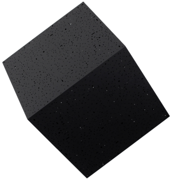
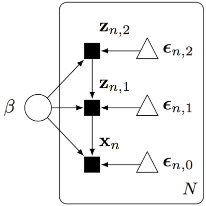
What Might Deep Learners Learn From Probabilistic Programming?

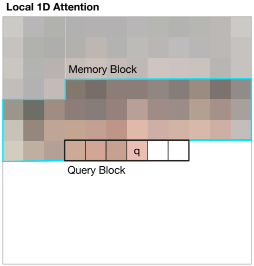


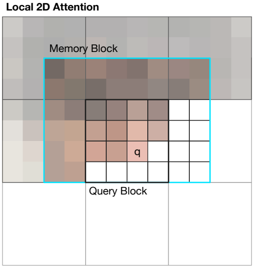
Dustin Tran
Google Brain

Interested in research for scientific applications?

That's not this talk.







Generation & compression of 10M colored 32x32 images

[Tran+ 2017; Parmar+ 2018]

23 Million Parameters	340 Million Parameters
title: "Pescadores Weekly" length: 2000 Pescadores Weekly Inc (organized in 1978 as the Pescadores Weekly) is the official Weekly organizing body of science. It was established in 1978 atop the city of the center of the City of Gravity. In 1978 it became the United States National Museum of Science and Design. The growing Berklee Plant represents the physical community of typical worldwide plants. Pescadores is named after the lord of the University of the Gravity of Drama, an interest he established in 1989 of the West Bird Plant. In the fall of 1978 the Faxophonius Institute, with the Today Natural History Foundation, published by the University of Georgia, and Anubis (Yale University). ==History== Pescadores provided classic books on science and other philosophical sciences such as geoscience that accomplished science-fiction in Europe during 1976–1977, being the first university campus of the University of New Zealand. …	title: "Pescadores Weekly" length: 2000 The "Pescadores Weekly" is the only daily newspaper in the Fort Hood metropolitan area, although the entire population of Fort Hood is majority white in population. Due to its not-for-profit focus as a newspaper, Carehead reporters and Hometime have expressed the need for a feature to focus on Fort Hood and its inner-city population. According to the Pescadores Weekly Editors' annual survey, the "Pescadores Weekly"{{'s}} circulation was the largest in Texas, surpassing the "Evening Star Sans Monthly" made in 1992. By contrast, Terry Truehead's "The Best of Fort Hood" daily was the fourth-fastest in Texas (behind rival "AfterEllena" and "Expensive Ellena", topping the chart thirteen years in a row) and expanded by two thirds during the 2002-2003 Edition. ==Political affiliation== The newspaper endorsed John McCain's majoritarian, Jack Abramoff, at the 2004 GOP convention. ...

Scaling up fundamental language models

[Liu+ 2018; Shazeer+ 2018]

Inference in a probabilistic program

```
(trace, weight) = query(program, args, observations)
```

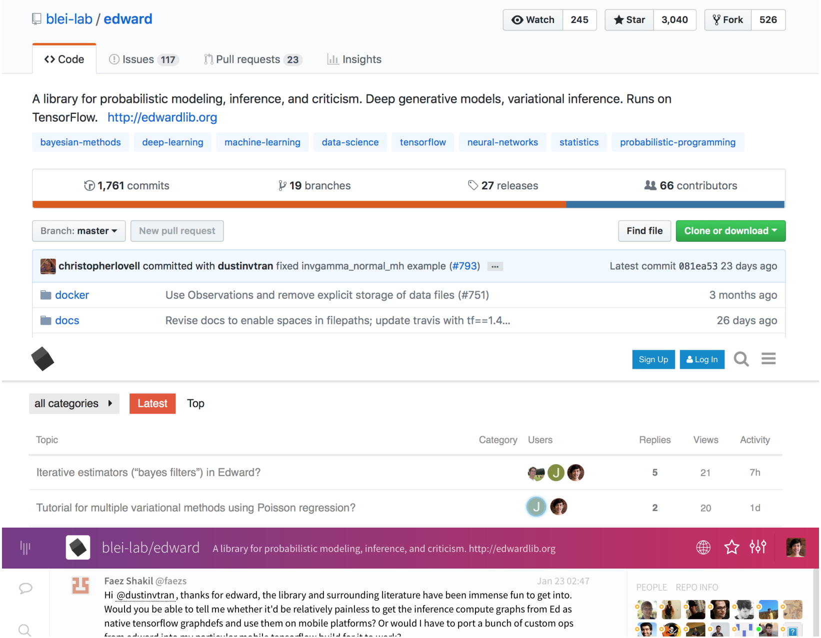
$\mathbf{x}$ ξ $\mathcal{P}$ α $\mathbf{y}$

[Tenenbaum+Mansinghka NIPS 2017 tutorial]

The Myth of Probabilistic Programming

Programming is infeasible if a core operation in the language is NP-hard.

For high-dimensional problems + modern probabilistic models, we haven't solved automated inference.


blei-lab / edward
Watch 245
Star 3,040
Fork 526
Code
Issues 117
Pull requests 23
Insights
A library for probabilistic modeling, inference, and criticism. Deep generative models, variational inference. Runs on TensorFlow. http://edwardlib.org
bayesian-methods
deep-learning
machine-learning
data-science
tensorflow
neural-networks
statistics
probabilistic-programming
1,761 commits
19 branches
27 releases
66 contributors
Branch: master
New pull request
Find file
Clone or download
christopherlovell committed with dustinvtran fixed invgamma_normal_mh example (#793)
Latest commit 081ea53 23 days ago
docker
Use Observations and remove explicit storage of data files (#751)
3 months ago
docs
Revise docs to enable spaces in filepaths; update travis with tf==1.4...
26 days ago
Sign Up
Log In
all categories
Latest
Top
Topic
Category
Users
Replies
Views
Activity
Iterative estimators ("bayes filters") in Edward?
5
21
7h
Tutorial for multiple variational methods using Poisson regression?
2
20
1d
blei-lab/edward
A library for probabilistic modeling, inference, and criticism. http://edwardlib.org
Faez Shakil @faezs
Jan 23 02:47
Hi @dustinvtran, thanks for edward, the library and surrounding literature have been immense fun to get into. Would you be able to tell me whether it'd be relatively painless to get the inference compute graphs from Ed as native tensorflow graphdefs and use them on mobile platforms? Or would I have to port a bunch of custom ops
PEOPLE
REPO INFO

Edward

Failure Modes

- **Inference is monolothic**. The average workflow requires understanding a new ecosystem, closed under its own compositions.
- **Can't it go faster?** Edward was not designed with TPUs and multiple machines in mind.

Some Iteration of Edward

Random Variables Are All You Need

Edward2 reifies any computable probability distribution as a Python function. Inputs to the program represent values the distribution conditions on.

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

```
import neural_net_negative, neural_net_positive

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

Tracing

A tracer from AD wraps the language's primitive operations. The tracer intercepts control just before those operations are executed.

Edward2 applies tracing in order to perform user-programmable manipulations.

```
INTERCEPTOR_STACK = [lambda f, *args, **kwargs: f(*args, **kwargs)]

@contextmanager
def interception(interceptor):
  INTERCEPTOR_STACK.append(interceptor)
  yield
  INTERCEPTOR_STACK.pop()

def interceptable(func):
  def func_wrapped(*args, **kwargs):
    INTERCEPTOR_STACK[-1](func, *args, **kwargs)
  return func_wrapped
```

Example: Latent Dirichlet Allocation

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import functools
import os

from absl import flags
import numpy as np
import scipy.sparse
from six.moves import cPickle as pickle
from six.moves import urllib
import tensorflow as tf

from tensorflow_probability import edward2 as ed


flags.DEFINE_float(
    "learning_rate", default=3e-4, help="Learning rate.")
flags.DEFINE_integer(
    "max_steps", default=180000, help="Number of training steps to run.")
flags.DEFINE_integer(
    "num_topics",
    default=50,
    help="The number of topics.")
flags.DEFINE_list(
    "layer_sizes",
    default=["300", "300", "300"],
    help="Comma-separated list denoting hidden units per layer in the encoder
flags.DEFINE_string(
    "activation",
    default="relu",
    help="Activation function for all hidden layers.")
flags.DEFINE_integer(
    "batch_size",
    default=32,
    help="Batch size.")
flags.DEFINE_float(
    "prior_initial_value", default=0.7, help="The initial value for prior.")
flags.DEFINE_integer(
    "prior_burn_in_steps",
    default=120000,
    help="The number of training steps with fixed prior.")
flags.DEFINE_string(
    "data_dir",
    default=os.path.join(os.getenv("TEST_TMPDIR", "/tmp"), "lda/data"),
    help="Directory where data is stored (if using real data).")
flags.DEFINE_string(
    "model_dir",
    default=os.path.join(os.getenv("TEST_TMPDIR", "/tmp"), "lda/"),
    help="Directory to put the model's fit.")
flags.DEFINE_integer(
    "viz_steps", default=10000, help="Frequency at which save visualizations.
flags.DEFINE_bool("fake_data", default=False, help="If true, uses fake data.'
flags.DEFINE_bool(
    "delete_existing",
    default=False,
    help="If true, deletes existing directory.")

FLAGS = flags.FLAGS


def _clip_dirichlet_parameters(x):
  """Clips the Dirichlet parameters to the numerically stable KL region."""
  return tf.clip_by_value(x, 1e-3, 1e3)
```

```python
def latent_dirichlet_allocation(concentration, topics_words):
  topics = ed.Dirichlet(concentration=concentration, name="topics")
  word_probs = tf.matmul(topics, topics_words)
  # The observations are bags of words and therefore not one-hot. However,
  # log_prob of OneHotCategorical computes the probability correctly in
  # this case.
  bag_of_words = ed.OneHotCategorical(probs=word_probs, name="bag_of_words")
  return bag_of_words


def make_lda_variational(activation, num_topics, layer_sizes):
  encoder_net = tf.keras.Sequential()
  for num_hidden_units in layer_sizes:
    encoder_net.add(tf.keras.layers.Dense(
        num_hidden_units, activation=activation,
        kernel_initializer=tf.glorot_normal_initializer()))
  encoder_net.add(tf.keras.layers.Dense(
      num_topics, activation=tf.nn.softplus,
      kernel_initializer=tf.glorot_normal_initializer()))

  def lda_variational(bag_of_words):
    concentration = _clip_dirichlet_parameters(encoder_net(bag_of_words))
    return ed.Dirichlet(concentration=concentration, name="topics_posterior")

  return lda_variational


def model_fn(features, labels, mode, params, config):
  del labels, config

  # Set up the model's learnable parameters.
  logit_concentration = tf.get_variable(
      "logit_concentration",
      shape=[1, params["num_topics"]],
      initializer=tf.constant_initializer(
          _softplus_inverse(params["prior_initial_value"])))
  concentration = _clip_dirichlet_parameters(
      tf.nn.softplus(logit_concentration))

  num_words = features.shape[1]
  topics_words_logits = tf.get_variable(
      "topics_words_logits",
      shape=[params["num_topics"], num_words],
      initializer=tf.glorot_normal_initializer())
  topics_words = tf.nn.softmax(topics_words_logits, axis=-1)

  # Compute expected log-likelihood. First, sample from the variational
  # distribution; second, compute the log-likelihood given the sample.
  lda_variational = make_lda_variational(
      params["activation"],
      params["num_topics"],
      params["layer_sizes"])
  with ed.tape() as variational_tape:
    _ = lda_variational(features)

  with ed.tape() as model_tape:
    with ed.interception(
        make_value_setter(topics=variational_tape["topics_posterior"])):
      posterior_predictive = latent_dirichlet_allocation(concentration,
                                                         topics_words)

  log_likelihood = posterior_predictive.distribution.log_prob(features)
  tf.summary.scalar("log_likelihood", tf.reduce_mean(log_likelihood))

  # Compute the KL-divergence between two Dirichlets analytically.
  # The sampled KL does not work well for "sparse" distributions
```

Example: Latent Dirichlet Allocation


```python
    # Ensure that the KL is non-negative (up to a very small slack).
    # Negative KL can happen due to numerical instability.
                          rt_greater(kl, -1e-3, message="kl")]):

    elbo = log_likelihood - kl
    avg_elbo = tf.reduce_mean(elbo)
    tf.summary.scalar("elbo", avg_elbo)
    loss = -avg_elbo

    # Perform variational inference by minimizing the -ELBO.
    global_step = tf.train.get_or_create_global_step()
    optimizer = tf.train.AdamOptimizer(params["learning_rate"])

    # This implements the "burn-in" for prior parameters (see Appendix D of [2]
    # For the first prior_burn_in_steps steps they are fixed, and then trained
    # jointly with the other parameters.
    grads_and_vars = optimizer.compute_gradients(loss)
    grads_and_vars_except_prior = [
        x for x in grads_and_vars if x[1] != logit_concentration]

    def train_op_except_prior():
      return optimizer.apply_gradients(
          grads_and_vars_except_prior,
          global_step=global_step)

    def train_op_all():
      return optimizer.apply_gradients(
          grads_and_vars,
          global_step=global_step)

    train_op = tf.cond(
        global_step < params["prior_burn_in_steps"],
        true_fn=train_op_except_prior,
        false_fn=train_op_all)

    # The perplexity is an exponent of the average negative ELBO per word.
    words_per_document = tf.reduce_sum(features, axis=1)
    log_perplexity = -elbo / words_per_document
    tf.summary.scalar("perplexity", tf.exp(tf.reduce_mean(log_perplexity)))
    (log_perplexity_tensor, log_perplexity_update) = tf.metrics.mean(
        log_perplexity)
    perplexity_tensor = tf.exp(log_perplexity_tensor)

    # Obtain the topics summary. Implemented as a py_func for simplicity.
    topics = tf.py_func(
        functools.partial(get_topics_strings, vocabulary=params["vocabulary"]),
        [topics_words, concentration], tf.string, stateful=False)
    tf.summary.text("topics", topics)

    return tf.estimator.EstimatorSpec(
        mode=mode,
        loss=loss,
        train_op=train_op,
        eval_metric_ops={
            "elbo": tf.metrics.mean(elbo),
            "log_likelihood": tf.metrics.mean(log_likelihood),
            "kl": tf.metrics.mean(kl),
            "perplexity": (perplexity_tensor, log_perplexity_update),
            "topics": (topics, tf.no_op()),
        },
    )


  def main(argv):
    del argv  # unused
```

```python
    if FLAGS.delete_existing and tf.gfile.Exists(FLAGS.model_dir):
      tf.logging.warn("Deleting old log directory at {}".format(FLAGS.model_dir))
      tf.gfile.DeleteRecursively(FLAGS.model_dir)
    tf.gfile.MakeDirs(FLAGS.model_dir)

    if FLAGS.fake_data:
      train_input_fn, eval_input_fn, vocabulary = build_fake_input_fns(
          FLAGS.batch_size)
    else:
      train_input_fn, eval_input_fn, vocabulary = build_input_fns(
          FLAGS.data_dir, FLAGS.batch_size)
    params["vocabulary"] = vocabulary

    estimator = tf.estimator.Estimator(
        model_fn,
        params=params,
        config=tf.estimator.RunConfig(
            model_dir=FLAGS.model_dir,
            save_checkpoints_steps=FLAGS.viz_steps,
        ),
    )

    for _ in range(FLAGS.max_steps // FLAGS.viz_steps):
      estimator.train(train_input_fn, steps=FLAGS.viz_steps)
      eval_results = estimator.evaluate(eval_input_fn)
      # Print the evaluation results. The keys are strings specified in
      # eval_metric_ops, and the values are NumPy scalars/arrays.
      for key, value in eval_results.items():
        print(key)
        if key == "topics":
          # Topics description is a np.array which prints better row-by-row.
          for s in value:
            print(s)
        else:
          print(str(value))
        print("")
      print("")


  if __name__ == "__main__":
    tf.app.run()
```

Mesh TensorFlow

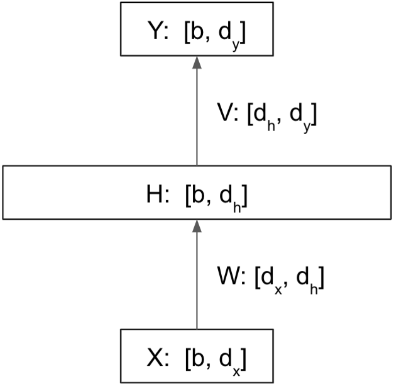
TPU Data Parallelism

- Parameters replicated on every core.
- Batch split between cores.
- Sum (allreduce) parameter gradients. (very efficient on locally-connected networks such as TPUs)

TPU Data Parallelism

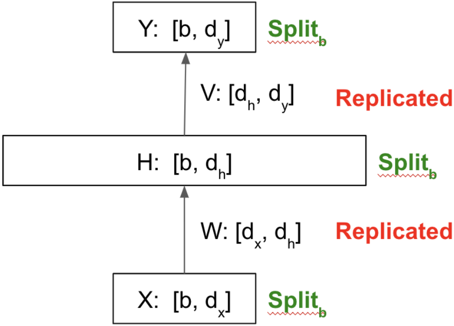
- Universal (any model/cluster)
- Fast to compile (SIMD)
- Full Utilization
- Allreduce is fast on any locally-connected network
- **All parameters must fit on one core.**

Example: Perceptron

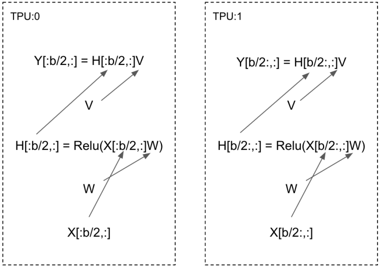


Example: Perceptron

$$Y = (H = \mathrm{Relu}(XW_1))V$$

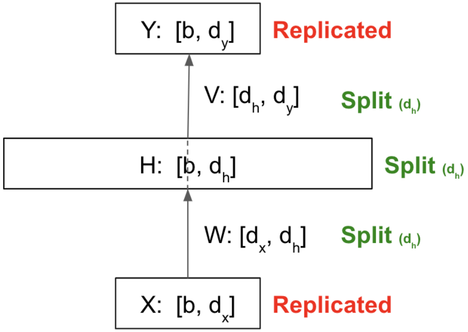


Data-Parallelism: Split dimension “b”

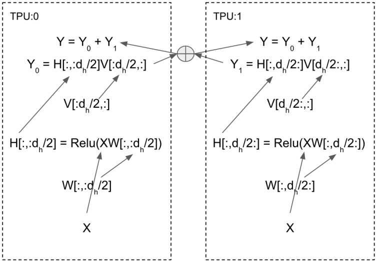


Example: Perceptron

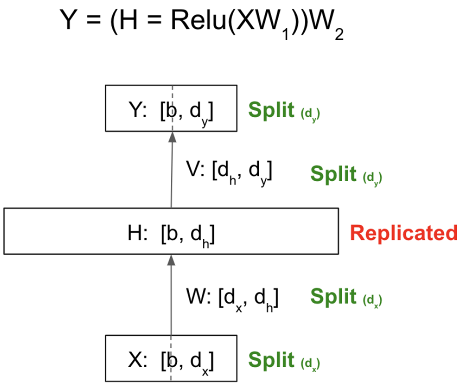
$$Y = (H = \mathrm{Relu}(XW_1))W_2$$

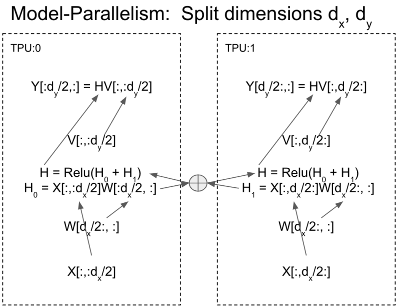


Model-Parallelism: Split dimension "d_h"



Example: Perceptron





Example: High-Quality Image Generation

50M+ parameter models (Image Transformer, VQVAE) on high-resolution images. Data parallelism.

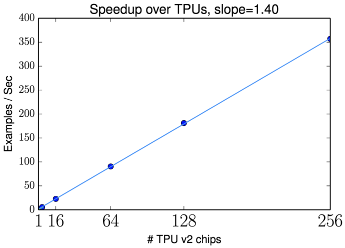


Figure 14: Vector-Quantized VAE on 64x64 ImageNet.

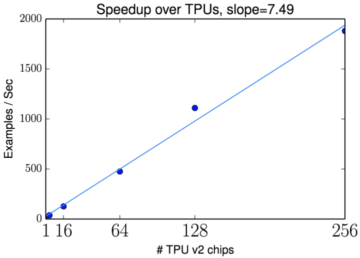


Figure 15: Image Transformer on 256x256 CelebA-HQ.

Edward2 achieves an optimal linear scaling from 1 to 256 TPUv2 chips.

[Tran+ 2018]

Example: NUTS

Time per leapfrog step for No-U-Turn Sampler (NUTS) on Bayesian logistic regression. Covertype, 500K data points, 54 features.

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**

Edward2 (GPU) achieves up to a 100x speedup over Stan and 7x over PyMC3. Dynamism is not possible in Edward 1.0.

Edward2 has negligible overhead over handwritten TF.

[Tran+ 2018]

Example: Language Modeling

d_ff	heads	Parameters (in Billions)	Billion-Word Benchmark Word-Perplexity (bits/dim)	Wikipedia Subword-Perplexity (bits/dim)
8192	8	0.22	30.7	8.47
16384	16	0.37	28.0	7.92
32768	32	0.67	26.0	7.47
65516	64	1.28	24.6	6.97
131072	128	2.48	23.6	6.64
262144	256	4.90	**23.1**	**6.41**

Transformer from 20M to 3B parameter models. Model parallelism. Roughly 50% utilization.

[Shazeer+ 2018]

Example: Machine Translation

d_ff	heads	Parameters (in Billions)	WMT'14 English-German BLEU	WMT'14 English-French BLEU
4096	8	0.24	28.6	41.7
8192	16	0.42	**29.1**	43.0
16384	32	0.77	28.9	43.2
32768	64	1.48	-	**43.7**
65516	128	2.89	-	**43.7**
4096	16	0.21	28.4	41.8 (Vaswani et. al)

Transformer from 20M to 3B parameter models. Model parallelism. Roughly 50% utilization.

[Shazeer+ 2018]

Summary

1. Designing probabilistic systems for deep learning requires careful consideration about what's really brought to the table.
2. Our attempts pushed on what we think are the core elements.

Current directions.

1. We're advancing fundamental understandings of generative models and Bayesian neural networks.
2. We're pushing Mesh TensorFlow to trillion-parameter language models, new architectures, and model-parallel VAEs.

References

Systems

- Edward2: Simple, Distributed, Accelerated. NIPS 2018.
- Deep Learning for Supercomputers. NIPS 2018.
- Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. NIPS 2018.

Methods

- Image Transformer. ICML 2018.
- Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches. ICLR 2018.
- Reliable uncertainty estimates in deep neural networks using noise contrastive priors. arXiv:1807.09289 2018.

AutoConj: find and exploit exponential family structure without a DSL

Matthew D. Hoffman*, Matthew J. Johnson*, Dustin V. Tran

Google AI · Google brain

TL;DR Write models in regular Python+Numpy with no mini-language, get exponential family structure-exploiting inference algorithms.

Why? Exploiting exponential family structure when it exists is labor-intensive, even for experts, which limits how we design new models and try new hybrid inference strategies (e.g. SVAEs). It's like neural nets before autodiff.

What is the autodiff for exponential family inference? AutoConj!

DSL? As with autodiff, don't want to be locked-in to a mini-language:

- New inference algorithms? Model classes?
- Optimization libraries? Automatic differentiation? Viz.?
- Compile to accelerators, distributed computing?

Need a system in native Python, and composable with others.

Background: exponential families

Define a probability model via a **statistic function** $t(x)$

$$p(x;\eta) = \exp\left\{\langle \eta,\, t(x)\rangle - \mathcal{A}(\eta)\right\}, \qquad \mathcal{A}(\eta) \triangleq \log \int \exp\left\{\langle \eta,\, t(x)\rangle\right\} \nu(\mathrm{d}x),$$

Derivatives of the log partition function $\mathcal{A}(\eta)$ yield cumulants

$$\nabla \mathcal{A}(\eta) = \mathbb{E}\left[t(x)\right], \qquad \nabla^2 \mathcal{A}(\eta) = \mathbb{E}\left[t(x)t(x)^\mathsf{T}\right] - \mathbb{E}\left[t(x)\right]\mathbb{E}\left[t(x)\right]^\mathsf{T},$$

Compound models' statistics are **polynomials** in component statistics

$$\log p(z_1, z_2, \ldots, z_M, x) = \textstyle\sum_{\beta\in\boldsymbol{\beta}} \langle \eta_\beta(x),\, t_{z_1}(z_1)^{\beta_1} \otimes \cdots \otimes t_{z_M}(z_M)^{\beta_M}\rangle$$
$$\triangleq g(t_{z_1}(z_1), \ldots, t_{z_M}(z_M)),$$

Too much math for a poster

When g is multi-linear (has max-degree 1), then

Claim 2.1. *Given an exponential family with density of the form (3), we have*

$$p(z_m \mid z_{\neg m}) = \exp\left\{\langle \eta^*_{z_m},\, t_{z_m}(z_m)\rangle - \mathcal{A}_{z_m}(\eta^*_{z_m})\right\} \text{ where } \eta^*_{z_m} \triangleq \nabla_{t_{z_m}} g(t_{z_1}(z_1), \ldots, t_{z_M}(z_M)).$$

Define a variational family using the same component statistics

$$q(z) = \prod_m q(z_m; \eta_{z_m}), \qquad q(z_m; \eta_{z_m}) = \exp\left\{\langle \eta_{z_m},\, t_{z_m}(z_m)\rangle - \mathcal{A}_{z_m}(\eta_{z_m})\right\},$$

$$\log p(x) = \log \int p(z,x)\, \nu_z(\mathrm{d}z) = \log \mathbb{E}_{q(z)}\left[\frac{p(z,x)}{q(z)}\right] \geq \mathbb{E}_{q(z)}\left[\log \frac{p(z,x)}{q(z)}\right] \triangleq \mathcal{L}.$$

Claim 2.2. *Given a model with density of the form (3) and variational problem (4)-(5), we have*

$$\operatorname*{arg\,max}_{\eta_{z_m}} \mathcal{L}(\eta_{z_1}, \ldots \eta_{z_M}) = \nabla_{\mu_{z_m}} g(\mu_{z_1}, \ldots, \mu_{z_M}) \text{ where } \mu_{z_{m'}} \triangleq \nabla \mathcal{A}_{z_{m'}}(\eta_{z_{m'}}),\ m' = 1, \ldots, M.$$

A general view on conjugacy: punchlines

- When energy is a **multi-linear polynomial** in **tractable statistic functions**...
  - Generic **Gibbs** via autodiff and a sampler for each statistic
  - Generic **structured mean field** and **SVI** via autodiff and a log normalizer for each statistic
  - Generic **marginalization** via autodiff and a log normalizer for each statistic
- Can write **generic implementations** of **structure-exploiting algorithms**...
  - but only once we're given the **polynomial representation**
  - ... and those are hard to write directly!
- Find polynomial representations automatically?

Term rewriting problem statement

Given a Python function denoting $f : \mathbb{R}^n \mapsto \mathbb{R}$ that has a representation

$$f = g \circ h \quad \text{for a multi-lin. polynomial} \quad g : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_M} \to \mathbb{R},$$

where the coordinate functions $h = (h_1, \ldots, h_M)$ come from a known set,

1. identify each h_m, and
2. produce a Python function to evaluate g.

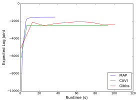
Domain-specific term graph rewriting implementation

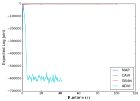
- **Tracer** using Autograd's API to map Python to term graphs
- **Pattern matcher** to do pattern-directed invocation
  - Python-embedded pattern language
  - Compiled into **continuation-passing matcher combinators** (~300 loc)

```
pat = (Einsum, Str('formula'), Segment('args1'),
       (Choice(Subtract('op'), Add('op')), Val('x'), Val('y')), Segment('args2'))
```

- **Rewriters** are syntactic graph macros using tracing to get **quasi-quasiquotes**

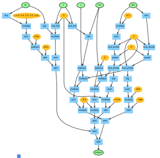
```
def rewriter(formula, op, x, y, args1, args2):
  return op(np.einsum(formula, *(args1 + (x,) + args2)),
            np.einsum(formula, *(args1 + (y,) + args2)))

distribute_einsum = Rule(pat, rewriter) # Rule is a namedtuple
```



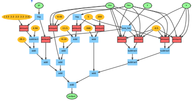



```
def normal_logpdf(x, loc, scale):
  prec = 1. / scale**2
  return -(np.sum(prec * mu**2) - np.sum(np.log(prec))
           + np.log(2. * np.pi)) * N / 2.

def log_joint(pi, z, mu, tau, x):
  logp = (np.sum((alpha-1)*np.log(x))
          - np.sum(gammaln(alpha))
          + np.sum(gammaln(np.sum(alpha, -1))))
  logp += normal_logpdf(mu, 0., 1./np.sqrt(kappa * tau))
  logp += np.sum(one_hot(z, K) * np.log(pi))
  logp += ((a-1)*np.log(tau) - b*tau + a*np.log(b)
           - gammaln(a))

  mu_z = np.dot(one_hot(z, K), mu)
  loglike = normal_logpdf(x, mu_z, 1./np.sqrt(tau))

  return logp + loglike
```

1 **Trace** log joint density given example values and supports



2 **Rewrite** term graph to expose exponential family structure



3 Generic implementations of mean field, marginalization, Gibbs, etc. (in plain Python!)

Model evaluation should be a first-class citizen in probabilistic programming

Alp Kucukelbir, Yixin Wang, Dustin Tran, David M. Blei

Alp Kucukelbir: Columbia CS, Fero Labs
Yixin Wang: Columbia Stats
Dustin Tran: Columbia CS, Google
David M. Blei: Columbia CS + Stats

1 | Introduction

Probabilistic programming research has been tightly focused on two things:

modeling and **inference**.

We argue that model evaluation deserves a similar level of attention.

Probabilistic programming enables the modern applied probabilist to craft bespoke probability models and perform inference with them. She can encode domain specific knowledge into her models with ease and express rich assumptions about the data she seeks to analyze. **With this freedom comes a pronounced need to evaluate such models.** Is there evidence for these assumptions? How well do these models work? We show how probabilistic programming languages offer practical solutions to some of these problems, but argue **that model evaluation deserves more interest from the community at large.**

2 | Methods for Model Evaluation

Focus | probability models with well-defined, evaluable joint distributions.

Scoring rules and point-wise evaluations
evaluating likelihood, computing losses, ideas around cross validation, posterior dispersion indices (Kucukelbir et al.).

Posterior predictive checks (PPCs)

1. Choose a statistic (e.g. min, max)
2. Simulate datasets from posterior predictive
3. Calculate statistics on simulated data
4. Compare to statistic evaluated on original data

Kernel-based methods
visualize smooth regions of data that is poorly explained by model (Lloyd and Ghahramani), kernel goodness-of-fit tests (Chwialkowski et al., Lieu et al.)

3 | Status Quo and Future of Model Evaluation in Probabilistic Programming

Status Quo
Most popular probabilistic programming frameworks offer none or limited high-level constructs to implement model evaluation. Performing model evaluation in these cases requires manual implementation of the methods in Section 2.

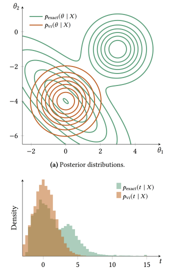
Stan offers a helpful structure that aids in implementing model evaluation. For example, the generated quantities section can be used to compute PPCs and evaluate losses.

PyMC3 and Edward offer a productive out-of-the-box experience for model evaluation. Both have built-in implementations of PPCs and explicit documentation to do model evaluation and comparison. PyMC3 implements information criteria and Edward offers a suite of default scoring rules.

Future
The languages that facilitate model evaluation empower its users to build accurate and powerful probability models; this is a key goal for all probabilistic programming languages.

However, model evaluation faces its own set of challenges, unique to its application within probabilistic programming. Almost all automated inference algorithms are approximate. What happens to our evaluation metrics when the posterior approximation is poor? Samples from MCMC algorithms may not have converged. Using a variational lower bound to the evidence can be dangerous for model comparison. PPCs may be incorrect due to approximation errors in the posterior distribution. Figure 1 shows an example of how this might occur.

Another open question is how to best integrate model evaluation into language semantics. Given the approximate nature of probabilistic programming inference algorithms, there are no accuracy guarantees for posterior computation under bounded time. How can language designers improve the language itself to expose the approximate nature of posterior computations and aid model evaluation?



(a) Posterior distributions.

(b) Discrepancy distributions.

Figure 1. Assume a model with two latent variables $\theta = (\theta_1, \theta_2)$ and likelihood $\prod_{n=1}^{N} \mathcal{N}(x_n; \mu = \theta_1, \sigma^2 = \exp\theta_2)$. Sub-panel **(a)** depicts a bimodal posterior. A variational approximation to this posterior may capture only one of the modes. Sub-panel **(b)** shows the inaccuracy of the discrepancy distribution, as computed by a posterior predictive check, of $t = \max(X)$ using the variational posterior. Evaluating this model based on $p_{\text{vi}}(t \mid X)$ could lead to incorrect conclusions.

- Kacper Chwialkowski, Heiko Strath-
- Been Kim, Rajiv Khanna, and Olu-
- Juho Piironen and Aki Vehtari. 2017.
- John Salvatier et al. 2016. Probabi-