LazyBones - A General Purpose Deferred ObjectProxy Which Enables Simple Symbolic Representations of Programs

authors: jvdillon, junpenglao
date: 25-sep-2020
revised: 15-oct-2020

1 Problem Statement

We believe that user experience during model building and inference is fundamentally interactive - especially if we want to develop interpretable models and understand black box methods. An interactive model building approach is where users can inspect different model components in isolation, ***and*** at the same time straightforward to see the effect of "what if I change the value of this component to X".

One of the answers to this user need is to develop a computational graph which tracks *future results*. This allows users to inspect / interact / alter computer programs by working with the graph's vertices and edges.

1.1 Graphical approach in PPL

Both PyMC3 and Edward1 enabled writing graphical models using the following syntax:

```
W = Normal(0, 1)
B = Normal(0, 1)
Y = LogitNormal(x * W + B, 1)
```

yet also supported immediate inspection of vertices in a REPL friendly manner, e.g.:

```
print(B.scale)
```

This style is very convenient. It enables fast prototyping and easy debugging.

Conversely, none of tfp.distributions.JointDistribution*, PyMC4, (num-)pyro, nor Edward2 enable this workflow. For example, here's the same graphical model in TFP:

```
import tensorflow_probability as tfp
tfd = tfp.distributions
Root = tfd.JointDistributionCoroutine.Root
def model(x):
  W = yield Root(tfd.Normal(0,1))
  B = yield Root(tfd.Normal(0,1))
  Y = yield tfd.LogitNormal(x * W + B, 1)
my_pgm = tfd.JointDistributionCoroutine(lambda: model(1.))
```

While this "trace pattern" enables control flow, it also means the PGM is not easily dissectable by the author. The atomic programmatic unit is the *function* not its *constituents*.

2 tfp.experimental.lazybones

"Lazy Bones" is an abstraction for building generic computation graphs. It began as a prototype for REPL friendly graphical model specification but has since expanded to a general purpose substrate for symbolic computation. It is designed to work with any python module, not just TF, Jax, TFP, Pytorch etc.

The basic idea behind LazyBones is to have a transparent object proxy (like wrapt) which implements every standard overloadable operator and allows the new object to be used *as if it is the result of some not-yet performed computation*. Unlike wrapt however, the behavior is not implemented by an underlying object but rather by a function called `__action__`. This enables a kind of "lazy dynamic dispatch" which means computation graphs can be built as if results are being computed yet they are not actually computed until time of need, eg, `print` or tab-completion. The ultimate goal is that users apply lazybones wrapping right after import and get instant graph experience, while the rest of the code still works as is.

Describing LazyBones is a lot harder than using it. Here's an example:

```
import math
import numpy
import tensorflow_probability as tfp
lb = tfp.experimental.lazybones

np = lb.DeferredInput(numpy)
m = lb.DeferredInput(math)

u = np.random.RandomState(42).normal(0., 0.5)
x = 1. / (1. + math.e**u)
y = m.sin(x)
```

All LazyBones vertexes are `Deferred`:

```
assert isinstance(u, lb.Deferred)
assert isinstance(x, lb.Deferred)
assert isinstance(y, lb.Deferred)
```

No node is evaluated by default:

```
print(u.value)  # ==> [Unknown]
print(x.value)  # ==> [Unknown]
print(y.value)  # ==> [Unknown]
```

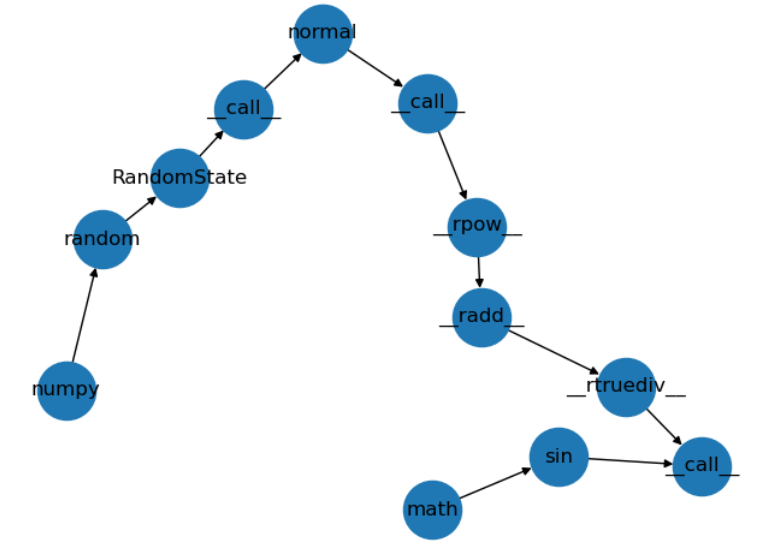
(Usually a user wouldn't directly access `value`; we do so here only to "prove" no evaluation actually happened.)

Nodes are evaluated only when they need to be, e.g.:

```
print(x)        # ==> 0.4382279198026049
print(x.value)  # ==> 0.4382279198026049
print(y.value)  # ==> [Unknown]
print(y)        # ==> 0.424335504617266
```

Graphs are visualizable:

```
lb.utils.plot_graph(y)
# ==>
```



Graphs are also iterable, e.g.:

```
for u,v in lb.utils.iter_edges(y):
  print(u.name, '-->', v.name)
# ==>
#   __rtruediv__ --> __call__
#   sin --> __call__
#   __radd__ --> __rtruediv__
#   __rpow__ --> __radd__
#   __call__ --> __rpow__
#   normal --> __call__
#   __call__ --> normal
#   RandomState --> __call__
#   random --> RandomState
#   numpy --> random
#   math --> sin
```

Deferred values can be manually re-assigned and under a reentrant scope:

```
with lb.DeferredScope() as s:
  u.value = 0.
  print(u,x,y)
  # ==> 0.0 0.5 0.479425538604203

print(u,x,y)
# ==> 0.24835707650561634 0.4382279198026049 0.424335504617266

with s:
  print(u,x,y)
```

```
  # ==> 0.0 0.5 0.479425538604203
```

DeferredScope is deceptively simple yet powerful. It enables counterfactual assignment which when combined with automatic descendant evaluation means the graph has semantics of a function yet all vertices have local scope and yet the original graph values are preserved. This is the core "trick" behind how LazyBones might be used for building a graphical model formalism and one which we now study in greater detail.

We are not aware of direct analogues of DeferredScope in TF2 or Jax. (Please let us know if there are!)

3 How could LazyBones be used to build a PPL?

We imagine LazyBones could be useful for a variety of symbolic computation problems. In the context of probabilistic programming, one of the most basic use cases is to evaluate the probability of particular values of a graphical model. We now explore how one can wrap TFP and simply "walk the LazyBones graph."

For example,

```
import tensorflow
import tensorflow_probability
lb = tensorflow_probability.experimental.lazybones
tf = lb.DeferredInput(tensorflow)
tfp = lb.DeferredInput(tensorflow_probability)

tfd = tfp.distributions
tfb = tfp.bijectors

a = tfd.Normal(0, 1)
b = a.mean()                      # Like a random variable "test value."
c = tf.exp(b)
d = tfb.Exp()(tfd.Normal(c, 2.))
e = d.sample(seed=42)             # Like a random variable "test value."

print(lb.utils.log_prob([b, e], [None, None]))   # ==> -11.846617  [random]
print(lb.utils.log_prob([b, e], [3., 4.]))       # ==> -52.125027
print(lb.utils.log_prob([e, b], [0.1, 0.2]))     # ==> -1.8007501
print(lb.utils.log_prob([b, e], [3., 4.]))       # ==> -52.125027
print(lb.utils.log_prob([b, e], [None, None]))   # ==> -9.62112    [random]
```

Note that we can also pin values in the global scope and that these values are "sticky."

```
# By pinning the random variables to a different set of values, we now show that the
values of the "global graph" are unaffected.
b.value = 0.1
e.value = 0.2
```

```
print(lb.utils.log_prob([b, e], [None, None]))   # ==> -1.847724
print(lb.utils.log_prob([b, e], [3., 4.]))       # ==> -52.125027
print(lb.utils.log_prob([e, b], [0.1, 0.2]))     # ==> -1.8007501
print(lb.utils.log_prob([b, e], [3., 4.]))       # ==> -52.125027
# We now show that the values of the "global graph" are unaffected.
print(lb.utils.log_prob([b, e], [None, None]))   # ==> -1.847724
```

The `lb.utils.log_prob` implementation is surprisingly simple. Under a DeferredScope it assigns values to LB vertices and returns the sum of the log_probs:

```
def log_prob(vertexes, values):
  """Returns log_prob when vertexes take on values."""
  return _distribution_measure(vertexes, values, 'log_prob', sum)

def _distribution_measure(vertexes, values, attr, combine):
  """Returns getattr(distribution) when vertexes take on values."""
  vertexes = tf.nest.flatten(vertexes)
  values = tf.nest.flatten(values)
  distributions = []
  with lb.DeferredScope():
    for x, v in zip(vertexes, values):
      if not isinstance(x, deferred.DeferredBase):
        raise ValueError()
      if v is not None:
        x.value = v
      # We assume the provided nodes are grandchildren of a distribution.
      d = x.parents[0].parents[0]
      distributions.append(d)
    r = combine(getattr(d, attr)(x) for d, x in zip(distributions, vertexes))
    return r.eval()
```

3.1 User Examples

In this section we demonstrate how LazyBones can trivially "wrap" TFP to enable building probabilistic graphical models in a REPL friendly way. See the accompanying colab for end-to-end executable/trainable versions of the following examples.

We emphasize that the following demonstrations exploit *no special treatment from LazyBones* other than the above 16 lines of code in `lb.utils.log_prob`. That is, the PPL use-case of LazyBones is *exactly the same as the general use of LazyBones*.

3.1.1 Simple Linear Regression with pure Scipy

The following example implements a simple linear regression model, with hyper priors on coefficients:

μ ~ Normal(loc=0, scale=100)

$\sigma_0, \sigma_1 \sim \text{HalfNormal}(\text{scale}=5)$
$\vec{\omega} \sim \text{Normal}(\text{loc}=\mu, \text{scale}=\sigma_0)$
$Y \sim \text{Normal}(\text{loc}=\vec{\omega} * X, \text{scale}=\sigma_1)$

Which in LazyBones looks like:

```
lb = import tfp.experimental.lazybones
sp = DeferredInput(sp)

# Hyperpriors:
hyper_mu = sp.stats.norm(0., 100.).rvs()
hyper_sigma = sp.stats.halfnorm(0., 5.).rvs()

# Priors:
beta = sp.stats.norm(hyper_mu, hyper_sigma).rvs(n_feature)
sigma = sp.stats.halfnorm(0., 100.).rvs()

# Likelihood
y_hat = sp.matmul(design_matrix, beta)
y = sp.stats.norm(y_hat, sigma).rvs()
```

3.1.2 Hierarchical Linear Model

The following example implements this graphical model (radon data set):

$g_0, g_1, g_2 \sim \text{Normal}(\text{loc}=0, \text{scale}=10)$
$\vec{\mu} = g_0 + g_1 * \text{predictor}_1 + g_2 * \text{predictor}_2$
$\sigma_0, \sigma_1 \sim \text{Exp}(\text{rate}=1.)$
$\vec{\omega} \sim \text{Normal}(\text{loc}=\vec{\mu}, \text{scale}=\sigma_0)$
for i = 1...n:
  $b \sim \text{Normal}(\text{loc}=0, \text{scale}=1)$
  $\theta_i = \vec{\omega}_{\text{index}[i]} + b * \text{predictor}_3$
  $Y_i \sim \text{Normal}(\text{loc}=\theta_i, \text{scale}=\sigma_1)$

Which in LazyBones looks like:

```
import tfp.experimental.lazybones as lb
tfw = lb.DeferredInput(tf)
tfpw = lb.DeferredInput(tfp)
tfdw = lb.DeferredInput(tfp.distributions)
tfbw = lb.DeferredInput(tfp.bijectors)

# Hyperpriors:
g = tfdw.Sample(tfdw.Normal(loc=0., scale=10.), sample_shape=3).sample()
sigma_a = tfdw.Exponential(rate=1.).sample()

# Varying intercepts uranium model:
a = g[0] + g[1] * uranium + g[2] * avg_floor
za_county = tfdw.Sample(
```

```
    tfdw.Normal(loc=0., scale=1.),
    sample_shape=counties).sample()
a_county = a + za_county * sigma_a

# Common slope:
b = tfdw.Normal(loc=0., scale=1.).sample()

# Expected value per county:
theta = a_county[county_idx] + b * floor_measure

# Model error:
sigma = tfdw.Exponential(rate=1.0).sample()

y = tfdw.Independent(
      tfdw.Normal(loc=theta, scale=sigma),
      reinterpreted_batch_ndims=1).sample()
```

3.1.3 Latent Mixture

The following example implements this graphical model (robust regression using mixture likelihood for outlier detection):

$b_0, b_1, \mu_{outlier} \sim \text{Normal}(loc=0, scale=10)$
$\vec{\omega} = b_0 + b_1 * predictor$
$\mu_{outlier} \sim \text{HalfNormal}(scale=1.)$
$\alpha \sim \text{Uniform}(low=0, high=0.5)$
$Y \sim \text{Mixture}([\alpha, 1 - \alpha]$
$[\text{Normal}(\vec{\omega}, \sigma_{observed}),$
$\text{Normal}(\mu_{outlier}, \sigma_{observed} + \sigma_{outlier})])$

Which in LazyBones looks like:

```
lb = import tfp.experimental.lazybones
jaxw = DeferredInput(jax)
jnpw = DeferredInput(jnp)
tfpw = DeferredInput(tfp)
tfdw = DeferredInput(tfp.distributions)

nobs = len(y_sigma)

# Priors
b0 = tfdw.Normal(loc=0., scale=10.).sample()
b1 = tfdw.Normal(loc=0., scale=10.).sample()
mu_out = tfdw.Normal(loc=0., scale=10.).sample()
sigma_out = tfdw.HalfNormal(scale=1.).sample()
weight = tfdw.Uniform(low=0., high=.5).sample()

# Likelihood
# note we are constructing components as distributions but not RV
mixture_dist = tfdw.Categorical(
    probs=jnpw.repeat(
```

```
        jnpw.array([1 - weight, weight])[None, ...], nobs, axis=0))
component_dist = tfdw.Normal(
    loc=jnpw.stack([b0 + b1*predictors,
                    jnpw.repeat(mu_out, nobs)]).T,
    scale=jnpw.stack([y_sigma, sigma_out + y_sigma]).T)
observed = tfdw.Independent(
    tfdw.MixtureSameFamily(mixture_dist, component_dist),
    reinterpreted_batch_ndims=1).sample()

# Posterior
target_log_prob_fn = lambda *values: lb.utils.log_prob(
    vertexes=[b0, b1, mu_out, sigma_out, weight, observed],
    values=[*values, obs])
```

(Note: in the above example we used Jax because….why not?!)

3.1.4 Autoregressive Time-Series

The following example implements this graphical model:

b ~ LogitNormal(loc=0.5, scale=1.)
X_0 ~ Normal(loc=0, scale=σ_1)
for i = 1...n:
  X_i ~ Normal(loc=b * X_{i-1}, scale=σ_1)
  Y_i ~ Normal(loc=X_i, scale=σ_2)

Which in LazyBones looks like:

```
lb = import tfp.experimental.lazybones
tfw = DeferredInput(tf)
tfpw = DeferredInput(tfp)
tfdw = DeferredInput(tfp.distributions)
tfbw = DeferredInput(tfp.bijectors)

b = tfdw.LogitNormal(loc=0.5, scale=1.).sample(seed=seed)
x0 = tfdw.Normal(loc=0., scale=driving_noise).sample(seed=seed)
x = [x0]

for t in range(1, T):
  x_ = tfdw.Normal(loc=b * x[t - 1], scale=driving_noise).sample()
  x.append(x_)

yobs2 = tfdw.Independent(
    tfdw.Normal(
        loc=tfw.repeat(tfw.stack(x)[..., None], n_obs, axis=-1),
        scale=measure_noise),
    reinterpreted_batch_ndims=2).sample()
```

4 Known Limitations

4.1 Control Flow

Currently LazyBones does not support python control flow, e.g., `for`, `while`, `if`. For most graphical models we don't actually anticipate this as a problem. For example, for time series one need only build the lazybones graph for the longest possible sequence then evaluate observations against the leading set of vertices of the full lazybones graph (and ignore the tail). Since lazybones only evaluates what's needed, any unaccessed tail vetexes have zero computational overhead.

Although control flow is not currently supported, wynnv@ is exploring minimal AST munging to make this possible. We believe minimal AST access (e.g. for "ForLoop") is ideal because of minimizing wall time and also keeping code complexity low. That is, a key part of the beauty of LazyBones is that it is surprisingly easy to debug because it's so lightweight.

4.2 Other Things?

TODO: Find other flaws and list them here.