deep-ei

Oct 28, 2020

Introduction:

CHAPTER 1

What's here?

This code accompanies the paper Examining the Causal Structures of Artificial Neural Networks Using Information Theory.

1.1 Introduction

1.1.1 Installation

The simplest way to install the `deep_ei` module is with:

```
pip install deep-ei
```

Becaues `pytorch` can be fragile, it is recommended that you install and test `pytorch` before installing `deep-ei` (such as with `conda install pytorch -c pytorch`). To install `deep-ei` directly from the GitHub repository:

```
git clone https://github.com/EI-research-group/deep-ei.git
cd deep-ei
pip install .
```

Basic tests can be executed with:

```
python setup.py test
```

Note that we have also provided an anaconda environment file. You can use it to create a new environment with `deep-ei` and all its dependencies:

```
conda env create --file environment.yml
```

1.1.2 Examples

Here are some basic examles:

```
import torch
import torch.nn as nn

from deep_ei import topology_of, ei, ei_parts, sensitivity, ei_parts_matrix

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
dtype = torch.float32
torch.set_default_dtype(dtype)

network = nn.Linear(5, 5, bias=False).to(device)
top = topology_of(network, input=torch.zeros((1, 5)).to(device))

EI = ei(network, top,
                samples=int(1e5),
                batch_size=100,
                in_range=(0, 1),
                in_bins=8,
                out_range=(0, 1),
                out_bins=8,
                activation=nn.Sigmoid(),
                device=device)
```

This will compute the EI of the `5 -> 5` dense layer `network` using a sigmoid activation and 100000 samples.

The function `topology_of` creates a `networkx` graph representing the connectivity of the network. `ei` can infer argument values using this graph, such as the ranges of the inputs and outputs of the layer and its activation function:

```
network = nn.Sequential(
    nn.Linear(20, 10, bias=False),
    nn.Sigmoid(),
    nn.Linear(10, 5, bias=False),
    nn.Tanh()
)
top = topology_of(network, input=torch.zeros((1, 20)).to(device))

layer1, _, layer2, _ = network

EI_layer1 = ei(layer1, top,
                    samples=int(1e5),
                    batch_size=100,
                    in_range=(0, 1),
                    in_bins=8,
                    out_bins=8,
                    device=device)

EI_layer2 = ei(layer2, top,
                    samples=int(1e5),
                    batch_size=100,
                    in_bins=8,
                    out_bins=8,
                    device=device)
```

Which will use an activation of `nn.Sigmoid` and an `out_range` of `(0, 1)` for the first layer and an activation of `nn.Tanh` and an `out_range` of `(-1, 1)` for the second layer. Note that we have to specify an `in_range` for the first layer.

EI_parts can be computed similarly:

```
import torch
import torch.nn as nn

from deep_ei import topology_of, ei, ei_parts

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
dtype = torch.float32
torch.set_default_dtype(dtype)

network = nn.Linear(5, 5, bias=False).to(device)
top = topology_of(network, input=torch.zeros((1, 5)).to(device))

EI = ei_parts(network, top,
                    samples=int(1e5),
                    batch_size=100,
                    in_range=(0, 1),
                    in_bins=8,
                    out_range=(0, 1),
                    out_bins=8,
                    activation=nn.Sigmoid(),
                    device=device)
```

With `ei_parts`, you can specify a `threshold` instead of setting a manual number of `samples` (indeed this is the default behavior of `ei_parts`, with default threshold of 0.05). The function will increase the number of samples it uses until EI_parts levels off (characterized by whether EI_parts will change by less than `threshold` of its current value even if we doubled the number of samples):

```
network = nn.Linear(10, 10, bias=False).to(device)
top = topology_of(network, input=torch.zeros((1, 10)).to(device))

EI = ei_parts(network, top,
                    threshold=0.05,
                    batch_size=100,
                    in_range=(0, 1),
                    in_bins=64,
                    out_range=(0, 1),
                    out_bins=64,
                    activation=nn.Sigmoid(),
                    device=device)
```

You can also measure the sensitivity of a layer like so:

```
network = nn.Linear(10, 10, bias=False).to(device)
top = topology_of(network, input=torch.zeros((1, 10)).to(device))

sensitivity = sensitivity(network, top,
                            samples=1000,
                            batch_size=100,
                            in_range=(0, 1),
                            in_bins=64,
                            out_range=(0, 1),
                            out_bins=64,
                            activation=nn.Sigmoid(),
                            device=device)
```

If you want to compute the EI of each edge in a layer (giving you each term that is summed to get EI_parts), use the `ei_parts_matrix` function:

```
network = nn.Linear(20, 10, bias=False).to(device)
top = topology_of(network, input=torch.zeros((1, 20)).to(device))

EI = ei_parts_matrix(network, top,
                        samles=50000,
                        batch_size=100,
                        in_range=(0, 1),
                        in_bins=64,
                        out_range=(0, 1),
                        out_bins=64,
                        activation=nn.Sigmoid(),
                        device=device)
```

Which will return a `20 x 10` matrix where the rows correspond with in-neurons and the columns correspond with out-neurons.

1.1.3 Ideas for future experiments

We'd love for people to use and expand on this code to make new discoveries. Here are some questions we haven't looked into yet:

- How does dropout effect the EI of a layer? In otherwise identical networks, does dropout increase or decrease the EI of the network layers?
- What can EI tell us about generalization? Does EI evolve in the causal plane in different ways when a network is memorizing a dataset vs generalizing? To test this, train networks on some dataset as you would normally, but then randomize the labels in the training dataset and train new networks. This label randomization will force the network to memorize the dataset.
- On harder tasks, where deep networks are required (in MNIST and Iris, which we studied, it is unnecessary that networks be deep for them to achieve good acuracy), do the hidden layers differentiate in the causal plane?
- Can EI be measured in recurrent networks? How would this work?

1.1.4 Contributing & Questions

We'd welcome feedback and contributions! Feel free to email me at `eric.michaud99@gmail.com` if you have questions about the code.

1.2 `deep-ei` Module

`deep_ei.`**`topology_of`**(*model*, *input*)

Get a graph representing the connectivity of `model`.

Because PyTorch uses a dynamic computation graph, the number of activations that a given module will return is not intrinsic to the definition of the module, but can depend on the shape of its input. We therefore need to pass data through the network to determine its connectivity.

This function passes `input` into `model` and gets the shapes of the tensor inputs and outputs of each child module in model, provided that they are instances of `VALID_MODULES`. It also finds the modules run before and after each child module, provided they are in `VALID_ACTIVATIONS`.

Parameters

- **model** (*nn.Module*) – feedforward neural network
- **input** (*torch.tensor*) – a valid input to the network

Returns

representing connectivity of `model`.

Each node of the returned graph contains a dictionary:

```
{
    "input": {"activation": activation module, "shape": tuple},
    "output": {"activation": activation module, "shape": tuple}
}
```

Return type nx.DiGraph

Examples

```
>>> network = nn.Sequential(nn.Linear(42, 20),
                            nn.Sigmoid(),
                            nn.Linear(20, 10))
>>> top = topology_of(network, input=torch.zeros((1, 42)))
>>> layer1, _, layer2 = network
>>> top.nodes[layer1]['output']['activation']
nn.Sigmoid instance
>>> top.nodes[layer1]['input']['shape']
(1, 42)
```

`deep_ei.`**`ei`**(*layer, topology, threshold=0.05, samples=None, batch_size=20, in_layer=None, in_range=None, in_bins=64, out_range=None, out_bins=64, activation=None, device='cpu'*)

Computes the vector effective information of neural network layer `layer`. By a "layer", we mean the function defined by the composition of some specified sequence of layers in the network:

$$EI(L_1 \to L_2) = I(L_1; L_2) \mid do(L_1 = H^{\max})$$

Parameters

- **layer** (*nn.Module*) – a module in `topology`
- **topology** (*nx.DiGraph*) – topology object returned from `topology_of` function
- **threshold** (*float*) – used to dynamically determine how many samples to use.
- **samples** (*int*) – if specified (defaults to None), function will manually use this many samples, which may or may not give good convergence.
- **batch_size** (*int*) – the number of samples to run `layer` on simultaneously
- **in_layer** (*nn.Module*) – the module in `topology` which begins our 'layer'. By default is the same as *layer*.
- **in_range** (*tuple*) – (lower_bound, upper_bound), inclusive. By default determined from `topology`
- **in_bins** (*int*) – the number of bins to discretize in_range into for MI calculation
- **out_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **out_bins** (*int*) – the number of bins to discretize out_range into for MI calculation

- **`activation`** (*`function`*) – the output activation of `layer`, by defualt determined from `topology`
- **`device`** – 'cpu' or 'cuda' or `torch.device` instance

Returns an estimate of the vector-EI of layer `layer`

Return type float

`deep_ei.`**`ei_parts`**(*layer*, *topology*, *threshold=0.05*, *samples=None*, *extrapolate=False*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Computes *EI_parts* of neural network layer `layer`. By a "layer", really mean the edges connecting two layers of neurons in the network. The EI_parts EI of these connections is defined:

$$EI_{parts}(L_1 \rightarrow L_2) = \sum_{(A \in L_1, B \in L_2)} I(t_A, t_B) \mid do(L_1 = H^{\max})$$

Parameters

- **`layer`** (*`nn.Module`*) – a module in `topology`
- **`topology`** (*`nx.DiGraph`*) – topology object returned from `topology_of` function
- **`threshold`** (*`float`*) – used to dynamically determine how many samples to use.
- **`samples`** (*`int`*) – if specified (defaults to None), function will manually use this many samples, which may or may not give good convergence.
- **`extrapolate`** (*`bool`*) – if True, then evaluate EI at several points and then fit a curve to determine asymptotic value.
- **`batch_size`** (*`int`*) – the number of samples to run `layer` on simultaneously
- **`in_layer`** (*`nn.Module`*) – the module in `topology` which begins our 'layer'. By default is the same as *layer*.
- **`in_range`** (*`tuple`*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **`in_bins`** (*`int`*) – the number of bins to discretize in_range into for MI calculation
- **`out_range`** (*`tuple`*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **`out_bins`** (*`int`*) – the number of bins to discretize out_range into for MI calculation
- **`activation`** (*`function`*) – the output activation of `layer`, by defualt determined from `topology`
- **`device`** – 'cpu' or 'cuda' or `torch.device` instance

Returns an estimate of the EI of layer `layer`

Return type float

`deep_ei.`**`ei_parts_matrix`**(*layer*, *topology*, *samples=None*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Computes the EI of all `A -> B` connections of neural network layer `layer`.

The EI of the connection `A -> B` is defined as:

$$EI(A \rightarrow B) = I(t_A, t_B) | do(L_1 = H^{\max})$$

where neuron A is in layer `L_1`. This is the mutual information between A's activation and B's activation when all the other neurons in `L_1` are firing randomly (independently and uniformly in their activation output range).

Parameters

- **layer** (*nn.Module*) – a module in *topology*
- **topology** (*nx.DiGraph*) – topology object returned from `topology_of` function
- **threshold** (*float*) – used to dynamically determine how many samples to use.
- **samples** (*int*) – if specified (defaults to None), function will manually use this many samples, which may or may not give good convergence.
- **extrapolate** (*bool*) – if True, then evaluate EI at several points and then fit a curve to determine asymptotic value.
- **batch_size** (*int*) – the number of samples to run `layer` on simultaneously
- **in_layer** (*nn.Module*) – the module in `topology` which begins our 'layer'. By default is the same as *layer*.
- **in_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **in_bins** (*int*) – the number of bins to discretize in_range into for MI calculation
- **out_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **out_bins** (*int*) – the number of bins to discretize out_range into for MI calculation
- **activation** (*function*) – the output activation of `layer`, by defualt determined from `topology`
- **device** – 'cpu' or 'cuda' or `torch.device` instance

Returns A matrix whose[A][B]th element is the EI from `A -> B`

Return type np.array

`deep_ei.`**`sensitivity`**(*layer*, *topology*, *samples=500*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Computes the sensitivity of neural network layer *layer*.

Note that this does not currently support dynamic ranging or binning. There is a good reason for this: because the inputs we run through the network in the sensitivity calculation are very different from the noise run though in the EI calculation, each output neuron's range may be different, and we would be evaluating the sensitivity an EI using a different binning. The dynamic ranging and binning supported by the EI function should be used with great caution.

$$Sensitivity(L_1 \to L_2) = \sum_{(A \in L_1, B \in L_2)} I(t_A, t_B) \mid do(A = H^{\max})$$

Parameters

- **layer** (*nn.Module*) – a module in `topology`
- **topology** (*nx.DiGraph*) – topology object returned from `topology_of` function
- **samples** (*int*) – the number of noise samples to run through `layer`
- **batch_size** (*int*) – the number of samples to run `layer` on simultaneously
- **in_layer** (*nn.Module*) – the module in `topology` which begins our 'layer'. By default is the same as `layer`.

- **in_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **in_bins** (*int*) – the number of bins to discretize in_range into for MI calculation
- **out_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **out_bins** (*int*) – the number of bins to discretize out_range into for MI calculation
- **activation** (*function*) – the output activation of `layer`, by defualt determined from `topology`
- **device** – 'cpu' or 'cuda' or `torch.device` instance

Returns an estimate of the sensitivity of layer `layer`

Return type float

`deep_ei.`**`sensitivity_matrix`**(*layer*, *topology*, *samples=500*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Computes the sensitivitites of each A -> B connection of neural network layer *layer*.

Note that this does not currently support dynamic ranging or binning. There is a good reason for this: because the inputs we run through the network in the sensitivity calculation are very different from the noise run though in the EI calculation, each output neuron's range may be different, and we would be evaluating the sensitivity and EI using a different binning. The dynamic ranging and binning supported by the EI function should be used with great caution.

$$Sensitivity(A \to B) = I(t_A, t_B) \mid do(A = H^{\max})$$

where neuron A is in layer `L_1`. This is the mutual information between A's activation and B's activation when A is firing randomly (uniformly) and all the other neurons in `L_1` are outputing 0 (not firing).

Parameters

- **layer** (*nn.Module*) – a module in `topology`
- **topology** (*nx.DiGraph*) – topology object returned from `topology_of` function
- **samples** (*int*) – the number of noise samples run through `layer`
- **batch_size** (*int*) – the number of samples to run `layer` on simultaneously
- **in_layer** (*nn.Module*) – the module in `topology` which begins our 'layer'. By default is the same as *layer*.
- **in_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **in_bins** (*int*) – the number of bins to discretize in_range into for MI calculation
- **out_range** (*tuple*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **out_bins** (*int*) – the number of bins to discretize out_range into for MI calculation
- **activation** (*function*) – the output activation of `layer`, by defualt determined from `topology`
- **device** – 'cpu' or 'cuda' or `torch.device` instance

Returns A matrix whose[A][B]th element is the sensitivity from `A -> B`

Return type np.array

`deep_ei.`**`vector_and_pairwise_ei`**(*layer*, *topology*, *samples=None*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Returns (vector_ei, pairwise_ei), both computed with the same *samples*.

`deep_ei.`**`eis_between_groups`**(*layer*, *topology*, *groups*, *samples=None*, *batch_size=20*, *in_layer=None*, *in_range=None*, *in_bins=64*, *out_range=None*, *out_bins=64*, *activation=None*, *device='cpu'*)

Computes the EI between subsets of nodes specified with *groups*.

Parameters

- **`layer`** (*`nn.Module`*) – a module in `topology`
- **`topology`** (*`nx.DiGraph`*) – topology object returned from `topology_of` function
- **`samples`** (*`int`*) – use this many samples, which may or may not give good convergence.
- **`groups`** (*`list`*) – list of tuples of tuples. For instance: [((1, 2, 3), (1,))] will compute vector-EI between neurons 1, 2, and three in the in-layer (as a group) and neuron 1 in the out layer.
- **`batch_size`** (*`int`*) – the number of samples to run `layer` on simultaneously
- **`in_layer`** (*`nn.Module`*) – the module in `topology` which begins our 'layer'. By default is the same as *layer*.
- **`in_range`** (*`tuple`*) – (lower_bound, upper_bound), inclusive. By default determined from `topology`
- **`in_bins`** (*`int`*) – the number of bins to discretize in_range into for MI calculation
- **`out_range`** (*`tuple`*) – (lower_bound, upper_bound), inclusive, by default determined from `topology`
- **`out_bins`** (*`int`*) – the number of bins to discretize out_range into for MI calculation
- **`activation`** (*`function`*) – the output activation of `layer`, by defualt determined from `topology`
- **`device`** – 'cpu' or 'cuda' or `torch.device` instance

Returns an estimate of the vector-EI of layer `layer`

Return type float

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index