Pyro

Introduction to Probabilistic Programming:
Models and Inference in Pyro

UBER AI Labs

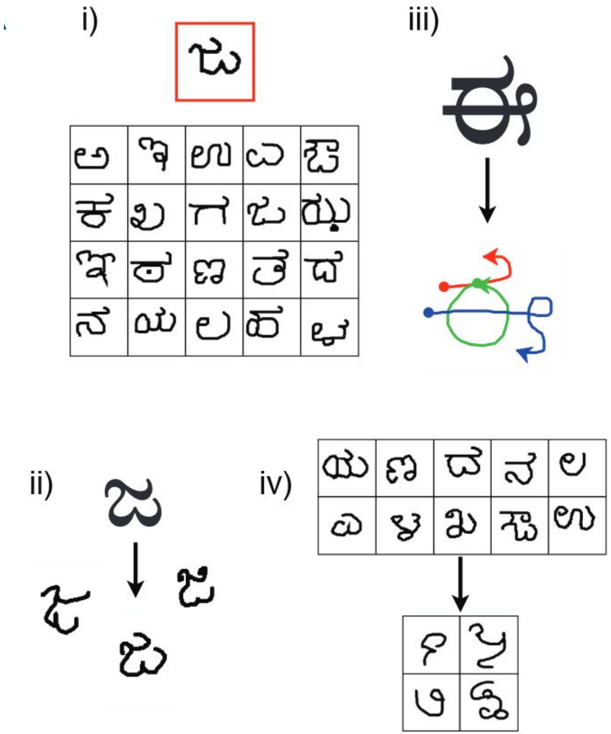
In this tutorial:

1. Why probabilistic modelling for AI and machine learning?
2. Why probabilistic programming? Why Pyro?
3. Building up models as probabilistic programs
4. Inference: fitting Pyro programs to observed data

Based on the following Pyro tutorials:

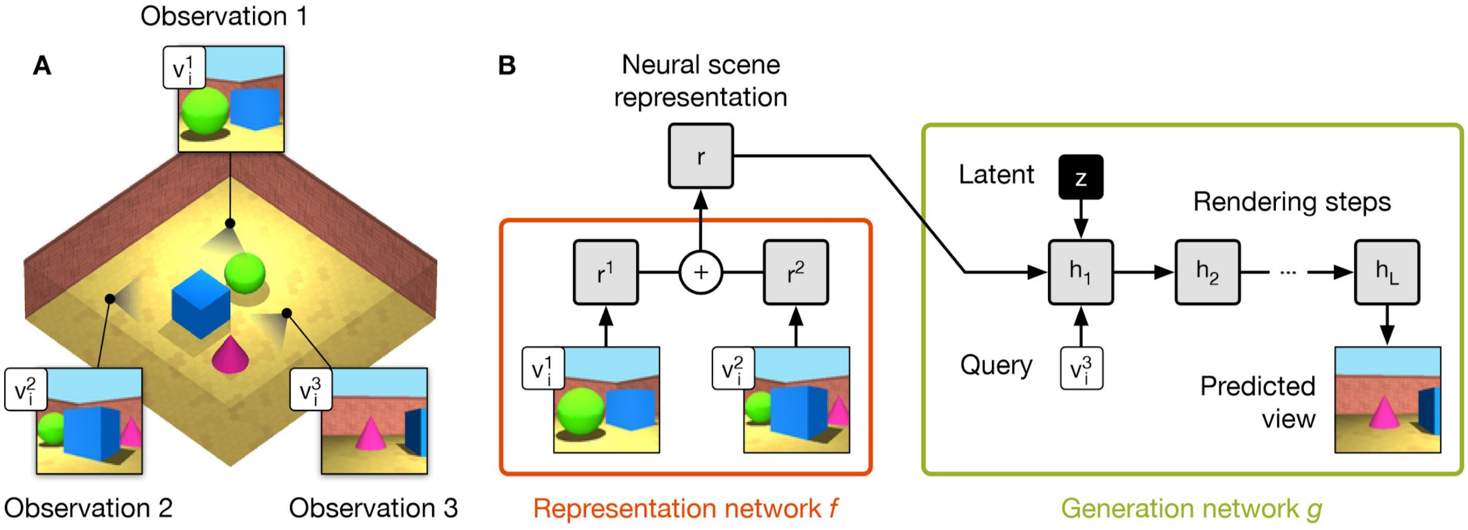
- Models in Pyro: http://pyro.ai/examples/intro_part_i.html
- Inference in Pyro: http://pyro.ai/examples/intro_part_ii.html

Probabilistic Modeling in AI: Frontiers



Human-level concept learning through probabilistic program induction, Lake et al.

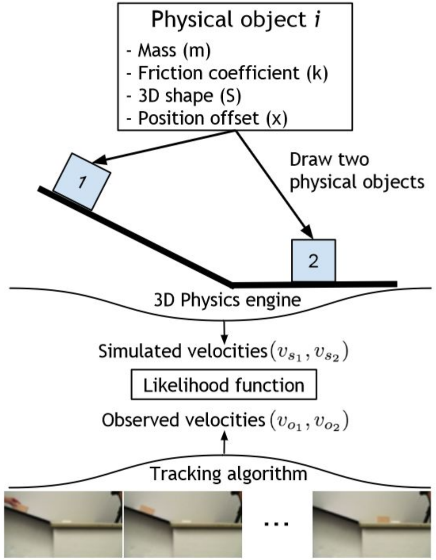
Probabilistic Modeling in AI: Frontiers

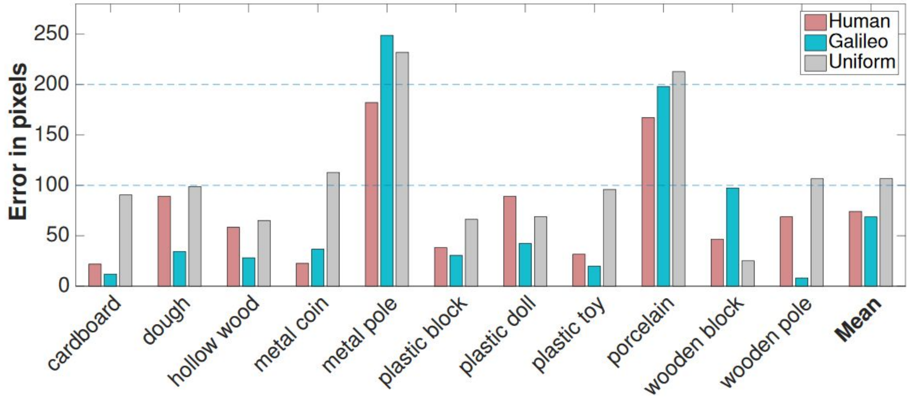


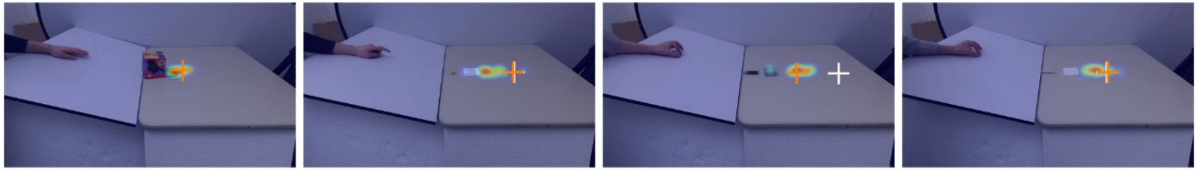
Neural Scene Representation and Rendering, Eslami et al.

U B E R

Probabilistic Modeling in AI: Frontiers

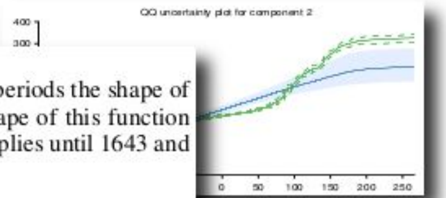






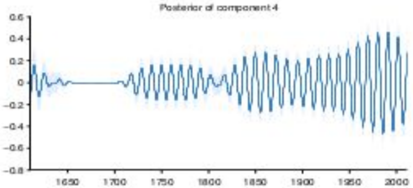
Galileo: Perceiving Physical Object Properties by Integrating a Physics Engine with Deep Learning, Wu et al.

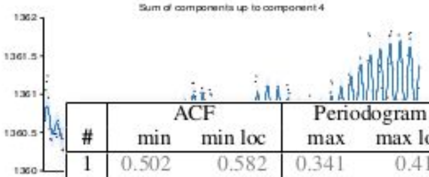
Probabilistic Modeling in AI: Frontiers



This component is approximately periodic with a period of 10.8 years. Across periods the shape of this function varies smoothly with a typical lengthscale of 36.9 years. The shape of this function within each period is very smooth and resembles a sinusoid. This component applies until 1643 and from 1716 onwards.

This component explains 71.5% of the residual variance; this increases the total variance explained from 72.8% to 92.3%. The addition of this component reduces the cross validated MAE by 16.82% from 0.18 to 0.15.





	ACF		Periodogram		QQ	
#	min	min loc	max	max loc	max	min
1	0.502	0.582	0.341	0.413	0.341	0.679
2	0.802	0.199	0.558	0.630	0.049	0.785
3	0.251	0.475	0.799	0.447	0.534	0.769
4	0.527	0.503	0.504	0.481	0.430	0.616
5	0.493	0.477	0.503	0.487	0.518	0.381

Automatic Construction and Natural-Language Description of Nonparametric Regression Models, Lloyd et al.

UBER

Probabilistic inference

Everything follows from two simple rules:

Sum rule: $P(x) = \sum_y P(x, y)$

Product rule: $P(x, y) = P(x)P(y|x)$

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})}$$

$P(\mathcal{D}\|\theta)$	likelihood of θ
$P(\theta)$	prior probability of θ
$P(\theta\|\mathcal{D})$	posterior of θ given $\mathcal{D}$

(h/t Zoubin)

Probabilistic programming languages

Probabilistic models: Representation of uncertain knowledge and reasoning.

+

Programming languages: Uniform, universal specification of process, with high-level abstractions.

Recipe:

A nice high-level PL,

Distribution objects,

Sample statements,

Condition, to affect weight of execution traces,

Inference to compute posterior and marginal distributions.

Why aren’t we building everything with PPLs?

Scalability: inference in high-dimensional models and large datasets requires high-performance algorithms and systems

Flexibility: advanced models require model-specific runtime behavior or inference algorithms that are difficult to implement in PPLs

Why aren’t we building everything with PPLs?

Expressivity: writing rich models quickly and concisely requires languages with advanced control flow, modularity, and tooling

Scalability: inference in high-dimensional models and large datasets requires high-performance algorithms and systems

Flexibility: advanced models require model-specific runtime behavior or inference algorithms that are difficult to implement in PPLs

Pyro: A Deep Universal PPL

Pyro is **expressive**:

- Models are functions with arbitrary Python code, including all control flow
- Pyro primitives for: sampling, observation, and learnable parameters

Pyro is **scalable**:

- Variational method takes a model and an inference model (or *guide*) and optimizes Evidence Lower Bound, with advanced features like subsampling and variance reduction
- High-performance automatic differentiation and tensor math with PyTorch

Pyro is **flexible**:

- Guides are arbitrary programs, allowing injection of knowledge or easy troubleshooting
- Inference algorithms built with Poutine, an extensible, hackable, composable library of declarative building blocks for modifying the behavior of probabilistic programs

Probabilistic programs

Probabilistic programs are regular programs that call stochastic functions:

```
def weather(p_cloudy):
    is_cloudy = torch.distributions.Bernoulli(p_cloudy).sample()

    if is_cloudy:
        loc, scale = 55.0, 10.0
    else:
        loc, scale = 75.0, 15.0

    temperature = torch.distributions.Normal(loc, scale).sample()
    return is_cloudy, temperature
```

Writing probabilistic programs in Pyro

Pyro code is just Python with stochastic calls wrapped in `pyro.sample`:

```python
def weather(p_cloudy):
    is_cloudy = pyro.sample("is_cloudy", pyro.distributions.Bernoulli(p_cloudy))

    if is_cloudy:
        loc, scale = 55.0, 10.0
    else:
        loc, scale = 75.0, 15.0

    temperature = pyro.sample("temp", pyro.distributions.Normal(loc, scale))
    return is_cloudy, temperature
```

Composing probabilistic programs in Pyro

Pyro programs can be composed freely, if sample site names are unique:

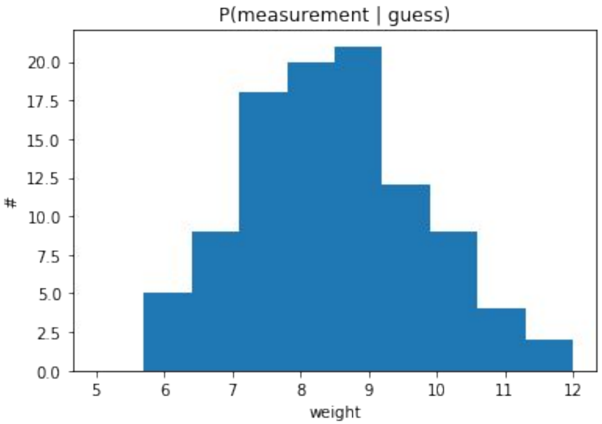
```
def ice_cream_sales():

    is_cloudy, temperature = weather(0.3)

    if not is_cloudy and temperature > 80.0:
        expected_sales = 200.
    else:
        expected_sales = 50.

    return pyro.sample('sales', Normal(expected_sales, 10.0))
```

Even simpler example: noisy scale

```python
def scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(weight, 0.75))
```

Inference: what measurements would we expect?

```python
def scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(weight, 0.75))
```



Inference: conditioning a model on data

Conditioning fixes the value of sample statements:

```
def conditioned_scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(weight, 0.75), obs=9.5)
```

Equivalent to:

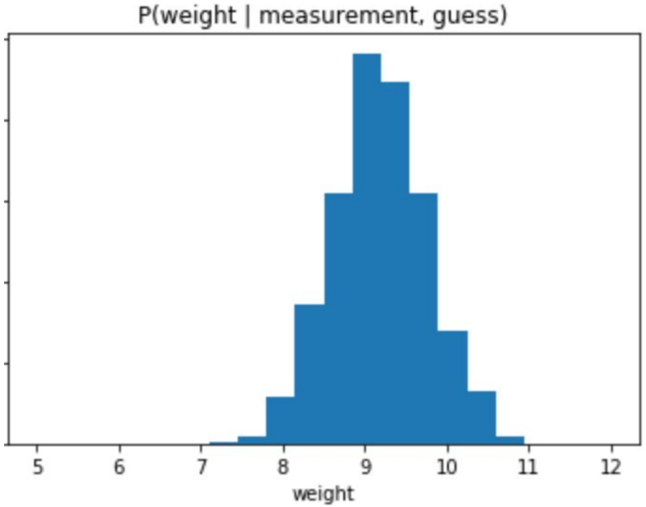
```
conditioned_scale = pyro.condition(scale, data={"measurement": 9.5})
```

UBER

Inference: conditioning a model on data

Inference algorithms compute the distribution of unconstrained sites:

```
conditioned_scale = pyro.condition(scale, data={"measurement": 9.5})

posterior = pyro.infer.Importance(conditioned_scale, num_samples=1000)
marginal = pyro.infer.EmpiricalMarginal(posterior.run(8.5), sites="weight")
```



UBER

Inference: guide functions

We can do inference by building a model of the posterior:

```
def conditioned_scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(weight, 0.75), obs=9.5)


def guide(guess):
    return pyro.sample("weight", ...)
```

Inference: guide functions

The scale model is so simple that the true posterior can be computed by hand:

```
def deferred_conditioned_scale(measurement, guess):
    return pyro.condition(scale, data={"measurement": measurement})(guess)


def true_posterior_guide(measurement, guess):
    a = (guess + torch.sum(measurement)) / (measurement.size(0) + 1.0)
    b = 1. / (measurement.size(0) + 1.0)
    return pyro.sample("weight", Normal(a, b))
```

Inference: guide functions

Guides estimate the posterior directly:

```
def true_posterior_guide(measurement, guess):
    a = (guess + torch.sum(measurement)) / (measurement.size(0) + 1.0)
    b = 1. / (measurement.size(0) + 1.0)
    return pyro.sample("weight", Normal(a, b))
```

Inference: intractability

In most interesting models, the true posterior cannot be computed by hand

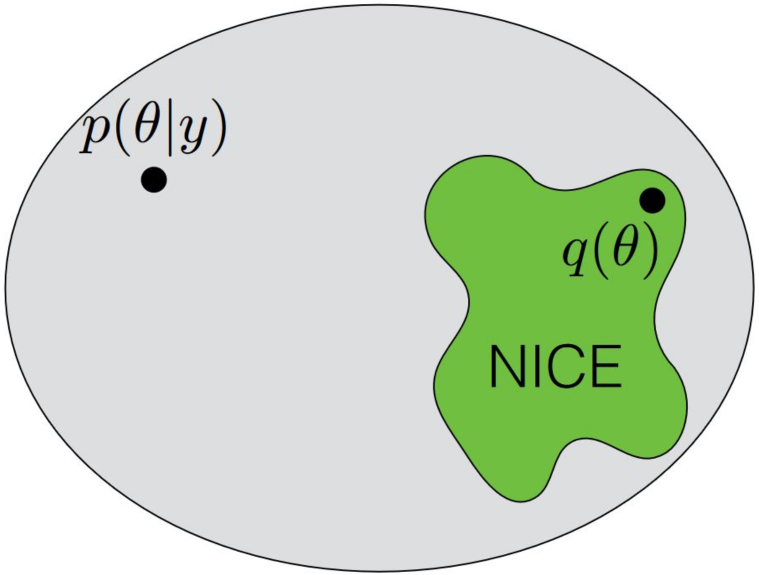
```
def scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(weight, 0.75))
```

Spot the difference!

```
def intractable_scale(guess):
    weight = pyro.sample("weight", Normal(guess, 1.0))
    return pyro.sample("measurement", Normal(fn(weight), 0.75))
```

Inference as optimization

Instead of one guide, we could guess an entire parametrized family:



UBER

[Variational Bayes and Beyond: Bayesian Inference for Big Data](), Broderick, T.

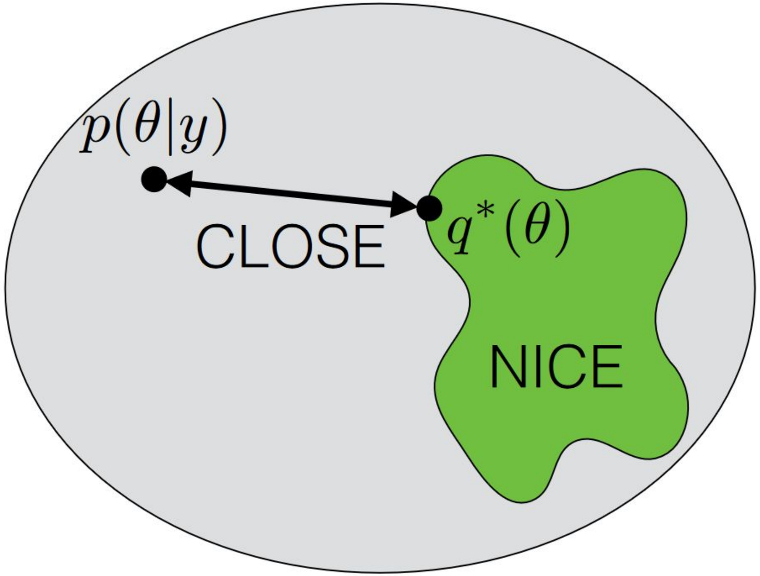
Inference as optimization

Instead of one guide, we could guess an entire parametrized family:

```
def parametrized_guide(guess):
    a = pyro.param("a", torch.tensor(torch.randn(1) + guess.detach()))
    b = pyro.param("b", torch.randn(1), constraint=constraints.positive)
    return pyro.sample("weight", Normal(a, b))
```

Inference as optimization

We search for the best guide by optimizing parameters with a loss function:



UBER

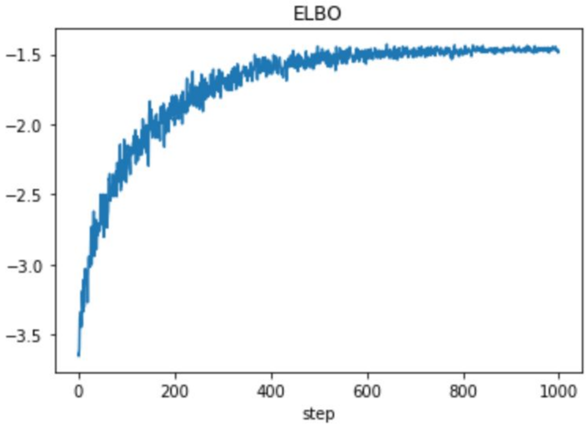
[Variational Bayes and Beyond: Bayesian Inference for Big Data](), Broderick, T.

Inference as optimization

We search for the best guide by optimizing parameters with a loss function using a light wrapper over PyTorch's stochastic gradient descent optimizer:

```
svi = pyro.infer.SVI(model=conditioned_scale,
                     guide=parametrized_guide,
                     optim=pyro.optim.SGD({"lr": 0.001}),
                     loss=pyro.infer.Trace_ELBO(...))
```

```
for t in range(1000):
    svi.step(guess)
```



UBER

Why build Pyro around inference as optimization?

Revisiting our design principles:

Express rich models: not constrained by needing to know lots of integrals

Scalable to large models and large datasets: gradient-based optimizers work in high dimensions, stochastic optimizers use minibatches of data and latents

Flexible guide programs offer large a surface area for incorporating knowledge and troubleshooting software or statistical failures

UBER

Recap

1. A whirlwind tour of some recent breakthroughs in AI research
2. The case for probabilistic programming, and for Pyro
3. Building up models as probabilistic programs
4. Inference: fitting Pyro programs to observed data
5. Inference as optimization in Pyro

Coming up: an introduction to Bayesian machine learning in Pyro

pyro.ai

Eli Bingham

JP Chen

Martin Jankowiak

Theo Karaletsos

Fritz Obermeyer

Neeraj Pradhan

Rohit Singh

Paul Szerlip

Noah Goodman

Special thanks to

Paul Horsfall
Dustin Tran
Soumith Chintala
Adam Paszke
Du Phan

Would you like to know more?

Pyro tutorials web page: http://pyro.ai/examples/index.html

Detailed walkthrough of Pyro implementation of VAE:

http://pyro.ai/examples/vae.html

Deep dive into the math and implementation of stochastic variational inference in Pyro:

http://pyro.ai/examples/svi_part_i.html

Detailed description of tensor and distribution shapes and broadcasting in Pyro:

http://pyro.ai/examples/tensor_shapes.html

Discussion: implications of Pyro's design

Pyro is **homoiconic**: inference algorithms are Pyro programs, and internal data structures like Traces are ordinary Pyro objects, enabling nested inference and metainference

Pyro code really is **just Python code**: same ecosystem and runtime performance, so making Pyro programs faster or more efficient is no different from optimizing any other Python code

Programmability allows for **automation**: parts of Pyro left up to user specification, like names or guides, can be targeted for automatic generation without affecting the rest of Pyro