

Inference Compilation

Tuan Anh Le
tuananh@robots.ox.ac.uk

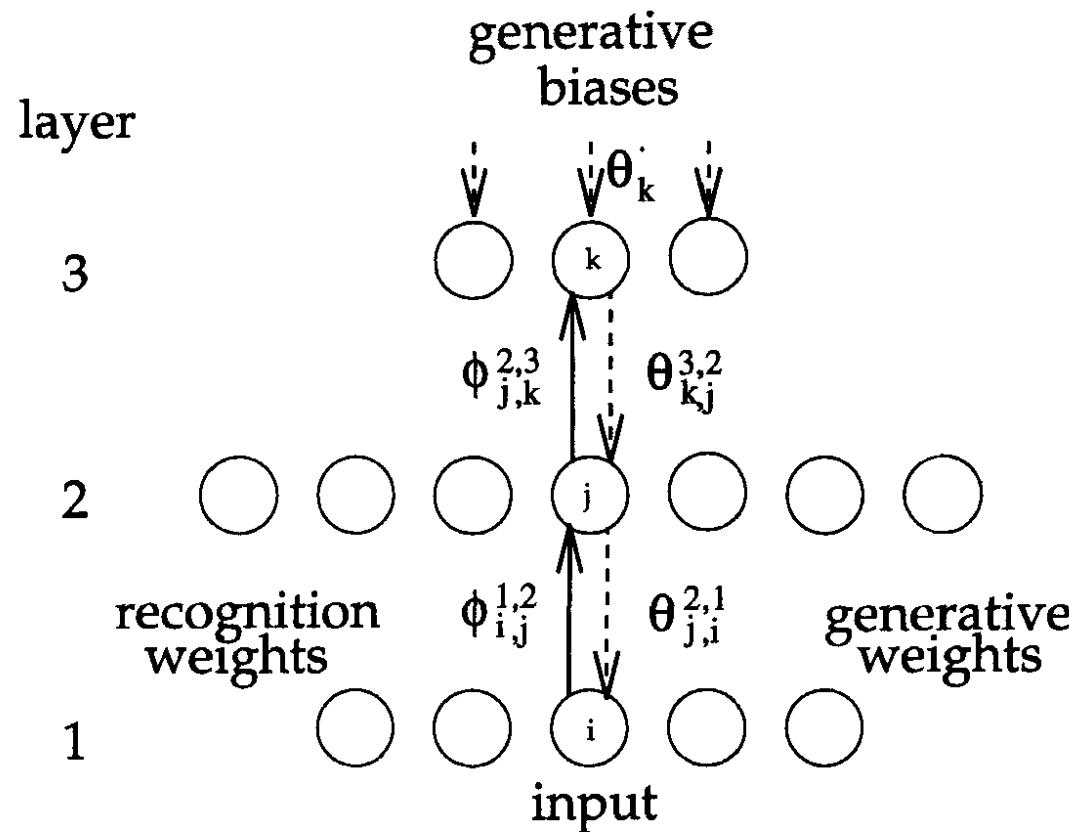
(slides mostly borrowed from Frank Wood)

*Accelerating Data Science with HPC Summer School
Lugano, September 4 – 6, 2017*

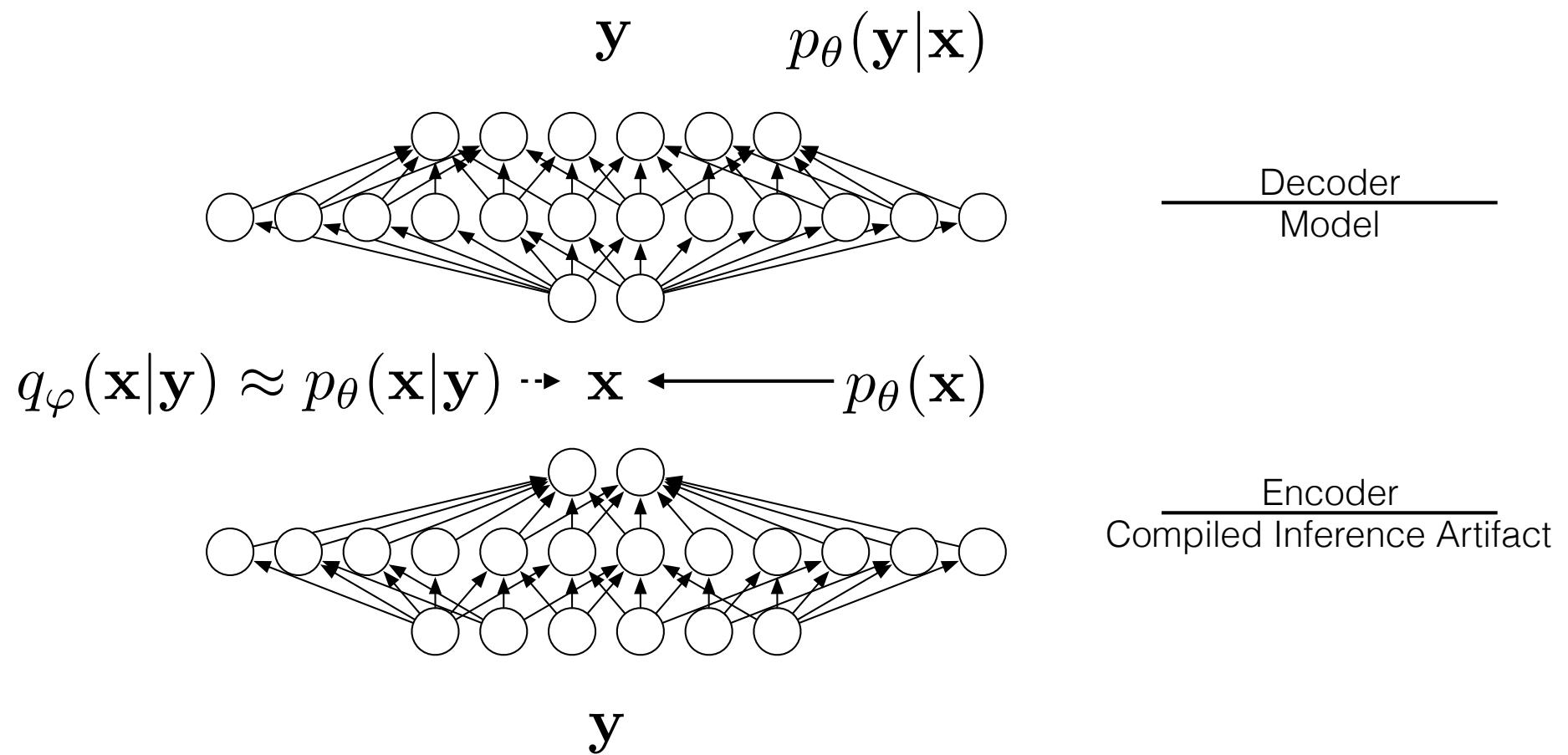
Problem

“**Bayesian inference is computationally expensive.** Even approximate, sampling-based algorithms tend to take many iterations before they produce reasonable answers. In contrast, human recognition of words, objects, and scenes is extremely rapid, often taking only a few hundred milliseconds —only enough time for a **single pass from perceptual evidence to deeper interpretation**. Yet human perception and cognition are often well-described by **probabilistic inference in complex models**. How can we reconcile the speed of recognition with the expense of coherent probabilistic inference? How can we **build systems**, for applications like robotics and medical diagnosis, **that exhibit similarly rapid performance** at challenging inference tasks?”

Helmholtz Machine



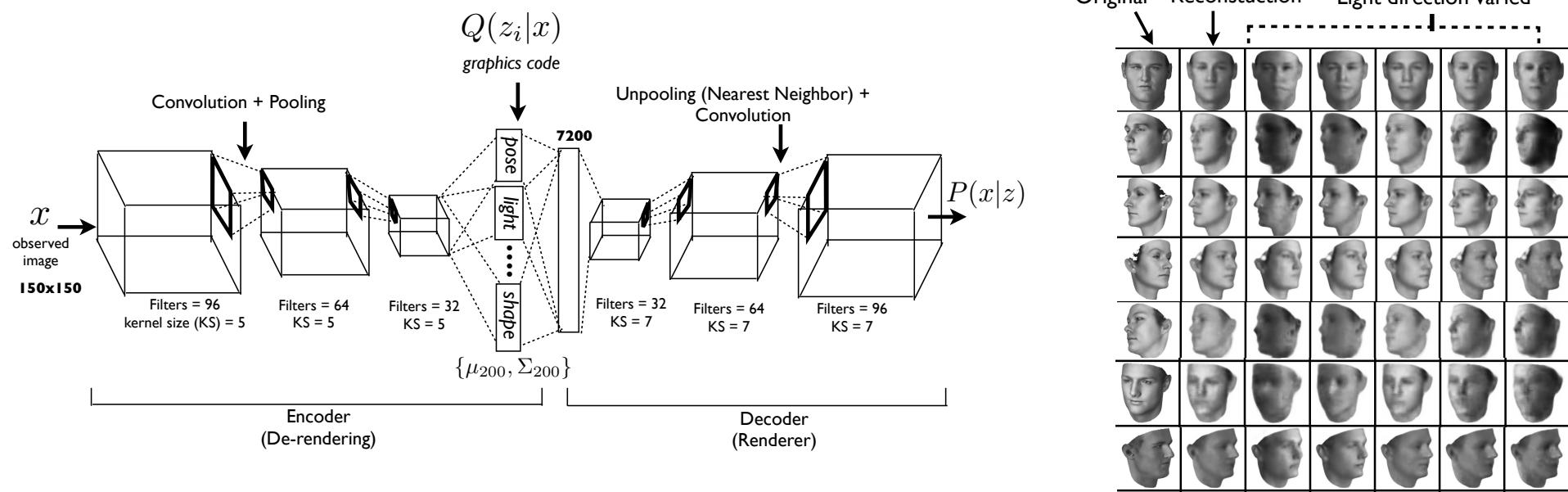
Variational Autoencoder - Neural Net View



- Helmholtz machine realization with one layer of latents + reparameterization trick, diagonal Gaussian distributions, and multi-layer encoder and decoder neural networks, variational inference

Structured Autoencoder Architectures

e.g. but many others



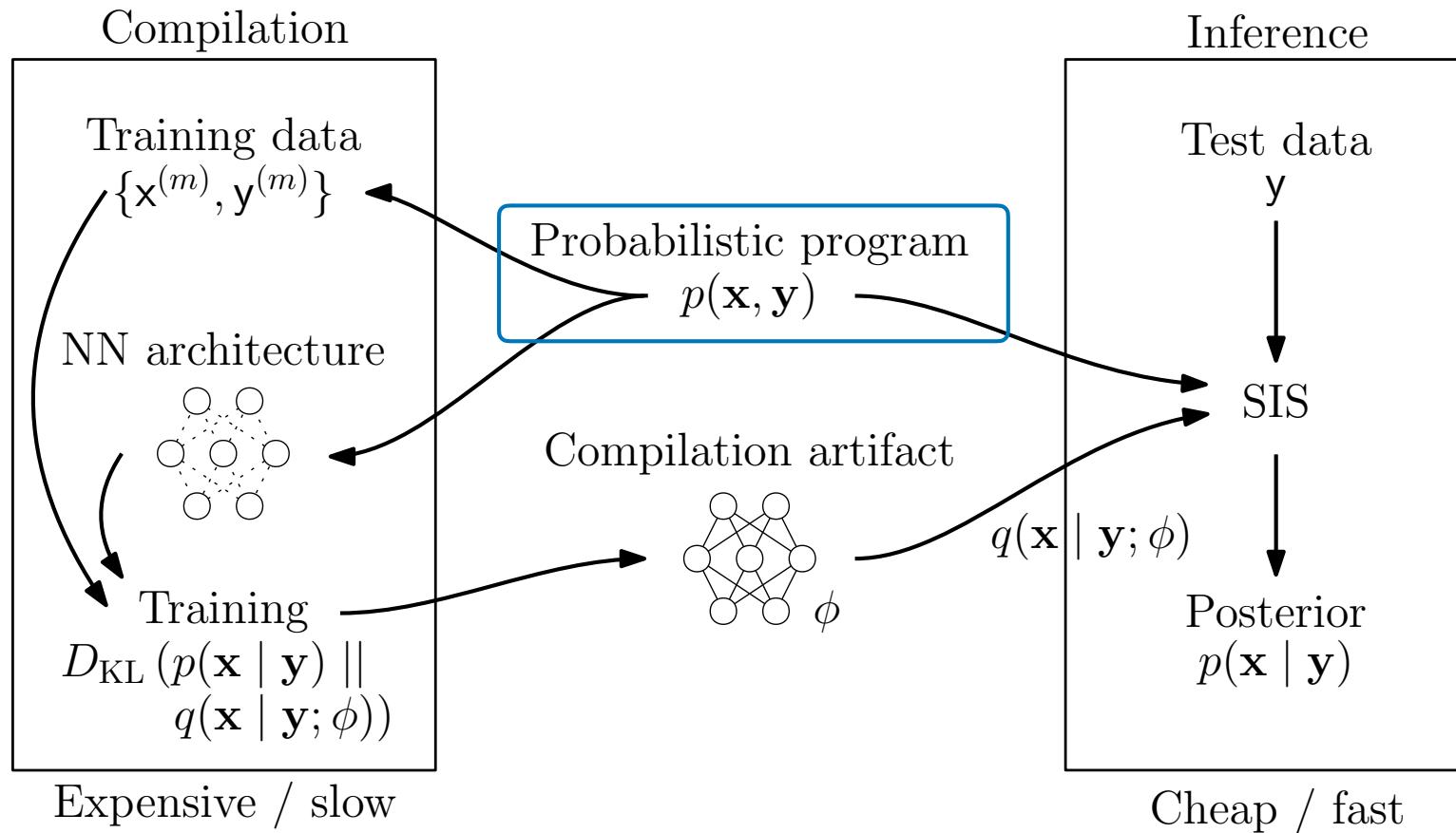
Posing and solving perception as inverse graphics requires

- Fast amortized/compiled inference
- Structured, interpretable model

Inference Compilation: Desiderata

- Denote a model and inference problem as a probabilistic program
- “Compile” for hours or days, depending on the problem, CPU/GPUs at disposal, etc.
- Get a “compilation artifact” controller that enables fast, repeated inference in the original model that is compatible with asymptotically exact inference
- Useful model-based submodules within end-to-end AI pipelines

Inference Compilation



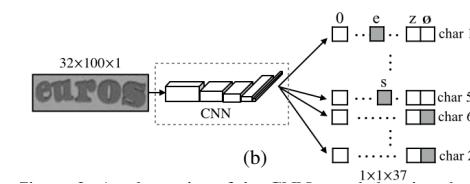
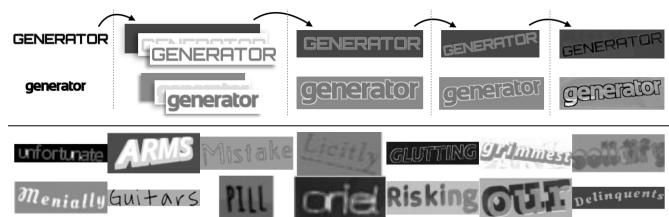
Input: an inference problem denoted in a probabilistic programming language (Anglican)

Output: a trained inference network (deep neural network “compilation artifact”)

Synthetic Data for Training Deep Networks



Goodfellow, Bulatov, Ibarz, Arnoud, Shet; Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. 2014.



Jaderberg, Simonyan, Vedaldi, Zisserman; Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition. 2014.



Gupta, Vedaldi, Zisserman; Synthetic Data for Text Localisation in Natural Images. 2016.

Example: Graphical Model Inference

Graphical model with latents \mathbf{x} and observations \mathbf{y} :

$$p(\mathbf{x}, \mathbf{y}) = \prod_{n=1}^N p(x_n | \text{PA}(x_n)) \prod_{m=1}^M p(y_m | \text{PA}(y_m))$$

Goal: Efficiently compute (approximation of):

- posterior $p(\mathbf{x}|\mathbf{y})$
- expectation under the posterior $\mathbb{E}_{p(\mathbf{x}|\mathbf{y})}[f(\mathbf{x})]$
- marginal likelihood $p(\mathbf{y})$

Importance sampling

For $k = 1, \dots, K$:

Propose:

$$\mathbf{x}_k \sim q(\mathbf{x})$$

Performance
depends on quality
of proposal

Weight:

$$w_k = \frac{p(\mathbf{x}_k, \mathbf{y})}{q(\mathbf{x}_k)}$$

Normalize:

$$W_k = \frac{w_k}{\sum_{\ell=1}^K w_\ell}$$

$$p(\mathbf{x}|\mathbf{y}) \approx \sum_{k=1}^K W_k \delta_{\mathbf{x}_k}(\mathbf{x})$$

Approximate:

$$\mathbb{E}_{p(\mathbf{x}|\mathbf{y})}[f(\mathbf{x})] \approx \sum_{k=1}^K W_k f(\mathbf{x}_k)$$

$$p(\mathbf{y}) \approx \sum_{k=1}^K w_k$$

Importance sampling

For $k = 1, \dots, K$:

Propose:

$$\mathbf{x}_k \sim q(\mathbf{x} | \lambda)$$

Performance
depends on quality
of proposal

Weight:

$$w_k = \frac{p(\mathbf{x}_k, \mathbf{y})}{q(\mathbf{x}_k | \lambda)}$$

Adaptation

Normalize:

$$W_k = \frac{w_k}{\sum_{\ell=1}^K w_\ell}$$

$$p(\mathbf{x} | \mathbf{y}) \approx \sum_{k=1}^K W_k \delta_{\mathbf{x}_k}(\mathbf{x})$$

Approximate:

$$\mathbb{E}_{p(\mathbf{x} | \mathbf{y})}[f(\mathbf{x})] \approx \sum_{k=1}^K W_k f(\mathbf{x}_k)$$

$$p(\mathbf{y}) \approx \sum_{k=1}^K w_k$$

One Notion of Optimal Proposal

Learning an importance sampling proposal for a single dataset

Target density $p(\mathbf{x}|\mathbf{y})$, approximating family $q(\mathbf{x}|\lambda)$

Single dataset \mathbf{y} : $\underset{\lambda}{\operatorname{argmin}} \text{KL}(p(\mathbf{x}|\mathbf{y})||q(\mathbf{x}|\lambda))$ fit λ to learn an importance sampling proposal

$\underset{\lambda}{\operatorname{argmin}} \text{KL}(q(\mathbf{x}|\lambda)||p(\mathbf{x}|\mathbf{y}))$ Note: opposite KL in VB/VAE

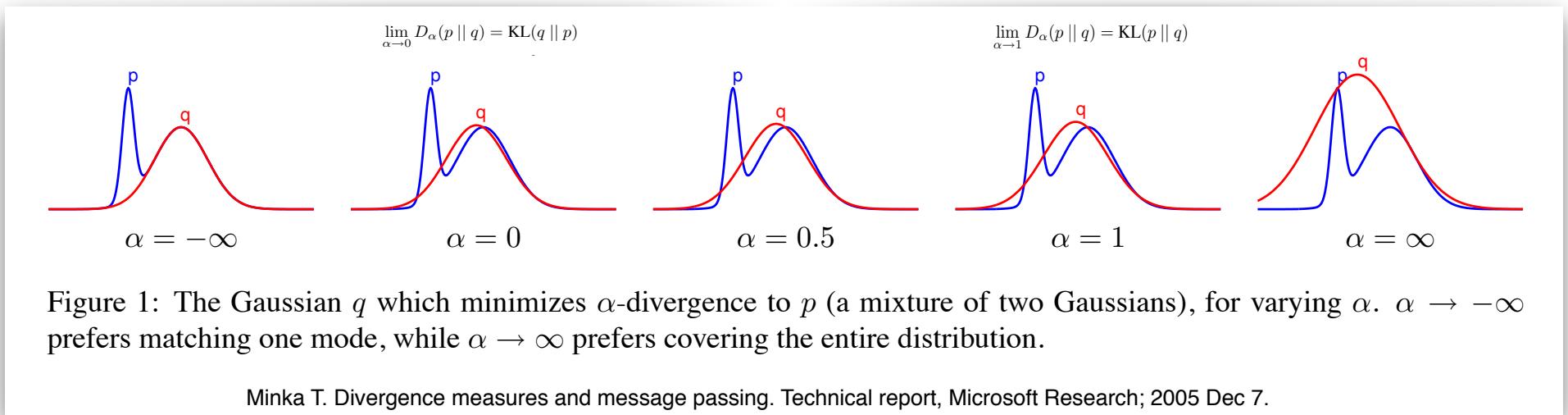


Figure 1: The Gaussian q which minimizes α -divergence to p (a mixture of two Gaussians), for varying α . $\alpha \rightarrow -\infty$ prefers matching one mode, while $\alpha \rightarrow \infty$ prefers covering the entire distribution.

Minka T. Divergence measures and message passing. Technical report, Microsoft Research; 2005 Dec 7.

Compiling Away Runtime Costs of Inference

Learn to invert the generative model, before seeing data

Averaging over
all possible datasets: $\lambda = \varphi(\eta, \mathbf{y})$

$$\underset{\eta}{\operatorname{argmin}} \mathbb{E}_{p(\mathbf{y})} [\text{KL}(p(\mathbf{x}|\mathbf{y})||q(\mathbf{x}|\varphi(\eta, \mathbf{y})))]$$

New objective function,
upper-level parameters: $\mathcal{J}(\eta) = \int p(\mathbf{y}) \text{KL}(p(\mathbf{x}|\mathbf{y})||q(\mathbf{x}|\lambda)) d\mathbf{y}$

$$\begin{aligned} &= \int p(\mathbf{y}) \int p(\mathbf{x}|\mathbf{y}) \log \left[\frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\varphi(\eta, \mathbf{y}))} \right] d\mathbf{x} d\mathbf{y} \\ &= \int \int p(\mathbf{x}, \mathbf{y}) [\log p(\mathbf{x}|\mathbf{y}) - \log q(\mathbf{x}|\varphi(\eta, \mathbf{y}))] d\mathbf{x} d\mathbf{y} \\ &= \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} [-\log q(\mathbf{x}|\varphi(\eta, \mathbf{y}))] + \text{const.} \end{aligned}$$

Compiling Away Runtime Costs of Inference

Learn to invert the generative model, before seeing data

Averaging over
all possible datasets: $\lambda = \varphi(\eta, \mathbf{y})$

$$\underset{\eta}{\operatorname{argmin}} \mathbb{E}_{p(\mathbf{y})} [\text{KL}(p(\mathbf{x}|\mathbf{y})||q(\mathbf{x}|\varphi(\eta, \mathbf{y})))]$$

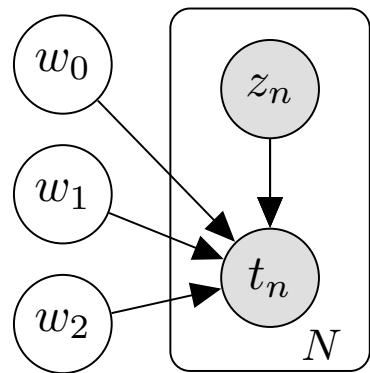
New objective function,
upper-level parameters: $\mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} [-\log q(\mathbf{x}|\varphi(\eta, \mathbf{y}))] + \text{const.}$

Can train using SGD
with samples from the
joint:

$$\nabla_{\eta} \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} [-\nabla_{\eta} \log q(\mathbf{x}|\varphi(\eta, \mathbf{y}))]$$

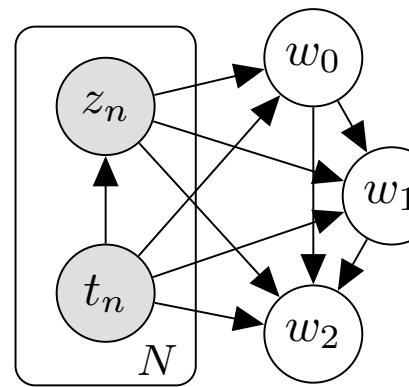
Example: Non-Conjugate Robust Polynomial Regression

Generative model



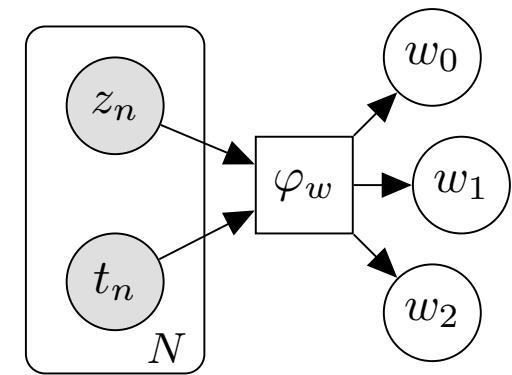
automatic

Inverted model



hand

Neural net proposal



$$w_d \sim \text{Laplace}(0, 10^{1-d})$$

for $d = 0, 1, 2$;

$$q(w_{0:2}|z_{1:N}, t_{1:N})$$

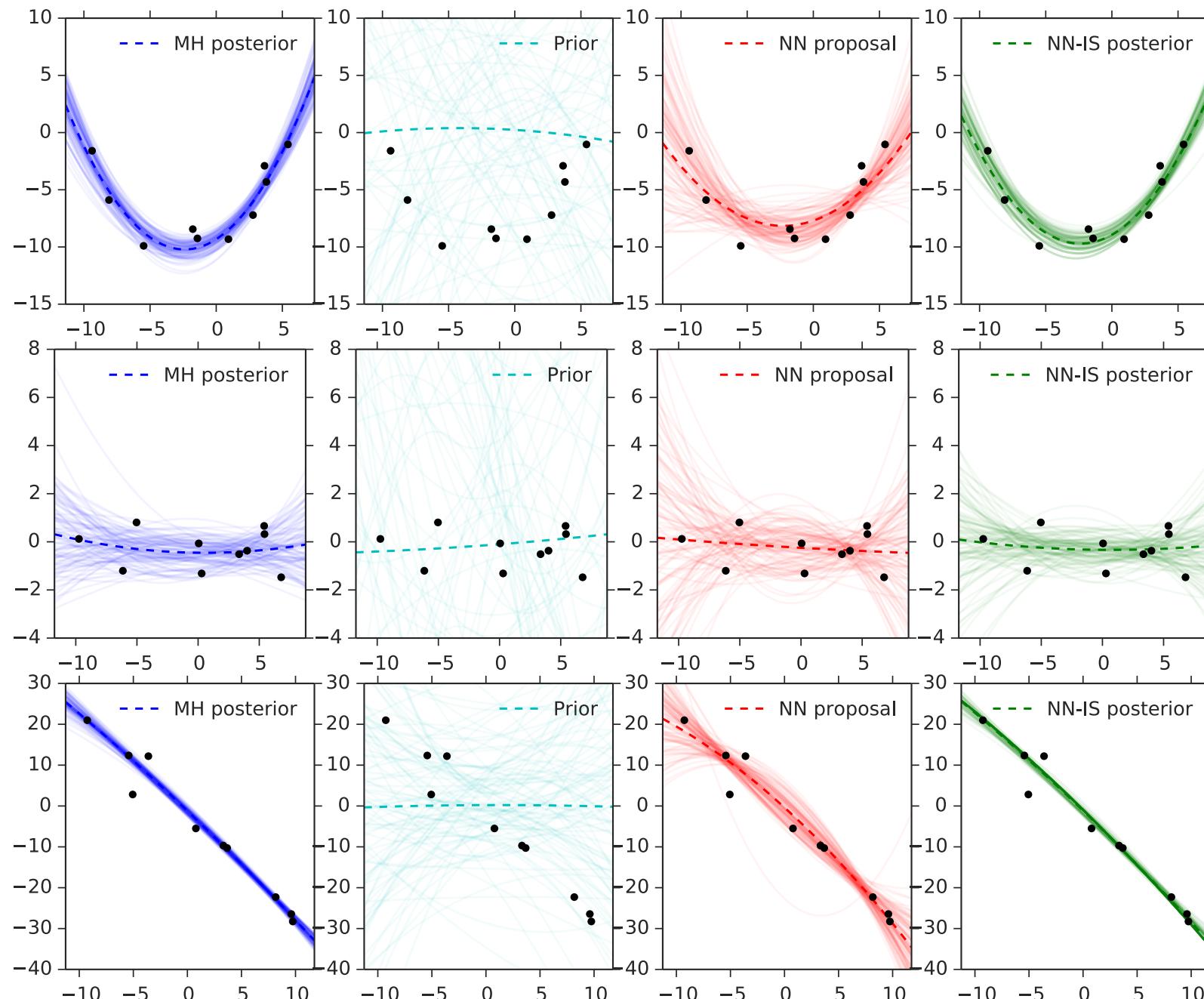
$$t_n \sim t_\nu(w_0 + w_1 z_n + w_2 z_n^2, \epsilon^2)$$

for $n = 1, \dots, N$

$\nu = 4, \epsilon = 1$, and $z_n \in (-10, 10)$

- Two layer MLP
- 200 units
- 3-component MOG for each output

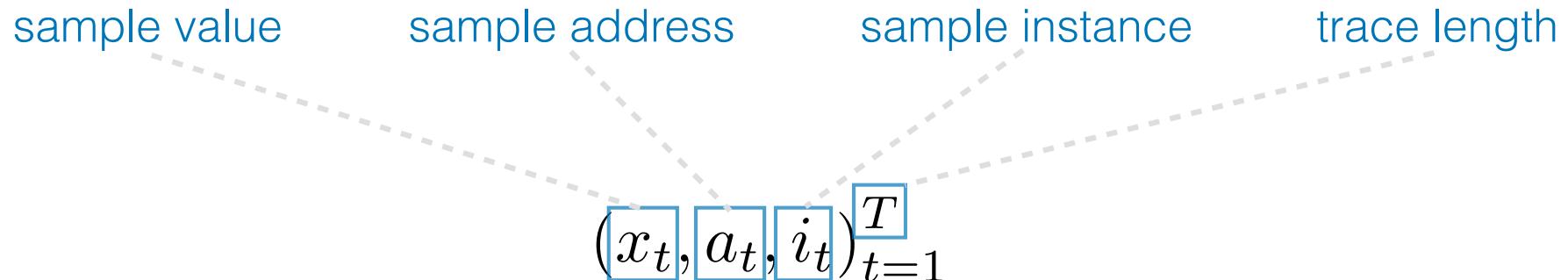
Example: Non-Conjugate Robust Polynomial Regression



Inference Compilation for Higher-Order Languages

Higher-Order Probabilistic Programs

Execution trace of a probabilistic program



Joint probability density

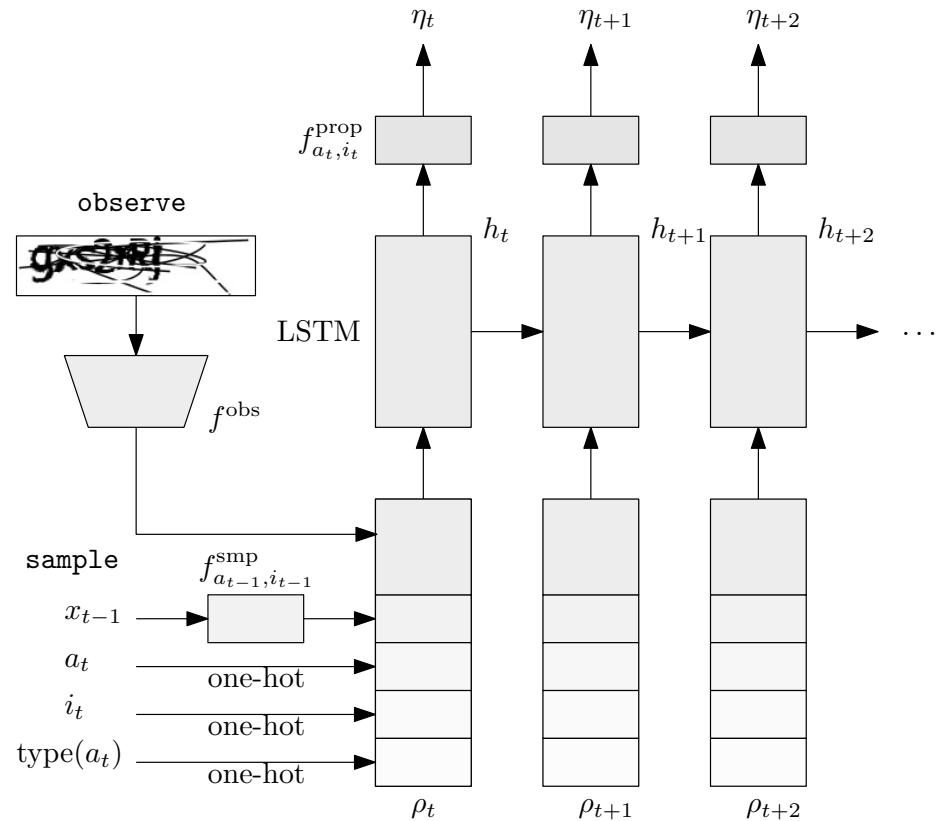
$$p(\mathbf{x}, \mathbf{y}) := \prod_{t=1}^T f_{a_t}(x_t | x_{1:t-1}) \prod_{n=1}^N g_n(y_n | x_{1:\tau(n)})$$

conditional priors fixed observations likelihoods

sample value nth observation
samples before x_t samples before nth observation

Proposal Neural Network Architecture

- A non-program-specific stacked LSTM core
- Proposal layers, observation and sample embeddings specified by the program
- **Compilation:** New layers are created and attached on-the-fly on first encounter with address-instance pairs
- **Inference:** The network is configured on-the-fly as the probabilistic program execution trace unravels



Example: Captcha

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))sample addresses  
a_1
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))a_2
observe(render(letters), observed_captcha)
return letters
```

Importance sampling

For $k = 1, \dots, K$:

Propose: $\mathbf{x}_k \sim q(\mathbf{x}|\varphi(\eta, \mathbf{y}))$

Already trained

Weight: $w_k = \frac{p(\mathbf{x}_k, \mathbf{y})}{q(\mathbf{x}_k | \lambda)}$

Normalize: $W_k = \frac{w_k}{\sum_{\ell=1}^K w_\ell}$

$$p(\mathbf{x}|\mathbf{y}) \approx \sum_{k=1}^K W_k \delta_{\mathbf{x}_k}(\mathbf{x})$$

Approximate: $\mathbb{E}_{p(\mathbf{x}|\mathbf{y})}[f(\mathbf{x})] \approx \sum_{k=1}^K W_k f(\mathbf{x}_k)$

$$p(\mathbf{y}) \approx \sum_{k=1}^K w_k$$

Example: Captcha (Sampling from proposal)

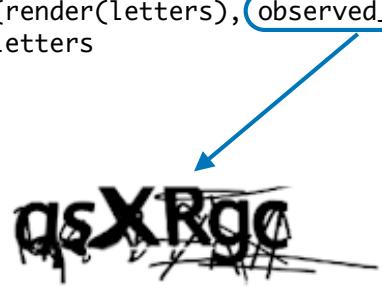
```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))

observe(render(letters), observed_captcha)
return letters
```

Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", "z", "A", "Z")))

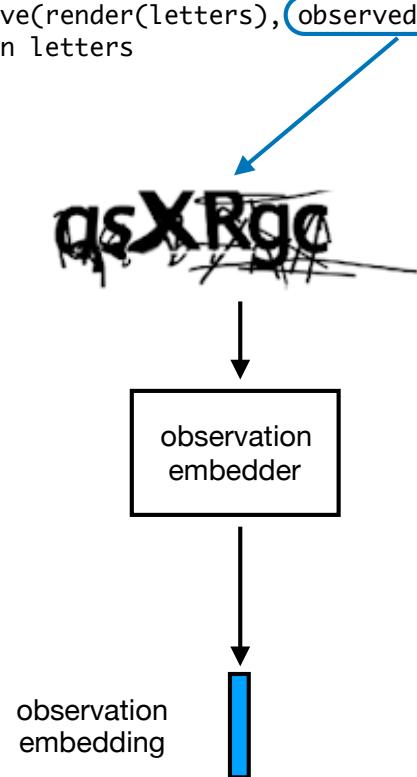
observe(render(letters), observed_captcha)
return letters
```



Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))

observe(render(letters), observed_captcha)
return letters
```

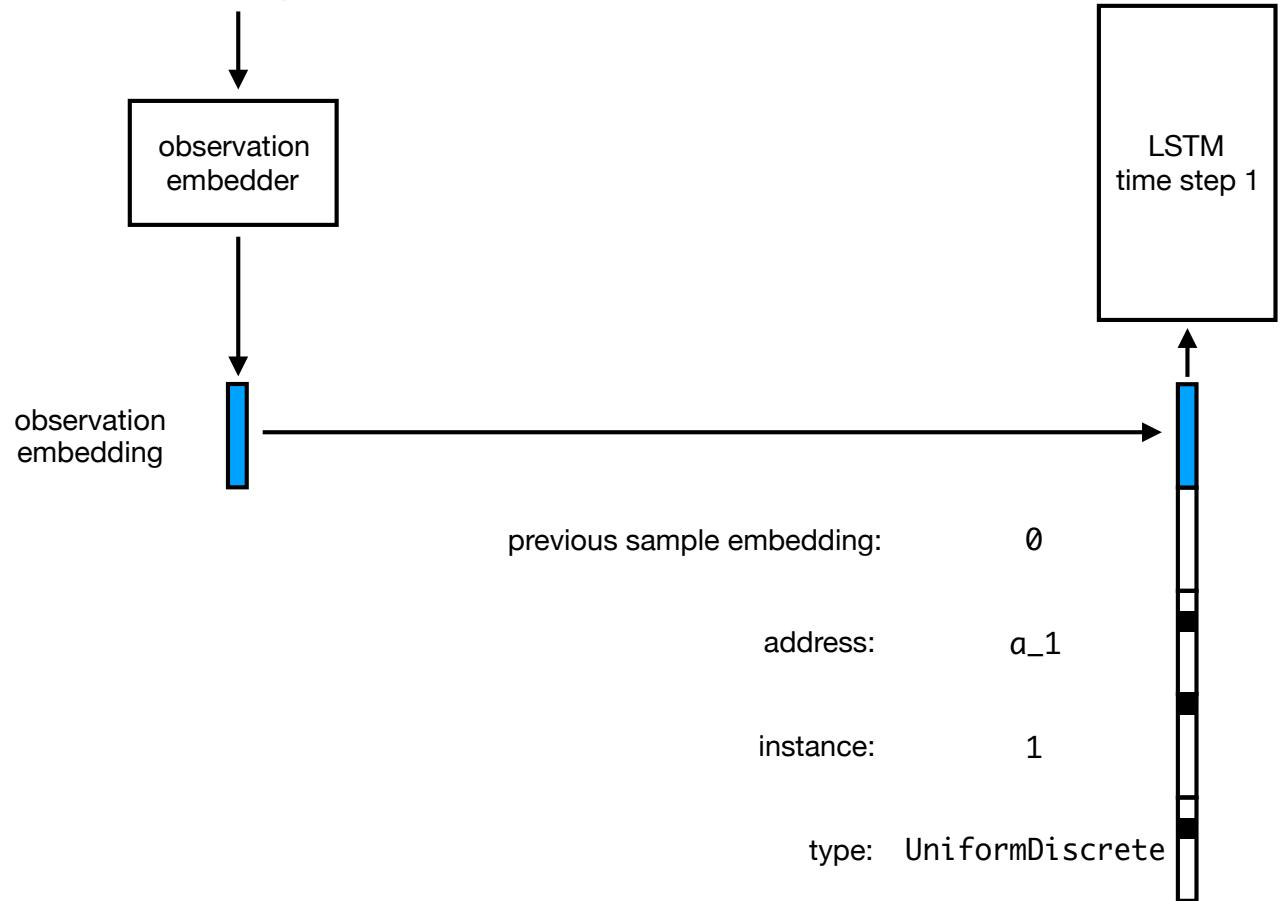


Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))

observe(render(letters), observed_captcha)
return letters
```

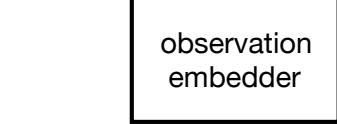
qsXRgc



Example: Captcha (Sampling from proposal)

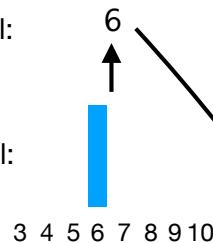
```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters): // i = 0
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```

qsXRgc



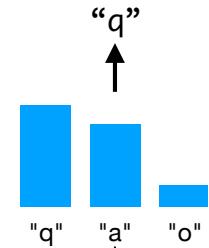
observation embedding

sample from the proposal:



Proposal layer

LSTM time step 1



Proposal layer

LSTM time step 2

previous sample embedding:

0

address:

a_1

instance:

1

type: UniformDiscrete

a_2

1

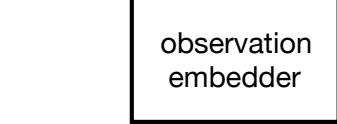
Uniform



Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```

qsXRgc



observation embedding

previous sample embedding:

address:

a_2

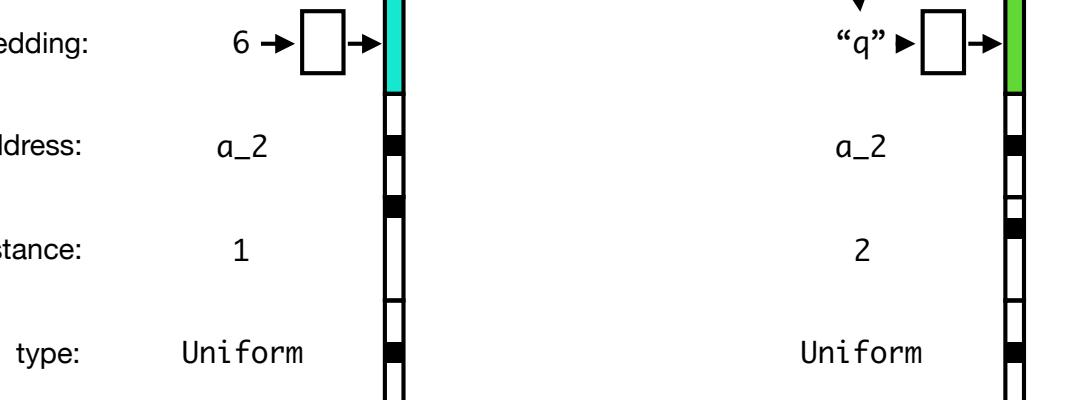
instance:

1

type:

Uniform

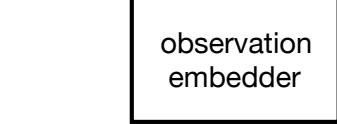
sample from the proposal:
proposal: "q" "a" "o"
Proposal layer
LSTM time step 2
...
LSTM time step 3
"q" "s"
Proposal layer
"s" "S"



Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```

qsXRgc



observation embedding

previous sample embedding:

address:

a_2

instance:

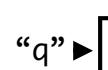
2

type:

Uniform

"q" ►

►



address:

a_2

instance:

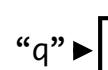
3

type:

Uniform

"s" ►

►



address:

a_2

instance:

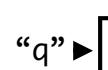
3

type:

Uniform

"X" ►

►



address:

"X"

instance:

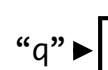
"X"

type:

Uniform

"s" ►

►



address:

"s"

instance:

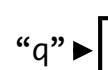
2

type:

Uniform

"S" ►

►



address:

"S"

instance:

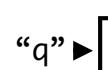
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

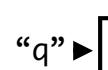
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

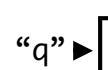
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

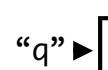
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

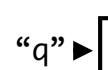
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

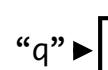
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

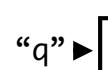
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

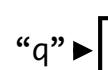
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

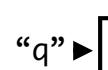
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

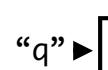
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

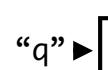
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

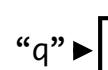
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

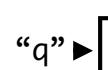
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

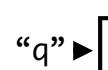
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

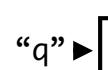
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

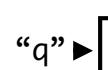
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

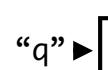
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

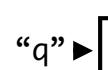
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

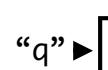
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

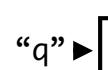
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

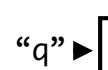
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

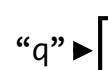
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

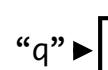
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

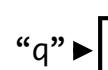
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

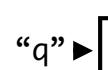
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

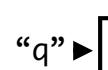
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

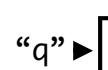
2

type:

Uniform

"s" ►

►



address:

"s"

instance:

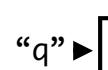
2

type:

Uniform

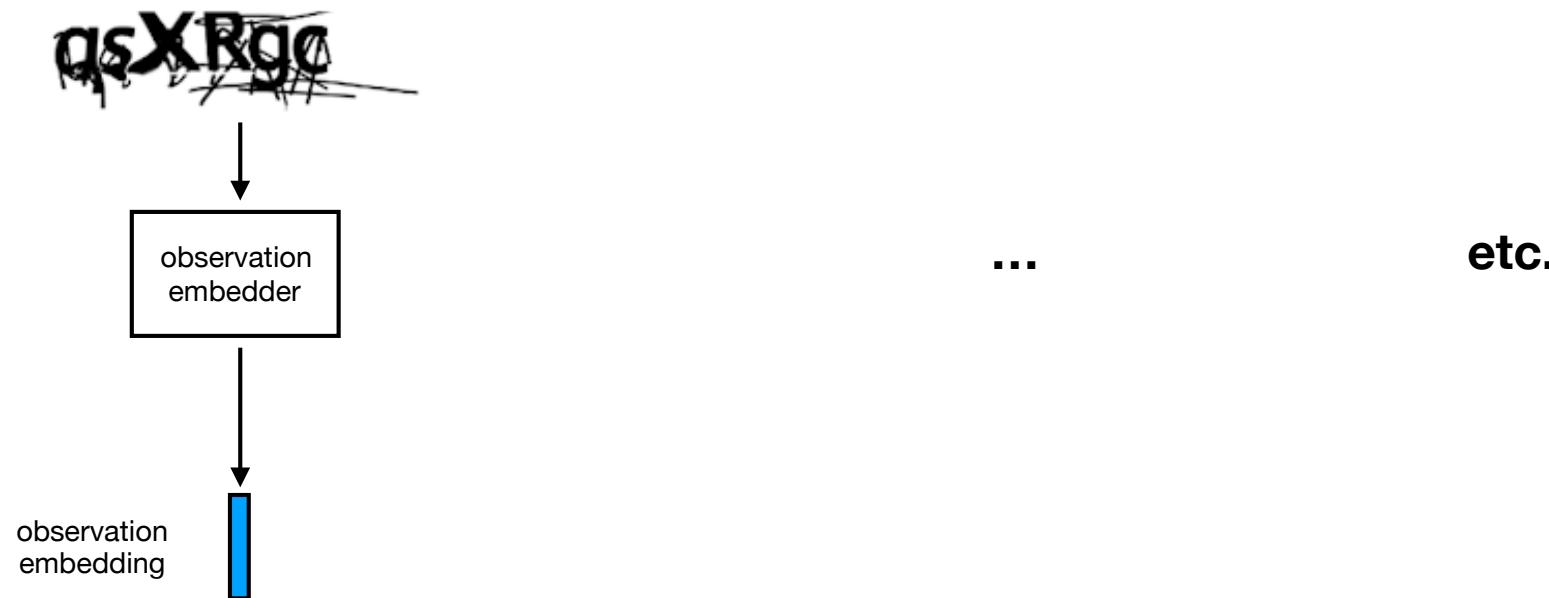
"s" ►

►



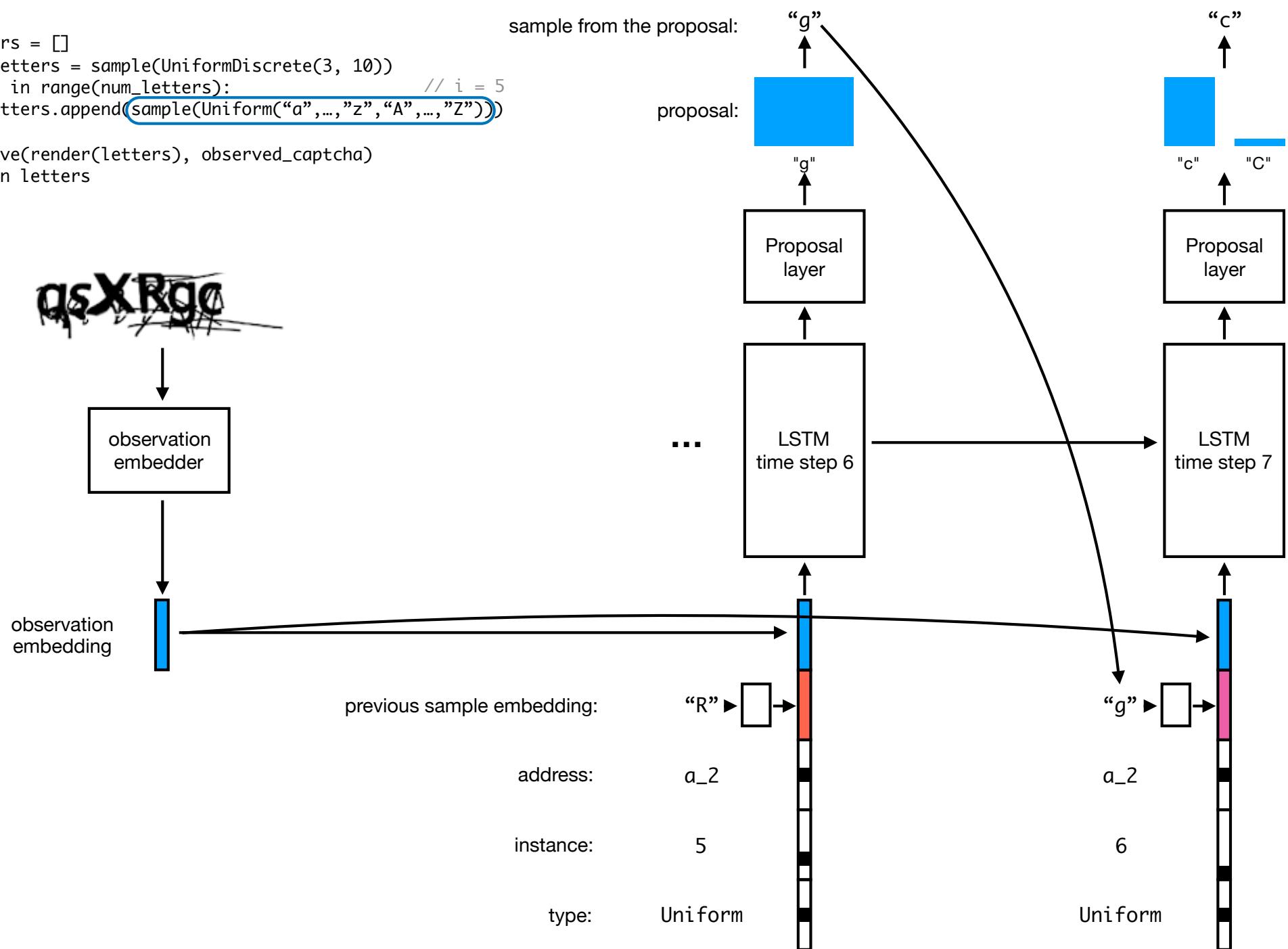
Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters):           // i = ...
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```



Example: Captcha (Sampling from proposal)

```
letters = []
num_letters = sample(UniformDiscrete(3, 10))
for i in range(num_letters): // i = 5
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```



Importance sampling

For $k = 1, \dots, K$:

Propose: $\mathbf{x}_k \sim q(\mathbf{x} | \varphi(\eta, \mathbf{y}))$

Weight: $w_k = \frac{p(\mathbf{x}_k, \mathbf{y})}{q(\mathbf{x}_k | \lambda)}$

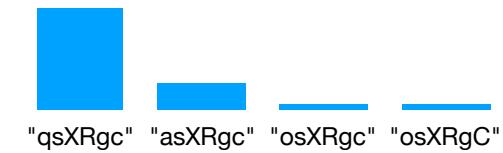
Normalize: $W_k = \frac{w_k}{\sum_{\ell=1}^K w_\ell}$

~~qsXRgc~~ $p(\mathbf{x} | \mathbf{y}) \approx \sum_{k=1}^K W_k \delta_{\mathbf{x}_k}(\mathbf{x})$

Approximate: $\mathbb{E}_{p(\mathbf{x} | \mathbf{y})}[f(\mathbf{x})] \approx \sum_{k=1}^K W_k f(\mathbf{x}_k)$
 $p(\mathbf{y}) \approx \sum_{k=1}^K w_k$

(6, "q", "s", "X", "R", "g", "c")

Can have different length depending
on num_letters



Compilation: Infinite Training Data

We generate minibatches of program traces

$$\mathcal{D}_{\text{train}} = \left\{ \left(\begin{pmatrix} x_t^{(m)}, a_t^{(m)}, i_t^{(m)} \end{pmatrix}_{t=1}^{T^{(m)}}, \left(y_n^{(m)} \right)_{n=1}^N \right)_{m=1}^M \right\}$$

from the model $p(\mathbf{x}, \mathbf{y})$.

This fixes the **inputs**, **outputs** and **length** of the LSTM.

Train using stochastic gradient descent.

Implementation

```
;; CAPTCHA Query
(with-primitive-procedures [render abc-dist repeatedly]
  (defquery captcha [baseline-image]
    (let [;; Render parameters
          num-letters (sample "numletters" (uniform-discrete 6 8))
          font-size (sample "fontsize" (uniform-discrete 38 44))
          kerning (sample "kerning" (uniform-discrete -2 2))
          letter-ids (repeatedly num-letters #(sample "letterid"
(uniform-discrete 0 (count letter-dict))))
          letters (apply str (map (partial nth letter-dict) letter-ids))]

      ;; Render image
      rendered-image (render letters font-size kerning))

    ;; ABC-style observe
    (observe (abc-dist rendered-image abc-sigma) baseline-image)

    letters)))
```

The screenshot shows two terminal windows. The top window is titled 'Oxford Compiled Inference 0.8.1' and displays the command line arguments: '--cuda'. It lists various artifacts being compiled, such as 'Polymorphing artifact, new address : numletters', 'Polymorphing artifact, new instance : 1', and 'Polymorphing artifact, new proposal type : discreteminmax'. The bottom window is also titled 'Terminal - gunes@Prec5510:~/compiled-inference/code/torch' and shows a table of training results. The table has columns for 'Train. time', 'Trace', 'Training loss', 'Last valid. loss', 'Best val. loss', and 'T.since best'. The data in the table includes various numerical values and error codes.

Train. time	Trace	Training loss	Last valid. loss	Best val. loss	T.since best
0d 00:00:05		+3.294497e+01 ✓	+3.523340e+01	+3.523340e+01	0d 00:00:05
0d 00:00:05		+3.729427e+01 ✓	+3.523340e+01	+3.523340e+01	0d 00:00:05
0d 00:00:07		+3.046592e+01 ✓	+3.523340e+01	+3.523340e+01	0d 00:00:07
0d 00:00:08		+3.616813e+01	+3.523340e+01	+3.523340e+01	0d 00:00:08
0d 00:00:10		+2.984722e+01 ✓	+3.523340e+01	+3.523340e+01	0d 00:00:10
0d 00:00:11		+3.411166e+01	+3.523340e+01	+3.523340e+01	0d 00:00:11
0d 00:00:13		+3.477	+3.523340e+01	+3.523340e+01	0d 00:00:13
0d 00:00:14		+3.465689e+01	+3.193555e+01 ✓	+3.193555e+01	0d 00:00:00
0d 00:00:17		+3.011164e+01	+3.193555e+01 ✓	+3.193555e+01	0d 00:00:01
0d 00:00:17		+3.387039e+01	+3.193555e+01 ✓	+3.193555e+01	0d 00:00:01
0d 00:00:20		+2.947744e+01 ✓	+3.193555e+01	+3.193555e+01	0d 00:00:04
0d 00:00:20		+3.376682e+01	+3.193555e+01	+3.193555e+01	0d 00:00:04
0d 00:00:22		+2.942153e+01 ✓	+3.193555e+01	+3.193555e+01	0d 00:00:06
0d 00:00:23		+3.308034e+01	+3.193555e+01	+3.193555e+01	0d 00:00:07
0d 00:00:25		+2.689663e+01 ✓	+3.193555e+01	+3.193555e+01	0d 00:00:09
0d 00:00:26		+3.722147e+01	+3.114152e+01	+3.114152e+01	0d 00:00:00
0d 00:00:29		+2.013007e+01	+3.114152e+01	+3.114152e+01	0d 00:00:00
0d 00:00:30		+1.152	+3.114152e+01	+3.114152e+01	0d 00:00:01
0d 00:00:32		+2.994558e+01	+3.114152e+01	+3.114152e+01	0d 00:00:03
0d 00:00:33		+3.106666e+01	+3.114152e+01	+3.114152e+01	0d 00:00:04
0d 00:00:35		+2.993464e+01	+3.114152e+01	+3.114152e+01	0d 00:00:06
0d 00:00:36		+3.191970e+01	+3.114152e+01	+3.114152e+01	0d 00:00:07
0d 00:00:38		+1.471	+2.915486e+01	+3.114152e+01	0d 00:00:09
0d 00:00:38		+3.715972e+01	+3.095194e+01	+3.095194e+01	0d 00:00:08
0d 00:00:42		+2.856380e+01	+3.095194e+01	+3.095194e+01	0d 00:00:01
0d 00:00:43		+1.664	+3.093398e+01	+3.095194e+01	0d 00:00:02
0d 00:00:45		+1.719	+2.995583e+01	+3.095194e+01	0d 00:00:04
0d 00:00:45		+1.792	+3.293678e+01	+3.095194e+01	0d 00:00:04
0d 00:00:48		1,856	+2.874953e+01 ✓	+3.095194e+01	0d 00:00:07
0d 00:00:48		1,920	+3.326929e+01	+3.095194e+01	0d 00:00:07
0d 00:00:51		1,991	+2.905301e+01	+3.095194e+01	0d 00:00:10
0d 00:00:51		2,048	+3.372107e+01	+3.072823e+01	0d 00:00:00
0d 00:00:55		2,112	+2.857211e+01 ✓	+3.072823e+01	0d 00:00:01
0d 00:00:55		2,176	+3.308041e+01	+3.072823e+01	0d 00:00:01
0d 00:00:57		2,238	+2.856231e+01 ✓	+3.072823e+01	0d 00:00:03
0d 00:00:58		2,384	+3.203077e+01	+3.072823e+01	0d 00:00:04
0d 00:01:00		2,376	+2.861978e+01	+3.072823e+01	0d 00:00:06
0d 00:01:01		2,432	+3.308773e+01	+3.072823e+01	0d 00:00:07
0d 00:01:03		2,496	+2.855996e+01 ✓	+3.072823e+01	0d 00:00:09
0d 00:01:04		2,560	+3.708694e+01	+3.071229e+01	0d 00:00:09

- Anglican* + PyTorch + ZeroMQ
- Compilation and inference on GPUs
- Models and artifacts can live on separate machines and distributed
- Code & tutorials on GitHub soon

* <http://www.robots.ox.ac.uk/~fwood/anglican/>

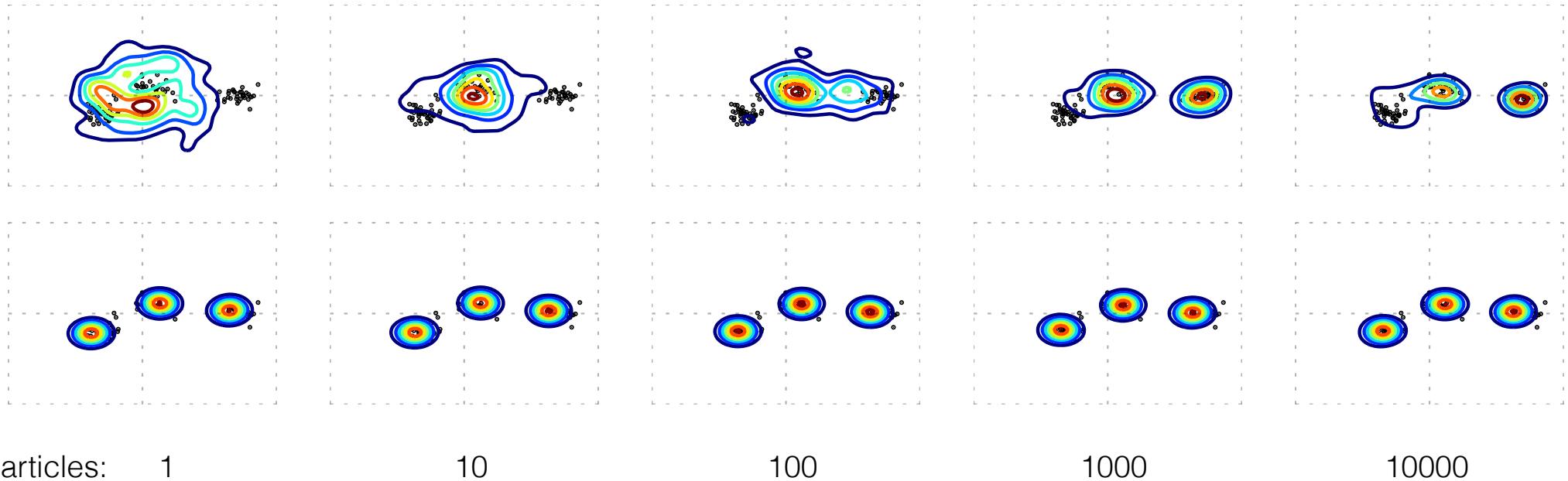
Examples

- Rich, structured, *interpretable* models
 - Open-universe Gaussian Mixture Model
 - Inverse graphics, e.g., Captcha generative model
 - Physics and engineering simulations — ongoing work

Open-Universe Gaussian Mixture Model

```
1: procedure GMM
2:    $K \sim p(K|\cdot)$                                  $\triangleright$  sample number of clusters
3:   for  $k = 1, \dots, K$  do
4:      $\mu_k, \Sigma_k \sim p(\mu_k, \Sigma_k | \cdot)$        $\triangleright$  sample cluster parameters
5:      $\pi \leftarrow \text{uniform}(1, K)$ 
6:     for  $n = 1, \dots, N$  do
7:        $z_n \sim p(z_n | \pi)$                            $\triangleright$  sample class label
8:        $y_n \sim p(y_n | z_n = k, \mu_k, \Sigma_k)$        $\triangleright$  sample or observe data
return  $\{\mu_k, \Sigma_k\}_{k=1}^K, K$ 
```

GMM Inference



Particles: 1

10

100

1000

10000

Kernel density estimation of the distribution over maximum a-posteriori values of the means $\{\max_{\mu_k} p(\mu_k | \mathbf{y})\}_{k=1}^3$ over 50 independent runs

Top: Sequential Monte Carlo
Bottom: Inference Compilation

Effect of Training Duration

Gaussian Mixtures / Counting & Localization

Observed points
(From a Fast R-CNN detector)

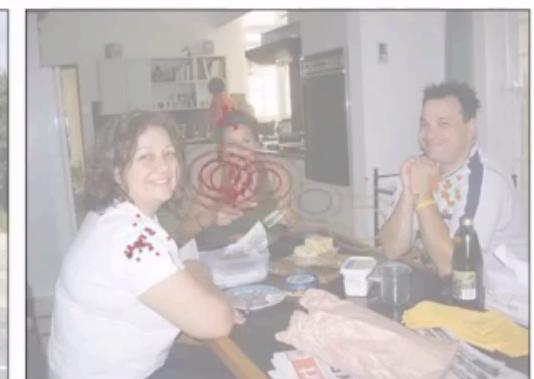
Inference

10^7



Training traces

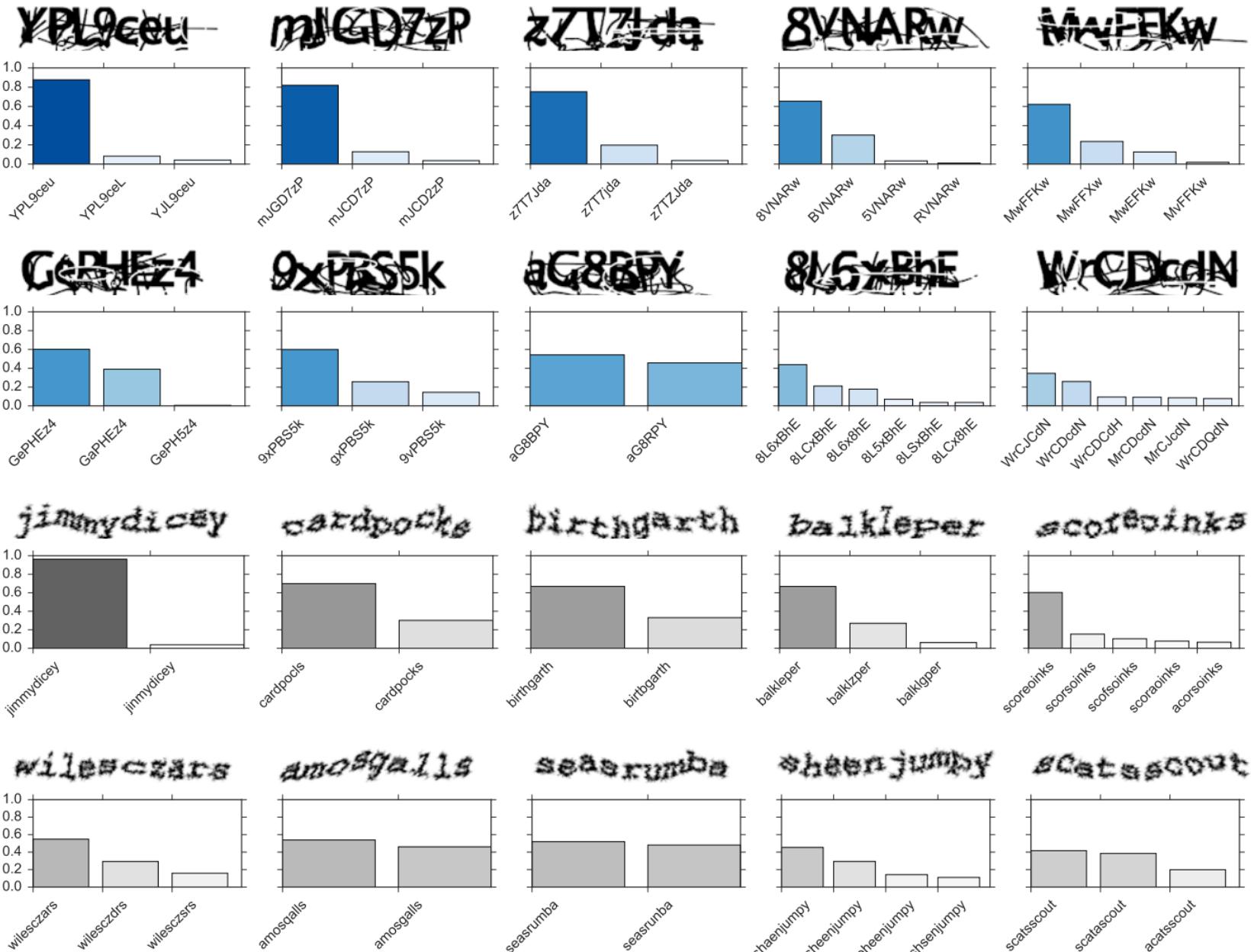
10^4



Captcha Generative Model

```
1: procedure CAPTCHA
2:    $\nu \sim p(\nu)$                                  $\triangleright$  sample number of letters
3:    $\kappa \sim p(\kappa)$                              $\triangleright$  sample kerning value
4:   Generate letters:
5:    $\Lambda \leftarrow \{\}$ 
6:   for  $i = 1, \dots, \nu$  do
7:      $\lambda \sim p(\lambda)$                            $\triangleright$  sample letter ID
8:      $\Lambda \leftarrow \text{append}(\Lambda, \lambda)$ 
9:   Render:
10:     $\gamma \leftarrow \text{render}(\Lambda, \kappa)$ 
11:     $\pi \sim p(\pi)$                                  $\triangleright$  sample noise parameters
12:     $\gamma \leftarrow \text{noise}(\gamma, \pi)$ 
      return  $\gamma$ 
```

Captcha Breaking Results



Captcha Breaking Results

Type	Baidu (2011)	Baidu (2013)	eBay	Yahoo	reCaptcha	Wikipedia	Facebook
Our method	RR 99.8% BT 72 ms	99.9% 67 ms	99.2% 122 ms	98.4% 106 ms	96.4% 78 ms	93.6% 90 ms	91.0% 90 ms
Bursztein et al. [15]	RR 38.68% BT 3.94 s	55.22% 1.9 s	51.39% 2.31 s	5.33% 7.95 s	22.67% 4.59 s	28.29%	
Starostenko et al. [16]	RR BT			91.5%	54.6% < 0.5 s		
Gao et al. [17]	RR 34%			55%	34%		
Gao et al. [18]	RR BT	51% 7.58 s		36% 14.72 s			
Goodfellow et al. [6]	RR				99.8%		
Stark et al. [8]	RR				90%		

Facebook Captcha

Observed images	 (W4kgvQ)	 (uV7FeWB)	 (MqhnpT)		\$40M raise
Inference					
Training traces	10^7	W4kgvQ	uV7EeWB	MqhnpT	
	10^6	WA4rjvQ	uV7FeWB	MypppT	
	10^5	Woxewd9	mTTTEMMm	RIrPEs	
	10^4	BKvu2Q	C9QDsoN	rS5FP2B	

Summary

- Probabilistic programming
 - Unified model denotation
 - Automated inference
- Deep learning
 - Powerful regressor
 - Automatic differentiation
 - Needs lots of labeled training data
- Inference compilation
 - Fast repeated inference
 - Training data for deep learning models

Exercise: Write generative model for Captcha in Anglican