# Emergent property inference python package design

Sean Bittner

December 19, 2019

## 1 Introduction

Throughout the emergent property inference (EPI) project [1], we hosted all development code on the cunningham-lab github repo: dsn (EPI was formerly DSN), which also relies on the cunningham-lab repo tf_util, which has customized tensorflow code for normalizing flows, and Hessians of deep generative models. Here, we design a new python software package for EPI that is more accessible for our user-base of theoretical neuroscientists. While functioning as a software development library for our users with high programming skill, this software shall also run seemlessly on NCAP through a graphical user interface.

## 2 Requirements

EPI is designed to be a tool for theoretical neuroscientists to automate inference in complex models of neural circuits. For more details, see [1].

The epi python package

- shall run emergent property inference given
  - a model specification
    - parameter names and continuous ranges
    - emergent property statistics as a differentiable function of parameters
  - an emergent property value
  - (optional) a specific normalizing flow architecture
  - (optional) and specific optimization hyperparameters.
- shall return the distribution sampler upon convergence.
- shall provide log probabilities, and Hessians throughout the support of the EPI distribution.
- shall visualize optimization diagnostics.
- shall visualize the EPI distribution and its properties.
- shall run on NCAP.

## 3 Dependency selection

The EPI method relies on automatic differentiation, so we should use one of the three established open source python packages: PyTorch, Tensorflow, or Jax. PyTorch has risen in popularity with respect to Tensorflow, possibly due to PyTorch's early implementation and release of a simplifying/intuitive eager execution mode. The Tensorflow 2.0 release puts these two packages on roughly equal footing. Jax, a python library for autodiff and efficient optimization with standard python/numpy code, is gaining popularity.

### 3.1 Normalizing flows support

In the end, Tensorflow is the only option with an established normalizing flows API – one that includes real NVP, MAF, spline flows – making it the most supportive for our purposes: tf.bijectors. PyTorch has a TransformedDistributions class, which doesn't yet support the popular normalizing flow architectures. PyTorch and Jax have a few options on github, but their level of support is contingent on independent developers (nf-jax, kamenbliznashki/normalizing_flows, karpathy/pytorch-normalizing-flows). If we want to go with these repos, we'll need to have faith in their current implementation, and future maintenance.

### 3.2 Model implementation requirement

The EPI package is designed to receive a differentiable function producing the emergent property statistics from the model parameters. In the event that our user-base is largely unfamiliar with Tensorflow/PyTorch, Jax is an interesting alternative, in that we could lower the barriers-to-entry for our software package. Specifically, users could submit native python functions through which Jax can take gradients. Otherwise, we must require our users to write Tensorflow or PyTorch functions, or find a reliable way of taking Tensorflow or PyTorch gradients through native python.

To gain an idea of the programming experience of our userbase, I asked 20 Theory Center members working on theoretical modeling projects whether they a.) used python and b.) if so, knew PyTorch or Tensorflow. The results suggest a 70% level of familiarity with at least one of PyTorch or Tensorflow.

Table 1: Theoretical neuroscientist programming experience

| no python | python (no autograd) | python autograd |
|---|---|---|
| 3 | 3 | 14 |

### 3.3 Tensorflow implementation rationale

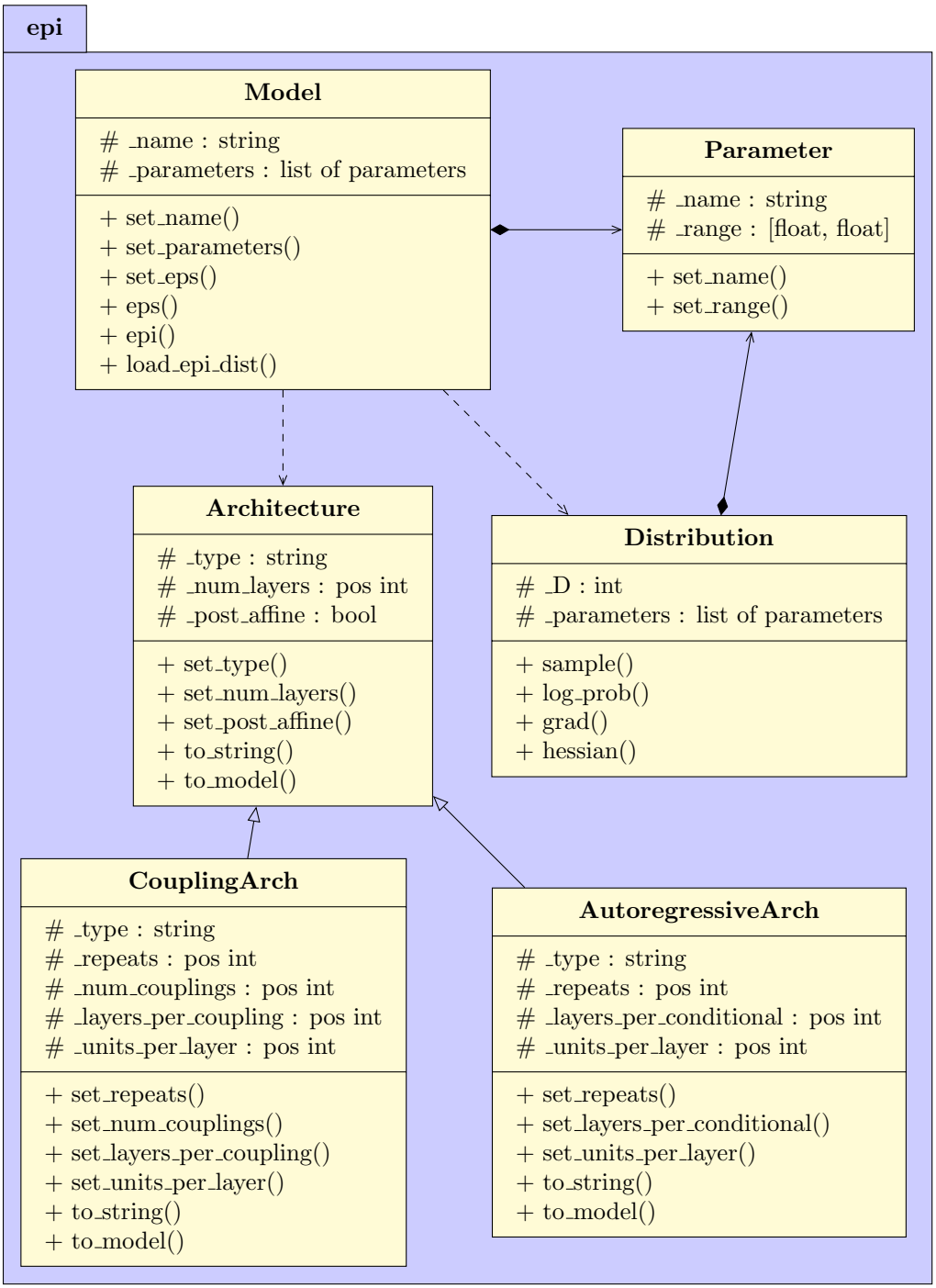
The lack of a well-supported normalizing flows library for PyTorch and Jax makes these option unattractive. Given the relatively high python autograd proficiency of our user-base, I conclude that it makes sense to design the package to accept a Tensorflow function defining the emergent property statistics.

# 4 EPI package design

EPI is a method designed to run for a broad space of theoretical models and behaviors. EPI's implementation changes according to the details of each model. It makes sense to have our users construct a model object (defined by the Model class), which has a set of defined parameters and an eps() (emergent property statistics) function. With these class members, we can then run the method epi(), which will return an optimized distribution object (Distribution class).

We create an Architecture class as well, to shield the user from Tensorflow, except through the supplied eps() function. As a result, the user can construct simplified python objects (constructors of these classes) according to their use case, run the EPI method, and have a simplified python distribution object returned. The normalizing flows library and autograd functionality of Tensorflow are used by the package, yet not exposed to the user.

### 4.1 EPI class diagram




### 4.2 Key functions

`Model.eps(): parameters (list of tf.Tensors) -> T_x (tf.Tensor)`

Tensorflow computation of the the emergent property statistics (eps) of the model from an arbitrary length list of parameters. The user can provide this function during initialization or through the Model.set_eps() method.

```
Model.epi(): mu (np.array), arch (Architecture), hps, ... ->
             epi_dist (Distribution)
```

Run EPI on the model (which has set parameters and an Tensorflow eps() function) for emergent property value "mu" with optional parameters for architecture "arch" and hyperparameters "hps".

```
Architecture.to_model(): self -> tf.Model
```

Convert Architecture class to a realization of a Tensorflow normalizing flow distribution with the given architecture.

# 5 Testing

## 5.1 Unit testing

We will unit test the API functionality with pytest and codecov. Package stability on multiple OS's and builds will be assessed with Travis CI, running these unit tests and coverage diagnostics.

## 5.2 Validation

Algorithm validation can happen be done on exponential families with known solutions, and simple systems with derivable contours like the oscillating 2D linear dynamical system.

# 6 Documentation

We will use ReadTheDocs for automatic documentation generation and hosting (no gitpage).

# 7 Style

We will use black as our automatic style formatter.

# References

[1] Sean R Bittner, Agostina Palmigiano, Alex T Piet, Chunyu A Duan, Carlos D Brody, Kenneth D Miller, and John P Cunningham. Interrogating theoretical models of neural computation with deep inference. *bioRxiv*, page 837567, 2019.