Pyro

Jinwon Lee
Jaeseong Lee
Soyoon Oh

What is Pyro ?

A Probabilistic Programming Language written in Python and built PyTorch

Primitive Stochastic Functions

```
import torch
import pyro

loc = 0.
scale = 1.
normal = torch.distributions.Normal(loc, scale)
x = normal.rsample()
print("sample", x)
```

Primitive Stochastic Functions

```
import torch
import pyro

loc = 0.
scale = 1.
normal = torch.distributions.Normal(loc, scale)
x = normal.rsample()
print("sample", x)
```

```
>> sample tensor(-1.3905)
```

Tensor in Pyro

```
>>>tensor.new_zeros((2, 3))

tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]]
```

```
x = d.sample()
assert x.shape == d.batch_shape + d.event_shape
assert d.log_prob(x).shape == d.batch_shape
x2 = d.sample(sample_shape)
assert x2.shape == sample_shape + batch_shape + event_shape
```

Tensor in Pyro

```
d = Bernoulli(0.5 * torch.ones(3,4))
assert d.batch_shape == (3, 4)
assert d.event_shape == ()
x = d.sample()
assert x.shape == (3, 4)
assert d.log_prob(x).shape == (3, 4)
```

```
d = MultivariateNormal(torch.zeros(3), torch.eye(3, 3))
assert d.batch_shape == ()
assert d.event_shape == (3,)
x = d.sample()
assert x.shape == (3,)
assert d.log_prob(x).shape == ()
```

Reshaping distribution

```
d = Bernoulli(0.5 * torch.ones(3,4)).to_event(1)
assert d.batch_shape == (3,)
assert d.event_shape == (4,)
x = d.sample()
assert x.shape == (3, 4)
assert d.log_prob(x).shape == (3,)
```

A Simple Model in Pyro

```
def weather():
    cloudy = pyro.sample('cloudy', pyro.distributions.Bernoulli(0.3))
    cloudy = 'cloudy' if cloudy.item() == 1.0 else 'sunny'
    mean_temp = {'cloudy': 55.0, 'sunny': 75.0}[cloudy]
    scale_temp = {'cloudy': 10.0, 'sunny': 15.0}[cloudy]
    temp = pyro.sample('temp', pyro.distributions.Normal(mean_temp,  scale_temp))
    return cloudy, temp.item()

for _ in range(3):
    print(weather())
```

```
>> ('cloudy', 64.5440444946289)
   ('sunny', 94.37557983398438)
   ('sunny', 72.5186767578125)
```

Conditioning

$$weight \mid guess \sim Normal(guess, 1)$$

$$measurement \mid guess, weight \sim Normal(weight, 0.75)$$

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

Conditioning

$$(weight \mid guess, measurement = 9.5) \sim ?$$

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

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

def scale_obs(guess):
    weight = pyro.sample("weight", dist.Normal(guess, 1.))
    return pyro.sample("measurement", dist.Normal(weight, 1.), obs=9.5)
```

Guide Function

`pyro.infer.SVI`

Importance

class `Importance`(*model, guide=None, num_samples=None*) [source]

MCMC

class `MCMC`(*kernel, num_samples, warmup_steps=None, num_chains=1, mp_context=None, disable_progbar=False*) [source]

SVI

class `SVI`(*model, guide, optim, loss, loss_and_grads=None, num_samples=10, num_steps=0, **kwargs*) [source]

Parametrized Stochastic Functions

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

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

function ← some nonlinear function

Parametrized Stochastic Functions

```
def scale_parametrized_guide(guess):
    a = pyro.param("a", torch.tensor(guess))
    b = pyro.param("b", torch.tensor(1.))
    return pyro.sample("weight", dist.Normal(a, torch.abs(b)))
```

```
from torch.distributions import constraints

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

Stochastic Variational Inference

Guide

1.

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

pyro.sample("measurement", dist.Normal(weight, 1.), obs=9.5)
```

2.

```
def model():
    pyro.sample(“x", ...)
```

→

```
def guide():
    pyro.sample(“x", ...)
```

Automatic Guide Generation

```
def model():
    . . .

def guide():
    . . .
```

```
def model():
    . . .

guide = AutoGuide(model)
```

AutoGuideList

class `AutoGuideList`(*model, prefix='auto'*) [source]

AutoContinuous

class `AutoContinuous`(*model, prefix='auto'*) [source]

AutoDelta

class `AutoDelta`(*model, prefix='auto'*) [source]

AutoMultivariateNormal

class `AutoMultivariateNormal`(*model, prefix='auto'*) [source]

Stochastic Variational Inference

Optimizers

```
from pyro.optim import Adam

def per_param_callable(module_name,  param_name):
    if param_name == 'my_special_parameter':
        return {"lr": 0.010}
    else:
        return {"lr": 0.001}

optimizer = Adam(per_param_callable)
```

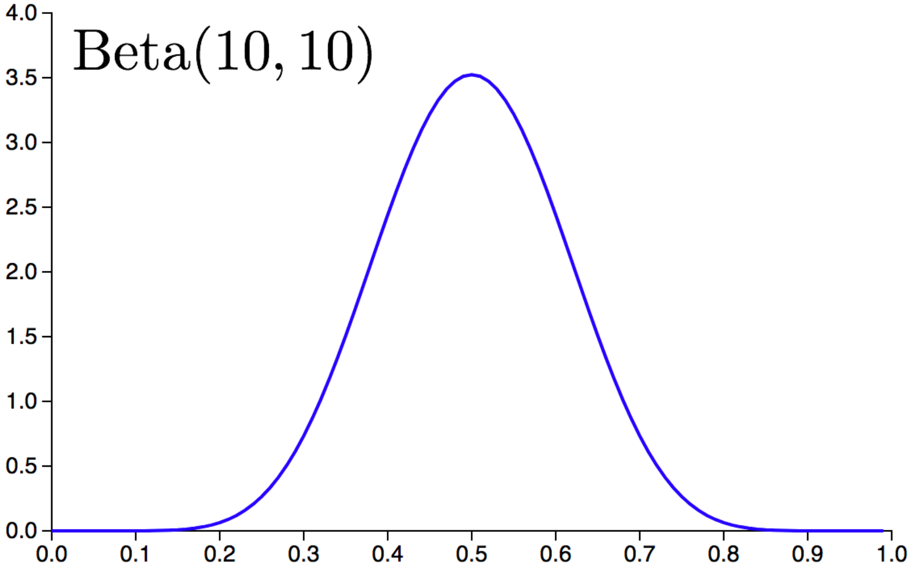
Stochastic Variational Inference

SVI Class

```
import pyro
from pyro.infer import SVI, Trace_ELBO
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
```

`step()` `evaluate_loss()`

A simple example



A simple example

```
import pyro.distributions as dist

def model(data):
    alpha0 = torch.tensor(10.0)
    beta0 = torch.tensor(10.0)
    f = pyro.sample("latent_fairness", dist.Beta(alpha0, beta0))
    for i in range(len(data)):
        pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])
```

A simple example

```
def guide(data):
    alpha_q = pyro.param("alpha_q", torch.tensor(15.0),
                         constraint=constraints.positive)
    beta_q = pyro.param("beta_q", torch.tensor(15.0),
                        constraint=constraints.positive)
    pyro.sample("latent_fairness", dist.Beta(alpha_q, beta_q))
```

A simple example

```
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

for step in range(n_steps):
    svi.step(data)

alpha_q = pyro.param("alpha_q").item()
beta_q = pyro.param("beta_q").item()

inferred_mean = alpha_q / (alpha_q + beta_q)
factor = beta_q / (alpha_q * (1.0 + alpha_q + beta_q))
inferred_std = inferred_mean * math.sqrt(factor)
```

Marking Conditional Independence

```
def model(data):
    f = pyro.sample("latent_fairness", dist.Beta(alpha0, beta0))
    for i in range(len(data)):
        pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])
```

↓

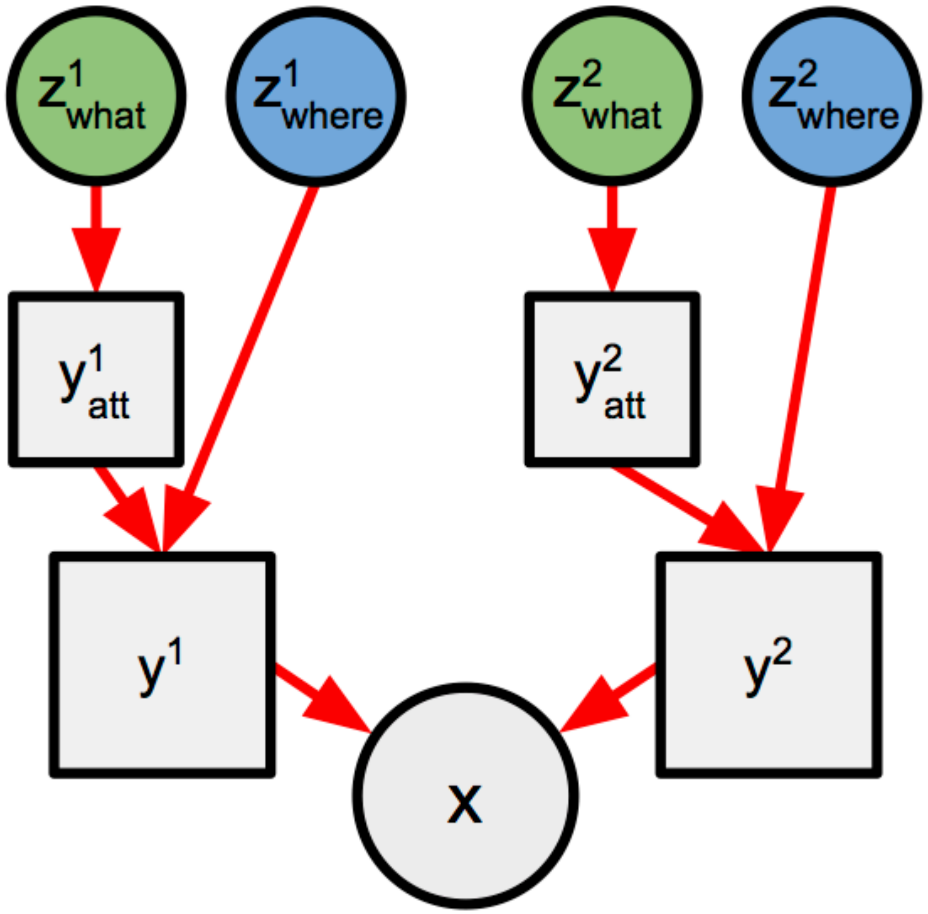
```
def model(data):
    f = pyro.sample("latent_fairness", dist.Beta(alpha0, beta0))
    for i in pyro.plate("data_loop", len(data)):
        pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])
```

Subsampling

```
for i in pyro.plate("data_loop", len(data), subsample_size=5):
    pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])
```

Example - Attend Inter Repeat

```
inpath = '../../examples/air/data'
(X_np, _), _ = multi_mnist(inpath, max_digits=2, canvas_size=50, seed=42)
X_np = X_np.astype(np.float32)
X_np /= 255.0
mnist = torch.from_numpy(X_np)
def show_images(imgs):
    figure(figsize=(8, 2))
    for i, img in enumerate(imgs):
        subplot(1, len(imgs), i + 1)
        axis('off')
        imshow(img.data.numpy(), cmap='gray')
show_images(mnist[9:14])
```



Two steps of the generative process

Generating a single object

- At each step a single object is generated.
- Each object is generated by passing its latent code through a neural network.
- We maintain uncertainty about the latent code used to generate each object, as well as its pose.

Generating a single object

```
# Create the neural network. This takes a latent code, z_what, to pixel intensities.
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.l1 = nn.Linear(50, 200)
        self.l2 = nn.Linear(200, 400)

    def forward(self, z_what):
        h = relu(self.l1(z_what))
        return sigmoid(self.l2(h))

decode = Decoder()

z_where_prior_loc = torch.tensor([3., 0., 0.])
z_where_prior_scale = torch.tensor([0.1, 1., 1.])
z_what_prior_loc = torch.zeros(50)
z_what_prior_scale = torch.ones(50)

def prior_step_sketch(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_loc.expand(1, -1),
                                      z_where_prior_scale.expand(1, -1))
                              .to_event(1))

    # Sample object code. This is a 50-dimensional vector.
    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.Normal(z_what_prior_loc.expand(1, -1),
                                     z_what_prior_scale.expand(1, -1))
                             .to_event(1))

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

    # Position/scale object within larger image.
    y = object_to_image(z_where, y_att)

    return y
```

Generating a single object

```
f expand_z_where(z_where):
    # Takes 3-dimensional vectors, and massages them into 2x3 matrices with elements like so:
    # [s,x,y] -> [[s,0,x],
    #             [0,s,y]]
    n = z_where.size(0)
    expansion_indices = torch.LongTensor([1, 0, 2, 0, 1, 3])
    out = torch.cat((torch.zeros([1, 1]).expand(n, 1), z_where), 1)
    return torch.index_select(out, 1, expansion_indices).view(n, 2, 3)

f object_to_image(z_where, obj):
    n = obj.size(0)
    theta = expand_z_where(z_where)
    grid = affine_grid(theta, torch.Size((n, 1, 50, 50)))
    out = grid_sample(obj.view(n, 1, 20, 20), grid)
    return out.view(n, 50, 50)
```

Generating an entire image

```
pyro.set_rng_seed(0)
def geom(num_trials=0):
    p = torch.tensor([0.5])
    x = pyro.sample('x{}'.format(num_trials), dist.Bernoulli(p))
    if x[0] == 1:
        return num_trials
    else:
        return geom(num_trials + 1)
```

```
def geom_prior(x, step=0):
    p = torch.tensor([0.5])
    i = pyro.sample('i{}'.format(step), dist.Bernoulli(p))
    if i[0] == 1:
        return x
    else:
        x = x + prior_step_sketch(step)
        return geom_prior(x, step + 1)
```

Vectorized mini-batches

```python
def prior_step_sketch(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_loc.expand(1, -1),
                                      z_where_prior_scale.expand(1, -1))
                              .to_event(1))

    # Sample object code. This is a 50-dimensional vector.
    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.Normal(z_what_prior_loc.expand(1, -1),
                                     z_what_prior_scale.expand(1, -1))
                             .to_event(1))

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

    # Position/scale object within larger image.
    y = object_to_image(z_where, y_att)

    return y
```

```python
def prior_step(n, t, prev_x, prev_z_pres):

    # Sample variable indicating whether to add this object to the output.

    # We multiply the success probability of 0.5 by the value sampled for this
    # choice in the previous step. By doing so we add objects to the output until
    # the first 0 is sampled, after which we add no further objects.
    z_pres = pyro.sample('z_pres_{}'.format(t),
                         dist.Bernoulli(0.5 * prev_z_pres)
                             .to_event(1))

    z_where = pyro.sample('z_where_{}'.format(t),
                          dist.Normal(z_where_prior_loc.expand(n, -1),
                                      z_where_prior_scale.expand(n, -1))
                              .mask(z_pres)
                              .to_event(1))

    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.Normal(z_what_prior_loc.expand(n, -1),
                                     z_what_prior_scale.expand(n, -1))
                             .mask(z_pres)
                             .to_event(1))

    y_att = decode(z_what)
    y = object_to_image(z_where, y_att)

    # Combine the image generated at this step with the image so far.
    x = prev_x + y * z_pres.view(-1, 1, 1)

    return x, z_pres
```

Vectorized mini-batches

```python
def prior(n):
    x = torch.zeros(n, 50, 50)
    z_pres = torch.ones(n, 1)
    for t in range(3):
        x, z_pres = prior_step(n, t, x, z_pres)
    return x
```

Guide

```python
rnn = nn.LSTMCell(2554, 256)

# Takes pixel intensities of the attention window to parameters (mean,
# standard deviation) of the distribution over the latent code,
# z_what.
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.l1 = nn.Linear(400, 200)
        self.l2 = nn.Linear(200, 100)

    def forward(self, data):
        h = relu(self.l1(data))
        a = self.l2(h)
        return a[:, 0:50], softplus(a[:, 50:])

encode = Encoder()

# Takes the guide RNN hidden state to parameters of
# the guide distributions over z_where and z_pres.
class Predict(nn.Module):
    def __init__(self, ):
        super(Predict, self).__init__()
        self.l = nn.Linear(256, 7)

    def forward(self, h):
        a = self.l(h)
        z_pres_p = sigmoid(a[:, 0:1]) # Squish to [0,1]
        z_where_loc = a[:, 1:4]
        z_where_scale = softplus(a[:, 4:]) # Squish to >0
        return z_pres_p, z_where_loc, z_where_scale

predict = Predict()
```

```python
predict = Predict()

def guide_step_improved(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_loc, z_where_scale = predict(h)

    z_pres = pyro.sample('z_pres_{}'.format(t),
                         dist.Bernoulli(z_pres_p * prev.z_pres)
                             .to_event(1))

    z_where = pyro.sample('z_where_{}'.format(t),
                          dist.Normal(z_where_loc, z_where_scale)
                              .to_event(1))

    # New. Crop a small window from the input.
    x_att = image_to_object(z_where, data)

    # Compute the parameter of the distribution over z_what
    # by passing the window through the encoder network.
    z_what_loc, z_what_scale = encode(x_att)

    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.Normal(z_what_loc, z_what_scale)
                             .to_event(1))

    return # values for next step
```

Data dependent baselines

```
bl_rnn = nn.LSTMCell(2554, 256)
bl_predict = nn.Linear(256, 1)

# Use an RNN to compute the baseline value. This network takes the
# input images and the values samples so far as input.
def baseline_step(x, prev):
    rnn_input = torch.cat((x,
                           prev.z_where.detach(),
                           prev.z_what.detach(),
                           prev.z_pres.detach()), 1)
    bl_h, bl_c = bl_rnn(rnn_input, (prev.bl_h, prev.bl_c))
    bl_value = bl_predict(bl_h) * prev.z_pres
    return bl_value, bl_h, bl_c
```

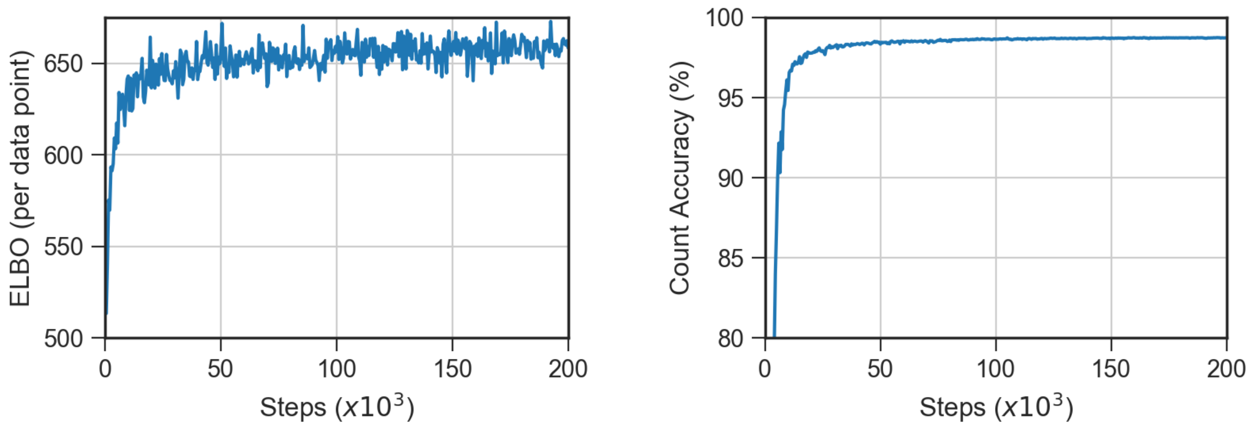
Data dependent baselines

```python
GuideState = namedtuple('GuideState', ['h', 'c', 'bl_h', 'bl_c', 'z_pres', 'z_where', 'z_what'
def initial_guide_state(n):
    return GuideState(h=torch.zeros(n, 256),
                      c=torch.zeros(n, 256),
                      bl_h=torch.zeros(n, 256),
                      bl_c=torch.zeros(n, 256),
                      z_pres=torch.ones(n, 1),
                      z_where=torch.zeros(n, 3),
                      z_what=torch.zeros(n, 50))

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_loc, z_where_scale = predict(h)

    # Here we compute the baseline value, and pass it to sample.
    baseline_value, bl_h, bl_c = baseline_step(data, prev)
    z_pres = pyro.sample('z_pres_{}'.format(t),
                         dist.Bernoulli(z_pres_p * prev.z_pres)
                             .to_event(1),
                         infer=dict(baseline=dict(baseline_value=baseline_value.squeeze(-1))))

    z_where = pyro.sample('z_where_{}'.format(t),
                          dist.Normal(z_where_loc, z_where_scale)
                              .mask(z_pres)
                              .to_event(1))

    x_att = image_to_object(z_where, data)

    z_what_loc, z_what_scale = encode(x_att)

    z_what = pyro.sample('z_what_{}'.format(t),
                         dist.Normal(z_what_loc, z_what_scale)
                             .mask(z_pres)
                             .to_event(1))

    return GuideState(h=h, c=c, bl_h=bl_h, bl_c=bl_c, z_pres=z_pres, z_where=z_where, z_what=z

def guide(data):
    # Register networks for optimization.
    pyro.module('rnn', rnn),
    pyro.module('predict', predict),
    pyro.module('encode', encode),
    pyro.module('bl_rnn', bl_rnn)
    pyro.module('bl_predict', bl_predict)

    with pyro.plate('data', data.size(0), subsample_size=64) as indices:
        batch = data[indices]
        state = initial_guide_state(batch.size(0))
        steps = []
        for t in range(3):
            state = guide_step(t, batch, state)
            steps.append(state)
        return steps
```

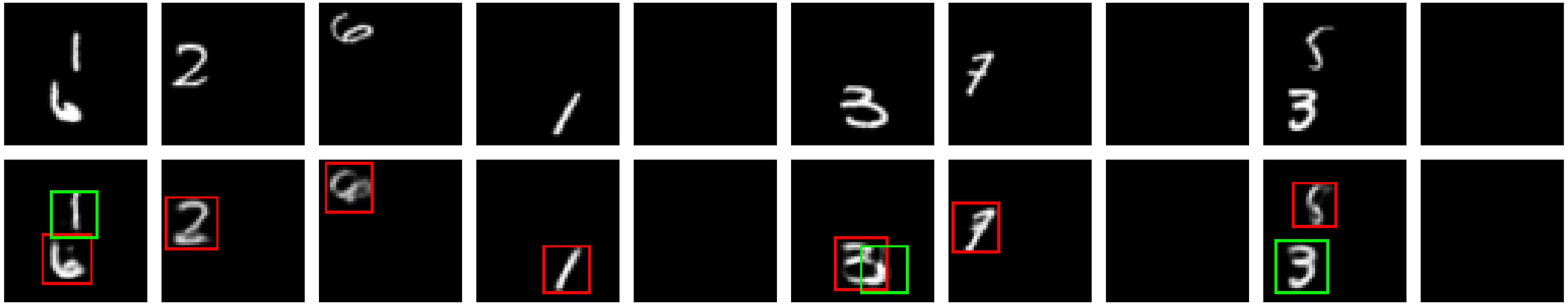
Putting it all together

```
data = mnist.view(-1, 50 * 50)

svi = SVI(model,
          guide,
          optim.Adam({'lr': 1e-4}),
          loss=TraceGraph_ELBO())

for i in range(5):
    loss = svi.step(data)
    print('i={}, elbo={:.2f}'.format(i, loss / data.size(0)))
```

Results



ELBO (per data point)
650
600
550
500
0
50
100
150
200
Steps ($x10^3$)
Count Accuracy (%)
100
95
90
85
80
0
50
100
150
200
Steps ($x10^3$)



Example - Rational Speech Act Framework

$$
\begin{array}{lcl}
L_1 & \textit{pragmatic listener} & P_{L_1}(s|u) \propto P_{S_1}(u|s) \cdot P(s) \\
\downarrow & & \\
S_1 & \textit{pragmatic speaker} & P_{S_1}(u|s) \propto \exp(\alpha U_{S_1}(u;s)) \\
\downarrow & & \\
L_0 & \textit{literal listener} & P_{L_0}(s|u) \propto [\![\mathrm{u}]\!](s) \cdot P(s)
\end{array}
$$

```python
@Marginal
def literal_listener(utterance):
    state = state_prior()
    factor("literal_meaning", 0. if meaning(utterance, state) else -999999.)
    return state
```

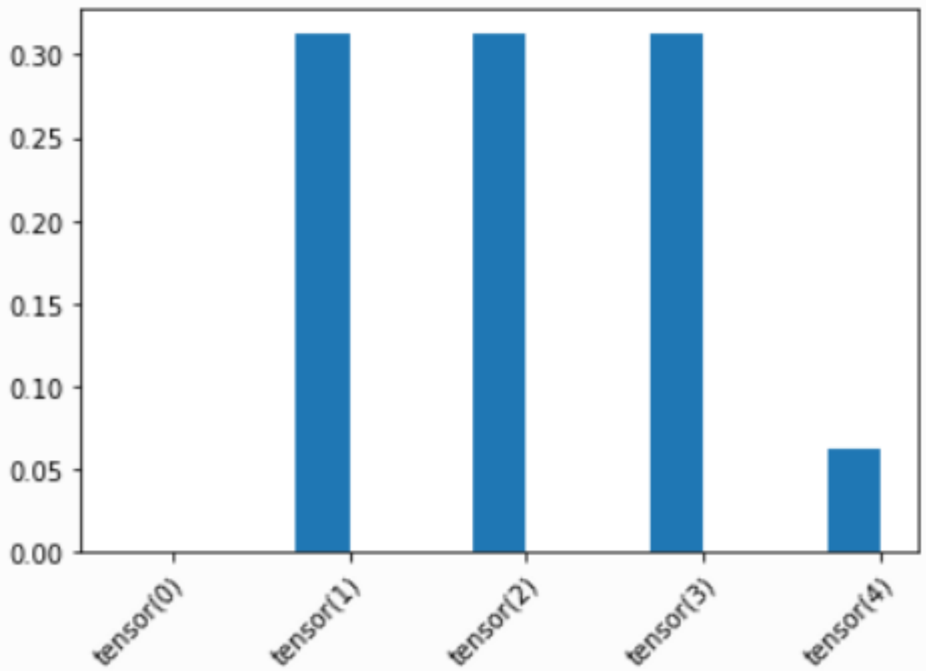
```python
@Marginal
def speaker(state):
    alpha = 1.
    with poutine.scale(scale=torch.tensor(alpha)):
        utterance = utterance_prior()
        pyro.sample("listener", literal_listener(utterance), obs=state)
    return utterance
```

```python
@Marginal
def pragmatic_listener(utterance):
    state = state_prior()
    pyro.sample("speaker", speaker(state), obs=utterance)
    return state
```

```python
total_number = 4

def state_prior():
    n = pyro.sample("state", dist.Categorical(probs=torch.ones(total_number+1) / total_numl
    return n

def utterance_prior():
    ix = pyro.sample("utt", dist.Categorical(probs=torch.ones(3) / 3))
    return ["none","some","all"][ix]
```

```python
meanings = {
    "none": lambda N: N==0,
    "some": lambda N: N>0,
    "all": lambda N: N==total_number,
}

def meaning(utterance, state):
    return meanings[utterance](state)
```

```
#silly plotting helper:
def plot_dist(d):
    support = d.enumerate_support()
    data = [d.log_prob(s).exp().item() for s in d.enumerate_support()]
    names = support

    ax = plt.subplot(111)
    width=0.3
    bins = map(lambda x: x-width/2,range(1,len(data)+1))
    ax.bar(bins,data,width=width)
    ax.set_xticks(map(lambda x: x, range(1,len(data)+1)))
    ax.set_xticklabels(names,rotation=45, rotation_mode="anchor", ha="right")

interp_dist = pragmatic_listener("some")
plot_dist(interp_dist)
```



Mini Pyro

- Effect Handlers (Poutine)

  library enables non-standard interpretations of Pyro primitives

  PYRO_STACK = []

- Parameters

  Unique names

  Play important role in stochastic variational inference

  PARAM_STORE = {}

Mini Pyro

- **Effect Handlers (Poutine)**

  library enables non-standard interpretations of Pyro primitives

  PYRO_STACK = []

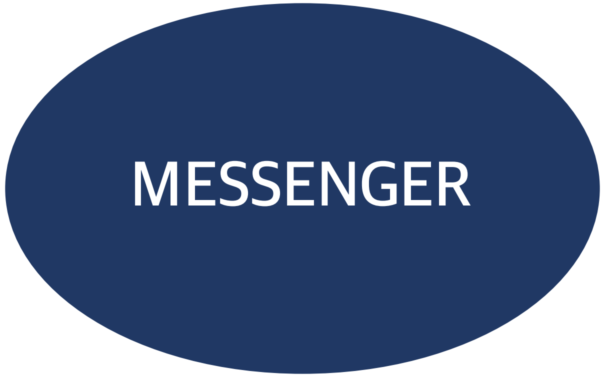
- Parameters

  Unique names

  Play important role in stochastic variational inference

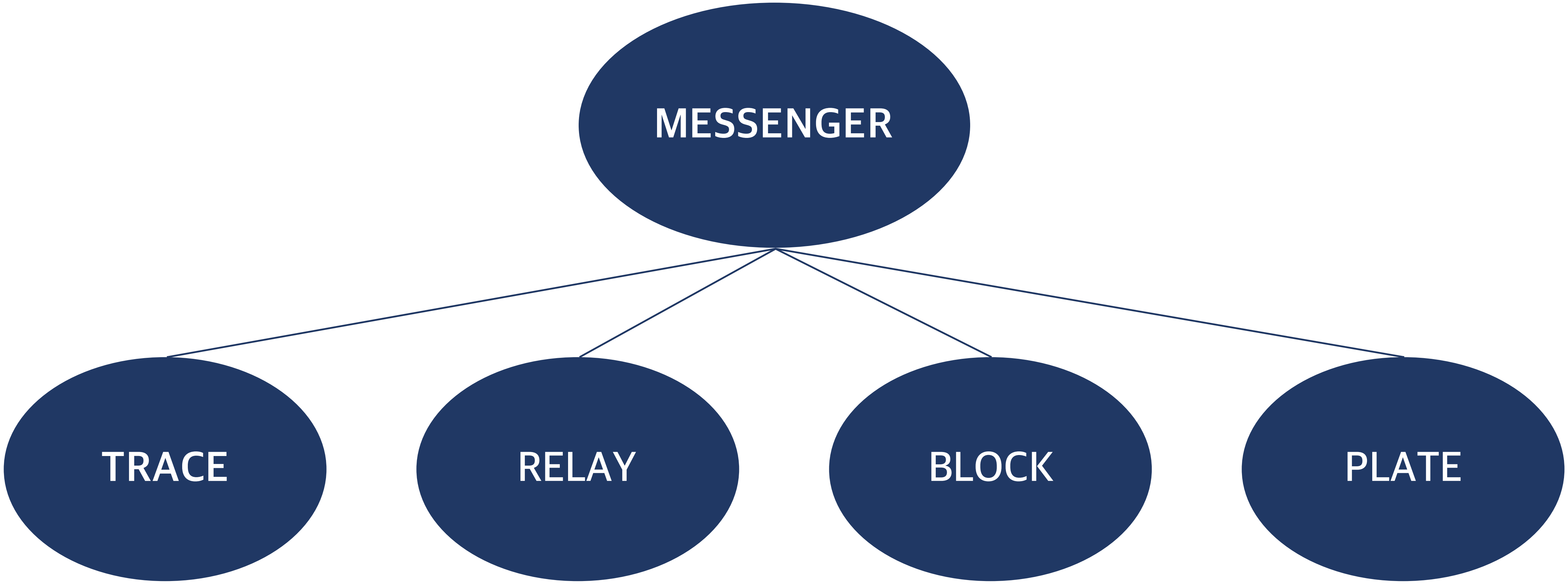
  PARAM_STORE = {}

Effect Handler




```
class Messenger(object) :
    def __init__(self, fn = None) :
        self.fn = fn

    def __enter__(self) :
        PYRO_STACK.append(self)

    def __exit__(self, *args, **kwargs) :
        assert PYRO_STACK [-1] is self
        PYRO_STACK.pop()

    def process_message(self, msg) :
        pass

    def postprocess_message(self, msg) :
        pass

    ...
```

Effect Handler



TRACE

```
class trace(Messenger) :
    def _enter_(self):
        super(trace, self)._enter_()
        self.trace = OrderedDict()
        return self.trace

    def postprocess_message(self, msg) :
        assert msg["name"] not in self.trace, "all sites must have unique name"
        self.trace[msge["name"]] = msg.copy()

    def get_trace(self, *args, **kwargs) :
        self(*args, **kwargs)
        return self.trace
```

REPLAY

```
class replay(Messenger) :
    def __init__(self, fn, guide_trace):
            self.guide_trace = guide_trace
            super(replay, self).__init__(fn)

    def process_message(self,  msg) :
            if msg[“name”] in self.guide_trace:
                msg[“value”] = self.guide_trace[msg[”name”]][“value”]
```

TRACE + REPLAY (e.g. ELBO)

```
def elbo(model, guide, *args, **kwargs) :
    guide_trace = trace(guide).get_trace(*args, **kwargs)

    model_trace = trace(replay(model, guide_trace)).get_trace(*args, **kwargs)

    elbo = 0

    for site in model_trace.values():
            if site["type"] == "sample":
                elbo = elbo + site["fn"].log_prob(site["value"]).sum()

    for site in guide_trace.values():
            if site["type"] == "sample":
                elbo = elbo - site["fn"].log_prob(site["value"]).sum()

    return -elbo
```

BLOCK

```
class block(Messenger) :
    def __init__(self, fn=None, hide_fn=lambda msg: True) :
            self.hide_fn = hide_fn
            super(block, self).__init__(fn)

    def process_message(self,  msg):
            if self.hide_fn(msg):
                msg["stop"] = True
```

Plate

```
class PlateMessenger(Messenger) :
    def _init_(self, fn, size, dim):
            assert dim < 0
            self.size = size
            self.dim = dim
            super (PlateMessenger, self)._init_(fn)

    def process_message(self, msg):
            if msg["type"] == "sample" :
                batch_shape = msg["fn"].batch_shape
                if len(batch_shape) < -self.dim or batch_shape[self.dim] != self.size:
                        batch_shape = [1] * (-self.dim - len(batch_shape)) + list(batch_shape)
                        batch_shape[self.dim] = self.size
                        msg["fn"] = msg["fn"].expand(torch.Size(batch_shape))

    def _iter_(self):
            return range(self, size)
```

Mini Pyro

- Effect Handlers (Poutine)

  library enables non-standard interpretations of Pyro primitives

  PYRO_STACK = []

- **Parameters**

  Unique names

  Play important role in stochastic variational inference

  PARAM_STORE = {}

Parameters

def param(name, init_value = None, constraint = torch.distributions.constraitns.real) :

```
def fn(init_value, constraint) :
    if name in PARAM_STORE :
        unconstrained_value, constraint = PARAM_STORE[name]
    else :
        ...
        with torch.no_grad() :
            constrained_value = init_value.detach()
            unconstrained_value = torch.distributions.transform_to(constraint).inv(constrained_value)
        ...
        PARAM_STORE[name] = unconstrained_value, constraint

    ….
    return constrained_value
```

Q&A