Pyro

Deep Probabilistic Programming 101: The Variational Autoencoder

UBER AI Labs

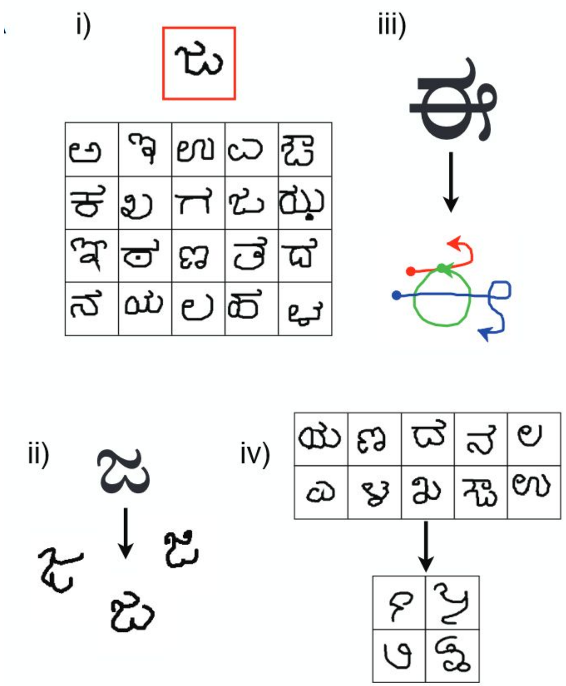
In this tutorial:

1. Introduction to deep generative models and model learning
2. Implementing a simple deep generative model with Pyro
3. Performing variational inference with model learning in the VAE
4. A brief look at how inference algorithms are implemented in Pyro

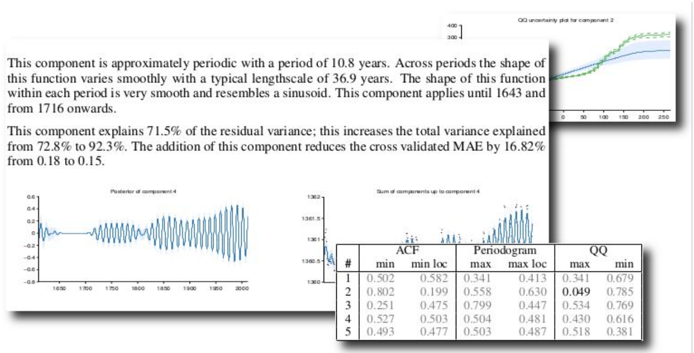
Based on the following Pyro tutorials:

- Variational Autoencoders: http://pyro.ai/examples/vae.html
- Intro to SVI: http://pyro.ai/examples/svi_part_i.html

Probabilistic Modelling in AI: Frontiers

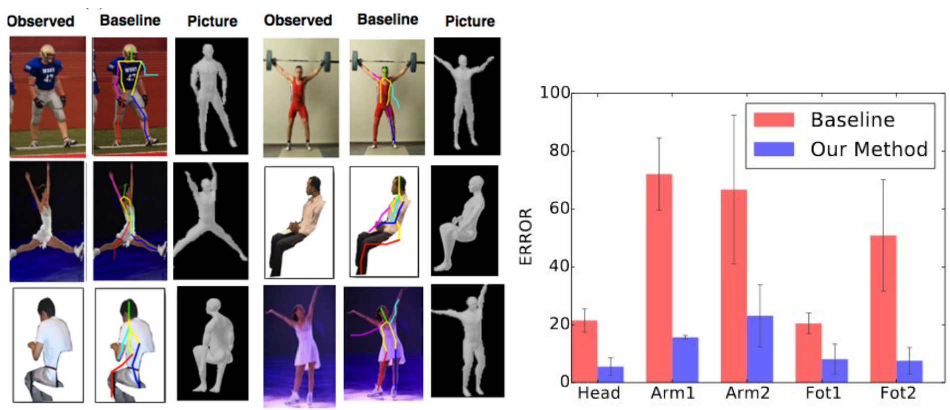


Human-level concept learning through probabilistic program induction, Lake et al.



#	ACF min	ACF min loc	Periodogram max	Periodogram max loc	QQ max	QQ 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

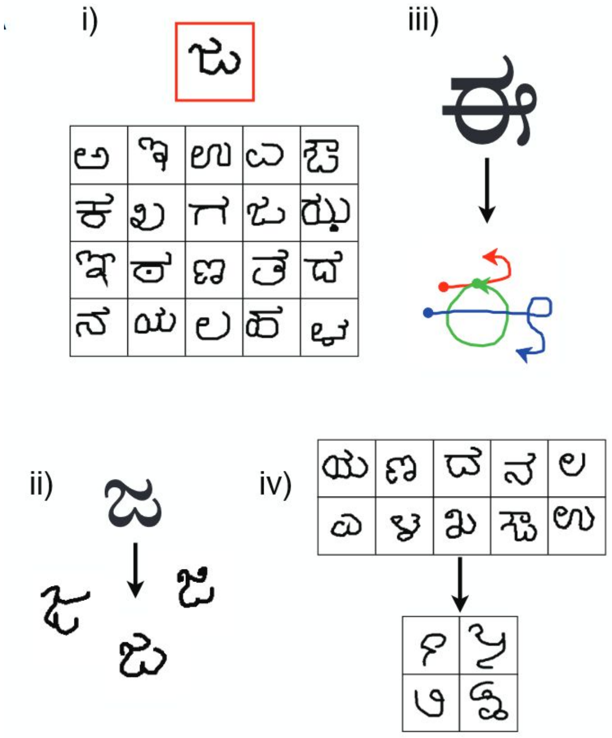
Automatic Construction and Natural-Language Description of Nonparametric Regression Models, Lloyd et al.



Picture: A Probabilistic Programming Language for Scene Perception, Kulkarni et al.

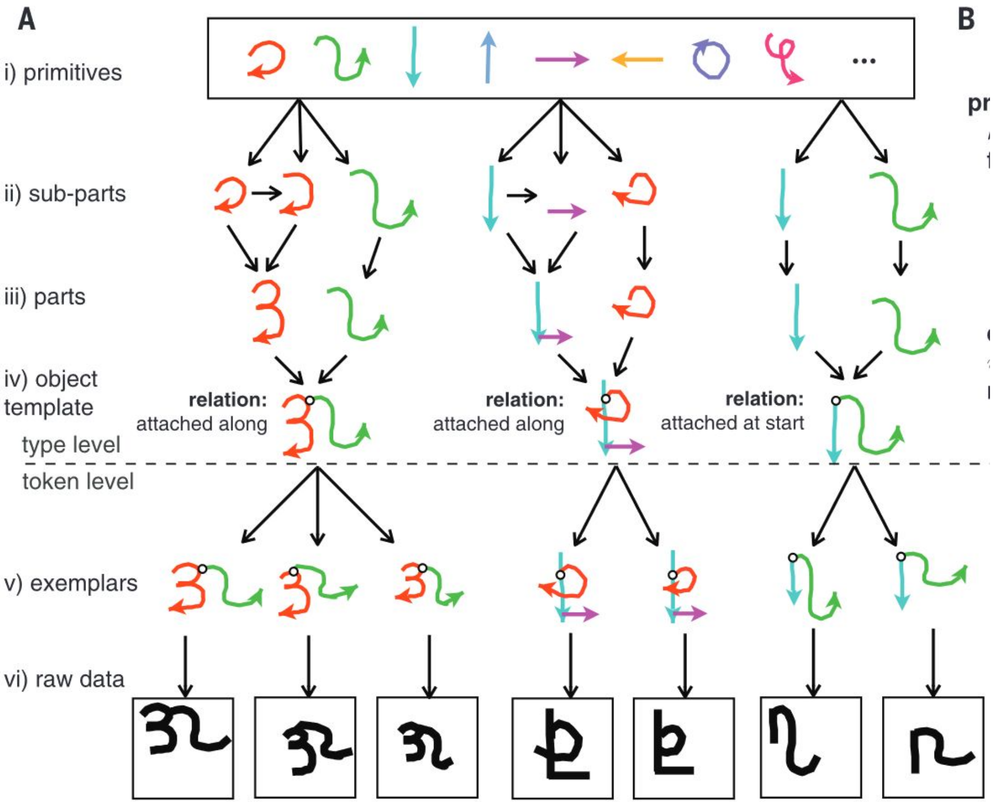
UBER

Probabilistic Modelling in AI: Frontiers



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

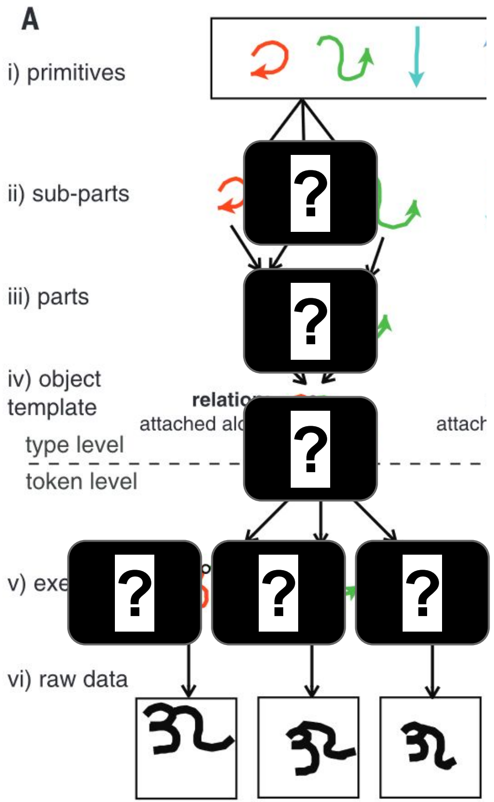
Probabilistic Modelling in AI: Frontiers



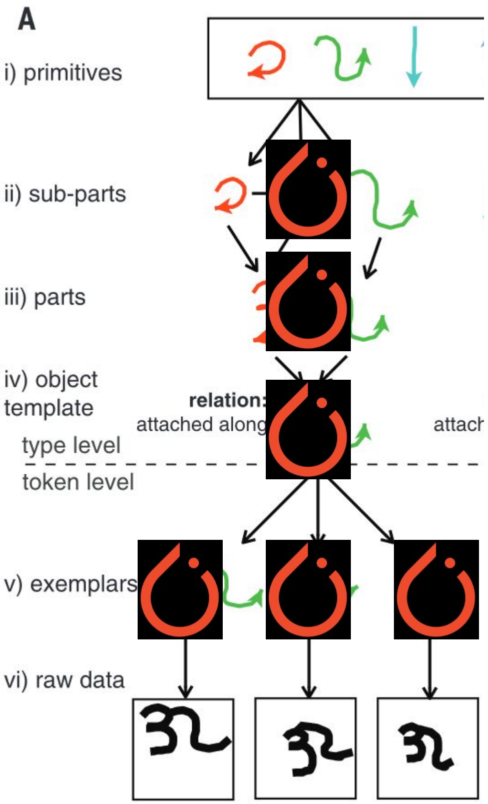
UBER

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

What if we can't write down a good model?

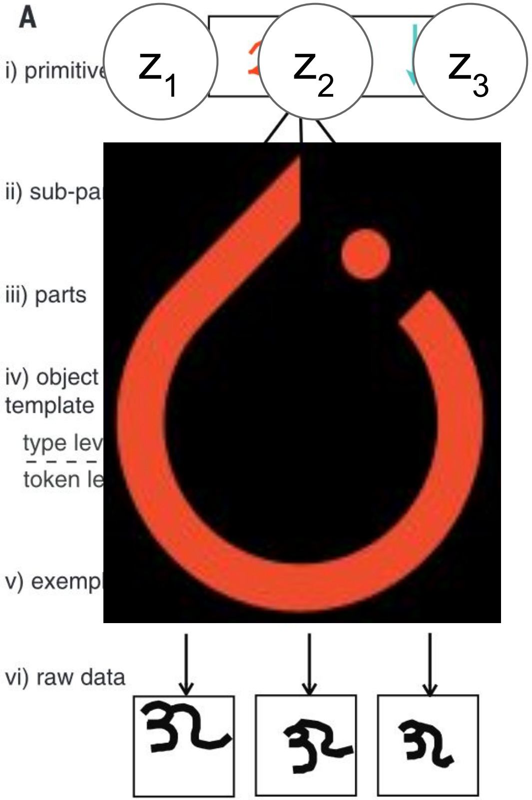


Deep generative models



UBER

Decoder network: the simplest deep generative model



Model learning

We want to maximize the probability that a model *p(**x**, **z**)* generates data **x**:

$$\log p_\theta(x) = \log \int d\mathbf{z}\; p_\theta(\mathbf{x}, \mathbf{z})$$

But computing this integral directly is too difficult for most interesting models

Model learning and variational inference

Recall the definition of the ELBO from the previous tutorial:

$$\mathrm{ELBO} = C - \mathrm{KL}(q_\phi(\mathbf{z})||p(\mathbf{z}|x))$$

It turns out that the constant C is exactly the model log-evidence log p(x):

$$\mathrm{ELBO} = \log p_\theta(x) - \mathrm{KL}(q_\phi(\mathbf{z})||p_\theta(\mathbf{z}|x))$$

Model learning and variational inference

Recall that the KL divergence is non-negative.

Then the ELBO is a ***lower bound*** to the model's log ***evidence*** for any guide q***:***

$$\mathrm{ELBO} \leq \log p_\theta(x)$$

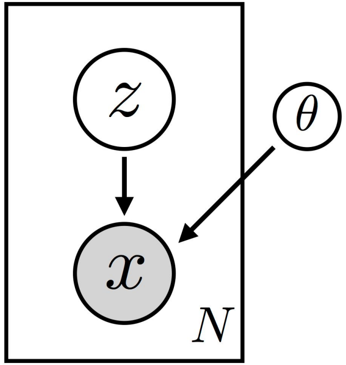
So we can learn a model by optimizing its parameters wrt the ELBO

Data: MNIST handwritten digits

Model

The decoder model samples a latent code, passes it through a neural network, and samples an observation:

```python
def model():
    pyro.module("decoder", nn_decoder)
    z = pyro.sample("z", Normal(0., 1.).expand_by([20]))
    bern_prob = nn_decoder(z)
    return pyro.sample("x", Bernoulli(bern_prob))
```



UBER

Auto-Encoding Variational Bayes, Diederik P Kingma, Max Welling

Model

The neural network `nn_decoder` is just a standard PyTorch `nn.Module`:

```
nn_decoder = nn.Sequential(nn.Linear(20, 100),
                           nn.Softplus(),
                           nn.Linear(100, 784),
                           nn.Sigmoid())
```

`pyro.module` just calls `pyro.param` on each of its parameters

Auto-Encoding Variational Bayes, Diederik P Kingma, Max Welling

Guide

The simplest guide: independent `Normal` distributions for each datapoint

```
def guide():
    ...
    loc_z = pyro.param("loc_z", ...)
    scale_z = pyro.param("scale_z", ...)
    return pyro.sample("z", Normal(loc_z, scale_z))
```

UBER

Inference by optimization

Model learning could require learning a new guide for each model:

```
svi_guide = pyro.infer.SVI(model=conditioned_model_fixed_params, guide, ...)

svi_model = pyro.infer.SVI(model=conditioned_model, guide_fixed_params, ...)
```

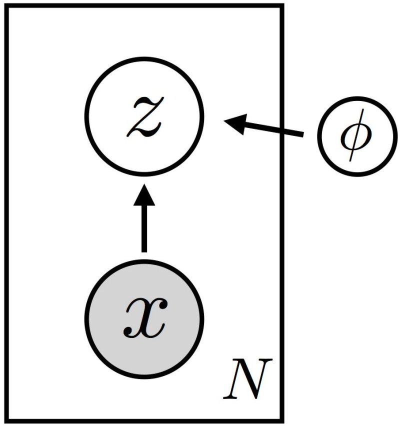
This is computationally infeasible with our mean-field guide:

```
for batch in batches:
    ...
    for t in range(100):
        svi_guide.step(batch)
    svi_model.step(batch)
```

Amortized guide

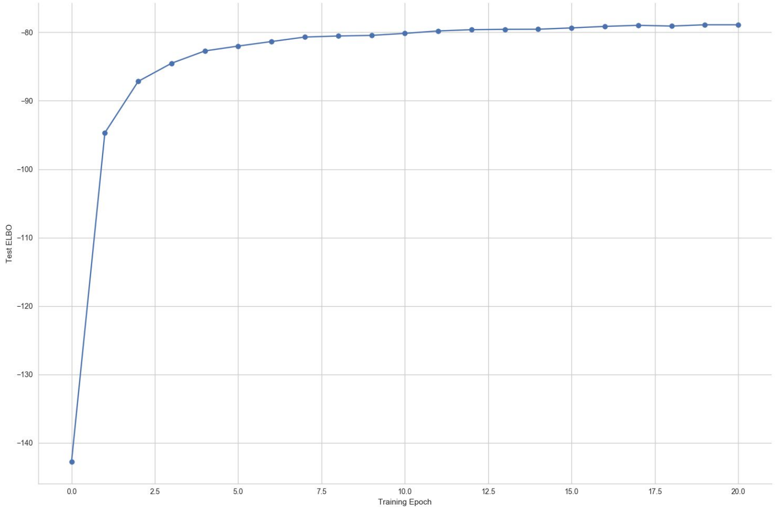
Instead of optimizing new parameters from scratch for each datapoint, we train a second neural network to guess the parameters of $q(\mathbf{z} \mid \mathbf{x})$:

```
def guide(x):
    pyro.module("encoder", encoder)
    loc_z, scale_z = encoder(x)
    return pyro.sample("z", dist.Normal(loc_z, scale_z))
```



VAE: inference

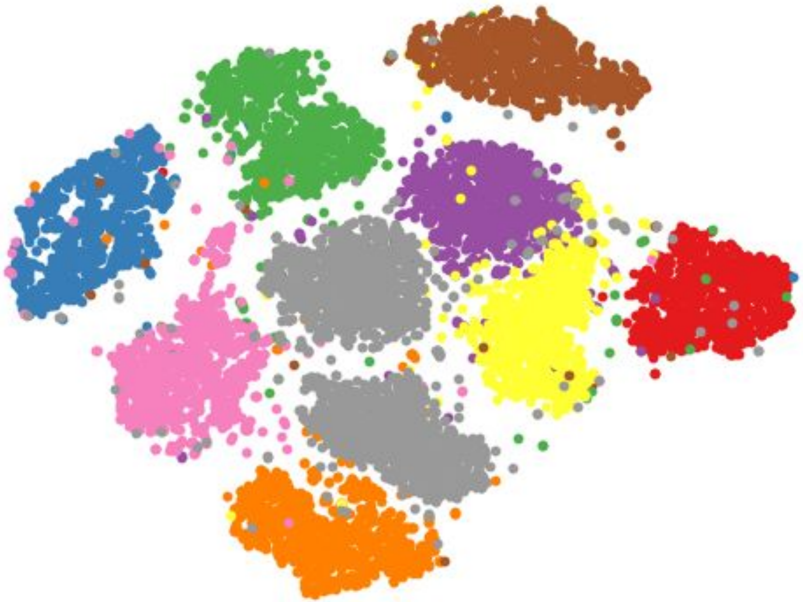
```
svi = pyro.infer.SVI(model=conditioned_model,
                     guide=guide,
                     optim=Adam({"lr": 0.001}),
                     loss=pyro.infer.Trace_ELBO())

for batch in batches:
    svi.step(batch)
```



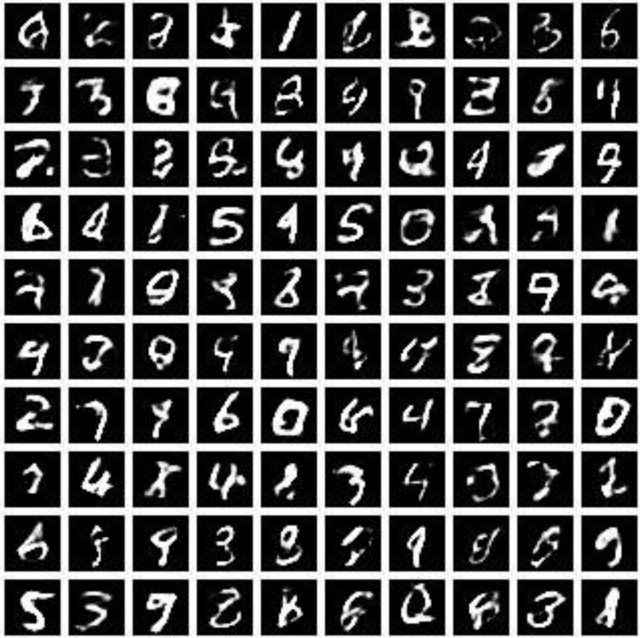
progress on elbo during training

VAE: results

latent embedding



samples from the generative model



[See our VAE tutorial for complete results]

Deep probabilistic programming on a postcard

```python
def model():
    pyro.module("decoder", decoder)
    z = pyro.sample("z", Normal(0., 1.).expand_by([20]))
    bern_prob = nn_decoder(z)
    return pyro.sample("x", Bernoulli(bern_prob))

def conditioned_model(x):
    return pyro.condition(model, data={"x": x})()


nn_decoder = nn.Sequential(
    nn.Linear(20, 100),
    nn.Softplus(),
    nn.Linear(100, 784),
    nn.Sigmoid()
)
```

```python
def guide(x):
    pyro.module("encoder", nn_encoder)
    m_z, s_z = nn_encoder(x)
    return pyro.sample("z", dist.Normal(m_z, s_z))


svi = pyro.infer.SVI(model=conditioned_model,
                     guide=guide,
                     optim=Adam({"lr": 0.001}),
                     loss=pyro.infer.Trace_ELBO())


for batch in batches:
    svi.step(batch)
```

UBER

Estimating the ELBO

$$\mathrm{ELBO} \equiv \mathbb{E}_{q_\phi(\mathbf{z})}\left[\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z})\right]$$

We keep using `pyro.infer.Trace_ELBO` for training guides:

```
svi = pyro.infer.SVI(..., loss=pyro.infer.Trace_ELBO())
```

Is this where all the complexity is hiding?

Poutine: building blocks for probabilistic programming

pyro.poutine: Composable higher-order functions (handlers) that compute side effects and modify behavior at sample and parameter sites

- **condition**: given a dict of sample site names and values, mark those sites as observed and set their outputs to the values in the dictionary
- **trace**: create a dictionary containing the inputs, functions, and outputs found at each sample and parameter site in a single execution
- **replay**: given a dictionary of sample sites and values, replace the output at each sample site with the value at that site in dictionary
- And others…

Internally, handlers install themselves on a global stack and pass messages up and down the stack at each sample and parameter site

Poutine: building blocks for probabilistic programming

pyro.poutine: Composable higher-order functions (handlers) that compute side effects and modify behavior at sample and parameter sites

- **condition**: given a dict of sample site names and values, mark those sites as observed and set their outputs to the values in the dictionary
- **trace**: create a dictionary containing the inputs, functions, and outputs found at each sample and parameter site in a single execution
- **replay**: given a dictionary of sample sites and values, replace the output at each sample site with the value at that site in dictionary
- And others…

Internally, handlers install themselves on a global stack and pass messages up and down the stack at each sample and parameter site

Estimating the ELBO

$$\text{ELBO} \equiv \mathbb{E}_{q_\phi(\mathbf{z})}\left[\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z})\right]$$

Pyro inference code can be as compact and readable as model code:

```
def simple_mc_elbo(model, guide, *args):
    guide_trace = trace(guide).get_trace(*args)
    model_trace = trace(replay(model, trace=guide_trace)).get_trace(*args)
    return model_trace.log_prob_sum() - guide_trace.log_prob_sum()
```

Estimating the ELBO

We add a negative sign, because PyTorch optimizers minimize:

```
def simple_mc_elbo(model, guide, *args):
    guide_trace = trace(guide).get_trace(*args)
    model_trace = trace(replay(model, trace=guide_trace)).get_trace(*args)
    elbo = model_trace.log_prob_sum() - guide_trace.log_prob_sum()
    return -elbo
```

This can now be used directly in place of `Trace_ELBO`:

```
svi = pyro.infer.SVI(..., loss=simple_mc_elbo)
```

Recap

1. Introduction to deep generative models and model learning
2. Implemented a simple deep generative model with Pyro
3. Performed variational inference with model learning in the VAE
4. Took a brief look at how Pyro works under the hood

Coming up: building up more complex models from the VAE

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