InferPy Documentation

Release 1.0

Javier Cózar, Rafael Cabañas, Antonio Salmerón, Andrés R. Mase

May 27, 2019

Quick Start

InferPy is a high-level API for probabilistic modeling written in Python and capable of running on top of Tensorflow. InferPy's API is strongly inspired by Keras and it has a focus on enabling flexible data processing, easy-to-code probablistic modeling, scalable inference and robust model validation.

Use InferPy if you need a probabilistic programming language that:

- Allows easy and fast prototyping of hierarchical probabilistic models with a simple and user friendly API inspired by Keras.
- Automatically creates computational efficient batched models without the need to deal with complex tensor operations.
- Run seamlessly on CPU and GPU by relying on Tensorflow, without having to learn how to use Tensorflow.

A set of examples can be found in the Probabilistic Model Zoo section.

CHAPTER 1

Getting Started:

1.1 Installation

Install InferPy from PyPI:

```
$ python -m pip install inferpy
```

1.2 30 seconds to InferPy

The core data structures of InferPy is a **probabilistic model**, defined as a set of **random variables** with a conditional dependency structure. A **random varible** is an object parameterized by a set of tensors.

Let's look at a simple non-linear **probabilistic component analysis** model (NLPCA). Graphically the model can be defined as follows,

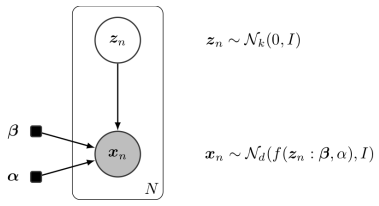


Fig. 1: Non-linear PCA

We start by importing the required packages and defining the constant parameters in the model.

```
import inferpy as inf
import tensorflow as tf

# number of components
k = 1
```

(continues on next page)

(continued from previous page)

```
# size of the hidden layer in the NN
d0 = 100
# dimensionality of the data
dx = 2
# number of observations (dataset size)
N = 1000
```

A model can be defined by decorating any function with `@inf.probmodel`. The model is fully specified by the variables defined inside this function:

```
@inf.probmodel
def nlpca(k, d0, dx, decoder):

    with inf.datamodel():
        z = inf.Normal(tf.ones([k])*0.5, 1., name="z")     # shape = [N,k]
        output = decoder(z,d0,dx)
        x_loc = output[:,:dx]
        x_scale = tf.nn.softmax(output[:,dx:])
        x = inf.Normal(x_loc, x_scale, name="x")    # shape = [N,d]
```

The construct `with inf.datamodel()`, which resembles to the **plateau notation**, will replicate N times the variables enclosed, where N is the size of our data.

In the previous model, the input argument `decoder` must be a function implementing a neural network. This might be defined outside the model as follows.

```
def decoder(z,d0,dx):
    h0 = tf.layers.dense(z, d0, tf.nn.relu)
    return tf.layers.dense(h0, 2 * dx)
```

Now, we can instantiate our model and obtain samples (from the prior distributions).

```
# create an instance of the model
m = nlpca(k,d0,dx, decoder)

# Sample from priors
samples = m.sample()
```

In variational inference, we must defined a Q-model as follows.

```
@inf.probmodel
def qmodel(k):
    with inf.datamodel():
        qz_loc = inf.Parameter(tf.ones([k])*0.5, name="qz_loc")
        qz_scale = tf.math.softplus(inf.Parameter(tf.ones([k]),name="qz_scale"))

        qz = inf.Normal(qz_loc, qz_scale, name="z")
```

Afterwards, we define the parameters of our inference algorithm and fit the data to the model.

```
# set the inference algorithm
VI = inf.inference.VI(qmodel(k), epochs=5000)

# learn the parameters
m.fit({"x": x_train}, VI)
```

The inference method can be further configure. But, as in Keras, a core principle is to try make things reasonably simple, while allowing the user the full control if needed.

Finally, we might extract the posterior of `z`, which is basically the hidden representation of our data.

```
#extract the hidden representation
hidden_encoding = m.posterior["z"]
print(hidden_encoding.sample())
```

CHAPTER 2

Guiding Principles

2.1 Features

The main features of InferPy are listed below.

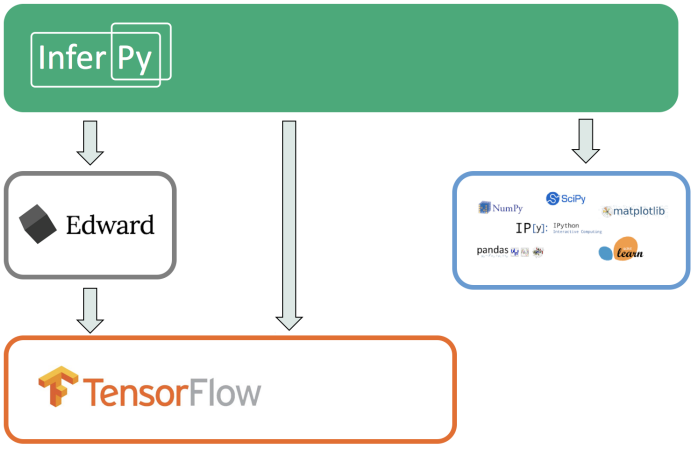
- The models that can be defined in Inferpy are those that can be defined using Edward, whose probability distribuions are mainly inherited from TensorFlow Distribuitons package.
- Edward's drawback is that for the model definition, the user has to manage complex multidimensional arrays called tensors. By contrast, in InferPy all the parameters in a model can be defined using the standard Python types (compatibility with Numpy is available as well).
- InferPy directly relies on top of Edward's inference engine and includes all the inference algorithms included in this package. As Edward's inference engine relies on TensorFlow computing engine, InferPy also relies on it too.
- InferPy seamlessly process data contained in a numpy array, Tensorflow's tensor, Tensorflow's Dataset (tf.Data API), Pandas' DataFrame or Apache Spark's DataFrame.
- InferPy also includes novel distributed statistical inference algorithms by combining Tensorflow computing engines.

2.2 Architecture

Given the previous considerations, we might summarize the InferPy architecture as follows.

Note that InferPy can be seen as an upper layer for working with probabilistic distributions defined over tensors. Most of the interaction is done with Edward: the definitions of the distributions, the inference. However, InferPy also interacts directly with Tensorflow in some operations that are hidden to the user, e.g. the manipulation of the tensors representing the parameters of the distributions.

An additional advantage of using Edward and Tensorflow as inference engine, is that all the paralelisation details are hidden to the user. Moreover, the same code will run either in CPUs or GPUs.



For some less important task, InferPy might also interact with other third-party software. For example, reading data is done with Pandas or the visualization tasks are leveraged to MatPlotLib.

CHAPTER 3

Requirements

3.1 System

Currently, InferPy requires Python 3.4 or higher. For checking your default Python version, type:

```
$ python --version
```

Travis tests are performed on versions 3.5 and 3.6. Go to https://www.python.org/ for specific instructions for installing the Python interpreter in your system.

InferPy runs in any OS with the Python interpreter installed. In particular, tests have been carried out for the systems listed bellow.

- Linux CentOS 7
- Linux Elementary 0.4
- Linux Mint 19
- Linux Ubuntu 14.04 16.04 18.04
- MacOS High Sierra (10.13) and Mojave (10.14)
- Windows 10 Enterprise

3.2 Package Dependencies

For a basic usage, InferPy requires the following packages:

```
tensorflow>=1.12.1,<2.0
tensorflow-probability>=0.5.0,<1.0
networkx>=2.2.0<3.0
matplotlib>=2.2.3,<3.0
Keras==2.2.4
```

(continues on next page)

(continued from previous page)

```
Keras-Applications==1.0.7
Keras-Preprocessing==1.0.9
```

CHAPTER 4

Guide to Building Probabilistic Models

4.1 Getting Started with Probabilistic Models

InferPy focuses on *hirearchical probabilistic models* structured in two different layers:

- A **prior model** defining a joint distribution $p(\mathbf{w})$ over the global parameters of the model. $\mathbf{w}$ can be a single random variable or a bunch of random variables with any given dependency structure.
- A **data or observation model** defining a joint conditional distribution $p(\mathbf{x}, \mathbf{z}|\mathbf{w})$ over the observed quantities $\mathbf{x}$ and the the local hidden variables $\mathbf{z}$ governing the observation $\mathbf{x}$. This data model is specified in a single-sample basis. There are many models of interest without local hidden variables, in that case, we simply specify the conditional $p(\mathbf{x}|\mathbf{w})$. Similarly, either $\mathbf{x}$ or $\mathbf{z}$ can be a single random variable or a bunch of random variables with any given dependency structure.

For example, a Bayesian PCA model has the following graphical structure,

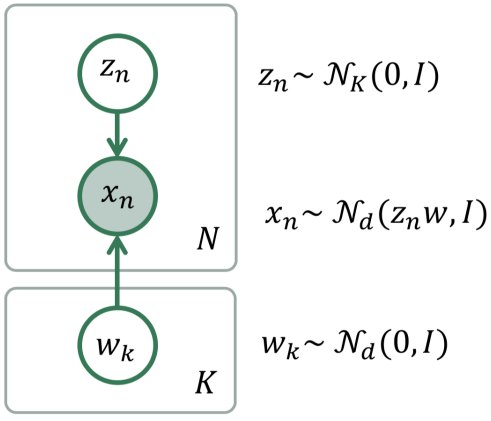


Fig. 1: Bayesian PCA

The **prior model** are the variables w_k. The **data model** is the part of the model surrounded by the box indexed by **N**.

And this is how this Bayesian PCA model is denfined in InferPy:

```
import inferpy as inf
import numpy as np

# definition of a generic model
@inf.probmodel
def pca(k,d):
    w = inf.Normal(loc=np.zeros([k,d]), scale=1, name="w")       # shape = [k,d]
    with inf.datamodel():
        z = inf.Normal(np.ones([k]),1, name="z")                 # shape = [N,k]
        x = inf.Normal(z @ w , 1, name="x")                      # shape = [N,d]


# create an instance of the model
m = pca(k=1,d=2)
```

The `with inf.datamodel()` sintaxis is used to replicate the random variables contained within this construct. It follows from the so-called *plateau notation* to define the data generation part of a probabilistic model. Every replicated variable is **conditionally idependent** given the previous random variables (if any) defined outside the **with** statement. The plateau size will be later automatically calculated, so there is not need to specify it. Yet, this construct has an optional input parameter for specifying its size, e.g., `with inf.datamodel(size=N)`. This should be consistent with the size of our data.

4.2 Random Variables

Any random variable in InferPy encapsulates an equivalent one in Edward (i.e., version 2), and hence it also has associated a distribution object from TensorFlow Probability. These can be access using the properties `var` and `dist` respectively:

```
>>> x = inf.Normal(loc = 0, scale = 1)

>>> x.var
<ed.RandomVariable 'randvar_0/' shape=() dtype=float32>

>>> x.distribution
<tfp.distributions.Normal 'randvar_0/' batch_shape=() event_shape=() dtype=float32>
```

Even more, InferPy random variables inherit all the properties and methods from Edward variables. For example:

```
>>> x.value
<tf.Tensor 'randvar_0/sample/Reshape:0' shape=() dtype=float32>

>>> x.sample()
-0.05060442
```

Following Edward's approach, we (conceptually) partition a random variable's shape into three groups:

- *Batch shape* describes independent, not identically distributed draws. Namely, we may have a set of (different) parameterizations to the same distribution.
- *Sample shape* describes independent, identically distributed draws from the distribution.
- *Event shape* describes the shape of a single draw (event space) from the distribution; it may be dependent across dimensions.

When declaring random variables, InferPy provides different ways for defining previous shapes. First, the batch shape could be obtained from the distribution parameter shapes or explicitly stated using the input parameter `batch_shape`. With this in mind, all the definitions in the following code are equivalent.

```
x = inf.Normal(0,1, batch_shape=[3,2])                    # x.shape = [3,2]

x = inf.Normal(loc = [[0.,0.],[0.,0.],[0.,0.]], scale=1)  # x.shape = [3,2]

x = inf.Normal(loc = np.zeros([3,2]), scale=1)            # x.shape = [3,2]

x = inf.Normal(loc = 0, scale=tf.ones([3,2]))             # x.shape = [3,2]
```

The `with inf.datamodel(size = N)` sintaxis is used to specify the sample shape. Alternatively, we might explicitly state it using the input paramenter `sample_shape`. This is actually inherit from Edward.

```
x = inf.Normal(tf.ones([3,2]), 0, sample_shape=100)      # x.sample = [100,3,2]

with inf.datamodel(100):
    x = inf.Normal(tf.ones([3, 2]), 0)                   # x.sample = [100,3,2]
```

Finally, the sample shape will only be consider in some distributions. This is the case of the multivariate Gaussian:

```
x = inf.MultivariateNormalDiag(loc=[1., -1], scale_diag=[1, 2.])
```

```
>>> x.event_shape
TensorShape([Dimension(2)])

>>> x.batch_shape
TensorShape([])

>>> x.sample_shape
TensorShape([])
```

Note that indexing is supported:

```
with inf.datamodel(size=10):
    x = inf.models.Normal(loc=0., scale=1., batch_shape=[5])       # x.shape = [10,5]


y = x[7,4]                                                  # y.shape = []

y2 = x[7]                                                   # y2.shape = [5]

y3 = x[7,:]                                                 # y2.shape = [5]

y4 = x[:,4]                                                 # y4.shape = [10]
```

Moreover, we may use indexation for defining new variables whose indexes may be other (discrete) variables.

4.3 Probabilistic Models

A **probabilistic model** defines a joint distribution over observable and hidden variables, $p(\mathbf{w}, \mathbf{z}, \mathbf{x})$. Note that a variable might be observable or hidden depending on the fitted data. Thus this is not specified when defining the model.

A probabilistic model is defined by decorating any function with `@inf.probmodel`. The model is made of any variable defined inside this function. A simple example is shown below.

```
@inf.probmodel
def simple(mu=0):
    # global variables
    theta = inf.Normal(mu, 0.1, name="theta")

    # local variables
    with inf.datamodel():
        x = inf.Normal(theta, 1, name="x")
```

Note that any variable in a model must be initialized with a name (this is not required when defining random variables outside the probmodel scope).

The model must be **instantiated** before it can be used. This is done by simple invoking the function (which will return a probmodel object).

```
>>> m = simple()
>>> type(m)
<class 'inferpy.models.prob_model.ProbModel'>
```

Now we can use the model with the prior probabilities. For example, we might get a sample:

```
>>> m.sample()
{'theta': -0.074800275, 'x': array([0.07758344], dtype=float32)}
```

or extract the variables:

```
>>> m.vars["theta"]
<inf.RandomVariable (Normal distribution) named theta/, shape=(), dtype=float32>
```

We can create new and different instances of our model:

```
>>> m2 = simple(mu=5)
>>> m==m2
False
```

4.4 Supported Probability Distributions

Supported probability distributions are located in the package `inferpy.models`. All of them have `inferpy.models.RandomVariable` as superclass. A list with all the supported distributions can be obtained as as follows.

```
>>> inf.models.random_variable.distributions_all
['Autoregressive', 'BatchReshape', 'Bernoulli', 'Beta', 'BetaWithSoftplusConcentration
↪',
 'Binomial', 'Categorical', 'Cauchy', 'Chi2', 'Chi2WithAbsDf',
↪'ConditionalTransformedDistribution',
  'Deterministic', 'Dirichlet', 'DirichletMultinomial', 'ExpRelaxedOneHotCategorical',
↪ '
  Exponential', 'ExponentialWithSoftplusRate', 'Gamma', 'GammaGamma',
  'GammaWithSoftplusConcentrationRate', 'Geometric', 'GaussianProcess',
  'GaussianProcessRegressionModel', 'Gumbel', 'HalfCauchy', 'HalfNormal',
  'HiddenMarkovModel', 'Horseshoe', 'Independent', 'InverseGamma',
   'InverseGammaWithSoftplusConcentrationRate', 'InverseGaussian', 'Kumaraswamy',
```

(continues on next page)

(continued from previous page)

```
   'LinearGaussianStateSpaceModel', 'Laplace', 'LaplaceWithSoftplusScale', 'LKJ',
  'Logistic', 'LogNormal', 'Mixture', 'MixtureSameFamily', 'Multinomial',
   'MultivariateNormalDiag', 'MultivariateNormalFullCovariance',
↪'MultivariateNormalLinearOperator',
   'MultivariateNormalTriL', 'MultivariateNormalDiagPlusLowRank',
↪'MultivariateNormalDiagWithSoftplusScale',
   'MultivariateStudentTLinearOperator', 'NegativeBinomial', 'Normal',
↪'NormalWithSoftplusScale',
   'OneHotCategorical', 'Pareto', 'Poisson', 'PoissonLogNormalQuadratureCompound',
↪'QuantizedDistribution',
   'RelaxedBernoulli', 'RelaxedOneHotCategorical', 'SinhArcsinh', 'StudentT',
↪'StudentTWithAbsDfSoftplusScale',
   'StudentTProcess', 'TransformedDistribution', 'Triangular', 'TruncatedNormal',
↪'Uniform', 'VectorDeterministic',
   'VectorDiffeomixture', 'VectorExponentialDiag', 'VectorLaplaceDiag',
↪'VectorSinhArcsinhDiag', 'VonMises',
   'VonMisesFisher', 'Wishart', 'Zipf']
```

Note that these are all the distributions in Edward 2 and hence in TensorFlow Probability. Their input parameters will be the same.

CHAPTER 5

Guide to Approximate Inference

5.1 Variational Inference

The API defines the set of algorithms and methods used to perform inference in a probabilistic model $p(x, z, \theta)$ (where x are the observations, z the local hidden variables, and θ the global parameters of the model). More precisely, the inference problem reduces to compute the posterior probability over the latent variables given a data sample $p(z, \theta|x_{train})$, because by looking at these posteriors we can uncover the hidden structure in the data. Let us consider the following model:

```
@inf.probmodel
def pca(k,d):
    w = inf.Normal(loc=np.zeros([k,d]), scale=1, name="w")      # shape = [k,d]
    with inf.datamodel():
        z = inf.Normal(np.ones([k]),1, name="z")                # shape = [N,k]
        x = inf.Normal(z @ w , 1, name="x")                     # shape = [N,d]
```

In this model, the posterior over the local hidden variables $p(w_n|x_{train})$ tell us the latent vector representation of the sample x_n, while the posterior over the global variables $p(\mu|x_{train})$ tells us which is the affine transformation between the latent space and the observable space.

InferPy inherits Edward's approach an consider approximate inference solutions,

$$q(z, \theta) \approx p(z, \theta|x_{train})$$

in which the task is to approximate the posterior $p(z, \theta|x_{train})$ using a family of distributions, $q(z, \theta; \lambda)$, indexed by a parameter vector λ.

For making inference, we must define a model 'Q' for approximating the posterior distribution. This is also done by defining a function decorated with `@inf.probmodel`:

```
@inf.probmodel
def qmodel(k,d):
    qw_loc = inf.Parameter(tf.ones([k,d]), name="qw_loc")
    qw_scale = tf.math.softplus(inf.Parameter(tf.ones([k, d]), name="qw_scale"))
```

(continues on next page)

(continued from previous page)

```
    qw = inf.Normal(qw_loc, qw_scale, name="w")

    with inf.datamodel():
        qz_loc = inf.Parameter(tf.ones([k]), name="qz_loc")
        qz_scale = tf.math.softplus(inf.Parameter(tf.ones([k]), name="qz_scale"))
        qz = inf.Normal(qz_loc, qz_scale, name="z")
```

In the 'Q' model we should include a q distribution for every non observed variable in the 'P' model. These varaiables are also objects of class `inferpy.RandomVariable`. However, their parameters might be of type `inf.Parameter`, which are objects encapsulating TensorFlow trainable variables.

Finally, when defining the inference algorithm, we must specify an instance of the 'Q' model:

```
# set the inference algorithm
VI = inf.inference.VI(qmodel(k=1,d=2), epochs=1000)
```

Then we must instantiate our 'P' model and fit the data with the inference algorithm defined.

```
# create an instance of the model
m = pca(k=1,d=2)
# run the inference
m.fit({"x": x_train}, VI)
```

The output generated will be similar to:

```
 0 epochs         44601.14453125.....................
 200 epochs         44196.98046875.....................
 400 epochs         50616.359375.....................
 600 epochs         41085.6484375.....................
 800 epochs         30349.79296875.....................
```

Finally we can access to the dictionary with the posterior distributions:

```
>>> m.posterior['w']
<inf.RandomVariable (Normal distribution) named w_2/, shape=(1, 2), dtype=float32>
```

5.2 Custom Loss function

Following InferPy guiding principles, users can further configure the inference algorithm. For example, we might be interested in defining our own function to minimise. As an example, we define the following function taking as input parameters the instances of the P and Q models, and the dictionary with the observations. Note that the output of this function must be a tensor.

```
from tensorflow_probability import edward2 as ed

# define custom elbo function
def custom_elbo(pmodel, qmodel, sample_dict):
    # create combined model
    plate_size = pmodel._get_plate_size(sample_dict)

    # expand the qmodel (just in case the q model uses data from sample_dict, use␣
↪interceptor too)
    with ed.interception(inf.util.interceptor.set_values(**sample_dict)):
        qvars, _ = qmodel.expand_model(plate_size)
```

(continues on next page)

(continued from previous page)

```
    # expand de pmodel, using the intercept.set_values function, to include the␣
↪sample_dict and the expanded qvars
    with ed.interception(inf.util.interceptor.set_values(**{**qvars, **sample_dict})):
        pvars, _ = pmodel.expand_model(plate_size)

    # compute energy
    energy = tf.reduce_sum([tf.reduce_sum(p.log_prob(p.value)) for p in pvars.
↪values()])

    # compute entropy
    entropy = - tf.reduce_sum([tf.reduce_sum(q.log_prob(q.value)) for q in qvars.
↪values()])

    # compute ELBO
    ELBO = energy + entropy

    # This function will be minimized. Return minus ELBO
    return -ELBO
```

For using our own loss function, we simply have to pass this function to the input parameter `loss` in the inference method constructor. For example:

```
VI = inf.inference.VI(qmodel(k=1,d=2), loss=custom_elbo, epochs=1000)

# run the inference
m.fit({"x": x_train}, VI)
```

After this, the rest of the code remains unchanged.

5.3 Coding the Optimization Loop

As an InferPy model encapsulates an equivalent one in Edward, we can extract the required tensors and explicitly code the optimization loop. However, this is **not** recommended for non-expert users in TensorFlow.

First, we get the tensor for the ELBO, but we must first invoke the method `inf.util.runtime.set_tf_run(False)` which avoids the evaluation of such tensor.

```
# for not evaluating ELBO tensor
inf.util.runtime.set_tf_run(False)
# extract the computational graph of the ELBO
loss_tensor = inf.inference.loss_functions.elbo.ELBO(m,q, {"x": x_train})
```

Then we must initialize the optimizer and the session:

```
# build an optimizer to minimize the ELBO
optimizer = tf.train.AdamOptimizer(learning_rate=0.1)
train = optimizer.minimize(loss_tensor)

# start a session
sess = tf.Session()
# intialize the TF variables
sess.run(tf.global_variables_initializer())
```

Afterwards, we code the loop itself, where the tensor `train` must be evaluated at each iteration for performing each optimization step.

```
for i in range(0,100):
    sess.run(train)
    t += [sess.run(loss_tensor)]
    print(t[-1])
```

After the optimization, we can extract the posterior distributions:

```
# extract the posterior distributions
posterior_qvars = {name: qv.build_in_session(sess) for name, qv in q._last_expanded_
↪vars.items()}
```

CHAPTER 6

Probabilistic Model Zoo

In this section, we present the code for implementing some models in Inferpy.

6.1 Bayesian Linear Regression

Graphically, a (Bayesian) linear regression can be defined as follows,

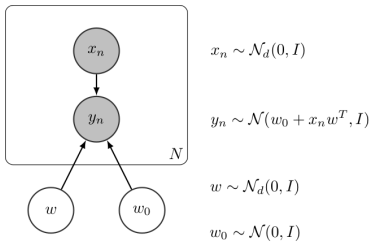


Fig. 1: Bayesian Linear Regression

The InferPy code for this model is shown below,

```
# required pacakges
import inferpy as inf
import tensorflow as tf


@inf.probmodel
def linear_reg(d):
    w0 = inf.Normal(0, 1, name="w0")
    w = inf.Normal(0, 1, batch_shape=[d,1], name="w")

    with inf.datamodel():
        x = inf.Normal(5, 2, batch_shape=d, name="x")
```

(continues on next page)

(continued from previous page)

```
        y = inf.Normal(w0 + x @ w, 1.0, name="y")


@inf.probmodel
def qmodel(d):
    qw0_loc = inf.Parameter(0., name="qw0_loc")
    qw0_scale = tf.math.softplus(inf.Parameter(1., name="qw0_scale"))
    qw0 = inf.Normal(qw0_loc, qw0_scale, name="w0")

    qw_loc = inf.Parameter(tf.zeros([d,1]), name="qw_loc")
    qw_scale = tf.math.softplus(inf.Parameter(tf.ones([d,1]), name="qw_scale"))
    qw = inf.Normal(qw_loc, qw_scale, name="w")


# create an instance of the model
m = linear_reg(d=2)

### create toy data
N = 1000
data = m.sample(size = N, data={"w0":0, "w":[[2],[1]]})
x_train = data["x"]
y_train = data["y"]

VI = inf.inference.VI(qmodel(2), epochs=10000)
m.fit({"x": x_train, "y":y_train}, VI)

sess = inf.get_session()
print(sess.run(m.posterior["w"].loc))

with tf.Session() as sess2:
    print(sess2.run(m.posterior["w"].copy().loc))
```

6.2 Bayesian Logistic Regression

Graphically, a (Bayesian) logistic regression can be defined as follows,

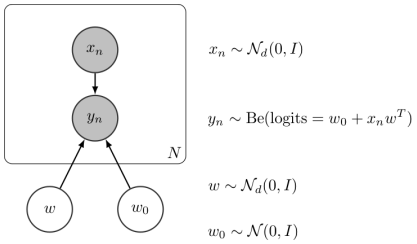


Fig. 2: Bayesian Linear Regression

The InferPy code for this model is shown below,

```
# required pacakges
import inferpy as inf
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import tensorflow as tf


@inf.probmodel
def log_reg(d):
    w0 = inf.Normal(0., 1, name="w0")
    w = inf.Normal(0., 1, batch_shape=[d,1], name="w")

    with inf.datamodel():
        x = inf.Normal(0., 2., batch_shape=d, name="x")
        y = inf.Bernoulli(logits = w0 + x @ w, name="y")




@inf.probmodel
def qmodel(d):
    qw0_loc = inf.Parameter(0., name="qw0_loc")
    qw0_scale = tf.math.softplus(inf.Parameter(1., name="qw0_scale"))
    qw0 = inf.Normal(qw0_loc, qw0_scale, name="w0")

    qw_loc = inf.Parameter(tf.zeros([d,1]), name="qw_loc")
    qw_scale = tf.math.softplus(inf.Parameter(tf.ones([d,1]), name="qw_scale"))
    qw = inf.Normal(qw_loc, qw_scale, name="w")


# create an instance of the model
m = log_reg(d=2)

### create toy data
N = 1000
data = m.sample(size = N, data={"w0":0, "w":[[2],[1]]})
x_train = data["x"]
y_train = data["y"]



VI = inf.inference.VI(qmodel(2), epochs=10000)
m.fit({"x": x_train, "y":y_train}, VI)

sess = inf.get_session()
print(m.posterior["w"].sample())
print(sess.run(m.posterior["w"].loc))
```

6.3 Linear Factor Model (PCA)

A linear factor model allows to perform principal component analysis (PCA). Graphically, it can be defined as follows,

The InferPy code for this model is shown below,

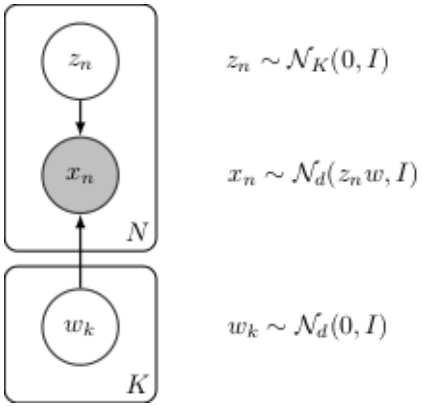


Fig. 3: Linear Factor Model (PCA)

```
# required pacakges
import inferpy as inf
import numpy as np
import tensorflow as tf

# definition of a generic model
@inf.probmodel
def pca(k,d):
    beta = inf.Normal(loc=np.zeros([k,d]),
                      scale=1, name="beta")                # shape = [k,d]

    with inf.datamodel():
        z = inf.Normal(tf.ones([k]),1, name="z")           # shape = [N,k]
        x = inf.Normal(z @ beta , 1, name="x")             # shape = [N,d]


# create an instance of the model
m = pca(k=1,d=2)



@inf.probmodel
def qmodel(k,d):
    qbeta_loc = inf.Parameter(tf.zeros([k,d]), name="qbeta_loc")
    qbeta_scale = tf.math.softplus(inf.Parameter(tf.ones([k,d]),
```

6.4 Non-linear Factor Model (NLPCA)

Similarly to the previous model, the Non-linear PCA can be graphically defined as follows,

Its code in InferPy is shown below,

```
# required pacakges
import inferpy as inf
import numpy as np
import tensorflow as tf
```

(continues on next page)

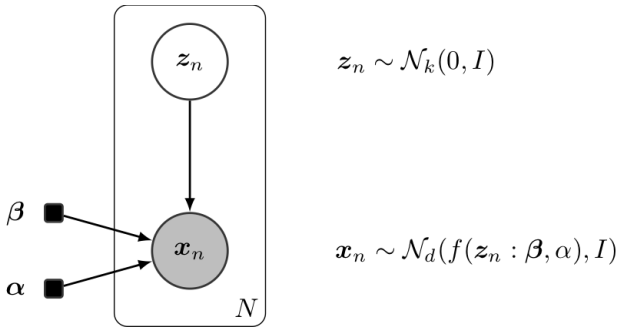


Fig. 4: Non-linear PCA

(continued from previous page)

```
# definition of a generic model


# number of components
k = 1
# size of the hidden layer in the NN
d0 = 100
# dimensionality of the data
dx = 2
# number of observations (dataset size)
N = 1000


@inf.probmodel
def nlpca(k, d0, dx, decoder):

    with inf.datamodel():
        z = inf.Normal(tf.ones([k])*0.5, 1., name="z")     # shape = [N,k]
        output = decoder(z,d0,dx)
        x_loc = output[:,:dx]
        x_scale = tf.nn.softmax(output[:,dx:])
        x = inf.Normal(x_loc, x_scale, name="x")    # shape = [N,d]


def decoder(z,d0,dx):
    h0 = tf.layers.dense(z, d0, tf.nn.relu)
    return tf.layers.dense(h0, 2 * dx)


# Q-model  approximating P

@inf.probmodel
def qmodel(k):
    with inf.datamodel():
        qz_loc = inf.Parameter(tf.ones([k])*0.5, name="qz_loc")
        qz_scale = tf.math.softplus(inf.Parameter(tf.ones([k]),name="qz_scale"))

        qz = inf.Normal(qz_loc, qz_scale, name="z")


# create an instance of the model
```

(continues on next page)

(continued from previous page)

```
m = nlpca(k,d0,dx, decoder)

# set the inference algorithm
VI = inf.inference.VI(qmodel(k), epochs=5000)

# learn the parameters
m.fit({"x": x_train}, VI)


#extract the hidden representation
hidden_encoding = m.posterior["z"]
print(hidden_encoding.sample())

sess = inf.get_session()
print(sess.run(hidden_encoding.loc))
```

CHAPTER 7

Contact and Support

If you have any question about the toolbox or if you want to collaborate in the project, please do not hesitate to contact us. You can do it through the following email address: inferpy.api@gmail.com

For more technical questions, please use Github issues.