# Simple, Distributed, Accelerated Probabilistic Programming

Dustin Tran*, Matthew Hoffman†, Dave Moore†, Christopher Suter†,
Srinivas Vasudevan†, Alexey Radul†, Matthew Johnson*, Rif A. Saurous†,
*Google Brain, †Google Research

Google AI

## TL;DR

- Deep probabilistic programming provides a vision for accelerating deep learning research with probabilistic primitives.
- However, it limits research flexibility. It is also an open challenge to scale PPLs to >50M parameter models and multi-machines.
- We describe a simple approach for embedding probabilistic programming in a deep learning ecosystem. Name: Edward2.

There are only two ingredients: 1. **random variables** for specifying models; 2. **tracing** for manipulating models for computation.

## 1. Random Variables

```
def model():
  p = ed.Beta(1., 1., name="p")
  x = ed.Bernoulli(probs=p,
                   sample_shape=50,
                   name="x")
  return x
```

```
import neural_net_negative, neural_net_positive

def variational(x):
  eps = ed.Normal(0., 1., sample_shape=2)
  if eps[0] > 0:
    return neural_net_positive(eps[1], x)
  else:
    return neural_net_negative(eps[1], x)
```

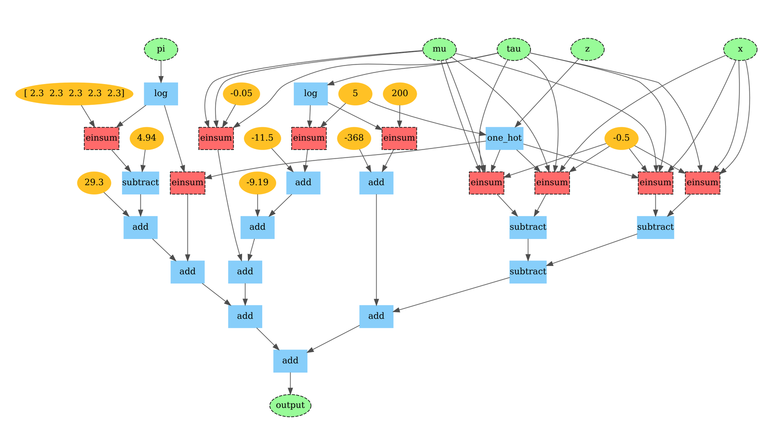
All computable probability distributions are Python functions (programs). Typically, it executes the generative process.
Programs compose Edward random variables. Random variables are TensorFlow Tensors augmented with distribution methods such as log_prob and sample.

## 2. Tracing

```
STACK = [lambda f, *a, **k: f(*a, **k)]

@contextmanager
def trace(tracer):
  STACK.append(tracer)
  yield
  STACK.pop()

def traceable(f):
  def f_wrapped(*a, **k):
    STACK[-1](f, *a, **k)
  return f_wrapped
```

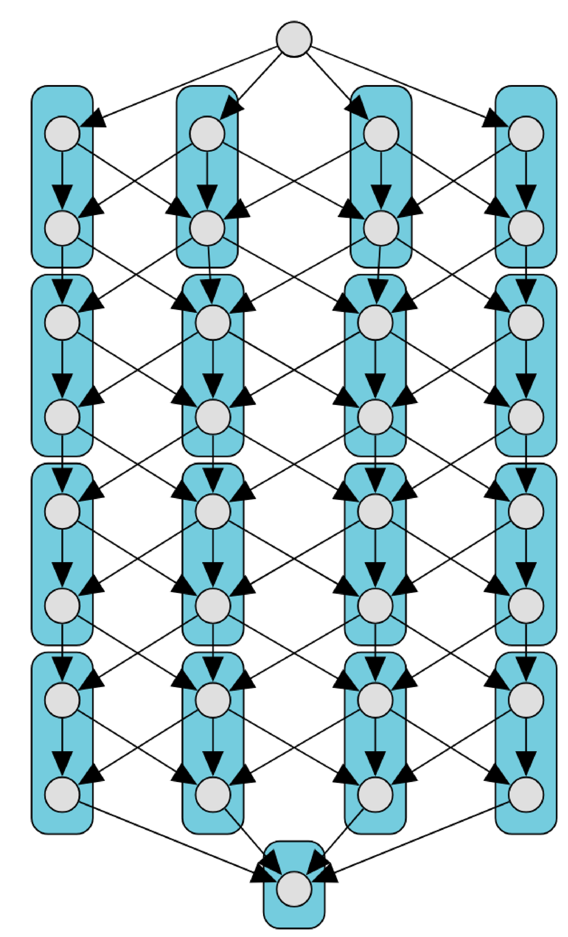


```
def make_log_joint_fn(model):
  def log_joint_fn(**model_kwargs):
    def tracer(rv_call, *args, **kwargs):
      name = kwargs.get("name")
      kwargs["value"] = model_kwargs.get(name
      rv = rv_call(*args, **kwargs)
      log_probs.append(tf.sum(rv.log_prob(rv)
      return rv
    log_probs = []
    with trace(tracer):
      model(**model_kwargs)
    return sum(log_probs)
  return log_joint_fn
```

```
def mutilate(model, **do_kwargs):
  def mutilated_model(*args, **kwargs):
    def tracer(rv_call, *args, **kwargs):
      name = kwargs.get("name")
      if name in do_kwargs:
        return do_kwargs[name]
      return rv_call(*args, **kwargs)
    with trace(tracer):
      return model(*args, **kwargs)
  return mutilated_model
```

Tracing wraps a language's primitive operations. A tracer intercepts control just before those operations are executed.
**Example: make_log_joint_fn**. Get density function given the generative process.
(This is core to probabilistic inference, e.g., MCMC and variational methods.)
**Example: mutilate**. Get model with causally intervened variables.
(This is core to causality, e.g., planning, counterfactuals, transfer.)

## Model-Parallel Variational Auto-Encoders



```
import SplitAutoregressiveFlow, masked_network
tfb = tf.contrib.distributions.bijectors

class DistributedAutoregressiveFlow(tfb.Bijector):
  def __init__(flow_size=[4]*8):
    self.flows = []
    for num_splits in flow_size:
      flow = SplitAutoregressiveFlow(masked_network, nur
      self.flows.append(flow)
    self.flows.append(SplitAutoregressiveFlow(masked_net
    super(DistributedAutoregressiveFlow, self).__init__(

  def _forward(self, x):
    for l, flow in enumerate(self.flows):
      with tf.device(tf.contrib.tpu.core(l//2)):
        x = flow.forward(x)
    return x

  def _inverse_and_log_det_jacobian(self, y):
    ldj = 0.
    for l, flow in enumerate(self.flows[::-1]):
      with tf.device(tf.contrib.tpu.core(l//2)):
        y, new_ldj = flow.inverse_and_log_det_jacobian(y
        ldj += new_ldj
    return y, ldj
```

```
import upsample, compressor

def prior():
  """Uniform noise to 8-bit latent, [u1,...,u(T/2)] -> [z1,...,z(T/2)]"""
  dist = ed.Independent(ed.Uniform(low=tf.zeros([batch_size, T/2])))
  return ed.TransformedDistribution(dist, DistributedAutoregressiveFlow(flow_size))

def decoder(z):
  """Uniform noise + latent to 16-bit audio, [u1,...,uT], [z1,...,z(T/2)] -> [x1,...,xT]""
  dist = ed.Independent(ed.Uniform(low=tf.zeros([batch_size, T])))
  dist = ed.TransformedDistribution(dist, tfb.Affine(shift=upsample(z)))
  return ed.TransformedDistribution(dist, DistributedAutoregressiveFlow(flow_size))

def encoder(x):
  """16-bit audio to 8-bit latent, [x1,...,xT] -> [z1,...,z(T/2)]"""
  loc, log_scale =  tf.split(compressor(x), 2, axis=-1)
  return ed.Normal(loc=loc, scale=tf.exp(log_scale))
```

With low-level flexibility, Edward2 lets you specify communication primitives for model-parallel computation.

## Image Transformer

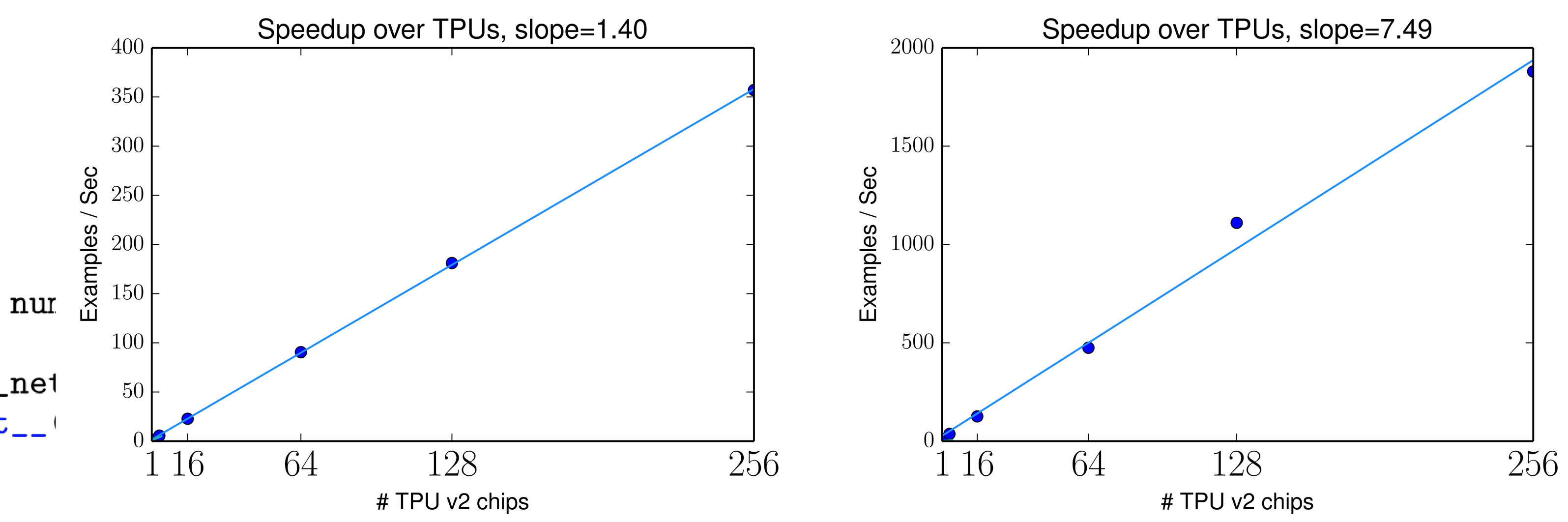
```
import get_channel_embeddings, add_positional_embedding_nd, local_attention_1d

def image_transformer(inputs, hparams):
  x = get_channel_embeddings(3, inputs, hparams.hidden_size)
  x = tf.reshape(x, [-1, 32*32*3, hparams.hidden_size])
  x = tf.pad(x, [[0, 0], [1, 0], [0, 0]])[:, :-1, :]  # shift pixels right
  x = add_positional_embedding_nd(x, max_length=32*32*3+3)
  x = tf.nn.dropout(x, keep_prob=0.7)
  for _ in range(hparams.num_layers):
    y = local_attention_1d(x, hparams, attention_type="local_mask_right",
                           q_padding="LEFT", kv_padding="LEFT")
    x = tf.contrib.layers.layer_norm(tf.nn.dropout(y, keep_prob=0.7) + x, begin_norm_axis=-1
    y = tf.layers.dense(x, hparams.filter_size, activation=tf.nn.relu)
    y = tf.layers.dense(y, hparams.hidden_size, activation=None)
    x = tf.contrib.layers.layer_norm(tf.nn.dropout(y, keep_prob=0.7) + x, begin_norm_axis=-1
  logits = tf.layers.dense(x, 256, activation=None)
  return ed.Categorical(logits=logits).log_prob(inputs)

loss = -tf.reduce_sum(image_transformer(inputs, hparams))  # inputs has shape [batch,32,32,3
train_op = tf.contrib.tpu.CrossShardOptimizer(tf.train.AdamOptimizer()).minimize(loss)
```

Most PPLs focus on a unifying model representation (e.g., generative process). In Edward2, you can use other represenations.

## High-Quality Image Generation




**(left)** VQ-VAE on 64x64 ImageNet. 6-layer Image Transformer prior; 4-layer conv/deconv encoder/decoder.
**(right)** Image Transformer on 256x256 CelebA-HQ. 5 layers.

## Learning to Learn by Variational Inference by Gradient Descent

```
import model, variational, align, x

def train(precond):
  def loss_fn(x):
    qz = variational(x)
    log_joint_fn = make_log_joint_fn(model)
    kwargs = {align[rv.name]: rv
              for rv in toposort(qz)}
    energy = log_joint_fn(x=x, **kwargs)
    entropy = sum([tf.reduce_sum(rv.entropy())
                   for rv in toposort(qz)])
    return -energy - entropy

  grad_fn = tfe.implicit_gradients(loss_fn)
  optimizer = tf.train.AdamOptimizer(0.1)
  for _ in range(500):
    grads = tf.tensordot(precond, grad_fn(x), [[1], [0]])
    optimizer.apply_gradients(grads)
  return loss_fn(x)
```

```
grad_fn = tfe.gradients_function(train)
optimizer = tf.train.AdamOptimizer(0.1)
for _ in range(100):
  optimizer.apply_gradients(grad_fn())
```

In Edward2, "inference algorithms" are simply numerical operations. You can take, e.g., gradients through them for flexible research.

## No-U-Turn Sampler

| System | Runtime (ms) |
|---|---|
| Stan (CPU) | 201.0 |
| PyMC3 (CPU) | 74.8 |
| Handwritten TF (CPU) | 66.2 |
| Edward2 (CPU) | 68.4 |
| Handwritten TF (1 GPU) | **9.5** |
| **Edward2 (1 GPU)** | **9.7** |
| **Edward2 (8 GPU)** | **2.3** |

100x speedup over Stan (CPU). 37x over PyMC3 (CPU). Negligible overhead over handwritten TensorFlow code.

# Where are we going next? Ask!

[1] Goodman, N. D. and Stuhlmüller, A. (2014). The design and implementation of probabilistic programming languages.
[2] Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. (2017). Deep probabilistic programming. In *International Conference on Learning Representations*.