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]
Scaling up fundamental language models

[Liu+ 2018; Shazeer+ 2018]
Inference in a probabilistic program

\[(\text{trace}, \text{weight}) = \text{query}(\text{program}, \text{args}, \text{observations})\]
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.

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

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

all categories  Latest  Top

<table>
<thead>
<tr>
<th>Topic</th>
<th>Category</th>
<th>Users</th>
<th>Replies</th>
<th>Views</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative estimators (“bayes filters”) in Edward?</td>
<td></td>
<td></td>
<td>5</td>
<td>21</td>
<td>7h</td>
</tr>
<tr>
<td>Tutorial for multiple variational methods using Poisson regression?</td>
<td></td>
<td></td>
<td>2</td>
<td>20</td>
<td>1d</td>
</tr>
</tbody>
</table>

Edward
Failure Modes

- **Inference is monolithic.** 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.

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

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.

```python
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 numpy as np
import os
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.environ["TEST_TMPDIR"], "/tmp"),  
    help="Directory where data is stored (if using real data)."
flags.DEFINE_string(  
    "model_dir", default=os.path.join(os.environ["TEST_TMPDIR"], "/tmp"),  
    help="Directory to put the model's fit.")
flags.DEFINE_integer(  
    "viz_steps", default=10000, help="Frequency at which save visualizations."
flags.DEFINE_boolean(  
    "fake_data", default=False, help="If true, uses fake data.
flags.DEFINE_boolean(  
    "delete_existing", default=False, help="If true, deletes existing directory.

FLAGS = flags.FLAGS

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.keras_normal_initializer()))
    encoder_net.add(tf.keras.layers.Dense(  
      num_topics, activation=tf.nn.softplus,  
      kernel_initializer=tf.keras_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.keras_normal_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.keras_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.interception(  
    make_valuesetter(topics=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 Dirichlet analytically.
  # The sampled KL does not work well for "sparse" distributions
```
Example: Latent Dirichlet Allocation

```python
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.
# 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)

# The perplexity is an exponent of the average negative ELBO per word.
perplexity = -elbo / words_per_document
perplexity_tensor = tf.exp(log_perplexity_tensor)
(perplexity_tensor, log_perplexity_update) = tf.metrics.mean(
    perplexity_tensor)

# Obtain the topics summary. Implemented as a py_func for simplicity.
topics = tf.py_func(
    get_topics_strings, [vocabulary], tf.string)
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()),
    },
)
```
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

\[ Y: \ [b, d_y] \]
\[ V: \ [d_h, d_y] \]
\[ H: \ [b, d_h] \]
\[ W: \ [d_x, d_h] \]
\[ X: \ [b, d_x] \]
Example: Perceptron

\[ Y = (H = \text{Relu}(XW_1))V \]

- **Y**: \([b, d_y]\)  \(\text{Split}_b\)
- **V**: \([d_h, d_y]\)  \(\text{Replicated}\)
- **H**: \([b, d_h]\)  \(\text{Split}_b\)
- **W**: \([d_x, d_h]\)  \(\text{Replicated}\)
- **X**: \([b, d_x]\)  \(\text{Split}_b\)

**Data-Parallelism:** Split dimension “b”

- **TPU:0**
  - \(H[b/2:,:] = \text{Relu}(X[b/2:,:]W)\)
  - \(Y[b/2:,:] = H[b/2:,:]V\)

- **TPU:1**
  - \(H[b/2:,:] = \text{Relu}(X[b/2:,:]W)\)
  - \(Y[b/2:,:] = H[b/2:,:]V\)
Example: Perceptron

\[ Y = (H = \text{Relu}(XW_1))W_2 \]

- **Y**: \([b, d_y]\) Replicated
- **V**: \([d_h, d_y]\) Split \(d_h\)
- **H**: \([b, d_h]\) Split \(d_h\)
- **W**: \([d_x, d_h]\) Split \(d_h\)
- **X**: \([b, d_x]\) Replicated

Model-Parallelism: Split dimension \(d_h\)
Example: Perceptron

\[ Y = (H = \text{ReLU}(XW_1))W_2 \]

- **Y**: \([b, d_y]\) \hspace{1cm} Split \((d_y)\)
- **V**: \([d_h, d_y]\) \hspace{1cm} Split \((d_y)\)
- **H**: \([b, d_h]\) \hspace{1cm} Replicated
- **W**: \([d_x, d_h]\) \hspace{1cm} Split \((d_x)\)
- **X**: \([b, d_x]\) \hspace{1cm} Split \((d_x)\)
50M+ parameter models (Image Transformer, VQVAE) on high-resolution images. Data parallelism.

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

**Figure 14:** Vector-Quantized VAE on 64x64 ImageNet.  
**Figure 15:** Image Transformer on 256x256 CelebA-HQ.

[Tran+ 2018]
Example: NUTS

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

<table>
<thead>
<tr>
<th>System</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stan (CPU)</td>
<td>201.0</td>
</tr>
<tr>
<td>PyMC3 (CPU)</td>
<td>74.8</td>
</tr>
<tr>
<td>Handwritten TF (CPU)</td>
<td>66.2</td>
</tr>
<tr>
<td>Edward2 (CPU)</td>
<td>68.4</td>
</tr>
<tr>
<td>Handwritten TF (1 GPU)</td>
<td>9.5</td>
</tr>
<tr>
<td>Edward2 (1 GPU)</td>
<td>9.7</td>
</tr>
<tr>
<td>Edward2 (8 GPU)</td>
<td>2.3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>8</td>
<td>0.22</td>
<td>30.7</td>
<td>8.47</td>
</tr>
<tr>
<td>16384</td>
<td>16</td>
<td>0.37</td>
<td>28.0</td>
<td>7.92</td>
</tr>
<tr>
<td>32768</td>
<td>32</td>
<td>0.67</td>
<td>26.0</td>
<td>7.47</td>
</tr>
<tr>
<td>65516</td>
<td>64</td>
<td>1.28</td>
<td>24.6</td>
<td>6.97</td>
</tr>
<tr>
<td>131072</td>
<td>128</td>
<td>2.48</td>
<td>23.6</td>
<td>6.64</td>
</tr>
<tr>
<td>262144</td>
<td>256</td>
<td>4.90</td>
<td><strong>23.1</strong></td>
<td><strong>6.41</strong></td>
</tr>
</tbody>
</table>

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

[Shazeer+ 2018]
Example: Machine Translation

<table>
<thead>
<tr>
<th>d_ff</th>
<th>heads</th>
<th>Parameters (in Billions)</th>
<th>WMT’14 English-German BLEU</th>
<th>WMT’14 English-French BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>8</td>
<td>0.24</td>
<td>28.6</td>
<td>41.7</td>
</tr>
<tr>
<td>8192</td>
<td>16</td>
<td>0.42</td>
<td>29.1</td>
<td>43.0</td>
</tr>
<tr>
<td>16384</td>
<td>32</td>
<td>0.77</td>
<td>28.9</td>
<td>43.2</td>
</tr>
<tr>
<td>32768</td>
<td>64</td>
<td>1.48</td>
<td>-</td>
<td>43.7</td>
</tr>
<tr>
<td>65516</td>
<td>128</td>
<td>2.89</td>
<td>-</td>
<td>43.7</td>
</tr>
<tr>
<td>4096</td>
<td>16</td>
<td>0.21</td>
<td>28.4</td>
<td>41.8 (Vaswani et. al)</td>
</tr>
</tbody>
</table>

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**

- Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. NIPS 2018.

**Methods**

- Image Transformer. ICML 2018.
def normal_logpdf(x, loc, scale):
    prec = 1. / scale
    return -(np.sum((x - loc) ** 2 * np.log(prec))
            - np.log(2.) - np.log(np.pi) * 0.5)

def log_joint(pi, z, mu, tau, x):
    logp = (np.sum((a - 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

def rewrite(formula, op, x, y, arg1, arg2):
    if op == "\oplus":
        return oplus, eisum(formula, \langle \text{arg1} + (x, y) \rangle, \text{arg2})
    if op == "\otimes":
        return oplus, eisum(formula, \langle \text{arg1} \rangle, \text{arg2})
    return formula, \text{arg1}, \text{arg2}

def distribute(xsum, rule, rewrite):
    return rule xsum, rewrite, rewrite # Rule to a namedtuple

Trace log joint density given example values and supports

Rewrite term graph to expose exponential family structure

Generic implementations of mean field, marginalization, Gibbns, etc. (in plain Python!)
Model evaluation should be a first-class citizen in probabilistic programming

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

Columbia CS Columbia CS Columbia CS Columbia CS + Stats
Fero Labs Columbia Stats Google

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?