DRM Documentation

Release 0.1

Michele D'Asaro, Umut Guclu, Marcel van Gerven

April 02, 2017

CHAPTER

ONE

INTRODUCTION

The Dynamic Representational Modeling (DRM) framework is used for end-to-end training of neural systems. The goal is to model human brains by constructing artificial brains that are 'as close as possible'. The way to realize this is by being explicit about how neuronal populations interact with sensory input, with each other and with motor output. Furthermore, we need to make explicit how *measurements* of neural activity are related to neuronal population activity.

There are two ways in which we can link physical to artificial brains: 1) Construct artificial brain and condition it on neural observations. The assumption is that population responses emerge in the artificial brain that are indicative of neural processing in the physical brain. 2) Construct artificial brain and have it perform the same cognitive task as the physical brain. Then compare population responses.

For now we focus on the first approach.

1.1 General approach

DRM assumes that all components in an artifical brain are neural networks. That is, we have populations, connections, and readout mechanisms that are all specified in terms of neural networks. These networks can implement very specific mechanisms such as conduction delays, neural response functions, etc.

Our goal at each modeling state is to be as explicit as possible about the underlying neural processes. That is, how does our artificial brain relate to neuronal time constants, connectivity structure, etc.

1.2 Usage

The goal of DRM is to provide a modeling framework where the artificial brains are trained end-to-end to provide the most explicit model of neural information processing possible for a task at hand.

Using DRM we can: 1) make explicit what kind of information is being processed by certain neuronal populations 2) compare different models (e.g. assumptions on connectivity structure, etc) through model comparison.

1.3 Artificial brains

An artificial brain (DRMNet) is a neural network consisting of sensory inputs and neuronal populations. Neuronal populations are connected to motor outputs and/or readouts (measurements). Populations, readouts and connections are all inherited from DRMNode.

The artificial brain is trained end-to-end, either on behavioural output or neural output. Training is facilitated by a DRM object which just wraps the standard neural network training procedure. Below we list components of a DRMNet.

1.4 Stimulus

A stimulus is a sensory input which can be connected to a subset of the neuronal populations. We can have multiple sensory inputs that are each connected to different subsets.

1.5 Populations

A DRMNet consists of a set of neuronal populations (DRMPopulation). Populations can have physical locations, measurement units, etc. Populations are linked to sensory input, each other and to the readouts via connections (DRMConnection).

1.6 Connections

Stimuli, neuronal populations and readouts are connected to each other via connections. These connections implement conduction delays, HRFs, etc. Below we list a default connection.

1.6.1 Tapped delay line

Stimuli are connected to populations and populations are connected to each other. A basic way to realize such connections is using *tapped delay lines*. These connections implement the notion that physical information transmission (via axons) takes time. Given a sampling resolution of r ms, a tapped delay line can delay the sample by n samples to instantiate conduction delay.

Note that this delay is necessary in order to make the artificial brain acyclic. I.e., if we run backpropagation then a delay of at least one sample ensures that we can unroll the neural network over time. A tapped delay line without any delay is an identity mapping.

Note further that this way of implementing an RNN gives a model in which updating is asynchronous in the sense that the same information may be processed by different populations at different points in time, like in biological neural networks.

1.7 Readout

A readout receives the output of all neuronal populations. It is itself responsible for how to handle these outputs. A readout can be either behavioural (motor) output or a neural measurement. The readout mechanism itself integrates population responses and translates this into something which can be passed onto a loss function. An important objective is to separate readout properties from neural information processing. We can have multiple readouts that yield (a sum of) multiple loss functions. We list some examples below.

1.7.1 BOLD readout

A BOLD readout could be implemented by linking each population to that subset of voxels that represent that population (e.g. V1). Suppose we have two populations (say V1 and V2). Then the readout will use the V1 population response to predict V1 voxels (or their average signal) and V2 population response to predict V2 voxels (or their average signal). The hemodynamic delay can be implemented by internally using parameterized functions (e.g. double gammas) or memory units to learn e.g. the HRF more flexibly.

1.7.2 M/EEG readout

In this case, we listen to all populations and use a lead field matrix to represent sensor output as a linear combination of the population responses

1.7.3 Single population recordings

In this case the readout mechanism ignores all populations except the one we record from.

1.7.4 Motor outputs

Motor outputs (button presses, eye movements) can be modelled using separate readouts that reflect behavioural ouput.

1.8 Confounds

Confounds such as breathing, heart beat, etc. can affect the readouts. We model these as direct additional inputs to the readout mechanism.

1.9 Connectivity

Connectivity between stimuli and populations and among populations are handled via a sparse vector and sparse matrix. None elements indicate absence of a connection. The other elements contain DRMConnection objects.

1.10 Data

Data is handled via a DRMIterator object. This object is responsible for producing sensory input, measurements and confounds at the sampling rate *r*. Note that each at time step, we may

have missing data. That is, the stimulus

may be absent, confounds may not be measured, and responses may not be measured (e.g. slow sampling of the hemodynamic response). On the input side, we use zeros to indicate absence of input (this may not always be valid). On the output side, this is handled by just ignoring outputs when computing the loss.

1.11 Training

Training takes place using truncated backpropagation on the (partially observed) data

CHAPTER

TWO

DRM PACKAGE

class DRM.base.**DRM**(*drm_net*)

Bases: object

wrapper object that trains and analyses the model at hand

estimate(*data_iter*, *val_iter=None*, *n_epochs=1*, *cutoff=None*)

Estimation via truncated backprop

Parameters

- **data_iter** – iterator which generates sensations/responses at some specified resolution
- **val_iter** – optional iterator which generates sensations/responses at some specified resolution used for validation
- **n_epochs** – number of training epochs
- **cutoff** – cutoff for truncated backpropagation

Returns train loss and validation loss

forward(*data_iter*)

Forward propagation

Parameters **data_iter** –

Returns generated response and population activity

class DRM.base.**DRMLoss**

Bases: torch.nn.modules.module.Module

MSE loss which ignores missing data

forward(*prediction*, *target*)

Computes loss on a prediction and a target

Computes MSE loss but ignores those terms where the target is equal to nan, indicating missing data.

Parameters

- **prediction** (*Variable*) – Prediction of output
- **target** (*Variable*) – Target output

Returns MSE loss

Return type Variable

class DRM.base.**DRMNet**(*populations*, *ws*, *Wp*, *readout*)

Bases: torch.nn.modules.container.Sequential

detach_()
Detach gradients for truncation

forward(*x*)
Forward propagation

Parameters **x** – sensory input at this point in time (zeros for no input); numpy array

Returns predicted output measurements

reset()
Reset states of model components

class DRM.base.**DRMNode**(*n_in=1*, *n_out=1*)
Bases: torch.nn.modules.module.Module

Base class for populations, readouts and connections

detach_()
Detach gradients for truncation

forward(*x*)
Forward pass for this node

Parameters **x** – input data

Returns output data

reset()
The function that is called when resetting internal state

class DRM.connection.**DRMConnection**(*n_in=1*, *n_out=1*, *delay=1*)
Bases: *DRM.base.DRMNode*

detach_()
Detach gradients for truncation

forward(*x*)
Forward propagation

Parameters **x** – input to connection

Returns connection output

reset()
Reset state

class DRM.iterators.**DRMIterator**(*resolution*, *stimulus*, *stim_time*, *response=None*, *resp_time=None*, *batch_size=None*, *n_batches=None*)
Bases: object

__iter__()
Initializes data generator. Should be invoked at the start of each epoch

Returns self

is_final()
Flags if final iteration is reached

Returns boolean if final batch is reached

next()
Produces next data item

Returns dictionary containing the stimulus and the response as torch variables

class DRM.population.**DRMPopulation**(*n_in=1*, *n_out=1*, *delay=1*)

Bases: *DRM.base.DRMNode*

forward(*x*)

Forward propagation

Parameters **x** (*list of afferent population outputs*) – population input

Returns population output

class DRM.readout.**DRMReadout**(*n_in=1*, *n_out=1*)

Bases: *DRM.base.DRMNode*

forward(*x*)

Forward propagation

Parameters **x** (*list of afferent population outputs*) – readout input

Returns predicted measurements

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

d

Symbols

D

E

F

I

N

R