UCLA

Rethinking Probabilistic Circuit Parameter Learning

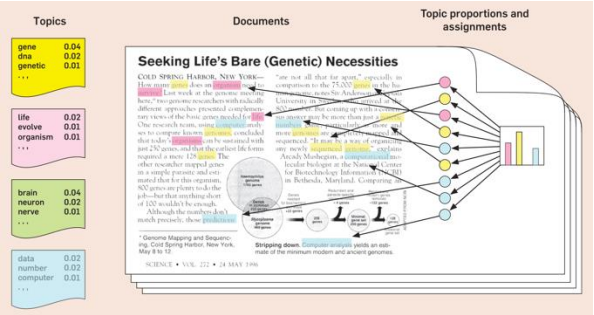
Anji Liu, Zilei Zoe Shao, Guy Van den Broeck

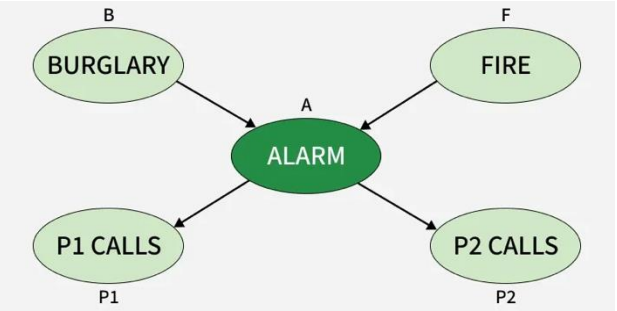
AISTATS 2026

Probabilistic Methods in the Era of LLMs

(Probabilistic) Machine Learning in 2000

- Dominance of Probabilistic Graphical Models
- Tractable inference
- Incorporation of domain knowledge/constraints

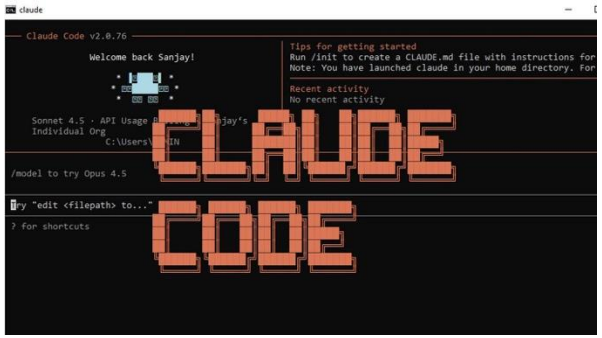


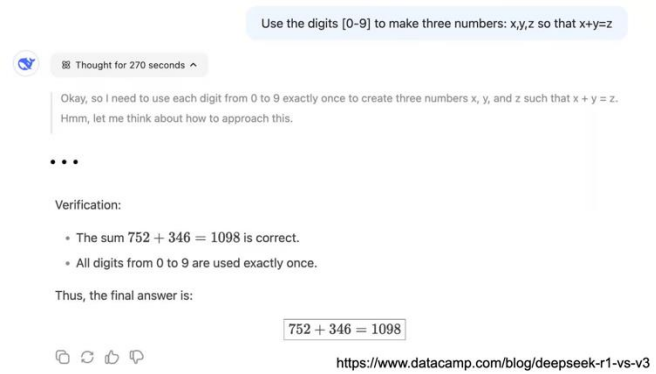


26 years

(Probabilistic) Machine Learning in 2026

- Better model architectures
- Massive data
- Improved prompting techniques

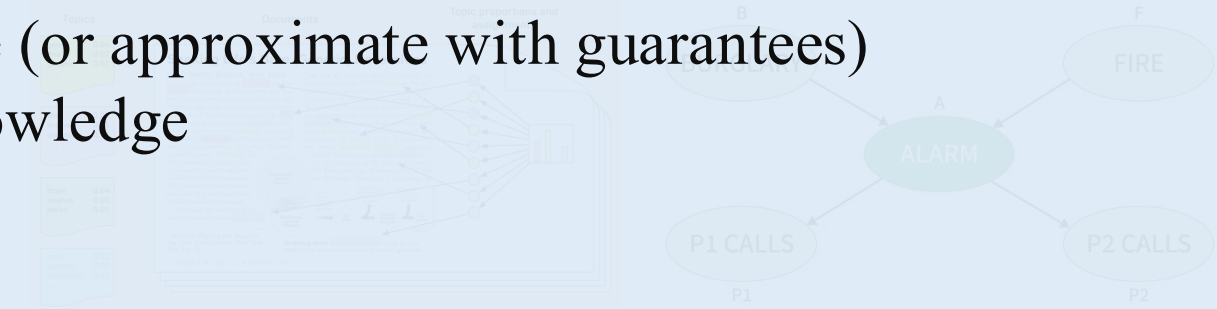




Probabilistic Methods in the Era of LLMs

(Probabilistic) Machine Learning in 2000

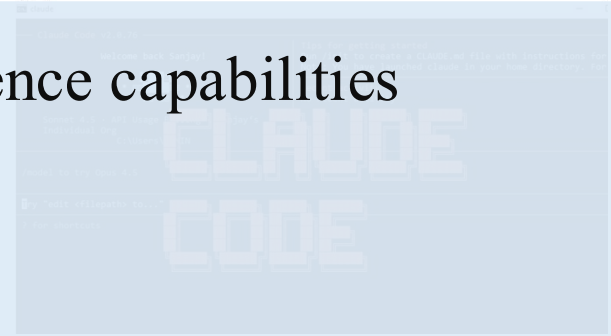
- Exact probabilistic inference (or approximate with guarantees)
- Incorporation of domain knowledge



26 years

(Probabilistic) Machine Learning in 2026

- Limited probabilistic inference capabilities
- Can only learn from data



Probabilistic Methods in the Era of LLMs

(Probabilistic) Machine Learning in 2000

- Exact probabilistic inference (or approximate with guarantees)
- Incorporation of domain knowledge

... but ...

- **Hard to scale**

26 years

(Probabilistic) Machine Learning in 2026

- Limited probabilistic inference capabilities
- Can only learn from data

... but ...

- **Extremely scalable**

Probabilistic Graphical Models Are Hard to Scale

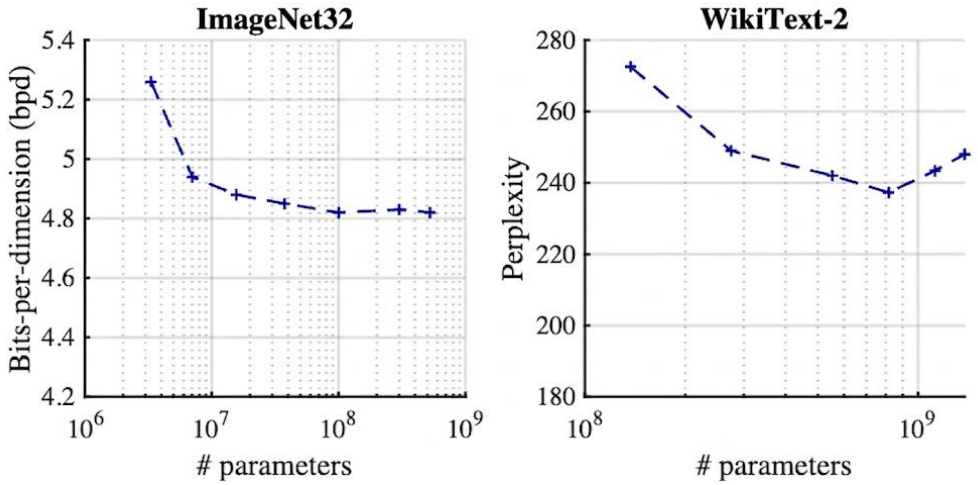
Structure learning is provably hard

Large-Sample Learning of Bayesian Networks is NP-Hard

David Maxwell Chickering DMAX@MICROSOFT.COM
David Heckerman HECKERMA@MICROSOFT.COM
Christopher Meek MEEK@MICROSOFT.COM
Microsoft Research
Redmond, WA 98052, USA

Parameter learning given a fixed structure is empirically hard

Log-likelihood saturates as the parameter count goes up (Liu et al.)

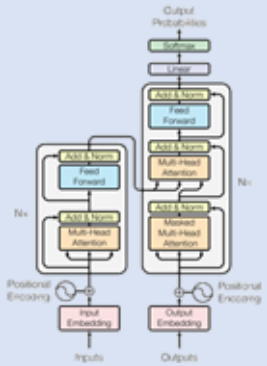


A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

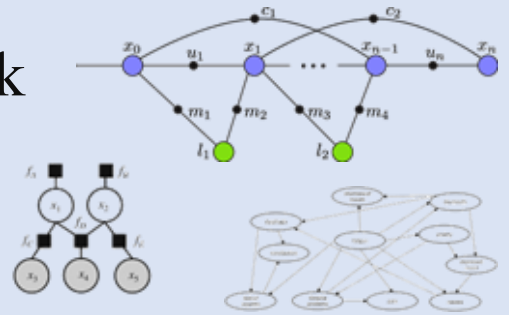
- Transformer
- Mamba
- Etc.



Probabilistic Graphical Models

Scalable Representation

- Bayesian Network
- Factor Graph
- Etc.

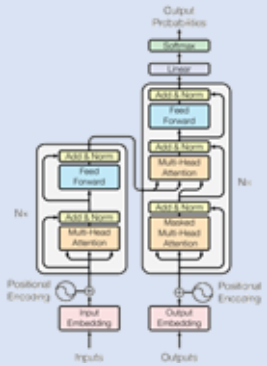


A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

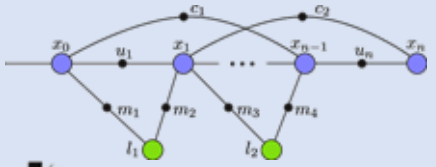
- Transformer
- Mamba
- Etc.

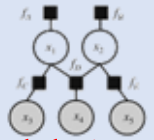


Probabilistic Graphical Models

Scalable Representation

- Bayesian Network
- Factor Graph
- Etc.





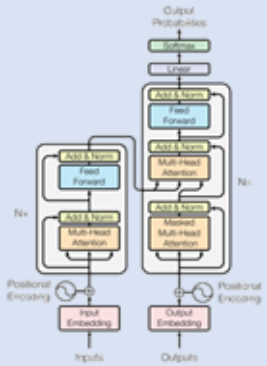
Not computationally scalable!

A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

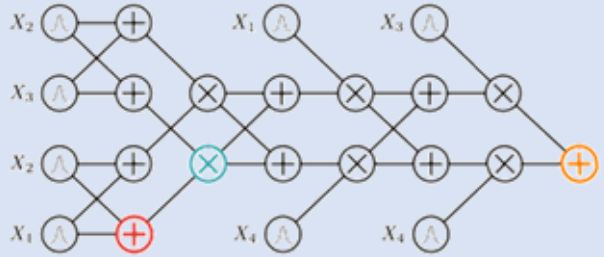
- Transformer
- Mamba
- Etc.



Probabilistic Graphical Models

Scalable Representation

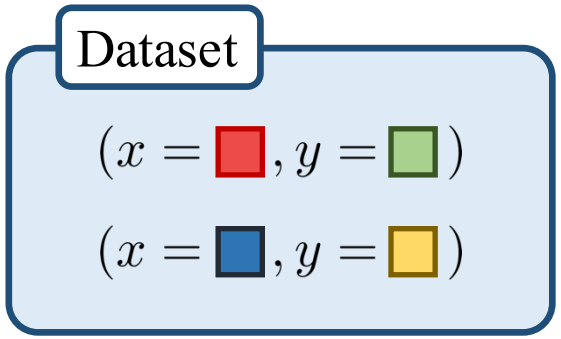
- Probabilistic Circuit (or Sum-Product Network)



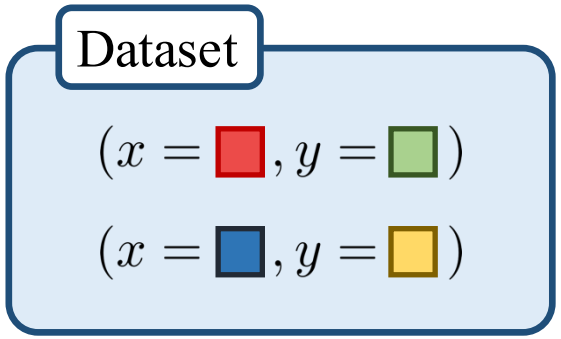
Tractable Deep Generative Models: Probabilistic Circuits

Encode complex distributions by starting with simple distributions and recursively combining them to represent more complex ones.

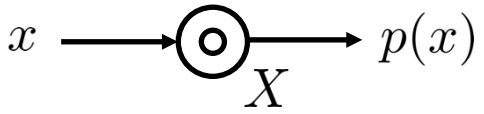
Tractable Deep Generative Models: Probabilistic Circuits



Tractable Deep Generative Models: Probabilistic Circuits

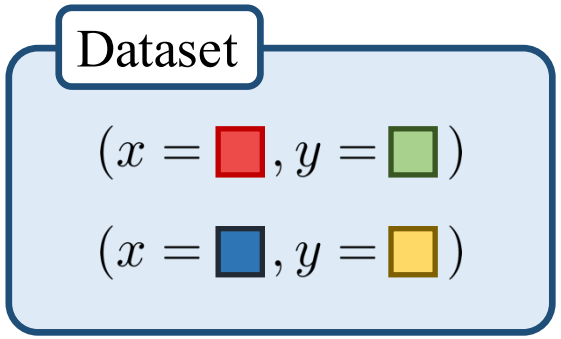


- **Input nodes** encode univariate primitive distributions.

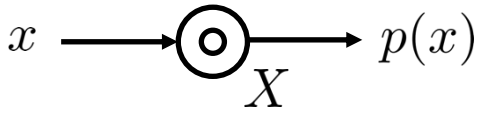


$$p(x) = \begin{cases} 1 & x = \square, \\ 0 & \text{otherwise.} \end{cases}$$

Tractable Deep Generative Models: Probabilistic Circuits

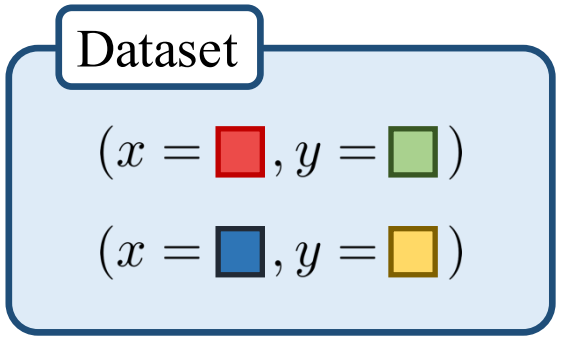


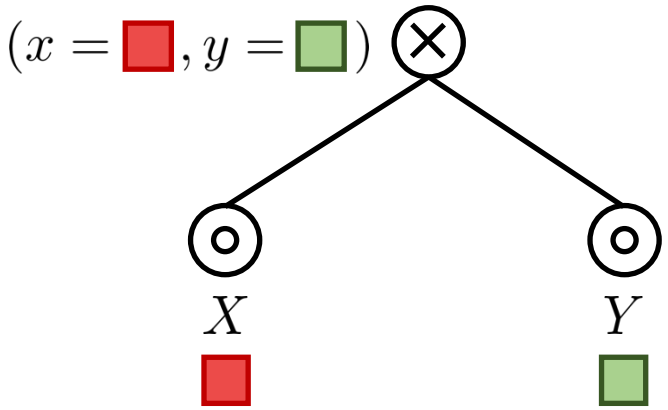
- **Input nodes** encode univariate primitive distributions.



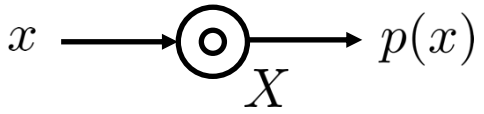
$$p(x) = \begin{cases} 1 & x = \square, \\ 0 & \text{otherwise.} \end{cases} \qquad p(y) = \begin{cases} 1 & y = \square, \\ 0 & \text{otherwise.} \end{cases}$$

Tractable Deep Generative Models: Probabilistic Circuits

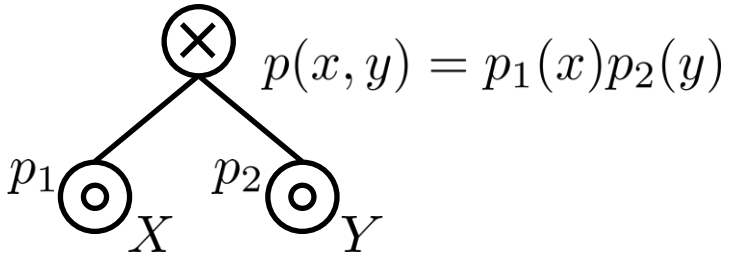




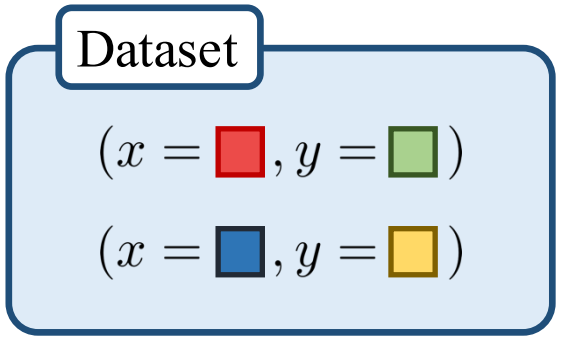
- **Input nodes** encode univariate primitive distributions.



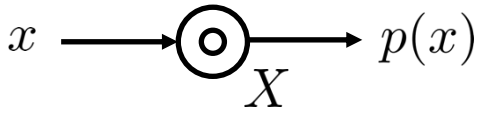
- **Product nodes** build factorized distributions over their children.



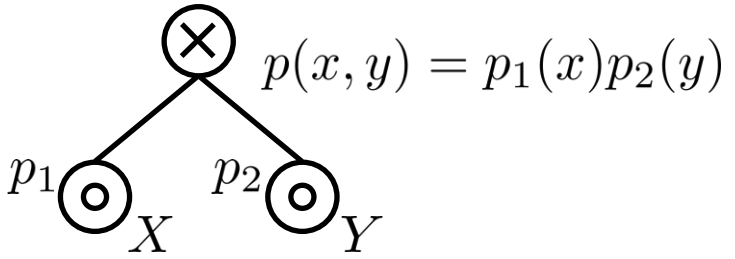
Tractable Deep Generative Models: Probabilistic Circuits

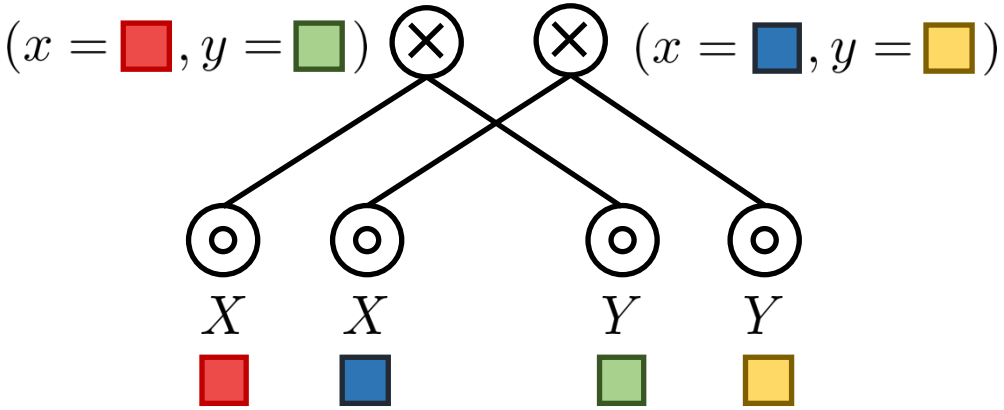


- **Input nodes** encode univariate primitive distributions.

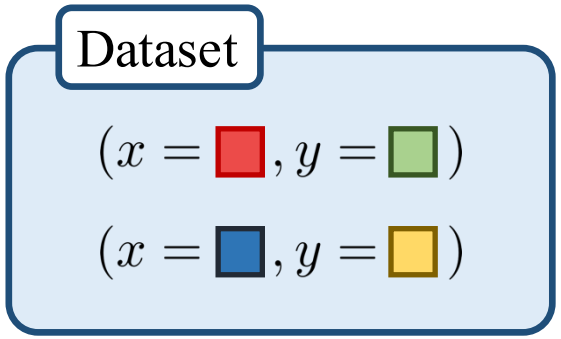


- **Product nodes** build factorized distributions over their children.

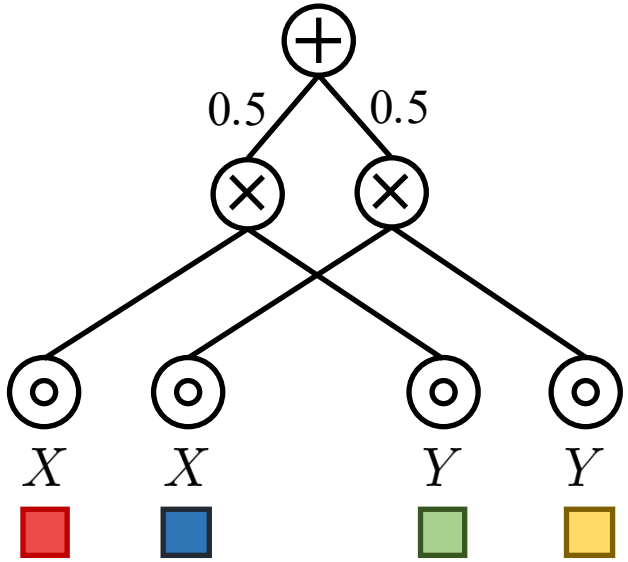




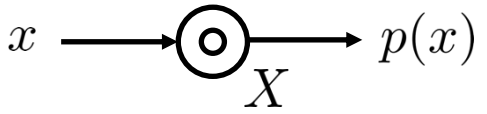
Tractable Deep Generative Models: Probabilistic Circuits



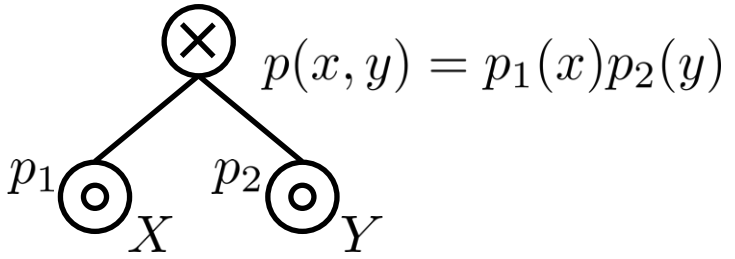
$0.5 \cdot (x = \square, y = \square) + 0.5 \cdot (x = \square, y = \square)$



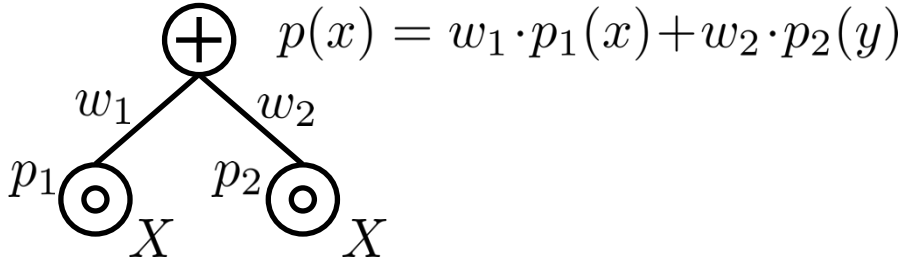
- **Input nodes** encode univariate primitive distributions.



- **Product nodes** build factorized distributions over their children.

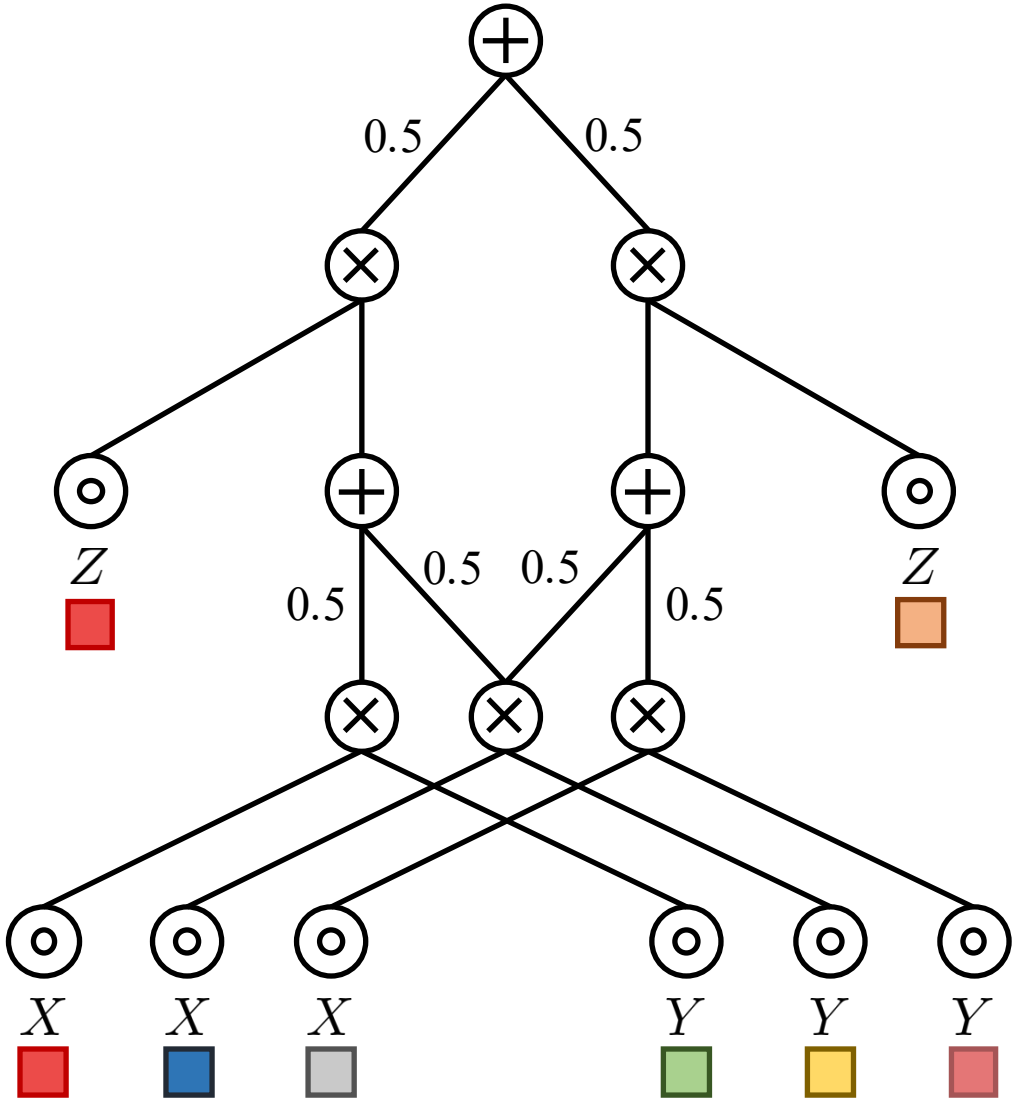


- **Sum nodes** construct weighted mixtures over their child distributions.



Compute Likelihood

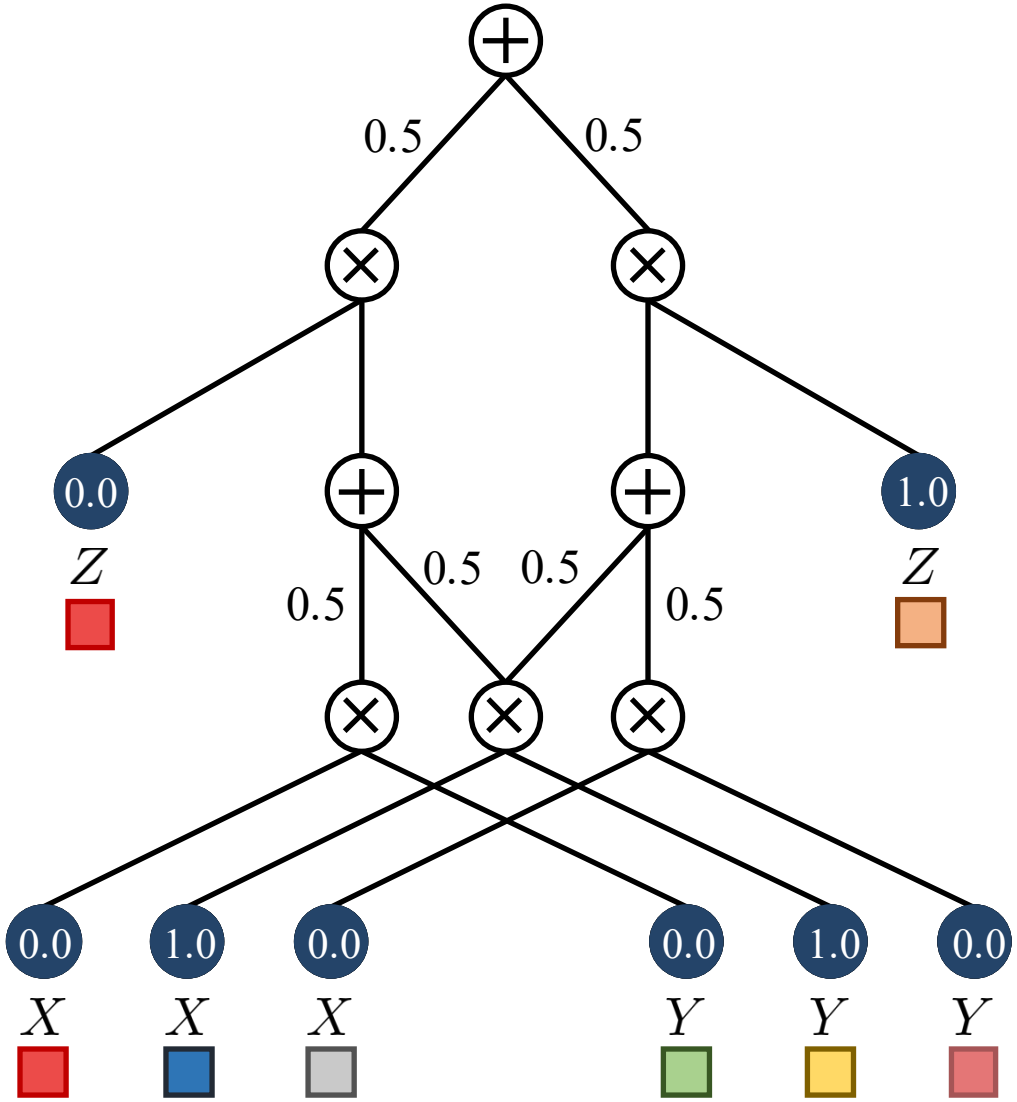
Compute $p(x = \blacksquare, y = \blacksquare, z = \blacksquare)$



Compute Likelihood

Compute $p(x = \blacksquare, y = \blacksquare, z = \blacksquare)$

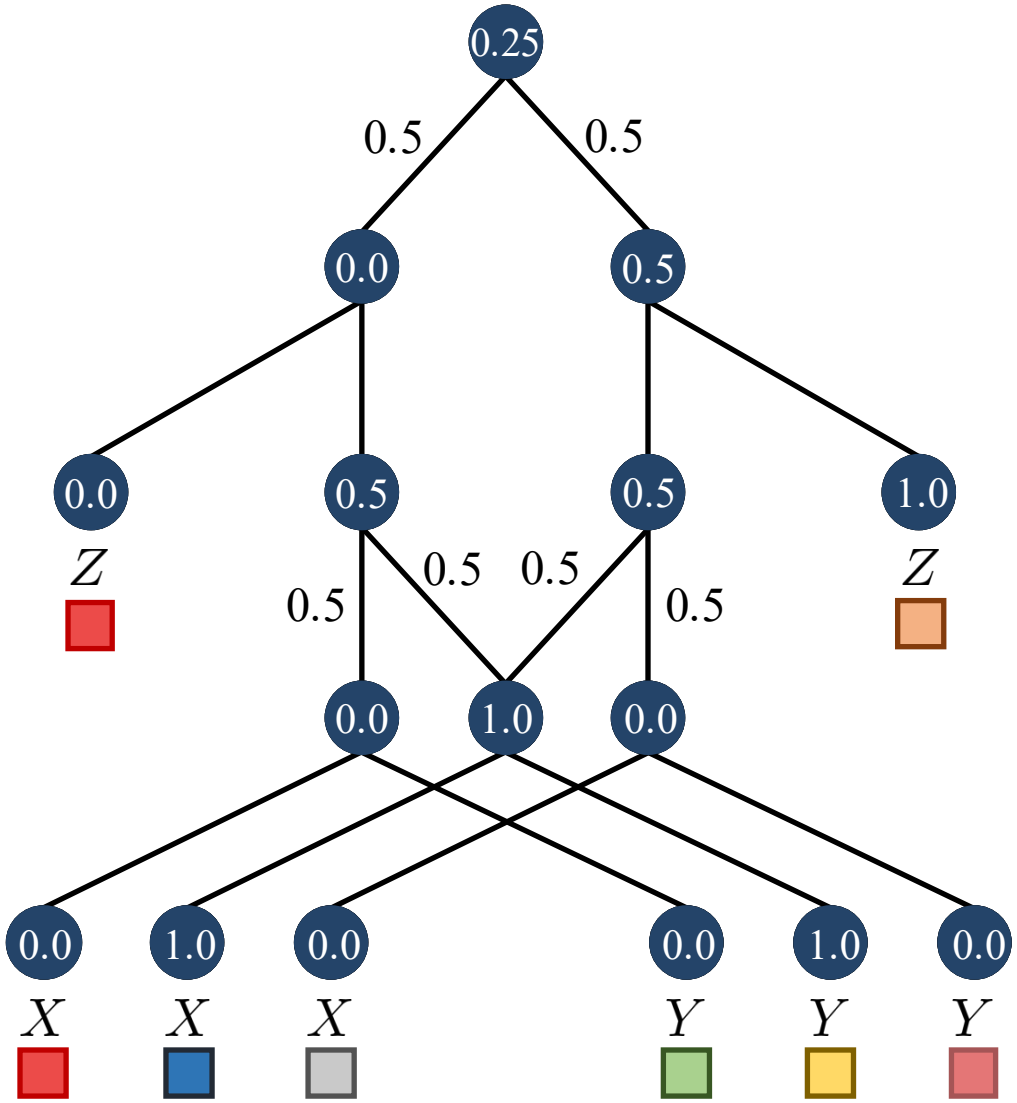
- Compute the likelihood of every **input node**.



Compute Likelihood

Compute $p(x = \blacksquare, y = \blacksquare, z = \blacksquare) = 0.25$

- Compute the likelihood of every **input node**.
- Compute the likelihood of every **sum/product node**.
- Readout likelihood from the **root node**.



Computing Marginal

Compute $p(x = \square) = \iint p(x = \square, y, z)dydz$

- **Sum node** $\oplus_a$

$$\iint p_a(x = \square, y, z)dydz$$

$$= \iint 0.5 \cdot p_b(x = \square, y, z) + 0.5 \cdot p_c(x = \square, y, z)dydz$$

$$= 0.5 \underbrace{\iint p_b(x = \square, y, z)dydz}_{\iint \otimes_b dydz} + 0.5 \underbrace{\iint p_c(x = \square, y, z)dydz}_{\iint \otimes_c dydz}$$

- **Product node** $\otimes_b$

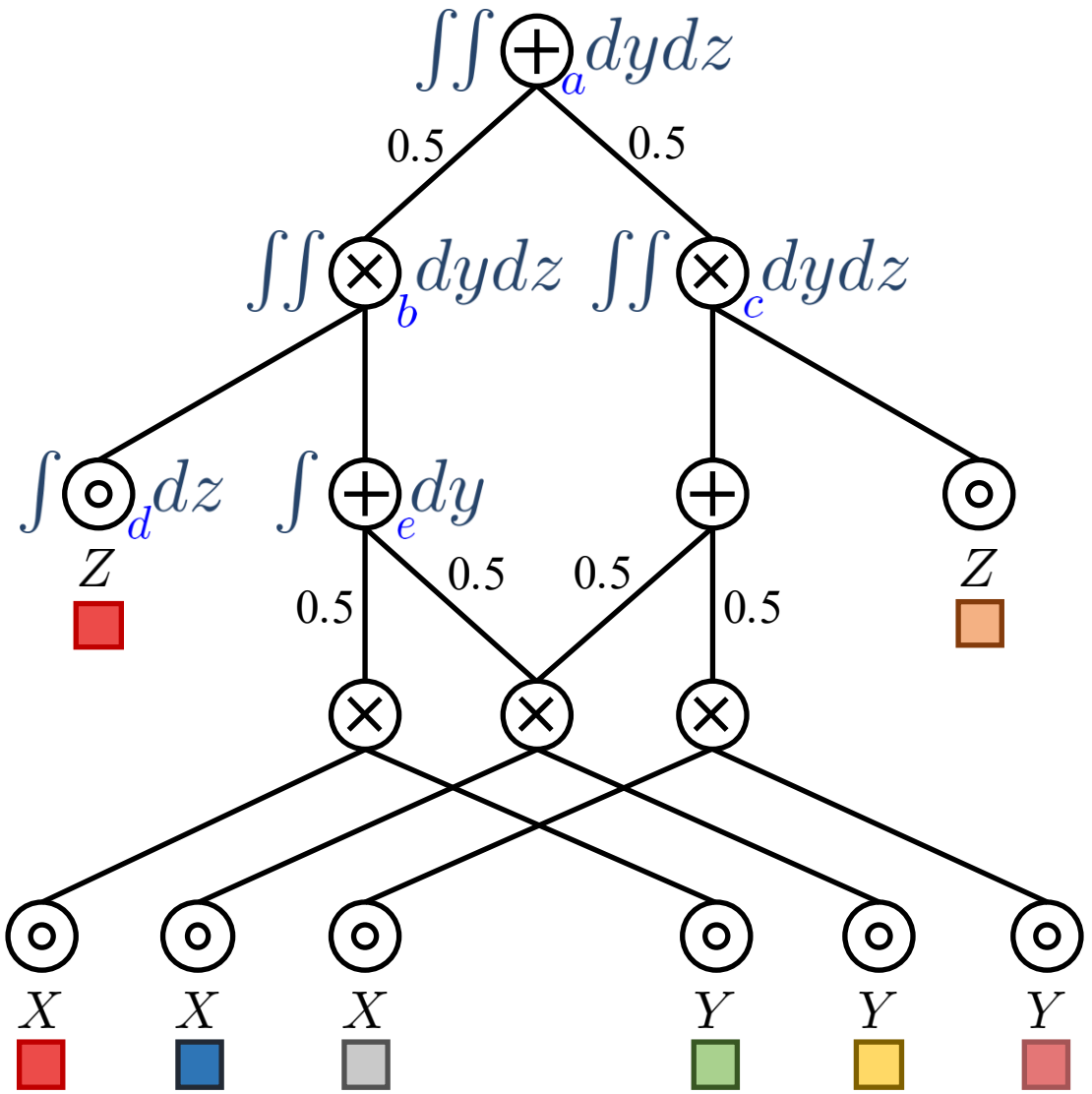
$$\iint p_b(x = \square, y, z)dydz$$

$$= \iint p_d(z) \cdot p_e(x = \square, y)dydz$$

$$= \underbrace{\int p_d(z)dz}_{\int \otimes_d dz} \cdot \underbrace{\int p_e(x = \square, y)dy}_{\int \otimes_e dy}$$

- **Input node** $\odot_d$

$$\int p_d(z) = 1$$

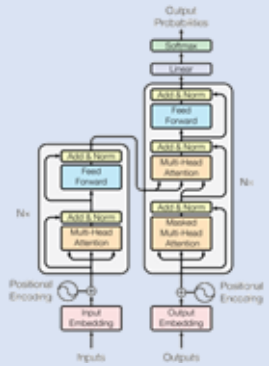


A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

- Transformer
- Mamba
- Etc.



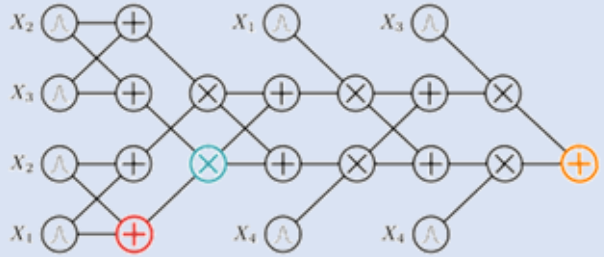
Efficient Systems

PyTorch

Probabilistic Graphical Models

Scalable Representation

- Probabilistic Circuit



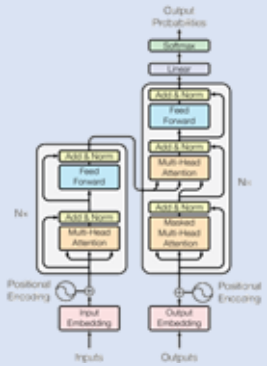
Efficient Systems

A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

- Transformer
- Mamba
- Etc.



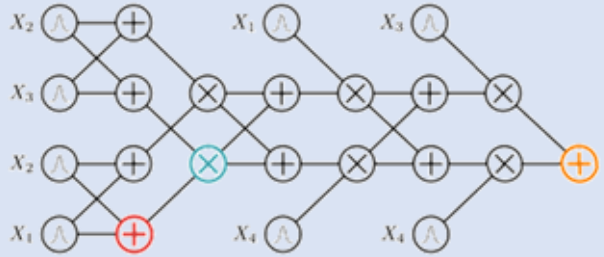
Efficient Systems

PyTorch

Probabilistic Graphical Models

Scalable Representation

- Probabilistic Circuit



Efficient Systems

PyJuice

- **100x speedup** vs. Google baselines.
- Scaled to **billions of parameters**.

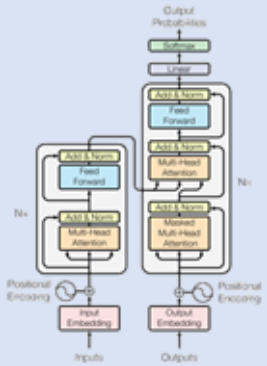
https://github.com/Tractables/pyjuice

A Pathway Towards Scaling Up Probabilistic Graphical Models

Neural Networks

Scalable Representation

- Transformer
- Mamba
- Etc.



Efficient Systems

PyTorch

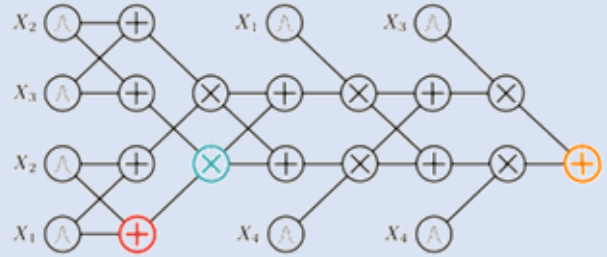
Strong Optimizers

Adam, Muon, etc.

Probabilistic Graphical Models

Scalable Representation

- Probabilistic Circuit



Efficient Systems

PyJuice

- **100x speedup** vs. Google baselines.
- Scaled to **billions of parameters**.

Strong Optimizers

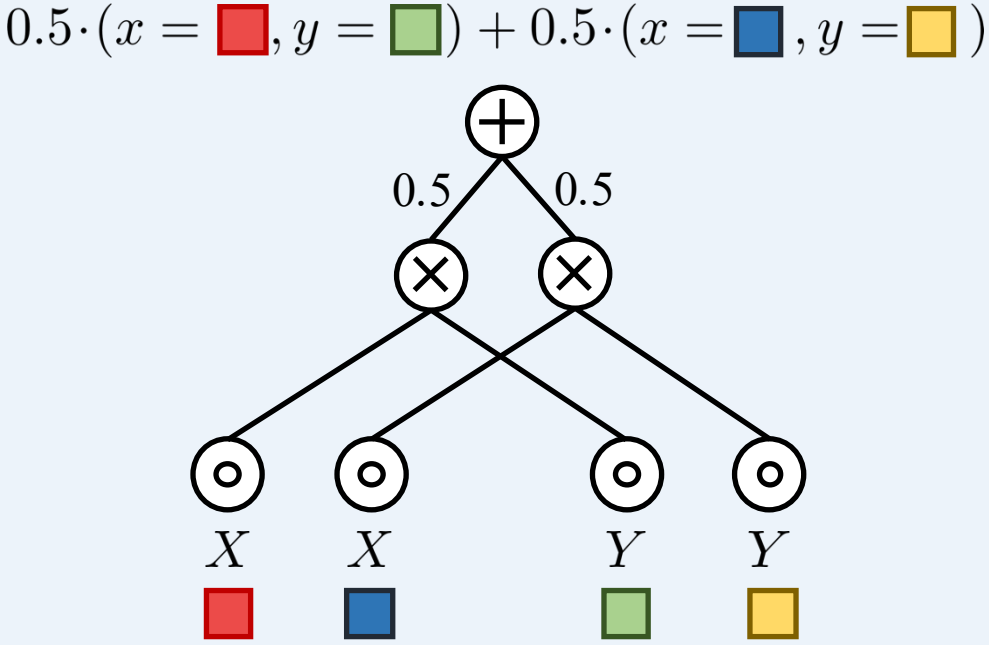
This work.

Optimizing PCs with Expectation Maximization

PCs are latent variable models

$$p_{\phi}(\boldsymbol{x}) = \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{x}, \boldsymbol{z})$$

Example

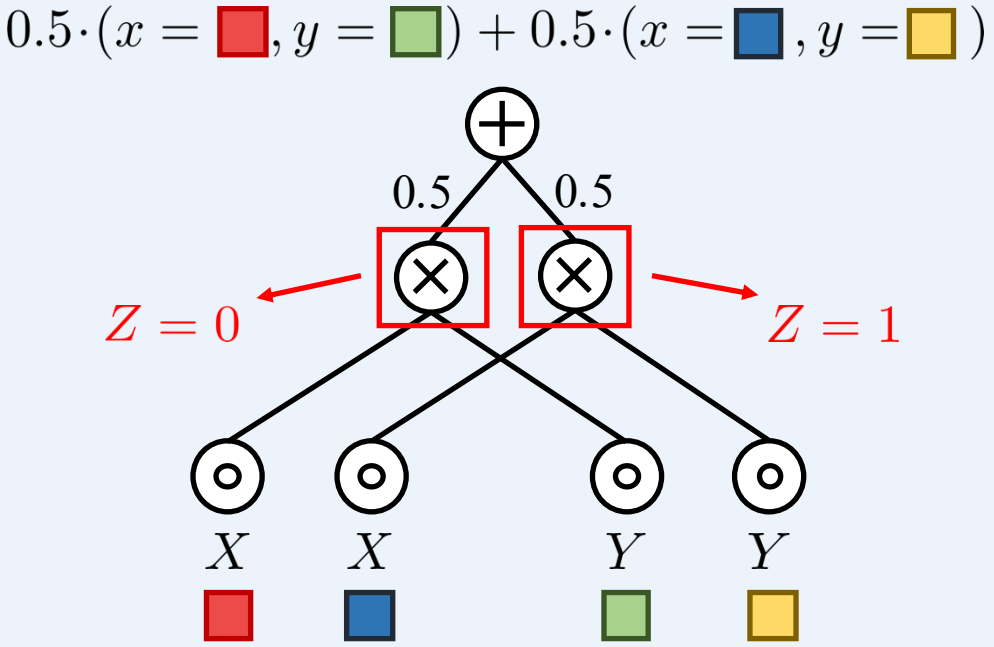


Optimizing PCs with Expectation-Maximization

PCs are latent variable models

$$p_{\phi}(\boldsymbol{x}) = \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{x}, \boldsymbol{z})$$

Example



Optimizing PCs with Expectation-Maximization

PCs are latent variable models

$$p_{\phi}(\boldsymbol{x}) = \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{x}, \boldsymbol{z})$$

Standard Expectation-Maximization

Instead of optimizing the log-likelihood

$$\mathrm{LL}(\phi) := \log p_{\phi}(\boldsymbol{x}) = \log \left(\sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{x}, \boldsymbol{z}) \right),$$

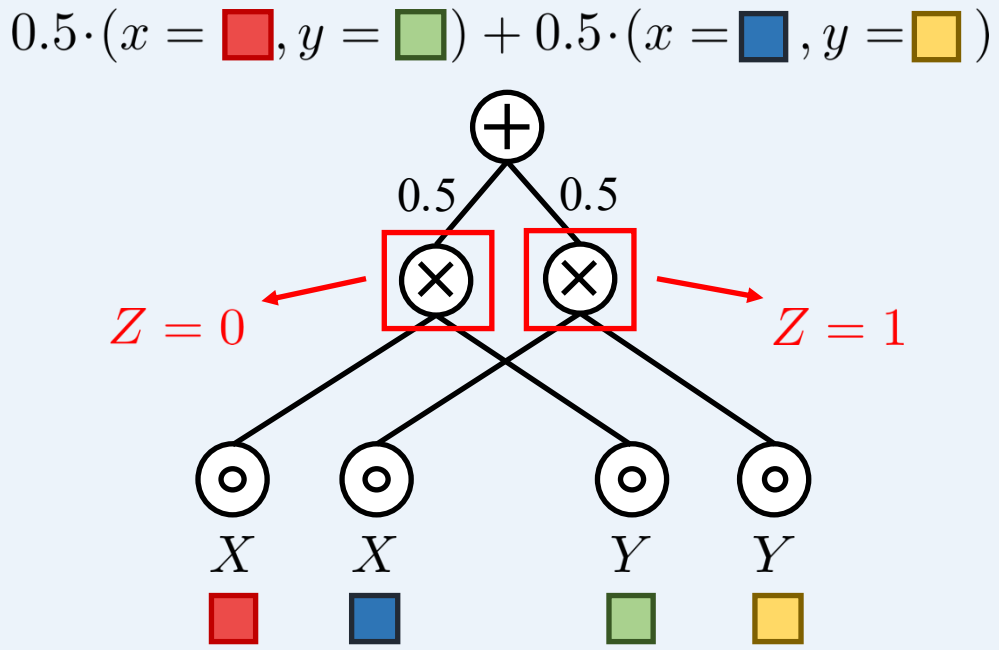
we maximize the following lower-bound

$$Q_{\phi}(\phi') := \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\phi'}(\boldsymbol{x}, \boldsymbol{z}).$$

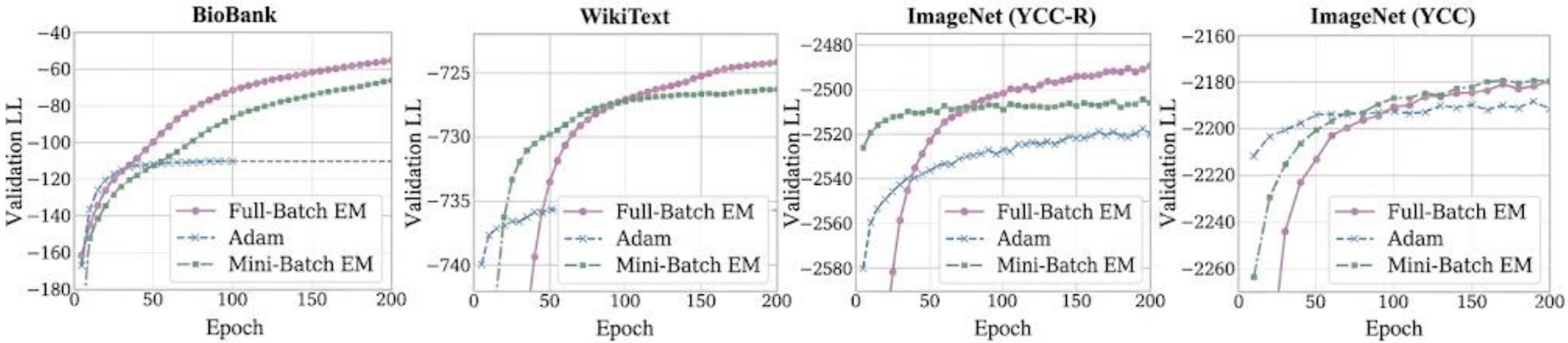
ϕ : the current parameters

ϕ' : parameters to be optimized

Example



The Pros and Cons of Full-Batch Expectation-Maximization



Pros:

- The best performance compared to baselines
- No need for hyperparameter tuning

Cons

- Infrequent updates
- Hard to scale to large datasets
- Slow convergence

The Existing Mini-Batch EM Algorithm

Full-batch EM:

Given a dataset $\mathcal{D}_{\text{full}}$, solve for $\boldsymbol{\phi}'$ that maximizes

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}_{\text{full}}}(\boldsymbol{\phi}') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{full}}} \sum_{\boldsymbol{z}} p_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\boldsymbol{\phi}'}(\boldsymbol{x},\boldsymbol{z}).$$

The Existing Mini-Batch EM Algorithm

Full-batch EM:

Given a dataset $\mathcal{D}_{\text{full}}$, solve for ϕ' that maximizes

$$Q_{\phi}^{\mathcal{D}_{\text{full}}}(\phi') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{full}}} \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\phi'}(\boldsymbol{x}, \boldsymbol{z}).$$

Mini-batch EM:

Given a mini-batch of data $\mathcal{D}_{\text{mini}}$, solve for ϕ' that maximizes

$$Q_{\phi}^{\mathcal{D}_{\text{mini}}}(\phi') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{mini}}} \sum_{\boldsymbol{z}} p_{\phi}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\phi'}(\boldsymbol{x}, \boldsymbol{z}).$$

The Existing Mini-Batch EM Algorithm

Full-batch EM:

Given a dataset $\mathcal{D}_{\text{full}}$, solve for $\boldsymbol{\phi}'$ that maximizes

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}_{\text{full}}}(\boldsymbol{\phi}') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{full}}} \sum_{\boldsymbol{z}} p_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\boldsymbol{\phi}'}(\boldsymbol{x},\boldsymbol{z}).$$

Mini-batch EM:

Given a mini-batch of data $\mathcal{D}_{\text{mini}}$, solve for $\boldsymbol{\phi}'$ that maximizes

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}_{\text{mini}}}(\boldsymbol{\phi}') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{mini}}} \sum_{\boldsymbol{z}} p_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\boldsymbol{\phi}'}(\boldsymbol{x},\boldsymbol{z}).$$

However, $\mathcal{D}_{\text{mini}}$ does not represent the entire dataset, so we should not "trust" $\boldsymbol{\phi}'$ entirely.

The Existing Mini-Batch EM Algorithm

Full-batch EM:

Given a dataset $\mathcal{D}_{\text{full}}$, solve for $\boldsymbol{\phi}'$ that maximizes

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}_{\text{full}}}(\boldsymbol{\phi}') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{full}}} \sum_{\boldsymbol{z}} p_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\boldsymbol{\phi}'}(\boldsymbol{x},\boldsymbol{z}).$$

Mini-batch EM:

Given a mini-batch of data $\mathcal{D}_{\text{mini}}$, solve for $\boldsymbol{\phi}'$ that maximizes

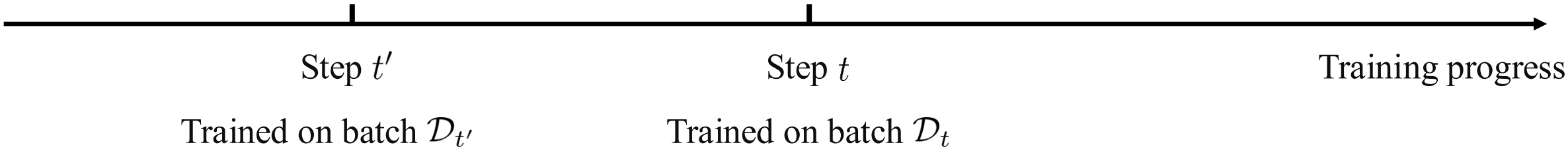
$$Q_{\boldsymbol{\phi}}^{\mathcal{D}_{\text{mini}}}(\boldsymbol{\phi}') := \sum_{\boldsymbol{x}\in\mathcal{D}_{\text{mini}}} \sum_{\boldsymbol{z}} p_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \cdot \log p_{\boldsymbol{\phi}'}(\boldsymbol{x},\boldsymbol{z}).$$

However, $\mathcal{D}_{\text{mini}}$ does not represent the entire dataset, so we should not "trust" $\boldsymbol{\phi}'$ entirely.

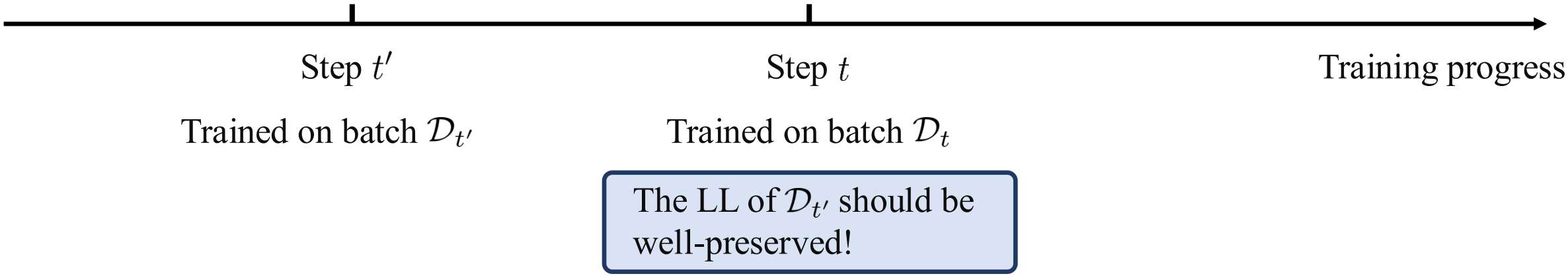
Instead, we update the parameters "incrementally":

$$\boldsymbol{\phi}_{\text{new}} := (1-\alpha)\cdot\boldsymbol{\phi} + \alpha\cdot\boldsymbol{\phi}', \text{ where } \alpha \in (0,1] \text{ is a learning rate.}$$

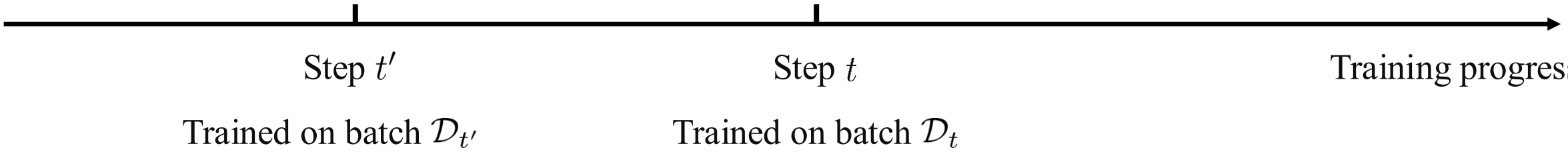
The “Catastrophic Forgetting” Problem



The “Catastrophic Forgetting” Problem



The "Catastrophic Forgetting" Problem

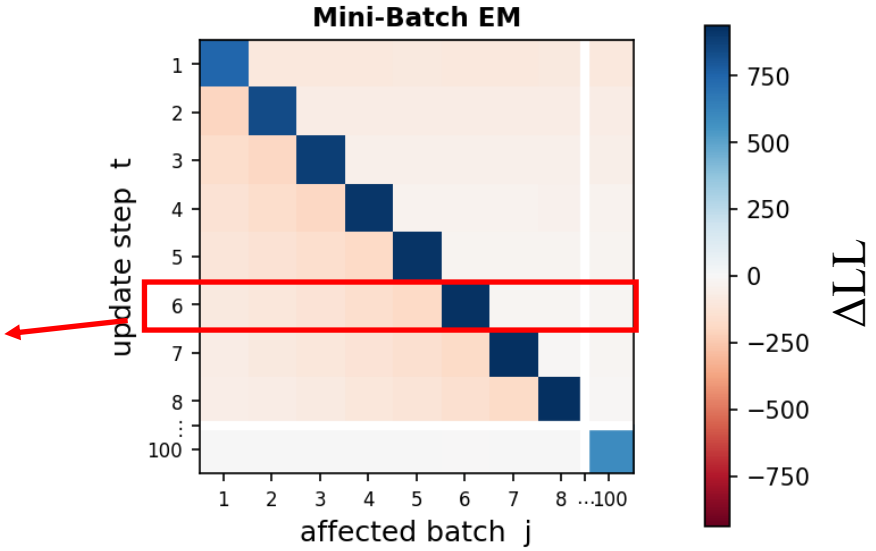


Trained on batch $\mathcal{D}_{t'}$

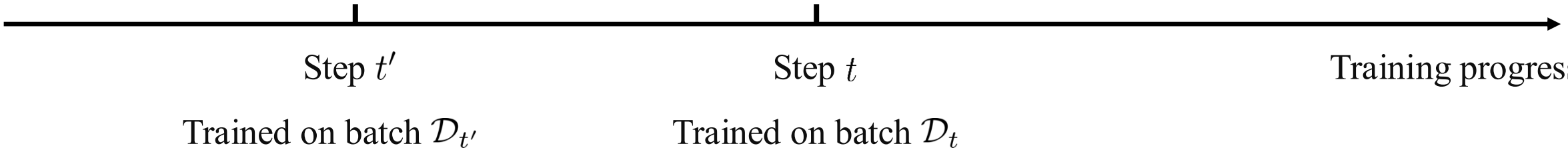
Trained on batch $\mathcal{D}_t$

The LL of $\mathcal{D}_{t'}$ should be well-preserved!

After training on batch #6 (in epoch 100)



The "Catastrophic Forgetting" Problem

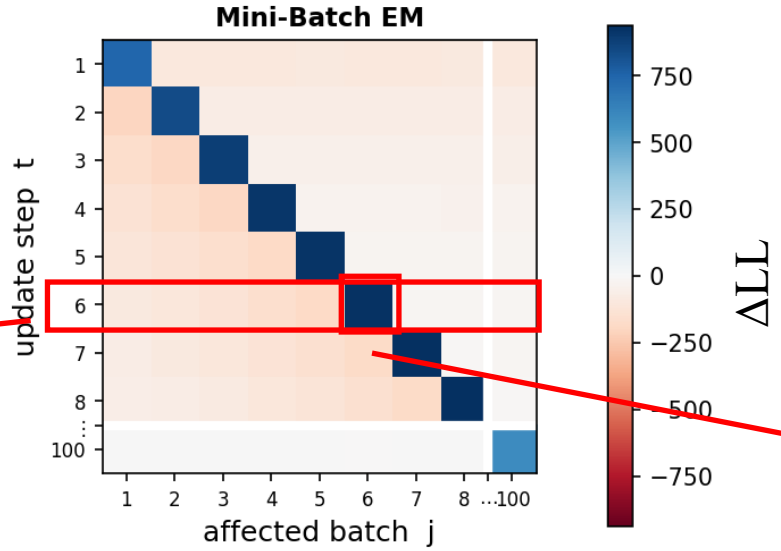


Trained on batch $\mathcal{D}_{t'}$

Trained on batch $\mathcal{D}_t$

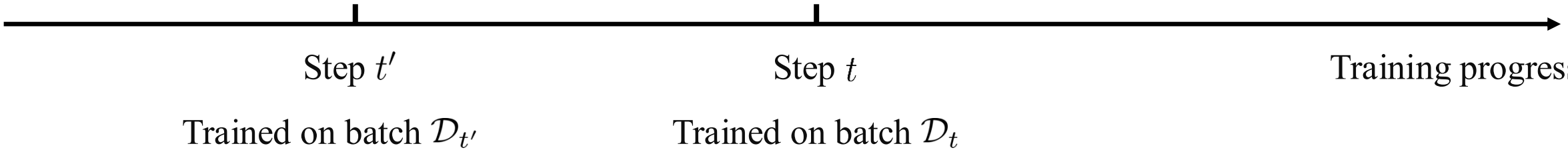
The LL of $\mathcal{D}_{t'}$ should be well-preserved!

After training on batch #6 (in epoch 100)



LL of the current batch increases

The "Catastrophic Forgetting" Problem



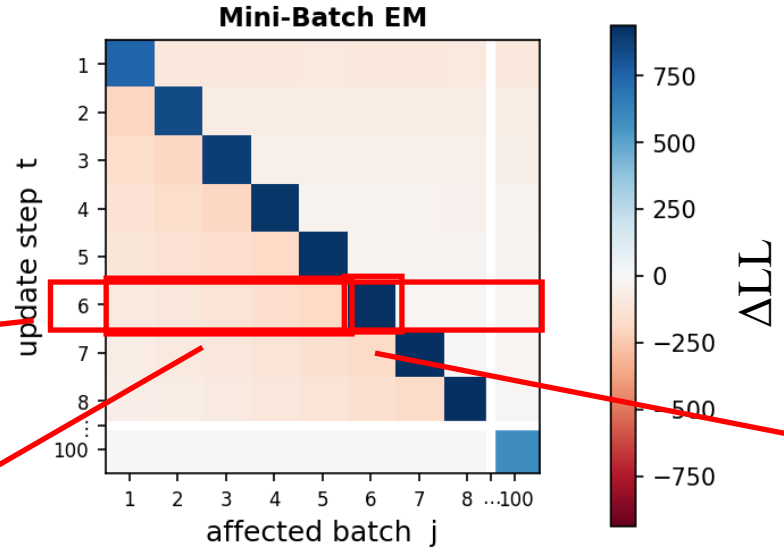
Trained on batch $\mathcal{D}_{t'}$

Trained on batch $\mathcal{D}_t$

The LL of $\mathcal{D}_{t'}$ should be well-preserved!

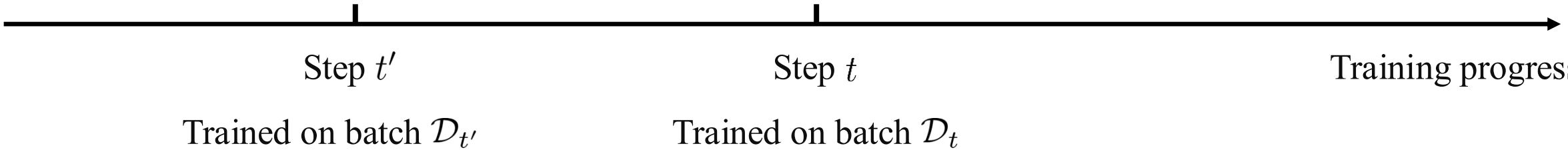
After training on batch #6 (in epoch 100)

LL of the previous batches went down



LL of the current batch increases

The "Catastrophic Forgetting" Problem

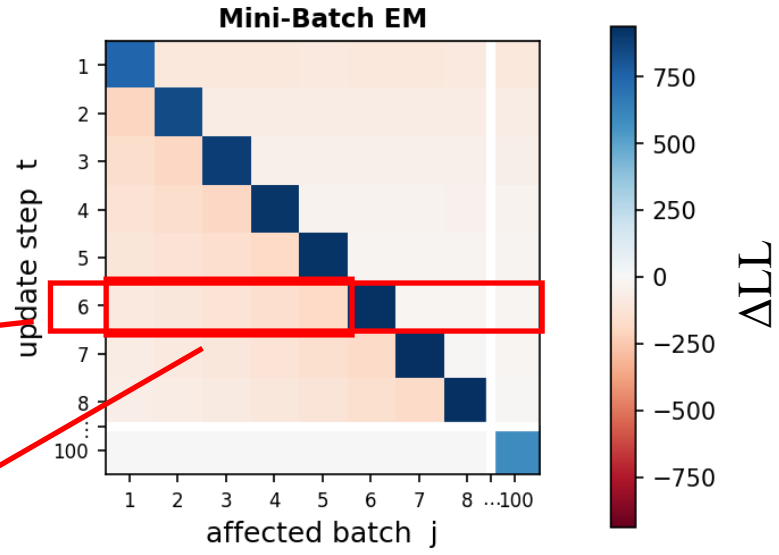


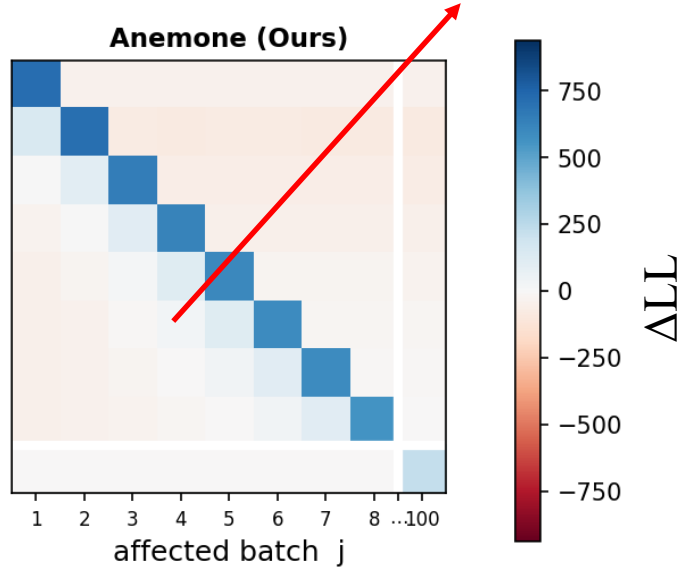
The LL of $\mathcal{D}_{t'}$ should be well-preserved!

Our optimizer largely maintains the LL of previous batches

After training on batch #6 (in epoch 100)

LL of the previous batch went down





Understanding the Catastrophic Forgetting Problem

Reinterpreting full-batch EM*

$$Q_{\phi}^{\mathcal{D}}(\phi') = \boxed{\frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\phi}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\phi}(\boldsymbol{x})}{\partial \phi}, \phi' - \phi \right\rangle} - \boxed{\mathtt{KL}_{\phi}(\phi')}$$

First-order Taylor expansion of the LL.

KL divergence: penalize changing the parameters too much.

*Based on some results from the paper "Homeomorphic-Invariance of EM: Non-Asymptotic Convergence in KL Divergence for Exponential Families via Mirror Descent". Kudos to the authors Frederik Kunstner, Raunak Kumar, and Mark Schmidt.

Understanding the Catastrophic Forgetting Problem

Reinterpreting full-batch EM*

$$Q_{\phi}^{\mathcal{D}}(\phi') = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\phi}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\phi}(\boldsymbol{x})}{\partial \phi}, \phi' - \phi \right\rangle - \mathtt{KL}_{\phi}(\phi')$$

Catastrophic forgetting happens because neither mini-batch EM nor Adam penalizes the parameters from **"changing too much"** properly.

*Based on some results from the paper "Homeomorphic-Invariance of EM: Non-Asymptotic Convergence in KL Divergence for Exponential Families via Mirror Descent". Kudos to the authors Frederik Kunstner, Raunak Kumar, and Mark Schmidt.

Understanding the Catastrophic Forgetting Problem

Reinterpreting full-batch EM*

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}}(\boldsymbol{\phi}') = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle - \mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}')$$

Catastrophic forgetting happens because neither mini-batch EM nor Adam penalizes the parameters from **"changing too much"** properly.

Gradient-based methods:

$$\text{maximize}_{\boldsymbol{\phi}'} \ \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle - \gamma\|\boldsymbol{\phi} - \boldsymbol{\phi}'\|_2^2$$

Small L2 distance in the parameter space could lead to large discrepancy in the distributions. Also vice versa.

Understanding the Catastrophic Forgetting Problem

Reinterpreting full-batch EM*

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}}(\boldsymbol{\phi}') = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle - \mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}')$$

Catastrophic forgetting happens because neither mini-batch EM nor Adam penalizes the parameters from **"changing too much"** properly.

Mini-batch EM:

Similar to gradient-based methods, it only regularizes on the parameter space.

The Anemone Algorithm

Reinterpreting full-batch EM*

$$Q_{\boldsymbol{\phi}}^{\mathcal{D}}(\boldsymbol{\phi}') = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle - \mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}')$$

Catastrophic forgetting happens because neither mini-batch EM nor Adam penalizes the parameters from **"changing too much"** properly.

Anemone (our proposed method):

anEMone

$$\text{maximize}_{\boldsymbol{\phi}'} \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle - \gamma\mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}')$$

The Anemone Algorithm

Anemone (our proposed method):

$$\text{maximize}_{\boldsymbol{\phi}'} \boxed{\frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_{\boldsymbol{\phi}}(\boldsymbol{x}) + \left\langle \frac{\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})}{\partial \boldsymbol{\phi}}, \boldsymbol{\phi}' - \boldsymbol{\phi} \right\rangle} - \boxed{\gamma \mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}')}$$

Can be computed by autodiff!

Can also be efficiently computed!

Lemma 1. *Assume the distributions defined by all nodes in a PC are normalized. For every $\boldsymbol{x}$, we have:*

$$(i)\ \partial \log p_{\boldsymbol{\phi}}(\boldsymbol{x})/\partial \boldsymbol{\phi} = \partial \log \tilde{p}_{\boldsymbol{\phi}}(\boldsymbol{x})/\partial \boldsymbol{\phi} - \mathtt{TD}(\boldsymbol{\phi}),$$
$$(ii)\ \mathtt{KL}_{\boldsymbol{\phi}}(\boldsymbol{\phi}') = -\langle \mathtt{TD}(\boldsymbol{\phi}), \boldsymbol{\phi}' \rangle + C,$$

where C is a constant term independent of $\boldsymbol{\phi}'$.

The Anemone Algorithm: An Intuition

Anemone can be viewed as EM with an **adaptive learning rate strategy (per parameter)**.

Full-batch EM:

$$\theta'_{n,c} = \hat{\mathtt{F}}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n,c)/Z$$

Anemone:

$$\theta'_{n,c} = \left(\theta_{n,c} + \eta \cdot \mathtt{rel}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n) \cdot \hat{\mathtt{F}}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n,c)\right)/Z$$

The adaptive learning rate.

The target

The Anemone Algorithm: An Intuition

Anemone can be viewed as EM with an **adaptive learning rate strategy (per parameter)**.

Full-batch EM:

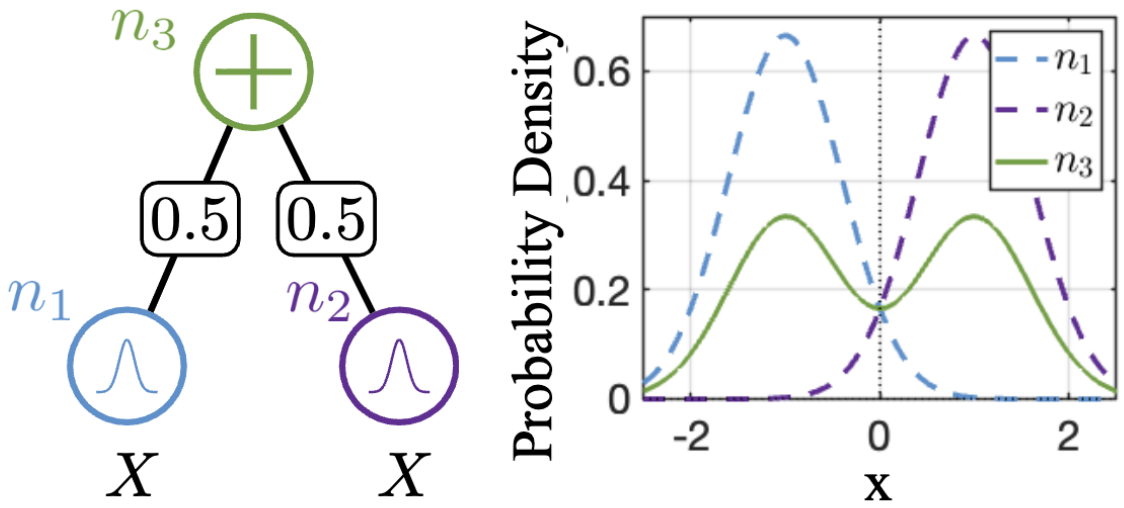
$$\theta'_{n,c} = \hat{\mathtt{F}}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n,c)/Z$$

Anemone:

$$\theta'_{n,c} = \left(\theta_{n,c} + \eta \cdot \mathtt{rel}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n) \cdot \hat{\mathtt{F}}^{\boldsymbol{x}}_{\boldsymbol{\phi}}(n,c)\right)/Z$$

The adaptive learning rate.

The target

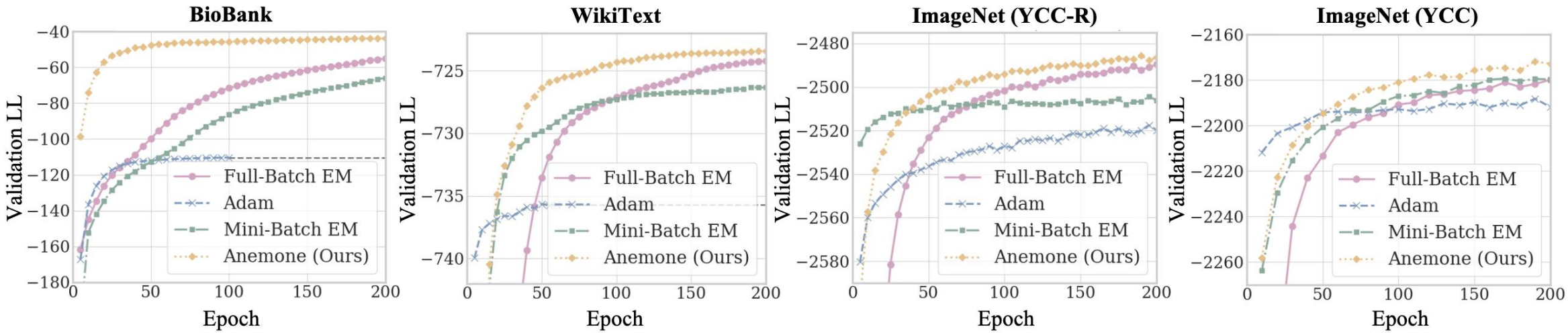


Given $X = -1.5$, the node n_1 is more "important" for producing high LL.

Anemone will implicitly assign a much **higher adaptive learning rate** to parameters of n_1.

The Anemone Algorithm: Results

- Better LL and faster convergence across benchmarks



The Anemone Algorithm: Results

- Better LL and faster convergence across benchmarks
- Robust across data types and model architectures.

Optimizer	BioBank Chr6			
	PDHCLT 512	PDHCLT 1024	HCLT 512	HCLT 1024
Full-batch EM	$45.9_{\pm 0.55}$	$45.0_{\pm 0.34}$	$55.2_{\pm 0.17}$	53.8
Adam	$113.7_{\pm 1.17}$	OOM	$102.9_{\pm 0.22}$	100.4
Mini-batch EM	$49.6_{\pm 0.42}$	$48.8_{\pm 1.41}$	$55.0_{\pm 0.56}$	52.9
Anemone (Ours)	$\mathbf{44.7}_{\pm 0.48}$	$\mathbf{42.3}_{\pm 0.31}$	$\mathbf{54.5}_{\pm 0.16}$	**52.1**

Optimizer	ImageNet32 YCC-R				
	PDHCLT 256	HCLT 512	HCLT 1024	Monarch HCLT 256	Monarch HCLT 1024
Full-batch EM	$2529_{\pm 0.48}$	$2483_{\pm 3.11}$	**2469**	2526	2468
Adam	$2555_{\pm 1.96}$	$2518_{\pm 0.29}$	OOM	–	2588
Mini-batch EM	$\mathbf{2527}_{\pm 1.52}$	$2506_{\pm 0.63}$	2470	–	2496
Anemone (Ours)	$2529_{\pm 1.87}$	$\mathbf{2480}_{\pm 2.46}$	**2469**	**2516**	**2465**

Optimizer	WikiText			
	HMM 256	HMM 512	HMM 1024	Monarch HCLT 1024
Full-batch EM	$723.3_{\pm 0.88}$	$703.2_{\pm 1.28}$	$683.4_{\pm 1.10}$	738.1
Adam	$735.5_{\pm 0.29}$	OOM	OOM	OOM
Mini-batch EM	$724.6_{\pm 0.34}$	$704.0_{\pm 1.11}$	$683.6_{\pm 1.41}$	734.6
Anemone (Ours)	$\mathbf{722.7}_{\pm 0.48}$	$\mathbf{702.5}_{\pm 0.81}$	$\mathbf{682.6}_{\pm 0.33}$	**734.0**

Thank you!
Please consider stopping by our poster #177 (today).