Deep Probabilistic Programming Languages: A Qualitative Study

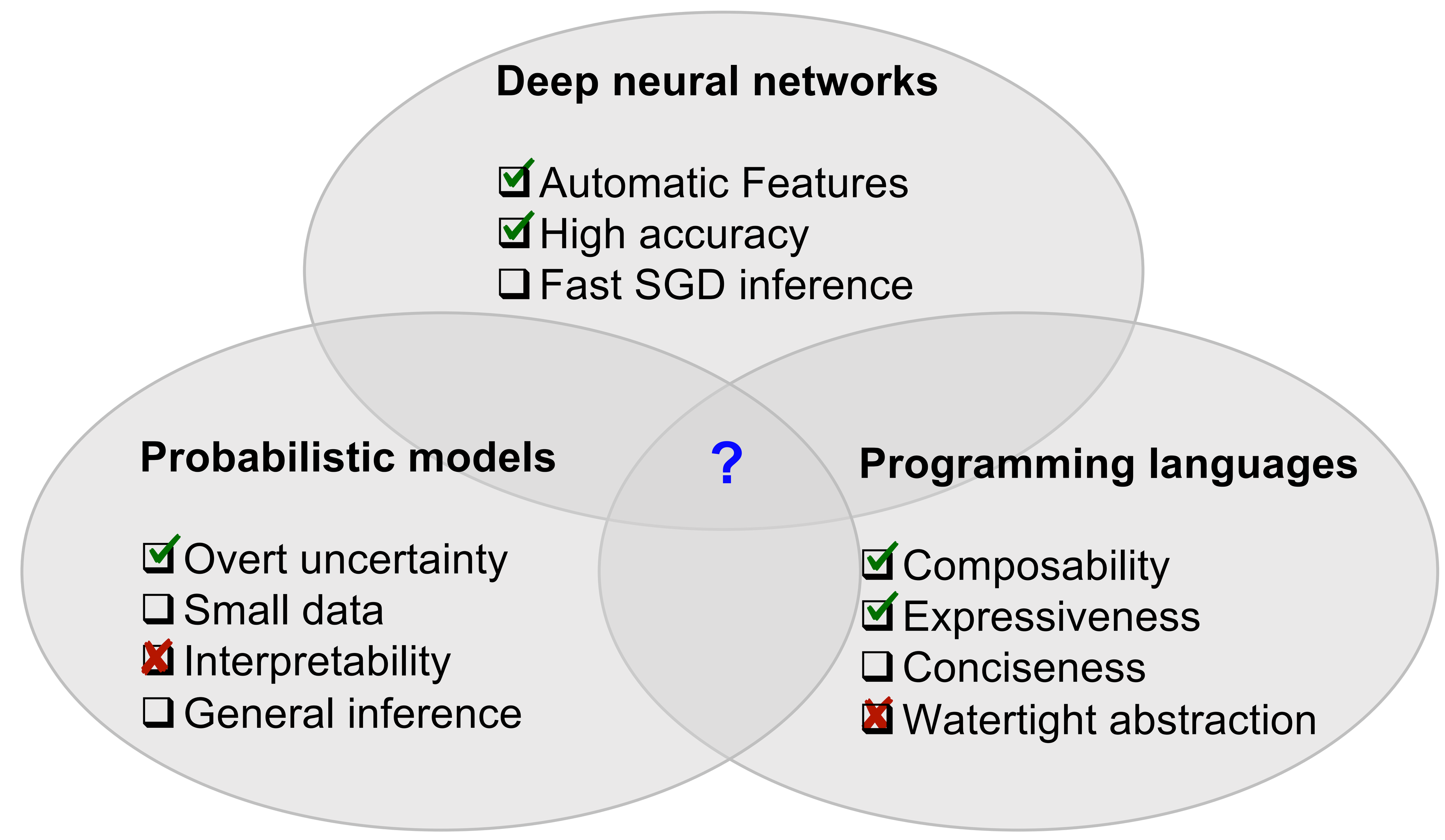
Guillaume Baudart, Martin Hirzel, Louis Mandel, *IBM Research*

Deep Probabilistic Programming Languages

DeepPPLs, which have emerged just recently, aim to combine the benefits of probabilistic programming and deep learning models.

These languages can be used to define:
- Probabilistic deep models: weight uncertainty in deep neural networks
- Deep probabilistic models: probabilistic models using deep learning

Goal: Compare and characterize DeepPPLs, focusing on two languages:
- Edward, based on TensorFlow (static computation graph)
- Pyro, based on PyTorch (dynamic computation graph)

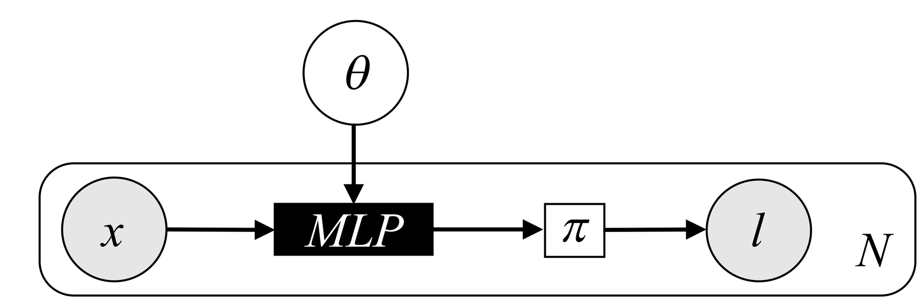


Probabilistic Deep Model

Example: Bayesian Multi-Layer Perceptron (MLP) with weight uncertainty
Lift network parameters to random variables

```python
# Model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.l1 = torch.nn.Linear(nx, nh)
        self.l2 = torch.nn.Linear(nh, ny)

    def forward(self, x):
        h = relu(self.l1(x.view((-1, nx)))))
        yhat = self.l2(h)
        return log_softmax(yhat)
mlp = MLP()

def model(x, y):
    priors = {'l1.weight': center_normal(nh, nx), 'l1.bias': center_normal(nh),
              'l2.weight': center_normal(ny, nh), 'l2.bias': center_normal(ny)}
    lifted_module = pyro.random_module("mlp", mlp, priors)()
    yhat = lifted_model(x)
    pyro.sample("obs", Categorical(logits=yhat), obs=y)

# Variational Inference
def guide(x, y):
    dists = {'l1.weight': rand_normal("W1", nh, nx), 'l1.bias': rand_normal("b1", nh),
             'l2.weight': rand_normal("W2", ny, nh),'l2.bias': rand_normal("b2", ny)}
    return pyro.random_module("mlp", mlp, dists)()

inference = SVI(model, guide, Adam({"lr": 0.01}), loss=Trace_ELBO())

# Predictions
def predict(x):
    sampled_models = [guide(None, None) for _ in range(args.num_samples)]
    yhats = [model(Variable(x)).data for model in sampled_models]
    mean = torch.mean(torch.stack(yhats), 0)
    return np.argmax(mean, axis=1)
```

(a) Pyro




```python
# Model
def mlp(theta, x):
    h = relu(tf.matmul(x, theta["W1"]) + theta["b1"])
    yhat = tf.matmul(h, theta["W2"]) + theta["b2"]
    return log_softmax(yhat)

theta = {'W1': center_normal(nx, nh), 'b1': center_normal(nh),
         'W2': center_normal(nh, ny), 'b2': center_normal(ny)}
x = tf.placeholder(tf.float32, [batch_size, nx])
l = tf.placeholder(tf.int32, [batch_size])
lhat = Categorical(logits=mlp(theta, x))

# Variational Inference
qtheta = {'W1': rand_normal(nx, nh), 'b1': rand_normal(nh),
          'W2': rand_normal(nh, ny), 'b2': rand_normal(ny)}
inference = ed.KLqp({theta["W1"]: qtheta["W1"], theta["b1"]: qtheta["b1"],
                     theta["W2"]: qtheta["W2"], theta["b2"]: qtheta["b2"]},
                    data={lhat: l})
# Predictions
def predict(x):
    theta_samples = [{"W1": qtheta["W1"].sample(), "b1": qtheta["b1"].sample(),
                      "W2": qtheta["W2"].sample(), "b2": qtheta["b2"].sample()}
                     for _ in range(args.num_samples)]
    yhats = [mlp(theta_samp, x).eval() for theta_samp in theta_samples]
    mean = np.mean(yhats, axis=0)
    return np.argmax(mean, axis=1)
```

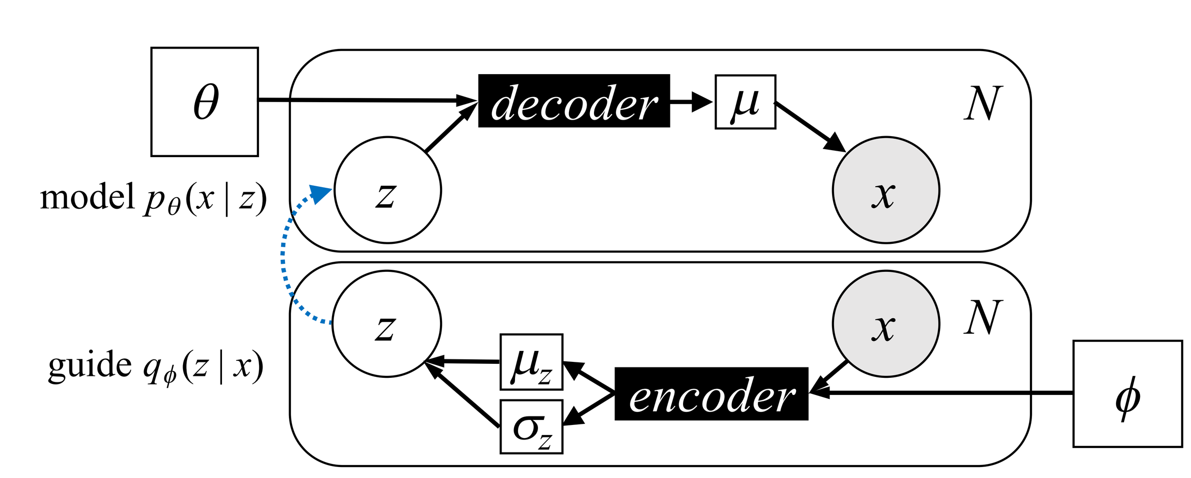
(b) Edward

Deep Probabilistic Model

Example: Variational Auto-Encoder (VAE). Unsupervised embedding learning.
Networks capture complex probabilistic dependencies with deep learning models

```python
# Model
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.lh = nn.Linear(nz, nh)
        self.lx = nn.Linear(nh, nx)
    def forward(self, z):
        return self.lx(relu(self.lh(z)))
decoder = Decoder()

def model(self, x):
    pyro.module("decoder", decoder)
    z = pyro.sample("latent", center_normal(batch_size, nz))
    logits = decoder.forward(z)
    pyro.sample("obs", Bernoulli(logits), obs=x.reshape(-1, nx))

# Inference Guide
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.lh = torch.nn.Linear(nx, nh)
        self.lz_mu = torch.nn.Linear(nh, nz)
        self.lz_sigma = torch.nn.Linear(nh, nz)
    def forward(self, x):
        hidden = relu(self.lh(x.view((-1, nx))))
        z_mu = self.lz_mu(hidden)
        z_sigma = softplus(self.lz_sigma(hidden))
        return z_mu, z_sigma
encoder = Encoder()

def guide(self, x):
    pyro.module("encoder", encoder)
    z_mu, z_sigma = encoder.forward(x)
    pyro.sample("latent", Normal(z_mu, z_sigma))

# Inference
inference = SVI(model, guide, Adam({"lr": 0.01}), loss=Trace_ELBO())
```

(a) Pyro




```python
# Model
X = tf.placeholder(tf.int32, [None, nx])
def decoder(theta, z):
    hidden = tf.nn.relu(tf.matmul(z, theta['Wh']) + theta['bh'])
    mu = tf.matmul(hidden, theta['Wy']) + theta['by']
    return mu

theta = {'Wh': rand_normal(nz, nh), 'bh': rand_normal(nh),
         'Wy': rand_normal(nh, nx), 'by': rand_normal(nx)}
z = center_normal(batch_size, nz)
logits = decoder(theta, z)
x = Bernoulli(logits=logits)

# Variational Inference
def encoder(phi, x):
    x = tf.cast(x, tf.float32)
    hidden = relu(tf.matmul(x, phi['Wh']) + phi['bh'])
    z_mu = tf.matmul(hidden, phi['Wy_mu']) + phi['by_mu']
    z_sigma = softplus(tf.matmul(hidden, phi['Wy_sigma']) + phi['by_sigma'])
    return z_mu, z_sigma

phi = {'Wh': rand_normal(nx, nh), 'bh': rand_normal(nh),
       'Wy_mu': rand_normal(nh, nz), 'by_mu': rand_normal(nz),
       'Wy_sigma': rand_normal(nh, nz), 'by_sigma': rand_normal(nz)}
z_mu, z_sigma = encoder(phi, X)
qz = Normal(loc=z_mu, scale=z_sigma)
inference = ed.KLqp({z: qz}, data={x: X})
```

(b) Edward

https://arxiv.org/abs/1804.06458