covariates (task condition or additional measurements such as trial-wise neural data). Here we take the example of a data set which is provided with the HDDM package. Codeblock 1 shows how to load this dataset into a python interpreter, which looks as follows,

```
cav_data = hddm.load_csv(hddm.__path__[0] + \
                '/examples/cavanagh_theta_nn.csv')
```

```
      subj_idx stim     rt  response     theta  dbs conf
0            0   LL  1.210       1.0  0.656275    1   HC
1            0   WL  1.630       1.0 -0.327889    1   LC
2            0   WW  1.030       1.0 -0.480285    1   HC
3            0   WL  2.770       1.0  1.927427    1   LC
4            0   WW  1.140       0.0 -0.213236    1   HC
...        ...  ...    ...       ...       ...  ...  ...
3983        13   LL  1.450       0.0 -1.237166    0   HC
3984        13   WL  0.711       1.0 -0.377450    0   LC
3985        13   WL  0.784       1.0 -0.694194    0   LC
3986        13   LL  2.350       0.0 -0.546536    0   HC
3987        13   WW  1.250       1.0  0.752388    0   HC

[3988 rows x 7 columns]
```

Codeblock 1. Loading package-included data

HDDM Model Once we have our data in the format expected by HDDM, we can now specify a HDDM model. We focus on a simple example here: A basic hierarchical model, which estimates separate drift rates (v) as a function of task condition, denoted by the `'stim'` column, and estimates the starting point bias z. (Boundary separation, otherwise known as decision threshold, a and non-decision time t, are also estimated by default).

This model assumes that the subject level z, a and t parameters are each drawn from group distributions, the parameters of which are also inferred. The v parameters derive from separate group distributions for each value of `'stim'`. Details about the choices of *group priors* and *hyperparameters* can be found in the original toolbox paper [58]. Codeblocks 2 and 3 show how to construct and sample from such a model.

```
basic_hddm_model = hddm.HDDM(cav_data,
                             include = ['z'],
                             depends_on = {'v': ['stim']})
```

Codeblock 2. Initializing HDDM model

Sample and Analyze Once we have defined our HDDM model, the goal is to fit the model to the data. In a bayesian context this implies obtaining a posterior distribution over model parameters. For completeness, we note that such posterior distributions are defined via Bayes' rule,

$$p(\theta|\mathbf{D}) \propto p(\mathbf{D}|\theta)p(\theta)$$

where $\mathbf{D}$ is our *data*, θ is our set of parameters, $p(\mathbf{D}|\theta)$ defines the *likelihood* (analytic in the case of the standard HDDM class) of our dataset under the model and $p(\theta)$ defines our initial *prior* over the parameters.

HDDM uses the probabilistic programming toolbox pymc [35] to generate samples from the posterior distribution via *markov chain monte carlo* (MCMC) (specifically, using coordinate-wise slice samples [28]). To generate samples from the posterior we simply type,

```
basic_hddm_model.sample(1000, burn = 500)
```

Codeblock 3. Sampling from a basic HDDM model

HDDM then provides access to a variety of tools to analyze the posterior and generate quantities of interest, including:

1. *chain summaries*: To get a quick glance at mean posterior estimates (and their uncertainty) for parameters.

2. *trace-plots* and the *gelman rubin statistic* [5]: To understand issues with chain-convergence (i.e., whether one can trust that the estimates are truly drawn from the posterior).

3. *the deviance information criterion* (DIC) [48] : As a score to be used for purposes of model comparison (with caution).

4. *posterior predictive plots*: To check for the absolute fit of a given model to data (potentially as a function of task condition etc).

The HDDM LAN extension maintains this basic HDDM workflow, which we hope facilitates seamless transition for current users of HDDM. After some brief explanations concerning *approximate likelihoods*, which form the spine of the extension, we will expose the added capabilities in detail.

3 Approximate Likelihoods

Approximate Bayesian inference is an active area of research. Indeed, the last decade has seen a multitude of proposals for new algorithms, many of which rely in one way or another on popular deep learning techniques [17, 18, 25, 30–32, 51]. Relevant to our goals here are algorithms which can estimate trial by trial likelihoods for a given model. The main idea is to replace the *likelihood* term in Bayes' Rule, with an approximation

$\hat{p}(\mathbf{D}|\theta)$, which can be evaluated via a forward pass through a simple neural network. Once the networks are trained, these "amortized" likelihoods can then be used as a plug-in (replacing the analytical likelihood function) to run approximate inference. Having access to approximate likelihoods, the user will now be able to apply HDDM to a broad variety of sequential sampling models.

The HDDM extension described here is based on a specific likelihood amortization algorithm, which we dubbed *likelihood approximation networks* (LANs) [9]. Details regarding this LAN approach, including methods, parameter recovery studies and thorough tests, can be found in [9].

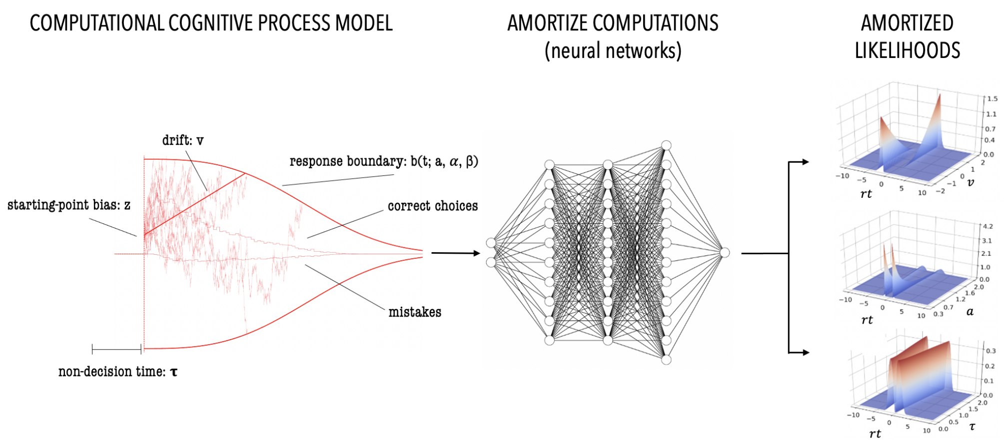

Figure 2. Depiction of the general idea behind *likelihood approximation networks*. We use a *simulator* of a *likelihood-free* cognitive process model to generate training data. This training data is then used to train a *neural network* which predicts the *log-likelihood* for a given feature vector consisting of model parameters, as well as a particular choice and reaction time. This neural network then acts as a standin for a *likelihood function* facilitating *approximate Bayesian inference*. Crucially these networks are then fully and flexibly reusable, for data deriving from any experimental design.

Note that in principle, our extension supports the integration of *any* approximate (or exact) likelihood, in the context of a now simple interface for adding models to HDDM. The scope remains limited only insofar as HDDM remains specialized towards choice / reaction time modeling.

4 HDDM Extension: Step by Step

A Central Database For Models: `hddm.model_config`

To accommodate the multitude of new models, HDDM > 0.9 now uses a model specification dictionary to extract data about a given model that is relevant for inference. The `hddm.model_config` module contains a central dictionary with which the user can interrogate to inspect models that are currently supplied with HDDM. Codeblock 4 shows how to list the models included by name.

```
hddm.model_config.model_config.keys()
```

Codeblock 4. `model_config` list available models.

For each model, we have a specification dictionary. Codeblock 5 provides an example for the simple DDM.

```
hddm.model_config.model_config["ddm"] =
    {
    "params": ["v", "a", "z", "t"],
    "params_trans": [0, 0, 1, 0],
    "param_bounds": [[-3.0, 0.3, 0.1, 1e-3], [3.0, 2.5, 0.9, 2.0]],
    "boundary": hddm.simulators.bf.constant,
    "hddm_include": ["z"],
    "choices": [-1, 1],
    "params_default": [0.0, 1.0, 0.5, 1e-3],
    "params_std_upper": [1.5, 1.0, None, 1.0],
    }
```

Codeblock 5. DDM specfic `model_config`

We focus on the most important aspects of this dictionary (more options are available). Under `"params"` the parameter names for the given model are listed. `"params_trans"` specifies if the sampler should *transform* the parameter at the given position (order follows the list supplied under `"params"`.[1] `"param_bounds"` lists the parameter-wise lower and upper bounds of parameters that the sampler can explore. This is important in the context of LAN based likelihoods, which are only valid in the range of parameters which were observed during training.[2]

HDDM uses the *inverse logistic* (or *logit*) transformation for the sampler to operate on an unconstrained parameter space. For a parameter θ and parameter bounds $[a, b]$, this transformation takes takes θ from a value in $[a, b]$ to a value x in $(-\infty, \infty)$ via,

$$x = \ln\left(\frac{\theta - a}{b - \theta}\right)$$

A given SSM usually has a `"decision boundary"` which is supplied as a function that can be evaluated over time-points $(t_0, ..., t_n)$, given boundary parameters (supplied implicitly via `"params"`). The values representing each choice are reported as a `list` under `"choices"`. A note of caution: If a user wants to estimate a new model that is not currently in HDDM, a new LAN (or generally likelihood) has to be created, for it to be added to the `model_config dictionary`. Simply changing a setting in an existing `model_config` dictionary will not work.

`"hddm_include"`, provides a working default for the `include` argument expected from the `hddm.HDDM` classes.

Lastly, `"params_default"` specify the parameter values that are fixed (*not fit*) by HDDM and `"params_std_upper"` specify upper bounds on group level standard deviations for each parameter (optional, but this can help constrain the parameter ranges proposed by the sampler, making it more efficient).

These `model_config` dictionaries provide a scaffolding for model specification which is applied throughout all of the new functionalities discussed in the next sections.

[1]Transforming parameters can be helpful for convergence, especially if the parameter space is strongly constrained a priori (e.g., between 0 and 1).

[2]We trained the LANs included in HDDM on a broad range of parameters, however it cannot be guaranteed that these were broad enough for any given empirical data set. If the provided LANs are deemed inappropriate for a given data set (e.g., if parameter estimates hit the bounds), it is always possible to retrain on an even broader range of parameters. Ruling out convergence issues however, should be the first order of business in such cases.

Batteries Included: `hddm.simulators`, `hddm.network_inspectors`

The new `HDDMnn` (where *nn* is for *neural network*), `HDDMnnRegressor` and `HDDMnnStimCoding` classes have access to a (growing) stock of supplied SSMs, enabling rapid compiled [2] simulators, and rapid likelihood evaluation via LANs [9] and their implementation in pytorch [34].

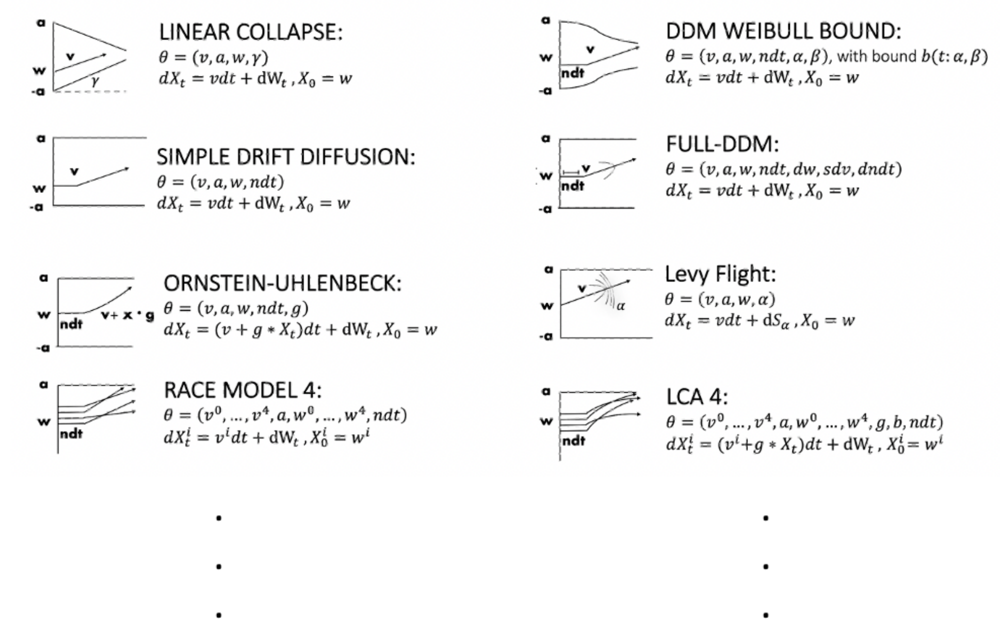

Figure 3. Graphical examples for some of the sequential sampling included in HDDM.

We will discuss how to fit these models to data in the next section. Here we describe how one can access the low level simulators and LANs directly, in case one wants to adopt them for custom purposes. We also show how to assess the degree to which the LAN approximates the true (empirical) likelihood for a given model. Users who just want to apply existing SSMs in HDDM to fit data can skip to the next section.

As described in the previous section, you can check which models are currently available in the `hddm.model_config.model_config` dictionary. For a given model, a doc-string includes some information (and possible warnings) about usage. As an example, let us pick the **angle** model, which is a DDM that allows for the decision boundary to decline linearly across time with some estimated angle. (Note that although other aspects of the model are standard DDM, even in this case the likelihood is analytically intractable. Nevertheless, we previously observed that inference using LANs yields good parameter recovery [9].) Executing codeblock 6, we get the following output,

```
print(hddm.model_config.model_config['angle']['doc'])
```

```
Model formulation is described in the documentation under LAN Extension.
Meant for use with the extension.
```

Codeblock 6. `angle` model specific `model_config`

Using the code in codeblock 7 we can simulate synthetic data from this model. Following the code, the variable `out` is now a three-tuple. The first element contains

```
from hddm.simulators import simulator
theta_tmp = hddm.model_config.model_config['angle']['params_default']
model = 'angle'
out = simulator(model = 'angle',
                theta = theta_tmp,
                n_samples = 100,
                delta_t = 0.001)
```

Codeblock 7. Using the `simulator` simulator

an array of *reaction times*, the second contains an array of *choices* and finally the third element returns a *dictionary of metadata* concerning the simulation run.

Next, we can access the LAN corresponding to our **angle** model directly by typing the code in codeblock 8.

```
lan_angle = hddm.network_inspectors.get_torch_mlp(model = 'angle')
```

Codeblock 8. Loading a torch network

`lan_angle` as defined in this codeblock is a method, which defines the forward pass through the LAN. It expects as input a matrix where each row defines a parameter vector suitable for the SSM of choice (here **angle**, so we need a value for each of the parameters `['t', 'a', 'v', 'z', 'theta']` which can be found in our `model_config` dictionary). Two elements are then added: a *reaction time* and a *choice* at which we would like to evaluate our likelihood. Codeblock 9 provides a full example. We can see the output below.

To facilitate a simple sanity check, we provide the `kde_vs_lan_likelihoods` plot, which can be accessed from the `hddm.network_inspectors` submodule.

This plot lets the user compare LAN likelihoods against empirical likelihoods from simulator data for a given matrix of parameter vectors [9]. The empirical likelihoods are defined via kernel density estimators (KDEs) [46]. We show an example in codeblock 10. Figure 4 shows the output.

Fitting data using HDDMnn, HDDMnnRegressor, and HDDMnnStimCoding classes

Using the `HDDMnn`, `HDDMnnRegressor` and `HDDMnnStimCoding` classes, we can follow the general workflow established by the basic `HDDM` package to perform Bayesian inference. In this section we will fit the **angle** model to the example data set provided with the HDDM package. Codeblock 11 shows us how to load the correponding dataset.

We can now set up our HDDM model, and draw 1000 MCMC samples using the code in codeblock 12.

We note a few differences between a call to construct a `HDDMnn` class and a standard `HDDM` class. First, the supply of the `model` argument specifying which SSM to fit (requires that this model is already available in HDDM; see above). Second, the inclusion of model-specific parameters under the `include` argument. The workflow is otherwise equivalent, a fact that is conserved for the `HDDMnnRegressor` and `HDDMnnStimCoding` classes. A third difference concerns the choice of argument defaults. The `HDDMnn` class uses non-informative priors, instead of the informative priors derived from the literature which form the default for the basic `HDDM` class. Since, as per our earlier discussions, variants of SSMs are historically rarely if ever fit to experimental

```
# Make some random parameter set
from hddm.simulators import make_parameter_vectors_nn

parameter_df = make_parameter_vectors_nn(model = model,
                        param_dict = None,
                        n_parameter_vectors = 1)

parameter_matrix = np.tile(np.squeeze(parameter_df.values),
                           (200, 1))

# Initialize network input
network_input = np.zeros((parameter_matrix.shape[0],
                          parameter_matrix.shape[1] + 2))
# Note the + 2 on the right
# we append the parameter vectors with
# reaction times (+1 columns) and choices (+1 columns)

# Add reaction times
network_input[:, -2] = np.linspace(0, 3,
                                   parameter_matrix.shape[0])

# Add choices
network_input[:, -1] = np.repeat(np.random.choice([-1, 1]),
                                 parameter_matrix.shape[0])

# Note: The networks expects float32 inputs
network_input = network_input.astype(np.float32)

# Show example output
print('Some network outputs')
print(lan_angle(network_input)[:10]) # printing the first 10 outputs
print('Shape')
print(lan_angle(network_input).shape) # original shape of output
```

```
Some network outputs
[[-2.9323568]
 [ 2.078088 ]
 [ 0.4104141]
 [-0.5943402]
 [-1.1136726]
 [-1.6901499]
 [-2.3512228]
 [-3.080151 ]
 [-3.8215086]
 [-4.4257374]]
Shape
(200, 1)
```

Codeblock 9. Check forward pass of supplied `angle` network.

data, we can not easily derive reasonable informative priors from the literature and 245

```python
from hddm.network_inspectors import kde_vs_lan_likelihoods

# Make a set of parameter vectors
parameter_df = make_parameter_vectors_nn(model = model,
                                         param_dict = None,
                                         n_parameter_vectors = 6)

# Generate plot
kde_vs_lan_likelihoods(parameter_df = parameter_df,
                       model = model,
                       n_samples = 1000,
                       n_reps = 10,
                       font_scale = 1.25)
```

Codeblock 10. Example usage of the `kde_vs_lan_likelihood()` function to compare LAN likelihoods to empirical kernel-density estimates.

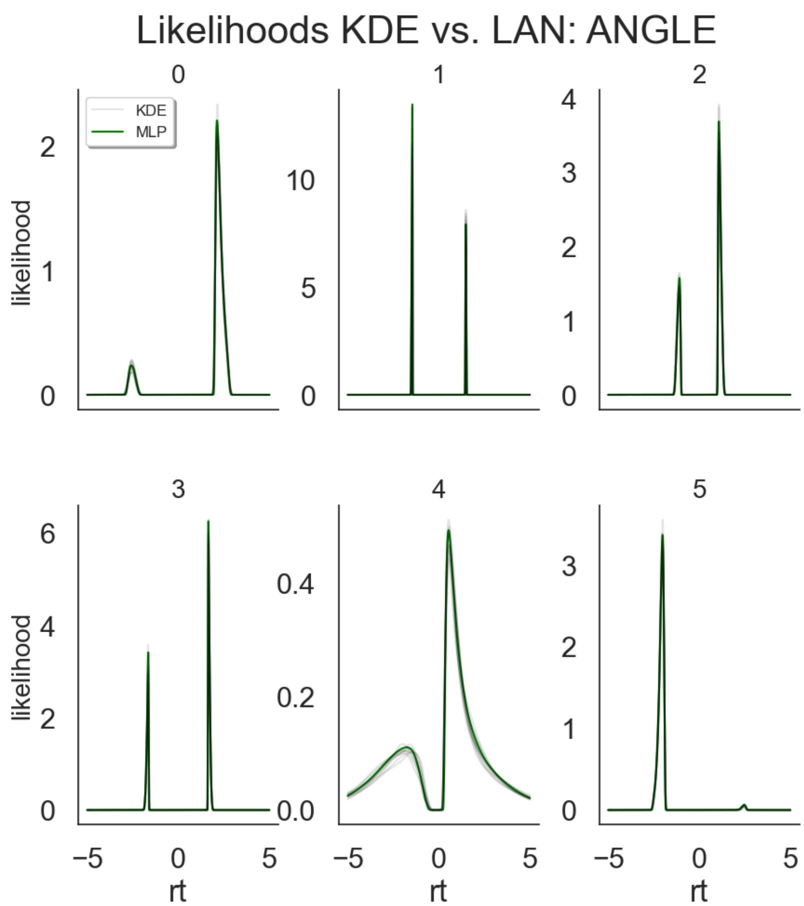

Figure 4. Example of a `kde_vs_lan_likelihoods` plot. If the green (deterministic) and gray (stochastic) lines overlap, then the approximate likelihood (MLP for multilayered perceptron, the neural network that provides our LAN) is a good fit to the actual likelihood.

```
cav_data = hddm.load_csv(hddm.__path__[0] + \
                '/examples/cavanagh_theta_nn.csv')
```

```
      subj_idx stim     rt  response     theta  dbs conf
0            0   LL  1.210       1.0  0.656275    1   HC
1            0   WL  1.630       1.0 -0.327889    1   LC
2            0   WW  1.030       1.0 -0.480285    1   HC
3            0   WL  2.770       1.0  1.927427    1   LC
4            0   WW  1.140       0.0 -0.213236    1   HC
...        ...  ...    ...       ...       ...  ...  ...
3983        13   LL  1.450       0.0 -1.237166    0   HC
3984        13   WL  0.711       1.0 -0.377450    0   LC
3985        13   WL  0.784       1.0 -0.694194    0   LC
3986        13   LL  2.350       0.0 -0.546536    0   HC
3987        13   WW  1.250       1.0  0.752388    0   HC

[3988 rows x 7 columns]
```

Codeblock 11. Loading package supplied **cavanagh** dataset.

```
hddmnn_model_cav = hddm.HDDMnn(cav_data,
                               model = 'angle',
                               include = ['z', 'theta'],
                               is_group_model = True)

hddmnn_model_cav.sample(1000, burn = 500)
```

```
 [-----------------100%-----------------]
 1001 of 1000 complete in 365.3 sec
```

Codeblock 12. Sampling from a HDDMnn model.

therefore choose to remain agnostic in our beliefs about the parameters underlying a given data set. If the research community starts fitting SSM variants to experimental data, this state of affairs may evolve through collective learning. At this point we caution the user to however not use these new models blindly. We strongly encourage conducting appropriate parameter recovery studies, specific to the experimental data set under consideration. We refer to the section on *inference validation tools* below, for how HDDM might help in this procedure.

New Visualization Plots: `hddm.plotting`

Based on our model fit from the previous section, we illustrate a few new informative plots, which are now included in HDDM. We can generally distinguish between two types of plots. Plots which use the traces only (to display posterior parameter estimates) and plots which make use of the model simulators (to display how well the model can reproduce empirical data given posterior parameters). The first such plot is produced by the the `plot_caterpillar` function, which presents an approximate posterior 99%-HDI (specifically we show the 1% to 99% range in the cumulative distribution function of the posterior), for each parameter. Codeblock 13 shows us how to invoke this function and Figure 5 illustrates the resulting plot.

```
from hddm.plotting import plot_caterpillar

plot_caterpillar(hddm_model = hddmnn_model_cav,
                 figsize = (8, 8),
                 columns = 3)
```

Codeblock 13. Example usage of the `caterpillar_plot()` function.

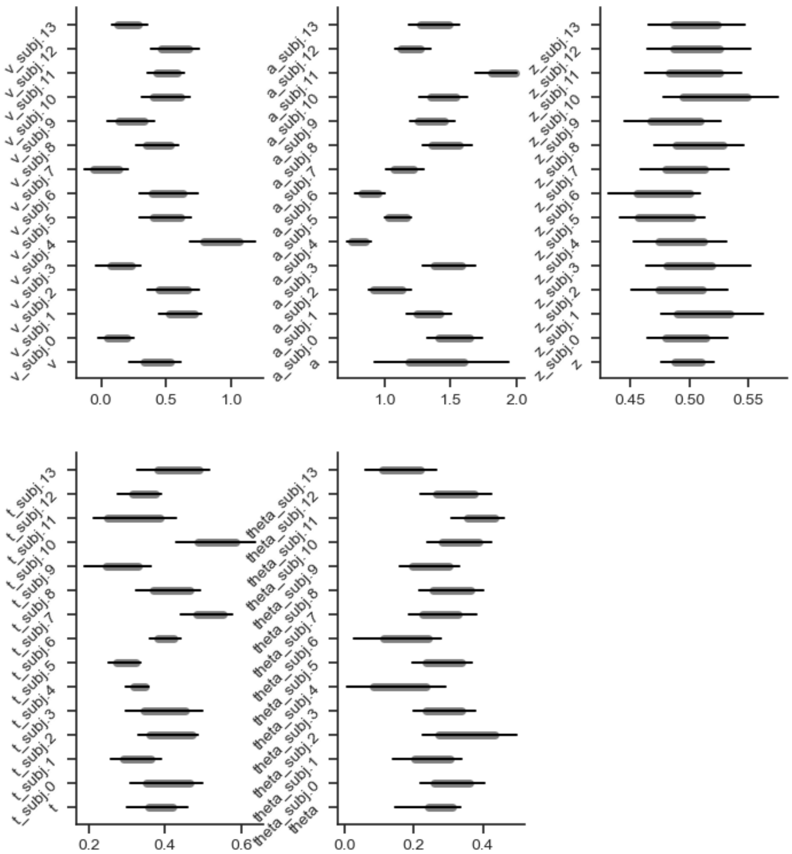

Figure 5. Example of a `caterpillar_plot`. The plot is split by model parameter, then shows parameter wise, the 99% (line-ends) and 95% (gray band ends) highest density intervals (HDIs) of the posterior. Multiple styling options exist.

The second such plot is the a posterior pair plot, called via the `plot_posterior_pair` function. This plot shows the pairwise posterior distribution, subject by subject (and, if provided, condition by condition). Codeblock 14 illustrates how to call this function, and Figure 6 exemplifies the resulting output.

```
from hddm.plotting import plot_posterior_pair

plot_posterior_pair(hddmnn_model_cav,
                    samples = 500,
                    figsize = (6, 6))
```

Codeblock 14. Example concerning usage of the `plot_posterior_pair()` function.

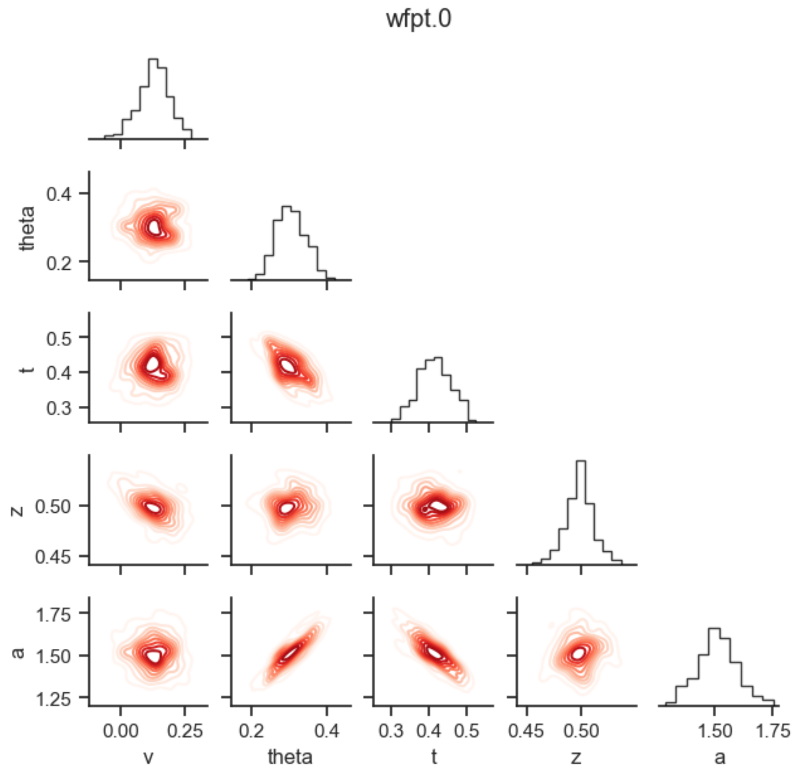

Figure 6. Example of a `posterior_pair_plot` in the context of parameter recovery. The plot is organized per stochastic node (here, grouped by the `'subj_idx'` column where in this example `'subj_idx' = '0'`). The diagonal shows the *marginal posterior* of a given parameter as a histogram. The elements below the diagonal show pair-wise posteriors via (approximate) level curves. These plots are especially useful to identify parameter collinearities, which indicate parameter-tradeoffs and can hint at issues with identifiability. This example shows how the `theta` (boundary collapse) and `a` (boundary separation) parameters as well as the `t` (non-decision time) and `a` parameters trade-off in the posterior. We refer to [9] for parameter recovery results using the underlying *angle* SSM. We note that such parameter trade-offs and attached identifiability issues derive not just from a given likelihood model, but are affected by the data and parameter structure as task design and modeling choices.

A last very useful plot addition is what we call the **model plot**, an extension to the standard posterior predictive plot, which can be used to visualize the impact of the parameter posteriors on decision dynamics. For example, if one is estimating a linear collapsing bound, instead of just interpreting the posterior angle parameter, one can see how that translates to the evolving decision bound over time in tandem with the estimating drift rate, etc. It is an extension of the `plot_posterior_predictive` function. This function operates by manipulating `matplotlib axes` objects, via a supplied *axes manipulator*. The novel *axis manipulator* in the example show in Codeblock 15 is the `_plot_func_model` function. Figure 7 show the resulting plot.

We use this moment to illustrate how the **angle** model in fact outperforms the **DDM** on this example dataset. For this purpose we take an example subject from Figure 7 and contrast the posterior predictive of the **angle** model with the posterior predictive of the **DDM** side by side in Figure 8. We clearly see that the **DDM** model has trouble capturing the leading edge and the tail behavior of the rt distributions simultaneously, while the **angle** model strikes a much better balance. While this

```
from hddm.plotting import plot_posterior_predictive

plot_posterior_predictive(model = hddmnn_model_cav,
                          columns = 3,
                          figsize = (10, 12),
                          groupby = ['subj_idx'],
                          value_range = np.arange(0.0, 3, 0.1),
                          plot_func = hddm.plotting._plot_func_model,
                          **{'alpha': 0.01,
                             'ylim': 3,
                             'samples': 200,
                             'legend_fontsize': 7.,
                             'legend_location': 'upper left',
                             'add_posterior_uncertainty_model': True,
                             'add_posterior_uncertainty_rts': False,
                             'subplots_adjust': {'top': 0.94,
                                                 'hspace': 0.35,
                                                 'wspace': 0.3}
                          })
```

Codeblock 15. Example usage of the `plot_posterior_predictive` function

example does not present a fully rigorous model comparison (DIC scores for example however bear out the same conclusion) exercise, it provides a hint at the benefits one may expect from utilizing the expanded model space that our HDDM extension brings forth.

Inference Validation Tools: `simulator_h_c()`

Validating that a model is identifiable on simulated data is an important aspect of a trustworthy inference procedure. We have two layers of uncertainty in this regard. First, LANs are approximate likelihoods. A model that is otherwise identifiable, could in principle lose this property when using LANs to estimate its parameters from a data set, should the LAN not have been trained adequately. Second, a given model can inherently be unidentifiable for a given data set and or theoretical commitments (regardless of whether its likelihood is analytic or approximate). A simple example is that a given experimental data set does not include enough samples to identify the parameter of a model of interest with any degree of accuracy. Slightly more involved, the posterior could tend to be multi-modal, a problem for MCMC samplers that can lead to faulty inference. While increasing the number of trials in an experiment and/or increasing the number of participants can help remedy this situation, this is not a guarantee. Apart from the size and structure of the empirical data set, our modeling commitments play an important role for identifiability too. As an example, we might have experimental data from a random dot motion task and we are interested in modeling the *choices* and *reaction times* of participating subjects with our **angle** model. A reasonable assumption is that the v parameter (roughly processing speed) differs depending on the difficulty of the trial, however the parameters t and a may not since we have not good a-priori theoretical reason to suspect that the *non-decision time* (t) and the initial the *boundary separation* a (the degree of evidence expected to take a decision) will differ across experimental conditions. These commitments are embedded in the model itself (they are assumption about the data generating process imposed by the modeler), and determine jointly with an experimental data set whether inference can be successful.

()

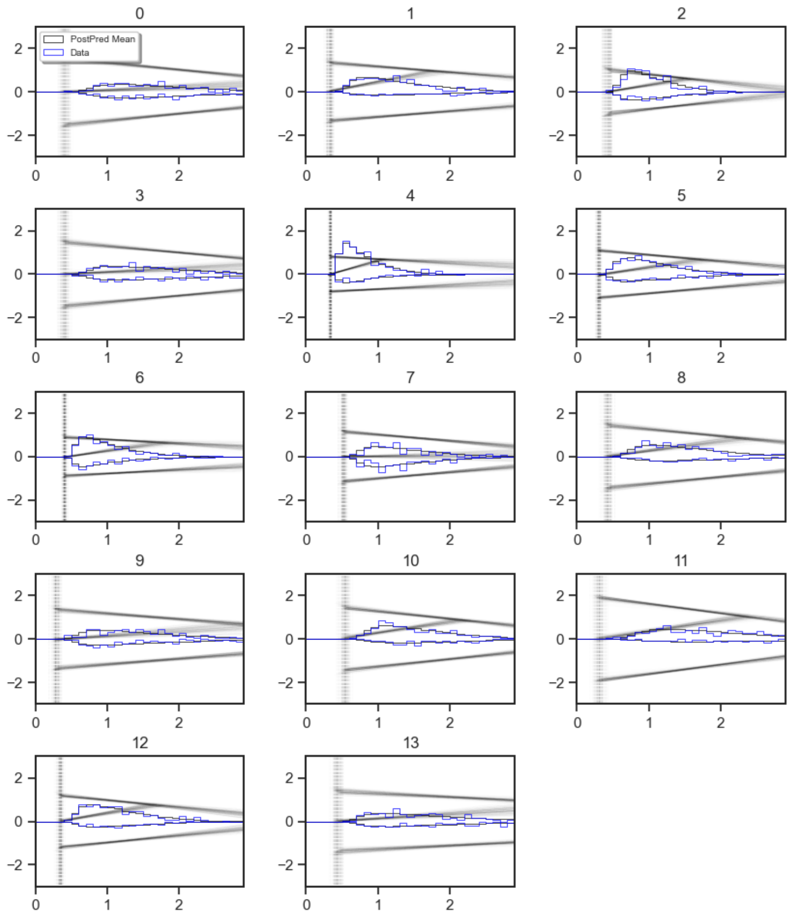

Figure 7. Example of a `model_plot`. This plot shows the underlying data in blue, choices and reaction times presented as a histograms (positive y-axis for choice option 1, negative y-axis or choice option 0 or -1). The Black histograms show the reaction times and choices under the parameters corresponding to the posterior mean. In addition the plot shows a graphical depiction of the model corresponding to parameters drawn from the posterior distribution in black. Various options exist to add and drop elements from this plot; the provided example corresponds to what we consider the most useful settings for purposes of illustration.

For a modeler it is therefore of paramount importance to check whether their chosen combinations of theoretical commitments and experimental data set jointly lead to an inference procedure that is accurate. Since the space of models incorporated into HDDM has been significantly expanded with the LAN extension, we provide a few tools to help facilitate parameter recovery studies which are relevant to real experimental data analysis and plan to supplement these tools even further in the future.

First, we provide the `simulator_h_c` function, in the `hddm_dataset_generators` submodule. The function is quite flexible, however we will showcase a particularly relevant use-case. Taking our `cav_data` data set loaded previously, we would like to

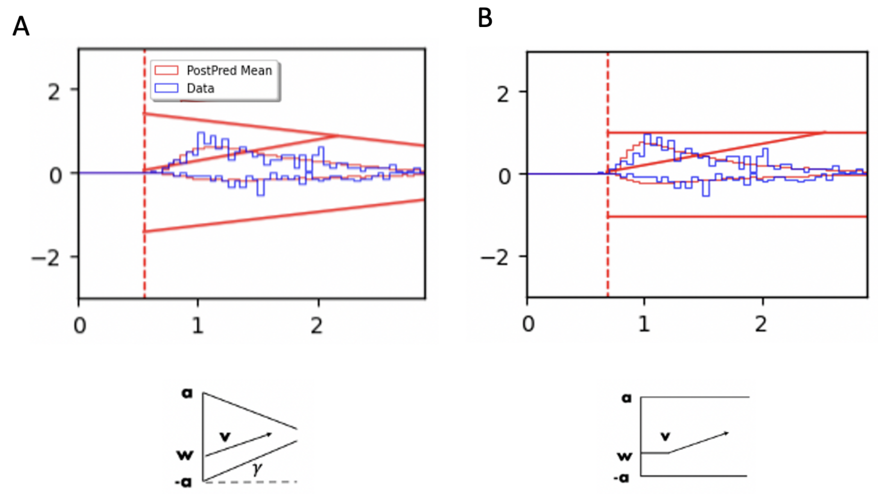

Figure 8. Contrasting the posterior predictive of the **angle** and **DDM** model on an example subject. **A**) shows the **angle** model and **B**) shows the **DDM**. We can clearly see how the **angle** model outperforms the **DDM** in capturing the leading edge of the rt distribution, while staying faithful to the tail behavior.

generate data from our **angle** model in such a way that we encode assumptions about our model into the generated data set. In the example below we assume that the v and *theta* parameters vary as a function of the "stim" column. For each value of "stim" a group level μ and σ (defining the *mean* and *standard deviation* of a group level Normal distribution) are generated and subject-level parameters are sampled from this group distribution. This mirrors exactly the modeling assumptions when specifying a HDDM model with the `depends_on` argument set to `{'v': 'stim', 'theta': 'stim'}`. Codeblock 16 provides an example on how to call this function.

The `simulator_h_c` function returns the respective data set (here `sim_data`) exchanging values in the previous *rt* and *response* columns with simulation data. Trial-by-trial parameters are attached to the dataframe as well. The `parameter_dict` dictionary contains all the parameters of the respective hierarchical model which was used to generate the synthetic data. This parameter dictionary follows the parameter naming conventions of HDDM exactly.

We can now fit this data using the `HDDMnn` class as illustrated in Codeblock 17.

The plots defined in the previous section allow us to specify a `parameter_recovery_mode` which we can utilize to check how well our estimation worked on our synthetic data set. Codeblocks 18, 19 and 20 and Figures 9, 10 and 11 show respectively code and plot examples.

Note how both the `plot_posterior_pair` function as well as the `plot_posterior_predictive` function take the `parameter_recovery_mode` argument to add a ground truth to the visualization automatically (the ground truth is expected to be included in the data set attached to the HDDM model itself). The `plot_caterpillar` function needs a `ground_truth_parameter_dict` argument to add the ground truth parameters. The `simulator_h_c` function provides such a compatible dictionary of ground truth parameters. Using the set of tools in this section, we hope that HDDM conveniently facilitates application relevant parameter recovery studies.

```python
# Generate some data
from hddm.simulators.hddm_dataset_generators import simulator_h_c
sim_data, parameter_dict = simulator_h_c(data = cav_data,
                                          model = 'angle',
                                          p_outlier = 0.00,
                                          depends_on = {'v': ['stim'],
                                                        'theta': ['stim']},
                                          regression_models = None,
                                          regression_covariates = None,
                                          group_only_regressors = False,
                                          group_only = None,
                                          fixed_at_default = None)

sim_data
```

```
      subj_idx stim        rt  response     theta  dbs conf         v  \
0            0   LL  2.020890       1.0  0.442532    1   HC -0.474451
1            0   WL  2.075889       1.0  0.844691    1   LC -0.865643
2            0   WW  2.119889       1.0  0.661660    1   HC  0.752663
3            0   WL  1.804893       0.0  0.844691    1   LC -0.865643
4            0   WW  2.410885       1.0  0.661660    1   HC  0.752663
...        ...  ...       ...       ...       ...  ...  ...       ...
3983        13   LL  2.874057       1.0  0.402371    0   HC -0.473813
3984        13   WL  2.169051       0.0  0.972350    0   LC -1.001207
3985        13   WL  1.798055       0.0  0.972350    0   LC -1.001207
3986        13   LL  1.709054       1.0  0.402371    0   HC -0.473813
3987        13   WW  2.115052       1.0  0.911009    0   HC  0.824063

             a         z         t
0     1.402356  0.577363  1.468893
1     1.402356  0.577363  1.468893
2     1.402356  0.577363  1.468893
3     1.402356  0.577363  1.468893
4     1.402356  0.577363  1.468893
...        ...       ...       ...
3983  1.283326  0.616165  1.618054
3984  1.283326  0.616165  1.618054
3985  1.283326  0.616165  1.618054
3986  1.283326  0.616165  1.618054
3987  1.283326  0.616165  1.618054

[3988 rows x 11 columns]
```

Codeblock 16. Using the `simulator_h_c()` function.

Adding to the bank of SSMs: User Supplied Custom Models 346

The new models immediately available for use with HDDM are just the beginning. 347
HDDM allows users to define their own models via adjusting the `model_config` and the 348
provision of custom likelihood functions. The goal of this functionality is two fold. First, 349
we aim to make HDDM maximally flexible for advanced users, cutting down red-tape to 350
allow creative usage. Second, we hope to motivate users to follow through with a 351

```
hddmnn_model_sim = hddm.HDDMnn(sim_data,
                               model = 'angle',
                               include = ['v', 'a', 't',
                                          'z', 'theta'],
                               is_group_model = True,
                               depends_on = {'v': ['stim'],
                                             'theta': ['stim']},
                               p_outlier = 0.00)

hddmnn_model_sim.sample(1000, burn = 500)
```

```
 [-----------------100%-----------------]
 1001 of 1000 complete in 1436.2 sec
```

Codeblock 17. Fitting a HDDMnn model to synthetic data.

```
from hddm.plotting import plot_caterpillar

plot_caterpillar(hddm_model = hddmnn_model_sim,
                 ground_truth_parameter_dict = parameter_dict,
                 figsize = (10, 15),
                 y_tick_size = 6,
                 columns = 3)
```

Codeblock 18. *Caterpillar plot* on fit to simulated data.

two-step process of model integration. Step one involves easy testing of new likelihoods through HDDM, however with somewhat limited auxiliary functionality (one can generate plots based on the posterior traces, but other plots will not work because of the lack of a simulator). Step two involves sharing the model likelihood and a suitable simulator with the community to allow full integration with HDDM as well as other similar toolboxes which operate across programming languages and probabilistic programming frameworks. In future work we hope to flesh out a pipeline that allows users to follow a simple sequence of steps to full integration of their custom models with HDDM. Here, we show how to complete step one, defining a `HDDMnn` model with a custom likelihood to allow fitting a new model through HDDM. See future work section for some guidance on producing your own LAN, or contact the authors.

We start with configuring the `model_config` dictionary. We add a `"custom"` key and assign the specifics of our new model. For illustration purposes we will add the **angle** model to HDDM (even though it is already provided with the LAN extension). Codeblock 21 illustrates.

Additionally we need to define a basic likelihood function that takes in a vector (or matrix / 2`d numpy array`) of parameters, ordered according to the list in the `"params"` key above. As an example, we load our LAN for the **angle** model (as supplied by HDDM) as if it is a custom network (illustrated in Codeblock 22.

We can then fit our newly defined *custom model* as per Codeblock 23

We note the only difference to a normal call to the `hddm.HDDMnn` class. We supply the `model` argument as `"custom"`, supply under `model_config` our own config dictionary and we explicitly add the `network` argument, our `custom_network` defining the likelihood.

```
from hddm.plotting import plot_posterior_predictive

plot_posterior_predictive(model = hddmnn_model_sim,
                          columns = 3,
                          figsize = (10, 12),
                          groupby = ['subj_idx'],
                          value_range = np.arange(0.0, 3, 0.1),
                          plot_func = hddm.plotting._plot_func_model,
                          parameter_recovery_mode = True,
                          **{'alpha': 0.01,
                          'ylim': 3,
                          'add_model': True,
                          'samples': 200,
                          'legend_fontsize': 7.,
                          'legend_location': 'upper left',
                          'add_posterior_uncertainty_rts': False,
                          'add_posterior_uncertainty_model': True,
                          'add_posterior_mean_model': True,
                          'add_posterior_mean_rts': True,
                          'subplots_adjust': {'top': 0.94,
                                              'hspace': 0.35,
                                              'wspace': 0.3}
                          })
```

Codeblock 19. Model plot for fit to simulated data.

```
from hddm.plotting import plot_posterior_pair
posterior_pair_plot(hddmnn_model_sim,
                    parameter_recovery_mode = True,
                    samples = 500,
                    figsize = (6, 6))
```

Codeblock 20. Posterior pair plot for fit to simulated data.

Connecting SSMs with Reinforcement Learning

While above we focused on SSMs in stationary environments, a host of commonly applied experimental task paradigms involve some form of learning that results from the agent's interactions with the environment. While SSMs can be used to model the decision processes, we need additional machinery to capture the learning dynamics that arise while subjects perform such tasks. Reinforcement learning (RL) [49] is one computational framework which can allow us to account for such learning processes. In reinforcement learning, researchers typically assume a simple *softmax* choice rule, informed by some 'utility' (or 'goodness') measure of taking a particular action in a given state. Mathematically choice probabilities are expressed as,

$$p(action_i; t) = \frac{e^{q_{action,i}(t)}}{\sum_j e^{q_{action,j}(t)}}$$

While reinforcement learning models can account for learning dynamics in basic choice behavior, the choice functions commonly employed (e.g., softmax) cannot capture the reaction time. To combine the strengths of sequential sampling models and

```
my_model_config = {
            "params": ["v", "a", "z", "t", "theta"],
            "params_trans": [0, 0, 1, 0, 0],
            "params_std_upper": [1.5, 1.0, None, 1.0, 1.0],
            "param_bounds": [[-3.0, 0.3, 0.1, 1e-3, 0.0],
                             [3.0, 2.5, 0.9, 2.0, 1.1]],
            "boundary": hddm.simulators.bf.constant,
            "params_default": [0.0, 1.0, 0.5, 1e-3],
            "hddm_include": ["z"],
            "choices": [-1, 1],
            "slice_widths": {"v": 1.5, "v_std": 1,
                             "a": 1, "a_std": 1,
                             "z": 0.1, "z_trans": 0.2,
                             "t": 0.01, "t_std": 0.15},
            }
```

Codeblock 21. Constructing a custom model config.

```
from hddm.torch.mlp_inference_class import load_torch_mlp

custom_network = load_torch_mlp(model = 'angle')
```

Codeblock 22. Loading a custom network.

```
hddm_model_custom = hddm.HDDMnn(data = data,
                                include = ["z", "theta"],
                                model = 'custom',
                                model_config = my_model_config,
                                network = custom_network)

hddm_model_custom.sample(1000, burn = 500)
```

Codeblock 23. Constructing a HDDMnn model using a custom network.

reinforcement learning models, recent studies have used the drift diffusion model to jointly model choice and response time distributions during learning [11,36,37]. Such an approach allows researchers to study not only the across-trial dynamics of learning but the within-trial dynamics of choice processes, using a single model. The main idea behind these models is to allow a reinforcement learning process to drive the trial-by-trial parameters of a sequential sampling model (such as the basic drift diffusion model), which in turn is used to jointly capture reaction time and choice behavior for a given trial. This can be applied in complex tasks which involve learning from feedback (see Figure 12). This results in a much more broadly applicable class of models. It naturally lends itself for use in computational modeling of numerous cognitive tasks where the 'learning process' informs the 'decision-making process'. Indeed, a recent study showed that joint modeling of choice and RT data can improve parameter identifiability of RL models, by providing additional information about choice dynamics [1]. However, to date, such models have been limited by the form the decision model. Many RL tasks involve more than two responses, making the DDM inapplicable. Similarly, the assumption of a fixed threshold may not be valid. For example, early during learning differences in Q values, and hence drift rates, will be close to zero and there is little value in accumulating evidence. A standard DDM model would predict that such choices are associated with very long tail RT distributions. A more appropriate assumption would be that learners use a collapsing bound so that when no evidence is present the decision process can terminate.

Utilizing the power of LANs, we can now further generalize the RL-DDM framework to include a much broader class of SSMs for the 'decision-making process' part. The rest of this section provides some details and code examples for these new RL-SSMs.

Test-bed We test our method on a synthetic dataset of the two-armed bandit task with binary outcomes. However, our approach can be generalized to any n-armed bandit task given a pre-trained LAN that outputs likelihoods for the corresponding n-choice decision process (e.g. race models). The model employed a simple delta learning rule [43] to update the action values

$$q_{action,i}(t+1) = q_{action,i}(t) + \alpha * [r(t) - q_{action,i}(t)],$$

where $q_{action}(t)$ denotes expected reward (Q-value) for the chosen action at time t, $r(t)$ denotes reward obtained at time t and α (referred to as *rl_alpha* in the result plots) denotes the learning rate. The trial-by-trial drift rate depends on the expected reward value learned by the RL rule. The drift rate is therefore a function of Q-value updates, and is computed by the following linking function

$$v(t) = [q_{action,1}(t) - q_{action,2}(t)] * s,$$

where s is a scaling factor of the difference in Q-values. In other words, the scalar s is the drift rate when the difference between the Q-values of both the actions is exactly one (Note that we refer to the scalar s as v in the corresponding figure). We show an example parameter recovery plot for this Rescorla-Wagner learning model connected to a SSM with collapsing bound in Fig. 13.

Model definitions for RL with `model_config_rl` Just like the `model_config`, the new HDDM version includes `model_config_rl`, which is the central database for the RL models used in the RLSSM settings. For each model, we have a specification dictionary which is specified as in the `model_config`. Below is an example for simple Rescorla-Wagner updates [43], a basic reinforcement learning rule. The learning rate (referred to as 'rl_alpha' in the result plot (Fig. 13) to avoid nomenclature conflicts with the 'alpha' parameter in some SSMs) is the only parameter in the update rule. We do not transform this parameter (`"params_trans"` is set to 0) and specify the parameter bounds for the sampler as $[0, 1]$. Note that for hierarchical sampling, the learning rate parameter α is transformed internally in the package. Therefore, the output trace for the learning rate parameter must be transformed by an inverse-logit function, $\frac{1}{(1+exp(-\alpha))}$ to get the learning rate values back in range $[0, 1]$. Codeblock 24 shows us an example of such a `model_config_rl` dictionary.

Analyzing instrumental learning data: The `HDDMnnRL` class Running `HDDMnnRL` presents only a few slight adjustments compared to the other `HDDM` classes. First, the data-frame containing the experimental data should be properly formatted. For every subject in each condition, the trials must be sorted in ascending order to ensure proper RL updates. The column `split_by` identifies each row with a specific task condition (as integer). The `feedback` column gives the reward feedback on the current trial and `q_init` denotes the initial q-values for the model. The rest of the data columns are the same as in other `HDDM` classes. Codeblock 25 provides an example.

```
hddm.model_config_rl.model_config_rl["RWupdate"] =
    {
        "doc": "Rescorla-Wagner update rule.",
        "params": ["rl_alpha"],
        "params_trans": [0],
        "params_std_upper": [10],
        "param_bounds": [[0.0], [1.0]],
        "params_default": [0.5],
    }
```

Codeblock 24. `model_config` definition for RL-SSM models.

```
import pandas as pd
data = pd.read_csv(hddm.__path__[0] + \
                '/examples/demo_HDDMnnRL/rlssm_data.csv')
```

```
      response  rt        feedback  subj_idx split_by trial    q_init
0        0.0    2.729579  0.0       0        0        1        0.5
1        1.0    3.090593  1.0       0        0        2        0.5
2        1.0    3.892617  1.0       0        0        3        0.5
3        1.0    2.429583  1.0       0        0        4        0.5
4        1.0    2.566581  1.0       0        0        5        0.5
...      ...    ...       ...       ...      ...      ...      ...
29995    1.0    3.381547  1.0       19       2        496      0.5
29996    1.0    3.324544  0.0       19       2        497      0.5
29997    1.0    3.132535  0.0       19       2        498      0.5
29998    0.0    3.206539  0.0       19       2        499      0.5
29999    1.0    5.009474  0.0       19       2        500      0.5

[30000 rows × 7 columns]
```

Codeblock 25. Reading in RL-SSM example data.

We can fit the data loaded in Codeblock 25 using the `HDDMnnRL` class. We showcase such a fit using the *weibull model* in conjunction with the classic Rescorla-Wagner learning rule [43]. The `HDDMnnRL` class definition (shown in Codeblock 26) takes a few additional arguments compared to the `HDDMnn` class: `"rl_rule"` specifies the RL update rule to be used and `non_centered` flag denotes if the RL parameters should be re-parameterized to avoid troublesome sampling from the neck of the funnel of probability densities [3, 33].

```
rlssm_model = hddm.HDDMnnRL(data,
    model="weibull",
    rl_rule="RWupdate",
    non_centered=True,
    include=["z", "alpha", "beta", "rl_alpha"],
    p_outlier=0.0)

rlssm_model.sample(3000, burn=1500)
```

Codeblock 26. Constructing and sampling from a HDDMnnRL model.

Figure 13 shows a caterpillar plot to verify the LAN-based parameter recovery on a sample RLSSM model.

Neural Regressors for RLSSM with the `HDDMnnRLRegressor` class We also include a `HDDMnnRLRegressor` class, which is aimed at capturing even richer (learning or choice) dynamics informed by neural activity, just like the `HDDMnnRegressor` class described above does for basic SSMs. The extension works the same as the bespoke `HDDMnnRegressor` class, except that the model is now informed by a reinforcement learning process to account for the across-trial dynamics of learning. The method allows estimation of the parameters (coefficients and intercepts) linking the neural activity in a given region and time point to the RLSSM parameters.

The usage of `HDDMnnRLRegressor` class is the same as `HDDMnnRL` class except that our dataframe will now have additional column(s) for neural (or other, e.g. EEG, pupil dilation etc.) trial-by-trial covariates. Just as when using the `HDDMnnRegressor` class, the model definition will also include specifying regression formulas which link covariates to model parameters. For example, if the boundary threshold parameter a is dependent on some neural measure $neural\_reg$, Codeblock 27 shows us how to specify a corresponding HDDMnnRLRegressor model.

```
rlssm_reg_model = hddm.HDDMnnRLRegressor(data,
    'a ~ 1 + neural_reg',
    model="weibull",
    rl_rule="RWupdate",
    include=["z", "alpha", "beta", "rl_alpha"],
    p_outlier=0.0)

rlssm_reg_model.sample(3000, burn=1500)
```

Codeblock 27. Constructing and sampling from a HDDMnnRLRegressors model.

More Resources

The original HDDM [58] paper as well as the original HDDMrl paper [36] are good resources on the basics of HDDM. The documentation provides examples for many complex use cases, including a long tutorial specifically designed to illustrate the `HDDMnn` classes and another tutorial specifically designed to showcase the `HDDMnnRL` classes. Through the hddm user group, an active community of HDDM users, one can find support on many problems and use cases which may not come up in the official documentation or published work.

5 Concluding Thoughts

We hope this tutorial can help kick-start a more widespread application of SSMs in the analysis of experimental choice and reaction time data. We consider the initial implementation with focus on LANs [9] as a starting point, which allows a significant generalization of the model space that can be considered by experimenters. The ultimate goal however is to lead towards community engagement, providing an easy interface for the addition of custom models as a start, which could greatly expand the space of models accessible to research groups across the world. We elaborate on a few possible directions for advancements in the next section.

6 Limitations and Future Work

The presented extension to HDDM greatly expands the capabilities of a tried and tested python toolbox, popular in the cognitive modeling sphere. However, using HDDM as the vehicle of choice, limitations endemic to the toolbox design remain and warrant a look ahead. First, HDDM is based on pymc2 [35] a probabilistic modeling framework that has since been superseded by it's successor pymc3 [44] (pymc 4.0.0, a rebranded pymc has just been released too). Since pymc2 is not an evolving toolbox, HDDM is currently bound to fairly basic MCMC algorithms, specifically a coordinate-wise slice sampler [29]. While we have confirmed adequate posterior sampling and estimation using our LANs, estimation may be rendered more efficient if one were to leverage more recent MCMC algorithms such as hamiltonian monte carlo [21]. Moreover, new libraries have emerged that act as independent functionality providers for other probabilistic programming frameworks, e.g. the ArViz [24] python library which provides a wide array of capabilities from posterior visualizations to the computation of model comparison metrics such as the WAIC [57]. While custom scripts can be used currently to deploy ArViz within HDDM, we are working on a successor to HDDM (we dub it HSSM) which will be built on top of one or more of these modern probabilistic programming libraries. Second, we realize that a major bottleneck in the wider adoption of LANs (and other likelihood approximators), lies in the supply of amortizers. While our extension comes batteries included, we focused on supplying a few SSM variants of proven interest in the literature, as well as some that we used for our or lab-adjacent research. It is not HDDM, but user friendly training pipelines for amortizers, which we believe to spur the quantum leap in activity in this space. Although we are working on the supply of such a pipeline for LANs [9], our hope is that the community will provide many alternatives. Third, we caution against uninformed use of approximate likelihoods. Before basing results of empirical studies on inference performed with LANs or other approximate likelihoods (e.g. user supplied), it is essential to test for the quality of inference that may be expected. Inference can be unreliable in manifold ways [14, 15, 50]. Parameter recovery studies and calibration tests, e.g. simulation based calibration [50] should form the backbone of trust in reported analysis on empirical (experimental) data sets. To help the application of a universal standard of rigor, we are working on a set of guidelines, such as a suggested battery of tests to pass before given user supplied likelihoods should be made available to the public. Other interesting work in this sphere is emerging [20, 26].

Acknowledgments

We thank Lakshmi N Govindarajan for useful tips concerning code quality. We also thank Thomas Wiecki for reviewing the additions to the HDDM package.

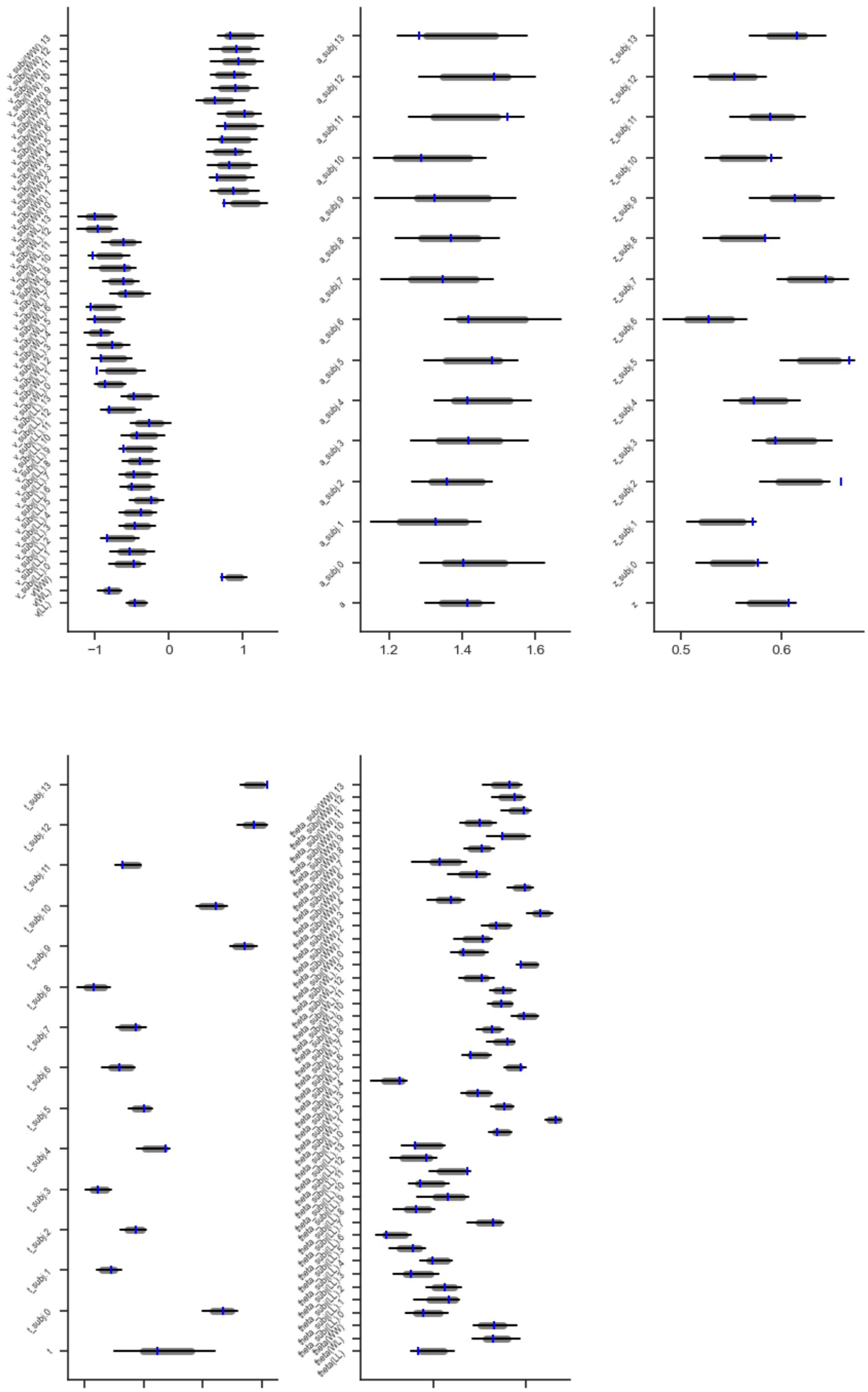

Figure 9. Example of a `caterpillar_plot`. The plot is split by model parameter kind, then shows parameter wise, the 99% (line-ends) and 95% (gray band ends) highest density intervals (HDIs) of the posterior. In the context of paramter recovery studies, the user can provide ground-truth parameters to the plot, which will be shown as blue tick-marks on top of the HDIs. Multiple styling options exist.

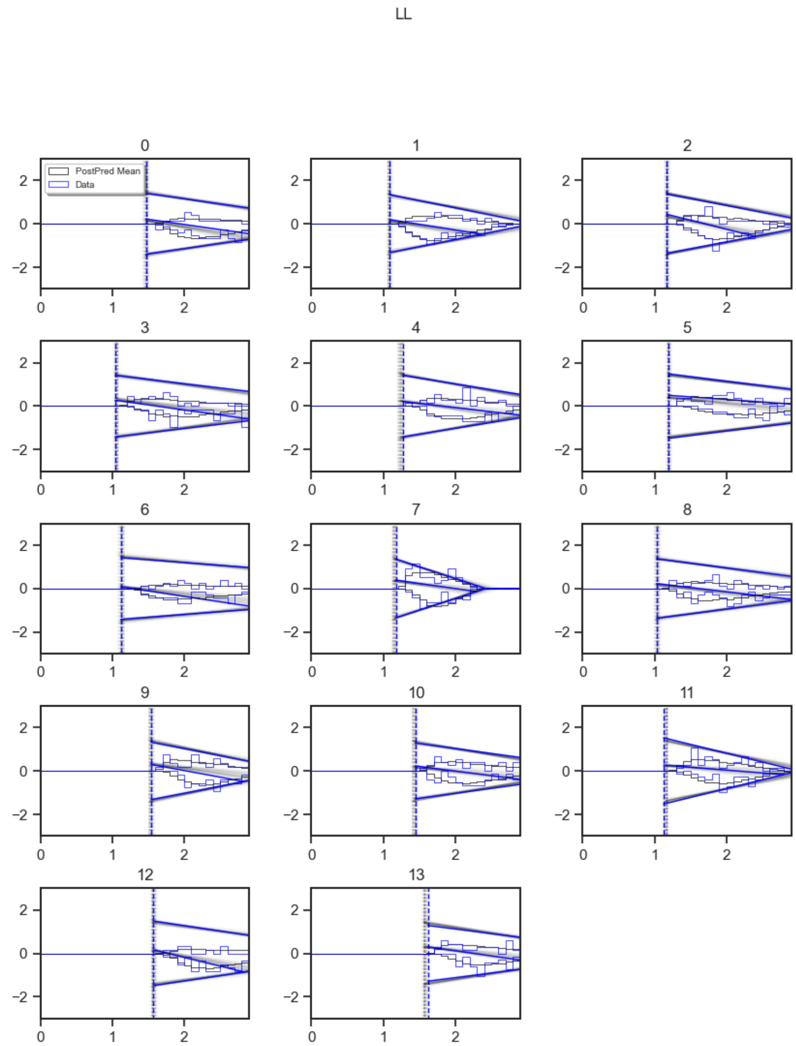

Figure 10. Example of a `model_plot`. This plot shows the underlying data in blue, choices and reaction times presented as a histograms (positive y-axis for choice option 1, negative y-axis or choice option 0 or -1). The Black histograms show the reaction times and choices under the parameters corresponding to the posterior mean. In addition the plot shows a graphical depiction of the model corresponding to parameters drawn from the posterior distribution in black, as well as such a depiction for the *ground truth* parameters, in case these were provided (e.g., if one is performing recovery from simulated data). Various options exist to add and drop elements from this plot, the provided example corresponds to what we consider the most useful settings for purposes of illustration.

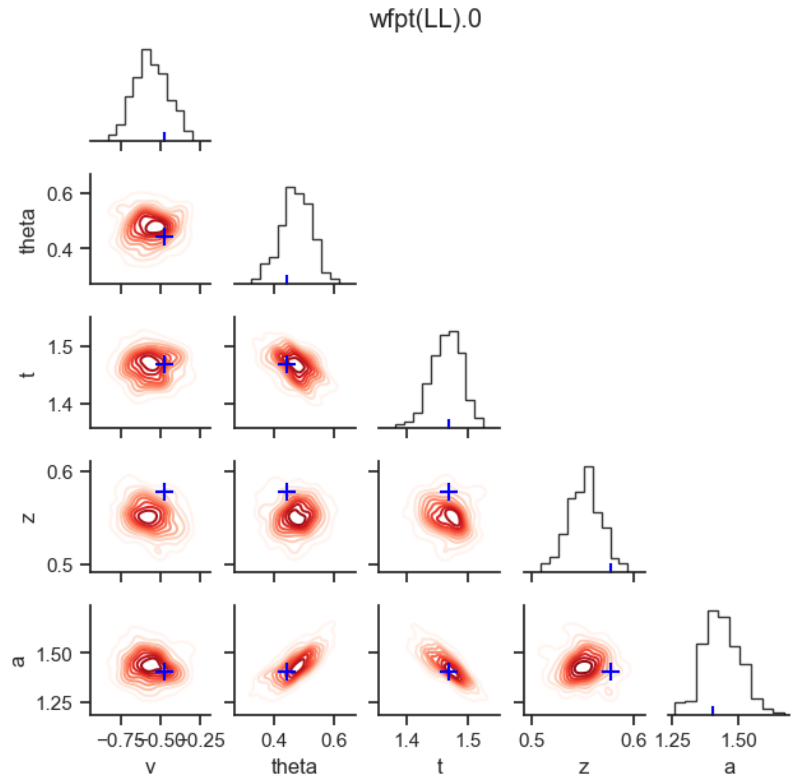

Figure 11. Example of a `posterior_pair_plot` in the context of parameter recovery. The plot is organized per stochastic node (here, grouped by the `'stim'` and `'subj_idx'` columns where in this example (`'stim' = 'LL'`, `'subj_idx' = '0'`). The diagonal show the *marginal posterior* of a given parameter as a histogram, adding the *ground truth* parameter as a blue tick-mark. The elements below the diagonal show pair-wise posteriors via (approximate) level curves, and add the respective *ground truths* as a blue cross.

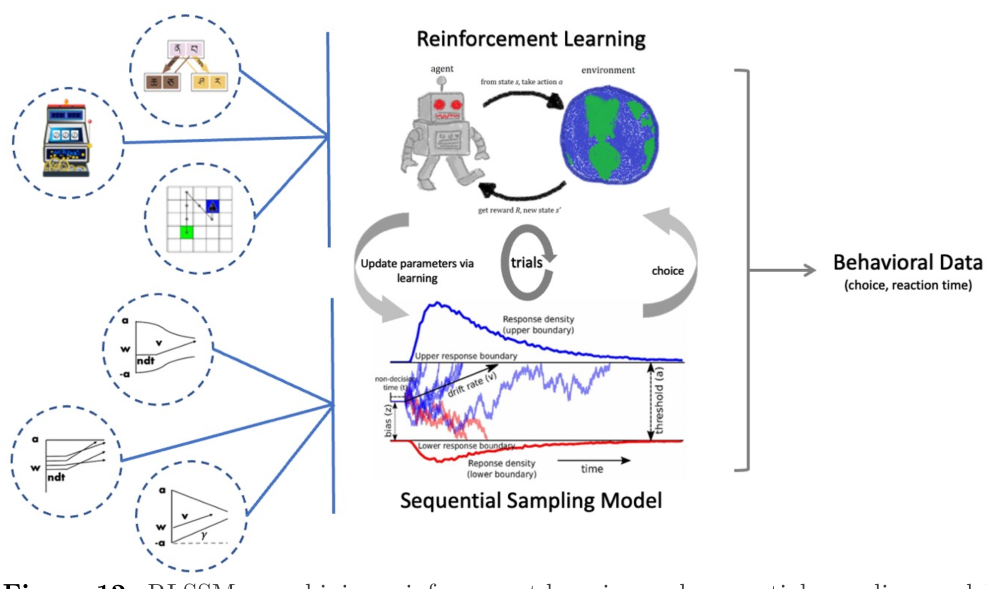

Figure 12. RLSSM - combining reinforcement learning and sequential sampling models.

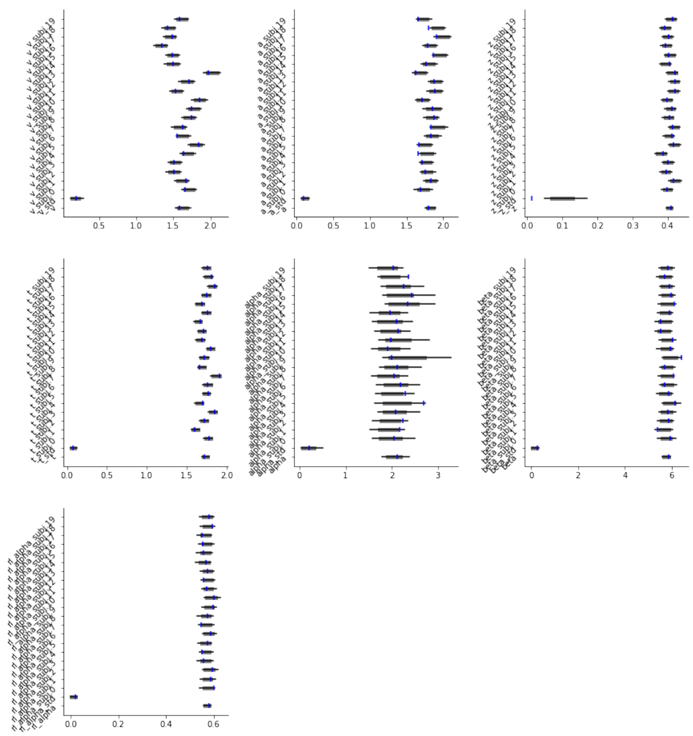

Figure 13. Parameter recovery on a sample synthetic dataset using RL+Weibull model. Posterior distributions for subject-level and group-level parameters are shown using caterpillar plots. The thick black lines correspond to 5-95 percentiles, thin black lines correspond to 1-99 percentiles. The blue ticks mark the ground truth values.

References

1. I. C. Ballard and S. M. McClure. Joint modeling of reaction times and choice improves parameter identifiability in reinforcement learning models. *Journal of Neuroscience Methods*, 317:37–44, 2019.

2. S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

3. M. J. Betancourt and M. Girolami. Hamiltonian monte carlo for hierarchical models, 2013.

4. U. Boehm, S. Cox, G. Gantner, and R. Stevenson. Fast solutions for the first-passage distribution of diffusion models with space-time-dependent drift functions and time-dependent boundaries. *Journal of Mathematical Psychology*, 105:102613, 2021.

5. S. P. Brooks and A. Gelman. General methods for monitoring convergence of iterative simulations. *Journal of computational and graphical statistics*, 7(4):434–455, 1998.

6. P. Cisek, G. A. Puskas, and S. El-Murr. Decisions in changing conditions: the urgency-gating model. *Journal of Neuroscience*, 29(37):11560–11571, 2009.

7. P. Cisek, G. A. Puskas, and S. El-Murr. Decisions in changing conditions: the urgency-gating model. *Journal of Neuroscience*, 29(37):11560–11571, 2009.

8. T. Doi, Y. Fan, J. I. Gold, and L. Ding. The caudate nucleus contributes causally to decisions that balance reward and uncertain visual information. *ELife*, 9:e56694, 2020.

9. A. Fengler, L. N. Govindarajan, T. Chen, and M. J. Frank. Likelihood approximation networks (lans) for fast inference of simulation models in cognitive neuroscience. *eLife*, 2021.

10. L. Fontanesi. rlssm.

11. L. Fontanesi, S. Gluth, M. S. Spektor, and J. Rieskamp. A reinforcement learning diffusion decision model for value-based decisions. *Psychonomic bulletin & review*, 26(4):1099–1121, 2019.

12. B. U. Forstmann, A. Anwander, A. Schäfer, J. Neumann, S. Brown, E.-J. Wagenmakers, R. Bogacz, and R. Turner. Cortico-striatal connections predict control over speed and accuracy in perceptual decision making. *Proceedings of the National Academy of Sciences*, 107(36):15916–15920, 2010.

13. M. J. Frank, C. Gagne, E. Nyhus, S. Masters, T. V. Wiecki, J. F. Cavanagh, and D. Badre. fmri and eeg predictors of dynamic decision parameters during human reinforcement learning. *Journal of Neuroscience*, 35(2):485–494, 2015.

14. A. Gelman, D. B. Rubin, et al. Inference from iterative simulation using multiple sequences. *Statistical science*, 7(4):457–472, 1992.

15. J. Geweke. Evaluating the accuracy of sampling-based approaches to the calculations of posterior moments. *Bayesian statistics*, 4:641–649, 1992.

16. J. I. Gold and M. N. Shadlen. The neural basis of decision making. *Annual review of neuroscience*, 30, 2007.

17. D. Greenberg, M. Nonnenmacher, and J. Macke. Automatic posterior transformation for likelihood-free inference. In *International Conference on Machine Learning*, pages 2404–2414. PMLR, 2019.

18. M. U. Gutmann, R. Dutta, S. Kaski, and J. Corander. Likelihood-free inference via classification. *Statistics and Computing*, 28(2):411–425, 2018.

19. G. E. Hawkins, B. U. Forstmann, E.-J. Wagenmakers, R. Ratcliff, and S. D. Brown. Revisiting the evidence for collapsing boundaries and urgency signals in perceptual decision-making. *Journal of Neuroscience*, 35(6):2476–2484, 2015.

20. J. Hermans, A. Delaunoy, F. Rozet, A. Wehenkel, and G. Louppe. Averting a crisis in simulation-based inference. *arXiv preprint arXiv:2110.06581*, 2021.

21. M. D. Hoffman and A. Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

22. I. Krajbich, D. Lu, C. Camerer, and A. Rangel. The attentional drift-diffusion model extends to simple purchasing decisions. *Frontiers in psychology*, 3:193, 2012.

23. I. Krajbich and A. Rangel. Multialternative drift-diffusion model predicts the relationship between visual fixations and choice in value-based decisions. *Proceedings of the National Academy of Sciences*, 108(33):13852–13857, 2011.

24. R. Kumar, C. Carroll, A. Hartikainen, and O. Martin. Arviz a unified library for exploratory analysis of bayesian models in python. *Journal of Open Source Software*, 4(33):1143, 2019.

25. J.-M. Lueckmann, G. Bassetto, T. Karaletsos, and J. H. Macke. Likelihood-free inference with emulator networks. In *Symposium on Advances in Approximate Bayesian Inference*, pages 32–53. PMLR, 2019.

26. J.-M. Lueckmann, J. Boelts, D. Greenberg, P. Goncalves, and J. Macke. Benchmarking simulation-based inference. In *International Conference on Artificial Intelligence and Statistics*, pages 343–351. PMLR, 2021.

27. W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

28. R. M. Neal. *Bayesian learning for neural networks*. PhD thesis, University of Toronto, 1995.

29. R. M. Neal. Slice sampling. *Annals of statistics*, pages 705–741, 2003.

30. G. Papamakarios and I. Murray. Fast ε-free inference of simulation models with bayesian conditional density estimation. In *Advances in Neural Information Processing Systems*, pages 1028–1036, 2016.

31. G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *arXiv preprint*, arXiv:1912.02762, 2019.

32. G. Papamakarios, D. Sterratt, and I. Murray. Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 837–848. PMLR, 2019.

33. O. Papaspiliopoulos, G. O. Roberts, and M. Sköld. A general framework for the parametrization of hierarchical models. *Statistical Science*, 22, 2007.

34. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

35. A. Patil, D. Huard, and C. J. Fonnesbeck. Pymc: Bayesian stochastic modelling in python. *Journal of statistical software*, 35(4):1, 2010.

36. M. L. Pedersen and M. J. Frank. Simultaneous hierarchical bayesian parameter estimation for reinforcement learning and drift diffusion models: a tutorial and links to neural data. *Computational Brain & Behavior*, 3:458–471, 2020.

37. M. L. Pedersen, M. J. Frank, and G. Biele. The drift diffusion model as the choice rule in reinforcement learning. *Psychonomic bulletin & review*, 24(4):1234–1251, 2017.

38. A. Rangel, C. Camerer, and P. R. Montague. A framework for studying the neurobiology of value-based decision making. *Nature reviews neuroscience*, 9(7):545–556, 2008.

39. R. Ratcliff. A theory of memory retrieval. *Psychological review*, 85(2):59, 1978.

40. R. Ratcliff and M. J. Frank. Reinforcement-based decision making in corticostriatal circuits: mutual constraints by neurocomputational and diffusion models. *Neural computation*, 24(5):1186–1229, 2012.

41. R. Ratcliff, P. L. Smith, S. D. Brown, and G. McKoon. Diffusion decision model: Current issues and history. *Trends in cognitive sciences*, 20(4):260–281, 2016.

42. R. Ratcliff, A. Thapar, and G. McKoon. Aging, practice, and perceptual tasks: a diffusion model analysis. *Psychology and aging*, 21(2):353, 2006.

43. R. A. Rescorla. A theory of pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. *Current research and theory*, pages 64–99, 1972.

44. J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

45. M. Shinn, N. H. Lam, and J. D. Murray. A flexible framework for simulating and fitting generalized drift-diffusion models. *Elife*, 9:e56938, 2020.

46. B. W. Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

47. P. L. Smith, R. Ratcliff, and D. K. Sewell. Modeling perceptual discrimination in dynamic noise: Time-changed diffusion and release from inhibition. *Journal of Mathematical Psychology*, 59:95–113, 2014.

48. D. J. Spiegelhalter, N. G. Best, B. P. Carlin, and A. Van der Linde. The deviance information criterion: 12 years on. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 76(3):485–493, 2014.

49. R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

50. S. Talts, M. Betancourt, D. Simpson, A. Vehtari, and A. Gelman. Validating bayesian inference algorithms with simulation-based calibration. *arXiv preprint arXiv:1804.06788*, 2018.

51. A. Tejero-Cantero, J. Boelts, M. Deistler, J.-M. Lueckmann, C. Durkan, P. J. Gonçalves, D. S. Greenberg, and J. H. Macke. sbi: A toolkit for simulation-based inference. *Journal of Open Source Software*, 5(52):2505, 2020.

52. B. M. Turner and P. B. Sederberg. A generalized, likelihood-free method for posterior estimation. *Psychonomic bulletin & review*, 21(2):227–250, 2014.

53. B. M. Turner and T. Van Zandt. Approximating bayesian inference through model simulation. *Trends in Cognitive Sciences*, 22(9):826–840, 2018.

54. M. Usher and J. L. McClelland. The time course of perceptual choice: the leaky, competing accumulator model. *Psychological review*, 108(3):550, 2001.

55. J. Vandekerckhove and F. Tuerlinckx. Diffusion model analysis with matlab: A dmat primer. *Behavior research methods*, 40(1):61–72, 2008.

56. A. Voss, V. Lerche, U. Mertens, and J. Voss. Sequential sampling models with variable boundaries and non-normal noise: A comparison of six models. *Psychonomic bulletin & review*, 26(3):813–832, 2019.

57. S. Watanabe. A widely applicable bayesian information criterion. *Journal of Machine Learning Research*, 14(Mar):867–897, 2013.

58. T. V. Wiecki, I. Sofer, and M. J. Frank. Hddm: Hierarchical bayesian estimation of the drift-diffusion model in python. *Frontiers in neuroinformatics*, 7:14, 2013.

59. E. M. Wieschen, A. Voss, and S. Radev. Jumping to conclusion? a lévy flight model of decision making. *The Quantitative Methods for Psychology*, 16(2):120–132, 2020.

60. M. M. Yartsev, T. D. Hanks, A. M. Yoon, and C. D. Brody. Causal contribution and dynamical encoding in the striatum during evidence accumulation. *Elife*, 7:e34929, 2018.