What is TensorFlow Probability?

A open source Python library built using TF which makes it easy to combine deep learning with probabilistic models on modern hardware. It is for:

- **Statisticians/data scientists.** R-like capabilities that run out-of-the-box on TPUs + GPUs.
- **ML researchers/practitioners.** Build deep models which capture uncertainty.
Why use TensorFlow Probability?

A **deep network** predicting binary outcomes is "just" a fancy parametrization of a Bernoulli distribution.

*Great! Now what?*

Enode knowledge through **richer distributional assumptions**!

- control prediction variance
- prior knowledge
- ask (and answer) tougher questions
Take Home Message

Express your domain knowledge as a probabilistic model.

Use TFP to execute it.
How do I use TensorFlow Probability?

Build model. 🔄 Do inference.
How do I use TensorFlow Probability?

Build model.  Do inference.

Canned approach: GLMs
# Build model.
model = tfp.glm.Bernoulli()

# Fit model.
coeffs, linear_response, is_converged, num_iter = \
tfp.glm.fit_sparse(
    model_matrix=x, response=y,
    l1_regularizer=0.5,   # Induces sparse weights.
    l2_regularizer=1.,    # Also prevents over-fitting.
    model=model)
How do I use TensorFlow Probability?

- **Build model.**
  - Distributions
  - Bijectors
  - Layers/Losses
  - Edward2

- **Do inference.**
  - MCMC
  - Variational Inference
  - Optimizers
class Distribution(object):
    def sample(self, sample_shape=(), seed=None): pass
    def prob(self, value): pass
    def cdf(self, value): pass
    def survival_function(self, value): pass
    def mean(self): pass
    def variance(self): pass
    def stddev(self): pass
    def mode(self): pass
    def quantile(self, p): pass
    def entropy(self): pass
    def cross_entropy(self, other): pass
    def event_shape(self): pass
    def batch_shape(self): pass
"Hello, World!"

```python
import tensorflow_probability as tfp
tfd = tfp.distributions
d = tfd.Normal(loc=0., scale=1.)
x = d.sample()  # Draw random point.
px = d.prob(x)  # Compute density/mass.
```
Distributions are *Expressive*

```python
factorial_mog = tfd.Independent(
    tfd.MixtureSameFamily(
        # Uniform weight on each component.
        mixture_distribution=tfd.Categorical(
            logits=tf.zeros([num_vars, num_components]),
            component_distribution=tfd.MultivariateNormalDiag(
                loc=mu, scale_diag=[sigma]),
            reinterpreted_batch_ndims=1)
    samples = factorial_mog.sample(1000)
```
How do I use TensorFlow Probability?

**Build model.**
- Distributions
- Bijectors
- Layers/Losses
- Edward2

**Do inference.**
- MCMC
- Variational Inference
- Optimizers
```python
class Bijector(object):
    def forward(self, x): pass
    def forward_log_det_jacobian(self, x): pass
    def inverse(self, x): pass
    def inverse_log_det_jacobian(self, x, event_ndims): pass
    def forward_event_shape(self, x): pass
    def forward_min_event_ndims(self, x): pass
    def inverse_event_shape(self, x): pass
    def inverse_min_event_ndims(self, x): pass
```

**Compute Samples**

**Compute Probabilities**

**Shape**
Bijectors *Transform* Distributions

# Masked Autoregressive Flow for Density Estimation.

```python
iaf = tfp.distributions.TransformedDistribution(
    distribution=tfp.distributions.Normal(loc=0., scale=1.),
    bijector=(
        tfp.bijectors.MaskedAutoregressiveFlow(
            shift_and_log_scale_fn=
                tfb.masked_autoregressive_default_template(
                    hidden_layers=[512, 512]))),
    event_shape=[dims])

loss = -iaf.log_prob(x)  # DNN powered PDF. Wow!
```

Or your own DNN.
Bijectors *Transform* Distributions

# Improved Variational Inference with Inverse Autoregressive Flow  
# Kingma, et. al., NIPS 2016.

```python
iaf = tfp.distributions.TransformedDistribution(
    distribution=tfp.distributions.Normal(loc=0., scale=1.),
    bijector=tfp.bijectors.Invert(
        tfp.bijectors.MaskedAutoregressiveFlow(
            shift_and_log_scale_fn=
                tfb.masked_autoregressive_default_template(
                    hidden_layers=[512, 512])),
            event_shape=[dims]),

loss = -iaf.log_prob(x) # DNN powered PDF. Wow!
```

Different paper but easy in TFP.
Use Case: Anomaly Detection

Did the user's texting habits change over time?

(“Bayesian Methods for Hackers” by Cameron Davidson-Pilon)
def joint_log_prob(count_data, lambda_1, lambda_2, tau):
    alpha = 1. / count_data.mean()
    rv_lambda = tfd.Exponential(rate=alpha)
    rv_tau = tfd.Uniform()
    indices = tf.to_int32(
        tau * count_data.size <=
        tf.range(count_data.size))
    lambda_ = tf.gather(
        [lambda_1, lambda_2], indices)
    rv_x = tfd.Poisson(rate=lambda_)
    return (rv_lambda.log_prob(lambda_1) +
            rv_lambda.log_prob(lambda_2) +
            rv_tau.log_prob(tau) +
            tf.reduce_sum(
                rv_x.log_prob(count_data)))

# Code this up in TFP

\[ \lambda^{(0)}_1 \sim \text{Exponential}(\text{rate} = \alpha) \]
\[ \lambda^{(0)}_2 \sim \text{Exponential}(\text{rate} = \alpha) \]
\[ \tau \sim \text{Uniform}[\text{low} = 0, \text{high} = 1] \]
for \( i = 1 \ldots N \):
\[ \lambda_i = \begin{cases} 
\lambda^{(0)}_1, & \tau > i/N \\
\lambda^{(0)}_2, & \text{otherwise}
\end{cases} \]
\[ X_i \sim \text{Poisson}(\text{rate} = \lambda_i) \]
def joint_log_prob(count_data, lambda_1, lambda_2, tau):
    alpha = 1. / count_data.mean()
    rv_lambda = tfd.Exponential(rate=alpha)
    rv_tau = tfd.Uniform()
    indices = tf.to_int32(
        tau * count_data.size <=
        tf.range(count_data.size))
    lambda_ = tf.gather(
        [lambda_1, lambda_2], indices)
    rv_x = tfd.Poisson(rate=lambda_)
    return (rv_lambda.log_prob(lambda_1)
            + rv_lambda.log_prob(lambda_2)
            + rv_tau.log_prob(tau)
            + tf.reduce_sum(
                rv_x.log_prob(count_data)))
What are the posterior distributions?
How do I use TensorFlow Probability?

**Build model.**
- Distributions
- Bijectors
- Layers / Losses
- Edward2

**Do inference.**
- MCMC
- Variational Inference
- Optimizers
Sampling Posterior

```python
[lambda_1, lambda_2, tau], _ = tfp.mcmc.sample_chain(
    num_results=int(10e3),
    num_burnin_steps=int(1e3),
    current_state=initial_chain_state,
    kernel=tfp.mcmc.TransformedTransitionKernel(
        inner_kernel=tfp.mcmc.HamiltonianMonteCarlo(
            target_log_prob_fn=lambda *s: joint_log_prob(count_data, *s),
            num_leapfrog_steps=2,
            step_size=tf.Variable(1.),
            step_size_update_fn=
                tfp.mcmc.make_simple_step_size_update_policy(),
        ),
        bijector=[
            tfp.bijectors.Exp(),  # Lambda_1
            tfp.bijectors.Exp(),  # Lambda_2
            tfp.bijectors.Sigmoid()  # Tau
        ]
    ),
)
```

Setup: We'll use transformed HMC to draw 10K samples from our posterior.
Sampling Posterior

\[
[s, \lambda_1, \lambda_2, \tau], _ = \text{tfp.mcmc.sample_chain}(
\text{num_results}=\text{int}(10e3),
\text{num_burnin_steps}=\text{int}(1e3),
\text{current_state}=\text{initial_chain_state},
\text{kernel}=\text{tfp.mcmc.TransformedTransitionKernel}(
\text{inner_kernel}=\text{tfp.mcmc.HamiltonianMonteCarlo}(
\text{target_log_prob_fn}=\lambda s: \text{joint_log_prob}(s, \text{count_data}),
\text{num_leapfrog_steps}=2,
\text{step_size}=\text{tf.Variable}(1.),
\text{step_size_update_fn}=
\text{tfp.mcmc.make_simple_step_size_update_policy}(),
\text{bijector}=\[
\text{tfp.bijectors.Exp()}, \quad \# \text{ Lambda1}
\text{tfp.bijectors.Exp()}, \quad \# \text{ Lambda2}
\text{tfp.bijectors.Sigmoid()}) \quad \# \text{ Tau}
\])
\]

Map random variables' supports to unconstrained reals. Ensures HMC samples always have >0 probability and chain doesn't get stuck.
Sampling Posterior

```python
[lambda_1, lambda_2, tau], _ = tfp.mcmc.sample_chain(
    num_results=int(10e3),
    num_burnin_steps=int(1e3),
    current_state=initial_chain_state,
    kernel=tfp.mcmc.TransformedTransitionKernel(
        inner_kernel=tfp.mcmc.HamiltonianMonteCarlo(
            target_log_prob_fn=lambda *s: joint_log_prob(count_data, *s),
            num_leapfrog_steps=2,
            step_size=tf.Variable(1.),
            step_size_update_fn=
                tfp.mcmc.make_simple_step_size_update_policy(),
        bijector=[
            tfp.bijectors.Exp(), # Lambda1
            tfp.bijectors.Exp(), # Lambda2
            tfp.bijectors.Sigmoid()]) # Tau
```
And the answer is?!
$\pi \sim \text{Uniform}(\text{low} = 0, \text{high} = 1)$

$\rho_0, \rho_1 \sim \text{Uniform}(\text{low} = 0, \text{high} = 50)$

for $i = 1 \ldots I : \#$ Items

$C_i | \pi \sim \text{Bernoulli}(\text{probs} = \pi)$

$\delta_i | C_i, \rho \sim \text{Normal}(\text{loc} = 0, \text{scale} = c_i \rho_1^2 + (1 - c_i) \rho_0^2)$

$\mu_0, \mu_1 \sim \text{Normal}(\text{loc} = 0, \text{scale} = 10)$

$\sigma_0 \sim \text{Uniform}(\text{low} = 0, \text{high} = 50)$

$\sigma_1 \sim \text{Uniform}(\text{low} = 0, \text{high} = 100)$

for $j = 1 \ldots J : \#$ Annotators

$\gamma_{0,j} | \mu_0, \sigma_0 \sim \text{Normal}(\text{loc} = \mu_0, \text{scale} = \sigma_0)$

$\gamma_{1,j} | \mu_1, \sigma_1 \sim \text{Normal}(\text{loc} = \mu_1, \text{scale} = \sigma_1)$

for $k = 1 \ldots K : \#$ Observations

$X_k | C, \delta, \gamma \sim \text{Bernoulli}(\text{probs} = c_{\text{item}(k)} \logit^{-1}(\gamma_{1,\text{annotator}(k)} - \delta_{\text{item}(k)}) + (1 - c_{\text{item}(k)})(1 - \logit^{-1}(\gamma_{0,\text{annotator}(k)} - \delta_{\text{item}(k)})))$
def joint_log_prob(x, annotators, items,
    pi, rho, c, delta, mu, sigma, gamma):
    # Items plate. (I)
    rv_pi = tfd.Uniform(low=0., high=1.)
    rv_rho = tfd.Uniform(low=0., high=50.)
    rv_c = tfd.Uniform(low=0., high=1.)
    rv_delta = tfd.Normal(
        loc=0, scale=tf.gather(rho, tf.to_int32(c<pi)))
    # Annotators plate. (J)
    rv_mu = tfd.Normal(loc=0., scale=10.)
    rv_sigma = tfd.Uniform(low=0., high=[50., 100.])
    rv_gamma = tfd.Normal(loc=mu, scale=sigma)
    # ...continued in next column.

    # Observations plate. (K)
    d = tf.gather(delta, items)
    g = tf.gather(gamma, annotators, axis=0)
    rv_x = tfd.Bernoulli(
        logits=tf.where(tf.gather(c < pi, items),
                        g[:, 1] - d, -g[:, 0] + d))
    # Compute the actual log prob.
    return sum(map(tf.reduce_sum, [
        rv_pi.log_prob(pi), rv_rho.log_prob(rho),
        rv_c.log_prob(c), rv_delta.log_prob(delta),
        rv_mu.log_prob(mu), rv_sigma.log_prob(sigma),
        rv_x.log_prob(x), rv_gamma.log_prob(gamma)]))

    # ...continued from previous column.
def joint_log_prob(x, annotators, items, 
   pi, rho, c, delta, mu, sigma, gamma):

   # Items plate. (I)
   rv_pi = tfd.Uniform(low=0., high=1.)
   rv_rho = tfd.Uniform(low=0., high=50.)
   rv_c = tfd.Uniform(low=0., high=1.)
   rv_delta = tfd.Normal(
      loc=0., scale=tf.gather(rho, tf.to_int32(c<pi)))

   # Annotators plate. (J)
   rv_mu = tfd.Normal(loc=0., scale=10.)
   rv_sigma = tfd.Uniform(low=0., high=[50., 100.])
   rv_gamma = tfd.Normal(loc=mu, scale=sigma)

   # ...continued in next column.

   # ...continued from previous column.
   # Observations plate. (K)
   d = tf.gather(delta, items)
   g = tf.gather(gamma, annotators, axis=0)
   rv_x = tfd.Bernoulli(
      logits=tf.where(tf.gather(c < pi, items),
                      g[:, 1] - d, -g[:, 0] + d))

   # Compute the actual log prob.
   return sum(map(tf.reduce_sum,
                  [rv_pi.log_prob(pi),
                   rv_rho.log_prob(rho),
                   rv_c.log_prob(c),
                   rv_delta.log_prob(delta),
                   rv_mu.log_prob(mu),
                   rv_sigma.log_prob(sigma),
                   rv_x.log_prob(x),
                   rv_gamma.log_prob(gamma)]))
def joint_log_prob(x, annotators, items, pi, rho, c, delta, mu, sigma, gamma):

    # Items plate. (I)
    rv_pi = tfd.Uniform(low=0., high=1.)
    rv_rho = tfd.Uniform(low=0., high=50.)
    rv_c = tfd.Uniform(low=0., high=1.)
    rv_delta = tfd.Normal(loc=0, scale=tf.gather(rho, tf.to_int32(pi)))

    # Annotators plate. (J)
    rv_mu = tfd.Normal(loc=0., scale=10.)
    rv_sigma = tfd.Uniform(low=0., high=[50., 100.])
    rv_gamma = tfd.Normal(loc=mu, scale=sigma)

    return sum(map(tf.reduce_sum, [
        rv_pi.log_prob(pi), rv_rho.log_prob(rho),
        rv_c.log_prob(c), rv_delta.log_prob(delta),
        rv_mu.log_prob(mu), rv_sigma.log_prob(sigma),
        rv_x.log_prob(x), rv_gamma.log_prob(gamma)]))
def joint_log_prob(x, annotators, items, pi, rho, c, delta, mu, sigma, gamma):
    # Items plate. (I)
    rv_pi = tfd.Uniform(low=0., high=1.)
    rv_rho = tfd.Uniform(low=0., high=50.)
    rv_c = tfd.Uniform(low=0., high=1.)
    rv_delta = tfd.Normal(
        loc=0, scale=tf.gather(rho, tf.to_int32(c < pi, items)))
    # Annotators plate. (J)
    rv_mu = tfd.Normal(loc=0, scale=10)
    rv_sigma = tfd.Uniform(low=0., high=[50, 100])
    rv_gamma = tfd.Normal(loc=mu, scale=sigma)
    # ...continued from previous column.
    # Observations plate. (K)
    d = tf.gather(delta, items)
    g = tf.gather(gamma, annotators, axis=0)
    rv_x = tfd.Bernoulli(
        logits=tf.where(tf.gather(c < pi, items),
                        g[:, 1] - d, -g[:, 0] + d))
    # Compute the actual log prob.
    return sum(map(tf.reduce_sum, [
        rv_pi.log_prob(pi), rv_rho.log_prob(rho),
        rv_c.log_prob(c), rv_delta.log_prob(delta),
        rv_mu.log_prob(mu), rv_sigma.log_prob(sigma),
        rv_gamma.log_prob(gamma)]))

for k = 1...K:
    # Observations
    X_k|C, δ, γ ~ Bernoulli(probs = c_{item(k)} logit^{-1}(γ_{1,annotator(k)} - δ_{item(k)}) +
        (1 - c_{item(k)})(1 - logit^{-1}(γ_{0,annotator(k)} - δ_{item(k)})))
But where's the deep learnin'? 

Learn both models *jointly*!
# Variational posterior, actually.
q_given_x = make_posterior(x)

# Latents, conditioned on evidence.
z = q_given_x.sample(num_draws)
p_given_z = make_likelihood(z)
r = make_prior()

logq = q_given_x.log_prob(z)
logp = p_given_z.log_prob(x) + r.log_prob(z)

# Approx KL[q(Z|x), p(x,Z)].
loss = tf.reduce_mean(logq - logp)

train = tf.train.Optimizer().minimize(loss)

VAE = Deep + Probability

# Create three random variables: encoder, decoder, prior.
The encoder and decoder are powered by neural nets.
Seriously, where do I start?

- Chapter 1 in TFP (GitHub PR)
- Chapter 2 in TFP (GitHub PR)
- Chapter 3 in TFP (GitHub PR)
- Chapter 4 in TFP (GitHub PR)
- Chapter 5 in TFP (GitHub PR)
- Chapter 6 in TFP (GitHub PR)
Conclusion

A open source Python library built using TF which makes it easy to combine \textbf{deep learning} with \textbf{probabilistic models} on modern hardware.

- **Install** `pip install tensorflow-probability[-gpu]`
- **Learn More** tensorflow.org/probability
- **Email** tfprobability@tensorflow.org
Join the TFP community!

tfprobability@tensorflow.org
www.tensorflow.org/probability
Deep learning for fundamental sciences using high-performance computing

Wahid Bhimji, Debbie Bard, Steven Farrell, Mustafa Mustafa, Thorsten Kurth, Prabhat and many others
NERSC, Lawrence Berkeley National Laboratory
Outline

- Fundamental sciences make heavy use of high performance computing [at NERSC] for simulation and data analysis
- Progress in Deep Learning and tools like Tensorflow can enable the use of higher dimensional data; increased sensitivity for new discoveries; faster computation and whole new approaches
- Illustrate this here with a few example projects running at NERSC
NERSC: US Dept. of Energy Mission Supercomputing Center, serves 7000+ scientists, 800+ projects

Cori: #10 most powerful supercomputer on the planet (27.9 PF) Top500.org
Growth in AI displacing Simulation and Data analysis

Secrets of the universe

- Big Bang Expansion: 13.7 billion years
- 1st Stars: about 400 million yrs.
- Quantum Fluctuations
- Inflation
- Dark Ages
- Development of Galaxies, Planets, etc.
- Dark Energy Accelerated Expansion
- Afterglow Light Pattern: 380,000 yrs.
Huge complex instruments

Large Synoptic Survey Telescope

ATLAS Detector at Large Hadron Collider / CERN
Secrets of the Universe: Nature of Dark Matter
New particles
Many areas where deep learning can help, e.g.:

- **Classification** e.g. to find physics objects within collision or which collisions produced new particles
- **Regression** e.g. to aid reconstruction of particle deposits or of fundamental physics parameters
- **Clustering / feature detection** in high-dimension raw data to find unexpected physics or instrument issues
- **Generation** of data to replace full physics simulation
Classification: Is this new physics? (e.g. supersymmetry)
LHC-CNN: Unroll cylindrical data to form an image and apply convolutional neural network (CNN) to classify known vs new physics
CNN architecture can outperform other methods
Regression: What possible universe would look like this (e.g. values of parameters like dark energy density)

Image Credit: M. Blanton and SDSS
CosmoFlow: Apply 3D CNN to large $128^3$ voxel data; run on 8000 CPU nodes, data-parallel, to predict 3 cosmological parameters in minutes

Achieves unprecedented accuracy
Generation: Does it need to take 2 weeks on a supercomputer to get a simulation? (e.g. for mass convergence maps to compare to observed data)
CosmoGAN: Apply DCGAN architecture to maps

[Diagram of DCGAN architecture]

Mustafa, Bard, Bhimji, Lukic, Al-Rfou, Kratochvil (LBNL, Google)
https://arxiv.org/abs/1706.02390 (see also GANs applied to particle physics in Paganini et. al. (arXiv:1705.02355))
Reproduces maps to very high-precision including higher-order statistics used by cosmologists

Mustafa, Bard, Bhimji, Lukic, Al-Rfou, Kratochvil, (LBNL, Google)
https://arxiv.org/abs/1706.02390 (see also GANs applied to particle physics in Paganini et. al. (arXiv:1705.02355))
Using extreme computing scales: data parallel distributed training with TF and MPI with Horovod and Cray PE ML Plugin

LHC-CNN (Kurth et al. Concurrency Computat Pract Exper. 2018;e4989)

CosmoFlow: (Mathuriya et al. arXiv:1808.04728)

CosmoGAN: (Kurth et al. Concurrency Computat Pract Exper. 2018;e4989)
And exploring supercomputer-scale distributed deep learning interactively via Jupyter notebooks
Conclusion

● Deep learning in combination with high-performance computing and productive software can accelerate science
● Various projects now highlight this potential: e.g. to discover new particles, determine fundamental parameters of the universe, simulate potential universes ....
● Requires developments in methods, applications and also computing that can benefit from collaboration between scientists and industry
Questions?
Ideas? Collaborations?
Want to help?

Wahid Bhimji
wbhimji@lbl.gov

Deep-learning@NERSC: http://www.nersc.gov/users/data-analytics/data-analytics-2/deep-learning/
Jobs@NERSC: https://lbl.referrals.selectminds.com/jobs/search/297137
Outtakes
Bijectors *Transform* Distributions

tfd = tfp.distributions
softmax_mvn = tfp.distributions.TransformedDistribution(
distribution=tfp.distributions.Normal(0., 1.),
bijector=tfp.bijectors.Chain([tfp.bijectors.SoftmaxCentered(),
                                 tfp.bijectors.Affine(
                                                 shift=[2.], scale_diag=[4.]),
                                 ]),
                                 event_shape=[1])
x = softmax_mvn.sample(int(1e3))
Gaussian Processes

posterior_samples = \
  tfp.distributions.GaussianProcessRegressionModel(
    kernel=tfp.positive_semidefinite_kernels.ExponentiatedQuadratic(),
    index_points=tf.linspace(-3., 3., 200)[..., tf.newaxis],
    observation_index_points=x,
    observations=y,
    jitter=1e-5).sample(50)
# ==> 50 posterior samples
#     conditioned on observed data.
Monte Carlo Integrals

# Example: Monte Carlo importance weighted approx integral.
d = tfp.distributions.Kumaraswamy(concentration1=0.9,
                                 concentration0=1.1)
x = d.sample(int(100e3))  # Samples are in unit interval.
z = tf.reduce_mean(((4. / (1. + x**2)) / d.prob(x)))
# ==> z is approximately 3.1416 (Easy as pie!)
# Variational posterior, actually.
q_given_x = make_posterior(x)

# Latents, conditioned on evidence.
z = q_given_x.sample(num_draws)
p_given_z = make_likelihood(z)
r = make_prior()

logq = q_given_x.log_prob(z)
logp = p_given_z.log_prob(x) + r.log_prob(z)

# Approx KL[q(Z|x), p(x,Z)].
loss = tf.reduce_mean(logq - logp)
train = tf.train.Optimizer().minimize(loss)

# Variational posterior, actually.
q_given_x = make_posterior(x)

# Latents, conditioned on evidence.
z = q_given_x.sample(num_draws)
p_given_z = make_likelihood(z)
r = make_prior()

logq = q_given_x.log_prob(z)
logp = p_given_z.log_prob(x) + r.log_prob(z)

# Approx KL[q(Z|x), p(x,Z)].
loss = tf.reduce_mean(logq - logp)
train = tf.train.Optimizer().minimize(loss)

# Variational posterior, actually.
q_given_x = make_posterior(x)

# Latents, conditioned on evidence.
z = q_given_x.sample(num_draws)
p_given_z = make_likelihood(z)
r = make_prior()

logq = q_given_x.log_prob(z)
logp = p_given_z.log_prob(x) + r.log_prob(z)

# Approx KL[q(Z|x), p(x,Z)].
loss = tf.reduce_mean(logq - logp)
train = tf.train.Optimizer().minimize(loss)

Find best surrogate posterior and maximize expected likelihood.