Probabilistic Programming with Pyro
Why Pyro?

Why probabilistic modeling? To correctly capture uncertainty in models and predictions, for unsupervised and semi-supervised learning, and to provide AI systems with declarative prior knowledge.

Why (universal) probabilistic programs? To provide a clear and high-level, but complete, language for specifying complex models.

Why deep probabilistic models? To learn generative knowledge from data and reify knowledge of how to do inference.

Why inference by optimization? To enable scaling to large data and leverage advances in modern optimization and variational inference.
What is Pyro?

Pyro models are functions with:

- Arbitrary Python and PyTorch code
- Pyro primitives for: sampling, observation, and learnable parameters

Pyro automates inference:

- Variational method takes a model and an inference model (or guide) and optimizes Evidence Lower Bound.
- Tools to reduce the variance of gradient estimates, handle mini-batching, etc.
- Can also do exact inference, importance sampling, and coming soon: MCMC, SMC.
Design Principles

Universal: Pyro can represent any computable probability distribution.

Scalable: Pyro scales to large data sets with little overhead.

Minimal: Pyro is implemented with a small core of powerful, composable abstractions.

Flexible: Pyro aims for automation when you want it, control when you need it.
Pyro primitives

```python
pyro.sample("my_sample", dist.normal, mu, sigma)
```

Variable containing:
-0.3098
[torch.FloatTensor of size 1]

```python
pyro.sample("my_observed_sample", dist.normal, mu, sigma,
            obs=Variable(torch.ones(1)))
```

Makes sense only in the context of an inference algorithm!

```python
pyro.param("mu", Variable(torch.ones(1), requires_grad=True))
```

Variable containing:
1
[torch.FloatTensor of size 1]

U B E R
Variational autoencoder (VAE): model

def model():
    pyro.module("decoder", nn_decoder)
    z = pyro.sample("z", dist.normal, ng_zeros(20), ng_ones(20))
    bern_prob = nn_decoder(z)
    return pyro.sample("x", dist.bernoulli, bern_prob)

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

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

Stochastic Backpropagation and Approximate Inference in Deep Generative Models,
Danilo Jimenez Rezende, Shakir Mohamed, Daan Wierstra
Variational autoencoder (VAE): model

```
def model(x):
    pyro.module("decoder", nn_decoder)
    z = pyro.sample("z", dist.normal, ng_zeros(20), ng_ones(20))
    bern_prob = nn_decoder(z)
    return pyro.sample("x", dist.bernoulli, bern_prob, obs=x)

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

Register parameters in decoder with Pyro
Sample latent code
Decode latent code
Observe image

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

Stochastic Backpropagation and Approximate Inference in Deep Generative Models,
Danilo Jimenez Rezende, Shakir Mohamed, Daan Wierstra
**VAE: inference**

```python
def guide(x):
    # nn_encoder is another neural network,
    # that takes in an image x and outputs the parameters
    # of a distribution over a latent code z given x
    pyro.module("encoder", nn_encoder)
    mu_z, sig_z = nn_encoder(x)
    return pyro.sample("z", dist.normal, mu_z, sig_z)
```
VAE: inference

svi = pyro.infer.SVI(model=model,
                     guide=guide,
                     optim=pyro.optim.Adam({'lr': 0.001}),
                     loss='ELBO')

losses = []
for batch in batches:
    losses.append(svi.step(batch))

progress on elbo during training
VAE: results

latent embedding

samples from the generative model
Cool.

With the VAE in hand we can now tackle all unsupervised learning problems, right?
VAE: Multi-MNIST

test set reconstructions (0/1/2 digits)

not so great... maybe we need a more sophisticated model?
Let’s build in an abstract **object concept**:  
- scenes are made of **some number** of things  
- each thing has a **what**  
- and a **where**


UBER
Let’s build in an abstract **object concept**:  
- scenes are made of **some number** of things  
- each thing has a **what** (VAE-style appearance model)  
- and a **where** (location and size to place into image)
Interlude: recursion, random control flow, and other awesomeness

the geometric distribution defined recursively with bernoulli random trials

```python
def geom(num_trials=0):
    p = Variable(torch.Tensor([0.5]))
    x = pyro.sample('x{}'.format(num_trials), dist.bernoulli, p)
    if x.data[0] == 1:
        return num_trials
    else:
        return geom(num_trials + 1)
```

dynamically named random variable

different executions of geom() can have different numbers of random variables
Back to AIR: A recursive prior for images

first we need to define a prior over a single object in the image

def prior_step(t):
    # Sample object pose. This is a 3-dimensional vector representing x,y position and size.
    z_where = pyro.sample('z_where_{}'.format(t),
                          dist.normal,
                          z_where_prior_mu, z_where_prior_sigma)

    # Sample object code. This is a 50-dimensional vector.
    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.normal,
                         z_what_prior_mu, z_what_prior_sigma)

    y_att = decode(z_what)    # Map latent code to pixel space using the neural network.

    y = object_to_image(z_where, y_att)    # Position/scale object within larger image
                                                # using (differentiable) spatial transformations

    return y
Back to AIR: A recursive prior for images

def geom_image_prior(x, step=0):
    p = Variable(torch.Tensor([0.5]))
    i = pyro.sample('{}i(\cdot{})'.format(step), dist.bernoulli, p)
    if i.data[0] == 1:
        return x
    else:
        x = x + prior_step(step)
    return geom_image_prior(x, step + 1)
AIR: Inference

```python
def guide_step(t, data, prev):
    rnn_input = torch.cat((data, prev.z_where, prev.z_what, prev.z_pres), 1)
    h, c = rnn(rnn_input, (prev.h, prev.c))
    z_pres_p, z_where_mu, z_where_sigma = predict(h)
    z_pres = pyro.sample('z_pres_{}'.format(t),
                         dist.bernoulli, z_pres_p * prev.z_pres)
    z_where = pyro.sample('z_where_{}'.format(t),
                          dist.normal, z_where_mu, z_where_sigma)
    x_att = image_to_object(z_where, data)  # Crop a small window from the input.

    # Compute the parameter of the distribution over z_what
    # by passing the window through the encoder network.
    z_what_mu, z_what_sigma = encode(x_att)
    z_what = pyro.sample('z_what_{}'.format(t),
                          dist.normal, z_what_mu, z_what_sigma)

    return  # values for next step
```

consume hidden state from previous step to inform sampling in this step
controls when to stop recursion
sample latent code for image patch
Variance Reduction

because we have discrete random variables a naive ELBO gradient estimator will contain terms of the form

\[ \nabla_\phi \log q(z_{\text{bern}}) \times \text{ELBO} \]

to reduce variance we instead use a gradient estimator with terms of the form

\[ \nabla_\phi \log q(z_{\text{bern}}) \times (D_{z_{\text{bern}}} - b) \]

in Pyro invoking the improved estimator is simple:

```python
pyro.sample('z_bern', dist.bernoulli, ..., baseline=dict(baseline_value=b))
svi = SVI(air.model, air.guide, ..., loss='ELBO', trace_graph=True)
```
AIR: Results

**training set ELBO**

**training set count accuracy**

test set reconstructions
Reusing components for image reconstruction

```python
import pyro.poutine as poutine

trace = poutine.trace(air.guide).get_trace(examples_to_viz, None)

z, recons = poutine.replay(air.prior, trace)(examples_to_viz.size(0))
```

- The trace contains all the random variables sampled by the guide.
- Replay the prior against the trace to get the return values of air.prior.

- Note that air.prior is the entire model except for the observation noise.
- This trace + replay paradigm is also how the backend constructs the ELBO.

Uber
pyro.ai

Special thanks to
Paul Horsfall
Dustin Tran
Soumith Chintala
Adam Paszke