Deep Causal Graphs for Causal Inference, Black-Box Explainability and Fairness

Supplementary Material

Álvaro Parafita Jordi Vitrià

Counterfactuals and Importance Sampling

With DCGs, in the presence of latent confounders or incomplete evidence, one must use importance sampling to properly deal with counterfactual distributions.

Consider an SCM $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{U}, \mathcal{P}(\mathcal{E}), \mathcal{P}(\mathcal{U}), \mathcal{F})$, an intervention set $\emptyset \subsetneq X \subsetneq \mathcal{V}$ with an atomic intervention x, an evidence set $\emptyset \subsetneq E \subseteq \mathcal{V}$ with sample e, and $\mathcal{V}_x$ the counterfactual variables $\mathcal{V}$ under intervention $do(X = x)$. Our object of study is $p(\mathcal{V}_x \mid e)$.

Let us denote $Z := \mathcal{U} \cup \{E_V \in \mathcal{E} \mid V \notin E\}$, the set of latent confounders (if any) and exogenous noise signals corresponding to any variables in $\mathcal{V}$ missing from our evidence set E, and $\mathcal{E}' := \mathcal{E} \setminus Z$, the set of exogenous noise signals not contained in Z. Note that $Z \cup \mathcal{E}' = \mathcal{U} \cup \mathcal{E}$, all latent variables. Then:

$$\begin{aligned} p(V_x \mid e) &= \mathbb{E}_{(Z,\mathcal{E}'|e)}\left[p(\mathcal{V}_x \mid e, Z, \mathcal{E}')\right] = \\ &= \mathbb{E}_{(Z,\mathcal{E}'|e)}\left[p(\mathcal{V}_x \mid Z, \mathcal{E}')\right] = \\ &= \mathbb{E}_{(Z|e)}\left[\mathbb{E}_{(\mathcal{E}'|e,Z)}\left[p(\mathcal{V}_x \mid Z, \mathcal{E}')\right]\right] = \\ &= \mathbb{E}_Z\left[\mathbb{E}_{(\mathcal{E}'|e,Z)}\left[p(\mathcal{V}_x \mid Z, \mathcal{E}')\right] \cdot \frac{p(e \mid Z)}{p(e)}\right] \end{aligned} \tag{1}$$

where $p(\mathcal{V}_x \mid e, Z, \mathcal{E}') = p(\mathcal{V}_x \mid Z, \mathcal{E}')$ since $(Z, \mathcal{E}')$ block any path between factual variables E and counterfactual variables $\mathcal{V}_x$. $p(\mathcal{V}_x \mid Z, \mathcal{E}')$ is modelled by a Dirac Delta distribution centered on $V_x(Z, \mathcal{E}')$, since $\mathcal{V}_x$ can be computed deterministically given Z and $\mathcal{E}'$ following the structural equations in our model $\mathcal{M}$.

We can estimate this expectation by Monte Carlo, taking N i.i.d. samples of unconditioned Z and M i.i.d. samples from $p(\mathcal{E}' \mid e, Z)$. We do have access to $p(\mathcal{E}' \mid e, Z)$ thanks to the *abduction* step in each DCU, since (e, Z) generates values for every variable in $\mathcal{V}$ by using $\mathcal{F}$. Therefore, $\log p(\mathcal{E}' \mid e, Z) = \sum_{E_k \in \mathcal{E}'} \log p(E_k \mid Pa_k, \mathcal{U}_{\{k,.\}})$, which is sampled node by node with the *abduct* operation. Finally:

$$\begin{aligned} p(\mathcal{V}_x \mid e) &= \mathbb{E}_Z\left[\mathbb{E}_{(\mathcal{E}'|e,Z)}\left[p(\mathcal{V}_x \mid Z, \mathcal{E}')\right] \cdot \frac{p(e \mid Z)}{p(e)}\right] \approx \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{M}\sum_{j=1}^{M} p(\mathcal{V}_x \mid z_i, \varepsilon'_{i,j})\right)\frac{p(e \mid z_i)}{p(e)} \approx \\ &\approx \sum_{i=1}^{N}\sum_{j=1}^{M} p(\mathcal{V}_x \mid z_i, \varepsilon'_{i,j})\frac{s(\log p(e \mid z))_i}{M} \end{aligned} \tag{2}$$

with $z_i = (u_i, \varepsilon'_i) \sim \mathcal{P}(Z)$ (sampling from its prior), $\varepsilon'_{i,j} \sim \mathcal{P}(\mathcal{E}' \mid e, z_i)$ (abduction), and $s(.)$ being the softmax operation on $(\log p(e \mid z_i))_{i=1}^N$. This last step comes from the realization that $p(e) \approx \frac{1}{N}\sum_{i=1}^{N} p(e \mid z_i)$ and, therefore, $\frac{p(e|z_i)}{p(e)} \approx \frac{p(e|z_i)}{\frac{1}{N}\sum_{i=1}^{N} p(e|z_i)} = \frac{\exp \log p(e|z_i)}{\frac{1}{N}\sum_{i=1}^{N} \exp \log p(e|z_i)} = N \cdot s(\log p(e \mid z))_i$. The N term is cancelled by the expectation-average denominator for Z, rendering the formula described by equation 2.

In practice, there are two operations we can perform with this approximation. If we are to compute the expectation of a function on our counterfactual variables, $\mathbb{E}\left[f(\mathcal{V}_x) \mid e\right]$:

1. Generate $i = 1..N$ $z_i \sim \mathcal{P}(Z)$ i.i.d. samples and $j = 1..M$ abducted samples $\varepsilon'_{i,j}$ for each z_i.
2. Compute the corresponding counterfactual variable values $v_{i,j} := \mathcal{V}_x(e, z_i, \varepsilon'_{i,j})$ and their images by function f, $y_{i,j} := f(v_{i,j})$.
3. Aggregate images $(y_{i,j})_{i,j}$ using the approximated weights $w_{i,j} := \frac{s(\log p(e|z))_i}{M}$.

If, instead, we want to sample from the counterfactual distribution, we can do so as a two-step process: we obtain samples $(v_{i,j})_{i,j}$ and weights $w_{i,j}$ following the previous procedure, and then generate a weighted Bootstrap sample (with replacement) from this subsample. Both methods allow us to inspect the counterfactual distribution, be it for estimating an expectation or to analyze its shape.

Code example

In order to illustrate the ease of use of DCGs for the end user, we include a code example to define, train and use a graph for counterfactual estimation. In particular, we assume a dataset with 3 variables, X (continuous), T (binary, our treatment) and Y (continuous, our outcome), with structure $X \leftrightarrow T, X \rightarrow Y; T \rightarrow Y$ (there is a latent confounder between X and T). Our objective is to evaluate the following counterfactual: $\mathbb{E}_{\mathcal{V}}\left[\mathbb{E}\left[Y_{T=1} \mid \mathcal{V}\right] - \mathbb{E}\left[Y_{T=0} \mid \mathcal{V}\right]\right]$.

Listing 1. Create graph, train and estimate counterfactual queries.

```
from dcg.graph import CausalGraph
from dcg.distributional.discrete \
    import Bernoulli
from dcg.flow import NCF
from dcg.latents import LatentNormal
from dcg.training import train

# Create the graph
graph = CausalGraph.from_definition(
    CausalGraph.parse_definition('''
        U latent 1

        X continous 1 U
        T bernoulli 1 U
        Y continous 1 X T
        ''',
        latent=LatentNormal,
        bernoulli=Bernoulli,
        continous=NCF
    )
).cuda() # can be run in CPU too

# Train it with early stopping
trainV, valV, testV = ... # our data
train(graph, trainV, valV)

#Evaluate query
cf = graph.counterfactual # function alias
with torch.no_grad():
    # Call counterfactual with:
    # - evidence
    # - target variable (optional)
    # - interventions
    y1 = cf(testV, 'Y', dict(T=1))
    y0 = cf(testV, 'Y', dict(T=0))

(y1 - y0).mean() # result
```

Practical Considerations

To speed up and stabilize training, it is very important to perform a ***warm start*** of each node's distribution. For continuous nodes, we do this by learning normalization parameters based on the training data statistics. We perform normalization of a node's sample before computing its log-likelihood, and denormalize after sampling from that node's distribution. Note that the resulting log-likelihood also needs to take into account the normalizing transformation.

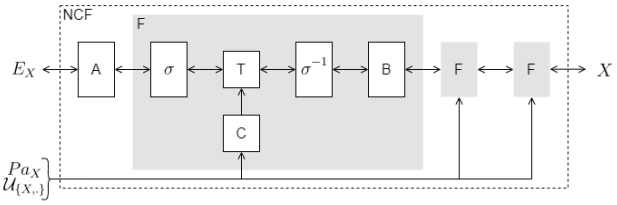

Figure 1. Normalizing Causal Flow (NCF) architecture. A - Affine layer; σ - Sigmoid layer; T - Spline Transformer; C - Flow Conditioner; σ^{-1} - Inverse Sigmoid layer; B - Batch Normalization; F - Spline Flow block.

Regarding latent confounders, we model them as non-learnable nodes that just output a noise signal (usually, a $\mathcal{N}(0, 1)$); however, some use cases could benefit from using specific priors (even learnable nodes) for $P(\mathcal{U})$. These samples from $\mathcal{U}$ are passed to their children as any other parent value and, by using the previously derived formula for the log-likelihood of a graph's sample, we can train the whole graph so that it learns the effect of this latent confounder. In that regard, we found that a 100 $\mathcal{U}$ samples per sample for the Monte Carlo estimator is effective enough to learn the desired dependency.

Another important aspect is that for any network, as its input dimensionality increases, the individual effect of any dimension decreases, even for the treatment. In order to avoid this issue, for the case of binary treatments, we can employ strategies such as bi-headed networks (TARNet or CFR, (Shalit et al., 2017)), XLearner (Künzel et al., 2019), or mirrored networks for each treatment (Alaa & Schaar, 2018). We follow the findings related to TARNet in that we employ **bi-headed networks** in our NCF conditioners and our discrete DCN's networks, when the treatment is binary.

Model Architecture and Training Hyperparameters

In this section, we discuss practical details about the implementation of our framework. As stated in the paper, DCGs are modelled node by node for each observable variable ($\mathcal{V}$), using either a Bernoulli or Categorical DCN (Parafita & Vitrià, 2019) for discrete variables, or a Normalizing Causal Flow (NCF) for continuous variables.

Bernoulli and Categorical DCNs follow the architecture proposed by the original authors, except that we use multi-head networks (following (Shalit et al., 2017), as discussed in the paper), with a head for each possible treatment. Each network consists of an initial Batch Normalization step (Ioffe & Szegedy, 2015) to normalize the input, a first block consisting of a Linear layer (100-dimension hidden layer), a Dropout layer (Srivastava et al., 2014) and a ReLU layer

(Glorot et al., 2011). Each head consists of another of these blocks (50-dimension hidden layer), followed by a last Linear layer with a specific activation function for each dimension, so as to adapt its domain to each DCN's parameter domain. In particular, for Bernoulli variables, we use a log-sigmoid activation, while the one for Categorical variables is a log-softmax activation layer. The use of logarithms in these functions results from the need for numerical stability.

In the case of NCFs, the possibilities are vast, since any Conditional Normalizing Flow works with our NCF framework. In our experiments, we have found that a 5-split quadratic Spline flow (Müller et al., 2019) works best with most univariate random variables. Figure 1 represents the NCF internals, which we describe now. We stack an initial Affine flow (to adjust the base distribution $\mathcal{E}_X \sim \mathcal{N}(0, 1)$ to an appropriate scale) with 3 consecutive blocks of Spline flows. Our implementation of Spline flows works in the interval $(0, 1)$, so we precede each block by a Sigmoid flow and end it by an inverse sigmoid flow, so as to go back and forth from the $(0, 1)$ interval to $\mathbb{R}$, and then an additional Batch Normalization flow, that takes its output distribution (from X's side) and scales it back to location 0 and scale 1. In between, we find our Spline flow, consisting of a conditioner network equivalent to the one in DCNs (but without Dropout layers), which outputs two sets of parameters: *splits* and *heights*. These are used by the flow's transformer in the following way: the *splits* divide the $(0, 1)$ interval in 5 (learnable) pieces, and each of these pieces has also a starting and ending *height*, which fit a linear curve for each split. This piece-wise linear curve is the density of the flow's distribution; this results in a quadratic spline for its CDF, which we use for sampling.

Regarding training setup, we use 10-split Cross Validation on a training set of 90% of the dataset. Each split is trained with the following hyperparameters: batch size 128, 100-patience Early Stopping (when the validation set's loss hasn't decreased for 100 epochs, training halts and the epoch with the best validation loss is returned) and an Adam optimizer (Kingma & Ba, 2015) with learning rate of 10^{-3} and weight decay of 10^{-2} (L_2 regularization on the network's weights). We found these hyperparameters to generally offer the best performance for many experiments. Once a model has been trained, the one with the best validation loss is selected and its evaluation metrics are computed on the complete training and test sets. These metric results are displayed in the main paper.

References

Alaa, A. and Schaar, M. Limits of estimating heterogeneous treatment effects: Guidelines for practical algorithm design. In *International Conference on Machine Learning*, pp. 129–138. PMLR, 2018.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings, 2011.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 448–456, Lille, France, 07–09 Jul 2015. PMLR.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, San Diego, CA, 2015.

Künzel, S. R., Sekhon, J. S., Bickel, P. J., and Yu, B. Metalearners for estimating heterogeneous treatment effects using machine learning. *Proceedings of the national academy of sciences*, 116(10):4156–4165, 2019.

Müller, T., McWilliams, B., Rousselle, F., Gross, M., and Novák, J. Neural importance sampling. *ACM Transactions on Graphics*, 38(5):145:1–145:19, 2019.

Parafita, Á. and Vitrià, J. Explaining visual models by causal attribution. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pp. 4167–4175. IEEE, 2019.

Shalit, U., Johansson, F. D., and Sontag, D. Estimating individual treatment effect: generalization bounds and algorithms. In *International Conference on Machine Learning*, pp. 3076–3085. PMLR, 2017.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.