Implicit MLE: Backpropagating Through Discrete Exponential Family Distributions

Mathias Niepert Pasquale Minervini Luca Franceschi

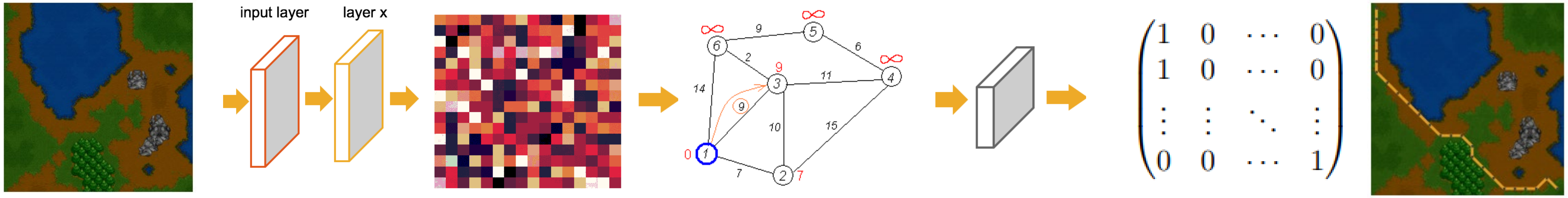
NEC

UCL

iit CSML ISTITUTO ITALIANO DI TECNOLOGIA COMPUTATIONAL STATISTICS AND MACHINE LEARNING

Motivating Example 1

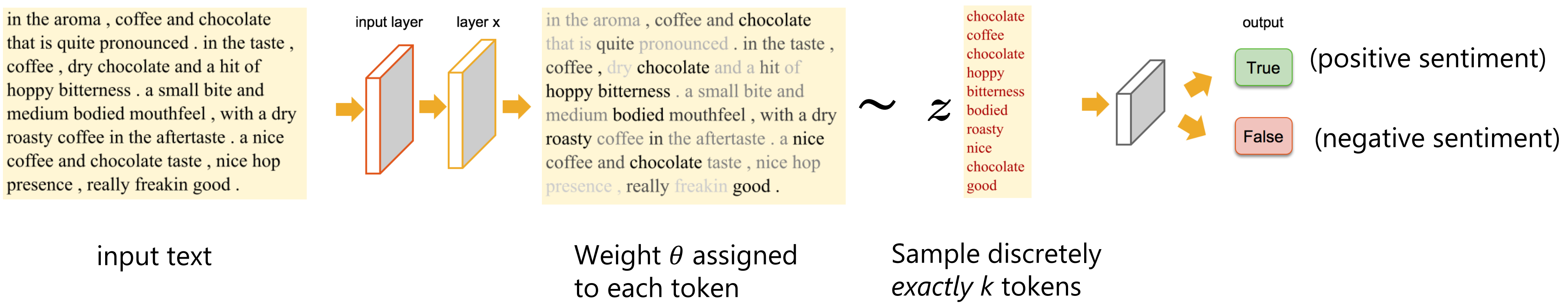
- Learning to Plan
  1. A complex neural network learns to assign weights to cells in a map
  2. Weights are used as input to a shortest-path solver
  3. A shortest path is returned by the solver and used in a downstream loss function, comparing the shortest path with a gold shortest path



Pogancic et al. Differentiation of Blackbox Combinatorial Solvers, ICLR 2019

Motivating Example 2

- Learning to Explain
  1. A complex neural network learns to assign weights to input features
  2. Weights are treated as parameters of a discrete distribution with *k*-subset constraint
  3. Subset of size *exactly k* is sampled and used as input to the classification model
  4. At test time, the argmax (MAP) is used to select *k* most important words



Chen et al. Learning to Explain: An Information-Theoretic Perspective on Model Interpretation, ICML 2018

Discrete Exponential Family Distribution

◆ We can turn any discrete combinatorial optimization problem (with linear objective) into a discrete probability distribution

$\boldsymbol{z}$ is a configuration of m *binary* variables

$$\max \ \boldsymbol{\theta}^T \boldsymbol{z}$$

$$\textit{subject to } \boldsymbol{z} \in \mathcal{C}$$

$\mathcal{C}$ is integral polytope

$$p(\boldsymbol{z};\boldsymbol{\theta}) = \begin{cases} \exp\left(\langle \boldsymbol{z}, \boldsymbol{\theta} \rangle - A(\boldsymbol{\theta})\right) & \text{if } \boldsymbol{z} \in \mathcal{C}, \\ 0 & \text{otherwise.} \end{cases}$$

Assigns probability mass to every z which statisfies contraints

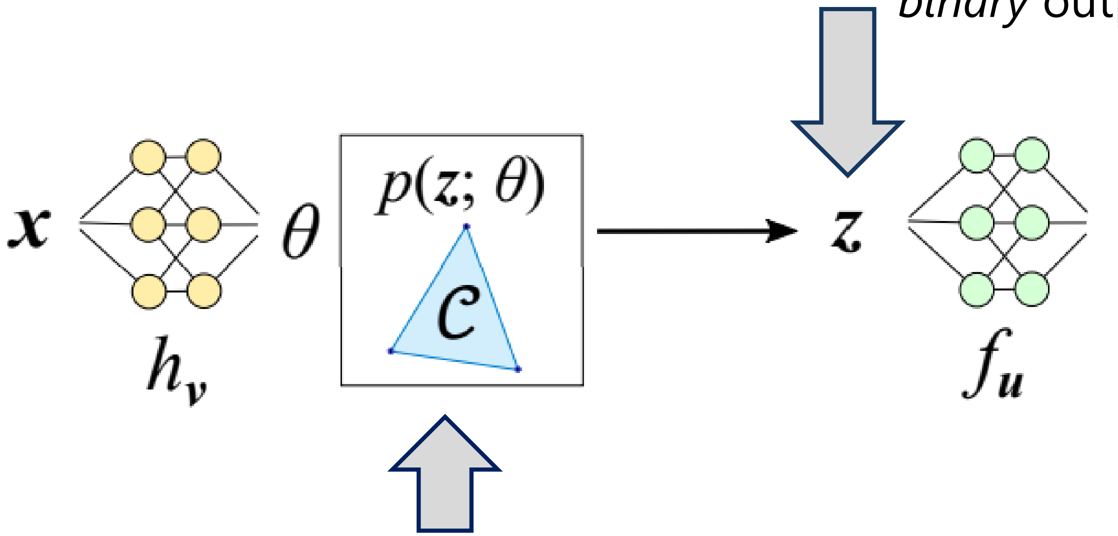
$$\operatorname{argmax} \boldsymbol{\theta}^T \boldsymbol{z}$$

$$\textit{subject to } \boldsymbol{z} \in \mathcal{C}$$

Maximum a-posteriori (MAP) state of p
(a most probable configuration)

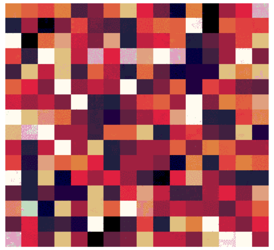
Problem Definition

z is a *binary* vector of size m, representing the *binary* output of the *discrete* component



$$L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega}) = \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z};\boldsymbol{\theta})} \left[\ell(f_{\boldsymbol{u}}(\boldsymbol{z}), \boldsymbol{y}) \right]$$

$\boldsymbol{\theta}$



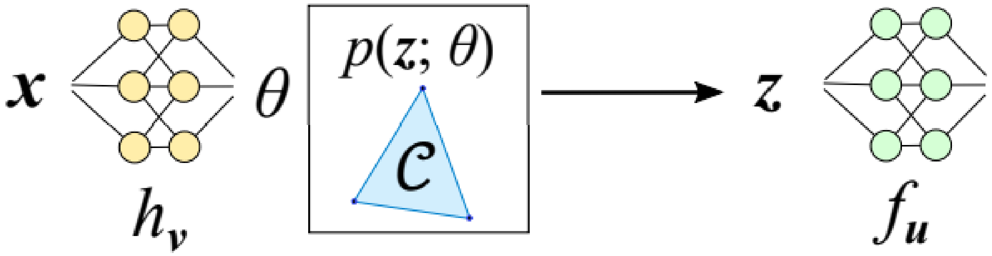
We treat the *discrete* component as a **blackbox**:

1. We *only* assume that the combinatorial optimization algorithm can be executed when given input $\boldsymbol{\theta}$
2. We *only* assume ability to compute MAP (most probable) states of the discrete probability distribution with parameters $\boldsymbol{\theta}$

$$\boldsymbol{z} \quad \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Central question: How do we compute/estimate $\nabla_{\boldsymbol{\theta}} L$?

Maximum Likelihood Learning (MLE)



ℓ directly compares $\boldsymbol{z}$ with *gold* $\boldsymbol{y}$

Central question: How do we compute/estimate $\boldsymbol{\nabla}_{\boldsymbol{\theta}} L$?

For the model above, we have exact MLE gradients for any $\boldsymbol{y}$:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\nabla}_{\boldsymbol{\theta}}[-\log p\,(\boldsymbol{y}; \boldsymbol{\theta})] = \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z};\, \boldsymbol{\theta})}[\boldsymbol{z}] - \boldsymbol{y}$$

We only need a way to approximate the *marginals* $\mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z};\, \boldsymbol{\theta})}[\boldsymbol{z}]$
(we use perturb and MAP explained on the next slide)

Maximum likelihood learning reduces the KL divergence between the model distribution $p\,(\boldsymbol{y}; \boldsymbol{\theta})$ and the data distribution

The setting has been addressed by several prior methods
[Pogancic et al. 2019, Berthet et al. 2020]

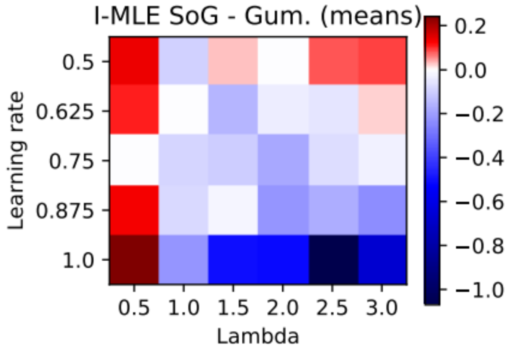
Perturb and MAP

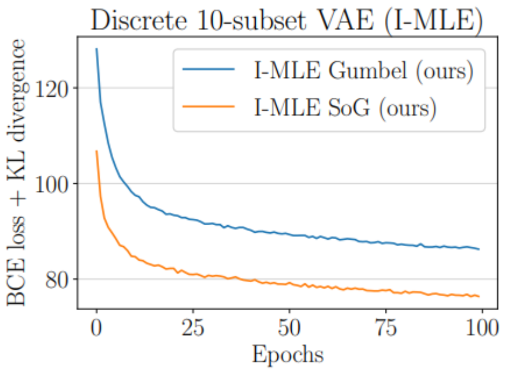
- Since we only assume the ability to compute MAP states (execute the discrete component), we use *local* perturb and MAP to approximately sample from p

Probability distribution from which we draw noise $\boldsymbol{\epsilon}$ to perturb $\boldsymbol{\theta}$

$$\overline{\boldsymbol{z}} = \mathtt{MAP}(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \text{ and } \boldsymbol{\epsilon} \sim \rho(\boldsymbol{\epsilon})$$

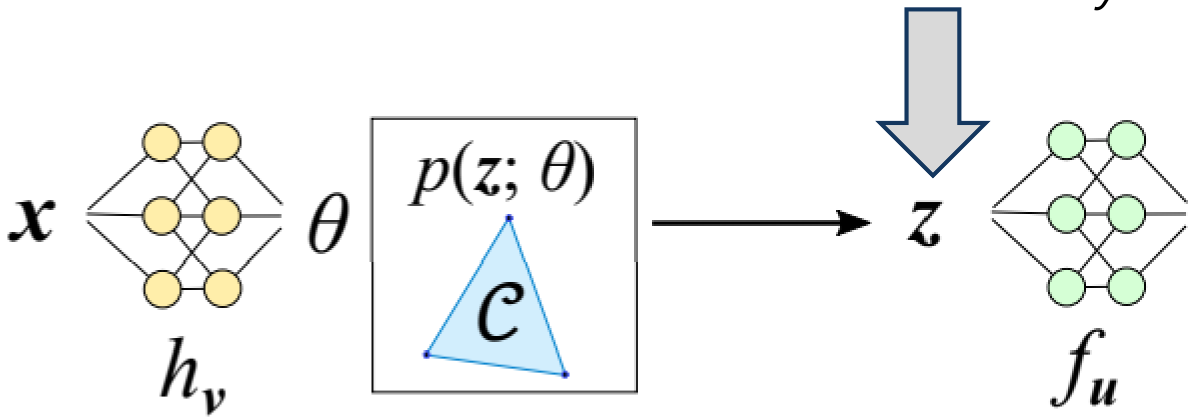
- We introduce a new Sum-of-Gamma distribution for noise perturbations which has beneficial properties





Implicit MLE

$\boldsymbol{z}$ is a *binary* vector of size m, representing the *binary* output of the *discrete* component



$$L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega}) = \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z};\boldsymbol{\theta})} \left[\ell(f_{\boldsymbol{u}}(\boldsymbol{z}), \boldsymbol{y}) \right]$$

Central question: How do we compute/estimate $\boldsymbol{\nabla}_{\boldsymbol{\theta}} L$?

In the above model, we do ***not*** have access to the empirical distribution $q(\boldsymbol{z}; \boldsymbol{\theta}^{'})$ ($\boldsymbol{y}$ here is not of the same type as the *latent* $\boldsymbol{z}$) → vanilla MLE ***not*** applicable

Idea: Construct a *surrogate empirical distribution* (= *target distribution*) $q(\boldsymbol{z}; \boldsymbol{\theta}^{'})$

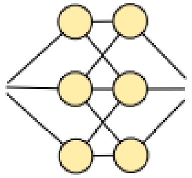
Target Distribution

- Based on perturbation-based implicit differentiation [Domke 2010]
- Change the parameters θ using the downstream, loss-induced gradients

$$q(\boldsymbol{z};\boldsymbol{\theta}') = p(\boldsymbol{z};\boldsymbol{\theta} - \underbrace{\lambda\nabla_{\boldsymbol{z}}\ell(f_{\boldsymbol{u}}(\overline{\boldsymbol{z}}),\boldsymbol{y})}_{\text{Gradient of point-wise loss function wrt } \boldsymbol{z}}) \text{ with } \underbrace{\overline{\boldsymbol{z}} = \texttt{MAP}(\boldsymbol{\theta}+\boldsymbol{\epsilon}) \text{ and } \boldsymbol{\epsilon} \sim \rho(\boldsymbol{\epsilon})}_{\text{Approximate sampling via perturb and MAP}}$$

- **Note:** Straight-through estimator uses these loss-induced gradients directly

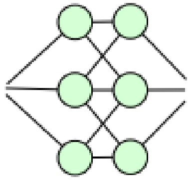
Putting the Pieces Together

$\boldsymbol{x}$  θ

$h_{\boldsymbol{v}}$

$$\overline{\boldsymbol{z}} = \mathtt{MAP}(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \text{ and } \boldsymbol{\epsilon} \sim \rho(\boldsymbol{\epsilon})$$

Perturb and MAP

$\overline{\boldsymbol{z}}$  $\ell(f_{\boldsymbol{u}}(\overline{\boldsymbol{z}}), \boldsymbol{y})$

$f_{\boldsymbol{u}}$

$$q(\boldsymbol{z}; \boldsymbol{\theta}') = p(\boldsymbol{z}; \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{z}} \ell(f_{\boldsymbol{u}}(\overline{\boldsymbol{z}}), \boldsymbol{y}))$$

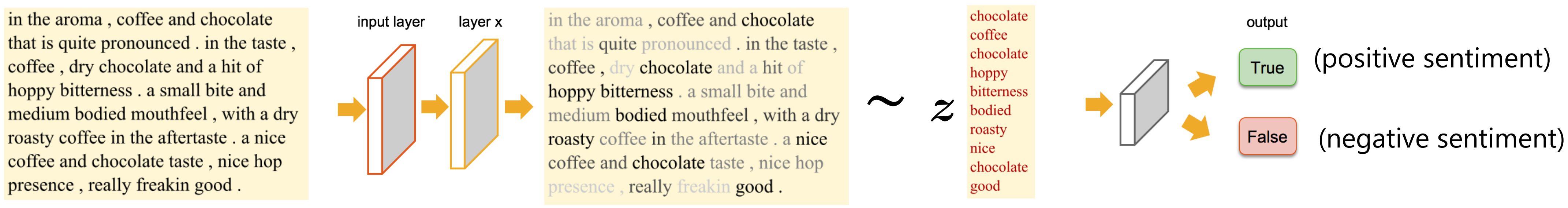
Construct target distribution

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L \approx \mathtt{MAP}(\boldsymbol{\theta} + \boldsymbol{\epsilon}) - \mathtt{MAP}(\boldsymbol{\theta}' + \boldsymbol{\epsilon})$$

Compute approximate MLE gradients

Experiments

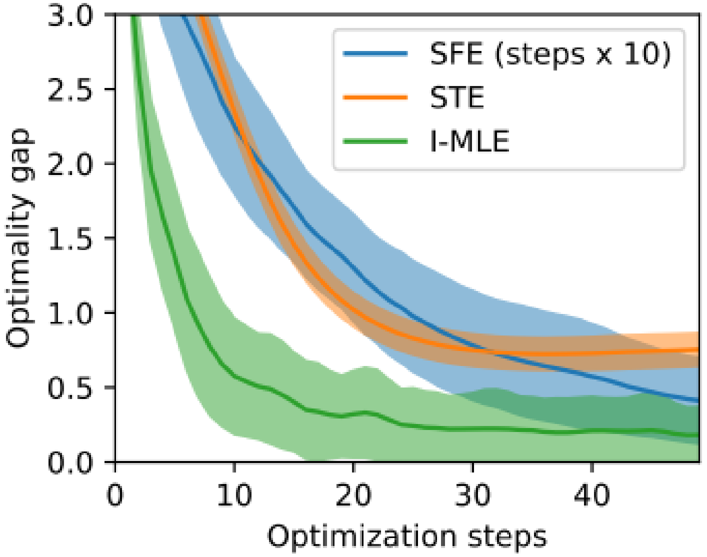
◆ Learning to Explain sentiment scoring

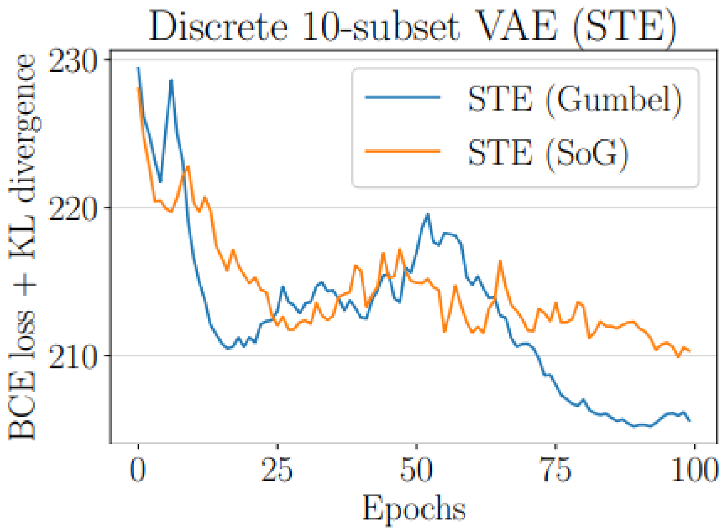


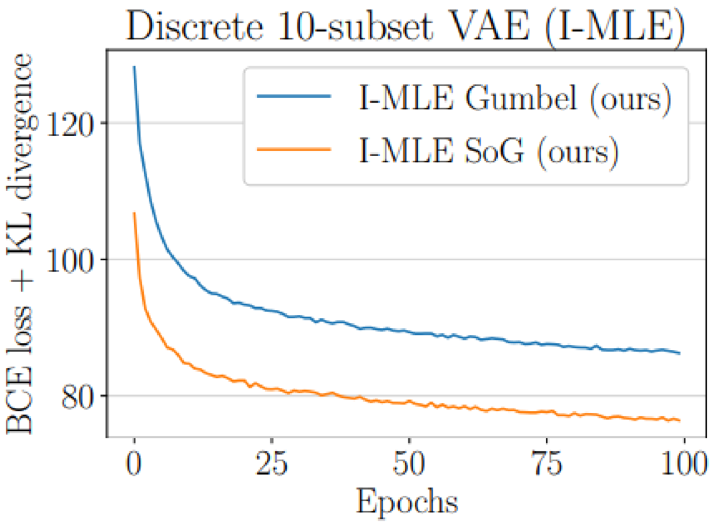
Method	Test MSE		Subset Precision	
	Mean	**Std. Dev.**	**Mean**	**Std. Dev.**
		$k = 10$		
L2X ($t = 0.1$)	6.68	1.08	26.65	9.39
SoftSub ($t = 0.5$)	**2.67**	0.14	44.44	2.27
STE ($\tau = 30$)	4.44	0.09	38.93	0.14
I-MLE MAP	4.08	0.91	14.55	0.04
I-MLE Gumbel	**2.68**	0.10	39.28	2.62
I-MLE ($\tau = 30$)	**2.71**	0.10	**47.98**	2.26

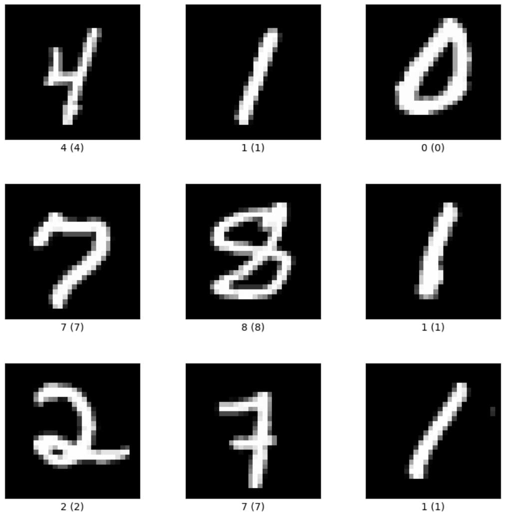
Experiments

Straight-through estimator vs.
Score function estimator vs.
I-MLE





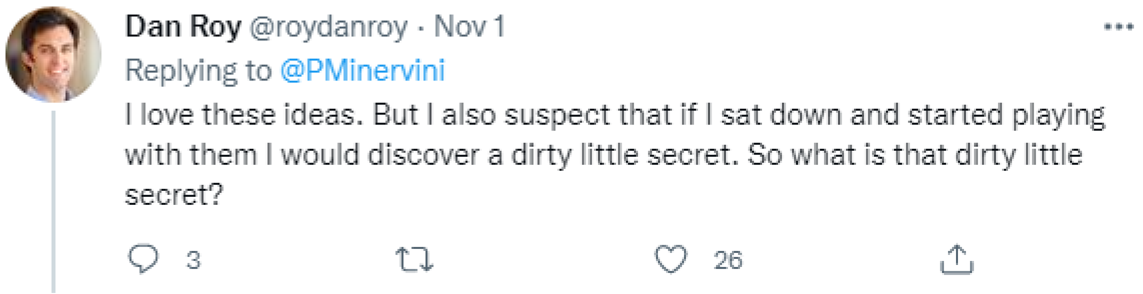




Additional Applications

- Discrete world models
- Neural Causal Discovery
- Reinforcement learning with complex multi-step actions
- Relational Structure Discovery (e.g., for GNNs)
- Integrating statistical relational models into deep learning architectures

Limitations and “Dirty Little Secrets”



- Noise perturbation temperature and target distribution are hyperparameters of the approach and need tuning
- We have found stable learning behavior across temperatures but final results are sensitive to these parameters

Code Repository Available

Repositories with TF and PyTorch code

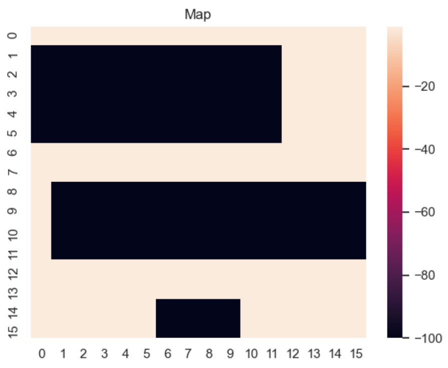
https://github.com/nec-research/tf-imle

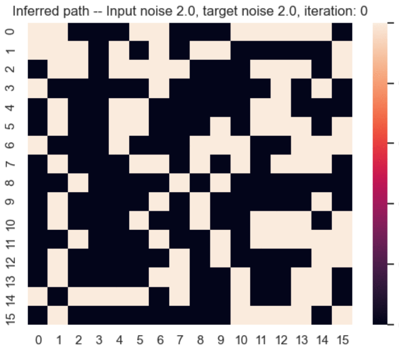
https://github.com/uclnlp/torch-imle

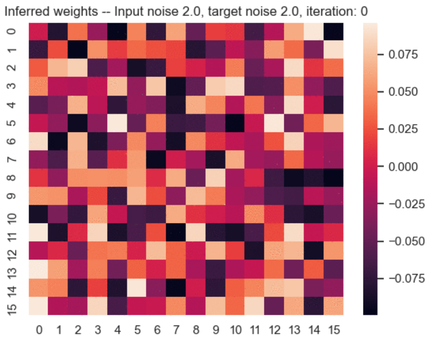
Algorithm 1 Instance of I-MLE with perturbation-based implicit differentiation.

function FORWARDPASS($\boldsymbol{\theta}$)
  // Sample from the noise distribution $\rho(\boldsymbol{\epsilon})$
  $\boldsymbol{\epsilon} \sim \rho(\boldsymbol{\epsilon})$
  // Compute a MAP state of perturbed $\boldsymbol{\theta}$
  $\hat{\boldsymbol{z}} = \texttt{MAP}(\boldsymbol{\theta} + \boldsymbol{\epsilon})$
  save $\boldsymbol{\theta}$, $\boldsymbol{\epsilon}$, and $\hat{\boldsymbol{z}}$ for the backward pass
  return $\hat{\boldsymbol{z}}$

function BACKWARDPASS($\nabla_{\boldsymbol{z}} \ell(f_{\boldsymbol{u}}(\boldsymbol{z}), \hat{\boldsymbol{y}}), \lambda$)
  load $\boldsymbol{\theta}$, $\boldsymbol{\epsilon}$, and $\hat{\boldsymbol{z}}$ from the forward pass
  // Compute target distribution parameters
  $\boldsymbol{\theta}' = \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{z}} \ell(f_{\boldsymbol{u}}(\boldsymbol{z}), \hat{\boldsymbol{y}})$
  // Single sample I-MLE gradient estimate
  $\widehat{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{\theta}}') = \hat{\boldsymbol{z}} - \texttt{MAP}(\boldsymbol{\theta}' + \boldsymbol{\epsilon})$
  return $\widehat{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{\theta}}')$








```python
@tf.custom_gradient
def subset_k(self, logits, k):

    # sample discretely with perturb and map
    z_train = self.sample_discrete_forward(logits)
    # compute the top-k discrete values
    threshold = tf.expand_dims(tf.nn.top_k(logits, self.k, sorted=True)[0][:,-1], -1)
    z_test = tf.cast(tf.greater_equal(logits, threshold), tf.float32)
    # at training time we sample, at test time we take the argmax
    z_output = K.in_train_phase(z_train, z_test)

    def custom_grad(dy):

        # we perturb (implicit diff) and then resuse sample for perturb and MAP
        map_dy = self.sample_discrete_backward(logits - (self._lambda*dy))
        # we now compute the gradients as the difference (I-MLE gradients)
        grad = tf.math.subtract(z_train, map_dy)
        # return the gradient
        return grad, k
```